Indian Institute of Science, Bengaluru

Proving Query Equivalence using LLMs

Authors

Suhas Kamath and Anish Tiwari

suhaskamath@iisc.ac.in
anishtiwari@iisc.ac.in

Project Advisor:
Dr. Harish Doraiswamy
Principal Researcher
Microsoft Research

Course Instructor and Advisor:
Prof. Jayant R Haritsa
Database Systems Lab (DSL)
Department of Computer Science and Automation
Department of Computational and Data Science

December 4, 2025

Contents

1 Introduction

This project aims at solving the problem which has been attempted for decades. Researchers across the world have tried to prove that two queries are semantically equivalent. This means that the two queries must return the same result on any valid database instance.

As we know, LLMs have improved to a point where they are able to solve complex problems due to the sheer weight of the dataset that they have been trained on. Hence, we can attempt to use LLMs to take a stab at this arduous task.

Once we get an answer from the LLM, we need to check the validity of the answer. As has been apparent at this point, LLMs have a habit of hallucinating and producing errors. [5] To verify the answer, we make use of a software called Apache Calcite.

1.1 Software

In this section, we describe the various software components that we have used.

1.1.1 LITHE

LITHE is a software that leverages the power of LLMs to improve query performance. Essentially, it works by rewriting a query to another equivalent query by leveraging the remarkable cognitive ability of LLMs. [2]

This software is used to optimise queries. For example, this software has the capability to use LLMs to flatten a nested query into an equivalent normal query. As we know, a lot of query optimisers are unable to see that these nested queries need not be nested. LITHE has the capability to see this, and fix it into a normal query, which shall execute much faster.

We shall use this software to test our program. This software is capable of completely changing the syntax of the query, while still keeping it equivalent to the original query. This shall be helpful in testing our program. We shall use a set of original queries, which are translated using LITHE to equivalent queries, so as to check if our program is able to equate the two queries.

1.1.2 ChatGPT

ChatGPT is a Large Language Model developed by OpenAI. The first public model, GPT-3 was released in 2022. [8] The most recent version of ChatGPT, GPT-5 was released on 7th August 2025. In this project, we have used this particular model.

Since its release, ChatGPT has demonstrated its various abilities, especially in the aspect of software engineering and programming. [4] [6] Because ChatGPT can reasonably answer programming-related questions, such as debugging a program, or checking the correctness of a program, we should believe, by induction, that it should be able to prove query equivalence also.

1.1.3 PostgreSQL

PostgreSQL is a very popular relational database management system (RDBMS) developed at the University of California, Berkley. [11] It has all of the features that we require to develop this project.

For starters, the RDBMS software is free and open source. Furthermore, it has the some very helpful commands, such as `EXPLAIN PLAN`, which has the capability to return the query plan for a particular query. The software returns a response in `JSON` format. As the command suggests, the entire query plan is returned, which depicts the step-by-step operations that shall be performed by the database engine to execute the given query.

1.1.4 Apache Calcite

Apache Calcite is an open source software for data management systems. It includes a SQL parser, and more importantly, has a feature to convert SQL queries to relational algebra. Furthermore, we can also apply various transformations to this relational algebra. [1] The two queries can be converted to relational algebra, transformed, and then be compared to check if they are equivalent. The transformations, however, need to be supplied manually to Calcite. In this project, we use ChatGPT to provide these transformations.

1.2 Basic Concepts

Some of the basic concepts that shall be used extensively in this report are:

1. **Query Plan:** When a query is provided to a database system, such as PostgreSQL, it will generate a plan on how to execute the query.

2. **Calcite Relational Node:** When a query is given to Calcite, it shall convert it to relational algebra tree. In the programming context, relational nodes are known as `RelNode`.

3. **Calcite Transformations:** These transformations are used to modify the structure of a `RelNode`. In this case, the transformations are provided by the LLM.

2 Implementation

Since Apache Calcite is built upon Java, we decided that Java is the best programming language to use for this application. We could have used the Calcite JAR file with Python using Python libraries such as `javabridge`, but eventually decided against it. This helps in maintaining the homogenous nature of the program, and makes it easier for debugging.

Furthermore, Java is a capable programming language, with inbuilt support for connecting to databases, i.e. Java DataBase Connectivity (JDBC). Also, OpenAI has a Maven

dependency, which can effortlessly integrate calls to the LLM from the program. The general flowchart of the program is as follows:

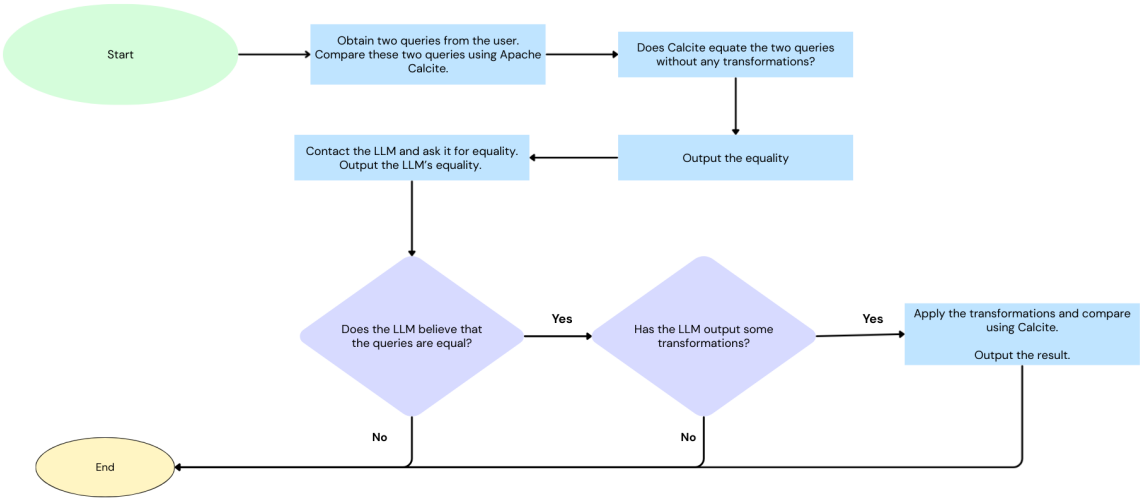

Figure 1: Overall flowchart of the program.

Note: A lot of the minute implementation details have been skipped in this diagram. The program in reality is much more complex in comparison to this image.

We can also propose an alternative approach to the flow of the program. This alternative flowchart decides to deterministically check the queries first, before consulting the LLM. In some cases (for small changes in queries, where query B is minutely different to query A), Apache Calcite combined with our heuristic-based logic, can decide deterministically equate the queries. One example of such a query is when the order of the predicates is swapped. This has several advantages:

1. If the LLM is contacted less frequently, the LLM tokens will be saved, thereby making the execution of the program more economical.

2. The LLM usually takes around a minute per query to give its verdict, but Calcite executes in a matter of seconds. This alternative program flow can help in making the program execute faster.

3. Provides a definitive, deterministic answer.

This alternative program flow can be visualised using the following flowchart:

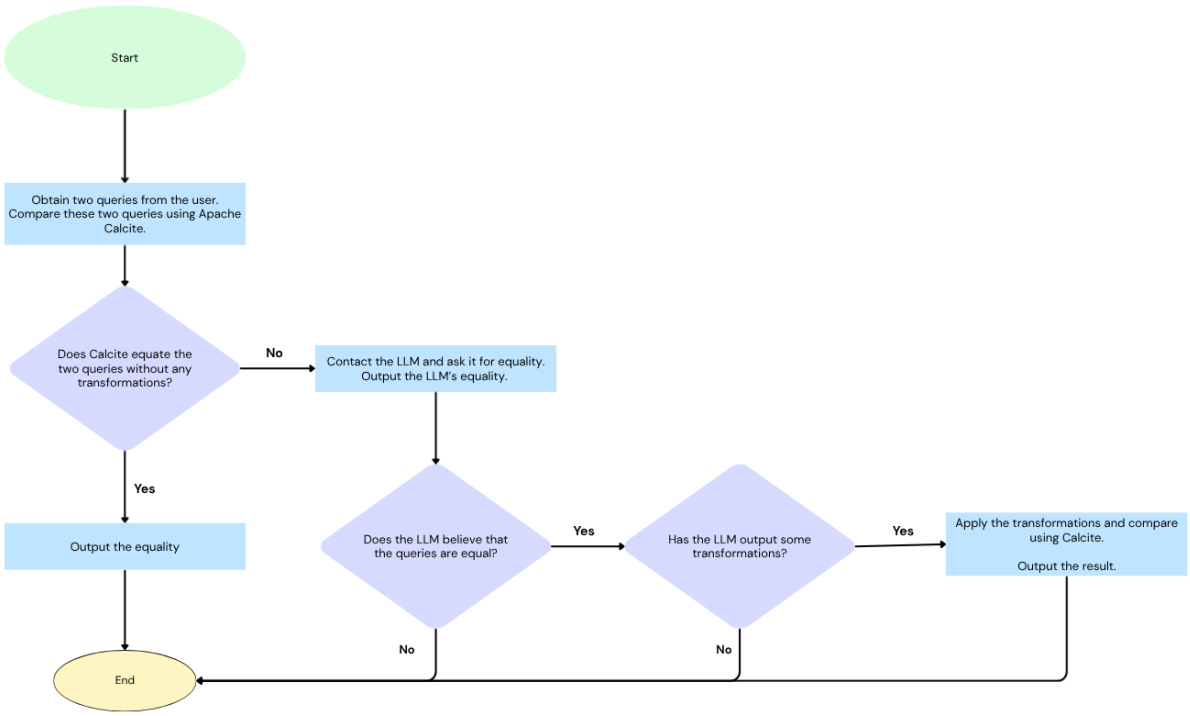

Figure 2: Alternative flowchart of the program.

2.1 How are queries compared?

The comparison is done using the method `compareQueries()`. This function compares queries step-by-step. We use Apache Calcite to compare two SQL queries. The first step is to "clean" the query by removing extra white space and removing additional semi-colons.

Then, the queries are converted to `RelNode` objects. These are basically relational query trees. For this, a connection to the database is required, so that we can obtain the `FrameworkConfig`, which educates the Calcite API on the schema of the database. Once the `FrameworkConfig` is obtained, we convert the SQL query to a `RelNode`.

The `RelNode` of query A can then be compared with the `RelNode` of query B. As advised by the LLM, we can then apply some transformations on `RelNode A`. If no transformations are specified, we do not apply any transformations. However, in some cases, it is possible that no transformations are required to equate the two queries. We handle this as well.

Furthermore, some Apache Calcite transformations tend to crash the program, which is why we donot support all of the transformations that are supposed to be available via the Apache Calcite API. There are 96 transformations that can be used, without risking the stability of the program. Some of the most popular transformations are as follows:

- **AggregateProjectMergeRule**: Merges a Project operator on top of an Aggregate into the Aggregate itself.

- **AggregateReduceFunctionsRule**: Rewrites complex aggregate functions into simpler primitives (e.g., AVG to SUM/COUNT).
- **FilterAggregateTransposeRule**: Pushes a Filter operator past an Aggregate operator.
- **FilterMergeRule**: Combines two consecutive LogicalFilter operators into one.
- **FilterProjectTransposeRule**: Pushes a Filter operator past a Project operator.
- **JoinAssociateRule**: Reorders a join tree based on the associativity rule.
- **ProjectJoinTransposeRule**: Pushes a Project past a Join by splitting the projection into a projection on top of each child.

Once the transformations are applied, we normalise the subqueries and then attempt to equate them using heuristics. However, if none of these heuristics work, we convert the Calcite `RelNode` to a `RelNodeTree`. `RelNodeTree` is a custom class that represents a `RelNode` in a custom tree structure, so that we can "play" with it efficiently.

Again, we have a custom implementation to compare the two trees. This is done by comparing the two trees, but we ignore the child order. This ensures that subqueries are handled, even if their orders are interchanged.

Warning: Apache Calcite has its own methods to compare two `RelNode` objects. These are the `RelNode.equals()` and `RelNode.deepEquals()`. These methods instantly cause a stack overflow, thereby crashing the program. Hence, we do not use these functions.

The detailed comparison steps followed by the `compareQueries()` function (after transformations have been applied to `RelNode A`) are as follows:

1. **Normalisation and Decorrelation**
   (a) *Normalisation*: We attempt to apply some planner rules to simplify the `RelNode`, by merging redundant projects.
   (b) *Decorrelation*: We attempt to 'flatten' any sub-queries that may be present. If we are unable to do so, we do not modify the plan.
2. **Comparison Attempt**: After the above normalisation and decorrelation are performed, we attempt to convert the `RelNode` objects into a String, and perform a simple String comparison. This comparison is sensitive to order, and requires the `RelNode` objects to be exactly identical to each other.
3. **Digest Normalisation**: We stabilise the digest[1] by neutralising the field positions

[1]A digest is a textual represententation of an Apache Calcite relational plan. It basically captures the structure of a `RelNode` in a stable, comparable String format.

and spacing, such that comparing structurally equivalent plans result in equality.

4. **Comparison Attempt**: We attempt to equate the two queries after the above digest normalisation is performed.

5. **Digest Canonicalisation**: We know that inner joins can be re-ordered, and still result in the same output. Hence, this step ensures that the order of inner joins does not affect the equality result. During this step, we care to ensure that the projection operators are not affected.

6. **Comparison Attempt**: We attempt to equate the two queries after the above digest canonicalisation is performed.

7. **Final Attempt**: In the last attempt, as described above, we attempt to translate the entire `RelNode` into a `RelTreeNode` object, and attempt to perform a tree comparison.

2.2 How is the LLM contacted?

As mentioned above, we contact the LLM by means of an API call. First, we obtain the query plans by means of a call to the database. We use the `EXPLAIN JSON` method to obtain the query plan. However, the query plan is usually riddled with plenty of useless information, such as the expected number of rows, etc., as output by the query optimiser. Because this is implementation-specific, we remove such fields so as not to confuse the LLM. Furthermore, we will also save on tokens. Most API calls are charged by the token, which almost directly corresponds to the length of the prompt.

We then construct the prompt using the `StringBuilder` class, which is a String concatenation method provided by Java. The LLM API call then returns its verdict with respect to the equivalence of the queries, and the transformations that should map from query A to query B in JSON format.

Once we obtain the response from the LLM, we parse into an `LLMResponse` object. An `LLMResponse` object contains a boolean value (to denote the equality as decided by the LLM, and a list of strings, that contain the transformations output by the LLM. We use `LLMResponse` constructors to parse the `JSON` response provided by the LLM. During construction of the object, we also verify that the transfomations output by the LLM are actually valid and supported. [2]

2.2.1 Prompt Engineering

Large Language Models are sensitive to how prompts are written. Their output can change significantly depending on the prompt structure, especially when the input is large. In our case, some query plans can reach several hundred lines, which increases the instability of the model's responses. Because our system depends on the LLM to (1) decide whether two query plans are likely to be equivalent and (2) output a valid

[2]During testing, we have observed that the LLM has a habit of "hallucinating", which is when it outputs transformations that do not exist. We ensure that such transformations are ignored.

sequence of Apache Calcite transformations, the response must be both stable and strictly machine-readable.

For the system to work reliably, the LLM must follow a fixed JSON format so that our program can parse the output using simple string-matching logic. In addition, the transformation names returned by the LLM must exactly match the list of supported transformations recognised by Apache Calcite. During testing, we observed occasional hallucinations, where the LLM produced transformation names that do not exist. These were filtered out, but good prompt design reduced such mistakes.

We evaluated both zero-shot prompting (no example provided) and one-shot prompting (a single example showing the required input–output structure). Zero-shot prompting led to more inconsistent outputs for longer plans. One-shot prompting improved stability, formatting consistency, and reduced hallucinated transformation names by giving the LLM a concrete pattern to follow.

Based on these observations, we designed a structured prompt that: (i) states the task clearly, (ii) enforces a strict JSON schema, (iii) specifies the expected plan format, (iv) includes the complete list of supported transformations, (v) incorporates a one-shot example, and (vi) finally appends the ORIGINAL and TARGET plan JSON objects. This structure produced the most reliable results.

Furthermore, we also include the schema structure of the database in the prompt. We found that attaching the schema structure ensure that the LLM has full knowledge of the joins, indexes and constraints (such as foreign keys). We found that this stabilises the output of the LLM further.

The final prompt that provided us with the best results is as follows:

```
System Message:
You are an expert in relational query optimization and Apache Calcite
 ↪  transformations. Always respond with exactly one JSON object
 ↪  matching the schema requested. Do not output anything else. Use only
 ↪  transformation names from SUPPORTED_TRANSFORMATIONS. Temperature
 ↪  must be 0.

TASK:
Given ORIGINAL_PLAN and TARGET_PLAN (both in simplified JSON plan
 ↪  format), decide whether they are equivalent. If equivalent, produce
 ↪  an ordered list of Apache Calcite transformations (from
 ↪  SUPPORTED_TRANSFORMATIONS) that map ORIGINAL_PLAN -> TARGET_PLAN.

INPUTS PROVIDED:
1) SCHEMA_SUMMARY: a compact JSON describing tables and primary/foreign
 ↪  keys. (Use the schema summary only for reasoning about join keys and
 ↪  uniqueness.)
2) SUPPORTED_TRANSFORMATIONS: <full list of valid transformations>
RESPONSE SCHEMA (MUST return exactly this JSON object):
```

```
{
"equivalent": <"true" | "false" | "dont_know">,
"transformations": [ <ordered list of exact transformation names from
↪  SUPPORTED_TRANSFORMATIONS> ],
"preconditions": [ <objects describing required preconditions for each
↪  transformation, in same order> ]
}

RULES:
1) If plans are identical, return "equivalent":"true",
↪  "transformations":[], and "preconditions": [].
2) Only use names from SUPPORTED_TRANSFORMATIONS. If none apply, return
↪  "equivalent":"dont_know" with empty lists.
3) For each transformation listed, provide a corresponding precondition
↪  object describing the minimal, strictly necessary condition (e.g.,
↪  "node X is Project on top of Aggregate", or "filter predicate P
↪  exists on child", "join keys (A = B) exist"). Keep preconditions
↪  concise.
4) Do NOT invent any transformation names. If unsure, return
↪  "dont_know".
5) Output NOTHING but the required JSON object.

ONE-SHOT EXAMPLE: <one shot example>

NOW PROCESS:
SCHEMA_SUMMARY: <schema summary>
ORIGINAL_PLAN_JSON: <query plan A>
TARGET_PLAN_JSON: <query plan B>
```

The above prompt is an example of rule-based role prompting, wherein we ask get the LLM to play a 'role' of a expert in a particular field. Along with the 'role', we also give the LLM 'rules' that it strictly needs to follow. Such a method of prompting has shown to be the most effective, especially in such fields, which require a lot of knowledge on the subject. [9]

3 Results

In this section, we shall discuss the results that we have obtained during our testing:

3.1 Equality

First, let us explore the observations made with regards to the equivalence factor. During testing, we observed the following results:

Some notes on the testing:

- The original set of queries were rewritten into new queries by means of LITHE.

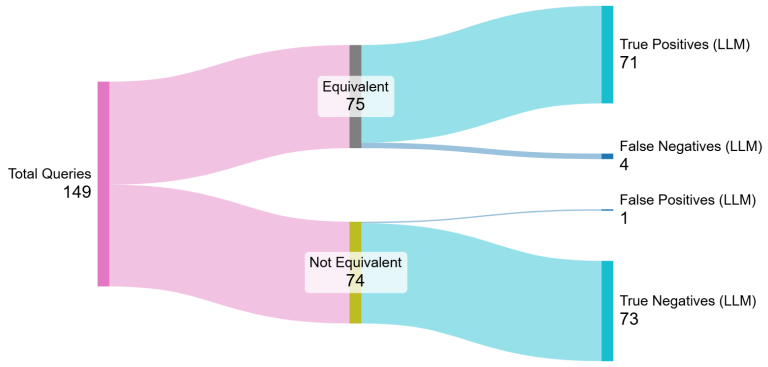

Figure 3: Sankey diagram depicting the results.

[2] This ensures entire structure of the query is changed.

- The mutated (unequivalent) queries were rewritten from the original set manually (by hand). During this process, we have attempted to keep the structure of the SQL as similar as possible to the original structure.

We observe that the LLM provided the correct equality 96.6% of the times. We can clearly see that the LLM is now able to provide a good estimate for the equality.

These results were acquired by means of the current program flow, which attempts to check both deterministically and with the LLM. Hence, we have made a total of 149 LLM API calls (one for every pair of queries). If we used the alternative program flow, the following data would have been obtained (for the same queries):

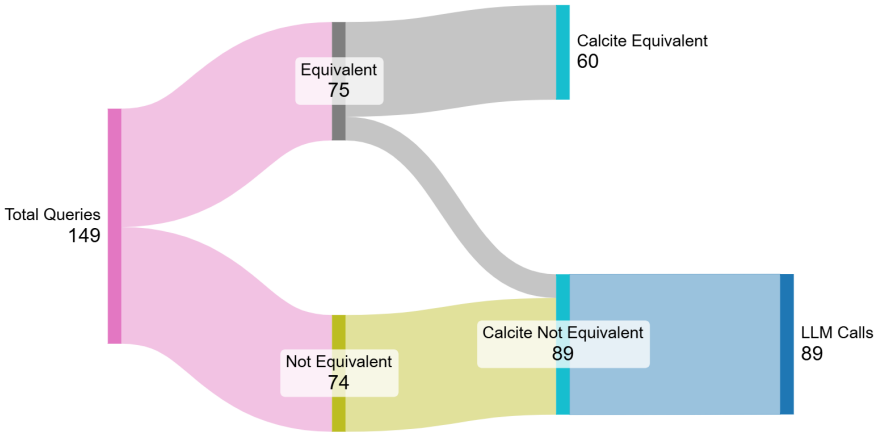

Figure 4: Sankey diagram depicting the reduction in LLM API calls.

As can be seen in the diagram, we reduce the LLM API calls by about 40%. This helps in reducing a significant amount of cost and time.

3.2 Transformations

In this subsection, we shall explore the transformations output by the LLM. If we recall, we also ask the LLM to output the Apache Calcite transformations required to map from query A to query B. This proves to be the more arduous task for the LLM, as compared to the equivalence checking.

We observed the following:

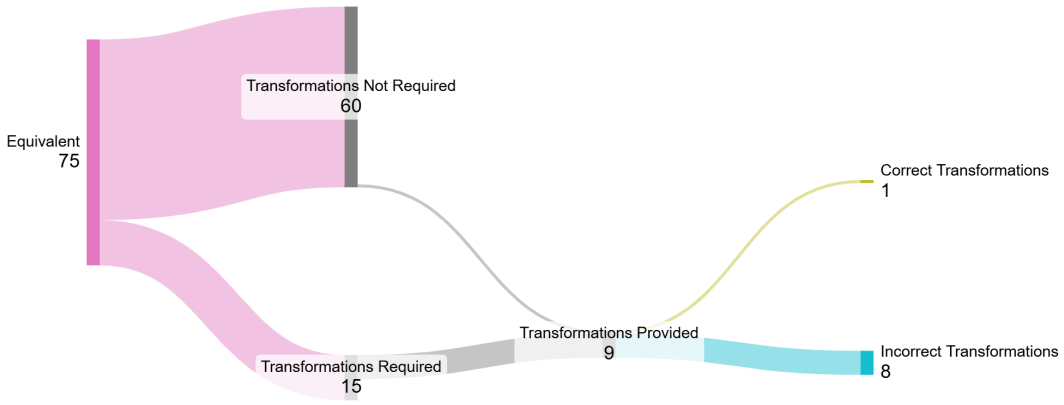

Figure 5: Sankey diagram depicting transformations.

During testing, we observed that the equivalence factor is relatively stable. In other words, the LLM outputs arrive at the same decision of equivalence during multiple runs of the program. However, the same cannot be said for the transformations. The transformations, on the other hand are highly unstable and tend to change (albeit slightly) every run. We also note that the LLM was not able to provide the correct Apache Calcite transformations for most of the queries.

A possible reason for these anomalies may be that the LLM is unfamiliar with the meaning (or effect) of each Apache Calcite transformation and hence, outputs what it feels is correct. It may make this decision based on the literal name of the transformation. The literal names of the transformations may not depict the effect that a particular transformation has on the `RelNode`. There are very few reference materials pertaining to applying transformations to `RelNode` objects and then comparing them using Apache Calcite. Hence, the LLM is unsure as to which transformations are correct.

Another possibility is that the LLM simply is not able to provide the correct order of the transformations. The LLM may be familiar with the effect of the transformations, and may be able to decide which transformations are correct. However, it is unable to provide the correct order of the transformations.

3.3 TPCH Queries

We know that the TPCH queries are the industry standard for database testing. Hence, we have also used all 22 of the TPCH queries for testing.

The following results were obtained during TPCH testing. All rewrites were done using LITHE.

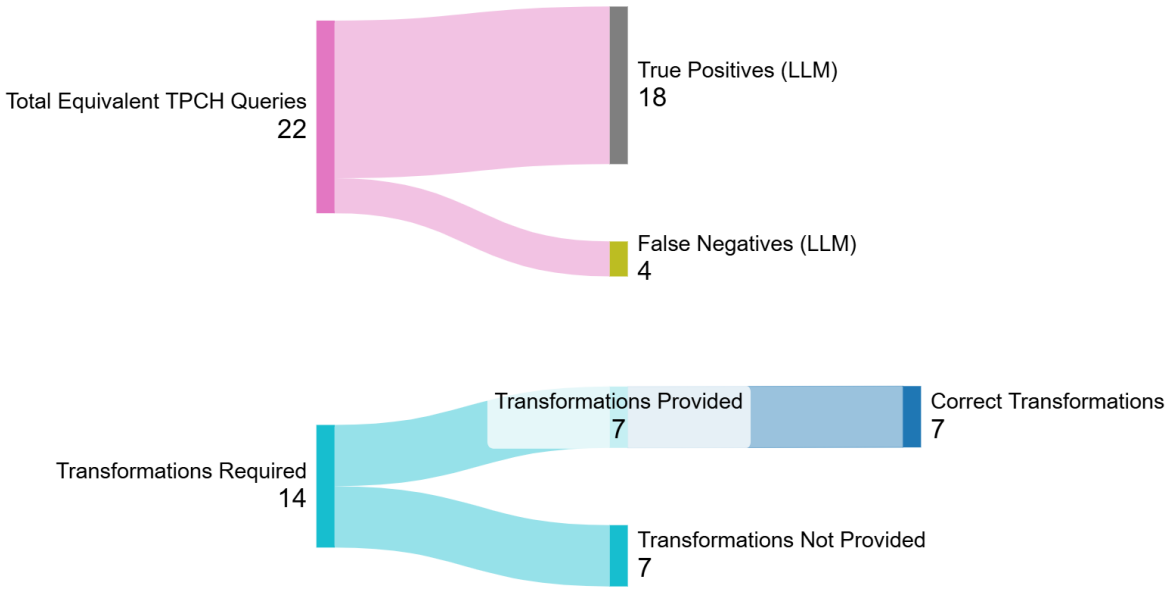

Figure 6: Test results on TPCH queries.

We can see that due to the complexity of these queries, the accuracy drops to a fair score of 81%. However, the transformations provided by the LLM are still wrong[3].

4 Future Work

4.1 Investigation into Transformation Rules

The reason for which the LLM is not providing the correct transformations have been speculated upon, but not verified. For all we know, the reason may be something totally different than what we assumed. We can investigate this further and try to find the proper reason for these anomalies. Once we figure out the reason, we can then attempt to fix those, either by prompt engineering, or other means.

During our testing, we have tried multiple methods to investigate this anomaly, but were unable to come up with a satisfying conclusion so as to why the transformations are wrong.

4.2 Integration of Better Query Equivalence Checkers

As of right now, we are depending on a heuristic-based system built upon Apache Calcite to deterministically check for query equivalence. However, we know that there

[3]Again, by 'wrong' we mean that either the order of transformations is wrong, or the transformations themselves are wrong.

exist more sophisticated tools, such as SQLSolver that can deterministically equate a wider variety of queries in comparison with our implementation based on Apache Calcite. [3] This helps in reducing our dependence on LLMs, and makes the program overall more deterministic (and help it in executing faster, while saving on cost).

However, SQLSolver is unable to equate several kinds of queries deterministically (or provably), where LLMs shall come in to fill the gap with their unprecedented knowledge. One example of this kind of query is lateral join queries, which allows a subquery to use columns of a relation referenced earlier in the FROM clause of the query. An example of this kind of query is:

```sql
SELECT c.category_name, p.title, p.views
FROM categories AS c
LEFT JOIN LATERAL (
    SELECT title, views
    FROM posts
    WHERE category_id = c.id
    ORDER BY views DESC
    LIMIT 2
) AS p ON true
ORDER BY c.category_name ASC, p.views DESC;
```

5 Conclusion

As can be seen in the above results, GPT-5 is very accurate at proving semantic equality of predicates. However, GPT-5 is unable to prove this. This leaves a major gap in the workflow. Albeit rare[4], GPT-5 is prone to equate unequal queries.

Furthermore, we have explored the vast capability that LLMs have, and how much they can help to solve computational problems.

[4] In our testing with 74 unequal queries, this happened only once.

References

[1] Begoli, E., Camacho-Rodríguez, J., Hyde, J., Mior, M. J., and Lemire, D. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data* (May 2018), SIGMOD/PODS '18, ACM, p. 221–230.

[2] Dharwada, S., Devrani, H., Haritsa, J., and Doraiswamy, H. Query Rewriting via LLMs. In *Proceedings of 29th International Conference on Extending DataBase Technology (EDBT), Tampere, Finland, March 2026* (2025), ACM.

[3] Ding, H., Wang, Z., Yang, Y., Zhang, D., Xu, Z., Chen, H., Piskac, R., and Li, J. Proving query equivalence using linear integer arithmetic. In *Proceedings of the ACM on Management of Data, Volume 1, Issue 4* (New York, NY, USA, Dec. 2023), vol. 1, Association for Computing Machinery.

[4] Kabir, S., Udo-Imeh, D. N., Kou, B., and Zhang, T. Is Stack Overflow Obsolete? An Empirical Study of the Characteristics of ChatGPT Answers to Stack Overflow Questions, 2024.

[5] Kalai, A. T., Nachum, O., Vempala, S. S., and Zhang, E. Why Language Models Hallucinate, 2025.

[6] Kashefi, A., and Mukerji, T. ChatGPT for Programming Numerical Methods, 2023.

[7] OpenAI. ChatGPT — Release Notes, 2023.

[8] OpenAI. GPT-4 Technical Report, 2023.

[9] Ruangtanusak, S., Taveekitworachai, P., and Pipatanakul, K. Talk less, call right: Enhancing role-play llm agents with automatic prompt optimization and role prompting, 2025.

[10] Schulhoff, S., Ilie, M., Balepur, N., Kahadze, K., Liu, A., Si, C., Li, Y., Gupta, A., Han, H., Schulhoff, S., Dulepet, P. S., Vidyadhara, S., Ki, D., Agrawal, S., Pham, C., Kroiz, G., Li, F., Tao, H., Srivastava, A., Costa, H. D., Gupta, S., Rogers, M. L., Goncearenco, I., Sarli, G., Galynker, I., Peskoff, D., Carpuat, M., White, J., Anadkat, S., Hoyle, A., and Resnik, P. The prompt report: A systematic survey of prompt engineering techniques, 2025.

[11] Stonebraker, M., Rowe, L., and Hirohama, M. The Implementation Of Postgres. *Knowledge and Data Engineering, IEEE Transactions on 2* (04 1990), 125 – 142.