Validating Query Rewriting using Mutant-Killing Data Generation

Anish Tiwari

June 22, 2025

Abstract

Large Language Models (LLMs) can rewrite complex SQL queries to improve performance or simplify logic. However, verifying that these rewrites are semantically equivalent to the original queries is challenging. In this project, we use XData's dataset generation technique to enrich the efficiency of sampled datasets in identifying semantic equivalence between original and rewritten SQL query pairs. Later we evaluated dataset-driven validation on queries rewritten by the LITHE system, evaluated how the system can check the subtle semantic differences.

1 Introduction

Query rewriting with Large Language Models (LLMs) have been found to be beneficial for SQL-to-SQL transformation tasks [1] enabling improvements in performance tuning, structural simplification, etc. However, verifying that the rewritten queries are semantically equivalent to the original ones is a non-trivial task. Even minor logic shifts in filters, joins, or subqueries may lead to different results, which are often difficult to detect through manual inspection or static analysis. This project focuses on evaluating semantic equivalence between two SQL queries using a test-data generation approach inspired by the XData framework [1].

We model the problem as one of mutant killing—treating a rewritten query as a possible mutation of the original. If a dataset can be constructed such that the rewritten query produces different results from the original, we say that the mutant is *killed* [2], indicating a semantic discrepancy. This approach allows us to systematically test whether the logic of two queries matches under all inputs, without requiring full formal verification. The XData framework introduces this data-driven method of generating tuples that satisfy (or falsify) particular query constraints. By encoding query logic into satisfiability constraints and solving them using SMT solvers such as Z3 [4], we can generate inputs that are specially designed to highlight any logical differences between the two queries. We adapt XData's techniques to handle more complex SQL constructs, such as nested subqueries and IN clauses, and to apply constraint mutations systematically.

Although our methodology applies to any pair of SQL queries, in this project we evaluate it specifically on rewritten query pairs generated by the LITHE system. LITHE uses LLMs to rewrite queries [1], offering a realistic set of query pairs to test our equivalence-checking strategy.

2 Prior Work

Testing SQL query correctness has been a long-standing challenge, particularly in educational settings, query optimization, and mutation-based validation. The XData framework proposed by Gupta et al. [2] introduced a data-driven approach that generates small datasets to differentiate between original and mutated SQL queries, effectively identifying semantic discrepancies. Gatterbauer et al. [3] further explored the problem in the context of grading SQL assignments by constructing datasets that highlight logical differences between a student's query and a reference solution. This methodology laid the groundwork for several extensions, including support for complex queries with nested subqueries, JOINs, and aggregation [5]. These works underscore the theoretical hardness of query equivalence which is undecidable in general and NP-complete in restricted cases, thus motivating empirical approaches for validation. Our project builds directly on this body of work, implementing an enhanced and modular data generation pipeline to find subtle logical discrepancies through targeted test data generation.

3 Methodology

Our approach follows a multistep pipeline adapted from the XData framework [5] to test the semantic correctness of SQL query rewrites. Each stage in the pipeline—from query parsing to constraint solving and test data generation—is built as a separate, manageable part so that each step can be developed and tested independently.

3.1 Query Parser and QueryTree Builder

We use the `sqlglot` library to parse input SQL queries and generate an abstract syntax tree (AST). This AST is then converted into a structured tree called a `QueryTreeNode`. This tree keeps track of important query components such as SELECT, FROM, WHERE, GROUP BY, and any subqueries.

Each node in the tree captures key elements like filter conditions (predicates), aggregation functions, and column references. This structure makes it easy to walk through different parts of the query and analyze constraints, even when there are subqueries or nested logic.

Example: Given the query:

```
SELECT ca_zip, SUM(ws_sales_price)
FROM web_sales, customer, customer_address
WHERE ws_bill_customer_sk = c_customer_sk
  AND c_current_addr_sk = ca_address_sk
  AND ca_state = ‘CA’
  AND SUBSTRING(ca_zip, 1, 5) = ‘85669’
GROUP BY ca_zip;
```

The parser identifies the table joins and filters like `ca_state = ‘CA’` and organizes them in the tree so that they can be accessed easily for constraint generation.

3.2 Constraint Extraction

From the parsed query tree, we extract logical constraints from WHERE clauses, JOINs, GROUP BY terms, and subqueries. These constraints describe the conditions that each tuple in the database must satisfy to appear in the result.

For example, in the query above, the following constraints are extracted:

- `ca_state = 'CA'`
- `SUBSTRING(ca_zip, 1, 5) = '85669'`
- `ws_bill_customer_sk = c_customer_sk`
- `c_current_addr_sk = ca_address_sk`

The SUBSTRING constraint is treated as a pattern constraint on strings. JOIN conditions are treated as equalities between foreign key and primary key columns. All these constraints are then passed to the solver for generating satisfying values.

Once collected, the constraints are translated into a logical formula that the Z3 SMT solver [4] can understand. Z3 treats each column as a variable and finds concrete values (tuples) that satisfy the formula. For multi-table queries, all constraints are combined into a single logical expression that represents the full query logic.

3.3 Constraint Solving

Once the constraints are extracted from the query structure, we use the Z3 SMT solver [4] to compute satisfying assignments. Z3 is a high-performance solver that works over logical formulas involving arithmetic, strings, and other data types. It checks whether a given set of constraints is satisfiable and, if so, returns a model—i.e., concrete values that fulfill all conditions.

Each column in the schema is treated as a symbolic variable. For example, an equality condition like `ca_state = 'CA'` is encoded as a string equality in Z3. A JOIN condition like `c_current_addr_sk = ca_address_sk` becomes a constraint asserting equality between two integer variables. Similarly, string functions such as `SUBSTRING(ca_zip, 1, 5) = '85669'` are translated into Z3 expressions using its built-in string operations.

All the encoded formulas are passed to the solver as a single conjunction. If the solver determines the formula is satisfiable, it returns a model with values for each column variable. These values are then used to construct full tuples for each involved table.

Example: For a constraint like `SUBSTRING(ca_zip, 1, 5) = '85669'`, Z3 may return a value such as `ca_zip = '85669AAAAA'` to satisfy the condition. Similarly, for join conditions, Z3 ensures the appropriate foreign keys and primary keys are assigned the same value (e.g., `c_customer_sk = 0`, `ws_bill_customer_sk = 0`).

These satisfying assignments are extracted and translated into actual table rows, which are then inserted into the database. This step ensures that the generated dataset is consistent with all query constraints and will lead to a non-empty result when the query is executed.

3.4 Dataset Generation and Insertion

To evaluate the semantic equivalence of rewritten SQL queries, we generate synthetic datasets that trigger specific logical behavior in the query outputs. This section explains our data generation pipeline in two stages: first, generating data to ensure the query returns a non-empty result, and second, modifying the data to expose semantic differences between the original and rewritten queries (mutation killing).

3.4.1 Non-Empty Result Data Generation

After Z3 provides a satisfying model for the query constraints, we follow a sequence of steps to convert this model into actual table rows that ensure the original query produces at least one result. Below, we describe the main steps involved in this phase.

Database Connection Setup We first establish a connection to the target database where the generated data will be inserted. If necessary, we also reset the schema or truncate relevant tables to start from a clean state.

Foreign Key Dependency Sorting To ensure that inserts do not violate referential integrity, we perform a topological sort of the tables based on their foreign key relationships. For example, if `web_sales` references `customer`, then `customer` must be inserted first. This guarantees that parent rows exist before any dependent rows are added.

Populating Constraint-Satisfying Columns All columns that appear in the Z3 model are populated directly with the values returned by the solver. These values are guaranteed to satisfy all WHERE, JOIN, and subquery constraints present in the original query. For example, Z3 might assign a value like `ca_zip = '85669AAAAA'` to satisfy a `SUBSTRING()` condition.

Populating Other Required Columns Some columns may not be constrained by the query logic but still appear in the SELECT, GROUP BY, ORDER BY, or aggregation functions. For such columns, we generate values consistent with their data type. For instance, if a string column is not constrained, we assign a default value like 'A' * 'length', while numeric fields may receive values like 0 or 1. This ensures that all required output fields are present without violating schema definitions.

Preparing and Executing Insert Statements After filling all column values, we construct SQL `INSERT` statements for each table in the topologically sorted order. These statements are then executed using the active database connection. Each tuple is inserted carefully to maintain foreign key consistency and to ensure that the query will return a non-empty result when run.

Guaranteeing Result Visibility For queries with GROUP BY or aggregation logic, we generate multiple rows with the same group key when necessary. This ensures meaningful aggregation results. If a HAVING clause exists, values are tuned so that the grouped output satisfies the condition.

This systematic data generation ensures that the original query, when executed, will return a verifiable and consistent non-empty result. It provides a stable baseline for comparing the outputs of original and rewritten query variants.

3.4.2 Mutation-Specific Data Generation

After generating a dataset that ensures the original query produces results, we proceed to test whether a rewritten version of the query behaves differently. This step aims to "kill" semantic mutations—logical changes introduced during rewriting—by generating test data that exposes the differences. This idea builds on the mutation testing technique described in the XData framework [5].

The key principle here is to reuse the original dataset structure but adjust specific column values so that they fall outside the logic introduced by the rewritten query, while still satisfying the original one. If the rewritten query fails to return the same output, the mutant is considered successfully killed.

Identifying Mutation Targets We begin by analyzing the abstract syntax tree (AST) of the rewritten query to locate modified logic. These typically include comparison operators (e.g., `>`, `=`), IN clauses, or added filtering subqueries. For each of these "mutated" locations, we generate new versions of the constraints to target the altered logic.

Mutating Comparison Predicates For comparison-based conditions like `sales > 100`, we apply several small logic mutations. We replace the operator one by one with `=`, `>=`, and `<`, and then re-run the Z3 solver with these new constraints. For each satisfying model Z3 returns, we extract values, prepare SQL insertions, and update the database. These new rows are designed to satisfy the original query but potentially violate the rewritten one.

Mutating IN Clauses When the rewritten query uses an IN clause such as:

```
i_item_id IN ('A', 'B', 'C')
```

we convert each value into its own equality—like `i_item_id = 'A'`—and generate one test record for each value. We also introduce a completely new value not present in the original IN list (e.g., `'ZZZZ'`) that still satisfies the rest of the query logic. This helps us test whether the rewritten query correctly includes or excludes records under IN filtering logic.

Rebuilding Constraints and Invoking Z3 For each mutation, the logical formula is updated to reflect the new constraint. We invoke the Z3 solver again, just as we did in the non-empty case. If Z3 finds a model, we populate the relevant tables with the new values. This step is repeated for each mutated version of the constraint.

Maintaining Original Query Semantics Even while mutating constraints, we make sure that the new rows still satisfy the original query. This ensures that any change in result is due to the logic introduced by the rewritten version, not due to invalid data. In effect, the rewritten query should filter out these rows, while the original should still return them—making the mutation detectable.

Summary: This two-step dataset strategy allows us to robustly evaluate the semantic equivalence between original and rewritten queries. First, we generate a base dataset that satisfies all constraints of the original query using Z3, ensuring it produces a non-empty and meaningful result by honoring all join paths, filters, and schema rules. This dataset acts as the foundation for comparative evaluation. Next, we focus on mutation-specific data generation, where we traverse the query tree to identify mutations—such as changes to comparison operators or IN clauses—mutate the targeted constraint, and regenerate a dataset using Z3 that continues to satisfy the original query but is filtered out by the mutated one. By observing whether the rewritten query shows a difference in output while the original continues to succeed, we detect and "kill" semantic differences, thus validating or rejecting the correctness of the rewrite.

4 Query Analysis

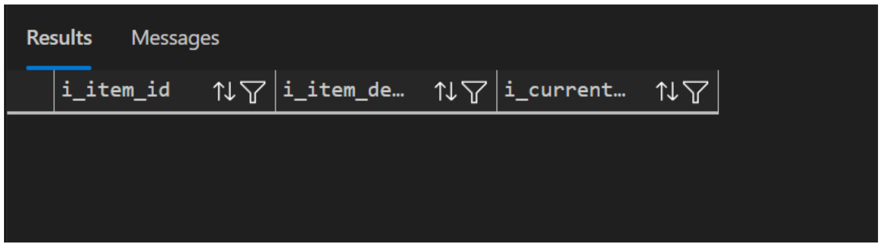
In this section, we evaluate our dataset generation framework on several rewritten SQL queries using the TPC-DS benchmark schema. These rewrites were automatically generated by the LITHE system [1], and our goal is to test whether they preserve the semantics of the original query. For each query pair, we apply both non-empty and mutation-specific data generation and compare the outputs to check for semantic differences.

Case 1: TPC-DS Q36, 84 – Non-Empty Case

Original Query

```
SELECT TOP 100 i_item_id, i_item_desc, i_current_price
FROM item, inventory, date_dim, catalog_sales
WHERE i_current_price BETWEEN 29 AND 29 + 30
  AND inv_item_sk = i_item_sk
  AND d_date_sk = inv_date_sk
  AND d_date BETWEEN CAST('2002-03-29' AS DATE)
                AND DATEADD(DAY, 60, CAST('2002-03-29' AS DATE))
  AND i_manufact_id IN (705, 742, 777, 944)
  AND inv_quantity_on_hand BETWEEN 100 AND 500
  AND cs_item_sk = i_item_sk
GROUP BY i_item_id, i_item_desc, i_current_price
ORDER BY i_item_id;
```

Initial Output of Original Query (Before Data Generation):


Results Messages

```
i_item_id | i_item_de... | i_current...
```

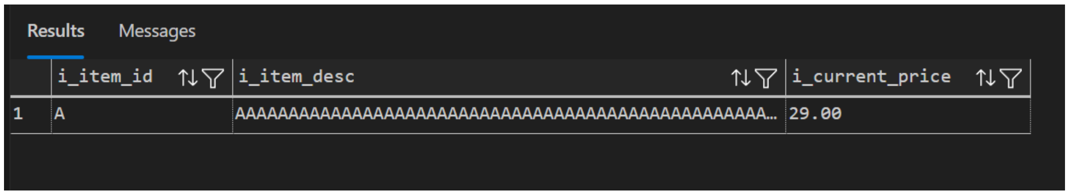
Rewritten Query

```
SELECT TOP 100
    i_item_id,
    i_item_desc,
    i_current_price
FROM
    item, inventory, date_dim, catalog_sales
WHERE
    i_current_price BETWEEN 29 AND 29 + 30
    AND inv_item_sk = i_item_sk
    AND d_date_sk = inv_date_sk
    AND d_date BETWEEN CAST('2002-03-29' AS DATE)
                AND DATEADD(DAY, 60, CAST('2002-03-29' AS DATE))
    AND EXISTS (
        SELECT 1
        FROM (
            SELECT 705 AS manufact_id UNION ALL
            SELECT 742 UNION ALL
            SELECT 777 UNION ALL
            SELECT 944
        ) AS v
        WHERE v.manufact_id = i_manufact_id
    )
    AND inv_quantity_on_hand BETWEEN 100 AND 500
    AND cs_item_sk = i_item_sk
GROUP BY
    i_item_id, i_item_desc, i_current_price
ORDER BY
    i_item_id;
```

After Data Generation:

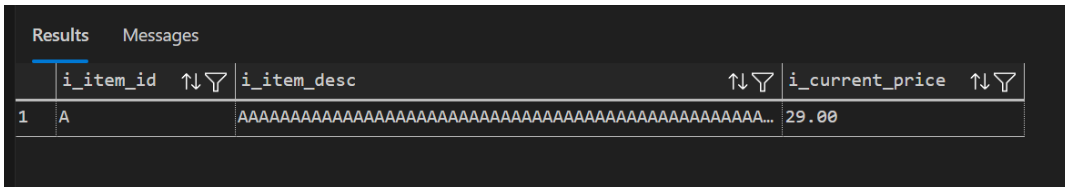
We apply our data generation pipeline to create a dataset that satisfies the constraints of the original query and guarantees non-empty output.

Original Query Output:



```
Results    Messages
   i_item_id   i_item_desc                                           i_current_price
1  A           AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...  29.00
```

Rewritten Query Output:



```
Results    Messages
   i_item_id   i_item_desc                                           i_current_price
1  A           AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...  29.00
```

Discussion:

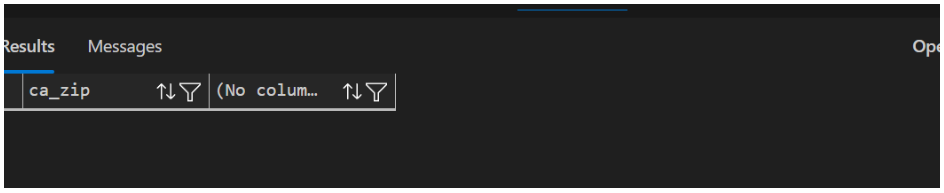
This case evaluates whether rewriting an `IN` clause to an equivalent `EXISTS` subquery impacts semantic correctness. The original query uses `i_manufact_id IN (705, 742, 777, 944)`, while the rewritten version replicates this logic using an inline `EXISTS` over a hardcoded virtual table. Although both are logically similar in structure, the output remains the same because the generated test data happens to include a record where `i_manufact_id` falls within the specified list. As a result, no observable difference is triggered between the queries. However, the absence of output differences does not prove semantic equivalence; it may simply indicate that the dataset did not activate a discrepancy. Our approach is effective for quickly catching incorrect rewrites, but occasional false negatives or undetected issues may still occur. In this case, the mutant is not killed, though future datasets may still reveal semantic deviations.

Case 2: Q15 – IN Clause Mutation

Original Query

```
SELECT TOP 100 ca_zip, SUM(cs_sales_price)
FROM   catalog_sales, customer, customer_address, date_dim
WHERE  cs_bill_customer_sk = c_customer_sk
       AND c_current_addr_sk = ca_address_sk
       AND (SUBSTRING(ca_zip, 1, 5) IN
            ('85669', '86197', '88274', '83405',
             '86475', '85392', '85460', '80348', '81792')
            OR ca_state IN ('CA', 'WA', 'GA')
            OR cs_sales_price > 500)
       AND cs_sold_date_sk = d_date_sk
       AND d_qoy = 2
       AND d_year = 2001
GROUP BY ca_zip
ORDER BY ca_zip;
```

Initial Output of Original Query (Before Data Generation): The original query, when executed on an empty or irrelevant dataset, produces no results.



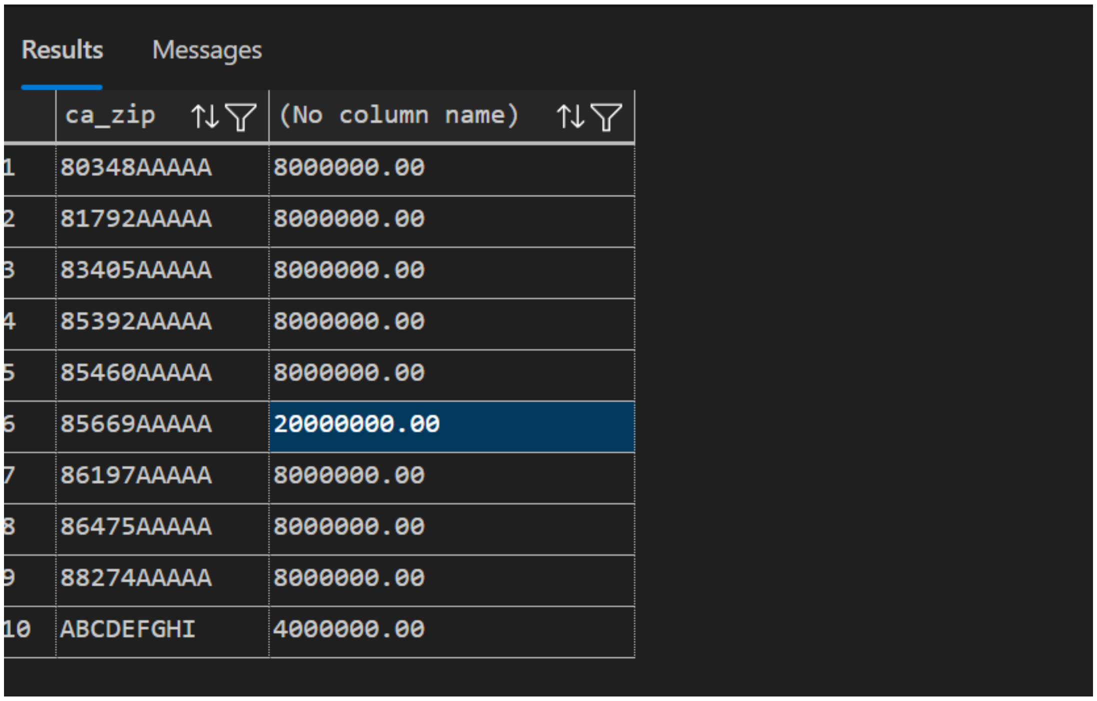
```
ca_zip        (No colum...
```

Rewritten Query (Mutated)

```
SELECT TOP 100 ca.ca_zip, SUM(cs.cs_sales_price) AS total_sales
FROM   catalog_sales cs, customer c, customer_address ca, date_dim d
WHERE  cs.cs_bill_customer_sk = c.c_customer_sk
       AND c.c_current_addr_sk = ca.ca_address_sk
       AND cs.cs_sold_date_sk = d.d_date_sk
       AND (
            ca.ca_state IN ('CA', 'WA', 'GA')
            OR cs.cs_sales_price > 500
       )
       AND d.d_qoy = 2
       AND d.d_year = 2001
GROUP BY ca.ca_zip
ORDER BY ca.ca_zip;
```

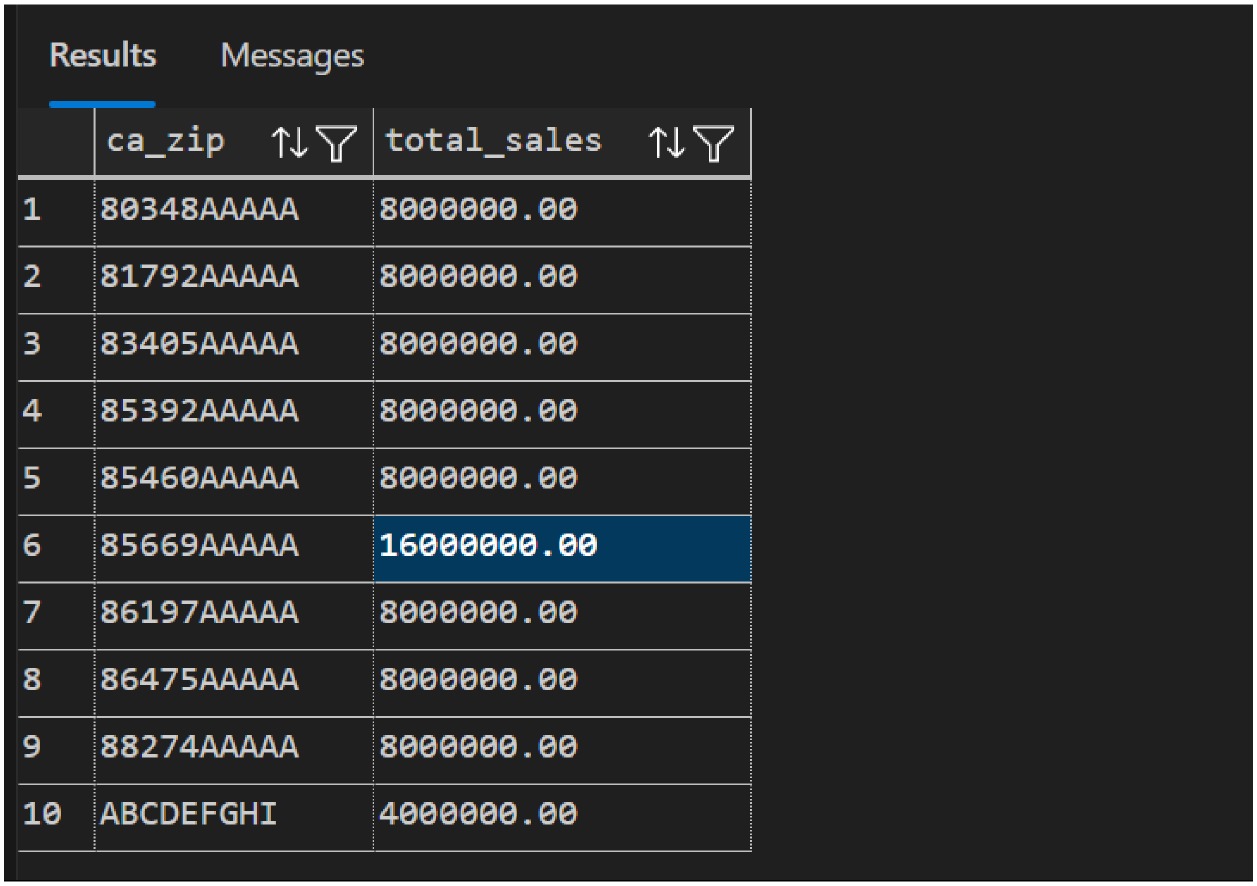
After Data Generation:

Original Query Output:



	ca_zip	(No column name)
1	80348AAAAA	8000000.00
2	81792AAAAA	8000000.00
3	83405AAAAA	8000000.00
4	85392AAAAA	8000000.00
5	85460AAAAA	8000000.00
6	85669AAAAA	20000000.00
7	86197AAAAA	8000000.00
8	86475AAAAA	8000000.00
9	88274AAAAA	8000000.00
10	ABCDEFGHI	4000000.00

Rewritten Query Output:



	ca_zip	total_sales
1	80348AAAAA	8000000.00
2	81792AAAAA	8000000.00
3	83405AAAAA	8000000.00
4	85392AAAAA	8000000.00
5	85460AAAAA	8000000.00
6	85669AAAAA	16000000.00
7	86197AAAAA	8000000.00
8	86475AAAAA	8000000.00
9	88274AAAAA	8000000.00
10	ABCDEFGHI	4000000.00

Discussion:
This case demonstrates a semantic difference due to an IN clause mutation. The original query uses a filter on ZIP code prefixes through a '$SUBSTRING(ca\_zip, 1, 5)$' condition with a specified list of values. The rewritten query omits this condition and retains only the state and sales price filters.

To detect this mutation, we generated customer address records with ZIP codes that satisfy the original query's substring condition but not the rewritten version's logic. For example, we inserted values like '85669AAAAA' and '86197AAAAA', which fulfill the substring match but are ignored by the rewritten query. Additionally, we injected one ZIP code value not present in the original list:

```
INSERT INTO customer_address(ca_address_sk, ca_address_id, ca_state, ca_zip)
VALUES (0, 'A', 'CA', 'ABCDEFGHI');
```

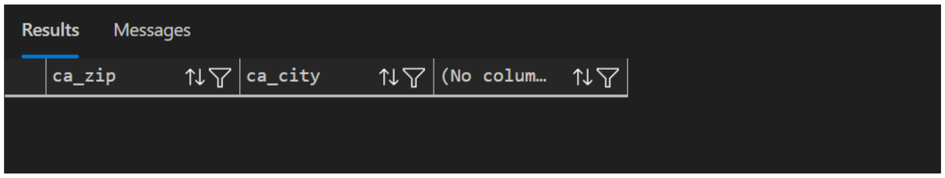
This caused the rewritten query to exclude rows that the original query includes, leading to lower aggregated totals. The difference in output confirms that the removal of the ZIP-based filtering logic introduces a semantic discrepancy. Therefore, the mutant is **killed**, demonstrating that our data generation approach effectively reveals the divergence between original and rewritten queries.

Case 3: Q45 – Complex IN + Nested Subquery Mutation

Original Query

```
SELECT TOP 100 ca_zip,
               ca_city,
               SUM(ws_sales_price)
FROM   web_sales, customer, customer_address, date_dim, item
WHERE  ws_bill_customer_sk = c_customer_sk
  AND  c_current_addr_sk = ca_address_sk
  AND  ws_item_sk = i_item_sk
  AND  (SUBSTRING(ca_zip, 1, 5) IN
           ('85669', '86197', '88274', '83405',
            '86475', '85392', '85460', '80348', '81792')
        OR i_item_id IN
           (SELECT i_item_id
            FROM item
            WHERE i_item_sk IN (2, 3, 5, 7, 11, 13, 17, 19, 23, 29)))
  AND  ws_sold_date_sk = d_date_sk
  AND  d_qoy = 1
  AND  d_year = 2000
GROUP BY ca_zip, ca_city
ORDER BY ca_zip, ca_city;
```

Initial Output of Original Query (Before Data Generation)

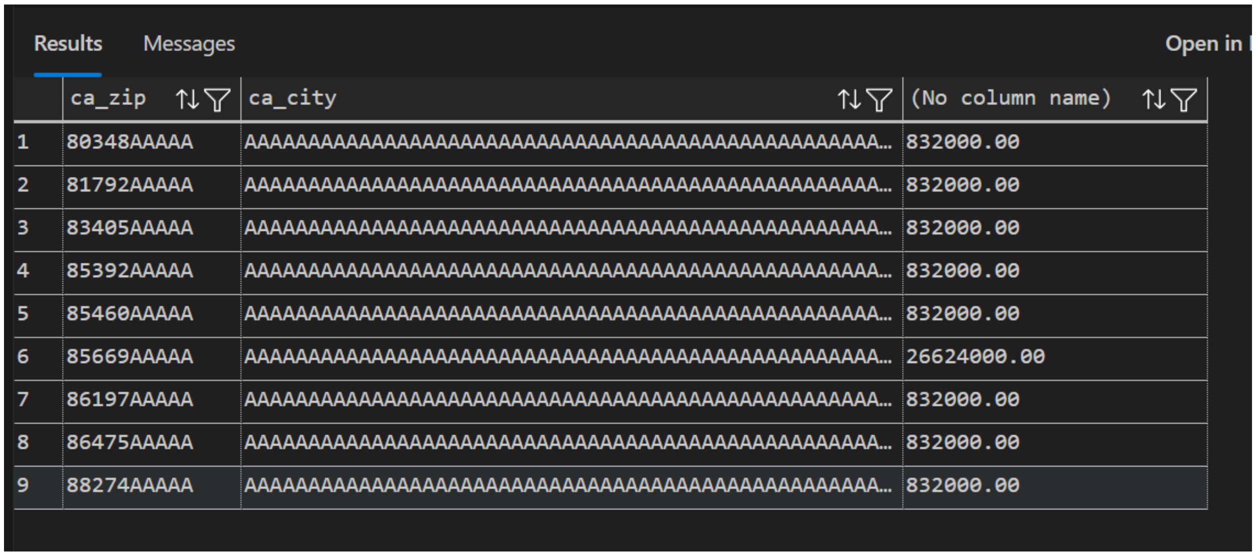


```
ca_zip | ca_city | (No colum...
```

Rewritten Query (Mutated)

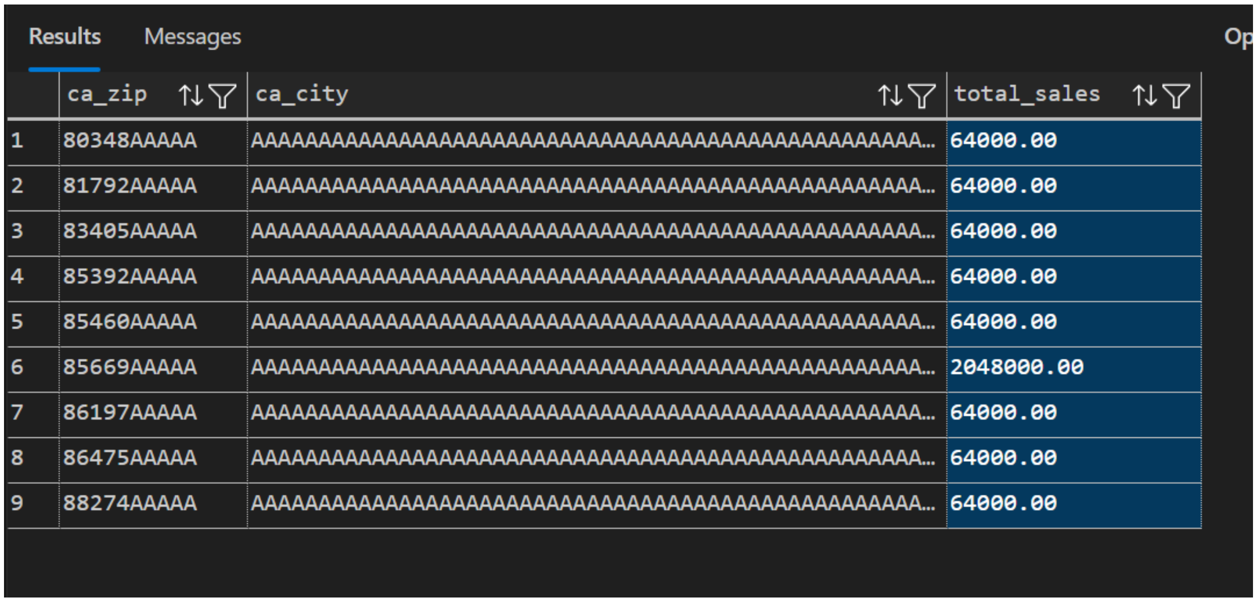
```
WITH filtered_web_sales AS (
  SELECT * FROM web_sales
  WHERE ws_item_sk IN (
    SELECT i_item_sk FROM item
    WHERE i_item_id IN (
      SELECT i_item_id FROM item
      WHERE i_item_sk IN (2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
    )
  )
)
SELECT TOP 100 ca_zip, ca_city, SUM(ws_sales_price) AS total_sales
FROM filtered_web_sales fws
JOIN customer c ON fws.ws_bill_customer_sk = c.c_customer_sk
JOIN customer_address ca ON c.c_current_addr_sk = ca.ca_address_sk
JOIN date_dim d ON fws.ws_sold_date_sk = d.d_date_sk
WHERE (LEFT(ca.ca_zip, 5) IN
         ('85669', '86197', '88274', '83405',
          '86475', '85392', '85460', '80348', '81792')
       OR fws.ws_item_sk IN (
         SELECT i_item_sk
         FROM item
         WHERE i_item_id IN (
           SELECT i_item_id
           FROM item
           WHERE i_item_sk IN (2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
         )
       ))
  AND d.d_qoy = 1
  AND d.d_year = 2000
GROUP BY ca_zip, ca_city
ORDER BY ca_zip, ca_city;
```

Output After Data Generation
Original Query Output:



	ca_zip	ca_city	(No column name)
1	80348AAAAA	AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...	832000.00
2	81792AAAAA	AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...	832000.00
3	83405AAAAA	AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...	832000.00
4	85392AAAAA	AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...	832000.00
5	85460AAAAA	AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...	832000.00
6	85669AAAAA	AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...	26624000.00
7	86197AAAAA	AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...	832000.00
8	86475AAAAA	AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...	832000.00
9	88274AAAAA	AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...	832000.00

Rewritten Query Output:



Results Messages

	ca_zip	ca_city	total_sales
1	80348AAAAA	AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA…	64000.00
2	81792AAAAA	AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA…	64000.00
3	83405AAAAA	AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA…	64000.00
4	85392AAAAA	AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA…	64000.00
5	85460AAAAA	AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA…	64000.00
6	85669AAAAA	AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA…	2048000.00
7	86197AAAAA	AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA…	64000.00
8	86475AAAAA	AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA…	64000.00
9	88274AAAAA	AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA…	64000.00

Discussion:
This case reveals the impact of a structural mutation involving nested subqueries and IN clauses. The original query evaluates multiple conditions directly within the `WHERE` clause using a flexible OR logic — allowing a match either through ZIP prefix or through item ID-based subquery. In contrast, the rewritten query first restricts the `web_sales` table using a deeply nested item ID resolution inside a CTE, filtering out many rows upfront.

Our logic includes values such as `ws_item_sk = 2` and `i_item_id = 'AAAAAAAAAAAAAAAA'` to satisfy the original query's broad filter conditions. However, due to the additional indirection in the rewritten query, these rows are not retained in the filtered `web_sales` set. This mismatch demonstrates that although the rewritten query appears logically equivalent, the structural changes result in reduced output. Thus, the test dataset successfully **kills the mutant**, highlighting the semantic deviation introduced by the rewrite.

5 Conclusion

In this project, we presented a systematic approach to evaluate the semantic correctness of SQL query rewrites generated by large language models (LLMs). By using the XData framework and symbolic constraint solving via Z3, we designed and implemented a pipeline that can automatically generate datasets to test for query equivalence. We demonstrated our methodology across multiple real query pairs involving non-empty results, simple and complex `IN` clause mutations, and nested subqueries. Our experiments showed that even small logical differences introduced by LLMs such as missing filters or transformed join paths can lead to substantial semantic deviations in the results. The data-driven nature of our approach allowed us to construct targeted examples that revealed these deviations and enabled us to "kill" the mutant queries effectively. This highlights the importance of rigorous semantic validation when using automated systems for query transformation.

It is important to note, however, that the absence of differences does not prove equivalence; if a mutant is not killed, it may simply mean that the dataset did not trigger the discrepancy. Our method is effective at quickly catching incorrect rewrites, but occasional false negatives or undetected issues may still occur.

6 Future Work

Several directions remain for extending this work. First, we plan to support additional types of mutations including those involving arithmetic and logical operators (e.g., `>`, `<`, `AND`, `OR`), join conditions (e.g., inner or outer joins), and more complex query constructs such as CTEs (Common Table Expressions). While our current implementation works effectively for mutations that can be triggered by generating one tuple at a time, this approach falls short for queries that inherently require multiple tuples to produce meaningful results—such as those using `RANK`, `ROW_NUMBER`, or other window functions. In such cases, a fundamentally different data generation strategy would be needed—possibly involving bulk generation of correlated records or constraint solvers that can reason about row-level interactions.

References

[1] S. Dharwada et al., "Query Rewriting via LLMs," arXiv:2403.20325, 2024.

[2] B. Gupta et al., "XData: Generating Test Data for Killing SQL Mutants," ICDE, 2010.

[3] W. Gatterbauer, S. Khanna, S. Roy, and Y. Tian, "Query-Based Data Pricing," *Data Generation for Testing and Grading SQL Queries*, arXiv:1411.6704v5, 2015.

[4] L. Mendonça de Moura and N. S. Bjørner, "Z3: An Efficient SMT Solver," *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science, Vol. 4963, Springer, 2008.

[5] S. Somwase et al., "Data Generation for Testing Complex Queries," arXiv:2409.18821, 2024.