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

Shetal Shah, S. Sudarshan, Suhas Kajbaje, Sandeep Patidar, Bhanu Pratap Gupta, Devang Vira

Slides by Sunny Raj Rathod
Updated and Presented by Saurabh Sarda
Outline

- Motivation
- Mutation Testing
- Related Work
- Contributions
- Implementation[XDa-TA]
- Experiments
- Extensions
- Future Work
Testing SQL Queries : A Challenge

- Complex SQL queries hard to get right
- **Question**: How to check if an SQL query is correct?
  - Formal verification is not applicable since we do not have a separate specification and an implementation
  - State of the art solution: manually generate test databases and check if the query gives the intended result
    - Often misses errors
Generating Test Data: Prior Work

- **Automated Test Data generation**
  - Based on database constraints, and SQL query
    - Agenda [Chays et al., STVR04]
  - Reverse Query Processing [Binning et al., ICDE07]
    - takes desired query output and generates relation instances
    - Handle a subset of Select/Project/Join/GroupBy queries
  - Extensions of RQP for performance testing
    - guarantees cardinality requirements on relations and intermediate query results

- None of the above guarantee anything about detecting errors in SQL queries

- **Question**: How do you model SQL errors? **Answer**: Query Mutation
Mutation Testing

- Mutant: Variation of the given query
- Mutations model common programming errors, like
  - Join used instead of outerjoin (or vice versa)
  - Join/selection condition errors
    - \(<\) vs. \(\leq\), missing or extra condition
  - Wrong aggregate (min vs. max)
- Mutant may be the intended query
Mutation Testing of SQL Queries

- Traditional use of mutation testing has been to check coverage of dataset
  - Generate mutants of the original program by modifying the program in a controlled manner
  - A dataset **kills** a mutant if query and the mutant give different results on the dataset
  - A dataset is considered **complete** if it can kill all non-equivalent mutants of the given query
- **Our goal**: generating dataset for testing query
  - Test dataset and query result on the dataset are shown to human, who verifies that the query result is what is expected given this dataset
  - Note that we do not need to actually generate and execute mutants
Related Work

- Tuya and Suarez-Cabal [IST07], Chan et al. [QSIC05] defined a class of SQL query mutations
  - Shortcoming: do not address test data generation
- More recently (and independent of our work) de la Riva et al [AST10] address data generation using constraints, with the Alloy solver
  - Do not consider alternative join orders
  - No completeness results
  - Limitations on constraints
Contributions

● Principled approach to test data generation for given query

● Define class of mutations:
  ○ Join/outerjoin
  ○ Selection condition
  ○ Aggregate function

● Algorithm for test data generation that kills all non-equivalent mutants in the above class for a (fairly large) subset of SQL.
  ○ Under some simplifying assumptions
  ○ With the guarantee that generated datasets are small and realistic, to aid in human verification of results
NP Hardness of Data Generation Problem

- Data Generation Problem: Is there an assignment of tuples to each relation in query, Q such that the result of Q and its mutation Q’ differ?
- Query Containment Problem: Given two SQL queries Q₁ and Q₂, is Q₂ contained in Q₁? (Already known to be NP-complete)
- Reduction: Consider Q₂ ⊨ Q₁ and Q₂ ⊬ Q₁.
  - If Data Generation Problem assigns tuples to the relation in Q₁ and Q₂ such that the results of above two trees differ than Q₂ is not contained in Q₁.
  - If there is no such assignment, Q₂ is contained in Q₁.
Killing Join Mutants : Example 1

Example 1: Without foreign key constraints

Schema: r(A), s(B)

- To kill this mutant: ensure that for some $r$ tuple there is no matching $s$ tuple
- Generated test case: $r(A) = \{(1)\}; s(B) = \{\}$
- Basic idea, version 1 [ICDE 2010]
  - run query on given database
  - from result extract matching tuples for $r$ and $s$
  - delete $s$ tuple to ensure no matching tuple for $r$
- Limitation: foreign keys, repeated relations
Killing Join Mutants: Example 2

Example 2: Extra join above mutated node

Schema: \( r(A,B), s(C,D), t(E) \)

- To kill this mutant we must ensure that for an \( r \) tuple there is no matching \( s \) tuple, but there is a matching \( t \) tuple.
- Generated test case: \( r(A,B)=\{(1,2)\}; s(C,D)=\{\}; t(E)=\{(2)\} \)
Killing Join Mutants : Example 3

Example 3: Equivalent mutation due to join

Schema: r(A, B), s(C, D), t(E)

- Note: right outer join this time
- Any result with a \( r.B \) being null will be removed by join with \( t \)
- Similarly equivalence can result due to selections
Killing Join Mutants: Example 4

- \textit{teaches} \Join \textit{instructor} is equivalent to \textit{teaches} \Join \textit{instructor} if there is a foreign key from \textit{teaches.ID} to \textit{instructor.ID}
- The two expressions are no longer equivalent if \textit{instructor} is replaced with
  \[
  \sigma_{\text{instructor.dept}=\text{CS}}(\text{instructor})
  \]
- Key idea: have a \textit{teaches} tuple with an \textit{instructor} not from CS
- Selections and joins can be used to kill mutations
Killing Join Mutants: Equivalent Trees

- **Space of join-type mutants**: includes mutations of join operator of a single node for all trees equivalent to given query tree
- Datasets should kill mutants across all such trees
Killing Join Mutants: Equivalent Trees

Whether a query is written in the form \( A.x = B.x \) AND \( B.x = C.x \), or \( A.x = B.x \) AND \( A.x = C.x \) should not affect set of mutants generated

Solution: Equivalence classes of attributes

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

- A1, A2: Only primary and foreign key constraints; foreign key columns not nullable
- A3: Single block SQL queries; no nested subqueries
- A4: Expr/functions: Only arithmetic expressions
- A5: Join/selection predicates: conjunctions of \{expr relop expr\}
- A6: Queries do not explicitly check for null values using IS NULL
- A7: In the presence of full outer join, at least one attribute from each of its inputs present in the select clause (and A8 for natural join: see paper)
Generating Constraints to kill Join Mutations

There exists a tuple in $A$ for which there does not exist any matching tuple in $B$.

There exists a tuple in $B$ for which there does not exist any matching tuple in $A$. 
Problems

- Translate high level requirements into constraints on individual tuples
- Ensure the difference exposed at an internal node is propagated to root
- Exponential number of join trees
- Repeated relation occurrences
Data Generation in 2 Steps

● Step 1: Generation of constraints
  ○ Constraints due to the schema
  ○ Constraints due to the query
  ○ Constraints to kill a specific mutant

● Step 2: Generation of data from constraints Using solver, currently CVC3
Data Generation Algorithm: Overview

Algorithm 1: Main Algorithm

1: Hashtable currentIndex; /* Maps distinct relation names to offsets in the CVC3 array created for the corresponding database relation */
2: procedure generateDataSet(query q)
3:  preprocess query tree
4:  initializeIndices() /* Initializes currentIndex and other related structures */
5:  generateDataSetForOriginalQuery()
6:  killEquivalenceClasses()
7:  killOtherPredicates()
8:  killComparisonOperators()
9:  killAggregates()
10: end procedure
Preprocess Query Tree

- **Build Equivalence Classes from join conditions**
  - A.x = B.y and B.y = C.z
  - Equivalence class: A.x, B.y and C.z

- **Foreign Key Closure** –
  - A.x -> B.y and B.y -> C.z then A.x -> C.z

- **Retain all join/selection predicates other than equijoin predicates**
Helper Functions

- **CvcMap**(*rel*.attr)
  - Takes a *rel* and *attr* and returns r[i].pos where
    - r is base relation of *rel*
    - pos is the position of attribute *attr*
    - i is an index in the tuple array *

- **GenerateEqConds** (P)
  - Generates equality constraints amongst all elements of an equivalence class P
Killing Equi Join Condition Mutations

Algorithm 2 : killEquivalenceClasses()

1: for each equivalence class ec do
2:     Let allRelations := Set of all \langle rel, attr \rangle pairs in ec
3:     for each element e in allRelations do
4:         conds := empty set
5:         Let e := R.a
6:         \( S := ( \text{set of elements in } ec \text{ which are foreign keys referencing } R.a \text{ directly or indirectly}) \text{ UNION } R.a \)
7:         \( P := ec - S \)
8:         if \( P \).isEmpty() then
9:             continue
10:        end if
11:        conds.add(generateEqConds(P))
12:        conds.add(
            “NOT EXISTS i: R[i].a = ” + cvcMap(P[0]))
Killing Equi Join Condition Mutations (contd.)

for all other equivalence classes \(oec\) do

\(cond{\text{s}}.add(\text{generateEqConds}(oe))\)

end for

for each other predicate \(p\) do

\(cond{\text{s}}.add(\text{cvcMap}(p))\)

end for

\(cond{\text{s}}.add(\text{genDBConstraints})\)

callSolver(\(cond{\text{s}}\))

if solution exists then

create a dataset from solver output

end if

end for
Killing Other Predicates

- Create separate dataset for each attribute in predicate
- e.g. For Join condition B.x = C.x + 10
  - Dataset 1 (nullifying B:x):
    
    \[
    \text{ASSERT NOT EXISTS } (i : \text{B\_INT}) : (B[i].x = C[1].x + 10);
    \]
  
  - Dataset 2 (nullifying C:x):
    
    \[
    \text{ASSERT NOT EXISTS } (i : \text{C\_INT}) : (B[1].x = C[i].x + 10);
    \]
Killing Comparison Operator Mutations

- Example of comparison operation mutations:
  - A < 5 vs. A <= 5 vs. A > 5 vs A >= 5 vs. A=5, vs A <> 5

- Idea: generate separate dataset for three cases (leaving rest of query unchanged):
  - A < 5
  - A = 5
  - A > 5

- This set will kill all above mutations
Killing Unconstrained Aggregation Mutations

- Aggregation operations
  - count(A) vs. count(distinct A)
  - sum(A) vs sum(distinct A)
  - avg(A) vs avg(distinct A)
  - min(A) vs max(A)
  - and mutations amongst all above operations

- Idea: given relation \( r(G, O, A) \) and query `select aggop(A) from r group by G` Tuples \((g1, o1, a1), (g1, o2, a1), (g1, o3, a2)\), with \(a1 <> 0\) will kill above pairs of mutations

- Additional constraints to ensure killing mutations across pairs
Aggregation Operation Mutation

- **Issues:**
  - Database/query constraints forcing A to be unique for a given G
  - Database/query constraints forcing A to be a key
  - Database/query constraints forcing G to be a key

- Carefully crafted set of constraints, which are relaxed to handle such cases
Completeness Results

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

- Completeness results for restricted classes of aggregation mutations
  - aggregation as top operation of tree, under some restrictions on joins in input
Experimental Results

- x86 machines, 1.86 GHz processor, 2 GB main memory
- Schema: University database from Database System COncepts (6th ed.)
- Queries with joins with varying number of foreign keys imposed
- Queries with comparison, aggregation and inner joins
## Inner Join Queries

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

**TABLE I**

**RESULTS FOR INNER JOIN QUERIES**
# Selection/Aggregation Queries

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

**TABLE II**  
**RESULTS FOR QUERIES WITH SELECTION/AGGREGATION**
Extensions

- Handling Nulls
- String Constraints
- Constrained Aggregation

Source:
Extending XData to kill SQL query mutants in the wild
Handling Nulls

- For text attributes, enumerate a few more values in the enumerated type and designate them NULLs.
  - 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.
- Add constraints forcing those attribute values to take on one of the above mentioned special values representing NULL.
- Add constraints to force all other values to be non null
- Enables handling of nullable foreign keys, and explicit IS NULL checks
### String Constraints

- **String Constraints**
  - S1 likeop pattern
  - S1 relop constant
  - strlen(S) relop constant
  - S1 relop S2

where
- S1 and S2 are string variables,
- *likeop* is one of LIKE, ILIKE (case insensitive like), NOT LIKE and NOT ILIKE
- *relop* operators are =, <, ≤, >, ≥, <>, and case-insensitive equality denoted by =.
String Constraints

- **String solver**

- **String constraint mutation:** \{=, \textless\textgreater, \#, \textgreater\textless, \leq, \geq\}
  
  - $S_1 = S_2$
  
  - $S_1 > S_2$
  
  - $S_1 < S_2$

- **LIKE predicate mutation:** \{LIKE, ILIKE, NOT LIKE, NOT ILIKE\}
  
  - Dataset 1 satisfying the condition $S_1$ LIKE pattern.
  
  - Dataset 2 satisfying condition $S_1$ ILIKE pattern, but not $S_1$ LIKE pattern
  
  - Dataset 3 failing both the LIKE and ILIKE conditions
Constrained Aggregation Operation

- Aggregation Constraints: Example: \( \text{SUM}\ (r.a) > 20 \)
- CVC3 requires us to specify how many tuples \( r \) has.
- Hence, before generating CVC3 constraints we must
  (a) estimate the number of tuples \( n \), required to satisfy an aggregation constraint
  (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.
XDa-TA System

- For each query in an assignment, a correct SQL query is given to the tool, which generates datasets for killing mutants of that query.
- Modes:
  - admin mode
  - student mode.
- Assignment can be marked as:
  - learning assignment
  - graded assignment.

Source:
XDa-TA : Automating Grading of SQL Query Assignments
Sample Query Set

<table>
<thead>
<tr>
<th>QId</th>
<th>DS</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q5</td>
<td>8</td>
<td>SELECT DISTINCT course.dept_name FROM course NATURAL JOIN section WHERE section.semester='Spring' AND section.year='2010'</td>
</tr>
<tr>
<td>Q7</td>
<td>4</td>
<td>SELECT course_id, COUNT(DISTINCT id) FROM course NATURAL LEFT OUTER JOIN takes GROUP BY course_id</td>
</tr>
<tr>
<td>Q8</td>
<td>11</td>
<td>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)</td>
</tr>
<tr>
<td>Q10</td>
<td>6</td>
<td>SELECT DISTINCT dept_name FROM course WHERE credits = (SELECT max(credits) FROM course)</td>
</tr>
</tbody>
</table>
| 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) |
## Results

<table>
<thead>
<tr>
<th>QId</th>
<th>Queries</th>
<th>XData</th>
<th>Univ. sm.</th>
<th>Univ. lg.</th>
<th>TA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Q1</td>
<td>55</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Q2</td>
<td>57</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Q3</td>
<td>71</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Q4</td>
<td>78</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Q5</td>
<td>72</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Q6</td>
<td>61</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Q7</td>
<td>77</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Q8</td>
<td>79</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Q9</td>
<td>80</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Q10</td>
<td>74</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Q11</td>
<td>69</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Q12</td>
<td>70</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Q13</td>
<td>72</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Q14</td>
<td>67</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Q15</td>
<td>72</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Future Work

- Integration with course management systems such as Moodle or Blackboard using the Learning Tools Interoperability (LTI) standard (complete)
- Partial Marking Scheme implementation (ongoing)
- Future work:
  - Handling SQL features not supported currently
  - Multiple queries
  - Form parameters
Thank You!

Questions?