Journal of Information and Data Management, 2026, 17:1, doi: 10.5753/jidm.2026.7393
This work is licensed under a Creative Commons Attribution 4.0 International License.

SQL Refactoring in Text-to-SQL Platforms: Optimizing Code for Performance

Laura Petrola [**Universidade Federal do Ceará** | ***laupetrola@gmail.com***]
Angelo Brayner [**Universidade Federal do Ceará** | ***brayner@dc.ufc.br***]
Wellington Franco [**Universidade Federal do Ceará** | ***wellington@crateus.ufc.br***]

Mestrado e Doutorado em Ciências da Computação, Universidade Federal do Ceará (UFC), Brazil.

Received: 14 January 2026 • **Published:** 27 November 2026

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 specialized open-source LLM **DeepSeek-Coder** 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 (Mapas), we demonstrate high execution efficiency with performance gains ranging from 1.5% to 99.3%, highlighting the impact of feedback loops and heuristic-based query rewriting on complex, real-world schemas.

Keywords: Text-to-SQL, LLMs, Query Optimization, DeepSeek, Relational Databases

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 for understanding and generating natural language, which naturally extends to applications that translate 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 of LLM-based Text-to-SQL systems, classifying methodologies into 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 factors, such as 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 real-world database schemas are often large and complex, contain multiple objects (tables, indexes, and views), and are not necessarily well-designed. As highlighted by recent studies, including [Sala *et al*., 2024], LLMs often fail 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, handling a large volume of data and database objects. Among various strategies to improve query processing performance, SQL query refactoring has emerged as a key approach to minimize 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 profession-

als 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].

Also, they 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, and 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].

Fundamentally, modern LLMs are built upon the **Transformer architecture** Vaswani *et al*. [2017]. This architecture relies on self-attention mechanisms to weigh the significance of different words in a sentence, allowing the model to capture long-range dependencies and complex syntactic structures more effectively than previous recurrent neural networks.

A defining capability of these models is **Few-Shot Learning**, as demonstrated by Brown *et al*. [2020]. Unlike traditional supervised learning, which requires extensive retraining for new tasks, few-shot learners can adapt to a specific task, such as generating SQL queries, simply by being provided with a few input-output examples (“shots”) in the prompt context. This allows the model to infer the desired output format and logic without weight updates.

To further enhance performance on complex reasoning tasks, recent approaches employ **Chain-of-Thought (CoT) prompting** Wei *et al*. [2022]. Instead of generating a direct answer, CoT encourages the model to produce a sequence of intermediate reasoning steps. In Text-to-SQL, this involves explicitly identifying primary keys, determining join paths, and filtering criteria before constructing the final SQL code, thereby reducing logical errors.

2.2 SQL Refactoring

SQL query refactoring is the systematic rewriting of SQL queries to improve their efficiency and maintainability without altering their intended functionality or results [Faroult 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, for instance, by 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 applying standard refactoring techniques produces SQL code that is semantically equivalent to the original query, 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 replacing ALL, ANY, and SOME with aggregation functions (MIN/MAX), removing redundant GROUP BY and HAVING clauses, and rewriting subqueries as joins or views, to transform SQL queries into more efficient, yet 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 that remains 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, including schema abstraction via 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 seeking to implement LLM-based querying systems in contexts beyond English.

This manuscript significantly extends our preliminary study presented in [Petrola *et al*., 2025], where we introduced the foundational architecture for heuristic-guided Text-to-SQL translation using the DeepSeek model. While the previous work established the baseline effectiveness of heuristic rewriting on standard benchmarks, this extended version introduces the comprehensive “Agent Flow” architecture (detailed in Section 4.1), expands the optimization rule set with new heuristics for filter mirroring and clause rewriting (H7 and H8), and provides a deeper experimental analysis on the Mapas database, specifically leveraging semantic embeddings for error correction.

Recent advancements in benchmarking have shifted focus towards more realistic database environments. While Spider focuses on structural query complexity, the **BIRD benchmark [Li *et al*., 2024]** highlights the challenges of “dirty” data and the need for external knowledge reasoning in large-scale databases. Our work aligns with this trend by evaluating performance on Mapas, a real-world dataset that requires navigating complex schemas and noisy data, rather than relying solely on sanitized academic benchmarks.

4 Text-to-SQL Translation and Heuristic Corrections: A Model-Agnostic Approach

A fundamental design decision for the Athena framework was to prioritize architectural modularity and model agnosticism. To validate the system’s effectiveness, **DeepSeek-Coder** (v1) was selected as a representative cognitive engine for the evaluation phase. This choice was driven by the practical need for a high-performance, **open-source** (open-weights) model specialized in programming tasks, providing a transparent and reproducible baseline for testing. As an open-source tool, DeepSeek-Coder allows for full auditability of the translation process, avoiding the ”black-box” limitations associated with proprietary models. However, it is important to emphasize that Athena is not tethered to any specific LLM provider. Technically, the integration is performed via a standardized OpenAI-compatible API, ensuring the framework can seamlessly swap the underlying cognitive engine for other models without requiring structural changes to the Grounding Module or the Heuristic Layer. This modularity demonstrates that the core innovation of Athena lies in its autonomous optimization logic rather than in the specific capabilities of a single language model.

4.1 Agent Architecture

Figure 1 illustrates the agent flow, the proposed modular architecture designed to bridge the gap between natural language ambiguities and precise database execution. The system is organized into three distinct processing layers: the *Grounding Module*, the *Cognitive Pipeline*, and the *Physical Execution Layer*.

The system supports two user roles: the End User, who submits questions, and the Admin, who calibrates embeddings. This design ensures usability while allowing for continuous improvement.

1. Grounding Module (Contextualization)

The process begins with the **Grounding Module**, which anchors the user’s natural language input to relevant domain knowledge. Upon receiving a query, the system performs **Vectorization** to convert the text into high-dimensional embeddings. These embeddings trigger a **Semantic Search** against a specialized Vector Database (Vector DB) containing curated pairs of questions and validated SQL queries. The result is a set of ”Contextual Pairs” that are injected into the model’s prompt via Retrieval-Augmented Generation (RAG).

2. Cognitive Pipeline (Dual-Stage Generation)

The core reasoning engine, labeled as the **Cognitive Pipeline**, employs a novel **Dual-Stage Generation** strategy to ensure both semantic accuracy and execution efficiency:

- **Stage A: Translation Agent (DeepSeek):** The first LLM instance receives the Database Schema (DDL)

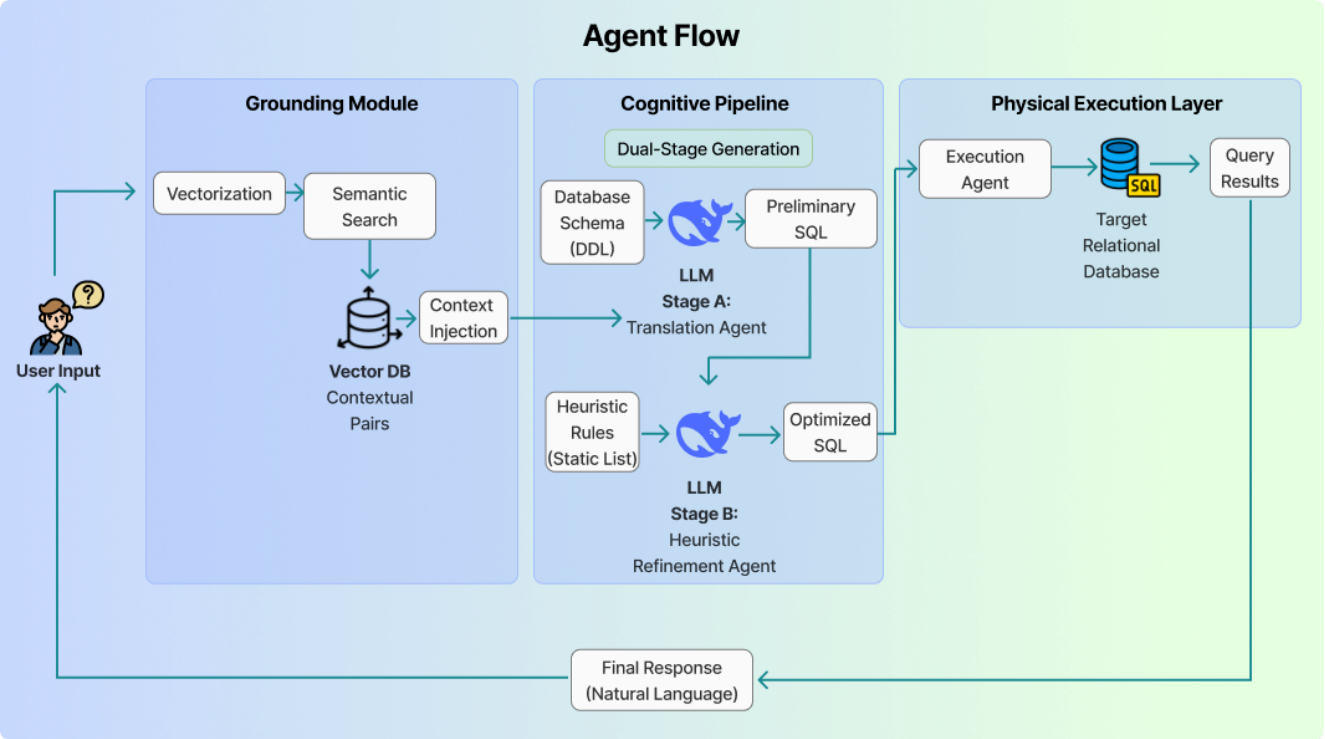

Figure 1. Agent Flow: The proposed Text-to-SQL agent architecture.

and the injected context. It operates primarily as a translator, converting the user's intent into a *Preliminary SQL* query. To enhance robustness, we employ **Chain-of-Thought (CoT) prompting [Wei *et al.*, 2022]**, instructing the model to generate intermediate reasoning steps, such as identifying primary keys and join paths, before producing the code. This decomposes the problem into logical constraints, reducing hallucinations.

- **Stage B: Heuristic Refinement Agent:** The output from Stage A is passed to a second processing stage. Here, the model integrates a static list of **Heuristic Rules** (detailed in Section 4.2) to refactor the code structure—for instance, replacing sub-optimal `IN` clauses with `JOIN`s—without altering business logic, producing the final *Optimized SQL*.

3. Physical Execution Layer

Finally, the *Optimized SQL* reaches the **Physical Execution Layer**. The **Execution Agent** submits the query to the Target Relational Database. Uniquely, this agent leverages a feedback loop inspired by **self-debugging techniques [Chen *et al.*, 2023]**: if an execution error occurs, the agent captures the error message and the failed query, referencing the retrieved embeddings to "debug" and correct the output before returning the *Final Response* to the user.

4.2 Heuristics

Table 1 summarizes eight heuristics applied in our SQL query optimization process. The first five are derived from ARe-SQL [de Araujo *et al.*, 2013], the sixth is based on principles outlined in [Faroult and L'Hermite, 2008], and the last two are also adapted from the work of Professor Dr.Brayner. 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 [de Araujo *et al.*, 2013].

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 and reduced computational overhead [de Araujo *et al.*, 2013].

H3: Simplifying `ALL` Comparisons. Transforms `ALL` comparisons into scalar comparisons using `MAX` or `MIN`, avoiding costly set scans while maintaining semantics [de Araujo *et al.*, 2013].

H4: Removing Redundant `DISTINCT`. Eliminates unnecessary `DISTINCT` when uniqueness is already guaranteed (e.g., via primary keys), reducing sort or deduplication operations [de Araujo *et al.*, 2013].

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 [de Araujo *et al.*, 2013].

H6: Rewriting Correlated Subqueries. Substitutes correlated subqueries with equivalent `JOIN`s or `CTE`s to improve

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.
H7	Mirror filters across join keys for early pruning.
H8	Rewrite `IN (...)` as multiple `OR` conditions.

performance by enabling set-oriented execution [Faroult and L'Hermite, 2008].

H7: Mirroring Filters Across Join Keys. Propagates filter conditions from one table to the joined table (e.g., if filtering on `t1.id`, also filter on `t2.foreign_id`). This allows the query optimizer to prune rows earlier in the execution plan, potentially reducing the size of intermediate result sets before the join operation.

H8: Rewriting `IN` as Multiple `OR` Conditions. Converts `IN` clauses with a list of values into a series of `OR` conditions. This transformation facilitates the use of index scans or more efficient evaluation strategies compared to processing the `IN` list as a set.

5 Experimental Evaluation

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 **Correction Agent** was employed during this stage to improve query validity when necessary. The validation of the generated queries was supported by a domain specialist at Funarte, who regularly runs 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 are four natural language questions, along with the queries generated by the agent to answer them. It produced semantically correct queries and satisfactory execution times compared to the questions in [Nascimento, 2024].

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?

```
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;
```

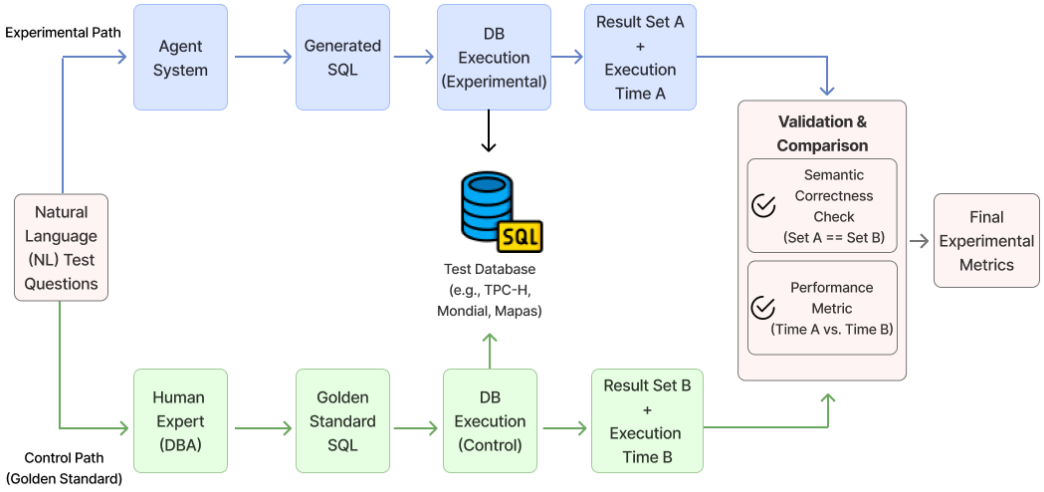

Figure 2. Experiments workflow.

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 using predefined heuristics, which were tested by comparing execution time as the metric to demonstrate that satisfactory query optimization was achieved after the heuristics were applied in the base case scenario. The Figures 3, 4, 5, 6, 7, 8 presents two queries each, both correct. The first, at the far left, is before the heuristic is applied, and the second, at the far right, is after the heuristic is applied and 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 achieved with 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, complex relational database maintained by Funarte and hosted in an open-source PostgreSQL environment. It contains 61 tables and approximately 205 million tuples, consuming over 24GB of storage. 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.

The evaluation covers eight natural language questions. Each question was specifically designed to trigger one of the optimization heuristics defined in Section 4.2. This comprehensive test suite allows us to isolate the impact of logical rewriting on query execution time.

Table 2 summarizes the execution times for the Non-Heuristic (NH) queries generated by the baseline LLM versus the With-Heuristic (WH) queries optimized by our agent. The results demonstrate consistent performance gains across all categories.

Below, we present the detailed SQL transformations for all eight experiments.

Question 1: Redundant Grouping (H2)

The user asked for a simple list of Agent IDs and Names, requesting uniqueness. The baseline model unnecessarily grouped by ID, forcing a sort operation. The optimized version removed the group by, relying on the primary key for uniqueness.

```sql
-- NH: Unnecessary GROUP BY
SELECT id, MAX(name) AS name
FROM agent
GROUP BY id
ORDER BY id;
-- Time: 0.5185s
```

```sql
-- WH: Simplified Selection
SELECT id, name
FROM agent;
-- Time: 0.4040s
```

Figure 9. Question 1: List IDs and Names of all Agents.

Question 2: Optimizing ALL (H3)

The user sought registrations with scores higher than *all* registrations with status 0. The heuristic replaced the `ALL` operator with a single scalar `MAX` subquery.

```sql
-- NH: Using ALL operator
SELECT r1.* FROM registration r1
WHERE r1.score > ALL (
    SELECT r2.score
    FROM registration r2
    WHERE r2.status = 0
);
-- Time: 0.0618s
```

```sql
-- WH: Using MAX aggregation
SELECT r1.* FROM registration r1
WHERE r1.score > (
    SELECT MAX(r2.score)
    FROM registration r2
    WHERE r2.status = 0
);
-- Time: 0.0408s
```

Figure 10. Question 2: Find registrations with scores higher than all status 0.

Question 3: Optimizing ANY (H1)

This case showed the highest performance gain (97.5%). The baseline used `ANY` to compare scores against opportunity 5, causing a nested loop. The agent rewrote this using `MIN`, allowing the database to compute the comparison value once.

```sql
-- NH: Using ANY operator
SELECT DISTINCT r.id, r.number
FROM registration r
... -- (Joins)
WHERE re.score > ANY (
    SELECT re2.score ...
    WHERE r2.opportunity_id = 5
);
-- Time: 0.0873s
```

```sql
-- WH: Using MIN aggregation
SELECT DISTINCT r1.id, r1.number
FROM registration r1
WHERE r1.score > (
    SELECT MIN(r2.score)
    FROM registration r2
    WHERE r2.opportunity_id = 5
      AND r2.score IS NOT NULL
);
-- Time: 0.0022s
```

Figure 11. Question 3: List registrations with scores > any score in Opp 5.

Question 4: Sargability (H5)

The baseline used `EXTRACT(YEAR)` on a timestamp column, preventing index usage. The agent rewrote this as a date range query.

```sql
-- NH: Non-sargable function
SELECT * FROM project
WHERE EXTRACT(YEAR FROM
  create_timestamp) = 2023;
-- Time: 0.0149s
```

```sql
-- WH: Index-friendly range
SELECT id, name, create_timestamp
FROM project
WHERE create_timestamp >=
      '2023-01-01 00:00:00'
  AND create_timestamp <
      '2024-01-01 00:00:00';
-- Time: 0.0056s
```

Figure 12. Question 4: List projects created in 2023.

Question 5: Unnecessary Distinct (H4)

The baseline applied `DISTINCT` on the primary key, incurring sort overhead. The agent identified the unique key and removed the keyword.

```sql
-- NH: Redundant DISTINCT
SELECT DISTINCT e.id, e.name
FROM event e
JOIN project p ON e.project_id = p.id
WHERE p.is_verified = true;
-- Time: 0.0034s
```

```sql
-- WH: Simplified
SELECT e.id, e.name
FROM event e
JOIN project p ON e.project_id = p.id
WHERE p.is_verified = true;
-- Time: 0.0013s
```

Figure 13. Question 5: List events from verified projects.

Question 6: Correlated Subquery (H6)

The baseline query failed initially due to schema errors, but was corrected via embeddings. The optimization involved replacing a correlated structure with a Window Function (CTE), improving time by 53.5%.

Table 2. Performance comparison on Mapas Database (Time in Seconds)

ID	Heuristic Applied	Time (NH)	Time (WH)	Improvement
Q1	H2: Remove Redundant GROUP BY	0.5185s	0.4040s	22.1%
Q2	H3: ALL → MAX/MIN	0.0618s	0.0408s	34.0%
Q3	H1: ANY → MAX/MIN	0.0873s	0.0022s	97.5%
Q4	H5: Index-Friendly Expressions	0.0149s	0.0056s	62.4%
Q5	H4: Remove Redundant DISTINCT	0.0034s	0.0013s	61.7%
Q6	H6: Correlated Subquery → Join	0.1103s	0.0513s	53.5%
Q7	H7: Mirror Filters	0.0041s	0.0021s	48.8%
Q8	H8: IN → OR	0.0140s	0.0031s	77.8%

```sql
-- NH: Group By / Aggregation
SELECT a.id, a.name,
       MAX(p.create_timestamp)
FROM agent a
JOIN project p ON a.id = p.agent_id
GROUP BY a.id, a.name, p.name;
-- Time: 0.1103s
```

```sql
-- WH: Window Function CTE
WITH agent_projects AS (
 SELECT p.agent_id, ...,
  ROW_NUMBER() OVER (
   PARTITION BY p.agent_id
   ORDER BY p.create_timestamp DESC
  ) AS rn
 FROM project p ...
)
SELECT * FROM agent_projects
WHERE rn = 1;
-- Time: 0.0513s
```

Figure 14. Question 6: Most recent project per agent.

Question 7: Mirror Filters (H7)

The agent added a redundant filter to the joined table (`registration`), allowing the optimizer to filter rows before the join.

```sql
-- NH: Single Filter
SELECT r.number, o.name
FROM registration r
JOIN opportunity o
  ON r.opportunity_id = o.id
WHERE o.id = 10;
-- Time: 0.0041s
```

```sql
-- WH: Mirrored Filter
SELECT r.number, o.name
FROM registration r
JOIN opportunity o
  ON r.opportunity_id = o.id
WHERE r.opportunity_id = 10
  AND o.id = 10;
-- Time: 0.0021s
```

Figure 15. Question 7: Registration number for Opportunity 10.

Question 8: IN to OR (H8)

The agent rewrote an `IN` list as multiple `OR` conditions, which in this specific PostgreSQL configuration resulted in a faster execution plan.

```sql
-- NH: IN Clause
SELECT id
FROM event_occurrence
WHERE frequency IN
 ('daily', 'weekly', 'monthly');
-- Time: 0.0140s
```

```sql
-- WH: OR Conditions
SELECT id
FROM event_occurrence
WHERE frequency = 'daily'
   OR frequency = 'weekly'
   OR frequency = 'monthly';
-- Time: 0.0031s
```

Figure 16. Question 8: Filter event occurrences by frequency.

Summary of Mapas Findings

The experiments on the Mapas database confirm the system's ability to handle real-world challenges, including complex schema navigation and Portuguese natural language processing. The integration of semantic embeddings proved essential for error correction, enabling the agent to resolve ambiguities that prevented baseline execution. Furthermore, the heuristic optimization layer delivered consistent performance gains across all test cases, with execution time reductions ranging from 22% to 97.5%. These results validate the proposed architecture as a robust solution for deploying efficient Text-to-SQL interfaces in production environments.

5.4 Accuracy Evaluation and Impact of Embeddings

To evaluate the reliability of the Athena agent, we compared its performance against a baseline (DeepSeek-Coder zero-shot) using a **Golden Standard** defined as the expert-curated version of each query. We employ two primary metrics to quantify success:

1. **Execution Accuracy (EX):** Measures the functional correctness of the generated query by comparing its result set (R_{gen}) with the reference result set (R_{ref}). It is defined as:

$$EX = \frac{\text{Number of queries where } R_{gen} = R_{ref}}{\text{Total number of queries}} \times 100\%$$

2. **Exact Match (EM):** Measures the structural identity between the generated SQL string (S_{gen}) and the reference SQL string (S_{ref}), after standardizing whitespace and casing. It is defined as:

$$EM = \frac{\text{Number of queries where } S_{gen} \equiv S_{ref}}{\text{Total number of queries}} \times 100\%$$

In our framework, EX is the gold standard for reliability, while EM serves to indicate structural alignment with expert-level optimization. Table 3 summarizes these results across the TPC-H and Mapas benchmarks.

Table 3. Accuracy Comparison: Baseline vs. Athena Agent.

Dataset	Baseline		Athena Agent	
	EX	EM	EX	EM
TPC-H (6 queries)	83.3%	0%*	100%	100%
Mapas (8 queries)	75%	0%*	100%**	100%

Legend: **EX**: Execution Accuracy; **EM**: Exact Match.
*Baseline EM is 0% because the model failed to apply performance-driven heuristics independently.
**Success achieved through the Correction Agent and Grounding Module.

The results highlight a critical gap in zero-shot LLM performance. While the baseline could occasionally retrieve correct data (EX), it failed the **Exact Match (EM)** test in all complex cases. This occurred because the baseline generated "naive" SQL structures that ignored optimization principles such as **Sargability (H5)** or **Join Refactoring (H6)**. Conversely, the Athena agent achieved 100% EM and EX by consistently transforming raw natural language into the optimized SQL structure defined in our Golden Standard, demonstrating that integrating semantic embeddings and heuristic-driven prompts is essential for generating production-ready, efficient queries.

Beyond quantitative gains, the integration of semantic embeddings and the Correction Agent proved essential for ensuring query validity. Below, we analyze specific cases in which the baseline failed and the Athena architecture, particularly its Grounding Module, provided the correct structural solution.

Analysis 1: Semantic Mapping in TPC-H (Client vs. Customer)

The baseline model failed to maintain consistency when the synonym "client" was used. The DeepSeek-Coder baseline attempted a self-join that resulted in a `GroupingError` (Question 5). Athena utilized its Grounding Module to semantically link "client" to the `h_customer` table, generating a stable correlated subquery (Question 6).

Question 5: Baseline Failure (Synonym: "Client")

```sql
-- FAILED BASELINE QUERY (Baseline EX: 0)
SELECT
    c.c_custkey AS client_key,
    c.c_name AS client_name,
    MAX(o.o_orderdate) AS most_recent_order_date,
    o_recent.o_totalprice AS order_total_price
FROM
    h_customer c
JOIN
    h_order o ON c.c_custkey = o.o_custkey
JOIN
    h_order o_recent ON o.o_custkey = o_recent.o_custkey
                    AND o.o_orderdate = o_recent.o_orderdate
GROUP BY
    c.c_custkey, c.c_name, o_recent.o_totalprice
HAVING
    MAX(o.o_orderdate) = o_recent.o_orderdate;
-- Result: column "o_recent.o_orderdate" must appear in the GROUP
    ↪ BY clause
```

In contrast, the Athena agent utilized its Grounding Module to perform a semantic link between "client" and the `h_customer` table. This anchored the LLM to the correct schema context, leading to the generation of a stable correlated subquery (Question 6). By isolating the maximum date calculation within a `WHERE` clause, Athena avoided unnecessary joins and ensured 100% execution accuracy.

Question 6: Athena Successful Translation

```sql
-- SUCCESSFUL ATHENA QUERY (Athena EX: 1)
SELECT
    c.c_custkey,
    c.c_name,
    MAX(o.o_orderdate) AS most_recent_order_date,
    o.o_totalprice
FROM
    h_customer c
JOIN
    h_order o ON c.c_custkey = o.o_custkey
WHERE
    o.o_orderdate = (
        SELECT MAX(o2.o_orderdate)
        FROM h_order o2
        WHERE o2.o_custkey = c.c_custkey
    )
GROUP BY
    c.c_custkey, c.c_name, o.o_totalprice;
```

Analysis 2: Resolving Ambiguity and Syntax in Mapas Database

In the Mapas experiments, we encountered two scenarios where the baseline generation failed due to schema ambiguity. By enabling the embedding-based retrieval system, the agent accessed curated examples that guided it to the correct syntax. Below, we present the JSON context retrieved from the Vector DB, along with the final queries.

Correction Case A: Column Ambiguity In Question 3, the baseline failed to reference subquery columns correctly. The embedding mechanism retrieved a similar pattern using the `ANY` operator, which Athena used to fix the references before H1 optimization.

Retrieved Context (JSON):

```
{
 "Liste o ID e o numero das inscricoes...": "SELECT id, number FROM registration WHERE
        ↪ score > ANY (SELECT score FROM registration WHERE opportunity_id = 5 AND
        ↪ score IS NOT NULL) AND score IS NOT NULL;"
}
```

Final Optimized Query:

```sql
SELECT DISTINCT r1.id, r1.number
FROM registration r1
WHERE r1.score > (
    SELECT MIN(r2.score)
    FROM registration r2
    WHERE r2.opportunity_id = 5
      AND r2.score IS NOT NULL
)
AND r1.score IS NOT NULL;
-- Execution Time: 0.0873s
```

Figure 17. Case 1: Embeddings provided the logic, Heuristic H1 optimized execution.

Correction Case B: Complex Join Resolution In Question 6, the baseline hallucinated table aliases. The embedding system retrieved a complex example using Window Functions, allowing Athena to correctly map the `agent` and `project` tables.

Retrieved Context (JSON):

```
{
 "Para cada agente, qual e a data...": "WITH RankedProjects AS (SELECT agent_id, name,
        ↪ create_timestamp, ROW_NUMBER() OVER (PARTITION BY agent_id ORDER BY
        ↪ create_timestamp DESC) as rn FROM project) SELECT ... JOIN RankedProjects rp
        ↪ ON a.id = rp.agent_id WHERE rp.rn = 1;"
}
```

Final Correct Query:

```sql
SELECT
    a.id AS agent_id,
    a.name AS agent_name,
    MAX(p.create_timestamp) AS latest_project_date,
    p.name AS project_name
FROM agent a
INNER JOIN project p ON a.id = p.agent_id
GROUP BY a.id, a.name, p.name
ORDER BY latest_project_date DESC;
-- Execution Time: 0.1065s
```

Figure 18. Case 2: Embeddings corrected the schema mapping, enabling successful execution.

Summary of Findings These results confirm that while LLMs have high raw capacity, the integration of semantic embeddings and feedback loops is essential for handling real-world challenges like complex schema navigation and linguistic variety in Portuguese.

5.5 Discussion

It is important to clarify that the proposed agent does *not* intend to replace or interfere with the database management system (DBMS) 's internal query optimizer. 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 knowledge of the internal DBMS.

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.

The optimization heuristics implemented in Athena are grounded in mature, field-tested database principles that serve as core tuning guidelines for industry-standard DBMS. For instance, Heuristic H5 (Sargability) ensures that indexed columns remain isolated from functions to allow the optimizer to perform efficient index scans, a practice universally recommended in database administration manuals. However, the automated application of these rules involves technical trade-offs that must be acknowledged.

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 it is sent to the DBMS.

6 Conclusion and Future Work

This paper presents a text-to-SQL agent based on the *DeepSeek* model, which translates natural language questions into structured SQL queries. The agent integrates heuristic-driven optimization 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 expand our heuristics and perform additional experiments using a broader 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 by difficulty level (easy, medium, hard) and by investigating how the model handles linguistic complexity and schema intricacy in Portuguese. Furthermore, we aim to refine our use of embedding functions to improve word similarity analysis.

Funding

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

References

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., *et al*. (2020). Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901.

Chen, X., Lin, M., Schärli, N., and Zhou, D. (2023). Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

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. `https://www.deepseek.com/`.

Fan, A., Urbanek, J., Ringshia, P., Dinan, E., Qian, E., Karamcheti, S., Prabhumoye, S., Kiela, D., Rocktaschel, T., Szlam, A., *et al*. (2020). Generating interactive worlds

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

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

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

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*.

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*.

Li, J., Hui, B., Qu, G., Li, B., Yang, J., Li, B., Wang, B., Qin, B., Geng, R., Huo, N., *et al*. (2024). Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. In *Advances in Neural Information Processing Systems*, volume 36.

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

Ministério da Cultura (2025). Mapas culturais - funarte. `https://mapas.cultura.gov.br/`.

Nascimento, E. R. S. (2024). Querying databases with natural language: The use of large language models for text-to-sql tasks. Master's thesis, Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil.

OpenAI (2025). Chatgpt.

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

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.

Petrola, L., Brayner, A., and Franco, W. (2025). Heuristic-guided text-to-sql translation with llms: Optimizing natural language interfaces for relational databases. In *Proceedings of the 40th Brazilian Symposium on Databases (SBBD)*, Fortaleza, Brazil.

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

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)*.

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

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Chi, E., Le, Q., and Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837.

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, 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*.

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*.