Heuristic-Guided Text-to-SQL Translation with LLMs: Optimizing Natural Language Interfaces for Relational Databases *

Laura Petrola[1], Angelo Brayner[2], Wellington Franco[3]

[1]Mestrado e Doutorado em Ciências da Computação – Universidade Federal do Ceará (UFC)

laupetrola@gmail.com, brayner@dc.ufc.br, wellington@crateus.ufc.br

Abstract. *Text-to-SQL mapping process plays a crucial role in enabling non-technical users to interact with relational databases using natural language. While Large Language Models (LLMs) have shown promising results in benchmark datasets, their performance in real-world settings often deteriorates. In this work, we propose a modular and adaptive agent that leverages the open-source LLM DeepSeek to perform translations with integrated feedback and query optimization. Our agent is prompt-engineered to refine query generation based on user or system-provided corrections. Using the TPC-H and Mondial benchmarks, as well as a real-world database, we demonstrate improved accuracy and execution efficiency, highlighting the impact of feedback loops and heuristic-based query rewriting.*

1. Introduction

The growing interest in using LLMs as facilitators for Text-to-SQL tasks has been a hot topic over the last few years, particularly following the rise of ChatGPT [OpenAI 2025]. These models have shown remarkable capabilities in understanding and generating natural language, which naturally extends to applications involving the translation of user questions into structured queries over relational databases. This task, commonly referred to as Text-to-SQL, holds substantial promise for democratizing data access by enabling non-technical users to query databases without the need to learn SQL.

Recent surveys [Zhu et al. 2024, Hong et al. 2024] have provided comprehensive overviews of the current landscape in LLM-based Text-to-SQL systems, classifying methodologies based on prompt engineering, fine-tuning, and agent-based strategies. These works emphasize not only the technical potential of LLMs to bridge the gap between natural and structured database language but also reveal limitations such as linking natural language words to database objects in a schema (schema linking problem) and building syntactically and semantically correct SQL queries. Even when a large language model (LLM) generates a correct query, it often fails to consider performance-related aspects, for instance, query execution time.

Nonetheless, various challenges inherent to real-world applications render the widespread adoption and effective use of LLMs in Text-to-SQL systems difficult. One of the main concerns is that in real-world database schemas, they are often large, complex, contain several objects (tables, indexes, and views), and are not necessarily well-designed. As highlighted by recent studies, including [Sala et al. 2024], LLMs often fail

*This work was partially funded by Fundação Nacional de Artes (Funarte), research grant TED 01/2024.

to generate correct or even executable queries when faced with non-trivial schema layouts and intricate user intents. This issue is compounded by the inherent ambiguity of natural language, which can lead to misinterpretations and incorrect mappings between user questions and database elements. For instance, terms used by users may not exactly match the column or table names in the schema, requiring sophisticated schema linking mechanisms to bridge this gap, a task that remains difficult even for advanced LLMs.

On the other hand, achieving low query response times has become a critical requirement. This is because database applications have become very complex, dealing with a huge volume of data and database objects. Among various strategies for improving query processing performance, SQL query refactoring has emerged as a key approach for minimizing query execution time. The primary objective of the query refactoring process is to rewrite an existing SQL query Q into a new query Q', such that both queries are semantically equivalent, i.e., they deliver the same result, while Q' has a lower execution time. In fact, SQL query refactoring optimizes joins, subqueries, and indexes to reduce execution time and resource usage.

In this sense, this paper proposes a novel approach to generate optimized SQL code by using DeepSeek [DeepSeek 2025], a LLM, in the context of the Text-to-SQL task. The model translates natural language queries into initial SQL queries. Then, a set of handcrafted SQL optimization heuristics is applied, to refactor query Q into an executable and optimized query Q', To illustrate potential performance gains, Figure 8 shows a refactored query that is eight times faster than the query delivered by DeepSeek.

We have implemented the proposed approach and evaluated its behavior in two benchmarks: the well-established TPC-H benchmark and a real-world public database, the MAPAS [Ministério da Cultura 2025] database. All experiments have been conducted in the PostgreSQL database system. The experimental results demonstrate that the approach is effective and may be suitably adopted by database professionals in practical scenarios.

The remainder of the paper is organized as follows. Section 2 provides foundational background concepts relevant to the paper. Section 3 reviews related work on SQL query rewrite optimization and recent advancements in Text-to-SQL systems using LLMs. Section 4 introduces the proposed agent architecture, detailing its heuristic-driven optimization mechanisms and semantic retrieval support. Section 5 presents experimental evaluations using the TPC-H and MAPAS databases, analyzing the effectiveness of DeepSeek and the impact of SQL rewriting heuristics. Finally, Section 6 concludes the work, summarizes key findings, and outlines future research directions.

2. Background

2.1. Large Language Models - LLMs

LLMs are machine learning models that use deep learning algorithms to process and understand natural language, that is, human language [Minaee et al. 2024]. These models are trained on vast amounts of textual data, enabling them to learn patterns and relationships among entities present in human-generated texts [Fan et al. 2020].

LLMs are capable of performing a wide range of linguistic tasks [Yang et al. 2024], including the comprehension of complex textual data, identification of entities and their

relationships, and the generation of coherent and grammatically accurate new texts [Zhang et al. 2024].

There are numerous potential applications for LLMs. They are frequently used for question answering, essay writing, text translation, document summarization, code generation in programming languages, among others [Ozdemir 2023]. Additionally, they are employed in chatbots, digital assistants, and various other applications that require text generation or comprehension [Hadi et al. 2023].

2.2. SQL Refactoring

SQL query code refactoring is the systematic rewriting of SQL queries in order to improve their efficiency and maintainability without altering their intended functionality or results [Farouult and L'Hermite 2008, Shasha and Bonnet 2003]. Thus, the refactored query must return the same results as the original.

Firstly, the refactoring process aims to reduce query execution time. Such a feature can be achieved by, for instance, rewriting inefficient joins or subqueries, and removing redundant conditions [de Araujo et al. 2013, Ramakrishnan and Gehrke 2002]. Intuitively, the idea is to give the query engine footprints to generate more efficient query execution plans. Secondly, refactoring improves the clarity and structure of SQL code. Consequently, refactored SQL queries are easier to understand and maintain. Finally, well-refactored queries are generally more scalable, performing well even when the database volumes increase [Garcia-Molina et al. 2000].

Standard refactoring techniques include: decomposing complex queries into simpler, modular parts; eliminating unnecessary columns in `SELECT`, `GROUP BY`, or `ORDER BY` clauses; replacing nested queries with joins; and enabling index usage. SQL query code refactoring is an essential part of database optimization and is often used alongside indexing strategies.

It is important to emphasize that the application of standard refactoring techniques produces SQL code that is semantically equivalent to the original query, thereby ensuring that both versions yield identical result sets.

3. Related Work

SQL rewriting is a key technique for optimizing query performance and is an essential practice in enhancing query performance in relational database systems. Rather than relying solely on the query planner or manual intervention, rewriting techniques aim to transform queries into more efficient, yet semantically equivalent, forms. This can significantly reduce execution time, especially in complex analytical workloads. In this context, [de Araujo et al. 2013] proposed ARe-SQL, an automatic, autonomous, and non-intrusive approach to rewriting SQL queries to improve execution performance in relational database systems. The approach uses a set of 11 heuristics, including the replacement of ALL, ANY, and SOME with aggregation functions (MIN/MAX), the removal of redundant GROUP BY and HAVING clauses, and rewriting subqueries as joins or views, to transform SQL queries into more efficient but semantically equivalent forms. These heuristics were validated over the TPC-H benchmark using PostgreSQL and demonstrated significant improvements in query response time.

Furthermore, the growing interest in employing LLMs to automate natural language translation into SQL has sparked an important discussion in data-centric AI. As LLMs like GPT-3.5 and GPT-4 demonstrate impressive performance on established benchmarks such as Spider, researchers have begun to explore their applicability in more realistic scenarios involving complex database schemas. In this direction, [Nascimento 2024] investigated the effectiveness of LLMs (specifically GPT-3.5 and GPT-4) in the Text-to-SQL task, focusing on real-world schema. Their findings show that while LLMs achieve strong results on benchmarks like Spider, they struggle with complex, large-schema databases such as Mondial due to schema linking, ambiguous naming, and join reasoning difficulties. To mitigate these issues, they proposed using LLM-friendly views and schema descriptions and tested several strategies based on LangChain (including C3, DIN-SQL, and hybrid approaches) to improve performance.

Another key challenge in scaling LLM-based solutions is their limited performance in non-English languages. This limitation is particularly relevant for Portuguese, a language spoken by over 260 million people, yet still underrepresented in most LLM training datasets. In addressing this gap, [Pedroso et al. 2025] systematically evaluated the performance of seven LLMs, including both general-purpose and code-specialized models, on a translated and validated Portuguese version of the Spider benchmark. Their findings highlight a significant performance disparity between English and Portuguese tasks, particularly among smaller models. However, larger, code-specialized models, such as Qwen 2.5 Coder 32B, show more robust cross-lingual capabilities. The authors also identify practical challenges in deploying LLMs on real-world databases, such as excessive schema complexity and ambiguous naming, and propose mitigation strategies like schema abstraction through views, prompt engineering, and domain-specific table selection. Their study establishes benchmark metrics for Portuguese Text-to-SQL performance and provides concrete guidance for organizations aiming to implement LLM-based querying systems in contexts other than English.

4. Text-to-SQL Translation with DeepSeek and Heuristic Corrections: A Continuous Learning Approach

In this work, we explore the capabilities of DeepSeek, a recently released large language model, in the Text-to-SQL task (a domain whose performance remains largely unexplored). Our goal is to assess whether it delivers improvements in generating more transparent and more efficient SQL queries compared to previous models.

4.1. Agent Architecture and Learning Mechanism

Figure 1 depicts the overall architecture of the proposed Text-to-SQL agent. The system is designed to accommodate two user roles. The first one is the End User, whose role is to submit natural language questions and receive correct answers, also in natural language. The second user type is the Admin, responsible for calibrating the embeddings. This dual-user design ensures both usability for nontechnical users and continuous improvement through expert feedback.

The proposed mechanism (see Figure 1) contains two core modules: the first being the DeepSeek model, which serves as the reasoning engine responsible for translating natural language questions into initial SQL queries, and the second is the Optimizer

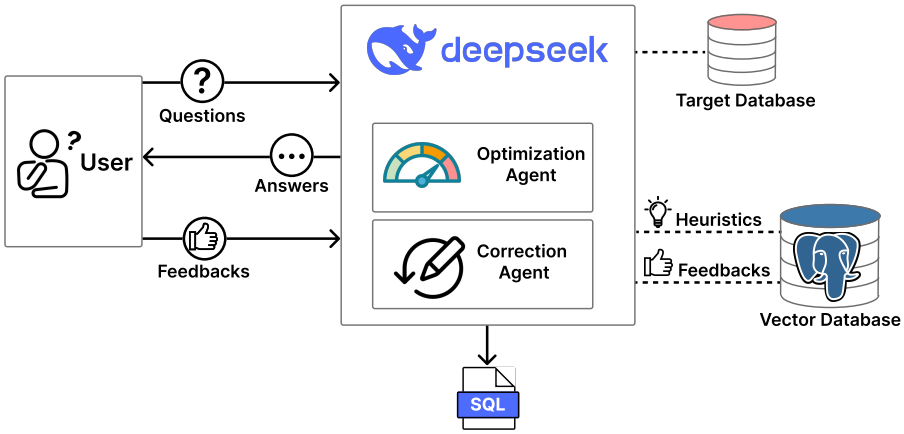

Figure 1. Text-to-SQL agent proposed Architecture.

and the Correction Agent, which enhances query robustness by applying heuristic-based rewrites, and retrieving semantically related past queries from a vector database—implemented using PostgreSQL with pgvector extension[1], whose main function is to look up similar terms and query fragments based on embedding similarity during the correction process.

The user flow begins with the End User submitting a question to the agent. Then, the agent first reads the relational schema to align key terms with database elements such as indexes and primary keys. At the same time, it queries the pgvector database to access previously stored heuristics and corrections. The agent then processes the input and generates an optimized SQL query. The result is executed and returned to the user in natural language, completing the interaction. In contrast, the Admin's workflow is composed mainly of one step, which is to add a list of natural language questions with its query template to the vector database.

Before concluding this section, it is important to note that the proposed approach is not restricted to accessing only PostgreSQL as the target database. In fact, it is designed to be compatible with any relational database management system.

4.2. Heuristics

Table 1 summarizes six heuristics applied in our SQL query optimization process. The first five are derived from ARe-SQL [de Araujo et al. 2013], and the sixth is based on principles outlined in [Farouult and L'Hermite 2008]. We evaluate these heuristics in Section 5 using the TPC-H benchmark, comparing execution times before and after applying them.

H1: Rewriting `ANY` Comparisons. Simplifies subqueries with the `ANY` operator by applying scalar aggregation (e.g., `MAX` or `MIN`), reducing the need for row-by-row comparisons and improving execution efficiency.

H2: Eliminating Redundant `GROUP BY`. Removes `GROUP BY` clauses on primary keys or unique columns that don't affect the result set, leading to simpler execution plans

[1]https://github.com/pgvector/pgvector

Table 1. Heuristics Applied in SQL Query Optimization

Heuristic	Description
H1	Replace `ANY` with `MAX`/`MIN` aggregation.
H2	Remove unnecessary `GROUP BY` on primary keys.
H3	Replace `ALL` with `MAX`/`MIN` aggregation.
H4	Eliminate redundant `DISTINCT` when keys are unique.
H5	Refactor expressions to isolate indexed columns.
H6	Replace correlated subqueries with `JOIN`s or `CTE`s.

and reduced computational overhead.

H3: Simplifying `ALL` Comparisons. Transforms `ALL` comparisons into scalar comparisons using `MAX` or `MIN`, avoiding costly set scans while maintaining semantics.

H4: Removing Redundant `DISTINCT`. Eliminates unnecessary `DISTINCT` when uniqueness is already guaranteed (e.g., via primary keys), reducing sort or deduplication operations.

H5: Enabling Index Usage in Expressions. Rewrites expressions (e.g., `column + 1 = value`) to isolate indexed columns (e.g., `column = value - 1`), enabling index-based access paths.

H6: Rewriting Correlated Subqueries. Substitutes correlated subqueries with equivalent `JOIN`s or `CTE`s to improve performance by enabling set-oriented execution.

5. Experimental Evaluation

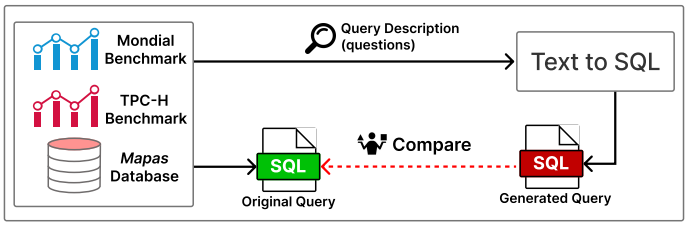

Figure 2. Experiments workflow.

As illustrated in Figure 2, the experimental setup used different testing environments. First, we used two well-known benchmarks: **the Mondial dataset**, to replicate and compare results with those reported in [Nascimento 2024], and **the TPC-H benchmark**, in which we assessed the performance of the LLM-based agent using a set of queries from the TPC-H workload. Additionally, we conducted a set of experiments on **Mapas**, a real-world relational database maintained by Funarte, possessing 61 tables and approximately 205 million tuples, consuming over 24GB of storage space. These tests were performed in *Portuguese* and evaluated both the agent's accuracy and its ability to interact naturally in the target language. The **the Correction Agent** was employed during

this stage to improve query validity when necessary. Validation of the generated queries was supported by a domain specialist at Funarte, who regularly performs such queries on the Mapas system.

The experiments for both Mondial and TPC-h benchmarks were conducted by using PostgreSQL 17.4 hosted in a local machine, with the following setup: 32GB of RAM, 1tb of SSD, AMD Ryzen 5 7600x processor, NVIDIA 4060 8gb graphics card, and using Windows 11. For the real-world database, MAPAS, we used a PostgreSQL 15.12 docker container hosted in a cloud server, with the following setup: 32GB of RAM, an Intel Xeon Gold 6278C processor, and using Ubuntu version 22.04.5 LTS. Each experiment followed a standardized pipeline: a natural language question was submitted to the Text-to-SQL agent, which generated and executed a corresponding SQL query. The result was then reformulated into natural language to complete the interaction cycle. Notably, the Correction Agent was **not employed** in the *Mondial* and *TPC-H*.

In the *Mondial* evaluation, the agent consistently generated valid and well-structured queries without requiring any post-processing. In contrast, the *TPC-H* experiments were designed to test the agent's ability to handle **query optimization challenges**. To this end, we repurposed **five heuristics from ARe-SQL** and **one heuristic from [Faroult and L'Hermite 2008]** in two novel ways:

1. We manually rewrote optimized ARe-SQL queries into natural language questions and created a specific prompt requiring the [Faroult and L'Hermite 2008] heuristic for optimization, enabling direct comparison with the agent's output.
2. We embedded these heuristics into prompt templates, guiding the LLM to apply optimization strategies during query generation.

This methodology introduces a new perspective on integrating classical database heuristics into LLM-based systems. Finally, in the *Mapas* evaluation, we submitted four natural language questions to the agent. The agent answered **three of them correctly**, while **one required intervention from the Correction Agent**, after which the correct SQL result was obtained.

5.1. Evaluation with Mondial Benchmark

The agent was initially evaluated using the *Mondial Benchmark* on a PostgreSQL database. Below, we have four natural language questions and the queries generated by the agent for answering them. It produced semantically correct queries and satisfactory execution times compared to [Nascimento 2024] questions.

The examples in Questions 1, 2 and 3 demonstrate the agent's ability to comprehend semantic context and perform complex operations that transcend basic SQL generation, validating its advanced analytical capabilities. In Question 4 we achieved the correct result that [Nascimento 2024] could not with his model, marking the impressive completeness of our DeepSeek agent.

Question 1. What is the average infant mortality rate for each continent?

```sql
SELECT c.NAME AS continent, AVG(p.infant_mortality) AS average_infant_mortality
FROM continent c
JOIN encompasses e ON c.NAME = e.continent
JOIN population p ON e.country = p.country
GROUP BY c.NAME
ORDER BY c.NAME;
```

Question 2. What is the area of the largest continent?

```sql
SELECT MAX(area) AS largest_continent_area FROM continent;
```

Question 3. What is the total number of provinces in the Netherlands?

```sql
SELECT COUNT(*) AS total_provinces FROM province WHERE country = 'NL';
```

Question 4. What are the languages spoken in Poland?

```sql
SELECT s.language, s.percentage
FROM spoken s
JOIN country c ON s.country = c.code
WHERE c.name = 'Poland'
ORDER BY s.percentage DESC;
```

5.2. Application of Heuristics — Evaluation with TPC-H Benchmark

The agent was prompt-optimized by using pre-defined heuristics, which are tested with the execution time being compared as the metric to prove that a satisfactory query optimization was made after the heuristics were applied at the base case scenario. The Figures 3, 4, 5, 6, 7, 8 presents two queries each, both correct. The first, being at far left, is the one before the heuristic being applied, and the second query, being at far right, is after the heuristic application, optimized. As mentioned in the Section 4.2, the execution time is discussed below.

```sql
SELECT o_orderkey, o_custkey,
    ↪ o_orderstatus, o_totalprice
FROM orders
WHERE o_totalprice > ANY (
    SELECT o_totalprice
    FROM orders
    WHERE o_orderpriority = '2-HIGH'
);
--Execution Time: 8.9442s
```

Query N1

```sql
SELECT o_orderkey, o_custkey,
    ↪ o_orderstatus, o_totalprice
FROM orders
WHERE o_totalprice > (
    SELECT MIN(o_totalprice)
    FROM orders
    WHERE o_orderpriority = '2-HIGH'
);
--Execution Time: 6.5231s
```

Query N2 rewritten from N1 using H1.

Figure 3. Question 5: Get the order key, customer key, order status, and total price for every order whose total price exceeds at least one order with a priority of '2-HIGH'.

The examples in Figure 3 show the experimental comparison of N1 and N2, the application of H1 resulted in a reduction of the execution time from 8.9442 seconds to 6.5231 seconds (1.37x faster), attributed to fewer comparisons of tuples and a more optimized execution plan generated by the database system.

```sql
SELECT c.c_custkey,c.c_name,c.c_address,c.
    ↪ c_phone, n.n_name, n.n_regionkey
FROM customer
JOIN nation n ON c.c_nationkey = n.
    ↪ n_nationkey
GROUP BY c.c_custkey, c.c_name,c.c_address
    ↪ , c.c_phone, n.n_name,n.n_regionkey
ORDER BY c.c_custkey;
--Execution Time: 0.7213s
```

Query N3

```sql
SELECT c.c_custkey,c.c_name,c.c_address,c.
    ↪ c_phone, n.n_name,n.n_regionkey
FROM customer c
JOIN nation n ON c.c_nationkey = n.
    ↪ n_nationkey
ORDER BY c.c_custkey;
--Execution Time: 0.6956s
```

Query N4, rewritten from N3 using H2

Figure 4. Question 6: Show me customer details (ID, name, address, and phone) along with their associated nation and region information grouped.

Figure 4 demonstrates the application with the comparison between queries N3 and N4, before and after the H2 heuristic application, the execution time decreased from 0.7213 seconds to 0.6956 seconds, representing a performance improvement of approximately 3.6%. This improvement results from eliminating unnecessary operations in the execution plan and reducing query processing overhead.

```sql
SELECT o_orderkey, o_custkey,
    ↪ o_orderstatus, o_totalprice
FROM orders
WHERE o_totalprice > ALL (
    SELECT o_totalprice FROM orders
    WHERE o_orderpriority = '2-HIGH'
);
--Execution Time: 0.94s
```

Query N5

```sql
SELECT o_orderkey, o_custkey,
    ↪ o_orderstatus, o_totalprice
FROM orders
WHERE o_totalprice > (
    SELECT MAX(o_totalprice) FROM orders
    WHERE o_orderpriority = '2-HIGH'
);
--Execution Time: 0.187s
```

Query N6, rewritten from N5 using H3

Figure 5. Question 7: Get the order key, customer key, order status, and total price for every order whose total price is greater than all orders that have a priority of '2-HIGH'

Figure 5 illustrates the optimization when using heuristic H3, by comparing queries N5 and N6. The execution time decreased significantly from 0.94 seconds to 0.187 seconds, resulting in a performance improvement of approximately 80.11%. This gain reflects the benefits of reducing subquery complexity and allowing the query engine to execute a more streamlined plan.

```sql
SELECT DISTINCT o.o_orderkey AS order_id,
    ↪ o.o_totalprice AS total_price, c.
    ↪ c_name AS customer_name
FROM orders o
JOIN customer ON o.o_custkey = c.c_custkey
ORDER BY o.o_orderkey;
-- Execution Time: 6.6789s
```

Query N7

```sql
SELECT o.o_orderkey AS order_id, o.
    ↪ o_totalprice AS total_price, c.
    ↪ c_name AS customer_name
FROM orders o
JOIN customer ON o.o_custkey = c.c_custkey
ORDER BY o.o_orderkey;
-- Execution Time: 6.5814s
```

Query N8 rewritten from N7 by using H4

Figure 6. Question 8: Show me a clean list of all orders with no duplicates, including each order's ID, total price, and the customer's name who placed it - sorted by order number.

Figure 6 shows the application for H4, which reduced the execution time from 6.6789 seconds to 6.5814 seconds, reflecting an improvement of approximately 1.5% in response time.

```sql
SELECT * FROM lineitem
WHERE l_quantity / 2 = 40;
--Execution Time: 0.2859s
```

Query N9

```sql
SELECT * FROM lineitem
WHERE l_quantity = 80;
--Execution Time: 0.0019s
```

Query N10 rewritten from N9 by using H5

Figure 7. Question 9: Show me all line items where half of the quantity equals exactly 40.

Figure 7 shows that the application of H5 reduced the execution time from 0.2859 seconds to 0.0019 seconds, representing a dramatic improvement of over 99.3%. This illustrates the impact even simple rewrites can have when enabled index access.

```sql
SELECT c.c_custkey,c.c_name, MAX(o.
    ↪ o_orderdate) AS
    ↪ most_recent_order_date,o.
    ↪ o_totalprice
FROM customer
JOIN orders o ON c.c_custkey = o.o_custkey
GROUP BY c.c_custkey, c.c_name, o.
    ↪ o_totalprice
ORDER BY c.c_custkey;
--Execution Time: 7.7952s
```

Query N11

```sql
WITH customer_recent_orders AS
(SELECT o_custkey,MAX(o_orderdate) AS
    ↪ most_recent_order_date
  FROM orders GROUP BY o_custkey)

SELECT c.c_custkey, c.c_name,cro.
    ↪ most_recent_order_date,o.
    ↪ o_totalprice
FROM customer c JOIN
    ↪ customer_recent_orders cro ON c.
    ↪ c_custkey = cro.o_custkey
JOIN orders o ON cro.o_custkey = o.
    ↪ o_custkey AND cro.
    ↪ most_recent_order_date = o.
    ↪ o_orderdate
ORDER BY c.c_custkey;
--Execution Time: 0.9525s
```

Query N12 rewritten from N11 by using H6

Figure 8. Question 10: For each customer, what is the date of their most recent order and the total price of that order?

In the case study comparing queries N11 and N12, applying H6 led to a drastic reduction in execution time — from 7.7952 seconds to 0.9525 seconds, as Figure 8 shows, yielding an 87.78% performance improvement. This improvement is consistent with established literature: subqueries in WHERE EXISTS or SELECT clauses, when correlated, often force nested-loop evaluations, while their refactored versions benefit from set-oriented execution plans. [Faroult and L'Hermite 2008]

5.3. Evaluation with Mapas Database

Mapas is a large and complex relational database maintained by Funarte, hosted in an open-source PostgreSQL environment. In this subsection, we present a series of experiments conducted in *Portuguese* to evaluate the performance of the LLM-based agent when interacting with real-world data through natural language prompts. In this link, the reader is invited to see more examples: Click here to visit the full list of examples.

```sql
SELECT o.name, COUNT(r.id) AS
    ↪ registration_count
FROM opportunity o
JOIN registration r ON o.id = r.
    ↪ opportunity_id
GROUP BY o.id, o.name
ORDER BY registration_count DESC LIMIT 1;
-- Execution Time: 0.2410s
```

Query N13 not optimized by any heuristics

```sql
WITH opp_reg_counts AS (
    SELECT o.id, o.name, COUNT(r.id) AS
        ↪ registration_count
    FROM opportunity o
    LEFT JOIN registration r ON o.id = r.
        ↪ opportunity_id
    GROUP BY o.id, o.name
)
SELECT name FROM opp_reg_counts
ORDER BY registration_count DESC LIMIT 1;
--Execution Time: 0.2382s
```

Query N14 rewritten from N13 by using the heuristics.

Figure 9. Question 11: Qual o nome da oportunidade com maior número de inscrições?

In Figures 9 and 10, we present two cases in which the LLM generated correct SQL queries without any schema-related mistakes. While the results were valid, we conducted additional experiments applying the proposed heuristics to explore potential performance improvements. In the first case (Query N13 to N14), we observed a slight reduction in execution time from 0.2410 seconds to 0.2382 seconds. Similarly, in the sec-

```sql
SELECT a.id AS agent_id, a.name AS
    ↪ agent_name, s.name AS space_name, p
    ↪ .name AS project_name
FROM agent a
JOIN space s ON a.id = s.agent_id
JOIN project p ON a.id = p.agent_id
WHERE s.name = p.name
ORDER BY a.name;
-- Execution Time: 0.2574s.
```

Query N15 not optimized by any heuristics

```sql
WITH opp_reg_counts AS (
    SELECT o.id, o.name, COUNT(r.id) AS
        ↪ registration_count
    FROM opportunity o
    LEFT JOIN registration r ON o.id = r.
        ↪ opportunity_id
    GROUP BY o.id, o.name
)
SELECT name FROM opp_reg_counts
ORDER BY registration_count DESC LIMIT 1;
--Execution Time: 0.2483s
```

Query N16 rewritten from N15 by using the heuristics.

Figure 10. Question 12: Quais os agentes possuem espaços e projetos culturais com o mesmo nome?

ond case (Query N15 to N16), execution time improved from 0.2574 seconds to 0.2483 seconds.

Figure 11 illustrates a case where the LLM initially selected incorrect tables during query generation (Query N17). Specifically, it relied on “addr”, “place”, and geographic filters using “ST_Contains”, which were not aligned with the intended query logic. Upon presenting the correction interface, we provided via prompt what was wrong, especially the wrong table names, and a right example to follow. After this, the LLM correctly applied the feedback and generated an accurate query, without ambiguity (Query N18), both semantically and structurally aligned with the desired result. With the example, the agent generated the correct query without confusion.

```sql
SELECT s.id, s.name, s.short_description,
    ↪ s.long_description
FROM space s
JOIN space_meta sm ON s.id = sm.object_id
WHERE sm.key = 'geoestado_form3' AND sm.
    ↪ value = 'CE'
AND s.id IN (
    SELECT object_id FROM space_meta
    WHERE key = 'geomunicipio_form3'
    AND value = 'Fortaleza'
)
AND s.id IN (
    SELECT object_id FROM space_meta
    WHERE
    key = 'metodologia1_capacidade_form8'
    AND CAST(value AS INTEGER) >= 100
);
--Execution Time: 0.1916s
```

Query N17 not optimized by any heuristics and not using the correct tables for querying.

```sql
SELECT s.name
FROM space s
WHERE EXISTS (
    SELECT 1 FROM space_meta sm
    WHERE sm.object_id = s.id
    AND sm.key = 'En_Municipio'
    AND sm.value = 'Fortaleza'
)
AND EXISTS (
    SELECT 1
    FROM space_meta sm
    WHERE sm.object_id = s.id
    AND sm.key = 'Capacidade'
    AND CAST(sm.value AS INTEGER) >= 100
);
--Execution Time: 0.1886s
```

Query N18 rewritten from N17 by using the heuristics and the correction function of the agent.

Figure 11. Question 13: Busque os espaços culturais na cidade de Fortaleza com capacidade para pelo menos 100 pessoas.

5.4. Discussion

It is important to clarify that the proposed agent does *not* intend to replace or interfere with the internal query optimizer of the database management system (DBMS). Instead, it operates as a preprocessing layer, rewriting SQL queries before they are processed by the optimizer. This aligns with standard practices in query refactoring for database performance tuning [Shasha and Bonnet 2003].

Our focus is solely on SQL code transformation—producing semantically equivalent queries ($Q \rightarrow Q'$) that enable the DBMS to choose better execution plans. Compared to query hints, which are engine-specific and often non-portable, our approach is auditable and engine-agnostic. Domain experts can inspect and validate rewritten queries without requiring internal DBMS knowledge.

While we do not offer formal algebraic guarantees, we ensure correctness through empirical validation: queries are only rewritten when their result sets remain unchanged. This strategy follows the precedent of prior works [de Araujo et al. 2013] and is sufficient for practical deployments. The heuristics used are well-established [Faroult and L'Hermite 2008, Shasha and Bonnet 2003], with known semantics-preserving properties.

Importantly, our evaluation compares the agent's initial and rewritten outputs—measuring improvements in execution time and correctness—rather than competing with the DBMS's internal optimizer. The gains reported stem from improving the LLM-generated SQL before the DBMS receives it.

6. Conclusion and Future Work

This paper presents a text-to-SQL agent based on the *DeepSeek* model, designed to translate natural language questions into structured SQL queries. The agent integrates heuristic-driven optimization strategies during query generation and incorporates a user correction mechanism to refine its output iteratively. Both components leverage the `pgvector` extension to function effectively.

Our experimental evaluation, conducted using the Mondial and TPC-H benchmarks, assessed both the base capabilities of the agent and its performance with applied heuristics. The results demonstrated that including heuristics significantly improved execution times while preserving semantic accuracy. Additionally, we evaluated the agent on a real-world, Portuguese-language dataset (Funarte/MapasCulturais) featuring a complex schema. Despite the added challenges, the agent answered two out of three questions, with the third accurately resolved through the correction mechanism.

For future work, we intend to broaden our heuristics and perform additional experiments using an expanded range of queries from the Mondial and TPC-H benchmarks. We also plan to deepen our evaluation of the Funarte/Mapas Culturais dataset by classifying questions according to difficulty levels (easy, medium, hard) and investigating how the model manages linguistic complexity and schema intricacy in Portuguese. Furthermore, we aim to refine our use of embedding functions to improve word similarity analysis.

References

[de Araujo et al. 2013] de Araujo, A. H. M., Monteiro, J. M., de Macedo, J. A. F., and Brayner, A. (2013). On using an automatic, autonomous and non-intrusive approach for rewriting sql queries. *Journal of Information and Data Management*, 3(3):1–15.

[DeepSeek 2025] DeepSeek (2025). Deepseek. `https://www.deepseek.com/`. Acessado em: 2 de maio de 2025.

[Fan et al. 2020] Fan, A., Urbanek, J., Ringshia, P., Dinan, E., Qian, E., Karamcheti, S., Prabhumoye, S., Kiela, D., Rocktaschel, T., Szlam, A., et al. (2020). Generating in-

teractive worlds with text. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1693–1700.

[Faroult and L'Hermite 2008] Faroult, S. and L'Hermite, P. (2008). *Refactoring SQL Applications*. O'Reilly Media, Sebastopol, CA.

[Garcia-Molina et al. 2000] Garcia-Molina, H., Ullman, J. D., and Widom, J. (2000). *Database System Implementation*. Prentice Hall, New Jersey, USA.

[Hadi et al. 2023] Hadi, M. U., Qureshi, R., Shah, A., Irfan, M., Zafar, A., Shaikh, M. B., Akhtar, N., Wu, J., Mirjalili, S., et al. (2023). Large language models: a comprehensive survey of its applications, challenges, limitations, and future prospects. *Authorea Preprints*, 1:1–26.

[Hong et al. 2024] Hong, Z., Yuan, Z., Zhang, Q., Chen, H., Dong, J., Huang, F., and Huang, X. (2024). Next-generation database interfaces: A survey of llm-based text-to-sql. *arXiv preprint arXiv:2406.08426*.

[Minaee et al. 2024] Minaee, S., Mikolov, T., Nikzad, N., Chenaghlu, M., Socher, R., Amatriain, X., and Gao, J. (2024). Large language models: A survey. arxiv 2024. *arXiv preprint arXiv:2402.06196*.

[Ministério da Cultura 2025] Ministério da Cultura (2025). Mapas culturais - funarte. `https://mapas.cultura.gov.br/`. Acessado em: 2025-05-02.

[Nascimento 2024] Nascimento, E. R. S. (2024). Querying databases with natural language: The use of large language models for text-to-sql tasks. Dissertação de mestrado, Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil. Advisor: Marco Antonio Casanova.

[OpenAI 2025] OpenAI (2025). Chatgpt.

[Ozdemir 2023] Ozdemir, S. (2023). *Quick start guide to large language models: strategies and best practices for using ChatGPT and other LLMs*. Addison-Wesley Professional.

[Pedroso et al. 2025] Pedroso, B. C., Pereira, M. R., and Pereira, D. A. (2025). Performance evaluation of llms in the text-to-sql task in portuguese. In *Proceedings of the SBSI25, Recife, PE*.

[Ramakrishnan and Gehrke 2002] Ramakrishnan, R. and Gehrke, J. (2002). *Database Management Systems*. McGraw-Hill, 3rd edition.

[Sala et al. 2024] Sala, L., Sullutrone, G., and Bergamaschi, S. (2024). Text-to-sql with large language models: Exploring the promise and pitfalls. In *Proceedings of the 32nd Symposium on Advanced Database Systems (SEBD 2024)*. CEUR Workshop Proceedings.

[Shasha and Bonnet 2003] Shasha, D. and Bonnet, P. (2003). *Database Tuning: Principles, Experiments, and Troubleshooting Techniques*. Morgan Kaufmann.

[Yang et al. 2024] Yang, J., Jin, H., Tang, R., Han, X., Feng, Q., Jiang, H., Zhong, S., Yin, B., and Hu, X. (2024). Harnessing the power of llms in practice: A survey on chatgpt and beyond. *ACM Transactions on Knowledge Discovery from Data*, 18(6):1–32.

[Zhang et al. 2024] Zhang, Y., Jin, H., Meng, D., Wang, J., and Tan, J. (2024). A comprehensive survey on process-oriented automatic text summarization with exploration of llm-based methods. *arXiv preprint arXiv:2403.02901*. Preprint, not peer-reviewed.

[Zhu et al. 2024] Zhu, X., Li, Q., Cui, L., and Liu, Y. (2024). Large language model enhanced text-to-sql generation: A survey. *arXiv preprint arXiv:2410.06011*. Preprint, not peer-reviewed.