Test Data Generation for Complex SQL Queries

SUNANDA SOMWASE*, Indian Institute of Technology Bombay, India
PARISMITA DAS†, Indian Institute of Technology Bombay, India
S. SUDARSHAN, Indian Institute of Technology Bombay, India

Generation of sample data for testing SQL queries has been an important task for many years, with applications such as testing of SQL queries used for data analytics and in application software, as well as grading of student SQL queries. More recently, with the increasing use of text-to-SQL systems, test data is key for the validation of generated queries. Earlier work on test data generation handled basic single-block SQL queries, as well as single-level nested SQL queries, but could not handle more complex queries.

In this paper, we present a novel architecture and associated techniques for test generation that are designed to handle complex queries. We show our approach significantly outperforms the prior work on test data generation in the handling of complex queries. We also show that our approach outperforms the state-of-the-art for the more restricted problem of showing non-equivalence of query pairs.

CCS Concepts: • **Information systems → Data management systems**; **Relational database query languages**.

Additional Key Words and Phrases: SQL Query Testing, Test Data Generation, SMT Solver

ACM Reference Format:
Sunanda Somwase, Parismita Das, and S. Sudarshan. 2025. Test Data Generation for Complex SQL Queries. *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 367 (December 2025), 26 pages. https://doi.org/10.1145/3769832

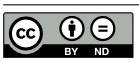
1 Introduction

Numerous software applications utilize SQL to store, manage, and query data stored in databases. Queries used for data analytics and reporting can be highly complex, and errors may be introduced during their creation. As a result, testing the correctness of SQL queries is a critical task for both data analysts and application developers. Unlike imperative programs, which typically have both a specification and an implementation that can be compared for correctness, SQL queries are often written directly from their specification. This limits the applicability of traditional verification techniques.

Testing is therefore typically done by creating appropriate test databases, and checking that the query returns the expected result on each one of them [5, 7, 21]. Such testing is often performed by application testers, who play a crucial role in large-scale application development projects. Regression testing of database applications typically involves executing the application on appropriate test database instances [10]. A similar approach is employed in educational settings, where student SQL queries are graded by executing them on test databases to check if they return the expected result,

*Current affiliation: Advanced Micro Devices, Inc., Bengaluru, India
†Current affiliation: Morgan Stanley, Mumbai, India

Authors' Contact Information: Sunanda Somwase, sunandasomwase@gmail.com, Indian Institute of Technology Bombay, Mumbai, India; Parismita Das, das.parismita@gmail.com, Indian Institute of Technology Bombay, Mumbai, India; S. Sudarshan, Indian Institute of Technology Bombay, Mumbai, India, sudarsha@cse.iitb.ac.in.



ACM 2836-6573/2025/12-ART367
https://doi.org/10.1145/3769832

e.g. [2], as well as for testing the correctness of LLM-generated SQL queries with benchmarks such as Bird [13] and Spider [12].

More recently, there has been an increasing use of text-to-SQL systems based on LLMs for data analytics. Given the possibility of errors (whether due to "hallucination" or due to ambiguity in the textual specification), there is concern about the correctness of the LLM-generated queries. Text-to-SQL systems are often targeted at users who are not SQL experts, and who are thus unable to manually check the correctness of generated SQL queries. While there are several approaches to help users check the correctness of such queries, showing results on example datasets is one of the preferred methods [15, 18]. The creation of effective test datasets is key to detecting errors in generated SQL queries: such users can use our system to generate datasets to catch mutations, and check if the generated SQL queries give the expected result on each dataset, without needing to learn/understand SQL queries.

Given a collection of test datasets, a human tester would take the query specification (in a natural language such as English), and check if the output of the query on each dataset matches the expected output. The expected outputs can be stored and used for automated regression testing on the respective datasets. Traditionally, the test datasets are either created manually, or an existing database is used for testing. However, manually created test cases or existing databases may not ensure that all the relevant corner cases are covered. Automated generation of data for testing SQL queries is thus important to check for correctness.

Errors in formulating a query are modeled as mutations of the query [7, 21]. A *mutation* of a query is a syntactic variant of the query, for example, one where < in a condition such as `r.A < 5` is replaced by <=, or where an inner join is replaced by a left outer join. Note that some mutations may be semantically equivalent to the original query, although syntactically different. A dataset is said to *kill* a mutation of the query if the query and its mutation give different results on that dataset.

Given a query, the goal of automated test data generation is to generate datasets that can (ideally) kill all non-equivalent mutations of the query. Smaller datasets are preferred where results have to be checked by humans. It is not necessary to actually generate the mutations of a query; if the original query is wrong, and a mutant is the correct query, as long as there is a dataset that kills the mutant, the human tester would find that the original query does not give the expected result.

Given that queries may be complex, and the number of mutations is much larger, any system targets a particular class of queries and a particular space of mutations.

Early work on generating test datasets to kill mutants includes [7] and [23]. The state of the art in this area are the techniques, developed as part of the XData system, described in [2, 21]. Unlike the earlier work, the XData system generates multiple datasets targeted at different types of mutations. However, the techniques described in [2, 21] only handle simple queries with up to a single level of subqueries; they cannot handle queries with multiple levels of subqueries or queries with complex expression trees, which may include constraints on aggregation results. Such complex queries are (increasingly) common in the real world.

In this paper, we present a novel architecture and associated techniques for test generation that are designed to handle complex queries. Similar to the existing XData approach [2] and VeriEQL [11], our approach is based on creating tuples of constraint variables, and arrays of these tuples, and generating constraints on these variables based on the given query. We then use an SMT constraint solver (Z3, [6]) to generate solutions, from which datasets can be directly created. Multiple sets of constraints are created to generate multiple datasets, to kill a variety of mutations.

The main novel contributions of this paper are as follows:

(1) We propose (in Section 3.1) a representation of multiset relations using tuple counts, and with improved handling of null values. Our multiset representation allows a tuple count of 0 to indicate that the tuple does not exist. This allows the solver to choose the required relation size, up to some limit, rather than have to retry with different sizes, which is a significant improvement over earlier approaches.
(2) We propose (in Section 4) an approach for creating constraints to represent results of relational operations, such as joins (including semi and outer joins), selection, projection/duplicate elimination, and group-by/aggregate operations. Our approach, which we call the *result tables approach*, allows us to handle complex queries with multiple nested levels of operations, which earlier approaches could not handle. Details of data generation using this approach are provided in Section 5.
(3) We present (in Section 5.5) an approach to generating datasets to kill mutations that allows easy handling of complex queries as well as new types of mutations.
(4) We describe (in Section 6) how to handle correlated subqueries by performing decorrelation.
(5) We have implemented all the above extensions on the XData system. (A docker installation of our system can be accessed from https://gitlab.com/xdata/xdatapublic.)
(6) We present (in Section 8) a performance study that shows the effectiveness of our approach, and its improved effectiveness compared to the XData system for generating datasets to kill a variety of mutations on complex SQL queries. The study also shows that data generation is very efficient, taking only a few seconds even for complex queries. Our approach also outperforms the state-of-the-art VeriEQL system [11] in showing non-equivalence of query pairs.

Section 2 describes background material focusing on the solution approach, with a focus on the existing techniques used in the XData system. Section 3 provides an overview of our contributions, which are then described in more detail in Sections 4, 5 and 6. Section 7 details the differences from related work, including earlier work on XData [2, 21], and the query equivalence testing approach of VeriEQL [11].

2 Background and Solution Approach

In this section, we sketch the basics of our approach to data generation. Details of our approach are described in later sections.

Our approach extends the existing techniques developed as part of the XData system [2, 21]. The goal of XData is to catch errors in queries by generating multiple datasets that are targeted at killing mutations (within a target space of mutations) of the query. The mutation space reflects common errors when writing SQL queries. The first dataset is designed to generate a non-empty dataset for the original query, which can kill mutations that would generate an empty result on that dataset. Other datasets target one or more types of mutations.

The data generation algorithm of XData works as follows.

- Each table in the given schema is modeled as an array of tuples, where each tuple has a variable corresponding to each attribute. For example, given a relation r, we use an array r, with $r[i].A$ denoting attribute A of the ith tuple in the r array, with each attribute being a constraint variable.
- Constraints are generated on these variables to enforce integrity constraints such as primary keys, foreign keys, not null, and check constraints.
  For example, a constraint that r.A is a primary key could be enforced by asserting that $\forall i, i \neq j \Rightarrow r[i].A \neq r[j].A$.

- Further constraints are added to ensure a non-empty result for the given query. We discuss these in later sections.
- These constraints are then given to the Z3 solver [6], which is a Satisfiability Modulo Theories (SMT) constraint solver[1] to generate concrete values. The values generated by the solver are used to create a dataset.
- Further datasets are created, targeting each type of mutation. The constraints for each of these datasets are similar to the non-empty case constraints, but with modifications to generate a difference in the result of a targeted node in the query tree, from the result of a targeted mutation of that node.

Consider the following example query:

```
SELECT R.A, S.B
FROM R, S
WHERE R.A = S.A AND R.C > 6
```

Mutations of the query include, among others, mutations of the comparison operation > to any one of =, <, >=, or <=, and mutation of the inner join to left/right/full outer join.

To kill mutations of the comparison operation, XData [21] creates 3 datasets, one having a tuple with `R.C < 6`, one having a tuple with `R.C = 6` and one having a tuple with `R.C > 6`. These datasets also kill missing selection conditions. To kill a mutation of a join $r \bowtie_{r.A=s.A} s$ to outerjoin $r \supset\bowtie_{r.A=s.A} s$, XData [21] generates a dataset with a tuple, say $r[1]$ that has no matching s tuples, i.e., a dataset satisfying the constraint that $\nexists i \; s.t. \; r[1].A = s[i].A$.

While we retain the same basic approach described above, there are a number of differences and novel aspects to our contributions, which we summarize in Section 3, and then describe in more detail in the rest of the paper. The techniques described in this paper allow us to handle complex queries in a modular and extensible fashion, unlike the earlier XData approach [2, 21] which focused on simple queries with at most a single level of nesting.

3 Overview of Our Contributions

The major contributions of this paper include a novel multiset representation, a novel approach for defining intermediate results (which we call the result table approach), a new approach for creating datasets to kill mutations, and support for handling multi-level correlated subqueries.

Our approach provides a modular way of adding support for new operations and features compared to earlier work. Our contributions enable us to handle complex queries, overcoming the limitations of the earlier XData approach [2, 21]. In this section we provide an overview of the novel contributions of this paper, none of which is present in the earlier work on XData [2, 21]. Connections to other related work are described in Section 7.

3.1 Multiset Representation

Current generation SMT solvers do not support the set/multiset (bag) collection type, which could have been used to directly model relations. Test data generation systems such as [21, 25] instead represent a relation as an array of tuples, where each attribute of each tuple is a constraint variable.

A drawback of directly using an array is that the exact number of tuples should be predetermined. If there are too few or too many tuples, some constraints may not be satisfiable. To ensure a solution is found, data generation systems typically iteratively retry with different numbers of tuples for each relation, until a solution is found. However, this can be very inefficient, in particular if there are many relations.

[1]SMT solvers generalize SAT solvers by supporting types such as integers and rationals, along with some class of constraints, such as linear constraints, on these types.

In contrast, we use an approach of adding a count attribute with each relation, which allows tuples with zero count to be treated as invalid and ignored. Specifically, we represent the multiplicity of each tuple in a table using the additional integer type attribute CNT. The CNT of each tuple t_i can take any value that is zero or higher. A zero value signifies that the tuple t_i is invalid and our constraints are designed to ignore such tuples, using a helper function *isValid(t)* which returns true if CNT > 0 and false otherwise. A CNT value of one or greater signifies the multiplicity of the tuple in the table.

Our approach allows the solver to choose a number of tuples for each relation, that can be any value bounded by the number of tuples in the array, avoiding the high cost of retrying with different numbers of tuples in each relation.

For example, for a relation $r(A, B)$ we add attribute CNT to get the schema $r(A, B, CNT)$. If the constraint solver generates an array of tuples $\{('x', 5, 1), ('y', 6, 0)\}$, that corresponds to a relation with only one tuple $('x', 5)$. If instead the solver generates an array $\{('x', 5, 2), ('y', 6, 0)\}$, the result corresponds to the multiset $\{('x', 5), ('x', 5)\}$.

We note that VeriEQL [11] supports multisets by allowing duplicate tuples, and uses a "delete" bit to denote that a tuple is invalid; our approach is more compact.

3.2 Data Types

Null values are represented as follows. For numeric datatypes, we designate the negative of a large value from the domain of the attribute as the null value for that datatype.

For string attributes, we create an enumerated domain type, containing values from the sample data as well as values created to satisfy query constraints on strings, as described later in Section 5.3. To support null values, we enumerate one more value in the domain, for example the string *'ID_null'* for an attribute ID, and designate it as the null value for that attribute. When converting the solver output to an SQL dataset, the designated null values are converted back to the null value.

A boolean function *ISNULL_Attr(attr)* checks if *attr* value is null i.e. is equal to the designated null value. This function is used to support the `IS NULL` clause, and also to check if inputs to comparisons are null. As in Qex [25], every comparison includes a check to return true only if the values being compared are not null.[2] The constraints that we generate for aggregate computation (described later in Section 4.3) ensure that aggregate function computations, other than `COUNT(*)`, skip null values.

Using one of the domain values to represent the null value does carry a small risk of confusion between the actual domain value and the null value. Using a value close to the minimum of the domain minimizes the risk of clashing with query ranges or with the result of an arithmetic expression. In the worst case, we may fail to generate some datasets, and thus fail to catch some query errors. However, the risk of such an occurrence is extremely small.

An alternative scheme is to associate a null bit with every variable; however, this would double the number of variables and constraints, with a resultant risk of increased time to find solutions, as also an increased risk of solver timeout. We therefore chose to use a designated value to represent the null value. We note that the earlier approach of XData [2] for handling comparisons of two null values is less general and may allow two null values to be treated as equal.

3.3 Handling Integrity Constraints

Primary key constraints are enforced by adding constraints to ensure that (i) for every tuple in the relation, the CNT is either 0 or 1, and (ii) if any two tuples in the relation have the same values on the primary key attributes, at least one of them is invalid, i.e. has CNT = 0.

[2]We do not currently support the truth value "unknown", which is required for a full implementation of three-value logic.

If a relation (whether a database relation or a result table) does not have a primary key, we add constraints to ensure that if any two tuples in the relation have the same values on all attributes, at least one of them is invalid. This is similar to using all attributes as the primary key, but does not require valid tuples to have CNT = 1.

Foreign key constraints are enforced by creating constraints to ensure that for each valid tuple in the referencing relation, if the foreign key attribute values are not null, then there is a valid matching tuple in the referenced relation.

The above approach for encoding primary and foreign key constraints is more general than the earlier approach of XData [2, 21] which could not handle multiset semantics; it is similar to the approach of VeriEQL but works with our more compact representation of multisets.

3.4 Result Table Approach

Our approach uses *result tables* to represent the result of subexpressions of a query, as well as results of subqueries. Given any operation, such as a join or aggregate operation, we create a new table representing the result of the operation, and add constraints to ensure that the contents of the result table match the result of the operation on the input tables.

A key benefit of this approach is that it enables composability and simplifies the handling of complex expressions. Each operation can treat its inputs as an independent relation, so constraints only need to consider a single operator at a time. This decouples the logic for handling individual operations from the overall query structure, making it easier to support expressions with multiple, nested operators.

Our approach contrasts with the earlier approach of XData [2, 21], which is limited to queries with a simple select–project–join–aggregate structure and at most one level of subqueries. In contrast, our method supports more complex query structures, including multiple levels of subqueries, as discussed later in Section 6.

For a join operation, for example, the constraints on the result table ensure that:

(1) For every combination of valid tuples in the input tables that also satisfy the join conditions, there is a matching tuple in the result relation (which we call "*forward mapping constraints*"), and
(2) Every valid tuple in the result is generated from valid tuples in the input relations that satisfy the join conditions (which we call "*backward mapping constraints*")

By creating result tables bottom up from the inner-most/lower levels of a query, we define the result table for the main query. Solving for the generated constraints, with an additional constraint that the final result has at least one valid tuple, gives concrete values for each relation that result in a non-empty query result.

Details of the constraints on result tables are described in Section 4, where we describe how to create constraints to define the results of various operations such as selections/joins, groupby/aggregate expressions, duplicate-elimination, and set operations.

3.5 Datasets to Kill Mutations

The creation of datasets to kill mutations as described in [2, 21] was targeted to queries with a single level of subqueries.

We propose a more general approach for handling mutations of complex queries. Our approach performs a traversal of the query tree to create a collection of (node, mutation) pairs where mutation is a specific mutation at the node. The data generation procedure is then passed a specific (node, mutation) pair along with the query, and it modifies the constraints it generates to ensure the result

on the generated dataset differs for the unmutated and mutated operations. Our implementation, sketched in Section 5.5, makes it easier to add support for killing new mutations.

3.6 Handling Correlated Queries

We describe how to handle correlated subqueries, by performing decorrelation. Details are presented in Section 6. We note that earlier work on data generation did not handle correlated subqueries beyond simple single-level subqueries, whereas our approach is more general.

4 Result Table Constraints

In this section, we describe our result table approach, which generates constraints to ensure that the table representing a result of an expression has exactly the tuples generated by applying the expression on the input tables. This allows us to use the result table in place of the input expression, when creating constraints for higher level operations, making it possible to handle more complex queries. And at the top level, by adding a constraint that the result table must have at least one valid tuple, we can ensure that the solver generates a dataset that has a non-empty result.

We first describe how to handle queries with joins and selections, and later extend the approach to handle other operations such as aggregates and set operations. In all cases we enforce constraints on the result tables to ensure that no two valid tuples agree on all attributes, or more specifically if two tuples agree on all attributes (with null values treated like regular values), one of the tuples has a CNT of 0.

4.1 Join Result Table

We describe the creation of constraints for the result of a multi-table join. Hereafter, we refer to the join result table as *JRT* and the tables involved in the join operation as input tables.

Attributes of *JRT* include all the attributes from all the input tables along with a CNT attribute. We give unique names to the attributes of JRT: the attribute of *JRT* corresponding to attribute A from input table IT is given the name IT_A. The number of tuples in the array for *JRT* is determined based on the number of tuples in each input relation, along with primary and foreign key constraints, and selection and join conditions; details are provided later in Section 5.2.

For example given relations $r(A, B, CNT)$ and $s(B, C, CNT)$, consider the result of r inner join s on the join condition `r.B=s.B`. The result table *JRT* would have attributes:

$(r\_A, r\_B, r\_CNT, s\_B, s\_C, s\_CNT, CNT)$

If r has tuples $\{(1, 2, 2), (2, 3, 1)\}$ and s has $\{(2, 4, 3), (2, 7, 2), (2, 9, 0)\}$, then *JRT* would have tuples $\{(1, 2, 2, 2, 4, 3, 6), (1, 2, 2, 2, 7, 2, 4)\}$. The CNT of the tuple is the product of the counts of the input tuples. Note that the last tuple of s has CNT of 0, indicating it is invalid, so even though its B value of 2 matches that of tuples in r, the tuple is not present in the output.

The goal of constraint generation is to ensure that *JRT* has exactly the required tuples, based on the input tuples. We consider different forms of the join.

Inner Join. We first create forward mapping constraints, which ensure that every tuple in the result of the join operation (along with any selection conditions) on the input tables is present in *JRT*. Listing 1 shows how the constraints are generated. For ease of readability, we use the notation $T[i].A$ to refer to the attribute A of the ith tuple of the array corresponding to table T. The *assert* statements create the constraints that are subsequently solved by the SMT solver. In the implementation, we use the SMTLIB2 syntax for representing constraints, but for readability, we use a simpler syntax in this paper.

Listing 1 Join Result Mapping

function *AttributeMap* (IT, i, JRT, k)
  Let attributes of IT be $A_1, \ldots A_m$
  return $(JRT[k].IT\_A_1 = IT[i].A_1) \wedge \ldots \wedge (JRT[k].IT\_A_m = IT[i].A_m)$
    $\wedge (JRT[k].IT\_CNT = IT[i].CNT)$

function *ForwardMappingForJoin*($IT_1, \ldots, IT_n, JC$)
  /* $IT_1, \ldots, IT_n$ are input tables, JC the join conditions */
  for each combination of tuples $(IT_1[i_1], \ldots, IT_n[i_n])$ in $(IT_1 \times \ldots \times IT_n)$ **do**
    assert
      if $(IT_1[i_1], \ldots, IT_n[i_n])$ satisfies Join conditions JC
        $\wedge$ (isValid($IT_1[i_1] \wedge \ldots \wedge$ isValid($IT_n[i_n]$)))
      there exists tuple $JRT[k]$ in JRT **such that**
        isValid($JRT[k]$)
        $\wedge$ *AttributeMap*(IT_1, i_1, JRT, k) $\wedge \ldots \wedge$ *AttributeMap*(IT_n, i_n, JRT, k)

function *TupleCountMap*($JRT, k, IT_1, \ldots, IT_n$)
  return IsValid($JRT[k]$) $\Rightarrow$
   $(JRT[k].\text{CNT} = JRT[k].IT_1\_CNT \times \ldots \times JRT[k].IT_n\_CNT)$

function *BackwardMappingForJoin*($JRT, IT_1, \ldots, IT_n, JC$)
  for each tuple $JRT[k]$ in JRT **do**
    assert isValid($JRT[k]$) $\Rightarrow$
      Join conditions JC, with attributes $IT_i.A$ renamed
        to IT_i_A, are satisfied on $JRT[k]$
      and *TupleCountMap*($JRT, k, IT_1, \ldots, IT_n$)
      and for each input table IT_j $\exists j_i$ s.t. **such that**
        isValid($IT_j[j_i]$)
        $\wedge$ *AttributeMap*(IT_j, j_i, JRT, k)

Mapping function *AttributeMap()* generates constraints to link each attribute of the input table to its corresponding attribute in JRT. This function is used in both forward and backward mapping processes.

In the backward mapping, we ensure that each valid tuple in JRT has corresponding valid tuples in the input tables that meet the specified join and selection conditions. The backward mapping function includes a call to *TupleCountMap()*, which ensures the tuple count of each tuple in the join output is set to the product of the tuple counts of the input tuples that it was generated from.

Furthermore, to prevent duplicates from being generated among the tuples in JRT, we introduce constraints such that for each pair of tuples in JRT, if the attribute values are identical, at least one of the tuples is invalid, i.e., its $CNT = 0$.

Outer Join. Since outer joins are binary operations, the constraints are on two relations, unlike inner join case where we considered n-ary joins. We modify the forward mapping constraints of the inner join case to handle IT_1 `LEFT OUTER JOIN` IT_2 `ON` (*cond*), as follows. If a valid tuple $IT_1[i_1]$ from input table IT_1 and a valid tuple $IT_2[i_2]$ from input table IT_2 together satisfy the condition *cond*, the tuples are mapped to a tuple $JRT[k]$ in the JRT, similar to the inner join case. However, if a valid tuple $IT_1[i_1]$ in IT_1 has no match with any valid tuple in IT_2 on *cond*, then it is mapped to

the outerjoin result tuple, similar to the inner join case, but with all attributes of $JRT[k]$ in JRT originating from IT_2 set to null using the function *AttributeMapNULL()*.

In backward mapping, the generated constraint ensures that for each valid tuple $JRT[k]$ whose attributes mapped from the input tables satisfy the join conditions, there is a combination of valid tuples $IT_1[i_1]$ and $IT_2[i_2]$ that map to $JRT[k]$. Conversely, if the join condition is not satisfied, there is a valid tuple $IT_1[i_1]$ from IT_1 that maps to $JRT[k]$, and all attributes in $JRT[k]$ originating from IT_2 are null, and there is no valid tuple in IT_2 that satisfies the join conditions with $IT_1[i_1]$.

For the example we saw earlier, compared to the inner join of r and s, the left outer join of r and s would have the extra tuple $\{(2, 3, 1, null, null, 1, 1)\}$, since the r tuple $(2, 3, 1)$ does not have a matching s tuple. Note that the S_CNT value is 1, rather than 0, to ensure that $CNT = R\_CNT * S\_CNT$.

This approach can be adapted to generate a join result table for a right outer join by swapping the roles of input tables IT_1 and IT_2. The constraints for full outer join can be created by a combination of the left and right outer join cases. We omit details for brevity.

Semijoin. To generate constraints for a semijoin, we modify the inner join constraints, noting that there are only two inputs IT_1 and IT_2, and the JRT table has only attributes from IT_1. We modify the *ForwardMappingForJoin()* function, retaining the call *AttributeMap*(IT_1, i_1, JRT, k) but dropping the corresponding call for IT_2. We modify the *TupleCountMap()* function to set $JRT[k].CNT = JRT[k].IT_1\_CNT$.

We modify the *BackwardMappingForJoin()* function to assert that if isValid($JRT[k]$), then (i) there is a valid tuple $IT_1[j_1]$ for some j_1, satisfying the conditions in *AttributeMap*(IT_1, j_1, JRT, k) and further (ii) there is a valid tuple $IT_2[j_2]$ for some j_2 that satisfies the join condition JC with $IT_1[j_1]$.

Anti-Semijoin. Anti-semijoin returns the tuples from the left input that do not have any matching tuples in the right input. The tuple count is the same as that for the left input tuple. We modify the forward and backward mapping functions described above for the case of semijoin accordingly; details are omitted for brevity.

4.2 Selection Result Table

Selection conditions in the `WHERE` clause of a query can be handled along with the join conditions, if join conditions are present in the `WHERE` clause. Thus, we do not create a separate result table for selection operations that occur along with join conditions.

The case of a query with a single relation, without a join, can be handled by creating forward constraints that ensure that every valid tuple satisfying the selection maps to a valid tuple at the same position in the result table , while the backward constraints ensure that every valid tuple in the result table maps to a valid tuple at the same position in the input that satisfies the selection conditions.

The general case of a `FROM` clause with a single join expression (e.g. using outerjoins), with predicates in a `WHERE` clause is treated similar to the case of a single relation, after creating a result table for the join expression.

The case of `HAVING` clause, which imposes a selection condition on an aggregation result is dealt with similarly, by creating a result table for the aggregation, and then imposing the selection constraint on that result, as described shortly in Section 4.3.

4.3 Aggregation Result Table

The aggregation result table represents the result of group-by/aggregation operations performed on a single input table. The case of aggregation on the result of the join is handled by first creating

Listing 2 Aggregation Result Mapping

```
1: function ForwardMappingForAgg(ART, IT, GBAttr)
2:     for each tuple IT[i] in input table IT
3:     assert isValid(IT[i]) ⇒ ∃ k s.t. isValid(ART[k])
4:         ∧ ART[k].GBAttr = IT[i].GBAttr
5: function BackwardMappingForAgg(ART, IT, GBAttr)
6:     for each tuple ART[k] in ART
7:     assert isValid(ART[k]) ⇒ ∃ i s.t. isValid(IT[i])
8:         ∧ IT[i].GBAttr = ART[k].GBAttr
```

the result table for the join and then creating the result table for the aggregation using the join result table as input. We denote the aggregation result table as *ART*.

The attributes of *ART* include all the attributes in the group by clause, and the aggregation operations. For example, for the aggregation operation `SUM(R.A) GROUP BY R.B`, the *ART* has attributes *B* and *SUM_A*. Each tuple in *ART* corresponds to a group.

We first add constraints to ensure that no two tuples in *ART* agree on all group by attributes. The function *ForwardMappingForAgg*() shown in Listing 2 then adds constraints to ensure that the group-by attribute value of any valid tuple in the input table is also present in a valid tuple in the *ART* table. The function *BackwardMappingForAgg*() next adds constraints to ensure that every valid tuple in *ART* maps to at least one valid tuple in the input table with the same group by attributes.

Finally, we add constraints to ensure the correct value of the aggregate results in each group. Function *CountAggConstraints()* in Listing 3 specifies the constraints for computing the aggregate `COUNT(R.A) GROUP BY R.B`. For the case of `COUNT(*)`, the *not IsNull()* check is omitted so nulls are counted. The function can be easily generalized for grouping by multiple attributes, as well as for aggregation without any group by clause.

Note that if all tuples for a particular group have the attribute A value as null, `COUNT(R.A)` will be 0. If the semantics of having the aggregate result as null instead of 0 is desired, the function can be easily modified to set *COUNT_A* to null instead of 0.

We can generate constraints for `SUM(R.A)` by modifying the `COUNT` calculation by using "**if** not isNull($IT[i].A$) **then** $IT[i].A * IT[i].CNT$ **else** 0" in place of $IT[i].CNT$. Furthermore, to handle the case where all the input tuples for a group have null value for the aggregate attribute, we also add constraints for computing the `COUNT(R.A)`, and set `SUM(R.A)` to null if `COUNT(R.A)` is null.

Listing 3 Calculate COUNT(R.A) with group by R.B

```
1: function CountAggConstraints(ART, IT)
2:     for each tuple ART[k] in ART do
3:         for each tuple IT[i] in input table IT do
4:             let aggVal[t_i] =
5:                 if (isValid(IT[i] ∧ IT[i].B = ART[k].B
6:                   ∧ not IsNull(IT[i].A)
7:                 then IT[i].CNT else 0
8:         assert ART[k].COUNT_A =
9:             aggVal[t_1] + aggVal[t_2] + ... + aggVal[t_n]
```

To compute AVG(R.A) we use SUM(R.A)/COUNT(R.A), if both values are not null, and set it to null otherwise. The MIN and MAX aggregates can be handled in a similar fashion to SUM, using MAXVAL and MINVAL in place of 0, and taking the min or max of the different values, respectively.

To handle aggregation with duplicate elimination, for example, COUNT(DISTINCT A), we can modify the definition of $aggVal[t_i]$ in Listing 3 by adding at line 6 the extra check that there is no $j < i$ such that $IT[j]$ is valid, with the same value for group by attribute B, and with the same aggregated value A. We omit details for brevity, and note that this is under implementation.

Projection Operation. We treat the projection operation as an aggregation operation, grouped by the columns that are projected. For the case of projection with duplicate removal, the CNT of the output tuples would simply be set to 1, while for the case where duplicates are not removed, CNT is set to the result of the COUNT(*) aggregate function, which adds up the CNT values of all the tuples in the group. Note that if we simply project out the relevant attributes, the result table may have two tuples whose attribute values are the same, which does not match our multiset representation; these tuples must be merged and their counts added, which is implemented by the COUNT(*) aggregate.

HAVING *Clause:* We next consider the case where there is a constraint on the aggregation result, via a HAVING clause, illustrated by the following example:

```
SELECT R.A FROM R GROUP BY R.A
HAVING COUNT(R.B) < 10
```

We first define the result table for the aggregation operation, including any aggregate values that are used in the HAVING clause. We can create another result table using the aggregation result table as input, and then impose the selection conditions from the HAVING clause on the aggregate result attribute of that table. As an optimization, we directly impose the condition on the aggregation results without creating another level of result tables.

It is worth noting that the result table approach permits a much simpler and cleaner way of handling constraints on aggregation results, compared to XData [2], which had a much more complicated way of handling such constraints.

4.4 Set Operation Result Table

Now we describe how we handle set operators using the result table approach. Set operator queries are of the form Q1 SETOP Q2, where SETOP is one of UNION, INTERSECT, or EXCEPT, with variants ALL or DISTINCT, which preserve or remove duplicates.

We first create result tables for Q1 and Q2 independently. Consider $Q1RT$ and $Q2RT$ be the result tables for queries Q1 and Q2 respectively. Note that both tables must have the same number/types of attributes, and we assume they have also been renamed to have the same attribute names.

The first step in set operation result table computation is similar to a full outer join of the two queries, with the difference being that the attributes are stored only once; but in addition, the CNT values from $Q1RT$ and $Q2RT$ are stored in the result as $Q1RT\_CNT$ and $Q2RT\_CNT$. If the tuple is present with non-zero counts in both $Q1RT$ and $Q2RT$, both $Q1RT\_CNT$ and $Q2RT\_CNT$ would be non-zero. Otherwise, at least one of the two counts would be zero.

The final step computes the CNT value for each tuple, and varies depending on the aggregate operation. For the case of UNION ALL, the CNT of the result is $Q1RT\_CNT + Q2RT\_CNT$, while for UNION, CNT is set to $min((Q1RT\_CNT + Q2RT\_CNT), 1)$. For INTERSECT ALL, CNT is set to $min(Q1RT\_CNT, Q2RT\_CNT)$ while for INTERSECT, CNT is set $min(Q1RT\_CNT, Q2RT\_CNT, 1)$. For EXCEPT ALL, CNT is set to $max(Q1RT\_CNT - Q2RT\_CNT, 0)$, while for EXCEPT the CNT is set to $min(max(Q1RT\_CNT - Q2RT\_CNT, 0), 1)$.

5 Dataset Generation

We now provide details of our algorithms for generating datasets, starting with generation of non-empty datasets, followed by an outline of how to decide the result table size, and then datasets for killing mutations.

The input to our data generation algorithm is the query tree, which is an extended relational algebra expression, allowing multi-way joins, as well as subqueries which may occur in the `FROM`, `WHERE` and `SELECT` clauses.

Before data generation, the query tree is preprocessed, and decorrelation is carried out, replacing nested execution by semi-joins and joins, as described later in Section 6. A `SELECT` clause without any aggregation is replaced by a projection operation, with duplicate elimination if `DISTINCT` is present; a `SELECT` clause with aggregation, along with the `GROUP BY` clause if present, is replaced by a group-by/aggregation operation. A `HAVING` clause becomes a selection operation on top of the group-by/aggregate operation.

Thus, an SQL query is converted to a tree of extended relational algebra operations. Note also that the inner join operation is a multi-way join which can take 2 or more inputs, and join order is not relevant for our purposes.

5.1 Dataset for Non-Empty Results

The overall algorithm for generating a non-empty dataset for a query without subqueries, other than non-correlated `FROM` clause subqueries, is shown in Listing 4.

Function *generateDatabaseConstraints()* is invoked to generate SMT constraints corresponding to database integrity constraints, and the function *generateConstraintsForQuery(q)* is invoked to generate constraints for the query. The function *generateDatasetUsingConstraints()* then solves the constraints using the Z3 SMT solver to get a non-empty dataset.

Function *generateConstraintsForQuery(q)* traverses the query tree bottom-up, and generates a result table for each operation node.

For each operation visited during the traversal, a result table is created, and constraints are generated to define the result table, as described earlier in Section 4. Then the operation is replaced by the result table, which becomes an input to the parent operation.

Note that subqueries can occur as children of join (including outer join), and semi/anti-semi join operations, and are thus traversed as part of the bottom-up traversal. The procedure handles subqueries that are children of (inner/outer/semi) joins as long as they do not have correlation variables. Subqueries that occur under connective operations, such as exists, in, or for scalar

Listing 4 Data generation algorithm

```
1:  function generateDatasetForQuery(querytree q)
2:      generateDatabaseConstraints( )
3:      generateConstraintsForQuery(q)
4:      createNonEmptyConstraints(q)
5:      generateDatasetUsingConstraints( ) /* Calls Z3
6:          solver on constraints generated */
7:  function generateConstraintsForQuery(querytree q)
8:      For each node n in bottom-up traversal of query blocks
9:          Create result table for operation n
10:         Create constraints to define result table for n
11:         Replace n by result table for n
```

subqueries, any expression, or have correlation variables, are first decorrelated as described later in Section 6, and data generation as above is done subsequently.

5.2 Deciding Number of Tuples

As described earlier, by allowing the tuple count CNT of a tuple to be 0, and treating such tuples as invalid in our constraints, the constraint solver is enabled to generate solutions with the number of tuples in a relation ranging from 0 to the maximum size of the array representing each relation. However, that still leaves the question of what the size of the array, i.e., the maximum number of tuples, should be for each relation, a problem not addressed in [2].

For each base relation, we decide on the number of tuples as follows. For each occurrence of the relation in the query, we add one tuple. Further, if a relation r with n tuples has a foreign key referencing relation s, we ensure s has at least n tuples, i.e., we use the maximum of the sizes of all referencing relations.

If there are any constraints on an aggregate computed on the relation, we determine the number of tuples required by solving constraints on aggregates, following the approach of [2]. For example, if there is a constraint `SUM(r.A) >= 500`, along with a constraint `r.A <= 100`, solving aggregate constraints shows that we require at least 5 tuples to be present in a group (this could be satisfied by generating a single tuple with a duplicate count of 5, unless a unique/primary key constraint requires 5 different tuples to be created). This lower bound is used in two ways. First, to determine the result array size for the input to the aggregation operation, and second, to get lower bounds on sizes of input relations. For example, if the input to the aggregation is a join, the lower bound requirement down to the input tables from which the join result table is generated, to ensure that the join can have the required number of tuples. The lower bound is propagated down through the query tree.

Next, we consider how to determine the size of result tables based on the sizes of the input relations.

Join Result Table: Consider IT_1 and IT_2 are the input tables for join result table JRT, with maximum number of tuples as m and n respectively. Then the maximum number of tuples in JRT can, in the worst case, be $m * n$. However, in practice, the number of tuples is much lower, due to selection conditions as well as because joins are almost never cross products, and are most commonly foreign-key to primary-key joins. We can get better size bounds, as follows.

To compute the maximum number of tuples in the join of n relations, we first compute a value $M(IT_i)$ for each relation IT_i as described below, and then multiply these numbers to get the size bound. The value $M(IT_i)$ is initialized to the number of tuples in IT_i, for all i. If there is a selection condition of a form { primary/unique key of IT_i = constant }, we set $M(IT_i) = 1$. If the join condition equates the primary/unique key of input table IT_i to attributes of another relation that do not form a primary/unique key, we set $M(IT_i) = 1$, since the join condition can map each tuple of the other table to at most 1 tuple of IT_i. If a join condition equates the primary/unique keys of two input tables IT_i and IT_j, we set $M(IT_i) = 1$ if IT_i is the larger of the two tables, since in this case the result cannot be larger than the smaller of the relations.

If the join condition involving two relations does not involve primary/unique keys, we heuristically reduce the size of JRT as follows: if $M(IT_i) > 2$ for both relations, we set $M(IT_i) = 2$ for the smaller $M(IT_i)$. As a further heuristic, we cap the size of the result table to some maximum value, to avoid problems with solver timeouts with large relation sizes.

We use the same bound for outer joins too, since the size bounds assume that every tuple has a matching tuple for each join condition. For semijoins, the result table size bound is the number of tuples in the left-hand side input.

Aggregate Result Table: The maximum number of tuples in the aggregate result table (*ART*) can be equal to the number of tuples in the input table, if each tuple in the input table has a different group by attribute values. Lesser tuples are possible in *ART*, based on the number of groups present.
Set Operator Result Table: For set operator result tables, we set the maximum number of tuples in the result table equal to the sum of the number of tuples in the input tables for union, to the minimum of input table sizes for intersect, and to the left input table size for except.

5.3 Handling String Types

Although Z3 supports solving of string constraints using either of two solvers (Z3str and seq), we found that the performance of these solvers to be very slow for handling string constraints, such as comparison and like clause matching.

String constraints using the `LIKE` clause (where the pattern is assumed as in [2] to be a constant value), as well as comparisons are handled by using the string solver from [2] to generate values that can satisfy the constraints. In [2] both comparison and `LIKE` constraints (e.g. `r.A < 'Bio' AND r.B LIKE 'C_'`) are replaced by equality constraints with satisfying values generated by the string solver (e.g. `r.A = 'A' AND r.B = 'Ca'`). Our implementation handles `LIKE` constraints in the same way, but unlike [2] our implementation retains string comparisons, such as < or >. This allows the solver flexibility in selecting values to satisfy the constraints.

String values are then mapped to enumerated values for the domain for that attribute as described earlier in Section 3.2. Since Z3 does not internally support comparison (such as < or >) between enumerated constants, we define a function to map the enumerated constants to distinct integers, satisfying the lexicographic sort order, and replace a comparison of the form $x < y$ by $attrmap(x) < attrmap(y)$ where attrmap is the above-mentioned mapping function for the attribute domain.

5.4 Support for Other SQL Features

Our techniques handle all the basic SQL operations, including join, /outer joins, selection, projection, group by/aggregation, duplicate elimination, and set operations. We also handle nested subqueries with correlation variables (as described later in Section 6). We handle numeric and string types, and limited operations on date types by converting dates to integers. There are several features that we are currently working on, such as ranking queries (with dense/sparse rank, and partitioning), OLAP expressions, and case expressions. We also plan to add support for string manipulation functions, such as substring and concat, and date manipulation functions, among others, which are widely used in data wrangling applications.

5.5 Datasets for Killing Mutations

Listing 5 shows how datasets are generated for killing mutations. The function *collectMutation-Structs*(q) traverses the query tree, and at each node it decides on what mutations are to be considered, and adds the pair (node, mutation), which we call a mutation structure, to a collection which it returns.

Listing 5 Dataset generation to kill mutants

```
1: /* A mutation structure contains (Node, mutation) pairs */
2: function generateDatasetToKillMutations(query q)
3:     msSet = collectMutationStructs(q)
4:     for each mutation struct ms in msSet
5:             generateDatasetForQuery(q, ms)
```

The function *generateDatasetForQuery*(q, ms), where ms is a mutation structure, is a variant of *generateDatasetForQuery*(q), which generates a dataset targeted at killing the specified mutation at the specified node.

The mutations we target include the following:

(1) Mutation of a relational operation $<, \leq, =, \geq, >$ by any other operation from the set in any predicate. Note that predicates may be part of a selection, join, or having clause condition. We generate datasets for 3 mutations, one with $<$, one with $=$ and one with $>$ which will together catch all the mutations.
(2) Mutation by dropping of all or parts of predicate conditions, for example, either one or both of the conjuncts of the predicate "`A < B AND C < 5`". To ensure the dropping has an impact, we can replace a condition by the negation of the condition, so the dataset generated for one variant will return false for the other variant.
(3) Mutation of any of the aggregate operations `SUM, COUNT, AVG, MIN`, and `MAX` by any of the others, as well as by variants which include the `DISTINCT` modifier. To ensure the change has an impact, we ensure there are three valid input tuples in one group, two of which have the same (non-zero) value to be aggregated, and one that has a different value.
(4) Mutation by dropping or adding attributes from the `GROUP BY` clause. To ensure the change has an impact, we ensure that there is at least one tuple that differs on the dropped group by attribute, but agrees on the other attributes.
(5) Mutation by adding or dropping the `DISTINCT` modifier in the `SELECT` clause. To ensure that the change has an impact, we ensure that one of the input tuples to the `DISTINCT` operation has a count > 1.
(6) Mutation between `EXISTS` and `NOT EXISTS` and similarly between `IN` and `NOT IN` in subquery connectives. Any dataset that satisfies `EXISTS` will fail `NOT EXISTS` and vice versa. The above also catches mutations that drop a predicate that involves a subquery with any of these connectives.
(7) Mutation between variants of string predicates, such as between `LIKE` and `ILIKE`, between "`_`" and "`%`" in `LIKE` patterns, etc. To ensure that the change has an impact, we create strings that satisfy `ILIKE` but not `LIKE`, "`%`" but not "`_`", etc.
(8) Replacement of inner join by outer join, handled as explained below.

Our basic approach for killing specific mutants is based on earlier approach used in XData [2, 21], although the approach for enumeration of the different datasets via recursive traversal and creation of mutations is different from the earlier approach of XData, which does not describe a systematic way of enumerating the mutations. Further, we consider mutations both in the top level and arbitrary inner level subqueries in a uniform manner.

To catch mutations between inner join and left, right, or full outer join, our approach is based on [21]. We consider each relation that participates in an inner join, and the predicates connecting it with other relations. For example, with an inner join of `R, S` and `T` on `R.A = S.A AND R.A = T.A`, we consider first relation `R`, and add constraints that ensure that for each of the remaining relations (`S` and `T` there is a tuple that satisfies the remaining selection/join conditions (here the inferred condition `S.A = T.A`), and further, there is no tuple in `R` that matches these `S` and `T` tuples on the predicate `R.A = S.A`. As shown in [21] this approach creates a small number of datasets that catch a large number of join type mutations.

For queries that already have outer joins, we currently use the above approach for killing join type mutations, after converting outer joins to inner joins.

6 Handling Nested Subqueries

In this section, we describe how to handle nested subqueries, that may use correlation variables, by performing decorrelation. Decorrelation replaces nested subquery invocation by joins or semijoins.

The mechanism for decorrelation of subqueries varies based on the subquery connective, such as `EXISTS, IN`, etc., and on the use of correlation variables. Our approach is based on decorrelation techniques described in [9, 16, 17]. Our approach focuses on making the rewriting simple, rather than on optimizing execution performance which is the usual goal of decorrelation, since we do not actually execute the queries, but just use them for data generation. Further, we can potentially use rewritings that do not preserve equivalence, since test data generated for such rewritings can still potentially catch errors in the original query. We first describe how to handle a single level of nesting, and then consider multiple levels. We present our approach informally using examples, for ease of understanding, and omit implementation details.

We use SQL syntax with the `SEMI JOIN` and `ANTI SEMI JOIN` operations. SQL implementations include semijoin as an internal operation, although they do not typically support the semijoin syntax in SQL. We use the syntax only for notational convenience, and do not actually execute semijoins on the database.

6.1 Single-Level Simple Subqueries

We first consider single-level where clause subqueries, starting with simple subqueries select-project-join subqueries.

Exists Connective: Consider the following query:

```
Q1: SELECT R.A, R.B
    FROM R
    WHERE EXISTS (
        SELECT S.A FROM S WHERE S.B = R.A)
```

The above query can be decorrelated by transforming it to the following query.

```
Q1R: SELECT R.A, R.B
     FROM R SEMI JOIN (SELECT S.A, S.B FROM S)
        ON S.B = R.A
```

The rewritten query has the `WHERE` clause subquery replaced by a `FROM` clause subquery without correlation, with a `SEMI JOIN` operation. Note that the attribute S.B used in the correlation condition has been added to the `SELECT` clause list for the inner query, and the correlation condition `S.B = R.A` has become the `SEMI JOIN` condition. We also note that, although the subquery in the `FROM` clause of `Q1R` could be replaced by the relation S in this specific case, the transformation illustrates the general case.

Not Exists Connective: Consider the following query:.

```
Q2: SELECT R.A, R.B   FROM R
    WHERE NOT EXISTS (
        SELECT S.A FROM S WHERE S.B = R.A)
```

We decorrelate this subquery using an `ANTI SEMI JOIN` (also known as an anti join) as follows:

```
Q2R: SELECT R.A, R.B
     FROM R ANTI SEMI JOIN
         (SELECT S.A, S.B FROM S)
             ON (S.B = R.A)
```

Note that `S.B` has been added to the `SELECT` clause of the subquery to make it available for the `ANTI SEMI JOIN`.

IN/NOT Connectives: Consider the following queries

```
Q3:  SELECT *
     FROM R
     WHERE R.B IN (SELECT S.B FROM S)

Q4:  SELECT *
     FROM R
     WHERE R.A NOT IN (SELECT S.B FROM S)
```

The decorrelated version of the two queries is as follows:

```
Q3R: SELECT *
     FROM R SEMI JOIN S ON (R.A = S.B)

Q4R: SELECT *
     FROM R ANTI SEMI JOIN S
          ON (R.A = S.B OR R.A IS NULL OR S.B IS NULL)
```

The `ANTI SEMI JOIN` condition in the decorrelated query allows `R.A` and `S.B` to be `NULL`. This is due to a subtle distinction between `NOT IN` and `NOT EXISTS` due to null values. If `R.A` or any `S.B` is null, the `NOT IN` would fail since we cannot be sure there is no matching tuple, whereas rewriting it without the null check (whether using an `ANTI SEMI JOIN` as above, or using `NOT EXISTS`) would result in an empty subquery result even with nulls, and the corresponding `R` tuple would appear in the result.

Subqueries with `ANY` connective (e.g. `= ANY`, or `< ANY`) can be decorrelated to a semijoin, whereas the `= ALL` connective can be translated to an `ANTI SEMI JOIN` with the negative of the condition (e.g., `=` is replaced by `!=`).

Scalar Subqueries: Scalar subqueries with comparison operations can be decorrelated as below:

```
Q5:  SELECT S.A
     FROM S
     WHERE S.B = (SELECT R.B FROM R)
```

The above query is decorrelated to

```
Q5R: SELECT S.A
     FROM S SEMI JOIN
        (SELECT R.B FROM R) ON (R.B = S.B)
```

under the assumption that the scalar subquery never returns more than 1 answer; if it returns more than 1 answer, there would be a runtime exception. Decorrelated queries cannot throw such runtime exceptions.

Scalar subqueries in the `SELECT` clause can be handled by using an `OUTER JOIN` instead of a `SEMI JOIN`.

`FROM` *Clause Subqueries:* Subqueries that appear in the `FROM` clause are linked to the outer query by a join connective. Such subqueries could have correlation variables from outer-level queries. The `LATERAL` keyword also allows correlation variables to be passed from one subquery to another subquery that is at the same nesting level. Such subqueries can be decorrelated in a way similar to `WHERE` clause subqueries, except that instead of a `SEMI JOIN` we would perform a join or outerjoin of the result table for that subquery.

6.2 Single-Level Complex Subqueries

More complex subqueries that involve correlation variables are handled by creating a result table that contains all relevant parameter values along with the results of the subquery if invoked with the parameter values.

In some simple cases, the relevant parameter values come from the relations available in the subquery itself. In other cases we need to add extra joins to provide the parameter values.

To decorrelate subqueries in general, we need relevant values for the correlation variables. Consider the following query.

```
Q6: SELECT R.A
    FROM R
    WHERE R.C = (
        SELECT SUM(S.C) FROM S
        WHERE S.B = R.B)
```

In this simple example, the correlation variable R.B is equated to S.B, we can use S.B to provide relevant values for the correlation variable. Since the subquery involves aggregation, which is computed separately for each invoked value of the correlation variable, we add the correlation variable to a group by clause in the nested subquery.

```
Q6R: SELECT R.A
     FROM R SEMI JOIN
            (SELECT S.B AS S_B, SUM(S.C) AS SUM_C
             FROM S
             GROUP BY S.B) AS ASQ1
           ON (R.B = S_B AND R.C = SUM_C)
```

In the above example if COUNT(*) were used, some minor changes are required to handle its special semantics.

If the subquery already had a group by clause, S.B would have been added to the existing group by attributes.

But in general, we cannot depend on a relation in the subquery to provide relevant values for correlation variables. To decorrelate subqueries in general, we have to simulate the invocation with different values of correlation variables by adding a join with relations that generate the distinct values of the correlation variables that the original subquery would have been invoked with. In general, a superset of the invoked values is also acceptable since it would generate the same final result.

To illustrate, consider the query below, where instead of the condition R.B = S.B, we have the condition S.B < R.B.

```
Q7:  SELECT R.A
     FROM R
     WHERE R.C = (SELECT SUM(S.C) FROM S
                  WHERE S.B < R.B)
```

The query is decorrelated as follows.

```
Q7R:  SELECT R.A
      FROM R SEMI JOIN
            (SELECT  R1.B AS R1_B , SUM(S.C) AS SUM_C
             FROM S, (SELECT DISTINCT R.B FROM R) AS R1
             WHERE S.B < R1.B
             GROUP BY R1.B) as ASQ1
            ON (R.B = R1_B AND R.C = SUM_C)
```

To make the distinct values of R.B available in the result of the decorrelated subquery, we have added a join (cross product) with the set of distinct R.B values using the FROM clause subquery (SELECT DISTINCT R.B FROM R). The value of R.B is also added to the result of the decorrelated subquery to be used in the semijoin condition.

6.3 Multi-Level Nesting of Subqueries

In the case of multi-level nested `WHERE` clause subqueries, the correlation variables used in a subquery may come from some level higher than the immediately higher level query. Decorrelation is done level by level from the innermost level subqueries upwards. When decorrelating a subquery at a particular level, lower level subqueries have already been decorrelated.

Consider a query `Q` with a subquery `SQ1` that itself has a subquery `SQ2`. If `SQ2` uses a correlation variable, say `A` from `Q`, we introduce a join when decorrelating `SQ2`, to provide values for `A`, as we saw in `Q7R`. Further, when decorrelating `SQ1`, even if `A` is not used in `SQ1`, we add a similar join to provide values for `A`, and `Q` is also added to the output of the decorrelated `SQ1`. The decorrelated `SQ1` has a semijoin with the decorrelated `SQ2` on condition `SQ2.A = SQ2.A`. Similarly, the decorrelated `Q` has a semijoin with the decorrelated `SQ1` on condition `Q.A = SQ1.A`. Together, these simulate the effect of passing correlation variable `A` from `Q` to `SQ2` through `SQ1`.

7 Related Work

The AGENDA system [7] was one of the first tools for the generation of test data, but it considered only the database schema, and not queries when generating data. Qex [24] [25] presented an approach for generating input tables to ensure non-empty results for a given SQL query, by generating constraints and solving them using the Z3 SMT solver [6]. However, the datasets generated by Qex are not targeted at killing mutations. Tuya et al. [23] provide a classification of a space of mutations, and describe techniques to kill some of these mutations, but do not provide a system to automatically generate test data to kill these mutations.

Our work builds on the earlier work on test data generation done as part of the XData system [2, 3, 21]. All the contributions we have described in Sections 3, 4, and 5, and the decorrelation techniques described in Section 6 are novel, and are not present in the earlier work on XData. Our contributions allow us to handle complex queries in a modular and extensible fashion, unlike the earlier XData approach, which focused on simple queries with at most a single level of nesting.

Work on query equivalence testing is closely related, although there are significant differences. Early work on query equivalence handled restricted classes of queries, such as conjunctive queries. Cossette [4] represents tuples with multiplicity, but does not use an SMT solver, and thus has limitations in showing equivalence. Recent results on query equivalence testing, such as [8, 11, 14, 26], have significantly extended the class of queries and integrity constraints considered. [14] uses a second-order theorem prover to check query equivalence, and thus handles relations of arbitrary size, but is limited in the class of queries it can handle.

These systems attempt to prove equivalence, but if the queries are not equivalent, some of them, such as [8, 11, 14], attempt to generate a dataset that is a witness to non-equivalence. The state of the art system for equivalence testing is VeriEQL [11].

A major limitation of the query equivalence tools is that they cannot be used in case a correct query is not available, and the goal is to generate datasets to check if a given query is correct, which is the problem that we address in this paper.[3]

There are some similarities between our result table approach and the encoding used by VeriEQL [11], which is also based on creating tables representing the result of each operation. VeriEQL [11] uses a "deleted" boolean value for a similar purpose to our use of 0 multiplicity, but handles multisets by having multiple copies of tuples.

[3]One could conceivably enumerate mutations of a given query, and use the query equivalence checking tool to generate datasets to prove non-equivalence for each of the mutants. However, the number of such mutations is very high (exponential in the query size as described in [21]), which will generate a very large number of datasets. In contrast, our approach, which extends the earlier XData approach to data generation, is able to generate a smaller number of datasets that can kill a much larger class of mutations.

Table 1. XDataBM Original Queries and Results

Query type	Num. Queries	Num. Mutants	Avg # Datasets	Avg # Tuples	Avg Time /query(s)
Single level	31	162	8	13	13.16
Set operator	6	41	6	23	6.40
`FROM` clause subqueries	13	48	6	16	7.29
`WHERE` clause subqueries	34	156	14	25	32.62
Total	84	407	10	21	19.64

Their representation for joins inherently requires the join result table size to be the product of input table sizes, resulting in an exponential growth in result table sizes, whereas our approach avoids this requirement, with sizes computed as described in Section 5.2. For the case of group-by and aggregation, we believe our approach gives a simpler mapping. Further, VeriEQL does not handle correlated subqueries, and has other limitations, such as not handling commonly used data types such as strings, dates, etc.

We present a performance comparison of our approach with VeriEQL in Section 8, which demonstrates the benefits of our approach.

It must be kept in mind that VeriEQL cannot generate datasets for testing queries when a correct query is not available, and thus cannot be used for testing queries in such cases, for example, for testing SQL queries generated by end users using LLMs. Even for cases where the correct query is available, such as when grading student queries, test data needs to be generated only once per correct query, and then used to check if student queries give the correct results, which is cheaper than having to check equivalence separately per student query.

8 Performance Study

In this section, we present a performance study of the effectiveness of our approach in catching errors in SQL queries, across queries using various SQL features, and across a variety of types of mutations, showing its benefits over prior work.

Student queries with errors were used to test XData (e.g., [1, 2]). The queries used there were mostly simple queries, and there was limited diversity in mutations. Existing benchmarks for query equivalence testing, such as Leetcode, Calcite, and Literature [11] consist of query pairs where most of the pairs are equivalent, and thus cannot be used to check for effectiveness in killing of mutants.

We therefore created a collection of more complex queries, along with a large number of non-equivalent mutations, which we refer to as XDataBM. These queries are based on the University schema [22]. The benchmark is available at https://gitlab.com/xdata/xdatabenchmark. We also created a collection of TPC-H queries with non-equivalent mutations, which we used for comparison.

The XDataBM benchmark is based on 84 original queries, encompassing single-level and multi-level nested queries with and without correlation among other features. The distribution of query types is summarized in Table 1. The data generation algorithm produces multiple datasets for each original query, with the initial dataset designed to yield non-empty results.

The tests were run on a computer with an AMD Ryzen 7 5700G with Radeon Graphics 3.8 GHz, and 16 GB of memory, running Ubuntu 22.04.3 LTS. As mentioned in Section 5.2, we cap the size of

Table 2. Effectiveness at Killing Mutations

Mutation type	Total Mutants	# of Mutants Killed			
		VeriEQL	XDataO	USSm	XDataN
Comparison	54	38	31	40	**52**
Join Type (inner vs. outer)/Condition	73	36	49	63	**68**
String comparison	78	22	38	51	**76**
Extra/missing group-by attribute	18	**18**	12	14	**18**
Column replacement	4	2	3	**4**	3
Constrained Aggregation	10	7	7	9	**10**
Distinct	7	**7**	5	**7**	6
Aggregation function	35	32	17	28	**33**
Missing subquery	33	12	20	25	**31**
Subquery connective	50	11	35	40	**49**
Set Operator	41	**41**	24	35	**41**
Null conditions	4	**4**	3	**4**	**4**
Total	407	230	244	320	**382**

result tables, with the default cap being 16 tuples. The data generation algorithm was executed for each original query, with a 120 sec. timeout for the Z3 SMT solver execution for each dataset.

As can be seen from Table 1, across all query types, with our system for test data generation, the average number of datasets per query is 10, and the average number of tuples per dataset is 21. The count of tuples includes all tuples in input tables and result tables. The average time taken for generating all the datasets for a query is 19.64 sec.

8.1 Effectiveness in Killing Mutations

Each of the 84 queries has associated with it a number of mutations of different types, with a total of 407 non-equivalent mutations across the 84 queries. Table 2 shows the distribution of mutation types; the table also shows the number of these mutations killed by different approaches. Note that a mutation is killed if the original query and the mutant generate different results on at least one of the datasets.

To study the effectiveness of our approach, which we refer to as XDataN to distinguish it from the earlier XData approach, we compare it with three alternatives for killing mutants. The first is using the sample University database, which we call USSm, a small database that was manually created by the authors of [22]. The second is the XData implementation from [2], (labeled XDataO).

The last is using the state-of-the-art query equivalence tool VeriEQL [11]. Note, however, that VeriEQL can only be used to compare pairs of queries. For our target applications, which require generation of datasets to test a single given query, VeriEQL is not applicable.[4] However, given a query and a mutant, we can use VeriEQL to check if they are equivalent, and to generate a dataset to prove non-equivalence, and our performance experiments use VeriEQL in this mode.

Overall, XDataN kills around 94% of the mutations, clearly outperforming USSm, XDataO, and VeriEQL, which kill 79%, 60% and 57% of the mutations. The reasons for XDataN failing to kill some mutations include lack of support for some mutations in our implementation, and timeouts by the

[4]XDataBM includes queries with natural join, which is not supported by VeriEQL; we therefore rewrote such queries using the inner join operator.

Table 3. Mutant Killing by Subquery Type

Query Type	Total Mutants	Killed Mutants	
		VeriEQL	XDataN
FROM clause subquery	48	44	**46**
WHERE clause with correlation	80	0	**79**
WHERE clause without correlation	33	0	**31**
Scalar/IN/NOT IN Subqueries	43	**42**	36
Total	204	86	**192**

Table 4. Results on TPC-H Queries

System	Queries Handled	Mutants Killed	
		Manual	ChatGPT
XDataN	17 / 22	175 / 213	71/118
VeriEQL	1 / 22	0 / 213	0/118

solver. Handling a larger class of queries/mutations, and optimizing the constraints to reduce solver execution time are areas of ongoing work.

Although [2] shows that XDataO outperforms the small university dataset (USSm) for queries presented in [2], XDataBM contains many complex queries with features that XDataO is not able to handle, such as subqueries with correlation and aggregation, and thus USSm beats XDataO on the queries in XDataBM. In contrast, XDataN clearly outperforms USSm on these queries. Further, the queries in XDataBM use constants that are present in USSm; for queries that use different constants in selection conditions, USSm would perform much worse.

Table 3 shows the effectiveness of XDataN and VeriEQL on killing mutations of different types of subqueries. VeriEQL is unable to handle WHERE clause subqueries with EXISTS connective, with or without correlation, whereas XDataN could kill almost all mutations of these and other types of subqueries.

We also ran experiments on the 22 queries from the TPC-H benchmark, which are relatively complex queries, with subqueries present in most queries.[5] We created two sets of mutants. One set of 213 mutants, covering 18 of the 22 queries, was created manually. To avoid bias in generation of mutations, we created a second set of mutants using ChatGPT by prompting it with the TPC-H queries, and asking it to generate about 6 or 7 syntactically correct mutants for each of the 22 TPC-H queries. Out of 124 mutations thus generated, 6 were found to be equivalent to the original query, and were dropped. The remaining 118 were used for testing.

The results are shown in Table 4. XDataN is currently able to handle 17 out of 22 queries, and generated on average 33 datasets for each of the 17 queries. These datasets killed 175 out of 213 mutants that we created, as well as 71 out of 118 non-equivalent mutants created by ChatGPT. Of the 47 that were not killed, 23 were from the 5 TPC-H queries that we do not currently handle; of the remaining 25 queries, 14 were mutations of ORDER BY and LIMIT clauses. Thus, among the queries that we handle, and mutation types other than ORDER BY and LIMIT, our system could kill 71 out of 81 mutations.

Mutations of ORDER BY cannot always be caught by data generation, since results may be ordered even without an ORDER BY clause, as a byproduct of the chosen query plan. Checking for mutations of ORDER BY and LIMIT by comparing queries directly, as well as handling the 5 queries that are not currently handled, and handling further types of mutations, are areas of ongoing work.

[5]The TPC-H queries we used may be found at https://docs.starrocks.io/docs/benchmarking/TPC-H_Benchmarking/.

8.2 Execution Time

The average time taken by XDataN to generate all datasets for a query, across all queries considered in XDataBM, was 19.64 sec., while the average time to generate each dataset is 2.04 sec. For TPC-H, the average time to generate all datasets for a query was 157 seconds, while the time to generate each dataset ranged from 360 msec to 29 seconds, with an average of 4.7 seconds,

To compare the overheads of our constraint generation approach versus that of XData, we measured the time for the solver to generate output for the first (non-empty) dataset for all queries on which XData ran successfully. We note that XData uses CVC3, while XDataN uses Z3, so the comparison is not exact. Nevertheless we note that generating the non-empty dataset with XData took an average of 63 msec, in comparison with 47 msec for XDataN.

The datasets generated by XDataN can kill a large class of mutations, whereas VeriEQL is executed for pairs of queries, and generates a dataset showing non-equivalence for that pair. Thus, the goals are different, and a direct comparison of execution time between XDataN and VeriEQL is not meaningful. However, to get some idea of the relative performance on creating and solving constraints, the average time taken by VeriEQL to check for equivalence for a pair of queries is 3.51 seconds, can be compared to the 2.04 seconds taken on average by XDataN to generate each dataset.

To study the impact of increasing query size on the cost of dataset generation, we created queries with chain joins which are of the form

```
SELECT R1.A1
FROM R1, R2, ..., Rn
WHERE R1.B1=R2.A2 AND R2.B2=R3.A3 AND ...
GROUP BY R1.B1
HAVING COUNT(*) > 3
```

Each relation `Ri(Ai,Bi)` has primary key `Ai`, and all but the last relation has foreign key `Bi` referencing `Ri+1.Ai+1`.

We also created a version of these queries with the same relations, but without any join conditions, to study the performance effect of number of tuples in join result relations due to the use of cross products.

To compare with VeriEQL, which only checks for (non)equivalence of a pair of queries, we used as the comparison (non-equivalent) query `SELECT R1.A1 FROM R1 WHERE 1=2` which generates the empty result.

Table 5 shows the average taken for solving SMT constraints to generate one dataset for XDataN, and the time for VeriEQL to solve SMT constraints for testing equivalence of the query with the empty-result query. TMO denotes timeout (set to 120 secs.). Results for the join and cross product cases are shown separately. For join queries, XDataN gives good performance even up to 10 joins; for cross product performance is worse, with 8 relation query still finishing, although the 10 relation join query times out. We note that cross products are very uncommon, and the much lower costs for the join case shows the effectiveness of our approach for limiting the result table sizes.

In contrast, VeriEQL is faster at smaller number of relations, but has timeouts even for 8 relation joins, with and without cross product. While we cannot be sure of the reasons behind timeout for VeriEQL, we believe that it is because their join constraint encoding results in the number of constraint tuples increasing exponentially with query size. This is similar to the results of our approach on cross products. VeriEQLs result table mapping approach requires creating cross products of table sizes, even if there is a join condition that prevents cross products; in contrast, our representation allows us to create smaller result tables for the normal case of joins, and thus efficiently handle larger queries.

Table 5. Execution Time With Increasing Query Size

#Tables	XDataN (s)		VeriEQL (s)	
	Join	Cross	Join	Cross
2	0.137	0.260	0.008	0.007
4	0.490	16.190	0.054	0.042
6	1.680	24.900	1.354	1.041
8	4.330	75.730	TMO	TMO
10	9.130	TMO	TMO	TMO

We note that the forward mapping and backward mapping constraints of the result table described in Section 4.1 employ nested quantifiers with forall and exist. It is noted in [21] that when quantifying over a fixed size array, unfolding of constraints often results in considerable speedup; for example instead of asserting "for all tuples t in r, predicate $P(t)$ is true", if r has 4 tuples we instead assert "$P(r[1]) \wedge P(r[2]) \wedge P(r[3]) \wedge P(r[4])$". Unfolding of exists quantifiers is similarly done by using or ("$\vee$") instead of and ($\wedge$). We have implemented such unfolding, and all the numbers we present are with unfolding. We also measured the performance of constraint solving without unfolding, and found it to be significantly worse; for brevity, we omit details.

9 Conclusion and Future Work

We have developed and implemented an architecture and techniques for test data generation that can handle complex queries. Our performance results show the significant benefits of our techniques over prior work.

We believe that test data generation will become increasingly important for checking the correctness of queries, as people who are not SQL experts use LLMs to generate SQL queries.

Future work includes (i) extensions to handle more SQL features such as `LIMIT` clause with `ORDER BY`, windowing functions, string and date manipulation functions that are common in data wrangling tasks, (ii) extensions to catch further types of mutations, and (iii) optimizing the generated constraints to handle larger queries and larger datasets. We also plan to extend our system to support checking of query equivalence. Another potential area for future work is to explore the use of data generation to improve the detection of bugs in database system implementations, following the approach of Rigger et al. [19, 20].

Acknowledgements:

We thank Chaitra Gurjar, Kartik Patel, and Kumaran Kartikeyan, for their contributions towards the implementation of some of the techniques described in this paper. We also thank all students who have contributed to the development of the XData system since the version described in [2], including: Bikash Chandra, Saurabh Chaturvedi, Ravi Shankar, Sai Balaji Polakampalli, Rahul Sharma, Pooja Gayakwad, and Deeksha Kasture. We also thank the referees for their detailed comments that have helped improve the presentation of the paper.

References

[1] Bikash Chandra, Ananyo Banerjee, Udbhas Hazra, Mathew Joseph, and S. Sudarshan. 2019. Automated Grading of SQL Queries. In *IEEE Intl. Conf. on Data Engineering, ICDE*. IEEE, 1630–1633. https://doi.org/10.1109/ICDE.2019.00159

[2] Bikash Chandra, Bhupesh Chawda, Biplab Kar, K. V. Maheshwara Reddy, Shetal Shah, and S. Sudarshan. 2015. Data generation for testing and grading SQL queries. *VLDB J.* 24, 6 (2015), 731–755. https://doi.org/10.1007/s00778-015-0395-0

[3] Bikash Chandra, Bhupesh Chawda, Shetal Shah, S. Sudarshan, and Ankit Shah. 2013. Extending XData to kill SQL query mutants in the wild. In *Procs. International Workshop on Testing Database Systems, DBTest 2013*. ACM, 2:1–2:6. https://doi.org/10.1145/2479440.2479442

[4] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries. *Proc. VLDB Endow.* 11, 11 (2018), 1482–1495. https://doi.org/10.14778/3236187.3236200

[5] Claudio de la Riva, María José Suárez-Cabal, and Javier Tuya. 2010. Constraint-based test database generation for SQL queries. In *Proceedings of the 5th Workshop on Automation of Software Test (AST '10)*. 67–74. https://doi.org/10.1145/1808266.1808276

[6] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS (Lecture Notes in Computer Science)*, Vol. 4963. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[7] Yuetang Deng, Phyllis Frankl, and David Chays. 2005. Testing database transactions with AGENDA. In *Proceedings of the 27th International Conference on Software Engineering* (St. Louis, MO, USA) *(ICSE '05)*. Association for Computing Machinery, New York, NY, USA, 78–87. https://doi.org/10.1145/1062455.1062486

[8] Haoran Ding, Zhaoguo Wang, Yicun Yang, Dexin Zhang, Zhenglin Xu, Haibo Chen, Ruzica Piskac, and Jinyang Li. 2023. Proving Query Equivalence Using Linear Integer Arithmetic. *Proc. ACM Manag. Data* 1, 4, Article 227 (dec 2023), 26 pages. https://doi.org/10.1145/3626768

[9] Mostafa Elhemali, César A. Galindo-Legaria, Torsten Grabs, and Milind Joshi. 2007. Execution strategies for SQL subqueries. In *Procs. ACM SIGMOD*. ACM, 993–1004. https://doi.org/10.1145/1247480.1247598

[10] Florian Haftmann, Donald Kossmann, and Alexander Kreutz1. 2005. Efficient Regression Tests for Database Applications. In *Procs. of the Conference on Innovative Database Systems Research (CIDR)*.

[11] Yang He, Pinhan Zhao, Xinyu Wang, and Yuepeng Wang. 2024. VeriEQL: Bounded Equivalence Verification for Complex SQL Queries with Integrity Constraints. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 1071–1099. https://doi.org/10.1145/3649849

[12] Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida Wang, and Tao Yu. 2025. Spider 2.0: Evaluating Language Models on Real-World Enterprise Text-to-SQL Workflows. arXiv:2411.07763 [cs.CL] https://arxiv.org/abs/2411.07763

[13] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chen-Chuan Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs. In *NeurIPS 2023*.

[14] Mudathir Mahgoub Yahia Mohamed, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. 2024. Verifying SQL queries using theories of tables and relations. In *Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning (EPiC Series in Computing)*, Nikolaj Bj{\o}rner, Marijn Heule, and Andrei Voronkov (Eds.), Vol. 100. EasyChair, 445–463. https://doi.org/10.29007/rlt7

[15] Arpit Narechania, Adam Fourney, Bongshin Lee, and Gonzalo Ramos. 2021. DIY: Assessing the Correctness of Natural Language to SQL Systems. In *Proceedings of the 26th International Conference on Intelligent User Interfaces (IUI '21)*. 597–607. https://doi.org/10.1145/3397481.3450667

[16] Thomas Neumann. 2025. Improving Unnesting of Complex Queries. In *Datenbanksysteme für Business, Technologie und Web (BTW 2025)*. Gesellschaft für Informatik, Bonn, 25–47. https://doi.org/10.18420/BTW2025-01

[17] Thomas Neumann and Alfons Kemper. 2015. Unnesting Arbitrary Queries. In *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*. Gesellschaft für Informatik e.V., Bonn, 383–402.

[18] Zheng Ning, Yuan Tian, Zheng Zhang, Tianyi Zhang, and Toby Jia-Jun Li. 2024. Insights into Natural Language Database Query Errors: from Attention Misalignment to User Handling Strategies. *ACM Trans. Interact. Intell. Syst.* 14, 4, Article 25 (Dec. 2024), 32 pages. https://doi.org/10.1145/3650114

[19] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. 1140–1152.

[20] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 211 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428279

[21] Shetal Shah, S. Sudarshan, Suhas Kajbaje, Sandeep Patidar, Bhanu Pratap Gupta, and Devang Vira. 2011. Generating test data for killing SQL mutants: A constraint-based approach. In *IEEE International Conference on Data Engineering, ICDE*. IEEE Computer Society, 1175–1186. https://doi.org/10.1109/ICDE.2011.5767876

[22] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. 2020. *Database system concepts* (7 ed.). McGraw-Hill.

[23] Javier Tuya, María José Suárez-Cabal, and Claudio de la Riva. 2007. Mutating database queries. *Information and Software Technology* 49 (04 2007), 398–417. https://doi.org/10.1016/j.infsof.2006.06.009

[24] Margus Veanes, Jonathan de Halleux, Nikolai Tillmann, and Peli de Halleux. 2009. *Qex: Symbolic SQL Query Explorer*. Technical Report MSR-TR-2009-2015. https://www.microsoft.com/en-us/research/publication/qex-symbolic-sql-query-explorer/ Updated January 2010.

[25] Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux. 2010. Qex: Symbolic SQL Query Explorer. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 425–446.

[26] Qi Zhou, Joy Arulraj, Shamkant B. Navathe, William Harris, and Jinpeng Wu. 2022. SPES: A Symbolic Approach to Proving Query Equivalence Under Bag Semantics. In *IEEE Intl. Conf. on Data Engineering, ICDE*. IEEE, 2735–2748. https://doi.org/10.1109/ICDE53745.2022.00250

Received April 2025; revised July 2025; accepted August 2025