Data Generation for Testing and Grading SQL Queries

Bikash Chandra · Amol Bhangadia · Bhupesh Chawda · Biplab Kar · K V Maheshwara Reddy · Shetal Shah · S Sudarshan
IIT Bombay
{bikash, amolb12, biplabkar11, kvmahesh12, shetals, sudarsha}@cse.iitb.ac.in, bhchawda@in.irl.com

Abstract
Correctness of SQL queries is usually tested by executing the queries on one or more datasets. Erroneous queries are often the results of small changes, or mutations of the correct query. A mutation Q' of a query Q is killed by a dataset D, if Q(D) ≠ Q'(D). 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.

In this paper we extend the XData data generation techniques to handle a wider variety of SQL queries and a much larger class of mutations. We have also built a system for grading SQL queries using the datasets generated by XData. We present a study of the effectiveness of the datasets generated by the extended XData approach, using a variety of queries including queries submitted by students as part of a database course. We show that the XData datasets outperform predefined datasets as well as manual grading done earlier by teaching assistants, while also avoiding the drudgery of manual correction. Thus, we believe that our techniques will be of great value to database course instructors and TAs, particularly to those of MOOCs. It will also be valuable to database application developers and testers for testing SQL queries.

Keywords
Mutation Testing, Test Data Generation

1 Introduction

Queries written in SQL are used in a variety of different applications. An important part of testing these applications is to test the correctness of SQL queries in these applications. The queries are usually tested using multiple ad hoc test cases provided by the programmer or the tester. Queries are run against these test cases and tested by comparing the results with the intended one manually or by automated test cases. However this approach involves manual effort in terms of test case generation and also does not ensure whether all the relevant test cases have been covered or not. Formal verification techniques involve comparing a specification with an implementation. However, since SQL queries are themselves specifications and do not contain the implementation, formal verification techniques cannot be applied for testing SQL queries.

A closely related problem is that of grading SQL queries written by students. Grading SQL queries is usually done by executing the query on small datasets and/or by reading the student query and comparing those with the correct query. Manually created datasets, as well as datasets created in a query independent manner can be incomplete and are likely to miss errors in queries. Manual reading and comparing of queries is difficult, since students may write queries in a variety of different ways, and is prone to errors as graders are likely to miss subtle mistakes. For example, when required to write the query Q below:

```
SELECT course.id, course.title FROM course LEFT OUTER JOIN (SELECT * from department WHERE department.budget > 70000) d USING (dept name);
```

students often write query Qs:

```
SELECT course.id, course.title FROM course LEFT OUTER JOIN department USING (dept name)
WHERE department.budget > 70000;
```

which looks sufficiently similar for a grader to miss the difference. These queries are not equivalent, since they give different results on departments with budget less than 70000.

Mutation testing is a well known approach for checking adequacy of test cases for a program. Mutation testing involves generating mutants of the original program by modifying the program in a controlled manner. For SQL queries we consider that a mutation is single (syntactically 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, allowing us to distinguish between the queries.

A test suite consisting of multiple datasets kills a mutant if at least one of the datasets kills the mutant.
Consider the query:

\[
\text{SELECT dept\_name, COUNT(DISTINCT id) FROM course LEFT OUTER JOIN takes USING(course\_id) GROUP BY dept\_name}
\]

One of the mutants obtained by mutating the join condition of the query is:

\[
\text{SELECT dept\_name, COUNT(DISTINCT id) FROM course INNER JOIN takes USING(course\_id) GROUP BY dept\_name}
\]

Similarly by mutating the aggregation we get the following mutation:

\[
\text{SELECT dept\_name, COUNT(id) FROM course LEFT OUTER JOIN takes USING(course\_id) GROUP BY dept\_name}
\]

We look at the problem of generating datasets that can catch commonly occurring errors in a large class of SQL queries. Queries with common errors can be thought of as mutants of the original query. We only consider single mutations in a query when generating test datasets, since the space of mutants is much larger with multiple mutations. It is possible that an erroneous query may contain multiple mistakes; queries with multiple mutations are likely, but not always guaranteed, to be killed by the datasets we generate. We generate datasets so as to kill a variety of query mutations. These datasets can be used in two distinct ways:

a) To check if a given query is what was intended, a tester manually examines the result of the query on each dataset, and checks if the result is what was intended

b) To check if a student query is correct, the results of the student query and a given correct query are compared on each dataset. A difference on any dataset indicates that the student query is erroneous (We note that checking query equivalence is possible in limited special cases but is in general hard or undecidable [16,17,13]).

There has been increased interest in the recent years in test data generation for SQL queries including [27,23,29,31]; [19] addresses a similar problem in the context of data-flow programs. Our earlier work on the XData system [11,25] showed how to generate datasets that can distinguish the correct query from some class of query mutations, including join and comparison operator mutations. However, real life SQL queries have a variety of features and mutations that were not handled in [11, 25]. (Related work is described in detail in Section 14.)

A few of the techniques described in this paper were sketched in a short workshop paper [8], but details were not presented there.

We describe a number of techniques to generate data for various query constructs and query mutations. Each data generation technique designed to handle a specific query constructs or specific mutations of the query.

The contributions of this paper are as follows.

1. We discuss (in Section 4) how to generate test data and kill mutations for queries involving string predicates such as string comparison and the LIKE predicate, using a string solver we have developed.
2. We support the NULL datatype and various mutations that may arise because of the presence of NULLs (Section 5).
3. We extend the class of mutations to include missing or additional join conditions (Section 6).
4. For queries containing constraints on aggregated results, we describe (in Section 7) a new algorithm to find the number of tuples that need to be generated for each relation to satisfy the aggregation constraints.
5. We support test data generation and mutation killing for a large class of nested subqueries. (Section 8)
6. The class of mutations considered has been extended to include missing or additional group by attributes (Section 9), and distinct clause mutations (Section 10). Queries containing set operators are also supported (Section 11).
7. The data types supported include floating point numbers, time and date values. The class of queries is extended to include insert, delete, update and parameterized queries as well as view creation statements (Section 12).
8. We describe (in Section 13) query testing techniques for grading student queries based on the datasets generated by XData. These techniques can be used for grading, as well as in a learning mode where it can give immediate feedback to students.
9. In Section 15 we present performance results of our techniques. We compare the performance of our string solver to existing string solvers. We generate test data for a number of queries involving constrained aggregation and subqueries on the University database [26] as well as queries of the TPC-H benchmark and show that the datasets generated by XData are able to kill most of the non-equivalent mutations. We also test the effectiveness of our grading tool by using as a benchmark a set of assignments given as part of a database course at IIT Bombay. We show that the datasets generated using our techniques catch more errors than the University datasets, provided with [26], as well as manual grading by the TAs, on almost all the queries.

We believe the techniques presented in this paper will be of great value to database application developers.
and testers for testing real life SQL queries. It will also be valuable to database course instructors and TAs by taking the drudgery out of grading and allow SQL query assignments to be properly checked in MOOC setting, where manual grading is not feasible.

2 Background

In our earlier work on XData [25], we presented techniques for generating test data for killing SQL query mutants; we briefly outline that work below.

2.1 Mutation Space

The mutation space considered consisted of the following

1. Join Type Mutations: A join type mutation involves replacing one of {INNER, LEFT OUTER, RIGHT OUTER} JOIN with another. Consider the mutation from course INNER JOIN department to course LEFT OUTER JOIN department. In order to kill this mutation we need to ensure that there exists a tuple in department relation that does not satisfy the join condition with any tuple in course relation. The INNER JOIN query would give an empty result while the LEFT OUTER JOIN would give a non empty result.

   In SQL, a join query can be specified in a join order independent fashion, with many equivalent join orders for a given query. Hence, the number of join type mutations across all these orders is exponential. From the join conditions specified in the query, XData forms equivalence classes of relation, attribute pairs such that elements in the same equivalence class need to be assigned the same value to meet (one or more) join conditions. Using these equivalence classes, XData generates a linear number of datasets to kill join type mutations across all join orderings. If a pair of relations involve multiple join conditions XData nullifies each join condition separately.

2. Selection Predicate Mutations: For selection conditions XData considers mutations of relop where any occurrence of one of {=, <>, <, >, ≤, ≥} is replaced by another. For killing mutations for the selection condition $A_1$ relop $A_2$ XData generate 3 datasets (1) $A_1 > A_2$, (2) $A_1 < A_2$, and (3) $A_1 = A_2$. These three datasets kill all non equivalent mutations from one relop to another relop. These datasets also kill mutations because of missing selection conditions.

3. Unconstrained Aggregation Mutation: Aggregation at the root of the query tree are not constrained to satisfy any condition. The aggregation function can be mutated among MAX, MIN, SUM, AVG, COUNT and their DISTINCT versions. In order to kill these mutations a dataset with three tuples is generated; 2 with the same value (non-zero) and another with a different value in the aggregate column.

2.2 Approach to Data Generation

Given a SQL query $Q$, XData[25] generates multiple datasets. The first dataset ensures a non-empty result for $Q$ (which itself kills several mutations that would generate an empty result on that dataset). Each of the remaining datasets is targeted to kill 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 to kill these mutations is small.

   To generate a particular dataset, XData generates a set of constraint variables, where each tuple in the target dataset is represented by a tuple of constraint variables along with constraints between these variables. For example, selection conditions, join conditions, primary key and foreign key conditions are all mapped to constraints on these variables. It then invokes a constraint (SMT) solver, CVC3 [3], to solve the constraints; the solution given by the solver defines a dataset on which the query is to be tested. Details are provided in the following section.

   In order to kill mutations the goal of XData is to generate datasets that produce different results on the query and its mutation. In order to do this constraints are added in a manner so as to ensure that the mutation in a node of a query tree is reflected above leading to different results for the query and its mutation. For example consider the following query:

   Example 1

   ```sql
   SELECT course_id, dept_name, budget
   FROM course INNER JOIN department
   WHERE dept_budget > 70000
   ```

   This query has two predicates course INNER JOIN department USING (dept_name) and dept_budget > 70000. When generating datasets to kill the mutations of join predicates we need to ensure that dept_budget > 70000 is satisfied for the tuple generated department table. In case dept_budget > 70000 is not satisfied both the query and the mutant could give empty result.

   Wherever feasible XData generates datasets that produce empty result on either the original query or the mutant but not both. This helps to kill mutations irrespective of the projected attributes.

2.3 Constraint Generation

In CVC3, text attributes are modeled as enumerated types while numeric attributes are modeled as subtypes
of integers or rationals. The data type declarations in CVC3 are as follows. For each attribute of each relation, we specify a set of acceptable values, taken from an input database, as datatypes in CVC3. While the input database is not necessary for data generation, its use makes for improved readability and comprehension of the query results. In case an input database is not specified we get the range from the data type of the corresponding column.

Consider an input database which has CS, Finance, Music and Physics as department names, and department budget is an integer constrained to be between 50000 and 120000. Then, this translates to the following the declarations in CVC3. A tuple type is created for each relation, where each element is a constraint variable of the specified type. A relation is represented as an array of constraint variables; the size of the array has to be determined before solving the constraints, and constraints have to be specified for each attribute of each tuple.

**DATATYPE**

\[
\text{dept\_name = CS | Finance | Music | Physics END;}
\]
\[
\text{dept\_budget:TYPE = SUBTYPE (LAMBDA (x: INT) : x > 49999 AND x < 120001);}
\]
\[
\text{department\_tuple\_type:TYPE = [dept\_name,dept\_budget];}
\]
\[
\text{department: ARRAY INT OF department\_tuple\_type;}
\]

Since, we want CVC3 to generate a dataset, i.e., to assign values to the elements of each tuple, each element of every tuple is a variable in CVC3 and every position in the tuple corresponds to an attribute in \( R \) is mapped to a position in a tuple. Tuple attributes are referenced by position, not by name; thus, \( \text{department}[2].0 \) refers to the value of the first attribute, which is \( \text{dept\_name} \), of the second tuple in \( \text{department} \).

To ensure a non-empty result for the query in Example 1, we need a tuple in \( \text{course} \) which matches a tuple in \( \text{department} \) on attribute \( \text{dept\_name} \) and where the \( \text{dept\_budget} \) > 70000. This is done by creating a tuple for each of the relations and adding the following constraints:

\[
\text{ASSERT course}[1].2 = \text{department}[1].0;
\]
\[
\text{ASSERT department}[1].2 > 70000;
\]

Primary key constraints are enforced by constraints that ensure that if two tuples match on the primary key, then the values of the remaining attributes for those two tuples should also match. Foreign key constraints are enforced by adding extra tuples that satisfy the foreign key condition. Foreign key constraints are specified as illustrated as follows, for the foreign key from \( \text{course} \), \( \text{dept\_name} \) to \( \text{department} \), \( \text{dept\_name} \):

\[
\text{ASSERT FORALL(i: course\_index):}
\]
\[
\text{EXISTS (j: dept\_index): course}[i].2 = \text{department}[j].0;
\]

where \( \text{course\_index} \) and \( \text{dept\_index} \) give the index range for the \( \text{course} \) and \( \text{department} \) arrays; \( \text{course}[1].2 \) stands for \( \text{dept\_name} \) of the ith tuple of \( \text{course} \).

In addition to the tuples generated for satisfying the query, additional tuples may be generated to satisfy foreign key constraints. In our example an extra tuple would be generated for \( \text{department} \) for each tuple in \( \text{course} \), although in this case the first tuple of \( \text{department} \) itself ensures the foreign key constraint is satisfied for the first tuple of \( \text{course} \).

The above constraints are given to CVC3 which generates satisfying values (assuming the constraints are satisfiable).

As explained earlier in this section, to kill a mutation of the inner join to right outer join, we need a value in \( \text{department}.\text{dept\_name} \) which does not match any value in \( \text{course}.\text{dept\_name} \). To do so, we replace the earlier equality constraint

\[
\text{ASSERT course}[1].2 = \text{department}[1].0;
\]

with:

\[
\text{ASSERT NOT EXISTS(i: course\_index):}
\]
\[
(\text{course}[i].2 = \text{department}[1].0);
\]

and generate the required dataset using CVC3. Datasets for killing other mutations are generated similarly.

### 2.4 Disjunctions

In [25], XData could not kill mutations in presence of disjunctions in selection conditions. We use the techniques presented by Tuya et al. in [23] for data generation and mutation killing in the presence of disjunctions.

We consider queries with disjunctions in the selection conditions, if the selection conditions are written in Conjunctive Normal Form (CNF). Selection conditions in other formats can be rewritten in CNF format. Generating data for a query with disjunctions is simple enough. For every tuple that we need to generate we add CVC3 constraints in CNF format, similar to the selection conditions.

For killing a mutation in a query, we need to ensure that the mutation of a where clause is reflected as a change at the root of the query tree. Consider the \( \text{where} \) clause \( P_1 \) or \( P_2 \), where \( P_1 \) and \( P_2 \) are conjuncts of selection conditions. If we mutate a condition in \( P_1 \), we need to ensure that \( P_2 \) is false so that the change in the condition of \( P_1 \) affects the output of the query. For example, let \( P_1 \) be \( (a > 50 \text{ AND } b = 40) \). If we mutate the first condition in \( P_1 \) to \( a < 50 \) we need to ensure that \( b = 40 \) is satisfied while \( P_1 \) is not satisfied. If \( P_2 \) is satisfied there would be no change in the output of the query. For the purpose of ensuring which conditions need to be satisfied and which conditions do not, we treat subqueries as selection conditions.
3 Queries and Mutations Considered

In this paper we consider single or multi 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 optional aggregation operations.

We remove the following assumptions made in [25]:

a) SQL queries do not contain string comparison or string like operators such as like, ilike, etc.

b) Aggregations are only present at the top of the query tree, hence they are not constrained.

c) SQL queries are single block queries with no nested subqueries.

d) NULL values are not allowed for attribute values.

e) Selection predicates are conjunctions of simple conditions of the form expr relop expr.

However we still retain the following assumptions

a) The only database constraints are unique, primary key and foreign key constraints.

b) Queries do not include numeric functions or expressions other than simple arithmetic expressions.

c) Join predicates are conjunctions of simple conditions.

d) No user defined functions are used.

e) NULL values are not allowed for attribute values.

We also consider a much larger class of mutations including missing join mutations, additional or missing group by mutations, string mutations and distinct mutations.

In general our data generation techniques may not be complete and datasets to generate some mutations may not be killed. However other datasets will be generated and many mutations will be killed. Thus our techniques should be considered best effort in general, although there are special cases, e.g. as described in [25], for which completeness can be shown.

4 Data Generation for String Constraints

SQL queries can have equality and inequality conditions on strings, and pattern matching conditions using the LIKE operator or its variants. Consider the SQL query

SELECT * from student WHERE name LIKE ‘Amol%’ AND name LIKE ‘%Pal’ AND tot_cred > 30

In order to generate the first dataset that produces a non-empty result for this query or to kill mutations of the condition tot_cred > 30 we need to generate a tuple for which attribute name satisfies the LIKE conditions ‘Amol%’ and ‘%Pal’. To generate such a value we need to solve the corresponding string constraints. For killing mutations of the LIKE operators also we need to solve similar string constraints.

Since CVC3 does not support string constraints, we need to solve the string constraints outside of CVC3. For this purpose we develop a string solver which is described in Section 4.2. We then discuss test data generation for killing mutations involving string operators in Section 4.3. Note that for this to work there should be no interplay between string and other constraints so that the string constraints can be solved independent of other constraints. In cases where this is not possible we may need multiple tries as described in Section 4.2.

4.1 Types of String Constraints Considered

For string comparisons, we consider the following class of string constraints: $S_1$ relop constant, and $S_2$ relop $S_2$, where $S_1$ and $S_2$ are string variables, and relop operators are $=, <, \leq, >, \geq, <>$ and case-insensitive equality denoted by $\sim=$. We support LIKE constraints of the form $S$ likeop pattern, where likeop is one of LIKE, ILIKE (case insensitive like), NOT LIKE and NOT ILIKE. We also support $\text{strlen}(S)$ relop constant where relop is one of $=, <, \leq, >, \geq$ or $<>$. We do not support constraints of the form $S_1$ likeop $S_2$, where both $S_1$ and $S_2$ are variables.

We support the string functions upper and lower in queries where these functions can be rewritten using one of the operators described above; for example upper($S$) = ‘ABC’ can be rewritten as $S \sim= ‘ABC’$, and similarly upper($S$) LIKE pattern can be replaced by $S$ ILIKE pattern. We rewrite these conditions as a pre-processing step. Conditions like upper($S$) = constant or upper($S$) LIKE pattern, where the pattern contains at least one lower case cannot be satisfied. Hence for such conditions we do not change the operators but return an empty dataset. If these functions are used on a constant string, we convert the string to upper or lower according to the function.

4.2 Solving String Constraints

There are several available string solvers that we considered, including Hampi [15], Kaluza [24], SUSHI [10] and Rex [30]. However we found that Hampi and Kaluza were rather slow, and while they handled regular expressions and length constraints, they could not handle constraints such as $S_1 < S_2$, where both $S_1$ and $S_2$ are variables. Rex and SUSHI, though much faster, could

1 The latest version of CVC (CVC4) provides some support for string theory but does not support operations like string comparison (CVC4 also lacks support for some CVC3 features such as subtypes which are used by XData).
not handle constraints involving multiple string variables. Hence we built our own solver which is described in Appendix A.1.

Once the values for string variables are obtained we solve the non-string constraints using CVC3 and get an overall solution as follows: enumeration types are created in CVC3 for string variables, with the enumeration names being the (suitably encoded) (strings generated by the string solver. For example, consider a query which has a single string constraint: $S_1 \text{like} ‘\text{Bio}\%’$. Let the string that satisfies the constraint be Biology, then the constraint is specified as

\[
\text{ASSERT(table[index].pos} = \text{Biology})
\]

in CVC3, where table[index].pos is the corresponding CVC3 variable of $S_1$. We then add constraints in CVC3 equating each string variable to its corresponding enumeration name, add other non-string constraints as described in Section 2 and invoke CVC3 to get a suitable dataset.

**Disjunctions With String Predicates :**
If there are disjunctions in the selection predicate, it is not possible to separate the string constraints since not all string constraints may need to be satisfied. For queries containing disjunctions (in CNF form) along with string constraints, each disjunct in the constraint is considered one at a time and the string constraints of that disjunct are passed to the string solver. If CVC3 can get a solution on incorporating the values returned by the string solver, the given solution is sufficient. If not we use the next conjunct.

4.3 Killing String Constraint Mutations

There can be different type of string mutations depending on whether the string condition is a comparison condition or a LIKE condition.

**String Comparison Mutation :**
Consider a string constraint of the form $S_1 \text{rel}op \ S_2$, where $S_1$ is a variable (attribute name), $S_2$ could be another variable or a constant. We consider mutations of relop where any occurrence of one of \{=, <>, <, >, \le, \ge\} is replaced by another. Three datasets are enough to kill all the relop mutations. These are the datasets generated for (1) $S_1 = S_2$ (2) $S_1 > S_2$ (3) $S_1 < S_2$. These datasets will also kill the mutation because of missing string selection mutations. In addition, to kill mutations between and $\sim =$, we generate an additional dataset, where $S_1 \not< S_2$, but $S_1 \not= S_2$.

**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 or the operator is missing. For a condition $S_1 \text{likeop} \ \text{pattern}$, 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 $S_1 \text{LIKE pattern}$.
**Dataset 2** satisfying condition $S_1 \text{ILIKE pattern}$, but not $S_1 \text{LIKE pattern}$, which is ensured by using the LIKE specification, with pattern modified by changing the case of one or more characters.

**Dataset 3** failing both the LIKE and ILIKE conditions, which is generated by modifying the LIKE pattern by replacing one or more characters (other than % and _) by different characters.

For e.g., for the condition $S_1 \text{LIKE ‘bio’}$, the conditions in the three cases would be (1) $S_1 \text{LIKE ‘bio’}$; (2) $S_1 \text{LIKE ‘BIO’}$; and (3) $S_1 \text{LIKE ‘CIO’}$.

The targeted mutations and the datasets that kill them are shown in Table 1.

**LIKE Pattern Mutations :**
A common error while using the LIKE operator is the specification of an incorrect pattern in the query, for e.g., specifying $S_1 \text{LIKE ‘Comp’}$ or $S_1 \text{LIKE ‘Com’}$ in place of $S_1 \text{LIKE ‘Comp’}$. There could be a very large number of such patterns to be considered. We handle mutations that involve ‘%’ in place of ‘%’ and \textit{vice versa} and also missing ‘%’ or ‘\_’.

- For killing the mutation of ‘%’ to ‘\_’ or for missing ‘%’, we generate separate datasets for each occurrence of the ‘%’ replaced with ‘\_’. The pattern with ‘%’ gives a non empty result while the mutated patterns will give empty results on the corresponding datasets.
- For killing the mutation of ‘\_’ to ‘%’ or for missing ‘\_’, we generate separate datasets for each occurrence of ‘\_’ with that occurrence of ‘\_’ removed. The original pattern gives an empty result while the mutated patterns give non empty results on the corresponding dataset.

5 Handling NULLs

In our earlier work [25], we could not handle NULLs. This is because CVC3 does not understand NULL values.

| Mutation to kill                  | Dataset |
|----------------------------------|---------|
| LIKE vs NOT LIKE                 | 1,2     |
| LIKE vs ILIKE                    | 1       |
| LIKE vs NOT ILIKE                | 2       |
| NOT LIKE vs ILIKE                | 2       |
| NOT LIKE vs NOT ILIKE            | 1       |
| ILIKE vs NOT LIKE                | 1,2     |
| Missing LIKE / ILIKE             | 3       |
| Missing NOT LIKE / NOT ILIKE     | 1       |

Table 1: Dataset required to kill like operator mutations
In this section, we discuss how we model NULLs in CVC3.

To model NULLs for string attributes, we enumerate a few more values in the enumerated type and designate them NULLs. For example, the domain of course_id is modeled as follows:

\[
\text{DATATYPE course_id = CS190 | CS632 | NULL_course_id_1 | NULL_course_id_2 END;}
\]

Here, the first two values are regular values from the domain of course_id, while the last two values are used as NULLs. For numeric values, we model NULLs as any integer in a range of negative values that are not part of the given domain of that numeric value.

Next, we define a function in CVC3 which identifies which values are NULL values and which are not. This function is syntactic sugar for dealing with NULLs cleanly, and is defined per domain to identify the NULLs in that particular domain. In addition to specifying what values are NULL, we also explicitly need to state that the other values are NOT NULL. Otherwise, CVC3 may choose treat a NON-NULL value as a NULL value. Following is an example of the function:

\[
\text{ISNULL_COURSE_ID : COURSE_ID -> BOOLEAN;}
\]

\[
\text{ASSERT NOT ISNULL_COURSE_ID(CS190);}
\]

\[
\text{ASSERT NOT ISNULL_COURSE_ID(CS632);}
\]

\[
\text{ASSERT ISNULL_COURSE_ID(NULL_crse_id_1);}
\]

\[
\text{ASSERT ISNULL_COURSE_ID(NULL_crse_id_2);}
\]

We also need to enforce another property of NULLs, namely, that nulls are not comparable. To do so, we choose 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, selection conditions involving IS NULL checks, and kill mutations of COUNT to COUNT(*)

### 5.1 Nullable Foreign Keys

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. Nullable foreign keys allow us to kill more mutants than is possible if the foreign key attribute as not nullable. (Our implementation handles multi-attribute foreign and primary keys.)

### 5.2 IS NULL / NOT IS NULL Clause

If the query contains a condition \( R.a \) IS NULL, we explicitly assign (a different) NULL to attribute \( a \) for each tuple \( R[x] \) provided the query contains only inner joins or only a single relation (provided the attribute is nullable; attributes declared as primary key or as not null cannot be assigned a NULL value).

However in case the tuple is a result of an outer join there maybe multiple ways to ensure that an attribute has a null value. Let us consider the join condition \( E1 \bowtie E2 \). If the IS NULL condition is on an attribute of \( E1 \) we need to ensure that the value of that attribute is NULL. If the IS NULL condition is on an attribute on \( E2 \) we need to ensure that either (a) that attribute is NULL (which may not be possible if \( E1 \) is a relation and the attribute is not nullable) or (b) for that tuple in \( E1 \) there does not exist any matching tuple in \( E2 \); this can be done by a minor change in the algorithm to handle NOT EXISTS subqueries as described in Section 8.1 (Algorithm 2). We omit details for brevity.

We consider mutation from IS NULL to NOT IS NULL. The first dataset (the one that generates non empty results on the original query) kills the mutation of IS NULL to NOT IS NULL if the IS NULL condition is present in the form of conjunctions with other conditions. In the presence of disjunctions we generate a dataset such that the IS NULL condition is satisfied while the conditions present in disjunction with the IS NULL condition are not satisfied. 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. We also consider the mutation where the mutant query does not contain the IS NULL condition In order to kill this mutation we generate a tuple with the IS NULL condition being replaced by NOT IS NULL (with the conditions present in conjunction with the IS NULL not being satisfied). The original query gives an empty result while the mutant gives a non-empty result.

If the query contains the condition NOT IS NULL the corresponding mutations can be killed in a similar manner.

### 5.3 NULLs and COUNT(*)

To kill the mutation from COUNT\((attr)\) to COUNT\(^{*}\), where \( attr \) is a set of attributes, we create a dataset such that all tuples in a group have \( attr \) as NULL (provided all attributes in \( attr \) are nullable and none of them is forced to be non-nullable by selection or join conditions). COUNT\((attr)\) gives a count of 0, while COUNT\(^{*}\) gives a count of equal to the total number of tuples.

In order to kill mutations of COUNT\(^{*}\) to COUNT \((attr)\), for any set of attributes \( attr \), we create a dataset such that all nullable columns (columns that can be assigned NULL values and do not have conditions that force them to be not null) have NULL values. If \( attr \) is
not nullable, COUNT(*) and COUNT(attr) are equivalent mutations.

6 Missing or Extra Joins Conditions

Consider the tables student (id, name, dept_name), course (course_id, course_name and dept_name) and takes (id, course_id, sec_id, semester, year) from the University schema in [26]. Consider the query,

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

One of the mutations of the query could be because of an additional join condition leading to a mutant query like

```
SELECT course_id, course_name
FROM student INNER JOIN takes ON(id)
INNER JOIN course ON(course_id, dept_name)
WHERE student.id = 1234
```

Such errors are common when using natural joins. For example if natural join was used in place of . . . INNER JOIN course ON(course_id) resulting in student.dept_name being equal to course.dept_name.

In order to kill such mutations, we do the following. Let the relations being joined be $R_i$ and $R_j$. For every attribute $p \in R_i$ such that (a) there is an attribute $q \in R_j$ with identical names and (b) there is no join condition involving $p$ and $q$ in the original query, we assert that the values held by the two attributes are not equal. The original query without the join condition would give a non-empty result while the mutation would give an empty result.

For the above example, we generate a dataset such as

| student | id | name | dept_name |
|---------|----|------|-----------|
|        | 1234 | Alice | EE        |

| course  | course_id | name      | dept_name |
|---------|------------|-----------|-----------|
|         | CS-317     | Database Systems | Comp. Sc. |

| takes   | id | course_id | sec_id | semester | year |
|---------|----|-----------|--------|----------|------|
|         | 1234 | CS-317     | 1      | Fall     | 2014 |

where attribute dept_name which occurs in student and course, but which are not equated in the query, is assigned different values in the two tuples. The mutated query would give an empty result for this dataset while the original query gives the result (CS-317, Database Systems).

Similarly, there could be mutants such that the mutant query contains some missing join conditions. Such mutations can be killed by the datasets that kill join type mutations (INNER / LEFT OUTER / RIGHT OUTER) described in Section 2.1. For example, if the original query contains a LEFT OUTER JOIN the query will produce an empty result on the dataset to kill the RIGHT OUTER JOIN mutation while the query with missing join conditions will produce a non-empty result.

7 Constrained Aggregation

In [25] we considered aggregates which did not have any constraints on the aggregation result e.g. via a HAVING clause, or in an enclosing SQL query of a subquery with aggregation. In this section we discuss techniques for data generation for queries which have constrained aggregation. We assume that each aggregate is on a single attribute, not on multiple attributes or expressions. We also assume that aggregation constraints do not involve disjunctions.

Consider the HAVING clause constraint, $SUM(r.a) > 20$. In case the domain of $r.a$ is restricted to $[0,5]$ it is not possible to generate a single tuple for $r$ such that the aggregation constraint is satisfied. CVC3 does not support a relation type where the number of tuples may be left unspecified. We model relations as an array of tuples and hence such aggregation constraints cannot be translated into CVC3 constraints like $SUM(r[i].a) > 20$, leaving the number of tuples in $r$ unspecified. Hence, before generating CVC3 constraints we must (a) estimate the number of tuples $n$, required to satisfy aggregation constraints, and (b) in case the input to the aggregate is a join of multiple base relations, translate this number $n$ to appropriate number of tuples for each base relation so that the join result contains exactly $n$ tuples.

In Section 7.1 we discuss how to estimate the number of tuples to satisfy an aggregation constraint. In Section 7.2 we discuss data generation for constrained aggregation on a single relation. Later, in Section 7.3.1 we describe our new method showing how the estimated number is translated to an appropriate number of tuples for each base relation, in case the the input to the aggregate is a join of two or more relations. In Section 7.3.2 we describe how to generate data satisfying the constraints.

7.1 Estimating Number of Tuples per Group

We now consider how to estimate the number of values (tuples) needed to satisfy aggregation constraints. For each attribute, $A$, on which there are aggregate constraints we collect all aggregation constraints on $A$, including domain constraints. These constraints, along with constraints (listed below) between various aggregation operators are then given to CVC3. Let constraint variables $sum_A$, $min_A$, $max_A$, $avg_A$, $count_A$ correspond to the results of aggregation operators $SUM$, $MIN$, $MAX$, $AVG$, $COUNT$ on attribute $A$. Let $dmin_A$, $dmax_A$ correspond to the minimum and maximum value in the
domain of $A$. Note that $count_A$ also indicates the number of tuples at the input to the aggregation. Then the following constraints are generated:

- Since the value of each tuple cannot be less than $min_A$ and greater than $max_A$, it follows that $min_A \cdot count_A \leq sum_A \leq max_A \cdot count_A$.
- If the domain of $A$ is integer and $A$ is unique, $\sum (min_A + 1) + ... + (max_A + count_A - 1) \leq \sum (max_A - count_A + 1) + (max_A - count_A + 2) + ... + (max_A - count_A + (count_A - 1)) + (max_A) - (avg_A \cdot count_A) = sum_A$.
- $dmin_A = Min$imum value in the domain of attribute $A$.
- $dmax_A = Max$imum value in the domain of attribute $A$.
- $dmin_A \leq min_A \leq max_A \leq dmax_A$. This constraint states that $min_A$ cannot be less than the domain minimum or greater than $max_A$.

If the query contains non-aggregate constraints on any attribute $A$, we add these to the tuple estimation constraints. For example, consider the query,

```sql
SELECT dept_name, SUM(credits) FROM course INNER JOIN dept USING (dept_name)
HAVING SUM(credits) < 13
```

Here, because credits column has a selection condition on it, its limit is constrained. Hence, $max_{\text{credits}} \leq 4$ is also added to the list of constraints above.

The solver returns a value for count which satisfies all the constraints above, but the value may not be the minimum. 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 $MAX_{\text{TUPLES}}$ and choose the smallest value of the count for which CVC3 gives a valid answer. We borrow the idea of calculating the number of tuples, using multiple tries, for the aggregation constraint from RQP [5]. However, note that the problem is different here, since, unlike RQP, we do not know the value of the aggregation in the query result.

Note that the above procedure works even in case of multiple aggregates on the same column or on different columns.

**Heuristic Extensions**

The value with which the aggregate is compared to may be a column (i.e. a variable) e.g. HAVING SUM(R.a) relop S.b. This can happen when S.b is a group by attribute or when the constrained aggregation is in a subquery and S.b is a correlation variable from an outer subquery. For such cases we replace the column name by a CVC3 variable when estimating the number of tuples. We also add the domain and selection conditions for that column as constraints on the CVC3 variable. The solver then chooses a value for the number of tuples such that the aggregate is satisfied for some value of the variable in its domain.

If the aggregation has a DISTINCT clause we add constraints to make the corresponding aggregated attribute unique.

Handling constraint aggregation in general for these cases is an area of future work.

### 7.2 Data Generation for Aggregate on a Single Relation

In case the aggregate is on a single relation the number of tuples estimated is assigned to the only relation. For each result tuple generated by an aggregation operator, we create a tuple of constraint variables where group by attributes are equated to the corresponding values in the inputs and aggregation results are replaced by arithmetic expressions for example, $\sum (r.x)$ is replaced by $R[i].x + R[i + 1].x + ... + R[i + k].x$, where $R[i]$ to $R[i+k]$ are the tuples assigned for a particular group. We also add constraints to ensure that no two tuples in $R[i]...R[i+k]$ are the same, if the relation has a primary key.

The tuple variables created as above can be used for other operations e.g. selection or join that use the aggregation result as inputs.

In case data generation for multiple groups is required we add constraints to ensure that at least one of the GROUP BY attributes is distinct across groups.

Consider the query

```sql
SELECT id, COUNT(*) FROM takes
WHERE grade = 'A+' AND year = 2010
GROUP by id HAVING COUNT(*) < 3
```

For this query the number of tuples in the group is estimated to be 1. We assign a single tuple to the takes relation and add constraints to ensure that grade for this tuple is ‘A+’ and year is 2010.

Note that if the XData system generates additional tuples for the takes relation (for example because this query is part of a subquery and there may be other instances of the takes relation outside the subquery or takes is referenced by some other relation and we need to generate additional tuples to satisfy foreign key dependencies) the value of COUNT() in the having clause may change and the constrained aggregation may no longer be satisfied. In order to ensure that the HAVING clause is not affected we need to ensure that no other tuple in the takes relation belongs to the same group.

---

2 Since, we are interested in small datasets, we set $MAX_{\text{TUPLES}}$ to 32 in our experiments.
In general to ensure that the additional tuples generated do not cause problems we add constraints to ensure that for any additional tuple either has a different value for the GROUP BY attribute and hence belongs to a different group or fails at least one of the selection conditions. In the above example we assert that either the id is different for the additional tuple or year \( \neq 2010 \).

In cases where there are no GROUP BY attributes and there are no selection conditions this technique does not work. For some special cases like \( SUM > val \) (for positive values), \( MIN < val, MIN <= val, MAX > val \) and \( MAX >= val \), where \( val \) is a constant the aggregate constraint will be satisfied irrespective of whether there are any additional tuples generated or not.

For cases like \( MIN > val, MIN >= val, MAX < val, MIN <= val, AVG > val, AVG < val \) we add constraints to ensure that any tuple that is added is \( > val, >= val, < val, <= val, > val \) and \( < val \) respectively if it belongs to same group. Handling such cases in general is an area of future work.

### 7.3 Data Generation for Aggregation on JOIN Results

In case the aggregate is on a join result we need to assign tuples to each of the relations such that the join results in the required number of tuples. In this section we address this issue.

#### 7.3.1 Estimating Number of Tuples per Relation

We assume here that all join conditions are equijoins. The required number of tuples is denoted by \( n \). We address this issue.

A naive way is to assign \( n \) tuples to a relation, \( R_i \) and assign the same value to all its joining attributes, \( \{R_i.a, R_i.b\} \). For relations joining with \( R_i \) only a single tuple is assigned and the joining attribute(s) are assigned the value to corresponding join attribute of \( R_i \). For all other relations also single tuple is assigned and the joining attributes are equated. It is easy to see that this assignment will lead to \( n \) tuples in the output. This however does not work in case the joining attribute(s) of \( R_i \) are unique (either due to primary keys or by inference from other primary keys) or multiple values are required for attributes of some other relations.

In this section we address the issue of how many tuples to assign to each relation taking into account the primary key constraints, unique constraints, group by attributes and join conditions.

We define the following types of attribute(s) that are used for assigning cardinality to relations.

1. **uniqueElements**: Sets of attributes for which no two tuples in a group can have the same value. These sets of attributes are placed in \( uniqueElements[R_i] \) where \( uniqueElements[R_i] \) contains sets of unique elements of relation \( R_i \).
2. **singleValuedAttributes**: The attribute(s) which have same value across all tuples in a group. These attributes are placed in \( singleValuedAttributes[] \).

The tuple estimation is done in 3 steps. First we construct a join graph. Then we infer attributes to be added to \( uniqueElements[] \) and \( singleValuedAttributes[] \). In the third step, we assign cardinality to each relation such that the resulting number of tuples is \( n \).

**Step 1: Construct Join Graph**

We construct a join graph \( G = (R, E) \), with each relation in the query as a vertex. The join conditions from one table to another are represented by a single edge between the nodes. Figure 1 shows a join graph involving relations \( A, B \) and \( C \). There are join conditions between \( A \) and \( B \), and between \( B \) and \( C \). However there are no join conditions between \( A \) and \( C \). Inferred join equalities are added to the graph. For example, the join conditions \( A.a = B.b \) and \( B.b = C.c \) imply that \( A.a = C.c \) is also a join condition and hence it would be added to the graph. Note that this may introduce a cycle in the graph; our algorithm can work with cyclic join graphs.

**Step 2: Infer Attribute Properties**

Next we apply the following sets of rules to infer properties of attributes

- **Rule 1**: Every group by attribute is a single valued attribute.
- **Rule 2**: Every set of attributes declared as primary key or unique key, is unique in the group.
- **Rule 3**: Every attribute which appears in conjuncts of the form \( A.a=\text{constant} \) is a single valued attribute.
- **Rule 4**: If each attribute of any \( uniqueElements[R_i] \) is a single valued attribute then all attributes of that relation are single valued attributes.
- **Rule 5**: If any attribute, \( R_i.x \), is a single valued attribute then every attribute of equivalence class (Section 2.1) in which \( R_i.x \) is present becomes a single valued attribute. For example, if the join condition is \( A.a = B.a \) and \( A.a \) is single valued, \( B.a \) also becomes single valued.
- **Rule 6**: If an attribute of a unique element is single valued then remaining attributes of unique element become unique. We apply this rule recursively on the unique element to get a minimal unique element. We then drop all non minimal sets from \( uniqueElements \). For example, if \( (A.a, A.b, A.c) \) is unique and \( A.a \) is single valued then \( (A.b, A.c) \) is unique and is added...
Algorithm 1: getAttributeInferences()

**Inputs:** joinConds[R_i, R_j]
groupByAttributes[]
unique keys and primary keys of relations ∈ query

**Output:** Effective uniqueElements, singleValueAttributes

1. Build equivalence classes using joinConds[R_i, R_j], ∀R_i, R_j ∈ R
2. Apply Rule 2 and update uniqueElements[ ]
3. Apply Rule 1, Rule 3 and update singleValueAttributes[ ]
4. while change in uniqueElements[ ] or singleValueAttributes[ ]
5. Apply Rule 4, Rule 5 and update singleValueAttributes[ ]
6. Apply Rule 6 and update uniqueElements[ ]
7. end while
8. return uniqueElements[ ], singleValueAttributes[ ]

to uniqueElements[R_i]. In this case (A.a, A.b, A.c) is dropped from uniqueElements[R_i].

The rules are applied according to the Algorithm 1 to infer which attributes are to added to uniqueElements[ ] and which to singleValueAttributes[ ].

**Step 3: Assign Cardinality**

We define some more terms

- joinAttributes[R_i, R_j]: attributes of relation R_i that are involved in join conditions with relation R_j
- unique[R_i, R_j]: \{S_k | S_k ⊆ joinAttributes(R_i) \land S_k \in uniqueElements[R_i]\}
- n_{R_i}: number of tuples assigned to relation R_i.

In order to find the number of tuples for each relation we use the attributes inferred using Algorithm 1 along with the following rules.

**Rule 7:** If n_{R_i} = n, n > 1 and unique[R_i, R_j] ≠ ∅ then n_{R_i} is set to n. We also infer further unique elements as follows. For each S_k \in unique[R_i, R_j], let S'_k be the attributes from R_j that are equated to S_k. Then add S'_k to uniqueElements[R_j].

The intuition behind Rule 7 is as follows. Consider the join of two relations A and B. Let the join condition be A.a = B.a and suppose that \{A.a\} ∈ uniqueElements[A]. Here joinAttributes[A, B] = \{A.a\}, joinAttributes[B, A] = \{B.a\}, unique[A, B] = \{A.a\} and unique[B, A] = ∅. If the cardinality of A is n, since A.a is unique, it must have n different values. The relation B has join condition with A.a which belongs to uniqueElements[A].

So B must contain n tuples with distinct values for the attribute B.a across n tuples and each value matches with the value of A.a for one of the tuples in R_i. So the cardinality of B become n and B.a becomes a unique attribute.
- constraints ascertaining `singleValuedAttributes` and `uniqueElements` for each relation
- for each relation such that all attributes are single valued (Rule 4) constraints to ensure that the number of tuples is 1
- constraints for Rule 7 and the Implementation Rule 1 for all the relations in the query as applicable
- constraints to ensure that the final count after joining the tables is \( n \)
- in case \( n \) values are required for some attribute \( R.a \) to satisfy some aggregate condition we add constraints to ensure that the relation \( R \) has \( n \) tuples. For example consider a case where \( \text{SUM}(R.a) = 17 \) where \( a \) is an integer attribute and there is a constraint \( R.a \leq 5 \), we need at least 4 tuples for the given group of \( R \) and they cannot all be the same. Here it is not possible to satisfy the aggregation condition if we assign a single tuple to \( R \) even if the join of \( R \) with other relations produces 4 tuples for the group. Similar is the case with \( \text{SUM DISTINCT} \) on an integer attribute.

On solving this set of constraints we get the number of tuples for each relation.

The constraint approach for tuple generation works well if the number of attributes is not very large. In practice we use a simple and fast heuristic approach described as follows. If any non empty set of attributes of a relation form a unique element and every attribute of that unique element is a single valued attribute then that relation must contain a single tuple (explained in Rule 4). For such relations the only possible choice of cardinality is 1. Of the remaining relations the heuristic algorithm chooses one relation and assign to it a cardinality of \( n \), making it the root node. The count of all other nodes of the join graph, \( n_{R_i} \) is initialized as 1. The root node \( R_s \) is then used as a starting relation to calculate the actual cardinality for other relations using Rule 7 and Implementation Rule 1. The procedure for this is described Algorithm A2 of Appendix B. If the heuristic fails we use the constraint approach.

### 7.3.2 Data Generation

After getting the tuple assignment for each relation we add CVC3 constraints to fix the number of tuples in a group to the value that is computed. For each join condition, constraints are generated depending on the above, for example if both relations \( R \) and \( S \) have \( n \) tuples, the constraint \( R[i].x = S[i].y \) is generated for all \( 1 \leq i \leq n \), while if \( R \) has \( n \) tuples and \( S \) has 1 tuple, the constraint \( R[i].x = S[1].y \) is generated for all \( 1 \leq i \leq n \). Constraints variables for the output of the aggregate operator are created as described earlier in Section 7.2. One difference is in handing aggregation for relations that have been assigned one tuple. For example, \( \text{SUM}(r.x) \) is replaced by \( R[i].x + R[i+1].x + \ldots + R[i+k].x \), where \( R[i] \) to \( R[i+k] \) are the tuples assigned for a particular group, if \( R \) has \( n \) tuples, otherwise it is replaced by \( n \times R[i].x \), where \( R[i] \) is the only tuple assigned for a group. Unique constraints are added as pairwise non equality constraints to ensure that sets of `uniqueElements` have distinct values.

In addition to the tuples generated for constrained aggregation we have to ensure there are no extra tuples in the group as described in Section 7.2. However the solution in Section 7.2 was for the case without joins. With joins, there could be additional tuples in other relations that join with the relations with the GROUP BY attributes and produce extra tuples.

Consider the query,

```sql
SELECT ... FROM R1..Rn
WHERE joinConds AND selConds
GROUP BY gbAttrs HAVING aggConds
```

In order to prevent the additional tuples from altering satisfaction of the aggregate condition we ensure that any additional tuples that need to be generated for the tables do not satisfy the conditions `selConds` and `joinConds`. Constraints for this are generated in similar manner as obtaining constraints for `NOT EXISTS` condition described in Section 8.1 (Algorithm 2) restricted to results containing at least one additionally generated tuple. We omit details for brevity. In the above example we assert constraints for all additional tuples such that the conjunct of join and selection conditions are not satisfied.

Data generation for multiple groups is done by adding constraints to ensure that at least one of the group by attributes is distinct across groups.

The constraints are then given as input to CVC3, and output of CVC3 gives us the required dataset.

#### 7.3.3 Discussion

Our tuple assignment techniques always assign either 1 tuple or \( n \) tuples to a relation. There could be cases where such an assignment is not possible and a different assignment is required to generate datasets. However in such an assignment it becomes difficult to assert constraints such that the join of the relations will generate exactly the required number of tuples. Handling tuple assignment if either 1 or \( n \) tuples cannot be assigned to all the relations to satisfy the aggregation constraint is an area of future work.

### 7.4 Constraint Aggregation and Mutant Killing

For constrained aggregation, we consider mutations between `MIN` and `MAX`, `SUM` and `SUM DISTINCT`, `AVG`
We now consider test data generation for SQL queries involving subqueries. Data generation for subqueries in the from clause is discussed in Section 12; in this section we consider data generation and mutation killing for subqueries in the where clause. We initially assume in Section 8.1 that subqueries do not have aggregations. Subqueries with aggregation are discussed in Section 8.2.

### 8 Where Clause Subqueries

We now consider test data generation for SQL queries involving subqueries. Data generation for subqueries in the from clause is discussed in Section 12: in this section we consider data generation and mutation killing for subqueries in the where clause. We initially assume in Section 8.1 that subqueries do not have aggregations. Subqueries with aggregation are discussed in Section 8.2.

#### 8.1 Data Generation for Subqueries Without Aggregation

**EXISTS Connective**

Consider a query Q with a nested subquery predicate EXISTS(SQ). To generate a non-empty result for Q we need to ensure that SQ gives a non-empty result. If SQ does not have any correlation variables we treat subquery SQ as a query in itself and add constraints to generate a non-empty dataset for the subquery using our data generation techniques. We then add constraints for Q for predicates other than the subquery. The dataset is then generated based on these constraints.

If SQ has correlation conditions then for every tuple that is generated for Q we call a function to generate the constraints for data generation of the subquery with the correlation conditions passed as parameters. The correlation conditions are treated as a selection in SQ with the given constraints variables and appropriate constraints are generated for SQ. For example, consider the query

```
SELECT course_id, title
FROM course INNER JOIN section USING(course_id)
WHERE year = 2010 AND EXISTS (SELECT * FROM prereq
    WHERE prereq_id = 'CS-201' AND prereq.course_id = course.course_id)
```

To generate a dataset for the outer query we generate a single tuple for each of the course and section relations. Let the tuples be `course[1]` and `section[1]`. We then add constraints to assert `section[1].year=2010` and `course[1].course_id = section[1].course_id`. We pass the correlation variable `course[1].course_id` as a parameter to the function for generating constraints for the subquery. For this tuple in the outer query block we generate a tuple in prereq relation, say prereq[1], for which we add constraints to ensure that `prereq[1].prereq_id = 'CS-201'` and `prereq[1].course_id = course[1].course_id`.

**NOT EXISTS Connective**

Consider a query Q with a nested subquery predicate NOT EXISTS(SQ). Here we need to ensure that the number of tuples from SQ is 0. Let us first consider the case where SQ has no correlation variables. If SQ has only a single relation, R we add constraints to ensure that every tuple in R fails at least one of the selection conditions. In case SQ has a join of two or more relations we traverse the tree of SQ and in a recursive manner add constraints on selections and joins so as to ensure that no tuples reach the root of SQ. In the join is an INNER JOIN we need to ensure that there exists no pair of tuples for which the join conditions are satisfied. In case the join is LEFT OUTER JOIN we need to ensure that there are no tuples in the left subtree. Similarly in case of RIGHT OUTER JOIN we need to ensure that no tuple is projected from the right subquery. Details are provided in Algorithm 2.

If SQ has correlation variables, correlation variables are treated in the same manner as EXISTS subquery and passed as parameters and the correlation conditions can be treated as selections and Algorithm 2 can be used.

**IN/NOT IN Connective**

We convert subqueries of the IN type to EXISTS type subquery by adding the IN connective as a correlation condition in the WHERE clause. The same techniques as that of EXISTS is then used. Similarly subqueries using a NOT IN connective are converted to use the NOT EXISTS connective. For example

```
r.a NOT IN (SELECT s.b FROM .. WHERE ..)
```

is converted to

```
NOT EXISTS
(SELECT s.b FROM .. WHERE .. AND r.a = s.b)
```

**ALL/ANY Connective**

Subqueries with ALL and ANY connectives, always appear with one of the comparison operators, for example “< ALL”, or “>= ANY”. We transform subqueries of the from `rel`< operator to an EXISTS query with `rel` condition as a correlation condition in the WHERE clause. Subqueries with `rel` ALL are transformed to a NOT EXISTS query with a negation of the `rel` condition as a correlation condition in the WHERE clause. For example
Algorithm 2 : genConstraintsForNotExists
Inputs: $T$ = the subquery tree
Output: constraints to ensure no tuple is projected from the subquery

1: constraints ← ""
2: $R$ ← $T$.ROOT
3: if $R$ is a relation then
4: Let the selection conditions on $R$ be $S_1, S_2, ..., S_n$
5: constraints ← NOT($S_1$ AND $S_2$ ... AND $S_n$) (for every tuple in $R$)
6: else if $R$ is an aggregate then
7: genConstraintsForNotExists($R$.CHILD)
8: else if $R$ is a LEFT OUTER JOIN then
9: constraints ← genConstraintsForNotExists($R$.LEFT)
10: else if $R$ is a RIGHT OUTER JOIN then
11: constraints ← genConstraintsForNotExists($R$.RIGHT)
12: else if $R$ is an INNER JOIN then
13: $JC$={}
14: Let the join conditions at $R$ be $J_1, J_2, ..., J_e$
15: Let the number of tuples in $R_1$ be $m$ and in $R_2$ be $n$
16: Let $J_k(i,j)$ denote the condition corresponding to the $k$th join condition for tuples $R_1[i]$ and $R_2[j]$
17: for $i$ in $1$ to $m$, $j$ in $1$ to $n$ do
18: $J[|i||j||j||i||j]$ ← NOT($J_1(i,j)$ AND $J_2(i,j)$ AND $..J_c(i,j)$)
19: end for
20: constraints← $JC[1][1]$ AND $JC[1][2]$ AND $..JC[m][n]$, $\forall 1 \leq i \leq m, 1 \leq j \leq n$
21: constraints ← constraints + “OR” + (genConstraintsForNotExists($R$.LEFT)) + “OR” + (genConstraintsForNotExists($R$.RIGHT))
22: end if
23: return constraints

r.a > ALL (SELECT s.b FROM .. WHERE .. )
is converted to
NOT EXISTS
(SELECT s.b FROM .. WHERE .. AND r.a <= s.b)

Scalar Subqueries:
Scalar subqueries are subqueries that return only a single result. We consider scalar subqueries in the WHERE clause which are used in conditions on the form $SSQ \ relop \ attr/value$, where $SSQ$ is a scalar subquery, $attr$ is an attribute from the outer block of query and $value$ is a constant. For such subquery we generate only a single tuple for the query and assert that the projected attribute satisfies the comparison operator. Correlation conditions, if any, are treated in the same manner as subqueries with the EXISTS connective.

8.2 Data Generation for Subqueries With Aggregation
In this section we consider subqueries that have aggregation. Constraints can be in the inner query (e.g. HAVING clause) or in outer query (e.g. r.s < (select agg(s.b...)))

Non Scalar Subqueries:
The techniques in Section 8.1 can be applied for EXISTS subqueries without constrained aggregation, since we only need to ensure empty / non empty results for the subquery. For NOT EXISTS Algorithm 2 covers the case of aggregate operators as well.

In case of constrained aggregation in EXISTS subquery we use the techniques described in Section 7 to generate tuples for the subquery; multiple tuples may be generated. In case there is a constrained aggregation in the NOT EXISTS subquery we assert constraints to ensure that either the constraint aggregation is not satisfied or there are no tuples input to the aggregation constraint.

Subqueries of the IN/NOT IN/ALL/ANY type having an aggregate as the projected attribute can be transformed into EXISTS/NOT EXISTS in a similar manner as shown in Section 8.1. In this case the projected aggregate is added as a HAVING clause. For example
r.a NOT IN (SELECT agg(s.b) FROM .. WHERE .. )
is converted to
NOT EXISTS (SELECT agg(s.b) FROM .. WHERE .. HAVING agg(s.b) = r.a)
The techniques for constrained aggregation in EXISTS/NOT EXISTS can then be applied.

Scalar Subqueries:
Consider the query involving the table takes(id, course_id, sec_id, semester, year, grades),

SELECT id FROM takes
WHERE grade < (SELECT MIN(grade)
FROM takes WHERE year = 2010)

To generate datasets for this query we add constraints to generate a tuple. $takes[1]$, for the $takes$ table in the outer query. The tuple estimation technique estimates that one tuple is required for $takes$ relation to satisfy the comparison operator corresponding to the subquery ($< MIN(grade)$). We add constraints to generate one more tuple, say $takes[2]$ for takes relation corresponding to the subquery and add constraint to ensure that $takes[2].year = 2010$ for that tuple. We then add constraint $takes[1].grade < takes[2].grade$ to ensure that the grade of the outer query tuple is greater than the grade of the subquery tuple. Since $takes[1]$ does not participate in aggregation we need to ensure that it does not satisfy the conditions of the subquery block (Section 7.3.2). For this the constraints $takes[1].year < 2010$ is added.

In general consider a query of the form
SELECT * FROM rel1 JOIN .. WHERE cond1 AND ...
AND attr relop (SELECT agg(sqrel1.attr2)
FROM sqrel1 JOIN ... WHERE sqcond1 AND ..)
For such subqueries we need to ensure that the aggregate, \( \text{agg}(\text{sqrel1.attr2}) \), satisfies the condition \( \text{attr1} \text{ relop} \text{ agg}(\text{sqrel1.attr2}) \). In order to do this we may need to project multiple number of tuples from the subquery. We use the techniques described in Section 7 to estimate the number of tuples, assign the desired number of tuples to each relation and generate constraints for data generation.

Similar to EXISTS subquery in the presence of correlation conditions we generate one group of tuples in the subquery for every tuple in the outer query.

8.3 Killing Subquery Connective Mutations

**EXISTS/NOT EXISTS, IN/NOT IN Mutation**: The dataset generated for the original query will kill the mutation between IN and NOT IN, and between EXISTS and NOT EXISTS if the subquery condition is present in the form of conjunctions with other conditions. In the presence of disjunctions we generate a dataset such that the subquery condition is satisfied and conditions in disjunction with it are not. The EXISTS clause give an empty result when NOT EXISTS gives a non-empty result, and vice versa. Similar is the case for IN versus NOT IN.

**Comparison Operator Mutation**: For conditions of the form “\( r.A \text{ relop} (SQ) \)” where SQ is a scalar subquery, as well as conditions of “\( r.A \text{ relop} \{\text{ALL}/\text{ANY}\} \text{ SQ} \)”, we consider mutations between the different relop s. Similar to the approach shown in Section 2.1 we generate data for three cases, with relop replaced by >, = and <.

**ANY/ALL Mutation**: This mutations involves changing from ANY to ALL or vice versa. Since the ANY subquery has been transformed to EXISTS this mutation from ANY to ALL becomes a double mutation - replacing EXISTS with NOT EXISTS and negating the correlation condition corresponding to the ANY comparison condition. The case for ALL to ANY mutation is similar.

Let the correlation condition added because of transformation of ALL/ANY to EXISTS/NOT EXISTS be \( R_1.A \text{ relop} R_2.b \). We generate a dataset with two tuples in the subquery for every tuple in the outer query. We add constraints for relop for one tuple and the negation of relop for the other tuple. The ANY query will produce a non-empty result since one of while the ALL query will produce an empty result since not all of the three conditions can be satisfied for any tuple in the outer query.

**Missing Subquery Mutation**: To kill the mutation of a query with missing EXISTS condition connective we generate a dataset with the EXISTS condition replaced by NOT EXISTS. If the EXISTS condition is missing the mutant query will give a non empty answer while the original query will give an empty answer. Similarly for killing mutations with missing subquery connectives in other cases we replace NOT EXISTS with EXISTS, IN with NOT IN and NOT IN with IN.

The datasets generated to kill comparison operator mutation will also kill mutations involving missing scalar/ALL/ANY subqueries. If the subquery is present the original query will give an empty result on at least one of these datasets while the mutant query will produce non-empty result on all of the three datasets.

8.4 Killing Mutations in a Subquery

We also need to generate test data for killing mutations in subquery blocks. For the EXISTS connective and for scalar subqueries we treat a subquery block as a normal query and generate sets of constraints to kill mutations in the subquery block. For each constraint set we also add constraints to ensure a non-empty result on the outer query block.

For killing selection, JOIN, NULL, and HAVING clause mutations the techniques described in [25] and this paper generate datasets that produce empty result on either the query or the mutant but not both. Thus for these mutations the subquery will satisfy the EXISTS condition or the comparison operator (for scalar subqueries) for either the subquery or its mutation enabling XData to kill the mutation. In case there are disjunctions with the subquery we add constraints to ensure that other conditions in disjunction with the subquery (e.g. \( \text{P or EXISTS(Q)} \)) are not satisfied as described in Section 2.4. Mutations like distinct or aggregation mutation in the project clause of the subquery create equivalent mutants of the query and hence need not be killed.

If the subquery uses the NOT EXISTS connective, we generate the datasets for killing mutations in the subquery treating the NOT EXISTS as an EXISTS conditions. Out of the original query and the mutant, the query that produces empty results on the subquery satisfies the NOT EXISTS conditions and produces non-empty results for the outer query. The query that does not produce empty results does not satisfy the NOT EXISTS condition and produces an empty result in the outer query. Thus these mutations can be killed.

Subquery connectives ANY and ALL are converted to EXISTS and NOT EXISTS as described earlier. Mutations in the subquery is killed after the conversion.
9 Group By Clause Mutations

In this section, we discuss the mutation of the query due to presence of additional attributes or absence of
some attributes in the group by clause.

9.1 Additional Group By Mutation

Consider the following query, $Q$, to find the number of
students taking each course every time it is offered.

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

Additional attributes included in the group by clause such as `section` as shown in the query, $Q_s$, below, could result in an erroneous query.

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

To catch such mutations, we generate a dataset for
each possible additional group by attribute, with more
than one tuple in the group, such that the additional
attribute (`section` in this case) has different values for
different tuples in the group. This ensures that the in-
correct query produces multiple groups while the correct
one produces only a single group, thereby killing the
mutation. Note that some because of selection conditions
resulting in attributes being single valued, functional
dependencies on group by attributes and equality con-
ditions on group by attributes some of the mutations
with additional GROUP BY may be equivalent to the
original query. We do not consider such attributes.

There are situations where the above approach would
not work e.g. if the group by is in an EXISTS subquery. If
there is no constrained aggregation the mutation would
be equivalent but if there are aggregation constraints
the mutation may not be equivalent and needs to be killed.

If the group has an aggregation that is constrained,
e.g., $SUM(a) > 20$ or $SUM(b) \leq 30$ the number of
tuples is assigned based on the aggregation constraint.
We try to ensure that the data generated is such that
the aggregation constraints of one of the queries, i.e.,
either of the original query or of its mutant are satisfied,
resulting in a non-empty result on either the original
query or its mutation but not both, hence killing the
mutation.

For each possible additional group by attribute, $g_i$
we generate up to 2 corresponding datasets. In our
first attempt, we try to generate two separate groups,
which agree on all other group by attributes (other than $g_i$),
such that each group satisfies the aggregation
constraints, but the group containing the union of these
tuples (i.e., group by other than $g_i$) does not. Note that
this may not be possible in case the aggregate is of the
form $SUM(x) > number$ for values in positive range or
$COUNT(x) > number$ etc. Hence, we also try to generate
a dataset such that the combined group satisfies the
aggregate but the individual groups do not. If either
succeeds, the mutation will be killed.

9.2 Missing Group By Mutation

Another common error is to miss specifying some of
the group by attributes. For example, if one misses
specifying the attribute `semester` in the GROUP BY
clause but query $Q$ then the resultant query is clearly
erroneous. Such erroneous queries can be easily detected,
if the number of attributes projected out is different.

However that may not be the case for all queries
where a group by attribute has been missed. For in-
stance, in the above example, if `semester` was not in the
projection list, the missing group by mutation would
be harder to catch. Although rare, we have found such
cases when the group by is in a subquery whose result
is an aggregation tuple.

We generate datasets to kill such mutations as fol-
loows: Let $g_1, g_2, \ldots, g_n$ be the group by attributes. For
missing group by attribute, $g_i$, we treat the original
query as the one with the missing group by attribute
and its mutation with additional group by attribute as
the original query. Datasets can be generated using the
techniques for killing mutations of additional group by
attributes.

10 Select Distinct Mutations

Users may erroneously omit the DISTINCT keyword
in the projection list of a select clause. For example,
consider the following query from [26], that finds the
department names of all instructors.

```sql
SELECT DISTINCT dept_name
FROM instructor
```

In this query the absence of DISTINCT keyword
would lead to the same department being tuples which
is not desired. We term mutations that add or delete the
DISTINCT keyword to the select as distinct mutations
(DISTINCT of aggregates is covered in Section 2.1). To
to kill such mutations we need a dataset which results
in two tuples in the output such that these tuples are
identical on the projected attributes. For such a dataset,
the query with the DISTINCT keyword will give only a
single tuple as output while the one without, will give
two tuples.

If the query is on only one relation, we generate two
tuples for the relation and ensure that all attributes in
projection list of the query are identical in these two
tuples. We do this by an additional constraint that for all
the attributes in the projection list, the value in the first tuple is same as second. If primary key attributes are present in the projection list, the mutation is equivalent to the original query and hence no dataset is generated.

For the above query, the following dataset is generated:

| id    | name   | dept, name | salary |
|-------|--------|------------|--------|
| 12345 | Alice  | Comp. Sc.  | 70000  |
| 12346 | Bob    | Comp. Sc.  | 80000  |

If the query has multiple relations or uses a from clause subquery we need to ensure that there are two tuples passed to the distinct operator so that the mutation is killed. For this we set the number of tuples to be two. Then using the techniques described in Section 7.3.1 for constrained aggregation we assign number of tuples to each base relation. When using those techniques we treat the projected attributes in the select clause as the group by attributes in constrained aggregation, which ensures that these have the same value across tuples.

11.1 Data generation

Set queries are of the form,

\[ P \text{ SETOP } Q \]

where \text{SETOP} is a set operator, and P and Q are queries that may be simple or compound queries themselves.

In order to generate a dataset that produces a non-empty result on this query if the \text{SETOP} is UNION(ALL) we add constraints to ensure non-empty results for P or Q or both (P and Q may have conflicting constraints so for both to have non-empty results may not always be possible).

Data generation for INTERSECT(ALL) is done in a similar manner as the EXISTS subquery described in Section 8.1. We treat the query as

\[ \text{SELECT * FROM (P) WHERE EXISTS (SELECT * FROM Q WHERE pred)} \]

where predicate \text{pred} equates each projected attribute of P to the corresponding attribute of Q. For each tuple in P we generate a corresponding tuple in Q that satisfies the correlation condition, \text{pred}, as described in Section 8.1. Data generation for EXCEPT(ALL) is done in a similar manner using NOT EXISTS instead of EXISTS, using the techniques described earlier for the NOT EXISTS operator.

11.2 Killing Set Operator Mutations

In order to kill the mutations among the different operators (UNION(ALL), INTERSECT(ALL), EXCEPT(ALL)) we generate a dataset as follows. For generating the dataset we set the number of tuples for P to be 4 and the number of tuples for Q to be 2. In case there are multiple relations in P or Q we use the approach described in Section 7.3.1 to assign tuples to individual relations. Next we assert that the first two tuples from P and the two tuples from Q all identical, while the 3rd and 4th tuples of P are identical to each other but different from the first two tuples of P.

For example suppose that the result of P and Q have only 1 attribute each and let the tuples generated for P= \{(a),(a),(b),(b)\} and the tuples generated for Q=\{(a),(a)\}. The results of queries with different set operators on this dataset are shown in Table 2 and it is clear that mutations between different set operators are killed.

| Mutant        | Result      |
|---------------|-------------|
| UNION         | a,b         |
| UNION ALL     | a,a,a,a,b,b |
| EXCEPT        | b           |
| EXCEPT ALL    | b,b         |
| INTERSECT     | a           |
| INTERSECT ALL | a,a         |

Table 2: Mutants output

11.3 Killing Mutations in Input to Set Operators

We also need to kill mutations in the input to the set operator. For this we need to ensure that the effect of the mutation makes a difference in result of the set operator.

If the set operator is UNION/UNION ALL and the mutation to the query is in P we add constraints to ensure that the mutation in P is killed. In addition to ensure that there are no tuples from Q that mask the changes in the result we add constraints similar to NOT EXISTS subquery for Q. Similarly data generation can be done for killing mutations in Q.

We treat INTERSECT and EXCEPT queries as EXISTS and NOT EXISTS respectively as described earlier. Mutations of P can be killed by datasets to kill mutations of the outer query block while the mutations in Q can be killing by killing mutations in the subquery block as described in Section 8.4.
12 Other Extensions

From clause subqueries: Our parser turns from clause subqueries into a tree which can be handled using our existing techniques. We do not handle from clause subqueries that project aggregates, if there are constraints on the aggregation result in the enclosing query (other than simple constraints which our techniques handle) or if the query uses the lateral construct. Handling such queries is an area of future work.

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.

Floating and Fixed Point Numbers: CVC3 allows real numbers to be represented as (arbitrary precision) rationals and hence when populating a real type data (floating or fixed precision) from the database or query, we represent it as a fraction in CVC3. Fixed precision values with only a few digits after the decimal point can cause problems in rare cases, since two rationals generated by CVC3 which are very close to each other may map to the same fixed precision number.

BETWEEN operator: We convert queries involving BETWEEN clause, say attr BETWEEN a AND b, we convert the BETWEEN clause to the clause attr > a AND attr < b. The datasets for killing selection mutations are also able to catch mutations where the user intended the range to include a or b or both.

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.

When testing queries in an application for correctness, we execute the original INSERT, DELETE or UPDATE queries against the generated datasets. To test student queries against a given correct query, we perform the transformation from INSERT, DELETE and UPDATE queries to SELECT queries as above for both the given student queries and the given correct queries, before comparing them as described in Section 13.

Handling WITH Clause and Views: We syntactically convert a query using a WITH clause or views by performing view expansion. The assumptions we make about the query structure must be satisfied by the resultant expanded query.

ORDER BY clause: ORDER BY clause mutations include missing ORDER BY clause or attributes, additional ORDER BY clause or attributes, using ORDER BY DESC instead of ORDER BY and vice versa. In the absence of any ORDER BY clause the order of tuples is determined by the query plan. However it is possible for a query without an order by clause or with an incomplete order by clause, to give a result in the same order as a correct query, depending upon the chosen plan. Thus order by mutations in general cannot be caught by comparing results on different datasets, although we can use such comparison as a heuristic. Mutations between ORDER BY and ORDER BY DESC can however be caught by generating appropriate datasets. To kill such a mutation we generate a dataset having two distinct values for the order by attributes.

As an alternative to checking results on generated datasets, mutations involving missing or additional ORDER BY clause or attributes can be detected by checking the ORDER BY clauses in the query. However care should be taken to handle equivalent ORDER BY clauses due to functional dependencies, equality predicates between variables, and equality selection conditions.

13 Grading Student SQL Queries

In [4] we describe the XDa-TA grading tool which uses datasets generated by the techniques presented in this paper for checking the correctness of student SQL queries. Here we describe how to efficiently check student queries against given correct queries. For each query in an assignment, a correct SQL query is given to the tool, for which it generates datasets for killing mutants of that query. To check if a student query is correct the results of the student and correct query are compared on each dataset.

It is to be noted that we do not aim to prove query equivalence of student query and correct query. Query equivalence between two queries $Q_1$ and $Q_2$ can be
proven if we are able to prove that \( Q_1 \) is contained in \( Q_2 \) and \( w \) vice versa\(^3\). Thus query equivalence can be modeled in terms of query containment. Under set-semantics it can be shown that the problem of query containment is NP-complete for conjunctive queries [7], and \( \Pi_i \) complete for queries involving inequalities [16,17]. For bag semantics the complexity of query containment is undecidable for conjunctive queries with inequalities [13]. We tried a sufficient condition for query equivalence, namely that both \( Q_i \) and \( Q_{i,j} \) generate the same optimal query plan, but as results in Section 15.5 show, this approach is often unable to establish equivalence of correct queries.

Thus we only aim to catch common errors and it is possible that a non-equivalent student query may be marked correct. However, in case we mark a student query as incorrect we have a dataset on which the student query and the correct query gives different results and hence guarantee that the queries are not equivalent.

The instructor needs to upload the schema and optionally small sample tables, by providing SQL script files. The instructor can then add assignment questions in text and correct queries for the same. For each correct query the tool then generates datasets, using the techniques of the XData system. Each dataset is tagged with a label indicating what kind of mutation the dataset was designed to kill. Student queries are submitted directly by the tool or can be uploaded in bulk.

For some assignments it may be possible to write correct queries using several very different approaches. Datasets generated for a correct query are designed to be used to kill mutations of that query, but may or may not succeed in killing mutations of a different formulation of the query. It could also happen that the question in text set by the instructor was ambiguous and there are multiple ways of interpreting it. For these cases the instructor mode allows multiple correct queries to be uploaded. Datasets generated from all the correct queries are used while evaluating student queries. The instructor may set whether datasets of all the queries are used while evaluating student queries. The tool executes an SQL query of the form

\[
(Q_{i,j} \text{ EXCEPT ALL } CQ_{i,m}) \text{ UNION } (Q_{i,m} \text{ EXCEPT ALL } Q_{i,j})
\]

on the temporary tables.

If the result of the above query is non-empty for any dataset \( D_{i,m,k} \), the student query \( Q_{i,j} \) is marked as incorrect. If the results of the above query are empty for all datasets, query \( Q_{i,j} \) is deemed correct for the purpose of grading. The instructor can also decide that the presence of duplicates do not matter and in this case the tool uses EXCEPT instead of EXCEPT ALL in the query above.

An assignment can be marked as a learning assignment or a graded assignment. When the tool is used in student mode, for graded assignments, the tool accepts queries from the student and saves the queries for later grading. Grading can be initiated from the admin mode by the instructor. For learning assignments, the system executes the queries and displays which datasets the query fails on (this can be done incrementally, one failed dataset at a time). Tagging datasets with the type of mutation that the dataset was intended to kill, as mentioned earlier, helps student understand what the mistake was.

Our approach for checking correctness of query relies on killing the mutations of the correct query and not of the student query. As a result we may not catch erroneous student queries that have extra selection conditions. We do catch extra join conditions if the column names are identical but may miss other extra join conditions also. Consider a query condition \( x > 3 \). We generate datasets for satisfying \( x > 3, x = 3 \) and \( x < 3 \). These datasets will catch the mutations in involving change in operator, change in the constant and also in case the condition is missing. However if the student query contains \( x > 3 \) \( \text{AND} \ x < 100 \), it may be marked as correct since we may not have any test case to test mutation of \( x < 100 \). One way to deal with this is to generate datasets based on mutations of the student query as well and use these also in grading. Since this requires a lot of overhead including constraint generation, constraint solving etc. for all the student queries, we do not implement this currently.

14 Related Work

The AGENDA toolset can generate test data for an application, given as input the database schema, the application source code and certain sample value files.
The data generated is however query agnostic, and does not address killing of query mutations. Reverse Query Processing (RQP) [5] takes as input a query $Q$ and a result $O$, and generates input data $I$ such that $O = Q(I)$, the result of $Q$ on input $I$. However, neither RQP, nor its extension [6], attempts to kill query mutations. Qex [29,31] is a tool for generating a dataset and parameter values for a given parameterized SQL query using the SMT solver Z3. The goal is to generate data so that the query has a non-empty result. This corresponds to the generation of the first dataset in our case. However, Qex does not address killing of query mutations.

Tuya et al. [28] describe a number of possible mutations for SQL queries. They however do not handle test data generation for killing these mutations. They divide the mutations into four classes: mutations of the main SQL clauses (SC), mutations of the operators that are present in conditions and expressions (OR), mutations related to the handling of NULL values (NL), and replacement of identifiers: column references, constants and parameters (IR). We generate dataset for all of SC, OR and NL mutations except for the following: mutations related to arithmetic expressions, some mutations of LIKE patterns, mutations between AND and OR, and some mutations related to three-valued logic. Currently we do not consider IR mutations. Handling the above mutations is an area of future work. However, we do consider some mutations that are not covered in [28] such as join type mutations on alternative join orders and mutations of the LIKE operator.

Riva et al. [23] introduce rules which they call SQL Full predicate coverage (SQLFpc) rules, which specify conditions that must be satisfied by test cases in order to kill each of a variety of SQL query mutations; further rules to handle a larger class of SQL constructs and mutations are described in their Web tool [1]. However, they do not describe how to actually generate data. [27] extends [23] by generating constraints based on SQLFpc and solving the constraints using a constraint solver called Alloy [12]. However, it considers data generation and mutation killing for only numeric selection conditions and joins. Queries involving strings, aggregation, subqueries, and mutation killing for only numeric selection conditions and joins are not handled.

Pan et al. [21] describe Mutagen which, given a database application, first generates program code mutants and SQL-query mutants by transforming constraints from SQL queries to program code, and then uses PexMutator [32] to generate data to kill the mutants. However, they only handle mutations of conditions in the where clause; as far as we can tell from their brief description, the class of mutations they consider is very small, and in particular, they do not handle aggregation, subqueries, join type mutations set operators, distinct mutations and a number of other query features and mutations that we consider.

The work in this paper extends our earlier work on XData [25,8,4]; details of the differences and novel contributions of this paper were described earlier in Section 2.

Olston et al. [19] 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.

There have been a number of papers for testing database applications. However these do not address the problem of testing queries in the applications. Emmi et al. [9] and Pan et al. [20,22] describe approaches to test applications based on creation of database states and test inputs, which can ensure code coverage. Kaphammer and Sofka [14] similarly consider test adequacy of database driven applications.

## 15 Experimental Results

We implemented the techniques for data generation described in this paper, as extensions to the XData system. (Some aspects for data generation for queries involving NOT EXISTS and set operator queries are currently being implemented.)

We show that our string solver is more efficient than the other string solvers and can successfully generate string values for a number of test cases (Section 15.1). We also show that our techniques for constrained aggregation (Section 15.2) and subqueries (Section 15.3) are able to generate non-empty datasets and kill mutations in a number of cases. In Section 15.4 we show that our techniques are capable of generating datasets and killing mutations for the queries in the TPC-H benchmark. In Section 15.5 we evaluate our grading tool and show that it is better at catching student query errors than fixed datasets or correction by TAs.

Each of the techniques we describe targets a different query construct or mutation and hence it does not make sense to compare the different techniques that we have proposed with each other.

### 15.1 String Solver

Section 4 describes our techniques for data generation for queries with string constraints. To deal with such constraints we have implemented a string solver. The experiment in this section focus on the performance of our string solver as compared to other solvers in terms of the time taken to solve constraints. The experiments for HAMPI [15], Kaluza [24], CVC4 [2], SUSHI [10] and
In this experiment we study the efficiency of the string solvers in a variety of common cases. The test cases for this experiment are listed in Appendix C. We include a mix of satisfiable and unsatisfiable cases. The last 3 test cases contain multiple string variables and can only be solved with our string solver. We include these cases to show that the performance does not drop much even when solving for multiple variables. For solvers other than the XData string solver the expressions in form of A `like`op/relop expr etc. were manually converted to regular expressions of the format recognized by the solvers. The running time does not take into account the conversion. For XData string solvers we fed constraints in the same form as in the SQL queries and let XData convert these to regular expressions.

The time taken by different string solvers for this experiment are shown in Table 3. The test cases that cannot be solved by a particular solver\(^4\) is marked with a “-” and cases that ran for a very long time (>20 min) but still did not terminate are marked with a “*”. In terms of time taken CVC4 and the XData solver turn out to the most efficient ones for these cases, but CVC4 cannot handle multiple variables.

More results on comparison of performance of the various string solvers is given in Appendix A.

### 15.2 Constrained Aggregation

In Section 7.3.1 we described our approach for estimating the number of tuples in case of data generation for queries containing constrained aggregation. In this section we provide experimental results on estimation of number of tuples per relation and subsequent data generation for a number of queries containing constrained aggregation. The objective here is to see if the our tuple assignment assign tuples in a manner that can produce datasets to generate to non-empty result on the original query (this is the first dataset as mentioned in Section 2) and kill mutations related to the HAVING clause (aggregate mutation and comparison operator mutation of the HAVING clause).

For this experiment queries which involve constraints on aggregate operators along with one or more GROUP BY attributes were chosen. (The list of queries is provided in Appendix C.) Aggregates in both outer query block and subqueries are considered. We also manually generated non-equivalent mutations by mutating the comparison operator (20 mutations) and aggregate operator (16 mutations) for the chosen queries to test if the datasets could kill these mutations\(^5\).

The results are are shown in Table 4. For each constrained aggregation the Tuples column shows the number of tuples assigned to each base relation. The columns Comparison Mutations and Aggregate Mutations show if the non equivalent mutations of comparison operator in the constrained aggregation and aggregate mutation respectively were killed by the generated datasets or not. Query CA8 had two constrained aggregation one in a subquery and one in the outer query block which are labeled as CA8a and CA8b respectively.

The datasets generated by XData was able to produce non-empty results on all queries. In terms of killing mutations 35 out of the 36 mutations were killed. The mutation from MAX to MIN was not caught for Test Case CA2. For killing mutation on MAX to MIN we need two distinct tuples, one which satisfies the aggregate constraint and one which does not. Our tuple assignment method assigned only one tuple to the relation which had the MAX aggregate value and hence this mutation was not caught. Handling such cases is an area of future work.

### 15.3 Subqueries

In Section 8 we describes various techniques for generating test data and killing mutations for queries containing where clause subqueries. For this experiment we chose queries involving various subquery connectives both with and without aggregates (The list of all queries is provided in Appendix C) and check to see if XData is able

\(^4\) HAMPI currently has a known bug because of which it cannot handle more than one constraints on the same variable in some cases. Test cases 4, 5 and 7 failed because of this.

\(^5\) We do not use any automated tool to generate mutations. The mutations generated by an automated tool may or may not be equivalent to the original query. If our dataset fails to kill some of the mutations we would not be sure if that was because of the incompleteness of our tool or because of equivalence of mutation and the original query.
Table 4: Tuple estimation for constrained aggregation

| Test Case | Tuples | Comparison Mutations | Aggregate Mutations |
|-----------|--------|----------------------|---------------------|
| CA1       | 1,2    | ✓                    | ✓                   |
| CA2       | 1,1,2  | ✓                    | ×                   |
| CA3       | 2,1,2  | ✓                    | ✓                   |
| CA4       | 3,3,1,3| ✓                    | ✓                   |
| CA5       | 1,2,2  | ✓                    | ✓                   |
| CA6       | 1,2,1,2,2| ✓                  | ✓                   |
| CA7       | 1,1,3  | ✓                    | ✓                   |
| CA8a      | 1,2,2  | ✓                    | ✓                   |
| CA8b      | 1,2,2  | ✓                    | ✓                   |
| CA9       | 5,5    | ✓                    | ✓                   |

Table 5: Number of mutants caught on TPC-H queries

| Mutation Type                   | Mutants Generated | Mutants Killed |
|---------------------------------|-------------------|---------------|
| Selection (Comparison)          | 32                | 31            |
| Join Type (INNER / OUTER)       | 15                | 15            |
| Aggregation (Distinct/ MIN vs MAX) | 19              | 17            |
| String Selection (String Comparison) | 14              | 14            |
| String Like                     | 12                | 12            |
| Missing Joins Conditions        | 16                | 15            |
| Having Clause (Comparison Operator) | 2               | 2             |
| Subquery Connective             | 8                 | 8             |
| Changed Group By                | 18                | 16            |
| Arithmetic Operator             | 27                | 24            |
| AND vs OR                       | 25                | 23            |
| **Total**                       | **188**           | **177**       |

Overall XData was able to kill over 90% of the non-equivalent mutants that we generated. The aggregate mutants that XData missed were because the aggregate was on an expression. The expression evaluated to 0 on the dataset to kill aggregate mutation and hence the result of SUM and SUM DISTINCT were the same. XData was not able to catch one of missing join and selection mutation each because of the presence of disjunctions with join conditions which we currently do not handle. For two cases XData could not generate datasets for killing extra group by attributes and hence these were not caught. Extending to system to handle these cases is an area of future work.

15.4 TPC-H queries

We also tried generating test data and killing mutations for queries from the TPC-H benchmark. We manually generated a number of non-equivalent mutations for these queries and tested them to see if the datasets generated could kill the mutations or not. Since our parser did not support certain query constructs we made minor changes (mainly syntactic) to the queries so that it could be parsed and the datasets could be generated. However for checking whether the datasets generated a non-empty result or not and for generation and killing of mutations we used the original queries.

We were able to successfully generate datasets for 16 out of the 22 queries. Of the 6 queries for which our techniques failed to generate correct datasets, 5 queries had query constructs which are not currently handled (subqueries that have aggregates with expressions, aggregate value compared to a subquery, aggregate in a from clause subquery and repeated relations in NOT EXISTS). One query failed because our solver (CVC3) crashed while generating datasets for that query.

The number of the different types of mutations killed across all queries is shown in Table 5. In addition to the mutations described in the paper we also considered mutation of the arithmetic expressions (replacing one arithmetic operator with another) and the AND operator to OR and vice versa.

15.5 Grading

We use the tool described in Section 13 to grade student queries. In order to compare grading done by XData to fixed datasets and the grading done by TAs, we used 14 SQL assignments, each of which were answered by students of an undergraduate database course at IIT Bombay. We omit questions which asked students to create DDL statements.

For each question, a correct SQL query \(CQ_i\) was used to generate datasets. The correct SQL queries are shown in Appendix C. For the 9th assignment question, the query could be written in 2 quite different ways which we denote \(CQ_9a\) and \(CQ_9b\); we generate datasets for both query formulations, and the results for \(CQ_9\) are using the combined sets of datasets. Query \(CQ_3\) was assigned at a point in the course where students had not been taught about the DISTINCT clause, and hence we set the testing tool parameters to ignore duplicates in the results of the correct query and the student query.
The time taken for generating all the datasets for these queries (including the time taken by our code and the CVC3 solver) ranged from 11 to 90 seconds, on a computer with an Intel(R) Core(TM) i5-2500K 3.30GHz CPU, and 16 GB of memory, running Ubuntu.

As comparison points, we also tested the queries with two sample University databases provided with the textbook by Silberschatz et al. [26], and with the result of manual correction by course TAs. The first University database, which we call USm is a small database which was manually created by the authors of [26] to catch common errors; the second larger database, which we call ULg 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.

We also tried an alternate way to grade student queries, by comparing the optimal query plans of the correct query with the optimal query plans for the student queries. If the plans match we flag the query as correct. We use PostgreSQL with the VERBOSE flag set to ensure that we get projected attributes of the query as well. We label this method of grading as Plan. Note that equivalent queries may not have identical plans. For example a condition $x > 3$ is equivalent to $x = 2$ when $x$ is an integer, but plans using these alternatives would be considered different. Also the optimizer could find different plans for different ways of expressing the same query (especially true with subqueries). In our experiments we found that most of the student queries did not have the same plan as the correct query, even if they were correct (verified manually on sample cases). For CQ3 the optimizer chose different join plans and hence a missing distinct keyword in the select clause was marked incorrect. For CQ14 the optimizer chose different join plans and hence most of the queries did not match. Same was the case with CQ7, CQ8, CQ13 and CQ14.

The results of the evaluations are given in Table 6. For XData, USm and ULg when the query is marked as incorrect if there is a dataset that produces different results on the correct query and the student query. Hence for these methods we can guarantee that a student query is marked incorrect only when it is not equivalent to the correct query. Consequently the number of queries marked incorrect can be used as a measure of effectiveness of the technique. Wherever our technique and/or some of the datasets find more incorrect queries than others, we have highlighted the results in bold.

As compared to TAs, our datasets performed significantly better on many queries, including CQ3, CQ4, CQ5, CQ6, CQ8 and CQ9. 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. On CQ14, however, TAs caught more errors that our datasets.

For CQ9a a large number of incorrect queries were caught by XData based on missing group by attributes and missing distinct clause. For CQ9b, again the University datasets USm and ULg both had some courses with two sections, which caught missing distinct specifications; in this case the TAs did check for the presence of the distinct specification.

For CQ4, the University dataset did not have any course taught by two different instructors in Spring 2010, and hence a missing distinct keyword in the select clause was not detected. The TAs did not enforce the check for distinct, which was required for this query. In contrast, for CQ4, the University dataset USm had a student who took CS-101 twice, and hence performed as well as XData. Again, the TAs had ignored the absence of a distinct specification. For CQ5, again the University datasets USm and ULg both had some courses with two sections, which caught missing distinct specifications; in this case the TAs did check for the presence of the distinct specification.

For CQ11a a large number of incorrect queries were caught by XData based on missing group by attributes and missing distinct clause. For CQ11, the data generation and mutation killing technique for disjunctions and NOT EXISTS, which are introduced in this paper, were

Table 6: Query grading results

As students had been told that their queries would be graded by a tool, they would probably taken more care to avoid such errors.
essential for catching a large number of student query errors.

Discussion:
In order to get a measure of our accuracy or completeness of our techniques on these queries we need an oracle to identify which queries are correct and which are not. This is very difficult for complex queries and doing this for classes with many students is extremely time consuming. The closest option is human evaluation, however our tool in its current version outperforms TAs (indicating TAs are not infallible). Hence it is difficult for us to provide any completeness results for our grading tool.

16 Conclusion
In this paper we have addressed the issue of testing SQL queries and automated testing of SQL student assignments. We used the XData system which we built earlier, to generate test datasets for detecting errors, and realized that there were several limitations that needed to be addressed. We described several novel extensions to address these limitations. We also tested the efficacy of our test generation techniques for grading SQL queries submitted by students, and showed that our techniques outperform fixed (query independent) datasets, as well as TAs in terms of catching errors, while avoiding the drudgery of manual correction. Our XData system has great potential for easing the life of database application developers and testers and also to database course instructors particularly to those of MOOCs.

We have successfully used the grading tool in a UG database course at IIT Bombay to correct student queries. We hope to release it soon (after doing some code cleanup) so that it can be used by course instructors for grading queries.

Areas of future work include handling some SQL features which we do not currently support, or support only partially, and handling further classes of mutations. These features include handling subqueries within subquery, arithmetic expressions and mutations involving replacement of identifiers. Another area of future work is to award partial marks to student queries in a way that reflects how close the student query is to some correct query.

Acknowledgements
We would like to thank Tata Consultancy Services(TCS), India for partially funding this project. We would also like to thank Bharath Radhakrishnan and Ankit Shah for their help in some running some experiments.

References
1. SQLFpc - Generation of Full Predicate Coverage Rules for SQL database queries (MCDC for SQL). Http://in2test.uniuevo.es/SQLFpc/SQLFpcWeb.aspx
2. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: Cvc4. In: Proceedings of the 23rd International Conference on Computer Aided Verification, CAV’11, pp. 171–177. Springer-Verlag, Berlin, Heidelberg (2011)
3. Barrett, C., Tinelli, C.: CVC3. In: Computer Aided Verification (CAV), pp. 298–302 (2007)
4. Bhangadiya, A., Chandra, B., Kar, B., Radhakrishnan, B., Reddy, K.V.M., Shah, S., Sudarshan, S.: The XDa-TA system for automated grading of SQL query assignments (to appear). In: ICDE (2015)
5. Binnig, C., Kossmann, D., Lo, E.: Reverse query processing. In: ICDE, pp. 506–515 (2007)
6. Binnig, C., Kossmann, D., Lo, E.: Multi-RQP: generating test databases for the functional testing of OLTP applications. In: DBTest, p. 5 (2008)
7. Chandra, A.K., Merlin, P.M.: Optimal implementation of conjunctive queries in relational data bases. In: STOC, pp. 77–90 (1977)
8. Chandra, B., Chawda, B., Shah, S., Sudarshan, S., Shah, A.: Extending XData to kill SQL query mutants in the wild. In: Proceedings of the Sixth International Workshop on Testing Database Systems, DBTest ’13, pp. 2:1–2:6 (2013)
9. Emmi, M., Majumdar, R., Sen, K.: Dynamic test input generation for database applications. In: Int’l Symp. on Software Testing and Analysis, pp. 151–162 (2007)
10. Fu, X., Powell, M., Banteegi, M., Li, C.C.: Simple linear string constraints. Formal Aspects of Computing pp. 847–891 (2013)
11. Gupta, B.P., Vira, D., Sudarshan, S.: X-data: Generating test data for killing SQL mutants. In: ICDE (2010)
12. Jackson, D.: Alloy: A new technology for software modelling. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Lecture Notes in Computer Science, vol. 2280, pp. 20–20 (2002)
13. Jayram, T.S., Kolaitis, P.G., Vee, E.: The containment problem for real conjunctive queries with inequalities. In: Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS ’06, pp. 80–89 (2006)
14. Kapfhammer, G.M., Soffa, M.L.: A family of test adequacy criteria for database-driven applications. SIGSOFT Softw. Eng. Notes pp. 98–107 (2003)
15. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.; HAMPI: a solver for string constraints. In: Intl. Symp. on Software Testing and Analysis, pp. 105–116. ACM, New York, NY, USA (2009)
16. Klug, A.: On conjunctive queries containing inequalities. J. ACM (1988)
17. van der Meyden, R.: The complexity of querying indefinite data about linearly ordered domains. In: PODS, pp. 331–345 (1992)
18. Miller, A.: Automaton pakage dk.brics.automaton. Http://www.brics.dk/automaton/
19. Olston, C., Chopra, S., Srivastava, U.: Generating example data for dataflow programs. In: SIGMOD Conference, pp. 245–256 (2009)
20. Pan, K., Wu, X., Xie, T.: Database state generation via dynamic symbolic execution for coverage criteria. In: Proceedings of the Fourth International Workshop on Testing Database Systems, DBTest ’11, pp. 4:1–4:6 (2011)
21. Pan, K., Wu, X., Xie, T.: Automatic test generation for mutation testing on database applications. In: Automation of Software Test (AST), 2013 8th International Workshop on, pp. 111–117 (2013)
22. Pan, K., Wu, X., Xie, T.: Guided test generation for database applications via synthesized database interactions. ACM Transactions on Software Engineering and Methodology (TOSEM) (2014)
23. de la Riva, C., Suárez-Cabal, M.J., Tuya, J.: Constraint-based test database generation for SQL queries. In: Workshop on Automation of Software Test, AST ’10, pp. 67–74 (2010)
24. Saxena, P., Akhawe, D., McCamant, S., Song, D.: KALUZA. http://webblaze.cs.berkeley.edu/2010/kaluza/
25. Shah, S., Sudarshan, S., Kajbaje, S., Patidar, S., Gupta, B.P., Vira, D.: Generating test data for killing SQL mutants: A constraint-based approach. In: ICDE (2011)
26. Silberschatz, A., Korth, H.F., Sudarshan, S.: Database System Concepts, 6th edn. McGraw Hill (2010)
27. Tuya, J., Cabal, M.J.S., de la Riva, C.: Full predicate coverage for testing SQL database queries. Softw. Test., Verif. Reliab. pp. 237–288 (2010)
28. Tuya, J., Suarez-Cabal, M.J., de la Riva, C.: Mutating database queries. Information and Software Technology pp. 398–417 (2007)
29. Veanes, M., Grigorenko, P., de Halleux, P., Tillmann, N.: Symbolic query exploration. In: ICFEM, pp. 49–68 (2009)
30. Veanes, M., de Halleux, P., Tillmann, N.: Rex: Symbolic regular expression explorer. In: ICST, pp. 498–507 (2010)
31. Veanes, M., Tillmann, N., de Halleux, J.: Qex: Symbolic SQL Query Explorer. In: LPAR, pp. 425–446 (2010)
32. Zhang, L., Xie, T., Zhang, L., Tillmann, N., De Halleux, J., Mei, H.: Test generation via dynamic symbolic execution for mutation testing. In: Software Maintenance (ICSM), 2010 IEEE International Conference on, pp. 1–10 (2010)
A Solving String Constraints

In this we describe our techniques to solve string constraints. We also show some more experimental results comparing our string solver to other available string solvers.

A.1 String Solver

In this section we describe the working of our string solver. To illustrate our method we use the following set of constraints as an example

Example 2

A > B
A like 'pqr%'
B ilike '_abc'
C >= B
C = 'Biology'
E = E
E like '%abc%'
F >= B
G like 'Bio%'

In this example for the purpose of simplicity of representation we consider that the strings may take only alphabetical values.

Our solver works as follows.

Step 1: Collect Conditions.

From all the constraints required for generating a dataset for the query, in the first step, we separate and collect the string constraints, i.e., selection conditions on strings, like conditions, and string length conditions.

Step 2: Reduce Number of Conditions.

Next, we reduce the number of string constraints by removing the conditions containing the equality operator as follows:

a) For each condition of the kind \( S_i = const_i \), where \( S_i \) is a string variable and \( const_i \) is a constant, we replace all occurrences of \( S_i \) with \( const_i \). This may lead to constraints of the form \( const, relop const_j \) or \( const, likeop pattern \). Using string operations, we then verify if such constraints are satisfiable. If they are satisfiable then we remove the equality conditions else we infer that there is no possible solution to the given set of conditions.

For example, if the conditions are \( A = 'Comp' \) and \( A \ LIKE \ 'Bio%' \), replacing the value of \( A \) as ‘Comp’ in the latter condition leads to an unsatisfiable constraint.

b) For constraints of the form \( S_i = S_j \), we replace all occurrences of \( S_i \) by \( S_j \) in all constraints and remove the constraint \( S_i = S_j \) from the set. When an instance of \( S_j \) has been found after solving rest of the constraints, we assign the same value to \( S_i \).

In Example 2 we assign \( C = 'Biology' \) and replace all occurrences of \( C \) with this value. Replacing \( C \), in \( C>=B \)

we get 'Biology'>=B. We rewrite this as \( B<= 'Biology' \).

Since \( A=E \) is a constraint we replace all occurrences of \( A \) by \( E \). After this step the constraints are

\[
E > B \\
E \ like \ 'pqr%' \\
E \ ilike \ '_abc' \\
B <= 'Biology' \\
E \ like \ '%abc%' \\
F >= B \\
G \ like \ 'Bio%'
\]

Step 3: Group Related Variables.

Next, we group variables that depend on each other, i.e., if \( S_i \ relop S_j \) or \( S_i \ likeop S_j \) is present in the set of constraints then \( S_i \) and \( S_j \) are in the same group. Once these groups are formed, we then solve the constraints for one group at a time. This grouping of variables helps in reducing the number of constraints that need to be solved at a time. In the above example \( E, B \) and \( F \) are dependent on one another and are hence grouped together in a group while \( G \) is put in another group.

For each group we construct a graph, where the variables form the vertices. Let vertex \( V_i \) represent the string variable \( S_i \). A constraint of the form \( S_i < S_j \) or \( S_i \leq S_j \) is represented by a directed edge from \( V_i \) to \( V_j \) in the graph. Constraint \( S_i < S_j \) is represented by an undirected edge between \( V_i \) and \( V_j \).

The graph for our example case would look like the one shown in Fig. A1 where the dotted edge between \( F \) and \( B \) implies \( \leq \) and the edge between \( E \) and \( B \) implies \(<\).

Additionally, for each string variable, \( S_i \), we store the following information.

- \( MaxLength \): The maximum allowable length of the string. It is initially assigned a default value. This value is modified based on string length constraints on \( S_i \), if any.

- \( MinLength \): The minimum allowable length of the string. Similar to the \( MaxLength \) this also has a default value and is modified based on length constraints.

- \( NotEqualLengths \): This is a set of values of length values not allowed for \( S_i \). This captures constraints of the kind \( strlen(S_i) <> constant \).

Fig. A1: Dependency among string variables
We traverse the graph and first collect all vertices whose outdegree is 0. These vertices represent the string variable $S_i$ mentioned below, for a vertex, say $V_i$, (and hence string variable $S_i$), for each vertex $V_j$ (string $S_j$) that has an edge to $V_i$ in the graph, we add appropriate constraints, using the solution of $S_i$ to the list of constraints for $S_j$. We then remove $V_i$ from the graph and solve for the remaining vertices by repeating this step on the modified graph.

We now describe the subroutine $SolveOneVariable$ for finding the solution for one vertex, $V_i$ which represents the string variable $S_i$. This subroutine consists of two parts a) building an automaton and b) finding the lexicographically smallest string on this automaton that satisfies all the constraints.

Step 4a: Building an automaton: We first convert the constraints of the form $S_i \text{ relop constant}$ and $S_i \text{ likeop}$ pattern to $S_i$ matches re where re is the corresponding regular expression in Java. This conversion is made by functions written specifically for each LIKE and comparison operator, as illustrated by the following examples of conversion:

- $S \geq 'Bio' \rightarrow \^[C-z]\w*|B\^[j-z]\w*|Bi\^[p-z]\w*|Bio\w*$
- $S \text{ LIKE} 'Bio\w*$
- $S \text{ LIKE} 'Bio,}' \rightarrow \text{ 'Bio}\w*$
- $S \text{ ILIKE} 'Bio\w*$

($\w*$ denotes a wild character)

We build an automaton, $A$ for the identity pattern (\w*). Then for every constraint that must be satisfied by $S_i$, we create another automaton, $B$ and modify the automaton $A := A \cap B$. We use a slightly modified version of the automaton package dk.bricks.automaton [18] operation on automata. We use our own methods for converting a given Java compatible regular expression to an automaton. If the number of constraints on a variable is above a certain threshold we minimize the automaton resulting from $A \cap B$ at each step so as to improve the performance.

Step 4b: Finding the lexicographically smallest string: Once we have the minimized automaton, $A$, for a variable, $S_i$, we find the lexicographically smallest possible string within MaxLength and MinLength for that $S_i$. To find such a string, we use a backtracking approach which traverses the automaton graph in a depth first manner. At each step we check if (a) the current depth $>$ MinLength and $<=$ MaxLength, (b) the state is a final state, (c) the current depth is not present in NotEqualLengths. If these conditions are satisfied then we return the string obtained by the traversal. If these conditions are not satisfied then even after reaching the dept of MaxLength, we backtrack. If after traversing the entire graph, we do not find a string that satisfies the conditions then we return a null value. Details of this algorithm are provided in Algorithm A1.

For our example, an automaton is created for $B$ using the constraints on B i.e $B < 'Biology'$. The smallest possible value for B is found to be 'A'. We then add the constraint $E > 'A'$ to E and $F > 'A'$ to F and remove B from E.Less and F.LessEqual. Now the remaining variables E and F do not have any dependency on each other and can be solved in any order. We create appropriate automata for both the variables and find suitable values using Algorithm A1. Now in order to satisfy the condition $A = E$ after solving the variables B, E and F.
we put the value of A the same as the one obtained for E.

Constraints containing "<>" and "~=":
We handle conditions of the kind \( S_i \sim S_j \) and \( S_i <> S_j \), where both \( S_i \) and \( S_j \) are string variables, such that one of \( S_i \) and \( S_j \) is unconstrained, i.e., there are no other string constraints constraining the value of one of them. For such cases, we first find an assignment to the constrained variable and then assign a value of other variable that satisfies the <> or ~= constraint as applicable.

A.2 String Solver Performance

We now show two experiments to compare the scalability of the solvers. Scalability can be measured in terms of length of string that can be successfully solved by the solver or by the number of simultaneous constraints it can handle. The setup for these experiments is the same as that used in Section 15.1.

For the first experiment we use the experimental benchmark from Rex [30] to measure the performance as the length of the string required in the output varies. The constraint to be satisfied by the string is that it must match intersection of regular expressions
\[
\omega \ast ([a-c] \ast a[a-c] \{n+1\}) \ast \omega
\]
and
\[
\omega \ast ([a-c] \ast b[a-c] \{n\}) \ast \omega
\]
\( n \) is a parameter which we varied from 0 to 1000. The results for this experiment are shown in Fig A2.

CVC4 and HAMPI failed to generate any result any value of \( n \) and hence could not be included. KALUZA gave the result as UNSAT (cannot be satisfied) for \( n > 6 \) while SUSHI ran out of memory for \( n > 13 \). Rex and XData solver were able to successfully generate string till \( n = 1000 \). In terms of efficiency the XData solver turned out to be the most efficient for most cases.

For the second experiment we measure the performance in terms of time taken to solve varying number of constraints. For each \( n \) the constraint to be satisfied is that the string must match the intersection of regular expressions
\[
\omega \ast ([a-c] \ast b[a-c] \{i\}) \ast \omega, \forall i, 0 \leq i \leq n
\]
We varied \( n \) from 0 to 15. The results are shown in Fig A3.

Here again CVC4 and HAMPI failed to generate any result any value of \( n > 0 \) and hence could not be included. KALUZA gave UNSAT result for \( n > 7 \). SUSHI and Rex ran out of memory for \( n > 9 \) and \( n > 6 \) respectively. In this experiment too the XData solver turned out to be the most efficient and did not run out of memory even at \( n = 15 \).

B Heuristic Algorithm for Tuple Assignment

In case of constrained aggregation the heuristic algorithm for assigning number of tuples to each relation is described in Algorithm A2 below.
Algorithm A2: getActualCardinalityHeuristic()

Inputs: $G = (R, E)$; Join graph
- singleValuedAttributes[]
- uniqueElements[]
- $R_c$: relation chosen as root node (cardinality can be $n$)

Output: assigned cardinality $n_{R_c}$, $\forall R_i \in R$

1: $\forall R_i \in R$, initialize $n_{R_i} \leftarrow 1$
2: $nQueue = \emptyset$
3: $n_{R_c} = n$
4: $nQueue.enqueue(R_c)$
5: while $nQueue.enqueue()$ do
6: for each edge $E_k \in E$ from $R_i$ to $R_j$ do
7: prevCardinality $\leftarrow n_{R_i}$
8: Apply Rule 7 from $R_i$ to $R_j$,
9: if change in uniqueElements[$R_j$] or
   (prevCardinality $= 1$ and $n_{R_j} = n$) then
10: $nQueue.enqueue(R_j)$
11: end if
12: end for
13: if Implementation Rule is applicable on $R_i$ then
14: Apply Implementation Rule 1 on $R_i$, let $R_k$ be the relation for which cardinality if to be changed to $n$
15: prevCardinality $\leftarrow n_{R_k}$
16: $n_{R_k} \leftarrow n$
17: if change in uniqueElements[$R_k$] or
   prevCardinality $= 1$ then
18: $nQueue.enqueue(R_k)$
19: end if
20: if change in uniqueElements[$R_i$] on Rule 8 then
21: $nQueue.enqueue(R_i)$
22: end if
23: end if
24: end while
25: return $n_{R_c}$, $\forall R_i \in R$

C Test Cases For Experiments

In this section we list the root test cases that were used for the experiments described in Section 15.

C.1 Test cases for String solver

The test cases that were used for the purpose of comparison of efficiency of various string solvers are listed in Table A1.

C.2 Test Queries For Constrained Aggregation

For the experiment involving constrained aggregation we used the following set of queries:

CA1: SELECT c.dept_name, SUM(c.credits)
FROM course c INNER JOIN department d
ON (c.dept_name = d.dept_name)
GROUP BY c.dept_name
HAVING SUM(c.credits) > 10 AND COUNT(c.credits) > 1

CA2: SELECT c.dept_name, SUM(i.salary)
FROM course c INNER JOIN department d
ON (c.dept_name = d.dept_name)
INNER JOIN instructor i
ON (c.dept_name = i.dept_name)
GROUP BY c.dept_name
HAVING SUM(i.salary) > 100000
AND MAX(i.salary) < 75000

CA3: SELECT c.dept_name, SUM(d.budget)
FROM course c INNER JOIN department d
ON (c.dept_name = d.dept_name)
INNER JOIN teaches t
ON (c.course_id = t.course_id)
GROUP BY c.dept_name
HAVING SUM(d.budget) > 100000 AND COUNT(d.budget) > 1

CA4: SELECT c.dept_name, AVG(i.salary)
FROM course c INNER JOIN department d
ON (c.dept_name = d.dept_name)
INNER JOIN teaches t
ON (c.course_id = t.course_id)
INNER JOIN instructor i
ON (d.dept_name = i.dept_name)
GROUP BY c.dept_name
HAVING AVG(i.salary) > 50000 AND COUNT(i.salary) = 3

CA5: SELECT t.semester, SUM(c.credits)
FROM course c INNER JOIN department d
ON (c.dept_name = d.dept_name)
INNER JOIN teaches t
ON (c.course_id = t.course_id)
GROUP BY t.semester
HAVING AVG(c.credits) > 2 AND COUNT(d.building) = 2

CA6: SELECT id
FROM course NATURAL JOIN department
NATURAL JOIN student NATURAL JOIN takes
NATURAL JOIN section
GROUP BY id, dept_name HAVING COUNT(dept_name) > 1

CA7: SELECT distinct dept_name
FROM course WHERE credits =
(SELECT MAX(credits)
FROM course NATURAL JOIN department
WHERE title='CS')
GROUP BY dept_name HAVING COUNT(course_id) > 2

CA8: SELECT id, name FROM
(SELECT id, time_slot_id, year, semester
FROM takes NATURAL JOIN section
GROUP BY id, time_slot_id, year, semester

| Test Case | Constraints |
|-----------|-------------|
| S1        | A like 'Comp_' |
| S2        | A like 'Mr%'  |
| S3        | A like '%sr%' |
| S4        | A like 'Comp%', A like '%Sc' |
| S5        | A like 'Comp%', A like ':Sc' |
| S6        | A > 'Bio'    |
| S7        | A like '%Sc', A like 'Life%', A.length > 6 |
| S8        | A < B, B like 'Bio', A like 'CSE%' |
| S9        | A<B, B like 'Bio%', B.length>4, A like '%101' |
| S10       | A > B, A like '%pq%', B like 'abc', C >= B, C = 'Biology', A = E, E like '%abc%', F >= B, G like 'Bio' |

Table A1: String solver test cases
HAVING COUNT(time_slot_id)>1)
as s NATURAL JOIN student
GROUP BY id, name
HAVING COUNT(id)>1

CA9: SELECT SUM(T) as su FROM
(SELECT year as T
FROM teaches NATURAL JOIN instructor
GROUP BY year, course_id HAVING COUNT(id)>4)
as temp
GROUP BY T

C.3 Test Queries For Subquery

For the experiment involving subqueries we used the following set of queries:

SQ1: SELECT * FROM department d
WHERE d.dept_name IN (SELECT c.dept_name
FROM course c
WHERE c.credits > 2)

SQ2: SELECT * FROM course c
WHERE EXISTS (SELECT * FROM department d
WHERE d.dept_name = c.dept_name)

SQ3: SELECT course_id, title FROM course
FROM course NATURAL JOIN section
WHERE semester = 'Spring' AND year = 2010
AND course_id NOT IN (SELECT course_id FROM prereq)

SQ4: SELECT DISTINCT course_id, title FROM course
WHERE credits > 3

SQ5: SELECT course_id, COUNT(DISTINCT id)
FROM course NATURAL LEFT OUTER JOIN takes
GROUP BY course_id

CQ1: SELECT course_id, title FROM course
WHERE dept_name = 'Comp. Sci.'

CQ2: SELECT course_id, title FROM course
WHERE dept_name = 'History'

C.4 Correct Queries For Grading Tool

Following are the correct queries that were used in the experiment to grade student queries:

CQ1: SELECT course_id, title FROM course

CQ2: SELECT course_id, title FROM course
WHERE dept_name = 'Comp. Sci.'

CQ3: SELECT DISTINCT course_id, title, id
FROM course NATURAL JOIN teaches
WHERE teaches.semester = 'Spring'
AND teaches.year = '2010'

CQ4: SELECT DISTINCT student.id, student.name
FROM takes NATURAL JOIN instructor
WHERE course_id = 'CS-101'

CQ5: SELECT DISTINCT course.dept_name
FROM course NATURAL JOIN section
WHERE section.semester = 'Spring'
AND section.year = '2010'

CQ6: SELECT course_id, title FROM course
WHERE credits IS NULL

CQ7: SELECT course_id, COUNT(DISTINCT id)
FROM course NATURAL LEFT OUTER JOIN takes
GROUP BY course_id

CQ8: 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)

CQ9: a) 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)

b) 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.year = B.year
AND A.course_id <> B.course_id
AND A.semester = B.semester
AND A.time_slot_id = B.time_slot_id

CQ10: SELECT DISTINCT dept_name FROM course
WHERE credits IS NULL

CQ11: SELECT DISTINCT instructor.id, name, course_id
FROM instructor LEFT OUTER JOIN TEACHES
ON instructor.id = teaches.id

CQ12: SELECT student.id, student.name FROM student
WHERE lower(student.name) like '%sr%'

CQ13: 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' AND course_id IS NULL
WHERE course_id IS NOT NULL)

CQ14: 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)