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

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

CSE Department, IIT Bombay

Presented By: Sunny Raj Rathod

Outline

- Motivation
- Mutation Testing
- Related Work
- Contributions
- Extensions
- Implementation[XDa-TA]
- Experiments
- Future Work

Testing SQL Queries: A Challenge

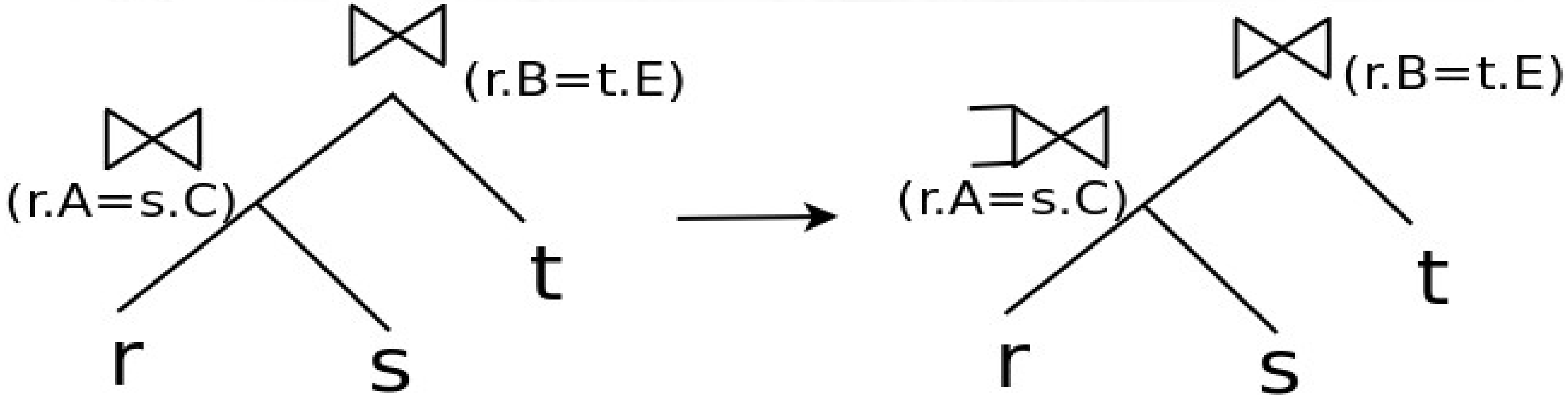
- Complex SQL queries hard to get right
- Question: How to check if an SQL query is correct?
  - Formal verification is not applicable since we do not have a separate specification and an implementation
  - State of the art solution: manually generate test databases and check if the query gives the intended 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. <=, 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

- Prior 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

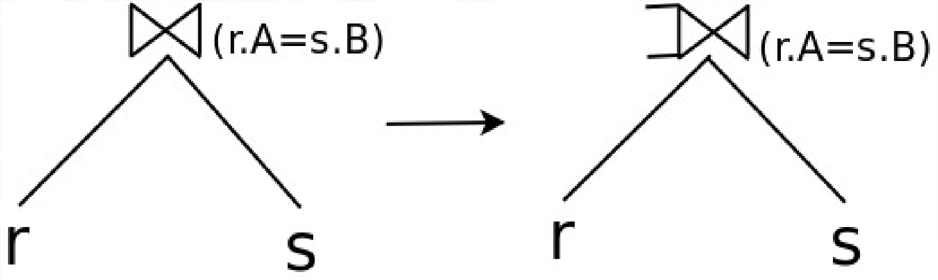
Our Contributions

- 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 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

Killing Join Mutants: Example 1

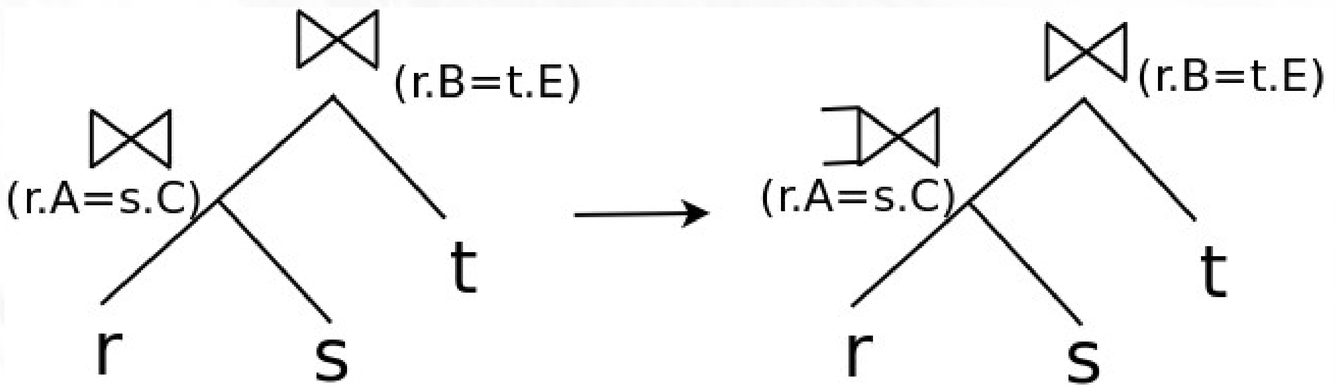
- **Example 1: Without** foreign key constraints
  - Schema: *r(A), s(B)*



- 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

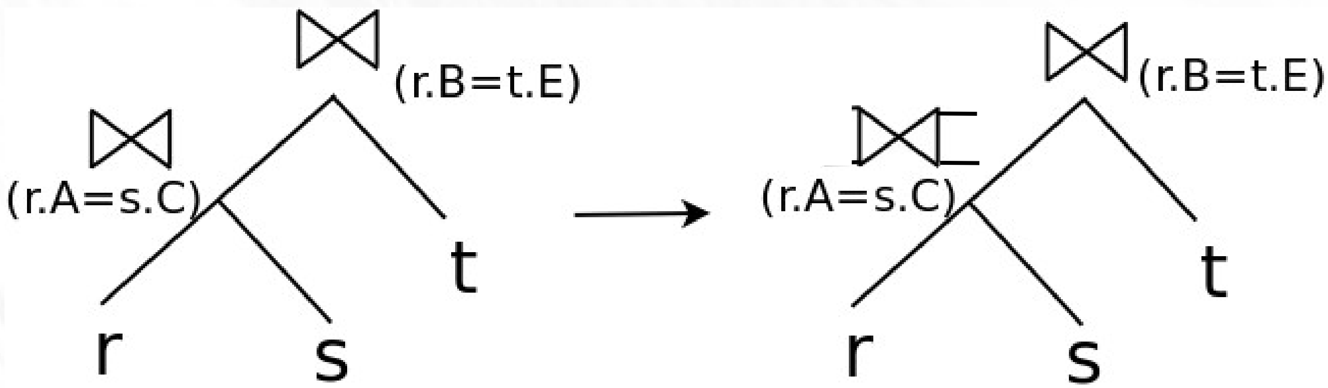
- **Example 2**: Extra join above mutated node
  - Schema: *r(A,B), s(C,D), t(E)*



- 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

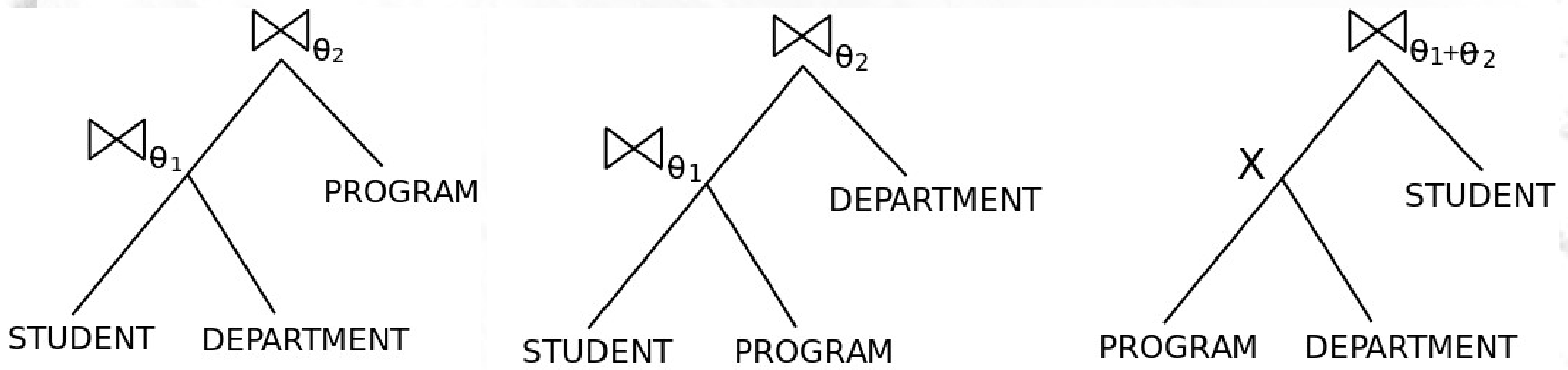
teaches ⟕ *instructor*
is ***equivalent*** *to teaches* $\bowtie$ *instructor if there is a foreign key from teaches.ID to instructor.ID*

BUT: teaches ⟕ $\sigma_{dept=CS}$*(instructor)*
is ***not equivalent*** *to*
teaches $\bowtie$ $\sigma_{dept=CS}$*(instructor)*

Key idea: have a teaches tuple with an instructor not from CS

Selections and joins can be used to kill mutations

Killing Join Mutants: Equivalent Trees



Query Tree 1 **Query Tree 2** **Query Tree 3**

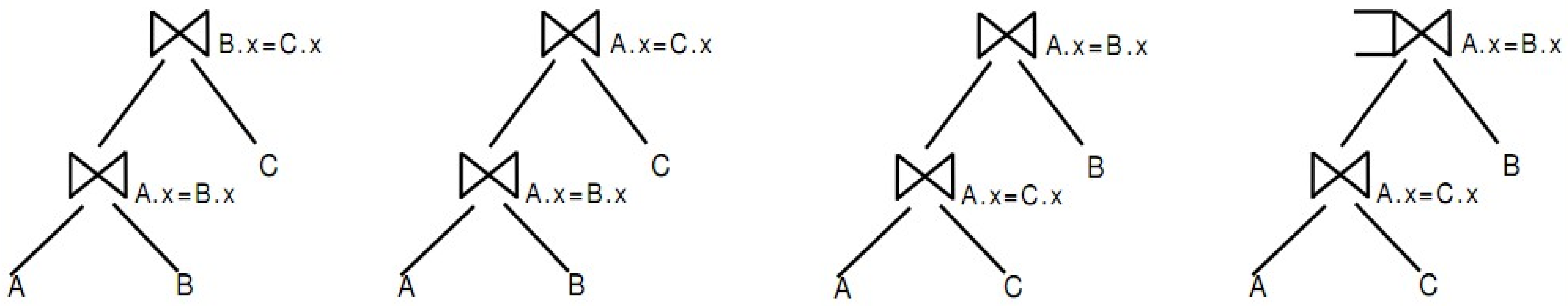
- **Space of join-type mutants**: includes mutations of join operator of a single node for all trees equivalent to given query tree
- Datasets should kill mutants across all such trees

Equivalent Trees and Equivalence Classes of Attributes

- Whether query conditions written as
  - A.x = B.x AND B.x = C.x or as
  - 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 exprs
- 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)

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

Running Example : University Schema (Book)

```
SELECT *
FROM crse, dept, teaches
WHERE crse.dept_name = dept.dept_name
AND  crse.course_id = teaches.course_id
```

Relations:

crse(*course_id*, dept_name, credits)

dept(*dept_name*, building, budget)

teaches(instructor_id, course_id, semester,acadyear)

Data Generation Algorithm - Overview

- procedure generateDataSet(query q)
  - preprocess query tree
  - generateDataSetForOriginalQuery()
  - killEquivalenceClasses()
  - killOtherPredicates()
  - killComparisonOperators()
  - killAggregates()

Preprocess Query Tree

- Build Equivalence Classes from join conditions
  - A.x = B.y and B.y = C.z then

    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

Dataset for Original Query

- Generate datatype declarations for CVC3

```
DATATYPE COURSE_ID = BIO101 | BIO301 | BIO399 | CS101 |
   CS190 | CS315 | CS319 | CS347 | CS630 | CS631 | CS632 |
   EE181 | FIN201 | HIS351 | MU199 | PHY101 END;

 CREDITS : TYPE = SUBTYPE (LAMBDA (x: INT) : x > 1 AND x < 5);
```

- Array of tuples of constraint variables, per relation

```
CRSE_TupleType: TYPE = [COURSE_ID, DEPT_NAME, CREDITS];

O_CRSE: ARRAY INT OF CRSE_TupleType;

TEACHES_TupleType: TYPE = [INSTRUCTOR_ID, COURSE_ID,
                           SEMESTER, ACADYEAR];

O_TEACHES: ARRAY INT OF TEACHES_TupleType
```

O_CRSE[1].0 is a constraint variable corresponding to COURSE_ID of the first tuple

Dataset for Original Query

- One or more constraint tuples from array, for each occurrence of a relation

```
O_CRSE_INDEX_INT : TYPE = SUBTYPE (LAMBDA (x: INT) : x > 0 AND x < 2);
O_DEPT_INDEX_INT : TYPE = SUBTYPE (LAMBDA (x: INT) : x > 0 AND x < 2);
O_TEACHES_INDEX_INT : TYPE = SUBTYPE (LAMBDA (x: INT) : x > 0 AND x < 2);
```

  - More than 1 tuple required for aggregation, repeated occurrences or to ensure f.k. Constraints

- Equality conditions between variables based on equijoins

```
ASSERT (O_CRSE[1].1 = O_DEPT[1].0) ;
ASSERT O_CRSE[1].0 = O_TEACHES[1].
```

- Other selection and join conditions become constraints

Dataset for Original Query (DB Constraints)

- Constraints for primary and foreign keys
  - f.k. from crse.deptname to dept.dept_name
    - `ASSERT FORALL i EXISTS j (O_CRSE[i].1 = O_DEPT[j].0);`
  - p.k. on R.A
    - `ASSERT FORALL i FORALL j (O_CRSE[i].0 = O_CRSE[j].0) => “all other attrs equal”`
    - Why not assert primary key value is distinct (supported by CVC3)?
- Since range is over finite domain, p.k. and f.k. constraints can be unfolded
  - Unfolded constraints:

f.k : `ASSERT O_CRSE[1].1 = O_DEPT[1].0 OR O_CRSE[1].1 = O_DEPT[2].0`

p.k : `ASSERT (O_DEPT[1].0 = O_DEPT[2].0 ) => (O_DEPT[1].1 = O_DEPT[2].1) AND (O_DEPT[1].2 = O_DEPT[2].2) ;`

Helper Functions

- CvcMap
  - 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 Join Mutants: Equijoin

killEquivalenceClasses()

- for each equivalence class ec do
  - Let allRelations := Set of all <rel, attr> 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
    - else … main code for generating constraints (see next slide)

Killing Join Mutants: EquiJoins

- conds.add(generateEqConds(P))
- conds:add(
  “NOT EXISTS i: R[i].a = ” + cvcMap(P[0]))
- for all other equivalence classes *oe* do
  - conds.add(generateEqConds(*oe*))
- for each other predicate p do
  - conds:add(cvcMap(p))
- conds.add(genDBConstraints()) /*P.K. and F.K*/
- callSolver(conds)
- if solution exists then
  - create a dataset from solver output

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):
    - `ASSERT NOT EXISTS (i : B_INT) : (B[i].x = C[1].x + 10);`
  - Dataset 2 (nullifying C:x):
    - `ASSERT NOT EXISTS (i : C_INT) : (B[1].x = C[i].x + 10);`

Comparison Operation 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

Aggregation Operation 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 Mutants

- 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

Complexity

- Number of datasets generated is linear in query size
- Although solving constraints is in general NP-hard, and even undecidable with arbitrary constraints, it is tractable in special cases.

Extensions

- Unintended Joins
- Nested subqueries
- Handling NULLs
- String Constraints
- Distinct
- Others – Set ops, Parameterized Queries, Date-Time, Insert, Update, Delete, Disjunctions

Sources :

Extending XData to kill SQL query mutants in the wild

XDa-TA : Automating Grading of SQL Query Assignments

Unintended Join Conditions

- Unintended join conditions can be explicitly added by the user in the where clause of the query or by using **natural joins** instead of theta joins.
- Example :
  - Schema :
    - student (id, name,dept name)
    - course (course id, name, dept name)
    - takes (id, course id, sec id, semester, year)
  - Query to find the list of all courses taken by a student with id = 1234 is:

```
SELECT course id,course name FROM student
INNER JOIN takes on(id)
INNER JOIN course ON(course id) WHERE student.id = 1234
```

  - Dataset Generated :
    - Student (1234, Alice, EE)
    - course (CS-317, Database Systems, CS)
    - takes (1234, CS-317, 1, Fall, 2014)

Constrained Aggregation Operation

- Aggregation Constraints: Example : 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.

Changed Group By Attributes

- Schema : `takes (id, course id, sec id, semester, year, section)`

- Example : find the number of students taking each course every time it is offered.

```
SELECT count(id), course id, semester, year FROM takes
GROUP BY course id, semester, year
```

- Erroneous query misses out students who have taken the same course in different sections.

```
SELECT count(id), course id, semester, year FROM takes
GROUP BY course id, semester, year, section
```

- Example tuples for dataset:

```
t1 (1234, CS-317, 1, Fall, 2014, section 1)
t2 (1234, CS-317, 1, Fall, 2014, section 2)
```

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.

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: {=, <>, <, >, ≤, ≥}
  (1) S1 = S2 (2) S1 > S2(3) S1 < S2

- LIKE predicate mutation: {LIKE, ILIKE,NOT LIKE, NOT ILIKE }
  - Dataset 1 satisfying the condition S1 LIKE pattern.
  - Dataset 2 satisfying condition S1 ILIKE pattern,
  but not S1 LIKE pattern
  - Dataset 3 failing both the LIKE and ILIKE conditions

XDa-TA

- 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: i) admin mode
  ii) student mode.

- Assignment can be marked as :
  1. learning assignment
  2. graded assignment.

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

Performance Results

- University database schema from Database System Concepts 6th Ed
- Queries with joins, with varying number of foreign keys imposed

Results for inner join queries

Query	#Joins (#Relations)	#FK	#Datasets Generated	#Mutants Killed	Total Time(s) without Unfolding	Total Time(s) 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

Results for queries with selections,aggregations

Query	#Joins	#Selections	#Aggregations	#Data sets Gen.	#Mutants killed	Total Time(s) without Unfolding	Total Time(s) 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

QId	DS	Query
Q0	5	CREATE VIEW rich_instructors AS SELECT id,name,dept_name,salary FROM instructor WHERE salary>50000
Q1	2	SELECT course_id, title FROM course
Q2	5	SELECT course_id, title FROM course WHERE dept_name= 'Comp. Sci.'
Q3	9	SELECT DISTINCT course.course_id, course.title, ID FROM course NATURAL JOIN teaches WHERE teaches.semester='Spring' AND teaches.year='2010'
Q4	6	SELECT DISTINCT student.id, student.name FROM takes NATURAL JOIN student WHERE course_id ='CS-101'
Q5	8	SELECT DISTINCT course.dept_name FROM course NATURAL JOIN section WHERE section.semester='Spring' AND section.year='2010'
Q6	5	SELECT course_id, title FROM course WHERE credits > 3
Q7	8	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)
Q9a	25	WITH s as (SELECT id,time_slot_id,year,semester FROM takes NATURAL JOIN section GROUP BY id,time_slot_id,year,semester HAVING count(time_slot_id)>1) SELECT DISTINCT id,name FROM s NATURAL JOIN student

Q9b	22	SELECT distinct A.id, A.name FROM (SELECT * from student NATURAL JOIN takes NATURAL JOIN section) A, (SELECT * from student NATURAL JOIN takes NATURAL JOIN section) B WHERE A.name = B.name and A.time_slot_id = B.time_slot_id and A.course_id <> B.course_id and A.semester = B.semester and A.year = B.year
Q10	7	SELECT DISTINCT dept_name FROM course WHERE credits = (SELECT max(credits) FROM course)
Q11	4	SELECT DISTINCT instructor.ID,name,course_id FROM instructor LEFT OUTER JOIN TEACHES ON instructor.ID = teaches.ID
Q12	5	SELECT student.id, student.name FROM student WHERE lower(student.name) like '%sr%'
Q13	10	SELECT id, name FROM student NATURAL LEFT OUTER JOIN (SELECT id, name, course_id FROM student NATURAL LEFT OUTER JOIN takes WHERE year = 2010 and semester = 'Spring') S WHERE course_id IS NULL
Q14	19	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 (T.grade ! = 'F' AND T.grade IS NOT NULL)

Query grading results

QId	Queries	XDa-TA		USm		ULg		TA		Plan	
		√	×	√	×	√	×	√	×	√	?
Q0	72	72	0	72	0	72	0	72	0	-	-
Q1	55	53	2	53	2	53	2	53	2	51	4
Q2	57	56	1	56	1	56	1	56	1	54	3
Q3	71	58	**13**	59	12	59	12	70	1	3	68
Q4	78	52	**26**	52	**26**	75	3	77	1	10	26
Q5	72	49	**23**	61	11	56	16	59	13	43	29
Q6	61	55	6	55	6	55	6	59	2	55	4
Q7	77	52	**25**	54	23	75	3	53	24	3	73
Q8	79	46	**33**	67	12	65	14	63	16	2	77
Q9a	80	12	68	56	24	10	70	57	23	2	78
Q9b	80	9	71	56	24	10	70	57	23	3	77
Q9	80	8	**72**	56	24	10	70	57	23	5	75
Q10	74	73	1	73	1	73	1	74	0	34	40
Q11	69	53	16	53	16	53	16	53	16	51	18
Q12	70	62	**8**	67	3	63	7	63	7	38	32
Q13	72	64	8	63	**9**	63	**9**	65	7	3	69
Q14	67	39	28	53	14	57	10	32	**35**	2	65

Table 2: Query grading results

Future Work

- Ongoing work
  - Integration with course management systems such as Moodle or Blackboard using the Learning Tools Interoperability (LTI) standard
- Future work:
  - Handling SQL features not supported currently
  - Multiple queries
  - Form parameters

Questions

Thank You