Extending XData to Kill SQL Query Mutants in the Wild

Bikash Chandra, Bhupesh Chawda, Shetal Shah, S. Sudarshan, Ankit Shah
Computer Science and Engg. Dept.
Indian Institute of Technology Bombay, India
{bikash2911,bhupeshchawda}@gmail.com {shetals,sudarsha,ankitshah10}@cse.iitb.ac.in

ABSTRACT

SQL queries are usually tested for correctness by executing them on one or more datasets, to see if they give the desired results on each dataset. Erroneous queries are often the result of small changes, or mutations, of the correct query. Earlier work on the XData system showed how to generate datasets that kill all mutations in a class of mutations that included join type and comparison operation mutations. However, the system could not handle a number of commonly used SQL features.

In this paper we extend the XData data generation techniques to handle features such as null values, string constraints, aggregation with constraints on aggregation results, and a class of subqueries, amongst others. We present a study of the effectiveness of our data generation approach for correcting student SQL assignments that were part of a database course. The datasets generated by XData outperform publicly available datasets, as well as manual grading done earlier by teaching assistants.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Algorithms, Experimentation, Verification

1. INTRODUCTION

Queries written in SQL are widely used, but are not amenable to the usual approaches of program verification, since the queries themselves act as formal specifications, and there is no separate implementation to compare against. Queries are instead tested against one or more datasets to check that they give the desired results. Generation of datasets that can catch commonly occurring errors is thus key to thorough testing of SQL queries. A closely related problem is that of grading SQL assignments, where students can write queries in a variety of different ways, making it hard for graders to check if a given query is correct. Grading of SQL queries is usually done by testing the query against one or more manually created datasets, but databases created in ad hoc ways often miss a variety of errors.[1] Grading by just reading queries is even more error prone, since graders often miss subtle mistakes. As an example, when required to write the query Q below, students may write query Q_s, which is similar enough that a grader can easily miss the difference.

Q:
```
SELECT course.id, course.title
FROM course LEFT OUTER JOIN
  (SELECT * from dept WHERE dept.budget > 70000) D
   USING (dept_name);
```
Q_s:
```
SELECT course.id, course.title
FROM course LEFT OUTER JOIN dept USING (dept_name)
WHERE dept.budget > 70000;
```

These queries are not equivalent, since they give different results on departments with budget less than 70000.

Most incorrect queries are small syntactic deviations of the correct one and can be thought of as mutants of the correct query/program. Specifically, a *mutation* is single syntactic correct change of the original query; and a *mutant* is the result of one of more mutations on the original query. A dataset *kills* a mutant if the original query and the mutant give different results on the dataset. A test suite consisting of multiple datasets kills a mutant if at least one of the datasets kills the mutant.

Data generation can be used in two distinct ways: (a) to check if a given query is correct, a tester manually examines the result of the query on each dataset, and checks if the result is what was intended, and (b) to check if a given query is not equivalent to a given correct query, the results of the given and correct query are compared on each dataset. The first mode is appropriate for testing database application queries, while the second is appropriate for grading student SQL queries. Datasets generated could also potentially aid in the testing of query optimizers wherein optimizer errors are viewed as mutations.

In our earlier work on the XData system [9] we described techniques for data generation targeted at killing of mutants, which handled several SQL constructs. The techniques described in [9] ensured all single mutations in a specific space of mutations would be killed. The space included mutations to the join type (between inner join, left and right outer join) *across all equivalent join orders* of a given query, mutations

[1]The Gradiance system provides support for grading SQL queries by running them against test databases, but does not provide support for automatic generation of test databases.

of comparison operations (between $<, <=, >, >=$ and $<>$), and mutations of aggregation operations (between min and max, between avg and avg distinct, and between sum and sum distinct) as long as there are no constraints on the aggregation results.

However, real life SQL queries have a variety of features that were not handled in [9]. When we used our code on SQL queries that were part of assignments given in a database course, we found a number of limitations. These limitations included handling strings constraints, handling aggregate queries with constraints on the aggregate results, and handling nested subqueries.

In this paper, we describe how we generate test data to handle such features. For each new feature, for example strings, we address the following issues, for a query that uses the feature: (a) how to generate data to handle mutations to other features, ignoring mutations to the new feature, and (b) how to detect mutations involving the new feature.

The contributions of this paper include novel data generation techniques for the following. a) Handling of NULL values (Section 3). b) Handling of string constraints, including case sensitive and insensitive comparisons and LIKE constraints, and upper() and lower() functions (Section 4). c) Handling of aggregation with constraints on the aggregate result (Section 5). d) Handling of subqueries (Section 6). e) Handling of set operations, date and time datatypes, views (permanent created using CREATE VIEW, and temporary created using the WITH clause), and insert/delete/update queries (Section 7). Although some of our data generation techniques are not complete, i.e., they may fail to generate some datasets on some complex queries, our experimental results show that they work very well in practice.

Our experimental study (Section 8), using as a benchmark a set of assignments given as part of a database course at IIT Bombay, show that the extensions described in this paper were critical for handling many queries, and the datasets we generated overall outperformed publicly available datasets, and manual grading by TAs.

2. RELATED WORK AND BACKGROUND

In our earlier work called XData [9], we presented techniques for generating test data for killing SQL query mutants. In SQL, a join query can be specified in an order independent fashion resulting in many equivalent join orders for a given query. Hence, the number of join type mutations across all these orders is exponential. XData generates a *linear* number of datasets to detect an error (kill mutations) across different join orderings. For the space of join type, selection condition, and a particular subset of unconstrained aggregation mutations (details in [9]), the datasets generated are complete, i.e., the datasets would kill *all* single mutations of the query.

C. de la Riva et al. [4] show how to generate test cases to kill SQL query mutants, generating constraints based on SQL coverage rules ([11]) and solving them using a constraint solver called Alloy. They consider queries with joins and where clause conditions. Unlike us, they do consider disjunctions. However, they do not consider aggregates, subqueries, strings and updates.

Qex [13] is a tool for generating input tables and parameter values for a given parameterized SQL query using the SMT solver Z3. The goal of Qex is to generate data so that the query has a non-empty result. Data generation for killing mutants is not considered. Olston et al. [7] 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 check for query correctness.

Emmi et al. [5] describe an approach to test applications based on the creation of database states and test inputs, which ensures path coverage. However, they do not address the problem of testing of queries within the application, nor do they address mutations.

In the rest of this section, we give some background of XData [9]. Given a query Q, the XData system [9] generates multiple datasets, each of which kills one or more mutations of the query; i.e., on each dataset the given query returns a result that is different from those returned by each of the mutations targeted by that dataset. The number of possible mutations is very large, but the number of datasets generated is small. To generate a particular dataset, XData generates a set of constraints, where each tuple in the target dataset is represented by a tuple of constraint variables. It then invokes a constraint (SMT) solver, CVC3 [1], to solve the constraints; the solution defines a dataset on which the query is to be tested.

The first dataset ensures a non-empty result for Q (which itself kills several mutations that would give an empty result on that dataset). The remaining datasets target specific mutations of Q. Mutations targeted by the techniques in [9] include join type mutations, selection mutations, and aggregation operation mutations with no constraints on the aggregation.

To illustrate what constraints are generated for each dataset, we use the following query Q:

```
SELECT *
FROM course INNER JOIN dept USING (dept_name)
```

The first dataset is designed to generate a non-empty result for Q. Hence we specify that the `course` and `dept` relations each contain 1 tuple, which match on the join attribute. In case more than one tuple is generated for a relation, primary key constraints are enforced by appropriate CVC3 constraints. Foreign key constraints are enforced by generating additional tuples, as required, for the referenced relations. In our example, the join condition between `dept` and `course` is on a foreign key from `course` referencing `dept`, so a tuple generated for `dept` may already ensure the foreign key constraint is satisfied, but in general, extra tuples need to be generated for `dept`, for each tuple in `course`.

As explained in [9], to kill a mutation of the inner join to right outer join, we need a value in `dept.dept_name` which does not match any value in `course.dept_name`, which we ensure by creating an extra tuple for `dept`, with CVC3 constraints ensuring it has no matching tuples in `course`.

In this paper, we consider single block SQL queries with join/outer-join operations and predicates in the where-clause, and optionally aggregate operations, corresponding to select/project/join/outer-join queries in relational algebra, with an optional aggregation operation on top. We also consider insert/delete/update queries. We allow from and where clause subqueries, and scalar sub queries. While the sub query can have unconstrained aggregation operations, we restrict constrained aggregation to the outer query. We do not handle subqueries within subqueries. We also make the same assumptions as in [9], for example that the only constraints are primary and foreign key constraints, that join predicates and selection are conjunctions, etc.

3. HANDLING NULLS

In [9], we did not handle NULLs, since our solver CVC3 does not understand NULL values. We now sketch how we extend the constraints to handle NULL values.

To model NULLs for text attributes, we enumerate a few more values in the enumerated type and designate them NULLs, for example, for an attribute `course_id`, we enumerate values `NULL_course_id_1`, `NULL_course_id_2`, etc.

For numeric values, we model NULLs as any integer in a range of negative values that we define to be not part of the allowable domain of that numeric value.

For each dataset, we choose which attributes of which tuples are to be made null, and add constraints forcing those attribute values to take on one of the abovementioned special values representing NULL. In addition we add constraints to force all other values to be non null. We also need to enforce the fact that nulls are not comparable. We do so by choosing different `NULL_` values for different constraint variables that may potentially be assigned a null value, thus implicitly enforcing an inequality between them.

The capability to generate NULLs enables us to handle nullable foreign keys, mutation of count to count(*) and explicit IS NULL checks. If a foreign key attribute fk, is nullable then the corresponding foreign key constraint to CVC3 is that fk is a subset of the corresponding primary key values or NULL values, allowing CVC3 to assign NULLs to foreign keys if required; thereby allowing to kill more mutants than is possible with non nullable foreign keys. (Our implementation handles multi-attribute foreign and primary keys.) If the query contains a condition $r.a$ IS NULL, we explicitly assign (a different) NULL to attribute a for each tuple $r[i]$. This dataset also kills the mutation of IS NULL to NOT IS NULL. If the query contains an IS NULL then the dataset will give a non-empty result whereas the NOT IS NULL mutant will generate an empty result and vice versa.

More details of how we handle nulls are given in [3].

4. STRING CONSTRAINTS

SQL queries often have equality, inequality constraints on strings, and pattern matching constraints using the LIKE operator. Our techniques handle the following class of string constraints: S1 *likeop* pattern, S1 *relop* constant, *strlen*(S) *relop* constant, and S1 *relop* S2, where S1 and S2 are string variables, *likeop* is one of *LIKE, ILIKE* (case insensitive like), *NOT LIKE* and *NOT ILIKE* and *relop* operators are $=, <, \leq, >, \geq, <>$, and case-insensitive equality denoted by $\sim=$.

4.1 The String Solver

As with NULLs, CVC3 does not support string operations, and hence we need to solve the string constraints outside of CVC3. To generate data for a query containing string and non-string constraints, we first solve string constraints, get an assignment of values for each string variable and then solve the non-string constraints using CVC3, to get an overall solution. This two-step solution works as long as there are no constraints that involve both string and non-string variables; an example of such a constraint is one that equates the length of a string variable to an integer variable. The class of constraints we support does not allow such constraints between string and non-string variables.

There are several string solvers available, including Hampi [6], Kaluza [8] and Rex [12]. However we found that Hampi and Kaluza are rather slow, and while they handle regular expressions and length constraints, they cannot handle constraints such as $S1 < S2$ which are commonly used. Rex, though much faster, cannot handle any constraints involving multiple string variables. Hence we built our own solver which we describe briefly below:

We first collect the string constraints, namely, selection conditions on strings, like conditions, and string length constraints. We then reduce the number of constraints by removing equality constraints by propagating the constants the string variable is equated to, replacing the occurrence of the string variable by that of the constant. This may result in constraints of the form $const_i$ *relop* $const_j$. If this constraint is unsatisfiable, then there is no possible solution to given set of constraints.

Next, we group variables that depend on each other, i.e., if V_i *relop* V_j then V_i and V_j are in the same group. Then for each group we construct a graph, where the variables form the vertices. A constraint of the form $V_i < V_j$ or $V_i \leq V_j$ is represented by a directed edge from V_j to V_i in the graph. We then traverse the graph and collect all variables/vertices, $V_i, \ldots, V_k$ whose outdegree is 0; i.e., that there is no variable whose value is less than $V_i, \ldots, V_k$ and for each such V_i, find the lexicographically smallest string that satisfies all the constraints for V_i. We repeat this step till we have assigned a value to all variables. Note that if no such V_i exists, i.e., there is a cycle in the graph, then it implies that all variables in the cycle are equal, if all edges are $\leq$, or that the given set of constraints is not satisfiable, if one of the edges is $<$.

Handling $<>$ and $\sim=$ comparisons: We handle notEqual conditions of the kind $V_i <>$ *constant* and $V_i <> V_j$ such that V_j is unconstrained, i.e., there are no other string constraints constraining the value of V_j. For the constraint $V_i <> const_i$, we simply ensure that the value assigned to V_i is not $const_i$. For constraints of the form $V_i <> V_j$, we heuristically first find an assignment to the constrained variable, V_i, and then assign a different value of V_j. Similarly, for $V_i \sim= V_j$, we find an assignment for V_i, then derive a value for V_j by changing the case of one of the letters in the string assigned to V_i.

Handling non-equality constraints where both variables are constrained is part of future work.

Handling string functions upper and lower: We do not directly support string functions *upper* and *lower*, but commonly used queries involving these functions can be rewritten using $\sim=$; for example $upper(S)$ = 'ABC' can be rewritten as $S \sim=$ 'ABC', and similarly $upper(S)$ LIKE constant can be replaced by S ILIKE constant (note that *upper*(S) = constant as well as *upper*(S) LIKE constant are replaced by false if the constant contains any lower case letters). We perform such rewriting before the string solver is called.

4.2 Killing String Constraint Mutations

We now discuss data generation to kill mutants of string conditions.

String constraint mutation: Consider a string constraint of the form $S1$ *relop* $S2$, where $S1$ is a variable (attribute name), $S2$ could be another variable or a constant.

As in [9], we consider mutations of *relop* where any occurrence of one of $\{=, <>, <, >, \leq, \geq\}$ is replaced by another. In [9], we show that 3 datasets are enough to kill all the *relop* mutations. These are the datasets generated for (1) $S1 = S2$ (2) $S1 > S2$ (3) $S1 < S2$. In addition, to kill mutations between = and $\sim=$, we generate two datasets, one where $S1 = S2$ and the other where $S1 <> S2$, but $S1 \sim= S2$.

LIKE predicate mutation: We also consider the mutation of the *likeop* operators where one of {*LIKE, ILIKE, NOT LIKE, NOT ILIKE*} is mutated to another. For a condition *S*1 *likeop pattern*, where *S*1 is an attribute name, the three datasets given below are sufficient to kill all mutations between the LIKE operators:
Dataset 1 satisfying the condition *S*1 *LIKE pattern*.
Dataset 2 satisfying condition *S*1 *ILIKE pattern*, but not *S*1 *LIKE pattern*, with *pattern* modified by changing the case of one or more characters.
Dataset 3 failing both the *LIKE* and *ILIKE* conditions, generated by replacing one or more characters (other than % and _) in *LIKE* pattern by different characters.

5. CONSTRAINED AGGREGATION

In [9], we addressed the problem of data generation for unconstrained aggregation operators. In this section, we discuss how to extend the earlier techniques to handle constraints on the aggregate result.

Aggregation constraints like $SUM(r.a) > 20$ cannot be translated into similar CVC3 constraints like $SUM(r[i].a) > 20$, leaving the number of tuples in r unspecified, since CVC3 requires us to specify how many tuples r has. For example if $r.a$ is unconstrained, even one tuple suffices, but if $r.a$ is constrained to be ≤ 5, 5 tuples are required to meet the constraint on SUM. Hence, before generating CVC3 constraints we must (a) estimate the number of tuples n, required to satisfy an aggregation constraint, and (b) translate this number n to appropriate number of tuples for each base relation so that the input of the aggregation contains exactly n tuples.

We assume that constraints on aggregation results occur at the top of a query tree. The query tree may contain joins, but we assume that there are no repeated relations in the query. Constraints on aggregate values, such as $AVG * COUNT = SUM$, result in non-linear arithmetic constraints, and since CVC3 only handles such non-linear constraints on fixed precision integers, we restrict our attention to fixed precision rationals.

We now consider how to estimate the number of values (tuples) needed to satisfy aggregation constraints on an attribute A. For each attribute, A, we collect all aggregation constraints on A, domain constraints and non-aggregate constraints on attribute A, i.e., constraints like $CREDITS < 13$. These constraints, along with constraints which capture the invariants between various aggregation operators such as $AVG * COUNT = SUM$, $MIN \leq MAX$, etc., are then given to CVC3. Since we are interested in small datasets, we want the count to be as small as possible. Hence, we run CVC3 with the count fixed to different values, ranging from 1 to *MAX_TUPLES* and choose the smallest value of the count for which CVC3 gives a valid answer.[2] We borrow the idea of calculating the number of tuples, using multiple tries, for the aggregation constraint from RQP[2]. However, note that the problem is different here, since, unlike RQP, we do not know the aggregation value in the query result.

If we have aggregation constraints on multiple attributes of the same relation, we estimate the smallest possible counts returned for each attribute which is involved in an aggregation constraint and then choose the maximum amongst these counts. The rationale behind this is that if any aggregation constraint needs n values, then that relation must contain at least n tuples.

Once we estimate the number of tuples, n, required to satisfy a given constraint, we then estimate the number of tuples to be generated for each relation, so that the input to the aggregation operator is the required number of tuples. This estimation is done with care, dictated by *unique*, *group by* constraints of the joining attributes of each relation such that either 1 or n tuples is assigned to any relation; we omit the algorithm due to lack of space. The algorithm to do so is given in [3], where we also prove that for the snowflake schema, with joins only between foreign key and primary key links, with a few other minor assumptions, a solution always exists.

The algorithm for determining the number of tuples for each relation, also determines, for each join/group by attribute, whether the attribute value is unique across tuples, or the same (duplicated) across all tuples. Note that these properties are per group.

To generate datasets that satisfy the aggregation constraints, we generate appropriate CVC3 constraints, with the number of tuples in each relation fixed based on the values computed above, and with constraints that ensure that for each join/group by attribute, the values are unique or duplicated as required.

We generate datasets to kill mutations between MIN and MAX, SUM and SUM DISTINCT, AVG and AVG DISTINCT and COUNT and COUNT(DISTINCT). In [9], we showed that in many cases, one dataset is sufficient to kill the above mutations; the dataset contains three tuples such that two tuples have distinct values on the aggregated attribute, A, and two tuples have the same value of A. In cases where A is unique the dataset contains only two tuples which distinct values for A. This technique also works with constrained aggregation (a minor change which is required in a few cases is described in [3]).

6. NESTED SUBQUERIES

We consider nested subqueries in the SQL WHERE clause, connected by IN, NOT IN, EXISTS, NOT EXISTS, ALL or ANY clauses, as well as scalar subqueries which return a single value. For simplicity we assume (a) correlation conditions in the subquery involve a single relation in the outer query, (b) IN connectives involve attributes from only one outer relation, and any correlation conditions are on the same relation, and (c) the depth of nesting is at most 1, i.e. there are no subqueries within subqueries.

To generate data which satisfies the subquery, we first generate constraints to create tuples for the outer query result, ignoring the subquery. For queries with aggregation, as we saw in Section 5, we may need to create more than one tuple for some of the relations used in the outer query. We then generate constraints to generate data for the subquery, while ensuring the connecting condition is satisfied.

Details of how to handle EXISTS, IN, NOT EXISTS,

[2]Since, we are interested in small datasets, we set *MAX_TUPLES* to 32 in our experiments.

NOT IN, ALL/ANY connectives, as well as how to handle scalar subqueries are described in [3]. The negated cases NOT IN and NOT EXISTS cause some complications, since there are potentially many ways of ensuring that a subquery result is empty or does not have a specific value in its result. For example, for an expression $r \bowtie s$, we could ensure each r tuple has no matching s tuple, or each s tuple has no matching r tuple; with multiple joins and selections, further options are possible. It is not possible to express all these options as a disjunctive condition, and hence we cannot just generate a single CVC3 constraint covering all these cases. As a heuristic, we generate a disjunction that covers several of these cases, and generate only one CVC3 constraint.

For mutations between IN and NOT IN, and between EXISTS and NOT EXISTS, the dataset generated for the original query will kill the mutation since the IN clause give an empty result when NOT IN gives a non-empty result, and vice versa, and similarly for EXISTS versus NOT EXISTS.

For conditions of the form "r.A *relop* (SQ)" where SQ is a scalar subquery, as well as conditions of "r.A *relop* [ALL/ANY] SQ", we consider mutations between the different *relop*s. Similar to the case of generating data for killing selection mutations from [9], we generate data for three cases, with *relop* replaced by $>$, $=$ and $<$.

To kill mutation of ALL versus ANY we generate two tuples for the inner query. We assert that one of the tuples satisfies the comparison operator in the ALL/ANY and the other does not. The ANY query will generate a non empty result since there is one tuple satisfying the subquery condition but the ALL query will generate an empty output.

We handle mutations of the subquery, as follows. We assume for simplicity that to generate data for the outer query, we need to generate only one tuple for each relation (which excludes some cases of constrained aggregation). Now, treating the subquery as a normal query, for each dataset generated to catch mutations, we generate constraints for the subquery, and then add constraints from the outer query.

Datasets generated thus can catch subquery mutations in many cases, but not always. For example, if the subquery connective is NOT EXISTS, and the subquery condition has a conjunction "P1 AND P2". To ensure the subquery result is empty, we enforce "NOT (P1 AND P2)", which may get enforced by just negating P1. Thus a mutation of P2 may not get detected.

7. OTHER EXTENSIONS

Set Operations: We handle the following set operations, UNION (ALL), INTERSECT (ALL), EXCEPT(ALL) and mutations between each of them. Generation of a dataset which satisfies the original query is straightforward. For example, for INTERSECT we need to generate datasets for SQ1 and SQ2 that agree on one tuple, which can be done by creating constraints to generate tuples for the two subqueries. To kill mutants among the above, we generate a dataset such that both $SQ1$ and $SQ2$ contain two tuples each, all of which are identical. (In case either of $SQ1$ or $SQ2$ cannot generate duplicate tuples due to key constraints, we just generate a single tuple for the subquery.) We also generate two more tuples for one of the subqueries, which are identical to each other but differ from the ones generated earlier (as before if integrity constraints prevent it, we just add one tuple). This dataset will kill all nonequivalent mutations between the set operators.

Handling Parameterized Queries: When generating datasets for a query with parameters, we assign a variable to every parameter. The solution given by CVC3 also contains a value for each parameter. It should be noted that since CVC3 assigns these values, every dataset may potentially have its own values for the parameters.

DATE and TIME: We handle SQL data types related to date and time, namely DATE, TIME and TIMESTAMP by converting them to integers.

Handling Insert/Delete/Update Queries: To handle INSERT queries involving a subquery, and DELETE queries, we convert them to SELECT queries by replacing "INSERT INTO relation" or "DELETE" by "SELECT *". UPDATE queries are similarly converted by creating a SELECT query whose projection list includes the primary key of the updated table, and the new values for each updated column; the WHERE clause remains unchanged from the UPDATE query. Data generation is then done to catch mutations of the resultant SELECT queries.

To test student queries with updates against a given correct update query, we perform the above transformation for both the given student queries and the given correct queries, before testing them as described in Section 8.

8. EXPERIMENTAL RESULTS

We implemented the techniques for data generation described earlier, as extensions to the XData system. We also implemented a tool for grading assignments submitted (using the Moodle course management system) by students.

We used the datasets generated by XData to grade SQL query assignments from the undergraduate database course at IIT Bombay in Fall 2011. We use 15 assignment queries for our study. The assignments also had a few more queries which we did not include because they needed some constructs that we do not currently handle, including CASE constructs in the SELECT clause, complex WHERE clause conditions with negation and disjunction, multiple levels of nesting in subqueries, and DDL statements.

For each assignment, a correct SQL query Qi was chosen, and was used to generate datasets. A few of these queries are shown in Table 1, which also shows the number of datasets generated for each query; a complete list can be found in [3]. We note that out of 15 queries, 7 queries used features that are addressed by the techniques described in this paper.

The time taken for generating all the datasets for these queries (including the time taken by our code and the CVC3 solver) ranged from 6.7 to 49 seconds, with an average of 24.6 seconds on a computer with an Intel(R) Core(TM) i5-2500K 3.30GHz CPU, and 8 GB of memory, running Ubuntu with Linux kernel 2.6.38. Each dataset was small in size. For simple queries, the datasets had at most two tuples per relation. And for queries involving aggregation or subqueries, the datasets had at most 5 tuples for some relations.

Each student query Qi_j for an assignment was compared with the correct query Qi. To do so, we execute an SQL query of the form (Qi_j EXCEPT ALL Qi) UNION (Qi EXCEPT ALL Qi_j), and if the result of the above query is non-empty, the student query Qi_j is marked as incorrect. Student queries which gave identical results to Qi on all test datasets are marked as correct.

As comparison points, we also tested the queries with two sample University databases provided with the textbook by Silberschatz et al. [10], and with the result of manual cor-

QId	DS	Query
Q5	8	SELECT DISTINCT course.dept_name FROM course NATURAL JOIN section WHERE section.semester='Spring' AND section.year='2010'
Q7	4	SELECT course_id, COUNT(DISTINCT id) FROM course NATURAL LEFT OUTER JOIN takes GROUP BY course_id
Q8	11	SELECT DISTINCT course_id, title FROM course NATURAL JOIN section WHERE semester = 'Spring' and year = 2010 and course_id not in (SELECT course_id FROM prereq)
Q10	6	SELECT DISTINCT dept_name FROM course WHERE credits = (SELECT max(credits) FROM course)
Q12	4	SELECT student.id, student.name FROM student WHERE lower(student.name) like '%sr%'
Q14	6	SELECT DISTINCT * FROM takes T WHERE (NOT EXISTS (SELECT id, course_id FROM takes S WHERE grade ≠ 'F' AND T.id = S.id AND T.course_id = S.course_id) and T.grade IS NOT NULL) or (grade ≠ 'F' AND T.grade IS NOT NULL)

Table 1: Partial List of queries, with number of datasets generated by XData

QId	Queries	XData		Univ. sm.		Univ. lg.		TA	
		√	×	√	×	√	×	√	×
Q1	55	53	2	53	2	53	2	53	2
Q2	57	56	1	56	1	56	1	56	1
Q3	71	58	13	59	12	59	12	70	1
Q4	78	52	26	52	26	75	3	77	1
Q5	72	49	23	61	11	56	16	59	13
Q6	61	55	6	55	6	55	6	59	2
Q7	77	52	25	54	23	75	3	53	24
Q8	79	46	33	67	12	65	14	63	16
Q9	80	37	43	56	24	10	70	57	23
Q10	74	73	1	73	1	73	1	74	0
Q11	69	53	16	53	16	53	16	53	16
Q12	70	62	8	67	3	63	7	63	7
Q13	72	64	8	63	9	63	9	65	7
Q14	67	58	9	53	14	57	10	32	35
Q15	72	72	0	72	0	72	0	72	0

Table 2: Query grading results

rection by course TAs. The first University database, which we call Univ. sm. is a small database which was manually created by the authors of [10] to catch common errors; the second larger database, which we call Univ. lg. is a larger database with randomly generated data. The TAs used a combination of testing against sample databases they created, and their own reading of the queries.

The results of the evaluations are given in Table 2. These results indicate that, overall, the datasets generated by XData were significantly more effective than the two University datasets from [10], and, overall, the manually constructed small University dataset fared better than the randomly generated University dataset. There were only two cases where XData performed significantly worse than at least one of the two University databases.

Overall, XData fared much better than the TAs, catching more errors. The actual effectiveness of TAs is a little better than what the table indicates, since there were some queries where students made minor errors such as including extra attributes, which the TAs decided to ignore as irrelevant, but which were caught by all the datasets.[3] However these cases do not affect the overall results significantly.

Several of the cases where XData performed better were related to duplicate results; the full version of the paper [3] provides more details.

There were only two queries where XData did not fare as well as the University datasets, Q9 and Q14. Q9 had a subquery with constrained aggregation, a case we don't currently handle. Further, some of the erroneous queries had missing group by attributes, a mutation we do not currently handle (we are working on this case). Q14 used a disjunction and two levels of nested subqueries, which our technique does not handle fully currently. On all other queries, XData fared as well as or better than the competition.

9. CONCLUSIONS

The XData system is designed to generate data for killing SQL query mutations. In this paper we introduced novel techniques to handle a number of widely used constructs that XData did not handle earlier, and presented a study showing their practical importance and effectiveness.

We are currently working on extending XData to handle disjunctions, nested subqueries within subqueries, constrained aggregation on subquery results, further string constraints, missing/extra group by attributes, etc. Creation of datasets that can handle multiple (possibly chained) queries in an application is another area of current work.

Acknowledgements: We would like to thank Biplab Kar and Junaid Mohammed for their contributions to the code and performance evaluation.

10. REFERENCES

[1] C. Barrett and C. Tinelli. CVC3. In *Computer Aided Verification (CAV)*, pages 298–302, 2007.

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

[3] B. Chandra, B. Chawda, S. Shah, S. Sudarshan, and A. Shah. Extending XData to kill SQL query mutants in the wild. In *Technical Report, IITB*, 2013.

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

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

[6] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: a solver for string constraints. In *Intl. Symp. on Software Testing and Analysis*, 2009.

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

[8] P. Saxena, D. Akhawe, S. McCamant, and D. Song. KALUZA. http://webblaze.cs.berkeley.edu/2010/kaluza/.

[9] 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 *ICDE*, 2011.

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

[11] 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.

[12] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *ICST*, pages 498–507, 2010.

[13] M. Veanes, N. Tillmann, and J. de Halleux. Qex: Symbolic SQL query explorer. In *LPAR*, pages 425–446, 2010.

[3] If students had been told that their queries would be graded by a tool, they would have taken care to avoid such errors.