November 2025

A Survey on LLM-based Querying vs SQL

Eduardo Cunha PG55939

Jorge Rodrigues PG55966

Tiago Rodrigues PG56013

Vasco Faria PG57905

Abstract

This report investigates the evolution of Text-to-SQL systems, tracing the transition from traditional SQL querying to modern AI-driven approaches powered by Large Language Models (LLMs). The study defines and compares three levels of Text-to-SQL capability: (1) Basic Text-to-SQL, where natural language is translated into SQL queries; (2) Context-Aware SQL Generation, which incorporates schema and domain knowledge for improved accuracy; and (3) Direct Data Access, where LLMs interact with structured data through retrieval-based or semantic methods instead of generating SQL. The evaluation is based on TPC-C and TPC-H workloads, combined with custom performance metrics designed to assess accuracy and efficiency across these levels. The results show that Level 1 systems produce structurally plausible but largely non-executable SQL, Level 2 systems achieve high execution accuracy when enriched with schema context and reasoning scaffolds, and Level 3 methods, despite offering an alternative to query generation, struggle with reliable aggregation, computation, and consistency at scale. Overall, the study provides a practical understanding of how LLM-driven database interaction can improve accessibility while highlighting the continued necessity of traditional database engines for correctness and performance.

Contents

1 Introduction

This report explores the evolution of Text-to-SQL systems and the broader use of relational databases, from direct SQL querying to AI driven and LLM-based solutions. Traditionally, users interact with databases using SQL, a structured query language. However, with recent advances in artificial intelligence, new methods have emerged: users can now ask LLMs to generate SQL queries automatically or even retrieve information directly from the data. These approaches are particularly appealing to users without SQL expertise and continue to show significant potential for growth.

The main goal of this report is to evaluate three different Text-to-SQL strategies, defined as follows:

- Level 1: Basic Text-to-SQL – LLMs translate natural language questions into SQL queries without direct access to data, relying mainly on pattern recognition.
- Level 2: Context-Aware SQL Generation – LLMs use database schemas, relationships, and domain context to improve query accuracy, often enhanced through prompt engineering.
- Level 3: Direct Data Access – LLMs interact directly with structured instead of SQL, enabling semantic search, data analysis, and cross-table reasoning.

The report evaluates these levels without using standard Text-to-SQL benchmarks, but instead performance tests based on TPC-C and TPC-H, combined with custom metrics. Each level includes sublevels reflecting different degrees of complexity.

The document is structured as follows: first, an overview of the topic; then, definitions of the three levels; a review of the state of the art for each level; a description of the benchmarks and metrics used; and finally, the results and analysis.

The methodology is based on a review of academic papers and existing research on Text-to-SQL systems and database performance evaluation.

2 Overview

To grasp the challenges and opportunities in Large Language Model-based querying versus traditional SQL, we need to first outline the main concepts and systems that define this area.

The SQL language started to obtain commercial relevancy in the early 1980s with the implementations of companies such as Oracle or IBM. SQL was designed so that ordinary people could “walk up and use it” without an IT degree [1]. However, with the overtime evolution of database schemas, the complexity of making a query becomes a task that requires advanced knowledge on the topic. According to Taipalus et al. (2020)[2], for students new to the concept of SQL, when the number of query concepts, expressions, or database objects in a task increases, it leads to an increase in errors. Usually, experts can deconstruct an answer into segments of operations such as aggregations and filters and identify meaningful entities present in the text. That, combined with the knowledge of that database schema, makes the composition of a query possible. This process corresponds to the flow of human Text-to-SQL.

2.1 Text-to-SQL

Text-to-SQL is a critical task in Natural Language Processing (NLP) aimed at bridging the gap between non-technical users who require data and the specialized technical knowledge necessary to retrieve it using SQL. This process involves translating natural language (NL) queries, often given by users without SQL proficiency, into structured SQL commands. Organizations rely on this capability as the volume and complexity of structured information in relational databases grow, making manual data exploration impractical [3].

The Text-to-SQL systems allows users to submit a natural language request, which the system automatically generates the necessary SQL query, making data extraction more accessible for non-technical users across different systems [3]. According to Conn (2005)[4], databases are a fundamental and central piece for many information systems. These systems can be divided into decision support systems and executive information systems, which usually relate to the concepts of Online Analytical Processing (OLAP) and Online Transaction Processing (OLTP), respectively. OLTP databases are built to store atomic values, which means each field holds only one piece of data instead of multiple values. As their name suggests, they work well for handling transactions. On the other hand, OLAP databases are usually organized around aggregated and summarized information to help with analysis and reporting. Understanding these two common types of systems is fundamental for solving Text-to-SQL because it helps to trace the profile of who uses the tool. For decision support systems, we might find more business intelligence-related personnel, so these professionals would be less knowledgeable on SQL concepts compared to the developers of executive information, who usually are experts in IT concepts.

2.2 Large Language Models

Large language models (LLMs) are Transformer-based models with hundreds of billions, or more, parameters. They are trained on huge text datasets. Some of the best-known examples include GPT-4 and LLama, with the latter representing the open-source alternatives. These models show strong skills in understanding NL and solving complex problems through text generation [5]. Often, these models show top performance in many NLP tasks, including question answering, summarization, translation, dialogue, code generation, and various forms of reasoning. Acording to Hong et al. (2025)[6], LLMs benefit from their strong semantic parsing capacity to help them with different aspects of Text to SQL tasks:

- **Question understanding**, which involves recognizing the NL question as the semantic representation of the user's intent
- **Schema comprehension**, which necessitates that the system understand the database schema, usually complicated by the complexity
- **SQL generation**, which incorporates the parsing resulting from the first two steps to predict the correct syntax and generate an executable SQL query capable of retrieving the required answer

Context is one of the most determinant factors in the quality of the content produced by the models. The Retrieval-Augmented Generation (RAG) framework appeared as a practical solution to tackle the problems LLMs encounter with domain-specific questions. These questions often lead to mistakes or hallucinations. RAG was introduced in the middle of 2020 as a notable improvement for generative tasks. Since then, the field has grown rapidly and gained significant attention from researchers. RAG works by integrating external data retrieval into the text generation process. This starts with the LLM searching an external data source to find relevant information before producing text or answering questions. By retrieving information from knowledge bases during inference, RAG offers a more informed and evidence-based approach. This significantly improves the accuracy and relevance of the generated content and reduces the risk of hallucinations [7].

3 Methodology

The integration of LLMs with SQL and structured data can be divided into three levels, each showing a greater ability to interact with and process database information.

1. **Level 1: Basic Text-to-SQL** translation is the starting point where LLMs translate natural language into SQL queries. At this level, the model takes natural language questions and creates SQL queries. The LLM mainly relies on pattern recognition to produce correct queries that can run on databases.

2. **Level 2: Context-Aware SQL Generation** builds on the first level by adding more contextual information during translation. This includes detailed database schemas, table relationships, sample data, and specific domain knowledge. Level 2 approaches often use prompt engineering to improve the accuracy of queries and manage more complex database structures. The LLM still generates SQL queries, but it has a much better understanding of the database context.

3. **Level 3: Direct Data Access** represents the most advanced approach, where LLMs interact directly with tabular data. Instead of creating SQL queries, structured data becomes vectors and is stored in retrieval systems. This setup allows LLMs to access and understand real data content using semantic search combined with retrieval methods or via parameter training as a means for adaptation to the data. With this approach, LLMs can perform tasks like answering questions about tables, analyzing data across multiple tables, and gaining insights from data without relying on traditional database queries.

Each level meets different needs and complexities. Level 1 focuses on generating queries efficiently, Level 2 emphasizes contextual accuracy, and Level 3 prioritizes direct analytical abilities.

3.1 Research Methodology

To analyze current approaches to integrating LLMs with SQL and structured data, a systematic academic review was conducted for each of the three levels outlined in the methodology framework. The goal of this review was to identify the techniques, architectural patterns, and practical implementations used in real-world systems across varying levels of complexity.

3.2 Basic Text-to-SQL

In Text-to-SQL systems, the most basic scenario assumes that users provide only a natural language request, with no access to database schemas, sample data, or query execution feedback. In this setting, the system is expected to generate the corresponding SQL query directly from the user's request. The focus lies on testing whether LLMs can transform plain questions into executable queries based solely on their internal knowledge and pattern-matching capabilities, without relying on external structural or contextual information.

This simplest form of interaction is referred to as the Naive Level. At this level, non-expert users interact with the model in a direct and intuitive way, posing NL requests that are immediately translated into SQL. The translation relies primarily on lexical associations, memorized patterns, and generalization from training data, making the approach simple and accessible. However, it is also highly prone to ambiguities, inconsistencies, and errors, particularly when queries are complex or underspecified. Despite these limitations, the Naive

Level establishes a baseline for evaluating the raw ability of LLMs to connect NL intent with SQL query generation in the absence of explicit contextual support.

Within the Naive Level, several prompting strategies can be employed to examine how LLMs interpret user requests and generate SQL statements in the absence of contextual information. Although no schema, data, or feedback is provided, the phrasing and structure of the prompt can still influence the model's reasoning process and output quality. Subtle differences in linguistic formulation may affect how the model infers intent, selects relevant concepts, and constructs the corresponding query.

Another significant point is to acknowledge that LLMs may produce different outputs for the same prompt across separate sessions. Since these models do not retain memory of prior interactions, they often rely on transient internal reasoning and probabilistic associations to fill in missing contextual details. In doing so, the model makes what can be described as logical assumptions to infer plausible table names, column labels, or query structures based solely on linguistic cues and prior statistical patterns.

Exploring these strategies allows for a better understanding of the model's internal mechanisms and its reliance on learned linguistic patterns rather than explicit contextual cues. This analysis also provides insight into the inherent limitations and potential biases that emerge when language models operate purely from textual input, serving as a foundation for comparing more advanced interaction levels where contextual or structural guidance is introduced.

It is important to note, however, that most existing research in the Text-to-SQL domain assumes the presence of at least the database schema as part of the input context. Consequently, studies focusing exclusively on schema-free interactions where models must rely solely on linguistic cues and prior knowledge remain scarce. This lack of dedicated resources and empirical evaluation makes the Naive Level an underexplored but valuable research direction, as it isolates the linguistic and reasoning abilities of large language models from any structural or contextual dependencies.

As a result, identical NL requests may yield slightly or even significantly different SQL statements when issued in distinct sessions, reflecting the model's adaptive reasoning and inherent variability. This behavior highlights both the flexibility and the instability of schema-free Text-to-SQL generation, reinforcing the importance of consistency analysis when evaluating model performance at the Naive Level.

3.2.1 Direct Prompting

Direct prompting represents the most fundamental form of interaction within the Naive Level. In this approach, the user provides a straightforward NL instruction, and the model is expected to generate the corresponding SQL query directly. Prompts in this method are typically concise and unstructured, lacking any role specification, illustrative examples, or additional meta-guidance.

This method serves as a baseline for evaluating the model's inherent ability to interpret NL and produce executable SQL statements without relying on any external context. By isolating the interaction from schema information, sample data, or feedback, direct prompting highlights the extent to which large language models can leverage prior training, pattern recognition, and internalized knowledge of database concepts to perform query translation.

Example:

User request: `“List all customers”`

Model output : `SELECT * FROM customers';`

Although simple and intuitive, this method is particularly sensitive to ambiguities and underspecified requests, as the model must infer the intended query solely from the linguistic input. As such, outputs may be inconsistent or partially incorrect, especially for complex queries or less common SQL operations.

3.2.2 Prompt Engineering

Prompt engineering refers to the deliberate design and phrasing of user instructions to effectively guide a language model's reasoning process. Even without access to explicit data or schema information, the way a question or request is structured can strongly influence how the model interprets the task and generates its response.

At the Naive Level, prompt engineering focuses on exploring how different formulations of the same task affect the quality, consistency, and correctness of the generated output. This involves experimenting with linguistic cues such as sentence structure, specificity, and the order of instructions. All of which can subtly shift the model's internal reasoning and output format.

For instance, a simple query like this one might produce a correct but basic result:

`“Write an SQL query that retrieves all product names and prices where the price exceeds 100. Sort the results from highest to lowest price.”`

However, rephrasing it to:

`“I'm a data analyst working on our weekly premium-product report. Generate an SQL query that selects the product name and price from the table products. Only include rows where the price is greater than 100, and order the results by price in descending order. Make sure the query follows standard SQL syntax.”`

Often leads to a more complete and well-structured SQL statement. These small linguistic adjustments help clarify intent and reduce ambiguity, allowing the model to generate more accurate, efficient, and domain-appropriate responses.

The primary insight gained from prompt engineering is that output quality is highly sensitive to the phrasing of the request. Even minor wording changes can improve the precision and readability of the resulting query, offering valuable understanding of the model's reasoning behavior, bias patterns, and generalization capabilities.

3.2.3 Iterative Refinement

Iterative refinement extends the Naive Level interaction by introducing multiple rounds of dialogue between the user and the model. Unlike the naive scenario, after the model generates its first SQL query, the user provides execution feedback in the form of error messages or runtime issues, allowing the model to correct and improve its query. The model is allowed to revise its output a maximum of two times based on this feedback.

This approach tests the model's ability to adapt and improve its output using a combination of linguistic cues and explicit execution feedback. When a generated query is incomplete, inaccurate, or fails to execute, the user shares the resulting errors, prompting the model to refine its previous responses.

Example:

User Input (1): `“Generate an SQL query to list all customers.”`

Model Output (1): ``` `SELECT name email FROM customer WHERE country = 'Portugal';` ```

User Input (2): `“PostgreSQL error: ERROR: syntax error at or near 'email'”`

Model Output (2): ``` `SELECT name, email FROM customer WHERE country = 'Portugal';` ```

User Input (3): “PostgreSQL error: ERROR: column ‘country’ does not exist”

Model Output (3): `` `SELECT name, email FROM customer WHERE nation = 'Portugal'; ``

This interaction illustrates how the model iteratively refines its output in response to both execution errors and successive natural language cues, improving accuracy and relevance even without explicit schema knowledge.

Methodologically, this strategy highlights the model’s adaptive reasoning capability, its ability to incorporate runtime feedback and textual guidance, and its effectiveness at incremental query improvement within a limited number of iterations.

3.3 Context-Aware SQL Generation

At the second level, relational databases and LLMs communicate in a way that goes beyond straightforward language translation and takes into account the database structure. The system is also given metadata that specifies the database (tables, columns, primary keys, foreign keys, etc.) so that it doesn’t have to rely solely on the model’s existing knowledge. This is a big step up from the last level since the additional context makes it easier to relate NL queries to the database’s actual structure. The LLM can better comprehend user intent, map words to the appropriate attributes, and create SQL queries that represent the logical relationships recorded in the schema with the use of that structural knowledge.

3.3.1 Prompting

One of the most widely used and relevant approaches to improving performance in generating accurate queries at this level is prompt engineering, which consists of designing detailed instructions (prompts) to guide the LLM to produce more accurate and consistent queries. The main advantage of this approach is that it can improve performance without requiring fine-tuning or additional training, which would typically entail high costs. Additionaly, it is easily adaptable to different databases or domains. By carefully structuring the prompt, the risk of incorrect or poorly reasoned responses is reduced, ensuring that the model better aligns with the user’s actual intent.

Within the in-context learning paradigm, which exploits the ability of LLMs to learn from examples provided in the prompt itself, there are two main variants in the context of Text-to-SQL:

- **Zero-shot prompting:** The model only receives the user’s question and the database schema, without any prior question–SQL examples. This technique allows the system to be applied immediately to new domains, but its accuracy can be limited, especially when dealing with more complex queries. [8].

- **Few-shot prompting:** In addition to the question and the schema, the model also receives a few examples of questions and the respective SQL queries. These examples act as demonstrations that help the LLM better understand the expected format and reasoning process, generally resulting in higher accuracy, particularly in more challenging scenarios [8].

3.3.2 Decomposition

Another approach that promises to improve the performance of few-shot prompting is task decomposition. Decomposition-based methods reduce the complexity of text-to-SLQ by dividing tasks or questions into simpler, more manageable components [6], allowing them to more effectively address the challenges that few-shot prompting faces when dealing with complex SQL queries, especially those involving schema linking, multiple joins, or nested queries.

Based on this principle, Pourreza and Rafiei (2023) [9] proposed the **DIN-SQL** (Decomposed In-Context SQL) method, which breaks the problem into smaller subproblems, solves each subproblem, and uses these solutions to construct the solution to the original problem. This approach promotes a more structured, interpretable, and reliable reasoning process, taking full advantage of the reasoning capabilities of LLMs.

DIN-SQL consists of four main modules. The first module, **Schema Linking**, identifies and associates relevant parts of the natural language query with elements of the database schema, such as tables, columns, and values. The second module, **Classification & Decomposition**, analyzes the complexity of the query and classifies it into different categories (easy, non-nested complex, nested complex). Based on this classification, the task is decomposed into subtasks appropriate to the query type, allowing the system to select the most appropriate prompting strategy for each case. The third module, **SQL Generation**, generates the SQL query itself, using the information produced in the previous steps. Depending on the level of complexity identified, the model can adopt intermediate representations, such as NatSQL, which simplifies the query structure and reduces ambiguities during generation. Finally, the **Self-Correction** applies an automatic review to detect and fix syntax errors, redundancies, or unnecessary functions.

Although DIN-SQL achieved considerable improvements over traditional methods, especially for complex queries, its performance depends heavily on the quality of the schema linking and the decomposition step. Additionally, because the method requires multiple sequential calls to the model for each module and subtask, it can be slower and more resource-intensive than standard few-shot prompting. Nevertheless, the results indicate that modular decomposition is a promising approach for enhancing the reliability and interpretability of text-to-SQL models.

3.3.3 Reasoning

Recent studies have shown that the reasoning ability of LLMs can be improved through specific prompting strategies. One of the most effective among them is chain-of-thought prompting (CoT), which guides LLMs to generate intermediate reasoning steps before providing the final answer. This structured approach allows complex problems to be decomposed into smaller, more manageable steps.

According to Wei et al. (2023)[10], LLMs can learn to reason through a few-shot examples organized as triples (input, chain-of-thought, output). If, in the examples provided in the prompting, we include not only the question and the answer but also the reasoning process that connects them, the model learns to systematically decompose the tasks and solve each step, as shown in the examples, before generating the final output. This few-shot CoT prompting showed good results, demonstrating the reasoning potential of LLMs.

Based on this idea, Kojima et al. (2023) [11] proposed a zero-shot version of CoT to demonstrate that LLMs can also perform step-by-step reasoning without the help of any examples. The idea is to add the phrase “Let’s think step by step.” to the prompt, so that the model automatically produces intermediate reasoning before giving the final answer. This zero-shot CoT approach revealed that reasoning capabilities are already latent in LLMs and can be activated through minimal prompt design.

Despite the benefits that CoT brings in few-shot settings, as shown by Wei et al. (2023)[10], this methodology requires considerable manual effort to carefully select examples and annotate the reasoning steps for each one. Furthermore, the examplars used are fixed and do not adapt dynamically to each new test case.

To overcome these limitations, Zhang et al. (2023) [12] proposed **ACT-SQL**, a method that automates the generation of CoT prompts for text-to-SQL tasks. This approach introduces a hybrid exemplar selection strategy for few-shot prompting: part of the examples (static exemplars) are randomly selected and remain fixed across all tests, while another part (dynamic exemplars) are dynamically selected for each question based on the semantic similarity between the new question and the questions in the training set. Following the recommendation of Kojima et al. (2023)[11], each prompt begins with the phrase “Let’s think step by step” to encourage step-by-step reasoning. Furthermore, ACT-SQL proposes an automatic CoT generation method, auto-CoT, inspired by the schema linking technique. In this process, the model automatically identifies and associates segments of the question (slices) with relevant database entities, namely tables and columns, in the format [table].[column]. For each column and table mentioned in the target SQL query, the semantic similarity is computed between the query fragments and the corresponding question segments, linking the most relevant fragment to each schema entity. This way, the prompt incorporates a structured reasoning trace that mirrors the relationship between natural language and database structure, without the need for manual annotation. This approach achieved strong results, demonstrating that it is possible to combine the reasoning effectiveness of CoT with the scalability of a fully automated process. Compared to DIN-SQL, ACT-SQL has the advantage of requiring only a single API call to generate the SQL query, instead of multiple calls for each module, resulting in significant cost and time savings.

3.4 Direct Data Access

The third level of LLM interaction with SQL and structured data represents a significant shift. In this stage, LLMs have direct access to actual data content instead of just metadata or schema details. Unlike earlier levels that focus on generating queries or translating schemas, Level 3 lets LLMs process, examine, and reason directly over tabular data. This approach results in better insights and responses.

This change affects how we work with structured information. Instead of generating SQL queries that must be executed separately, Level 3 methods allow LLMs to use tabular data directly. This leads to more complex reasoning tasks, such as cross-table analysis, detailed aggregations, and explanations of data patterns in everyday language. The LLM functions as both the query engine and the analytical interpreter. It understands data relationships, identifies trends, and produces insights that are easy for humans to read, all without needing traditional database query languages.

However, this increased capability also brings unique challenges. These include concerns about scaling with large datasets, privacy issues when exposing raw data to LLMs, and the need for effective data representation methods that allow LLMs to process structured information efficiently while keeping performance and accuracy high.

3.4.1 Table Question Answering

Table Question Answering (Table QA), is a task in NLP that focuses on providing precise answers to user queries by understanding and reasoning over tabular data [13]. Unlike knowledge base question answering, which targets fully s tructured data, or text-based question answering, which handles unstructured passages, Table QA operates on semi-structured or structured tables, such as those found in financial reports, web pages, or databases. Approaches in this field include semantic-parsing-based methods that translate questions into executable queries, as well as generative, extractive, matching-based, and retriever-reader models that directly interact with table contents to yield answers. As Jin et al. (2022)[13] emphasize in their survey, Table QA represents a critical technique for unlocking the information embedded in tables while posing unique challenges related to table encoding, numerical reasoning, and inference across tabular structures.

Acording to D. Min et al. (2024)[14], two key approaches currently employed for enhancing the performance of LLMs in domain-specific QA: Domain-Specific Fine-Tuning (DSFT) and Retrieval-Augmented Generation (RAG)

1. **DSFT QA** enhances the performance of LLMs in specialized QA by directly training the models on a domain-specific corpus. This process involves first pre-training an original LLM on the domain data to enable the incremental learning of domain knowledge, yielding an updated foundation model. Subsequently, this updated model is further fine-tuned using instruction tuning on an instruction set tailored for the QA task, resulting in the final QA-oriented model [14].

2. **RAG** is a prompt-based method for improving LLMs in specific areas of QA. It uses a domain-specific corpus as an external knowledge base to support the model's answers without changing the model's parameters. This approach includes several steps. First, the domain corpus is divided into smaller text chunks. Next, these chunks are turned into semantic vectors and stored in an indexed vector store for quick access. When a query comes in, the RAG system finds the most relevant text chunks based on their similarity to the query vector. Finally, these retrieved chunks, along with the original question, are fed into the LLM to create the final answer [14].

Both described paradigms require table-to-text methods to allow the systems to work. For that reason, is imperative to choose the most reliable system considering the available resources.

Paradigm	Pros	Cons
DSFT	Embeds domain-specific knowledge directly into the LLM parameters through pre-training and instruction tuning.	Requires a lot of computer power and incurs high costs due to extensive training
RAG	Lower cost method as it does not require updating model parameters. Uses the domain corpus as an external knowledge base, making knowledge updates potentially easier.	Building a vector retrieval library demands substantial memory resources. Performance is critically dependent on retrieval accuracy and the quality of semantic representations in the text chunks

Table 1: Pros and Cons of each Paradigm

When choosing a cheaper solution that doesn't sacrifice results, the sources recommend focusing on the RAG approach. Considering the cost, RAG is naturally preferred because it is less expensive. It uses a prompt-based system, which avoids the high costs of updating model parameters. In contrast, DSFT demands significant computational power and expenses for pre-training and fine-tuning large models. RAG also provides easier options for preparing the domain corpus. For RAG systems, the straightforward Markdown format for table-to-text conversion serves as a good alternative to more complex methods. The Markdown approach is simple and can be processed quickly with scripts. It does not require model training or manual input, needing only CPU resources.

Several practical solutions for handling tabular and structured data have emerged from academic and non-academic sources. These approaches vary in complexity and architecture, but they all focus on either transforming tables into retrievable text chunks that can be dynamically fed into an LLM or bettering the training data of the model. Below are some representative options to consider.

- **TableRAG** is an efficient RAG framework proposed by Chen et al. (2024)[15], designed for LLM-based table understanding. It addresses the scalability issues of very large tables, including those with millions of tokens that often go beyond LLM context limits. The framework uses a special schema cell retrieval method improved by tabular query expansion. In this approach, the model creates precise queries to pull only the most relevant column names and cell values required to answer a question. By limiting the information given to the LM to this important subset, TableRAG greatly cuts down on prompt lengths and token use while keeping retrieval quality high.

- **Table-GPT** is a model created through a fine-tuning process. It aims to improve how language models like GPT-3.5 and ChatGPT understand and perform various table tasks, including question answering. This fine-tuning solves the issue that standard language models often struggle with table-related tasks. They are usually trained mostly on one-dimensional natural language texts, while relational tables are two-dimensional. The approach involves ongoing training of these models using various table tasks based on real tables. As a result, Table-GPT models show better table-understanding abilities and consistently perform better than untuned versions across many table tasks [16].

After looking at current solutions, we can see practical limitations that impact their use in our situation. For DSFT QA, the biggest challenge is the high computational and financial cost. Training or fine-tuning large models takes a lot of GPU resources and time. Additionally, testing across various datasets would require training separate models for each domain, which would raise the costs even more. On the other hand, RAG-based solutions are usually cheaper, but they often rely on proprietary APIs or vector databases that are behind paywalls. This can limit access for projects with tight budgets or open-source needs.

3.4.2 Hybrid Solutions

End-to-end Table QA (E2E TQA) described in the prior subsections aims to ignore the generation of SQL queries as an intermediate step, skipping directly to the prediction of the final answer [17]. The solutions discussed in the "Table Question Answering" subsection have several concerns that may prevent commercial solutions from fully fitting the paradigm. Some of the issues are:

1. **Scalability**: a vector database cannot handle transactions, so two databases would be needed for an OLTP system.
2. **Data privacy**: some of the most powerful models are closed source and only accessible through APIs. This means some companies may not benefit from the best LLM performance due to data privacy and governance issues.
3. **Deterministic and auditable**: without the intermediate step of SQL code, there is no way to trace back the response to confirm the answer because of how the LLM operates as a black box.

For the industry, developing a solution in the Table QA paradigm is a risk that could drive away many promising clients. Many academic and open-source projects have chosen to build hybrid solutions: using text-to-SQL for data retrieval from large databases, Table QA for analysis and explanation of the retrieved results, and sometimes other complementary approaches.

- **H-STAR** is an algorithm that combines SQL logic and semantic methods in a two-stage LLM-driven process. It aims to tackle the challenge of merging language understanding with structured data analysis in tabular reasoning. The first stage, table extraction, uses a step-by-step multi-view chain to filter tables effectively. It starts by identifying the relevant columns, using both the original table and its transposed form. After that, it extracts the relevant rows, which helps focus the LLM on the right context. In the second stage, adaptive reasoning, the LLM's language comprehension skills help guide the choice of the best reasoning strategy based on the type of question. Specifically, this method consistently uses semantic reasoning for direct lookup and complex lexical queries. It enhances textual reasoning with support from symbolic reasoning, such as an extra SQL step for quantitative, mathematical, and logical tasks. The final step of this approach involves text-based reasoning, which acts as the Table QA step, using the retrieved query-specific table to analyze the context and generate the final textual answer or explanation [18].

- **SYNTQA** is an approach designed to integrate the complementary strengths of the two main strategies for Table QA: Text-to-SQL and E2E TQA. The core idea is to aggregate candidate answers from both a Text-to-SQL model and an E2E TQA model and then employ an answer selector to choose the most probable correct final answer. This synergistic method is motivated by the finding that Text-to-SQL models excel in handling arithmetic operations and processing long tables, while E2E TQA models are superior in addressing ambiguous questions, non-standard table schema, and complex table contents. SYNTQA's answer selection mechanism can be implemented either through a feature-based selector that utilizes features like question and answer characteristics or via an LLM-based selector that leverages few-shot capabilities or heuristic-enhanced prompting [17].

4 Benchmarks

Benchmarking is essential for evaluating Text-to-SQL systems. With the rapid advancement of LLMs, robust benchmarks are critical for measuring progress, identifying areas for improvement, and guiding both research and practical deployment. Typically, Text-to-SQL benchmarks use standardized datasets containing pairs of NL questions and their corresponding SQL queries, enabling researchers to rigorously and reproducibly compare different algorithms.

Although benchmarks such as SPIDER and BIRD have driven significant progress in Text-to-SQL research, they have notable limitations. These benchmarks often ignore aspects like query efficiency and optimization, focusing mainly on surface-level accuracy rather than practical execution. Relying solely on such benchmarks can give a skewed perspective, highlighting the need for more diverse, challenging, and realistic datasets, as well as more comprehensive metrics that include execution accuracy and efficiency [19].

All the models discussed in previous sections have already been evaluated using these types of benchmarks. Therefore, the objective here is to explore how traditional database benchmarks, such as TPC-C and TPC-H, can be integrated into this context.

4.1 Text-to-SQL Benchmarks

The most robust benchmarks in the text-to-SQL field are Spider and BIRD. Both employ structured datasets paired with NL question descriptions to systematically evaluate a model's ability to translate NL inputs into SQL queries and accurately retrieve information from relational databases.

4.1.1 SPIDER

SPIDER is a Text-to-SQL evaluation framework designed to handle diverse datasets from both cloud and local sources across multiple database systems and SQL dialects. It supports complex operations such as data transformation and analytical queries [20].

The latest version, SPIDER 2.0, evaluates 632 tasks inspired by real-world enterprise scenarios. These tasks involve large-scale databases often containing more than 1,000 columns from systems like BigQuery and Snowflake. To solve them, models must understand extensive metadata, adapt to different SQL dialects, interpret project-specific code, and manage long contextual inputs. Additionally, they need to perform advanced reasoning and generate multi-query solutions that can exceed 100 lines of SQL code [20].

4.1.2 BIRD

BIRD is the first large-scale, cross-domain benchmark designed to bridge the gap between academic research and practical text-to-SQL applications. Unlike SPIDER, BIRD prioritizes database contents alongside schemas, introducing new challenges such as integrating external knowledge, managing data cleanliness, and ensuring query efficiency on large databases. Models must go beyond semantic parsing and understand database values to generate accurate SQL [21].

The benchmark features 12,751 text-to-SQL pairs across 95 databases in 37 domains (33.4 GB), including schema descriptions, full SQLite databases, curated queries, external knowledge evidence, gold-standard SQL annotations, and tools for both inference and model fine-tuning [21].

This setup allows for rigorous assessment of model performance on realistic, data-rich scenarios where deep understanding of database values is essential [21].

4.2 Metrics

The performance of Text-to-SQL models is evaluated using several key metrics that assess the accuracy, execution quality, and efficiency of the generated SQL queries. These metrics quantify a model's ability to convert NL questions into syntactically valid and executable SQL statements within a specific database schema. Commonly adopted evaluation metrics include Exact Matching Accuracy (EM), Execution Accuracy (EX), Test-suite Accuracy (TS) and Valid Efficiency Score (VES), collectively capturing surface-level correctness, execution robustness, and practical efficiency [22].

4.2.1 Exact Matching Accuracy (EM)

Exact Matching Accuracy (EM) measures whether the SQL query generated by the model is identical to the reference query, providing a stringent assessment of Text-to-SQL performance. This metric requires a character-for-character match between the produced SQL and the ground-truth query. However, because SQL often permits multiple syntactically distinct queries that yield the same results, Exact Matching Accuracy can be overly restrictive and may not capture cases where the model produces semantically equivalent alternatives [22]. This metric can also be used in an SQL-free context where, instead of using a reference query, we use a reference output value, like a table or a numerical value.

$$\text{Exact Matching Accuracy} = \frac{1}{N}\sum_{i=1}^{N} I\left(\hat{Y}_i = Y_i\right)$$

Where:

N represents the total number of NL questions used in the evaluation.

$\hat{Y}_i$ is the output generated by the model for the i-th NL question.

Y_i is the correct output for the i-th NL question, serving as the reference standard.

I is the indicator function, if the generated output $\hat{Y}_i$ exactly matches the reference output Y_i, the indicator function I equals 1; otherwise, it equals 0.

4.2.2 Execution Accuracy (EX)

Execution Accuracy (EX) measures whether the SQL query generated by a Text-to-SQL model, when executed on the target database, produces the same result as the ground-truth query. In contrast to EM, this metric evaluates semantic equivalence, accepting queries that yield identical outputs regardless of syntactic differences. Execution Accuracy is widely adopted, as it directly aligns with the primary objective of Text-to-SQL systems: delivering the correct answer to a NL question, independently of the precise structure of the generated query [22].

$$\text{Execution Accuracy} = \frac{1}{N}\sum_{i=1}^{N} I(f(Q_i, S_i) = A_i)$$

Where:

N is the total number of queries.

Q_i denotes the i-th NL question.

S_i denotes the database schema corresponding to the i-th question.

$f(Q_i, S_i)$ denotes the result returned by executing the SQL query generated by the model for the i-th question on the database schema S_i.

A_i denotes the reference answer for the i-th question.

I is the indicator function, which equals 1 if the condition inside is true, and 0 otherwise.

If, instead of evaluating against a fixed database schema (S_i), the queries were assessed on a variety of different databases, the metric would correspond to Test-Suite Accuracy (TS). This approach extends the evaluation to measure a model's ability to produce correct results across diverse database scenarios, providing a more robust assessment of generalization and semantic consistency [22].

4.2.3 Valid Efficiency Score (VES)

The Valid Efficiency Score (VES) is a commonly used evaluation metric in the text-to-SQL domain that measures both the correctness and efficiency of generated SQL queries. VES assesses:

1. Query Validity (Correctness): Whether the generated SQL query executes successfully and returns the same result as the ground-truth query.
2. Query Efficiency: The relative efficiency with which the generated query executes compared to the ground-truth query.

By integrating both accuracy and execution efficiency, VES offers a more comprehensive evaluation than correctness metrics alone [22].

$$\text{Valid Efficiency Score} = \frac{1}{N}\sum_{i=1}^{N} I(f(Q_i, S_i) = A_i) * \frac{T^{\text{gold}}}{T^{\text{gen}}}$$

Where:

N represents the total number of queries;

Q_i denotes the i-th NL question.

S_i denotes the database schema corresponding to the i-th question.

$f(Q_i, S_i)$ denotes the result returned by executing the SQL query generated by the model for the i-th question on the database schema S_i.

A_i denotes the reference answer for the i-th question.

I is the indicator function, equal to 1 if the generated SQL is equal to the ground truth, and 0 otherwise;

$T_{\{\text{gold}\}}$ is the execution time of the ground truth SQL query;

$T_{\{\text{gen}\}}$ is the execution time of the generated SQL query.

4.2.4 Structural Similarity (SS)

Structural Similarity (SS) is a metric designed to quantify the degree to which the syntactic structure of a generated SQL query aligns with that of a reference query [23]. Unlike EM or EX, which focus respectively on character-level equality or execution results, SS evaluates the structural composition of queries, capturing differences in the arrangement of clauses, expressions, and operators.

In practice, SS is computed by first parsing both the reference and generated SQL queries into Abstract Syntax Trees (ASTs), which represent the hierarchical syntactic structure of the queries. Each node in the AST corresponds to a logical or syntactic element of the query, such as SELECT, WHERE, GROUP BY, JOIN, or aggregate functions like SUM and AVG.

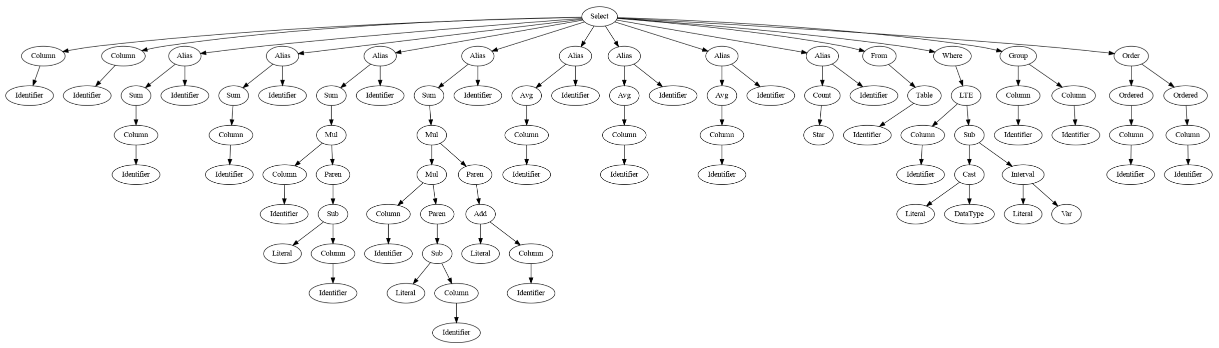

Figure 1: Example of an AST representing the hierarchical structure of an SQL query.

To better understand how SS operates, consider the AST shown in Figure 1. An AST is a hierarchical representation of the syntactic structure of a query, where each node corresponds to a logical or grammatical construct. The edges capture parent–child relationships, indicating how clauses and expressions are nested within one another.

Once the ASTs are obtained, the metric compares the two trees by analyzing the types and counts of nodes, capturing both the presence and frequency of syntactic elements. A common approach is to compute a weighted similarity based on the intersection and union of node counts:

$$\mathbb{S}_i = \frac{\sum_{\{tin \text{ Types}\}} \min\left(C_{i[t]}^{\text{ref}}, C_{i[t]}^{\text{gen}}\right)}{\sum_{\{tin \text{ Types}\}} \max\left(C_{i[t]}^{\text{ref}}, C_{i[t]}^{\text{gen}}\right)}$$

Where:

- "Types" is the set of all node types appearing in either the reference or generated AST for query i.
- $C_{i[t]}^{\text{ref}}$ and $C_{i[t]}^{\text{gen}}$ are the counts of nodes of type t in the reference and generated ASTs, respectively.
- $\mathbb{S}_i$ measures the structural similarity for the i-th query, ranging from 0 (no structural overlap) to 1 (perfect structural match).

The overall Structural Similarity across the dataset of N queries is then computed as the mean of individual query scores:

$$\text{Structural Similarity} = \frac{1}{N} \sum_{\{i=1\}}^{\{N\}} \mathbb{S}_i$$

This metric provides a fine-grained assessment of a model's ability to replicate the logical and syntactic organization of queries, even when minor differences in column names, aliases, or literal values exist. SS therefore complements EM and EX, offering an additional perspective on the quality of query generation in Text-to-SQL systems.

4.2.5 Token F1 (F1)

Token F1 is one of the most widely adopted metrics in QA evaluation. It measures the degree of token-level overlap between a generated answer and the reference answer. Unlike EM, which requires identical strings, F1 rewards partial matches by computing the harmonic mean between token precision and token recall. This makes F1 less brittle than EM, but it still suffers from several systematic weaknesses [24]

Given a set of reference answers A_r, a model-generated answer c, and a tokenization function $t(c\ \cdot)$, the F1 score is defined as:

$$F1(c, A_r) = \max_{\{a \in A_r\}} \left(\frac{|t(a)|}{|t(a) \cap t(c)|} + \frac{|t(c)|}{|t(a) \cap t(c)|} \right)^{-1}$$

Where:

- $t(a)$ and $t(c)$ are the token sets for the reference and candidate answers,
- Precision measures how many tokens in c appear in a,
- Recall measures how many tokens in a are recovered by c,
- The maximum is taken across all reference answers to account for datasets with multiple gold annotations.

Although Token F1 mitigates some of the strictness of EM, several limitations have been documented. These issues cause Token F1 to systematically underestimate true answer quality and fail to discriminate between semantically correct and incorrect answers in many cases, motivating the development of more nuanced evaluation metrics such as Answer Equivalence and learned matching models [24].

4.2.6 Mean Absolute Error (MAE)

Mean Absolute Error (MAE) is a fundamental evaluation metric widely used in regression, forecasting, and numerical prediction tasks. It quantifies the average magnitude of errors between predicted values and ground-truth targets, without considering their direction. MAE is valued for its interpretability and robustness, as it measures deviations directly in the original scale of the output variable. Unlike Mean Squared Error (MSE), which disproportionately penalizes large errors due to the squaring operation, MAE treats all deviations linearly, making it less sensitive to outliers and more suitable for scenarios where uniform error weighting is desired [25].

Given a set of N prediction–ground truth pairs, the MAE is defined as:

$$\text{MAE} = \frac{1}{N} \sum_{i=1}^{N} |\ y_i - \hat{y}_i\ |$$

Where:

- y_i is the ground-truth value for instance i,
- $\hat{y}_i$ is the model-generated prediction for instance i,
- N is the total number of evaluated instances,
- $|c\ \cdot|$ denotes the absolute value operator.

MAE yields a score of 0 when predictions perfectly match the ground truth, with larger values indicating proportionally larger errors. Because MAE is expressed in the same units as the predicted variable, it provides a domain-independent measure of predictive precision

In the context of Text-to-SQL evaluation, MAE is particularly relevant for questions where the expected return is scalar numerical values, such as aggregated statistics or results of quantitative reasoning over database records.

4.3 Datasets

Despite significant research efforts, Text-to-SQL systems have not yet achieved widespread adoption, largely because their effectiveness and ability to generalize depend heavily on the datasets used for training and evaluation resources that are also essential for measuring progress in this field [19].

Models trained on specific benchmarks often perform well in-domain but struggle to generalize to unseen databases or real-world contexts. Performance is influenced by factors such as domain diversity, SQL query distribution, and dataset scale. Models also tend to underperform on query types that are underrepresented during training. Consequently, relying solely on benchmark accuracy can be misleading, as datasets dominated by simple queries may inflate results. A deeper understanding of dataset characteristics and systematic error analysis is therefore essential for building robust, transferable Text-to-SQL systems [19].

4.3.1 Dataset Dimensions

Text-to-SQL datasets can be analyzed along several descriptive dimensions that capture their structural and linguistic diversity [22]:

- Domain Coverage: Datasets may be single-domain restricted to queries over a single database or cross-domain, encompassing multiple databases and application areas.
- Dialogue Turns: Depending on the number of conversational interactions required to formulate an SQL query, datasets can be single-turn or multi-turn.
- SQL Complexity: Query complexity is often assessed through the number of SQL clauses, levels of nesting, and overall structural diversity.

These dimensions shape how well models can generalize beyond their training data and adapt to varied linguistic or structural inputs.

4.3.2 Dataset Types

In practice, Text-to-SQL datasets are typically organized into four main categories based on how they are constructed and used for evaluation [19]:

- Single-domain datasets — Focused on queries within a specific database or domain. While limited in scope, they provide valuable benchmarks for domain-specific modeling.
- Cross-domain datasets — Contain multiple databases spanning diverse domains, supporting the study of model generalization and adaptability.
- Perturbed datasets — Created by modifying existing benchmarks (e.g., through paraphrasing or schema alterations) to assess model robustness under controlled variation.
- Augmented datasets — Automatically expanded using generation techniques to increase linguistic and structural diversity, often supporting large-scale model training.

These categories are not mutually exclusive, a dataset may simultaneously be cross-domain and perturbed, for example. Careful design and selection of datasets along both descriptive dimensions and construction categories are therefore crucial for advancing fair, comprehensive, and transferable Text-to-SQL evaluation.

4.4 Traditional Database Benchmarks

Text-to-SQL models have been extensively evaluated using several established public benchmarks in both academic and industrial settings. In this work, we assess the proposed capability levels using two widely adopted standards: TPC-C, representing transactional workloads, and TPC-H, representing analytical workloads.

4.4.1 TPC-C

TPC-C is an industry-standard benchmark used to evaluate the performance of OLTP systems. It simulates a wholesale supply company with multiple warehouses and sales districts, focusing on a mixture of concurrent transactions typical of enterprise environments, such as new orders, payments, deliveries, order status checks, and inventory monitoring. The benchmark measures system throughput using the metric transactions per minute (tpmC), providing an objective basis for comparing how database systems handle large volumes of transactional operations under realistic conditions. The database schema includes nine interrelated tables and represents the business workflow from order entry through delivery, requiring both read and write operations as well as complex transaction processing [26]

4.4.2 TPC-H

TPC-H is a standard benchmark designed to evaluate the performance of database systems in OLAP environments. It consists of a set of business-oriented, ad hoc queries and concurrent data modifications, simulating analytical operations that organizations perform when analyzing large volumes of transactional data related to sales, purchasing, and logistics. TPC-H assesses a system's ability to execute complex queries on large datasets to answer critical business questions, such as promotions analysis, profit management, and customer satisfaction. The primary metric used by TPC-H is the number of complex queries processed per hour (QphH@Size), enabling objective comparisons of database solutions for real-world analytical scenarios. [27]

5 Evaluation methodology

Both benchmarks define fixed relational schemas and canonical query sets. TPC-C consists of parameterized transactional templates, whereas TPC-H comprises complex analytical queries. Importantly, each benchmark includes detailed NL descriptions of the queries, making them suitable for evaluating progressively more advanced Text-to-SQL competencies.

5.1 TPC-C

5.1.1 Basic Text to SQL

For Level 1, the model receives only the NL description of each TPC-C transaction, without access to the underlying schema. The objective is to evaluate whether the model can infer the SQL structure corresponding to the described transactional logic. The output is an approximate SQL template, which is compared against the canonical TPC-C transaction definitions.

Evaluation focuses primarily on:

- Structural similarity: presence of key SQL operators, join patterns, clause usage (e.g., WHERE, GROUP BY), and general logical flow.
- Schema element recall: whether the generated query references the expected conceptual entities (tables, keys, predicates), even if the names are incorrect.
- Logical coverage: inclusion of essential components required to implement the described transaction semantics.

Because the model is intentionally not provided with schema information, executability is not a central metric at this stage. Instead, the emphasis is on reasoning, structure, and conceptual alignment. Two prompting modes are employed: single-shot generation, and iterative refinement with minimal corrective feedback.

5.1.2 Context Aware Text to SQL

For Level 2, the complete TPC-C database schema (tables, primary and foreign keys, constraints, and relevant metadata) is provided to the model. The expectation is that the model will generate executable and semantically correct SQL transactions.

Evaluation includes:

- Semantic correctness: equivalence between model-generated results and reference outputs under controlled, benchmark-compliant parameter settings.
- Performance analysis: execution latency and error rates under benchmark conditions.

Comparisons are made between: the hand-crafted TPC-C reference SQL, and model-generated SQL derived through zero-shot, few-shot, and reasoning-enhanced prompting strategies.

Evaluation consists of two phases: controlled testing with database snapshots to assess functional correctness in isolation, and full benchmark execution to evaluate transaction behavior under concurrent workloads and measure error rates in realistic conditions.

5.1.3 Direct Data Acess

Level 3 is not applied to TPC-C. TPC-C is fundamentally a write-heavy, stateful, transactional benchmark involving inserts, updates, and deletes. These operations require consistent database state transitions, strict transactional isolation, and high throughput (tpmC), all of which are incompatible with the operational and safety constraints of autonomous LLM-based systems.

LLMs cannot perform state-altering operations such as updates or deletes, as they lack persistent memory and cannot guarantee transactional correctness. Even if limited to the read-only TPC-C components (Stock Level and Order Status transactions), the throughput-centric metric (tpmC) remains inappropriate because model inference latency dominates execution time. The benchmark fundamentally assumes instantaneous SQL execution, whereas LLM-mediated access introduces unpredictable model reasoning time. For these reasons, Level 3 is intentionally omitted for TPC-C.

5.2 TPC-H

5.2.1 Basic Text to SQL

As in the TPC-C Level 1 setting, the model receives only the NL description of each of the 22 TPC-H queries (as provided in the official specification). The benchmark is used here primarily to analyze structural reasoning because without schema access, the model cannot reliably reference table names, join keys, or canonical expressions.

The evaluation therefore focuses on:

- Structural similarity between the generated query and the official TPC-H query.
- Alignment of logical intent, including the presence of required joins, filters, aggregations, and grouping semantics.

Two prompting modes are used: simple single-shot prompting, and iterative refinement with incremental feedback.

5.2.2 Context Aware Text to SQL

For Level 2, the full TPC-H schema is provided. Evaluation follows two complementary benchmarking protocols:

- Baseline Benchmarking: Execution of the standard, hand-crafted TPC-H SQL queries to establish reference performance and correctness metrics.
- Model-Based Benchmarking: Execution of model-generated queries. Quality is assessed via: Execution accuracy: whether the results returned by the model-generated query match the reference outputs; Efficiency analysis: whether the generated query achieves performance comparable to the official version, measured through execution time.

These metrics allow us to determine how accurately the model translates NL business questions into SQL, and whether its SQL is as efficient as the benchmark's optimized reference queries.

Prompting modes include zero-shot, few-shot, and reasoning-oriented prompts.

5.2.3 Direct Data Acess

For Level 3, the model is asked to autonomously retrieve results directly from his training corpus using only the NL description of each TPC-H query. The model must determine the necessary reasoning or other retrieval mechanism without explicitly being asked to produce SQL.

Evaluation includes:

- Result equivalence between retrieved results and the reference TPC-H outputs.
- Quantitative similarity metrics, such as string-distance measures or MAE, depending on data type.

Experiments include both fine-tuned models and RAG-based approaches, exploring how external knowledge conditioning affects retrieval performance.

6 Results

6.1 Basic Text-to-SQL, TPC-H

6.1.1 Simple prompting

To evaluate the automatic conversion of NL into SQL, we conducted initial tests using three text-to-SQL models: Deepseek, Llama, and Gemini. The primary objective of these experiments was to assess how well each model could interpret NL queries and generate executable SQL statements over the benchmark database TPC-H.

- Each query was submitted twice per model, using a schema-free approach where, the models were only given the NL prompt without explicit information about the database schema, table structures, or column names.
- The main evaluation metric was EX, which measures the percentage of generated queries that produce correct results when executed against the database.

The results of the initial experiments are summarized in table 2:

Model	Execution Accuracy (%)
DeepSeek: R1 Distill Llama 70B	4%
Google: Gemini 2.0 Flash Experimental	0%
Meta: Llama 3.3 70B	0%

Table 2: Initial Execution Accuracy results for each text-to-SQL model.

As anticipated, the EX observed across all models was exceedingly low. The primary factor underlying this performance deficit was the incorrect generation of table and column identifiers. In the absence of explicit schema information, the models frequently produced queries referencing tables or columns that do not exist within the benchmark database, rendering the generated SQL statements non-executable. This highlights a fundamental limitation of the schema-free approach, in which the models are required to infer the database structure solely from the NL prompt.

In addition to this critical issue, inherent limitations in the model's training further contributed to the low EX. Specifically, the models may not have been exposed during training to queries of comparable complexity, domain, or structure as those found in the TPC-H benchmark. Consequently, their ability to generalize to this dataset is inherently constrained, resulting in both syntactic and semantic errors.

Overall, these observations underscore the challenges associated with schema-free text-to-SQL generation, particularly for benchmarks involving complex queries with multiple joins, nested subqueries, and aggregations. They also justify the exploration of alternative evaluation metrics, such as SS, which can provide additional insights into the model's capacity to generate logically coherent SQL structures even when execution fails.

For this evaluation, we create a script that compares original and generated SQL queries via their ASTs. The script parses each query, extracts the types of nodes in the AST, and computes similarity as the intersection over union of node types. This approach captures the logical alignment of queries even in the presence of syntactic differences or execution errors. All query pairs are processed, individual similarities are reported, and the results, including the overall average, are saved to a CSV file.

By focusing on SS, we obtain a more forgiving and insightful measure of the model's reasoning capabilities, separating logical understanding from mere syntactic correctness. This is particularly important in schema-free scenarios, where execution failures are common but do not necessarily indicate a failure to grasp the intended query.

For each model, we computed the average SS across all queries. Table 3 summarizes these results:

Model	Average Similarity (%)
DeepSeek: R1 Distill Llama 70B	67%
Google: Gemini 2.0 Flash Experimental	65%
Meta: Llama 3.3 70B	61%

Table 3: Average Structural Similarity for each text-to-SQL model.

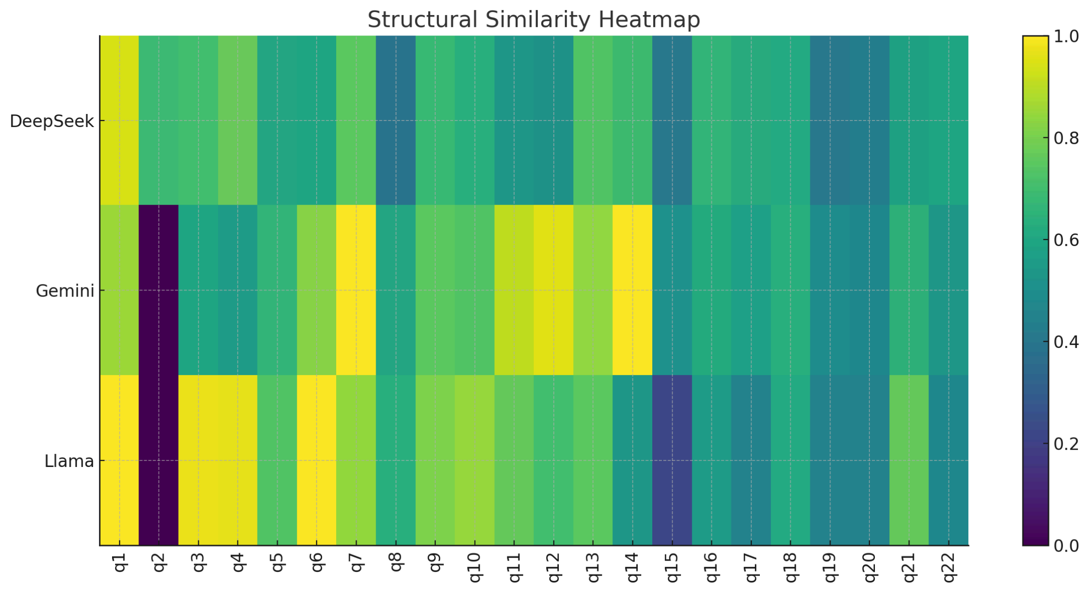

Figure 2: Heatmap of per-query structural similarity across models.

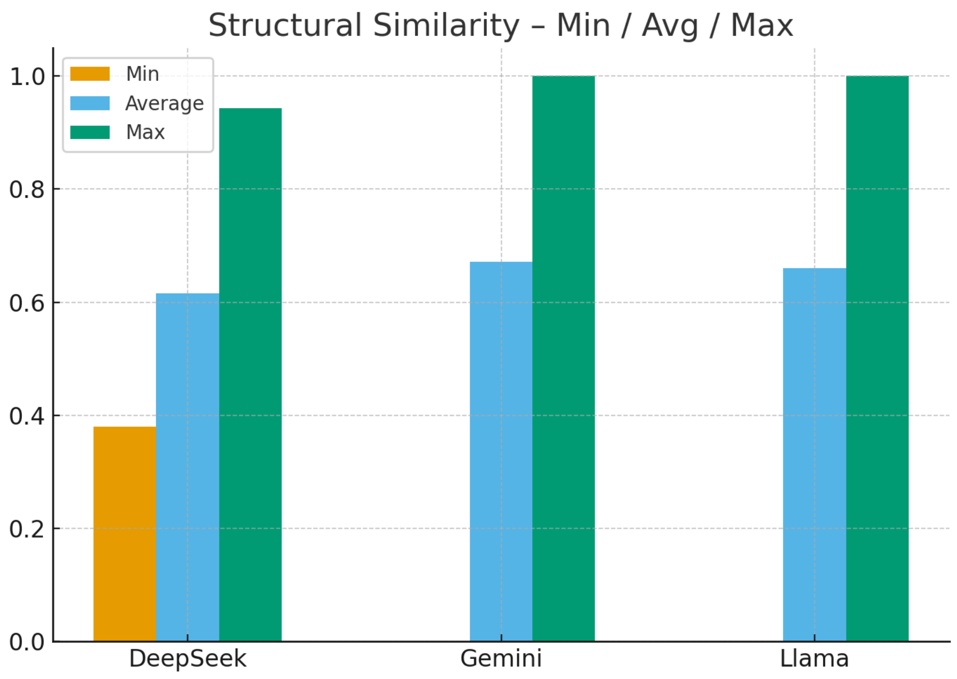

Figure 3: Average structural similarity by model (bar chart).

These results, detailed in full in the appendix, reveal that although EX was very low, all models achieved substantially higher SS scores. This suggests that the models were often able to approximate the general shape of the intended SQL query even when they failed to generate correct table or column names.

In particular, Gemini achieved the highest SS, indicating greater consistency in reproducing the logical layout of the reference queries. Deepseek and Llama followed closely, with only modest differences between them. Overall, this evaluation highlights that while schema-free text-to-SQL remains highly unreliable for producing executable queries, the underlying reasoning patterns of the models are more promising than EX alone would suggest.

6.1.2 Iteractive Refinement Prompts

To further investigate the capabilities of text-to-SQL models, we conducted experiments using iterative refinement of queries with ChatGPT, Gemini, and DeepSeek. In this setup, each model was provided with additional prompts to improve or correct initial SQL outputs, aiming to enhance both SS and EX. This approach allows us to evaluate whether prompt iteration can effectively address limitations observed in single-prompt generation and produce more reliable and accurate SQL queries.

Model	Execution Accuracy (%)
ChatGpt	36%
Gemini	45%
DeepSeek	31%

Table 4: Execution Accuracy results for text-to-SQL models.

The EX results, shown in Table 4, indicate the proportion of text-to-SQL queries that produced correct results when executed against the database. Gemini leads with an EX of 45%. While none of the models achieve high EX, Gemini shows a moderate advantage, suggesting better alignment with actual query execution despite potential structural deviations.

In the following table, the average SS across models is shown:

Model	**Average Similarity (%)**
ChatGpt	65%
Gemini	81%
DeepSeek	82%

Table 5: Average Structural Similarity for each text-to-SQL model.

Both Gemini and DeepSeek demonstrate higher structural fidelity compared to ChatGPT, suggesting these models capture query structure more accurately, even if execution may fail due to syntactic or semantic nuances.

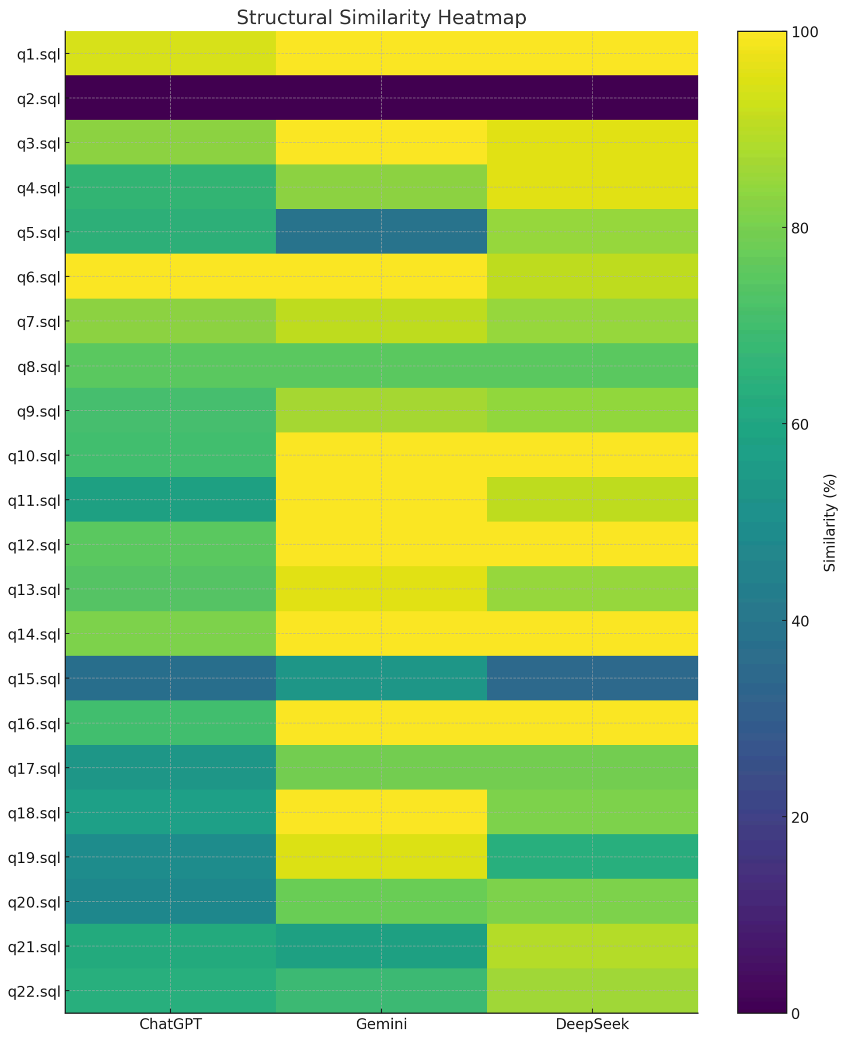

Figure 4: Heatmap of per-query structural similarity across models.

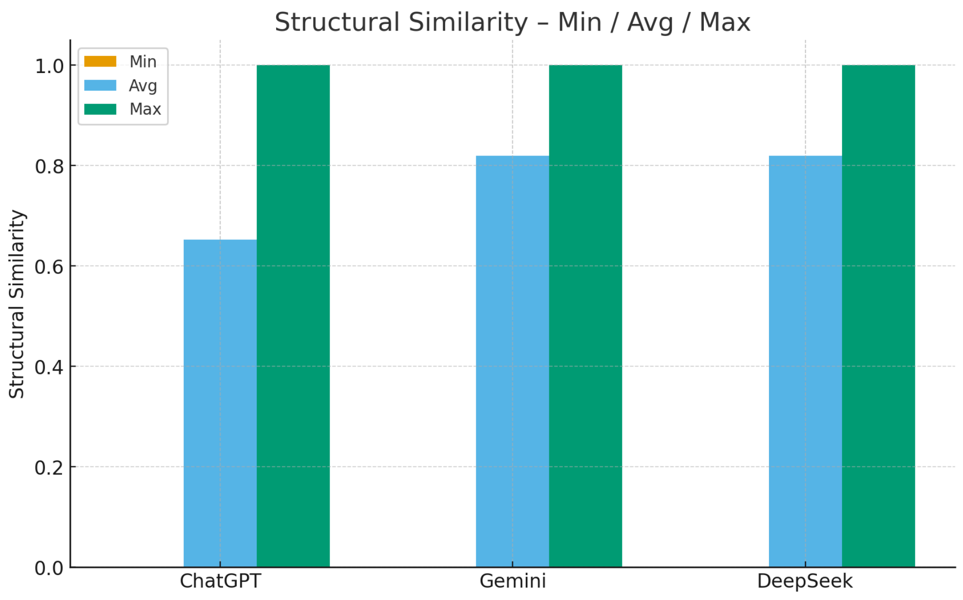

Figure 5: Average structural similarity by model (bar chart).

The per-query heatmap at Figure 4 illustrates variability across individual queries, highlighting that some queries (e.g., query 2) consistently challenge all models, while others (e.g., query, query 6) achieve near-perfect similarity. The bar chart at Figure 5 provides a clear visual summary of each model's average SS, reinforcing that Gemini and DeepSeek generally outperform ChatGPT in preserving query structure.

The results indicate that iterative refinement of queries using additional prompts consistently improves SS compared to generating a query from a single prompt. Models such as ChatGPT, which required multiple prompts for complex queries (e.g., query 15 and query 17), demonstrate that iteration helps better capture the intended SQL structure. However, despite these improvements, schema-free text-to-SQL still exhibits limitations, particularly in terms of EX and in fully aligning semantically with the database schema.

6.2 Basic Text-to-SQL, TPC-C

To assess SS between TPC-C transaction implementations generated by different LLMs, we developed a robust static-analysis tool designed to handle the significantly higher complexity of TPC-C logic compared to TPC-H. The tool focuses on structural, operational, and control-flow characteristics of SQL, rather than lexical similarity.

The analysis pipeline begins with a preprocessing stage in which all SQL scripts are normalized by removing comments and string literals and converting the text to uppercase, ensuring that similarity comparisons depend solely on structural elements rather than formatting or naming variability. After preprocessing, the system performs weighted feature extraction by identifying a predefined set of structural features relevant to SQL transaction logic, such as SELECT, INSERT, UPDATE, DELETE, JOIN, FOR UPDATE, aggregates, subqueries, exception blocks, and concurrency related patterns like SKIP LOCKED. Each feature is assigned a weight proportional to its structural importance, and the tool produces a weighted feature map for each program.

Next, the tool computes operation level similarity by comparing occurrences of core DML operations using a min/max ratio for each operation, averaging the results across features. A similar min/max ratio method is applied to SQL control flow constructs (including IF, CASE, LOOP, FOR, and WHILE) to capture differences in procedural logic between two implementations.

These components are combined into a SS metric using a weighted distance formulation:

Overall Similarity = $0.5 * \text{Feature\_Sim} + 0.3 * \text{Operation\_Sim} + 0.2 * \text{Control\_Sim}$

Finally, the tool reports the overall similarity score along with all individual component metrics. This methodology enables systematic evaluation of how closely LLM-generated TPC-C transactions reflect the structural patterns defined in the official specification. Extended per-transaction results for each model are included in the appendix.

DeepSeek-V3.2 - Overall Similarity

Transaction	**Single Prompt**	**Iteractive Refinement**
Delivery	49.80%	49.80%
New Order	62.36%	61.16%
Order Status	25.83%	37.30%
Payment	56.26%	56.26%
Stock Level	20.34%	54.24%

Table 6: Deepseek Basic Text-to-SQL results

GPT-5 - Overall Similarity

Transaction	**Single Prompt**	**Iteractive Refinement**
Delivery	51.90%	48.74%
New Order	68.90%	69.55%
Order Status	37.12%	37.12%
Payment	56.26%	58.22%
Stock Level	86.36%	86.36%

Table 7: ChatGPT Basic Text-to-SQL results

Gemini 2.5 - Overall Similarity

Transaction	**Single Prompt**	**Iteractive Refinement**
Delivery	54.57%	48.85%
New Order	77.56%	77.74%
Order Status	45.67%	45.67%
Payment	77.54%	77.54%
Stock Level	84.78%	84.78%

Table 8: Gemini Basic Text-to-SQL results

Across the three evaluated models several consistent trends emerge in both single prompt and iterative refinement settings. Iterative refinement has only a limited impact: for nearly all transactions and models, it changes similarity by only a few percentage points, and often not at all. This suggests that most models converge to a stable structural pattern on their first attempt. Because most refinements address syntactic issues caused by missing schema information, they rarely alter the deeper structural elements captured by the similarity metric.

Among the models, Gemini consistently achieves the highest SS, with ChatGPT also performing strongly. Gemini shows the strongest alignment with TPC-C structural patterns, particularly for complex transactions such as New Order and Payment, where similarity scores reach approximately 77–78%. ChatGPT follows with solid performance across most transactions, generally around 52–86%. DeepSeek's scores remain lower overall, ranging from about 20–62%, indicating that it less reliably reproduces the procedural and SQL structural patterns required by TPC-C logic.

The Order Status transaction stands out as the most challenging. All three models exhibit only moderate similarity for this case, typically around 26–46%, reflecting its dependency on nested conditions and selective row retrieval patterns that the models do not consistently capture. In contrast, Stock Level is reproduced more accurately by ChatGPT and Gemini, with similarity scores around 84–86%, showing that these models can reliably reflect its structural logic. DeepSeek continues to struggle with this transaction, achieving only about 20–54%.

Overall, although the initial analysis showed fairly good structural quality in a zero-schema setting, the results indicate that current LLMs remain far from generating fully correct TPC-C transactions without explicit schema guidance.

6.3 Context-Aware SQL Generation, TPC-H

In this phase, we tested several prompting methods to evaluate the ability of different language models to generate SQL queries from NL questions, now using the database schema as context.

We used four different LLM models: Gemini 2.5 Pro, GPT-5.1, Llama 4 Maverick, and DeepSeek V3.1. Of the original 22 TPC-H queries, we used 21, excluding query 20 due to its extremely long execution time in our test environment.

For result evaluation, we chose two main metrics. The first, EX, assesses the validity of the results obtained from executing the generated queries, evaluating them even when the SQL is syntactically different from the original, as long as it produces the same results. We also used the VES, which simultaneously evaluates the correctness and efficiency of the generated queries, taking execution time into account.

The methodology consisted of first running the original TPC-H queries and storing their results as a reference. Then, for each prompting method and model, we executed the queries generated by the LLMs and compared their results with the reference output. During testing, we found that the names and order of the columns returned by the generated queries often differed from the originals. For this reason, the comparison focused on the content of the results rather than solely on the column structure, ensuring a more robust evaluation of the functional correctness of the queries.

All detailed results are available in the appendix.

6.3.1 Zero-shot

To establish an initial baseline, we started with the most basic prompting method: zero-shot. In this approach, we provided the LLMs with only the database schema and the NL question, without any additional context or examples.

Model	Execution Accuracy	Valid Efficiency Score
Gemini	95.24%	77.86%
GPT	71.43%	53.12%
DeepSeek	52.38%	40.57%
Llama	52.38%	35.06%

Table 9: TPC-H results using Zero-shot prompting.

The results obtained show that, by providing the database schema context, even with this minimalist approach we can already achieve considerable performance, although with significant variation across models. We can observe notable differences in performance across the tested models. For example, Gemini achieved 95.24% EX, while Llama and DeepSeek both reached 52.38%, and GPT obtained 71.43%. This variation indicates that model selection has a significant impact on zero-shot performance, with different models exhibiting distinct capabilities for this task.

6.3.2 Zero-shot with Reasoning

To complement the zero-shot method, we introduced a variant with explicit reasoning by adding the instruction "Let's think step by step" to the prompt. The goal was to encourage the LLMs to break down the problem and reason before producing the final query.

Model	Execution Accuracy (EX)	Valid Efficiency Score
Gemini	95.24%	79.66%
GPT	61.90%	50.72%
DeepSeek	52.38%	42.56%
Llama	42.86%	38.54%

Table 10: TPC-H results using Zero-shot with Reasoning prompting.

Contrary to expectations, the results did not show significant improvements. In most cases, performance remained the same or even slightly worsened. This lack of improvement may be due to several factors. It is possible that the explicit reasoning instruction confuses some models or introduces unnecessary verbosity. Additionally, more advanced models may already activate internal reasoning mechanisms given the complexity of the TPC-H queries, even without explicit instructions. Thus, adding the "step by step" prompt may be redundant or even counterproductive in certain cases.

6.3.3 Few-shot

To enrich the prompting context, we implemented the few-shot approach, providing examples of question–SQL pairs along with the schema. The selection of examples was performed using the Maximal Marginal Relevance (MMR) algorithm, which balances similarity and diversity.

The MMR algorithm uses the formula:

$$\text{MMR\_score} = \lambda \times \text{relevance} - (1 - \lambda) \times \text{redundancy}$$

where:

- Relevance: similarity between the candidate example and the current query
- Redundancy: maximum similarity of the candidate with examples already selected
- λ (lambda): a weight that controls the balance between similarity and diversity

The algorithm works iteratively, selecting at each step the example with the highest MMR_score. Instead of simply choosing the examples most similar to the current query, MMR balances two objectives: maintaining relevance by selecting examples related to the query, and avoiding redundancy by preventing the selection of examples that are too similar to each other. Depending on the value of λ, the behavior changes significantly. With λ = 0.6, which emphasizes similarity, the algorithm prioritizes examples that are closer to the current query. This can be advantageous when the query requires specific SQL patterns. With λ = 0.4, which emphasizes diversity, the algorithm favors a wider variety of examples, exposing the model to different SQL structures and approaches, which may improve generalization. By avoiding the selection of multiple similar examples that offer little additional information, this approach maximizes the informational value of the set provided to the model.

The method was tested with different configurations: 2 and 5 examples, combined with λ = 0.4 (greater diversity) and λ = 0.6 (greater similarity).

Model	**2Ex, λ=0.4**		**2Ex, λ=0.6**		**5Ex, λ=0.4**		**5Ex, λ=0.6**	
	EX	**VES**	**EX**	**VES**	**EX**	**VES**	**EX**	**VES**
Gemini	95.24%	84.76%	100.00%	79.00%	90.48%	73.14%	85.71%	67.49%
GPT	57.14%	49.09%	61.90%	51.15%	66.67%	48.44%	66.67%	53.18%
DeepSeek	47.62%	41.60%	52.38%	38.21%	61.90%	51.35%	42.86%	35.03%
Llama	52.38%	51.53%	52.38%	47.31%	57.14%	49.44%	47.62%	39.20%

Table 11: TPC-H results using Few-shot prompting.

The results show substantial improvements compared to zero-shot, although with important variations depending on the model and the chosen parameters. With Gemini, we were able to obtain very remarkable results, achieving 100% EX with 2 examples and λ=0.6.

Overall, all models presented at least one configuration that outperformed zero-shot, confirming the effectiveness of the few-shot approach. However, these results show that the number of examples and the value of λ significantly influence performance, but in different ways for each model. Certain configurations can even worsen the results compared to zero-shot, suggesting that poorly selected or excessive examples may confuse the models or introduce noise into the context. For instance, there were cases where increasing the number of examples from 2 to 5, or changing the value of λ, resulted in performance degradation, while in other cases more examples led to better results. This indicates that there is no universally optimal configuration, and it is necessary to tune the parameters to the model and the specific characteristics of the use case.

6.3.4 Few-shot with reasoning

Finally, we tested an approach that combines few-shot with reasoning, specifically CoT, including not only examples of NL questions and their corresponding SQL queries, but also the reasoning process that links each question to its SQL query. The examples were selected using the MMR algorithm, as in simple few-shot.

The CoT generation was inspired by the ACT-SQL method, using automatic schema linking. This process begins by automatically extracting the tables, columns, and values that appear in the SQL query, ensuring that they correspond to actual elements in the database. Simultaneously, relevant key phrases are identified in the NL description. Then, a sentence-transformers model is used to compute the semantic similarity between the description phrases and the schema elements (tables and columns). When the similarity is sufficiently high, a link is established between the phrase and the corresponding element.

The final result is a structured text that explains, step by step, which tables and columns to use for each part of the description, enabling the automatic generation of reasoning examples that explicitly show the connections between the question and the schema.

CHAIN OF THOUGHT:

- According to "given part type", columns [lineitem.l_partkey, part.p_partkey, part.p_type] may be used.
- According to "l_discount from the", columns [lineitem.l_discount] may be used.
- According to "nation the", columns [nation.n_name, nation.n_nationkey] may be used.
- According to "to n_regionkey", columns [customer.c_nationkey, nation.n_regionkey] may be used.
- Values [1995-01-01, 1996-12-31, AFRICA, ECONOMY PLATED BRASS, KENYA] may be used.

Figure 6: Chain of Thought reasoning example

Model	2Ex, λ=0.4		2Ex, λ=0.6		5Ex, λ=0.4		5Ex, λ=0.6	
	EX	VES	EX	VES	EX	VES	EX	VES
Gemini	100%	82.40%	90.48%	95.29%	100%	80.56%	95.24%	76.06%
GPT	57.14%	46.79%	66.67%	55.52%	61.90%	42.75%	71.43%	54.99%
DeepSeek	80.95%	65.60%	76.19%	59.76%	52.38%	49.47%	76.19%	60.97%
Llama	57.14%	51.62%	61.90%	67.78%	57.14%	43.04%	57.14%	45.88%

Table 12: TPC-H results using Few-shot with Reasoning prompting.

The results of this approach reveal trends similar to those observed in few-shot without reasoning, where parameter choices continue to have a significant impact on performance. Gemini remained consistent, achieving 100% EX in two configurations (2 examples with $\lambda = 0.4$ and 5 examples with $\lambda = 0.4$). Notably, it also achieved 95.29% VES with 2 examples and $\lambda = 0.6$, the highest value recorded across all experiments, indicating that this configuration produced queries that were both correct and efficient.

Overall, this was the approach that allowed models to achieve the best results in this phase. DeepSeek, for example, showed a particularly remarkable improvement with the introduction of CoT. While it achieved only 52.38% EX in zero-shot, with this approach it reached 80.95% (2 examples, λ = 0.4).

However, as observed in few-shot without reasoning, parameter choices still strongly influence performance and can, in some cases, harm results. GPT illustrates this variability well, with EX ranging from 57.14% (2 examples, λ = 0.4) to 71.43% (5 examples, λ = 0.6).

6.3.5 Comparative Analysis of Results

Conducting an overall analysis of all results obtained with the different prompting methods tested, we can draw important conclusions about the effectiveness of each approach for the text-to-SQL task.

The graph below presents the EX for each model and method, with the best accuracy achieved among the different configurations shown for the few-shot approaches.

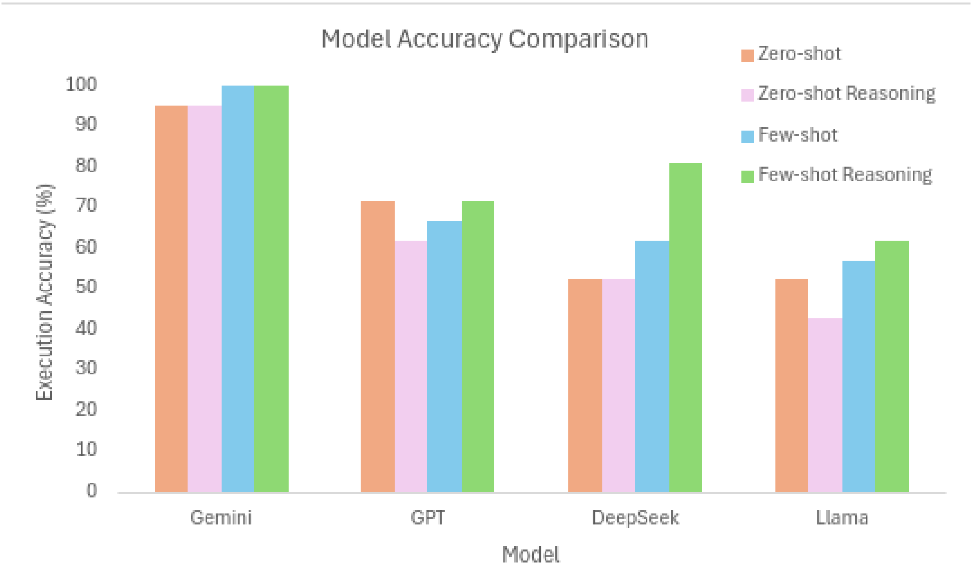

Figure 7: Best Execution Accuracy achieved by each model across different prompting methods.

Examining the graph, we can see that the best results were obtained with few-shot with CoT, which demonstrates that the use of examples in context, combined with explicit schema linking reasoning, provides significant gains in the models' ability to generate correct SQL queries. However, as previously mentioned in the individual analysis of each method, the correct choice of parameters (number of examples and λ value) is crucial. Inadequate configurations can result in performance inferior even to basic zero-shot, indicating that improper use of examples can introduce noise and confuse the models rather than assist them. This sensitivity reinforces that there is no universally optimal configuration, making it necessary to adjust parameters to the characteristics of each model.

Zero-shot with reasoning consistently showed results inferior to or similar to basic zero-shot, not delivering the expected benefits. This result suggests that simply adding a step-by-step reasoning instruction, without concrete examples, is insufficient or even counterproductive for this task.

Regarding the VES values obtained throughout the experiments, we observe that queries generated by the models tend to have lower execution performance than the original TPC-H queries.

In summary, the results demonstrate that LLMs can already generate quality SQL queries from NL questions, especially when provided with adequate context. With the correct approach and parameterization, it was possible to achieve EX levels of up to 100%.

6.4 Context-Aware SQL Generation, TPC-C

Following the methodology applied to TPC-H, we extended our evaluation to the TPC-C benchmark, testing the same prompting techniques and the four LLMs with transactional workloads. We focused on the five standard TPC-C transactions: New Order, Payment, Order Status, Delivery, and Stock Level. Unlike TPC-H, where we tested multiple configurations for few-shot prompting, here we fixed the number of examples at 2 and λ at 0.6. This decision was motivated by the limited number of transactions available, with only 4 possible examples (5 transactions minus 1 for the target), meaning that extensive exploration of parameters would have been less meaningful.

The evaluation methodology for TPC-C differed from that used for TPC-H because now we are working with transactions. Unlike TPC-H queries that simply retrieve data, TPC-C transactions (most of them) modify the database state, requiring a more comprehensive evaluation approach that considers not only the output but also the resulting database state.

Our approach consisted of two phases:

In the first phase, we used database snapshots to ensure reproducible testing conditions. The process was as follows:

1. Capture the initial state of the database using snapshots;
2. Execute each original TPC-C transaction and record both the output and the resulting database state;
3. Restore the database to its initial state;
4. For each prompting method and model, execute the transaction generated by the LLM and record its output and resulting database state;
5. Compare the results of the LLM-generated transaction with the original reference, evaluating both output correctness and database state consistency.

Each transaction was executed 8 times to account for variability, with the results and database states recorded for each execution. This controlled testing environment allowed us to systematically assess the functional correctness of the generated transactions under predictable conditions. However, it is important to note that this phase, although rigorous, represented a limited set of tests that did not cover all possible scenarios.

In the second phase, we executed the full TPC-C benchmark to evaluate how the transactions generated by the LLMs performed under concurrency and a broader range of input scenarios. In this phase, we recorded performance metrics including latency, number of transactions executed, and errors encountered. For reference, the original transactions achieved an average latency of 0.005s across all transaction types during the benchmark execution. For each model and prompting method, we calculated the average latency and the error rate (averaged across all five transactions to ensure equal weighting).

All detailed results are available in the appendix.

6.4.1 Zero-shot

The zero-shot results revealed a moderate performance, with significant variation across models. GPT was the model that stood out the most, achieving 90% EX with an extremely low error rate of only 2.1% during benchmark execution. This suggests that GPT was able to generate functionally correct transactions that remained stable under concurrent workloads.

In contrast, Gemini, DeepSeek, and Llama showed considerably lower accuracies (60%, 40%, and 20% respectively) and much higher error rates during benchmark execution (49.6%, 73.5%, and 72.0%).

Model	**Overall Execution Accuracy**	**TPC-C Benchmark**	
		Avg Latency (s)	**Error Rate (%)**
Gemini	60.0%	0.0042	49.6%
GPT	90.0%	0.0052	2.1%
DeepSeek	40.0%	0.0048	73.5%
Llama	20.0%	0.0058	72.0%

Table 13: TPC-C results using Zero-shot prompting.

6.4.2 Zero-shot with Reasoning

Adding explicit reasoning instructions produced mixed results. Overall EX improved for most models, with Gemini rising from 60% to 77.5% and GPT reaching 92.5%. DeepSeek also showed improvement while Llama remained stagnant.

However, looking at the benchmark error rates, these did not follow the gains in accuracy. Gemini's error rate increased from 49.6% to 66.5%, and GPT's rose from 2.1% to 15.0%. The only exception was DeepSeek, which improved both its accuracy and significantly reduced its error rate.

Model	**Overall Execution Accuracy**	**TPC-C Benchmark**	
		Avg Latency (s)	**Error Rate (%)**
Gemini	77.5%	0.0054	66.5%
GPT	92.5%	0.0048	15.0%
DeepSeek	60.0%	0.0066	51.0%
Llama	20.0%	0.0046	74.0%

Table 14: TPC-C results using Zero-shot with Reasoning prompting.

6.4.3 Few-shot

The few-shot approach brought substantial improvements in EX across all models. Gemini reached 95%, GPT maintained its result, DeepSeek improved significantly, and even Llama showed progress. This confirms that providing concrete examples of transaction implementations helps the models understand the required structure and logic.

Compared to the basic zero-shot setting, the benchmark error rates showed interesting patterns. Gemini's error rate notably improved from 49.6% to 41.5%, demonstrating that the examples helped generate more stable implementations. However, GPT's error rate increased from 2.1% to 17.7%, and DeepSeek's decreased slightly from 73.5% to 60.1%. Llama continued to struggle, with high error rates of 80.0%.

Model	**Overall Execution Accuracy**	**TPC-C Benchmark**	
		Avg Latency (s)	**Error Rate (%)**
Gemini	95.0%	0.0050	41.5%
GPT	92.5%	0.0056	17.7%
DeepSeek	75.0%	0.0046	60.1%
Llama	30.0%	0.0036	80.0%

Table 15: TPC-C results using Few-shot prompting.

6.4.4 Few-shot with Reasoning

Combining few-shot examples with CoT reasoning produced the best overall results. Both Gemini and DeepSeek achieved 97.5% EX, representing an almost perfect performance in the controlled test phase. GPT declined slightly to 85%, while Llama improved to 40%.

More importantly, this approach also delivered the lowest benchmark error rates for the top-performing models. Gemini's error rate dropped to 20.4%, the lowest across all its configurations, and DeepSeek reached 41.7%. GPT maintained a low error rate. These results suggest that combining concrete examples with explicit reasoning about schema relationships helps models generate transactions that are both functionally correct and stable under concurrent execution.

Model	**Overall Execution Accuracy**	**TPC-C Benchmark**	
		Avg Latency (s)	**Error Rate (%)**
Gemini	97.5%	0.0062	20.4%
GPT	85.0%	0.0082	16.3%
DeepSeek	97.5%	0.0060	41.7%
Llama	40.0%	0.0066	60.0%

Table 16: TPC-C results using Few-shot with Reasoning prompting.

6.4.5 Results Analysis and Discussion

Analyzing the results obtained throughout this process, we can draw several important conclusions.

Across all models and methods, we observed consistent patterns in transaction difficulty. Read-only transactions, Order Status and Stock Level, generally produced good results, with most models achieving correct implementations that executed without errors in the benchmark.

In contrast, the New Order and Payment transactions proved to be particularly challenging. These transactions modify multiple tables and require careful management of database consistency constraints. Although many configurations achieved good results in the controlled testing environment (Phase 1), they frequently encountered errors when executed under concurrent load in the benchmark (Phase 2). This discrepancy highlights the critical challenge of generating transactions that are not only functionally correct in isolation but also robust under concurrent execution. Notably, the New Order transaction only achieved both high accuracy and zero benchmark errors with GPT in the zero-shot and zero-shot with reasoning configurations. Similarly, the Payment transaction only executed without errors in the benchmark with GPT in zero-shot with reasoning, few-shot, and few-shot with reasoning. This suggests that GPT may have an advantage in understanding and implementing the subtle concurrency control requirements of these complex transactions.

Few-shot prompting methods generally improved EX across models, with few-shot with reasoning delivering the best overall results. Both Gemini and DeepSeek achieved 97.5% accuracy with this approach, demonstrating that examples combined with explicit schema reasoning provide substantial benefits for transaction generation. However, an interesting counterexample emerged with the New Order transaction, which only achieved error-free benchmark execution with zero-shot methods (specifically with GPT). This suggests that for certain complex transactions, the additional context of examples and reasoning may actually introduce patterns that interfere with proper concurrency handling. It is possible that the simpler zero-shot approach allowed the model to generate more straightforward implementations that handled concurrent access more cleanly.

There is also a disconnect between controlled test accuracy and benchmark error rates. Many transactions that achieved excellent accuracy in the controlled environment frequently exhibited high error rates under concurrent execution. This reveals a fundamental challenge in using LLMs for transaction generation: ensuring that generated code properly handles concurrent access requires considerations beyond functional correctness. The error rates observed during benchmark execution, ranging from a minimum of 2.1% for GPT in zero-shot to a maximum of 80% for Llama in few-shot, reflect issues such as race conditions, inadequate locking, or incorrect transaction isolation handling. These are precisely the types of subtle bugs that are difficult to detect without concurrent testing.

Regarding execution efficiency, none of the generated transactions matched the performance of the original implementations. The average latencies of generated transactions varied significantly, indicating room for optimization not only in functional correctness but also in the efficiency of generated code.

In summary, the TPC-C evaluation demonstrates that LLMs can generate transactions with reasonable accuracy, particularly when provided with examples and reasoning guidance. However, significant challenges remain in ensuring that generated transactions properly handle concurrent execution. The discrepancy between performance in controlled environments and under concurrent load highlights the need for more comprehensive testing methodologies and, potentially, prompting techniques that explicitly emphasize concurrency considerations.

6.5 Direct Data Access, TPC-H

This phase of experimentation used a small, controlled subset of the TPC-H dataset. A maximum of 10,000 rows per table were included to ensure tractable execution, lightweight retrieval operations and lower training time. This subset served to feed the database used to evaluate EX and as the corpus for RAG or fine-tuning models, depending on the experiment.

For each query:

1. A ground-truth result was obtained by executing the official benchmark SQL query against the dataset, using the same seed employed in Level 2.

2. Models were prompted using standardized instructions that enforced the use of correct SQL formatting, consistent aliasing conventions, and explicit restrictions on assumptions about nonexistent tables or fields

6.5.1 Fine-tuning

Across all Direct Data Access experiments, fine-tuning consistently delivered the lowest performance among the evaluated solutions. Despite its theoretical appeal as a way to embed database content directly into a model's parameters, our empirical findings reveal fundamental limitations that make this paradigm unsuitable for QA over relational data.

Contrary to expectations, increasing model size within the range of small open-source LLMs did not significantly improve performance. Larger models did not memorize substantially more database values, nor did they maintain better reasoning abilities after fine-tuning. This suggests that, for this task, parameter count is not the primary bottleneck; instead, the interaction between memorization and general reasoning capabilities is the limiting factor.

The most decisive factor affecting performance was the number of training epochs. Training for more than one epoch resulted in models that lost the ability to properly interpret questions. They began producing incoherent or hallucinated answers, indicating forgetting of general language skills. Training for a single epoch preserved the model's ability to understand questions, but was insufficient for memorizing the database content reliably. As a result, the model could neither recall specific values nor generalize to unseen phrasings of the same query. So, increasing epochs improves memorization but destroys reasoning, yet reducing epochs preserves reasoning but prevents memorization. No configuration provided a usable balance.

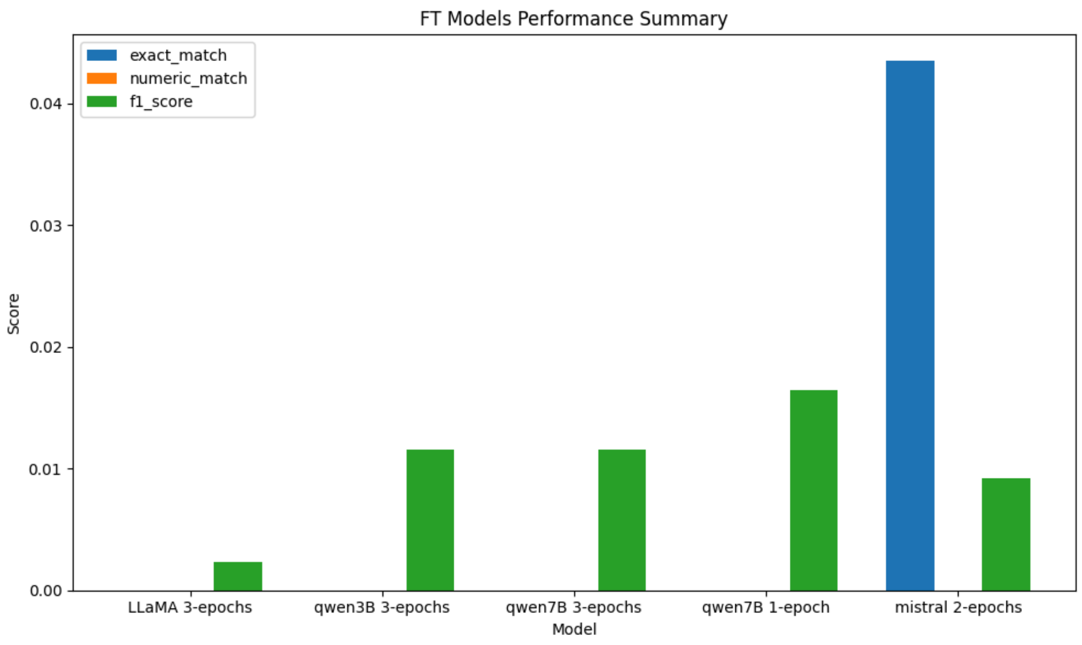

Figure 8: Average Exact Match, Numeric Match and F1 Score per FT Model.

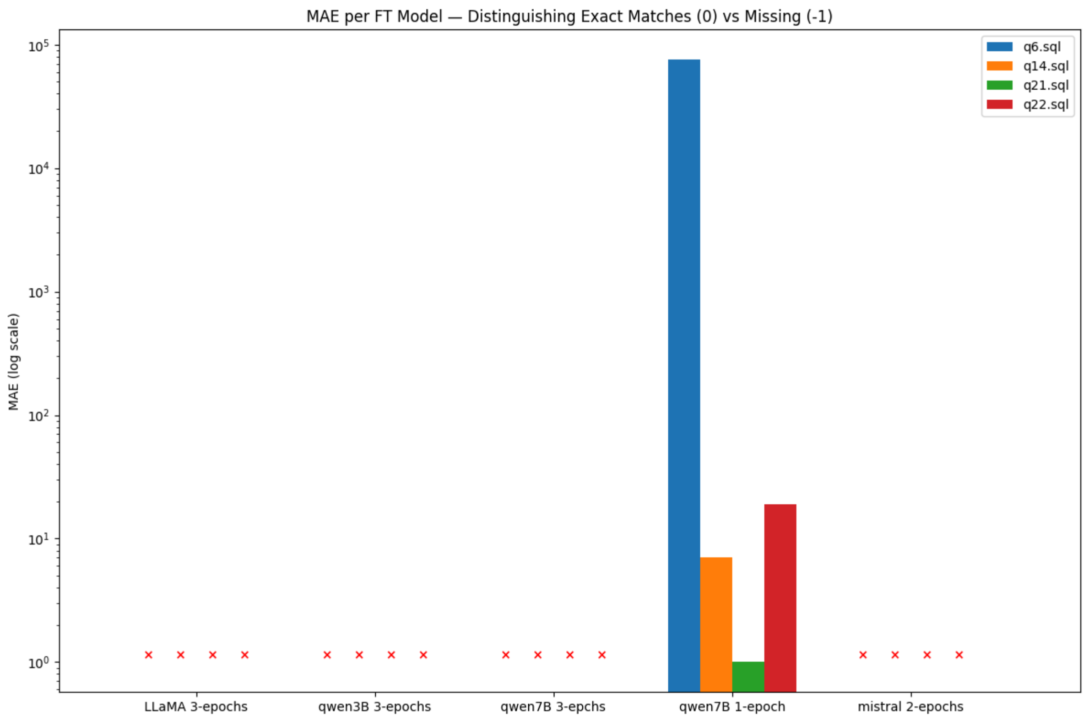

Figure 9: Average MAE per FT Model.

Beyond the numerical results, some structural reasons explain why fine-tuning is inherently ill-suited for QA. Models cannot encode database-scale information reliably in parameters. Relational datasets contain thousands to millions of unique values. LLMs have no mechanism to guarantee exact memorization of structured data, making answers brittle, incomplete, or incorrect. Also, fine-tuned knowledge becomes stale immediately. Any change to the database requires a full retraining cycle. This is infeasible for environments where data is constantly updated. In addition, there is no mechanism for multi-row or multi-table reasoning. Even when individual values are memorized, the model cannot dynamically combine or aggregate them. Tasks such as joins or aggregations become impossible to emulate reliably. Finally, these models lack determinism and verifiability. Unlike SQL, which is interpretable and auditable, a fine-tuned model provides no transparency into where an answer came from or whether it is complete.

6.5.2 RAG

The RAG paradigm yielded the ostrongest performance among the direct data access approaches. However, despite being the most effective configuration, the overall results remained poor, with most generated answers exhibiting semantic inconsistencies or incorrect numerical reasoning, as reflected by both the MAE distributions, shown in Figure 11, and the low F1 scores, shown in Figure 10.

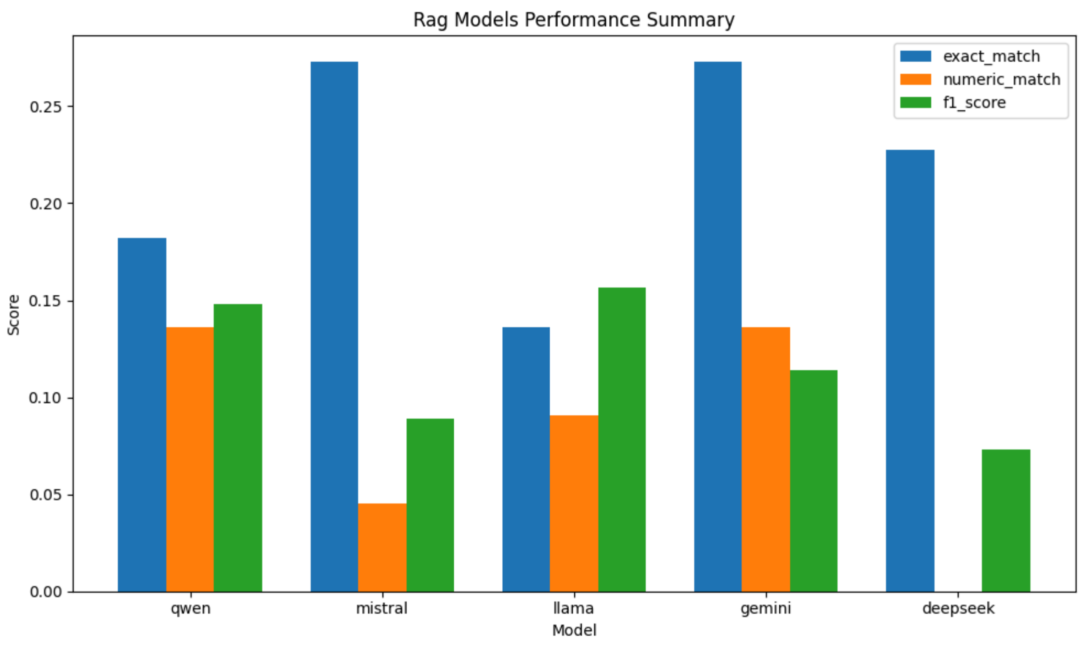

Figure 10: Average Exact Match, Numeric Match and F1 Score per RAG Model.

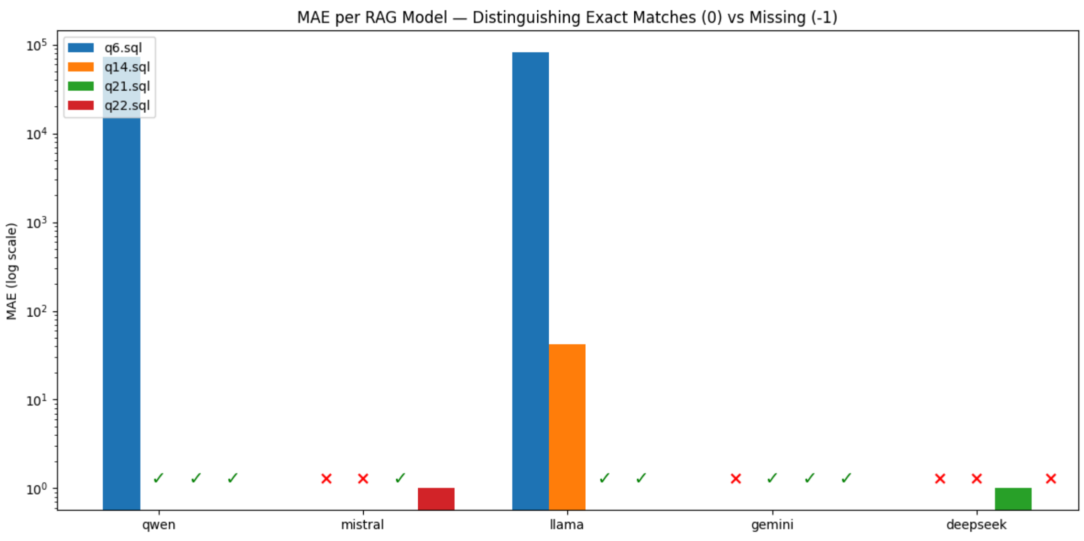

Figure 11: Average MAE per RAG Model.

A central limitation across models was the restricted context window, which prevented them from receiving the full set of relevant table rows or sufficient aggregated evidence to answer TPC-H queries correctly. Even when relevant data was retrieved, models struggled to integrate large volumes of numerical information, highlighting that current LLMs are not yet equipped to perform large scale data aggregation directly from RAG context.

Performance was noticeably better for two specific categories of questions:

1. Simple lookup style questions (not part of TPC-H but added for comparison), where models only needed to extract a single value or confirm the presence/absence of a record.
2. Unanswerable or missing-answer cases, where the model correctly inferred that no valid row satisfied the conditions. This inflated metrics like exact-match for some models.

These nuances also reveal why some of the metrics summarized in the Level 3 results require careful interpretation. For example, certain models such as Mistral and Gemini, achieved relatively high exact match scores, but many of these “correct answers” corresponded to declining to answer or detecting that the dataset lacked a matching record. Conversely, some models appear to have extremely large MAE values, such as Qwen, but this is largely driven by a single outlier, where the model drastically underestimated a numeric quantity due to insufficient retrieved context.

Overall, the RAG experiments highlight two fundamental limitations of existing LLMs:

- RAG does not compensate for the inability of LLMs to aggregate or summarize large datasets into accurate numeric outputs. LLMs can read numbers but cannot reliably compute with them on the scale required by TPC-H analytical queries.
- Retrieval helps grounding but does not guarantee reasoning. Even when the correct rows were retrieved, models often produced plausible-sounding but numerically incorrect answers.

Despite these shortcomings, RAG remains the most promising direction for improving text-to-data workflows. The experiments demonstrate that retrieval can meaningfully assist in cases requiring factual grounding and that models can detect contradictions or missing data surprisingly well. However, for analytical workloads such as TPC-H, RAG alone is insufficient without an external computation layer capable of performing aggregations, joins, and arithmetic reliably.

7 Conclusion

In this survey, we examined the relevance of LLMs on Text-to-SQL operations and interactions with relational data systems across three escalating levels of complexity: Basic Text-to-SQL, Context-Aware SQL Generation, and Direct Data Access. By evaluating modern LLMs under these distinct paradigms and using TPC-H and TPC-C as performance baselines, we were able to characterize both the current capabilities of LLM-based data querying and their fundamental limitations.

Across all experiments, we concluded that while LLMs can produce SQL-like structures and approximations of query intent, their reliability varies dramatically depending on the form and richness of context provided.

- Level 1 results revealed that schema-free Text-to-SQL is inherently unreliable: execution accuracy was extremely low, although structural similarity showed that models can capture the general shape of a query even when they fail to reference correct tables or fields.
- Level 2 experiments demonstrated substantial improvements. When given schemas, examples, or reasoning scaffolds, models (especially Gemini and GPT) achieved high execution accuracy and efficiency, confirming that context is the key determinant of Text-to-SQL success.
- Level 3 experiments highlighted the sharp limitations of direct data access approaches. Fine-tuning proved ineffective for retaining both reasoning skills and database content, while RAG, although the best-performing method in this category, remained incapable of reliably aggregating or computing over large datasets.

We acknowledge several practical limitations of our study. First, the outputs generated by LLMs inherently exhibit stochastic variability: even when prompted with identical inputs, models may produce different SQL queries, reasoning steps, or interpretations. This unpredictability complicates strict reproducibility and should be considered when interpreting the results. Second, although TPC-H and TPC-C served as valuable analytical and transactional benchmarks, our implementation did not use their full industrial-scale configurations. Due to constraints on time, computational resources, and the wide range of experiments required, we worked with reduced or minimal versions of these benchmarks. Despite these limitations, we believe our framework significantly mitigates these issues. Consequently, our findings should be viewed as robust trend-level insights rather than definitive, full-spec benchmark claims.

With these considerations in mind, we successfully met all objectives outlined in the introduction. Throughout the study, we performed a comprehensive evaluation of three tiers of Text-to-SQL strategies using analytical (TPC-H) and transactional (TPC-C) workloads, while experimenting with a wide range of prompting techniques, including zero-shot, few-shot, and reasoning-enhanced approaches. To support deeper analysis, we introduced custom metrics and tooling that revealed nuances beyond what execution accuracy alone could capture. By examining multiple LLMs under varied conditions, we identified consistent performance trends and characteristic failure modes, grounding our conclusions in both empirical results and established research.

This work was significant because it provides a unified, end-to-end view of how LLMs behave across the entire spectrum of data interaction: from generating SQL blindly, to reasoning with schemas, to attempting direct semantic access to relational data. It shows that while LLMs excel at interpretation, explanation, and structured query generation, they cannot replace database engines for computation or reasoning over large datasets. Instead, they are most effective when integrated into hybrid architectures, where they function as planners, translators, and assistants, while deterministic systems handle execution, aggregation, and verification.

Ultimately, the findings reinforce a central conclusion: the future of AI-driven data access lies not in replacing relational systems with LLMs, but in tightly coupling their complementary strengths. LLMs provide natural language understanding and flexible reasoning, while database engines ensure correctness, scalability, and transparency. This synergy represents the most promising direction for building reliable, intelligent, and user-friendly data interfaces.

8 Appendix

8.1 TPC-C

8.1.1 Level 1: Simple prompting

DeepSeek

Transaction	Feature Similarity	Operation Similarity	Control Flow Similarity	Overall Similarity
DELIVERY	37.93%	80.56%	33.33%	49.80%
NEW_ORDER	59.33%	74.81%	51.25%	62.36%
ORDER_STATUS	26.67%	25.00%	25.00%	25.83%
PAYMENT	38.86%	84.72%	60.00%	56.85%
STOCK_LEVEL	20.69%	33.33%	0.00%	20.34%

Chat GPT

Transaction	Feature Similarity	Operation Similarity	Control Flow Similarity	Overall Similarity
DELIVERY	50.46%	65.00%	35.83%	51.90%
NEW_ORDER	64.26%	74.60%	71.94%	68.90%
ORDER_STATUS	25.00%	80.00%	3.12%	37.12%
PAYMENT	49.53%	79.05%	38.89%	56.26%
STOCK_LEVEL	72.73%	100.00%	100.00%	86.36%

Gemini

Transaction	Feature Similarity	Operation Similarity	Control Flow Similarity	Overall Similarity
DELIVERY	42.48%	83.33%	41.67%	54.57%
NEW_ORDER	73.71%	93.33%	63.54%	77.56%
ORDER_STATUS	40.00%	80.00%	8.33%	45.67%
PAYMENT	72.94%	86.90%	75.00%	77.54%
STOCK_LEVEL	69.57%	100.00%	0.00%	84.78%

8.1.2 Level 1: Iteractive Refinement Prompts

DeepSeek

Transaction	Feature Similarity	Operation Similarity	Control Flow Similarity	Overall Similarity
DELIVERY	37.93%	80.56%	33.33%	49.80%
NEW_ORDER	57.82%	73.33%	51.25%	61.16%
ORDER_STATUS	32.61%	66.67%	5.00%	37.30%
PAYMENT	49.53%	79.05%	38.89%	56.26%
STOCK_LEVEL	48.48%	100.00%	0.00%	54.24%

Chat GPT

Transaction	Feature Similarity	Operation Similarity	Control Flow Similarity	Overall Similarity
DELIVERY	46.81%	65.00%	29.17%	48.74%
NEW_ORDER	64.43%	82.22%	63.33%	69.55%
ORDER_STATUS	25.00%	80.00%	3.12%	37.12%
PAYMENT	47.10%	93.33%	33.33%	58.22%
STOCK LEVEL	72.73%	100.00%	100.00%	86.36%

Gemini

Transaction	Feature Similarity	Operation Similarity	Control Flow Similarity	Overall Similarity
DELIVERY	36.04%	80.56%	33.33%	48.85%
NEW_ORDER	74.06%	93.33%	63.54%	77.74%
ORDER_STATUS	40.00%	80.00%	8.33%	45.67%
PAYMENT	72.94%	86.90%	75.00%	77.54%
STOCK_LEVEL	69.57%	100.00%	100.00%	84.78%

8.1.3 Level 2: Context-Aware SQL Generation

8.1.3.1 TPC-C benchmark results for the original transactions.

Original Transactions

Transaction	TPC-C Benchmark					
	Latency				Count	Errors
	90th %	95th %	99th %	Avg		
NEW_ORDER	0.008s	0.008s	0.009s	0.005s	292	0
PAYMENT	0.007s	0.007s	0.008s	0.004s	236	0
ORDER_STATUS	0.009s	0.009s	0.009s	0.006s	22	0
STOCK_LEVEL	0.008s	0.009s	0.009s	0.004s	30	0
DELIVERY	0.010s	0.010s	0.011s	0.006s	26	0

8.1.3.2 Model: Gemini 2.5 Pro

Gemini Zero-Shot

Transaction	**Execution Accuracy (%)**	**TPC-C Benchmark**					
		Latency					
		90th %	**95th %**	**99th %**	**Avg**	**Count**	**Errors**
NEW_ORDER	0%	0.002s	0.002s	0.005s	0.001s	267	267
PAYMENT	0%	0.004s	0.004s	0.004s	0.003s	220	220
ORDER_STATUS	100%	0.008s	0.008s	0.009s	0.005s	25	12
STOCK_LEVEL	100%	0.004s	0.004s	0.004s	0.004s	12	0
DELIVERY	100%	0.010s	0.010s	0.010s	0.008s	22	0

Gemini Zero-Shot with Reasoning

Transaction	**Execution Accuracy (%)**	**TPC-C Benchmark**					
		Latency					
		90th %	**95th %**	**99th %**	**Avg**	**Count**	**Errors**
NEW_ORDER	87.5%	0.008s	0.011s	0.012s	0.006s	236	207
PAYMENT	100%	0.004s	0.005s	0.005s	0.004s	255	255
ORDER_STATUS	100%	0.010s	0.011s	0.011s	0.008s	20	9
STOCK_LEVEL	100%	0.010s	0.010s	0.010s	0.006s	24	0
DELIVERY	0%	0.003s	0.003s	0.003s	0.003s	22	22

Gemini Few-Shot

Transaction	**Execution Accuracy (%)**	**TPC-C Benchmark**					
		Latency					
		90th %	**95th %**	**99th %**	**Avg**	**Count**	**Errors**
NEW_ORDER	87.5%	0.007s	0.010s	0.012s	0.006s	230	211
PAYMENT	87.5%	0.009s	0.009s	0.010s	0.004s	284	169
ORDER_STATUS	100%	0.006s	0.007s	0.007s	0.004s	16	9
STOCK_LEVEL	100%	0.005s	0.005s	0.005s	0.004s	14	0
DELIVERY	100%	0.010s	0.011s	0.011s	0.007s	20	0

Gemini Few-Shot with Reasoning

Transaction	Execution Accuracy (%)	TPC-C Benchmark					
		Latency				Count	Errors
		90th %	95th %	99th %	Avg		
NEW_ORDER	87.5%	0.011s	0.012s	0.012s	0.006s	265	3
PAYMENT	100%	0.008s	0.008s	0.010s	0.004s	252	146
ORDER_STATUS	100%	0.010s	0.011s	0.011s	0.007s	14	6
STOCK_LEVEL	100%	0.009s	0.009s	0.010s	0.006s	24	0
DELIVERY	12.5%	0.013s	0.013s	0.013s	0.008s	26	0

8.1.3.3 Model: GPT-5.1

GPT Zero-Shot

Transaction	Execution Accuracy (%)	TPC-C Benchmark					
		Latency				Count	Errors
		90th %	95th %	99th %	Avg		
NEW_ORDER	62.5%	0.009s	0.012s	0.013s	0.006s	234	0
PAYMENT	87.5%	0.008s	0.008s	0.010s	0.005s	272	29
ORDER_STATUS	100%	0.007s	0.007s	0.007s	0.005s	18	0
STOCK_LEVEL	100%	0.004s	0.004s	0.004s	0.003s	24	0
DELIVERY	100%	0.010s	0.011s	0.011s	0.007s	18	0

GPT Zero-Shot with Reasoning

Transaction	Execution Accuracy (%)	TPC-C Benchmark					
		Latency				Count	Errors
		90th %	95th %	99th %	Avg		
NEW_ORDER	62.5%	0.008s	0.009s	0.010s	0.005s	257	0
PAYMENT	100%	0.008s	0.008s	0.009s	0.005s	239	0
ORDER_STATUS	100%	0.005s	0.005s	0.005s	0.003s	28	21
STOCK_LEVEL	100%	0.010s	0.010s	0.010s	0.005s	20	0
DELIVERY	100%	0.009s	0.010s	0.010s	0.006s	30	0

GPT Few-Shot

Transaction	Execution Accuracy (%)	TPC-C Benchmark					
		Latency					
		90th %	95th %	99th %	Avg	Count	Errors
NEW_ORDER	75%	0.008s	0.010s	0.016s	0.006s	266	236
PAYMENT	87.5%	0.010s	0.010s	0.012s	0.006s	248	0
ORDER_STATUS	100%	0.008s	0.008s	0.009s	0.006s	22	0
STOCK_LEVEL	100%	0.004s	0.004s	0.005s	0.003s	34	0
DELIVERY	100%	0.011s	0.011s	0.011s	0.007s	30	0

GPT Few-Shot with Reasoning

Transaction	Execution Accuracy (%)	TPC-C Benchmark					
		Latency					
		90th %	95th %	99th %	Avg	Count	Errors
NEW_ORDER	37.5%	0.008s	0.010s	0.012s	0.006s	257	210
PAYMENT	87.5%	0.024s	0.025s	0.028s	0.019s	242	0
ORDER_STATUS	100%	0.007s	0.008s	0.008s	0.006s	24	0
STOCK_LEVEL	100%	0.005s	0.005s	0.005s	0.003s	19	0
DELIVERY	100%	0.010s	0.010s	0.010s	0.007s	20	0

8.1.3.4 Model: DeepSeek V3.1

DeepSeek Zero-Shot

Transaction	Execution Accuracy (%)	TPC-C Benchmark					
		Latency					
		90th %	95th %	99th %	Avg	Count	Errors
NEW_ORDER	0%	0.006s	0.009s	0.010s	0.004s	250	250
PAYMENT	0%	0.003s	0.003s	0.004s	0.002s	255	255
ORDER_STATUS	100%	0.014s	0.014s	0.014s	0.006s	37	25
STOCK_LEVEL	100%	0.017s	0.017s	0.019s	0.009s	30	0
DELIVERY	0%	0.003s	0.004s	0.004s	0.003s	31	31

DeepSeek Zero-Shot with Reasoning

Transaction	Execution Accuracy (%)	TPC-C Benchmark					
		Latency				Count	Errors
		90th %	95th %	99th %	Avg		
NEW_ORDER	0%	0.002s	0.002s	0.002s	0.001s	263	263
PAYMENT	0%	0.003s	0.003s	0.003s	0.002s	275	275
ORDER_STATUS	100%	0.013s	0.013s	0.013s	0.007s	20	11
STOCK_LEVEL	100%	0.020s	0.021s	0.024s	0.012s	32	0
DELIVERY	100%	0.013s	0.013s	0.014s	0.011s	22	0

DeepSeek Few-Shot

Transaction	Execution Accuracy (%)	TPC-C Benchmark					
		Latency				Count	Errors
		90th %	95th %	99th %	Avg		
NEW_ORDER	87.5%	0.007s	0.007s	0.009s	0.006s	281	274
PAYMENT	87.5%	0.009s	0.010s	0.010s	0.005s	233	7
ORDER_STATUS	100%	0.008s	0.008s	0.008s	0.007s	16	16
STOCK_LEVEL	100%	0.004s	0.004s	0.004s	0.003s	26	0
DELIVERY	0%	0.003s	0.003s	0.003s	0.002s	34	34

DeepSeek Few-Shot with Reasoning

Transaction	Execution Accuracy (%)	TPC-C Benchmark					
		Latency				Count	Errors
		90th %	95th %	99th %	Avg		
NEW_ORDER	87.5%	0.011s	0.012s	0.013s	0.007s	256	232
PAYMENT	100%	0.009s	0.009s	0.009s	0.004s	193	119
ORDER_STATUS	100%	0.011s	0.012s	0.012s	0.005s	16	9
STOCK_LEVEL	100%	0.010s	0.010s	0.012s	0.006s	30	0
DELIVERY	100%	0.013s	0.013s	0.014s	0.008s	26	0

8.1.3.5 Model: Llama 4 Maverick

Llama Zero-Shot

Transaction	Execution Accuracy (%)	TPC-C Benchmark					
		Latency				Count	Errors
		90th %	95th %	99th %	Avg		
NEW_ORDER	0%	0.002s	0.003s	0.007s	0.002s	267	267
PAYMENT	0%	0.006s	0.006s	0.009s	0.005s	238	238
ORDER_STATUS	0%	0.016s	0.017s	0.017s	0.010s	20	12
STOCK_LEVEL	100%	0.010s	0.010s	0.010s	0.007s	10	0
DELIVERY	0%	0.006s	0.006s	0.007s	0.005s	24	24

Llama Zero-Shot with Reasoning

Transaction	Execution Accuracy (%)	TPC-C Benchmark					
		Latency				Count	Errors
		90th %	95th %	99th %	Avg		
NEW_ORDER	0%	0.003s	0.003s	0.006s	0.002s	260	260
PAYMENT	0%	0.003s	0.003s	0.007s	0.002s	237	237
ORDER_STATUS	0%	0.014s	0.014s	0.014s	0.009s	10	7
STOCK_LEVEL	100%	0.014s	0.015s	0.015s	0.009s	30	0
DELIVERY	0%	0.002s	0.002s	0.002s	0.001s	17	17

Llama Few-Shot

Transaction	Execution Accuracy (%)	TPC-C Benchmark					
		Latency				Count	Errors
		90th %	95th %	99th %	Avg		
NEW_ORDER	0%	0.002s	0.002s	0.003s	0.001s	252	252
PAYMENT	0%	0.004s	0.004s	0.007s	0.003s	251	251
ORDER_STATUS	50%	0.003s	0.003s	0.003s	0.003s	20	20
STOCK_LEVEL	100%	0.013s	0.013s	0.013s	0.009s	10	0
DELIVERY	0%	0.002s	0.002s	0.003s	0.002s	30	30

Llama Few-Shot with Reasoning

Transaction	Execution Accuracy (%)	TPC-C Benchmark					
		Latency				Count	Errors
		90th %	95th %	99th %	Avg		
NEW_ORDER	0%	0.004s	0.009s	0.010s	0.003s	263	263
PAYMENT	0%	0.002s	0.002s	0.004s	0.001s	225	225
ORDER_STATUS	100%	0.014s	0.014s	0.014s	0.011s	21	0
STOCK_LEVEL	100%	0.016s	0.017s	0.017s	0.011s	25	0
DELIVERY	0%	0.009s	0.009s	0.009s	0.007s	16	16

8.2 TPC-H

8.2.1 Level 1: Simple Prompting

Model: DeepSeek: R1 Distill Llama 70B

Query	Structural Similarity
Q1	94%
Q2	69%
Q3	70%
Q4	77%
Q5	60%
Q6	60%
Q7	75%
Q8	38%
Q9	68%
Q10	63%
Q11	54%
Q12	52%
Q13	73%
Q14	69%
Q15	40%
Q16	67%
Q17	63%
Q18	61%
Q19	41%
Q20	44%
Q21	58%
Q22	59%
Average Similarity	62%

Model: Google: Gemini 2.0 Flash Experimental

Query	**Structural Similarity**
Q1	85%
Q2	0%
Q3	59%
Q4	56%
Q5	66%
Q6	82%
Q7	100%
Q8	61%
Q9	75%
Q10	73%
Q11	91%
Q12	96%
Q13	84%
Q14	100%
Q15	51%
Q16	62%
Q17	57%
Q18	64%
Q19	49%
Q20	48%
Q21	65%
Q22	53%
Average Similarity	67%

Model: Meta: Llama 3.3 70B

Query	**Structural Similarity**
Q1	100%
Q2	0%
Q3	97%
Q4	96%
Q5	73%
Q6	100%
Q7	84%
Q8	63%
Q9	81%
Q10	85%

Query	Structural Similarity
Q11	76%
Q12	70%
Q13	75%
Q14	55%
Q15	21%
Q16	55%
Q17	45%
Q18	61%
Q19	45%
Q20	45%
Q21	76%
Q22	48%
Average Similarity	65%

8.2.2 Level 1: Iteractive refinement prompting

Model: Chatgpt

Query	Structural Similarity	Number of Prompts
Q1	94%	1
Q2	0%	1
Q3	83%	2
Q4	67%	2
Q5	65%	2
Q6	100%	1
Q7	83%	2
Q8	75%	2
Q9	71%	1
Q10	70%	2
Q11	58%	2
Q12	75%	2
Q13	74%	2
Q14	81%	1
Q15	37%	3
Q16	70%	2
Q17	53%	3
Q18	57%	2
Q19	50%	3
Q20	48%	3
Average Similarity	65%	

Query	Structural Similarity	Number of Prompts
Q21	62%	2
Q22	63%	1
Average Similarity	65%	

Model: Gemini

Query	Structural Similarity	Number of Prompts
Q1	100%	1
Q2	0%	2
Q3	100%	1
Q4	83%	2
Q5	38%	2
Q6	100%	1
Q7	91%	2
Q8	75%	2
Q9	87%	2
Q10	100%	1
Q11	100%	1
Q12	100%	1
Q13	96%	1
Q14	100%	1
Q15	53%	2
Q16	100%	1
Q17	79%	2
Q18	100%	1
Q19	95%	1
Q20	78%	2
Q21	58%	2
Q22	69%	2
Average Similarity	82%	

Model: Deepseek

Query	Structural Similarity	Number of Prompts
Q1	100%	1
Q2	0%	2
Q3	96%	1
Average Similarity	82%	

Query	**Structural Similarity**	**Number of Prompts**
Q4	96%	1
Q5	85%	2
Q6	91%	1
Q7	85%	2
Q8	75%	2
Q9	84%	1
Q10	100%	1
Q11	91%	2
Q12	100%	1
Q13	85%	2
Q14	100%	1
Q15	35%	2
Q16	100%	1
Q17	79%	2
Q18	81%	1
Q19	64%	2
Q20	81%	1
Q21	89%	1
Q22	86%	2
Average Similarity	82%	

8.2.3 Level 2: Context-Aware SQL Generation

8.2.3.1 Model: Gemini 2.5 Pro

Gemini Zero-Shot

Query	Correct	Time (s)	Ref Time (s)
Q1	1	0.706	0.499
Q2	1	8.333	5.236
Q3	1	0.131	0.139
Q4	1	0.201	0.181
Q5	1	0.108	0.108
Q6	1	0.098	0.072
Q7	1	0.128	0.093
Q8	1	0.119	0.086
Q9	1	0.262	0.181
Q10	0	0.209	0.117
Q11	1	0.038	0.029
Q12	1	0.169	0.111
Q13	1	0.275	0.235
Q14	1	0.099	0.091
Q15	1	0.103	0.008
Q16	1	0.087	0.083
Q17	1	204.273	209.713
Q18	1	1.633	1.614
Q19	1	0.134	0.132
Q21	1	0.440	0.443
Q22	1	0.057	0.054
Execution Accuracy (EX)		**95.24%**	
Valid Efficiency Score (VES)		**77.86%**	

Gemini Zero-Shot with Reasoning

Query	Correct	Time (s)	Ref Time (s)
Q1	1	0.700	0.499
Q2	1	6.773	5.236
Q3	1	0.109	0.139
Q4	1	0.213	0.181
Q5	1	0.106	0.108
Q6	1	0.094	0.072
Q7	1	0.126	0.093
Q8	1	0.118	0.086
Q9	1	0.253	0.181
Q10	0	0.162	0.117
Q11	1	0.038	0.029
Q12	1	0.147	0.111
Q13	1	0.271	0.235
Q14	1	0.096	0.091
Q15	1	0.100	0.008
Q16	1	0.084	0.083
Q17	1	202.031	209.713
Q18	1	1.962	1.614
Q19	1	0.138	0.132
Q21	1	0.450	0.443
Q22	1	0.057	0.054
Execution Accuracy (EX)		**95.24%**	
Valid Efficiency Score (VES)		**79.66%**	

Gemini Few-Shot

Query	Ref Time(s)	2Ex, λ=0.4		2Ex, λ=0.6		5Ex, λ=0.4		5Ex, λ=0.6	
		Correct	t(s)	Correct	t(s)	Correct	t(s)	Correct	t(s)
Q1	0.499	1	0.503	1	0.776	1	0.696	1	0.738
Q2	5.236	1	6.060	1	7.983	1	8.431	1	8.397
Q3	0.139	1	0.100	1	0.154	1	0.132	1	0.132
Q4	0.181	1	0.153	1	0.207	1	0.200	1	0.201
Q5	0.108	1	0.087	1	0.108	1	0.108	1	0.109
Q6	0.072	1	0.076	1	0.096	1	0.096	1	0.098
Q7	0.093	1	0.112	1	0.130	1	0.130	1	0.129
Q8	0.086	1	0.094	1	0.121	1	0.121	1	0.123
Q9	0.181	1	0.241	1	0.249	1	0.252	1	0.250
Q10	0.117	1	0.178	1	0.164	0	0.164	0	0.190
Q11	0.029	1	0.041	1	0.040	1	0.039	1	0.039
Q12	0.111	1	0.154	1	0.152	1	0.150	1	0.150
Q13	0.235	1	0.280	1	0.279	1	0.275	1	0.278
Q14	0.091	1	0.102	1	0.099	1	0.099	1	0.099
Q15	0.008	1	0.104	1	0.106	1	0.110	1	0.103
Q16	0.083	1	0.087	1	0.093	1	0.086	1	0.085
Q17	209.713	1	217.377	1	219.050	1	205.933	0	0.197
Q18	1.614	1	1.590	1	1.831	1	1.651	1	1.890
Q19	0.132	1	0.140	1	0.137	1	0.135	1	0.135
Q21	0.443	0	0.458	1	0.460	0	0.704	0	0.451
Q22	0.054	1	0.059	1	0.057	1	0.060	1	0.059
EX		**95.24%**		**100.00%**		**90.48%**		**85.71%**	
VES		**84.76%**		**79.00%**		**73.14%**		**67.49%**	

Gemini Few-Shot with Reasoning

Query	Ref Time(s)	2Ex, λ=0.4 Correct	2Ex, λ=0.4 t(s)	2Ex, λ=0.6 Correct	2Ex, λ=0.6 t(s)	5Ex, λ=0.4 Correct	5Ex, λ=0.4 t(s)	5Ex, λ=0.6 Correct	5Ex, λ=0.6 t(s)
Q1	0.499	1	0.504	1	0.510	1	0.707	1	0.708
Q2	5.236	1	7.290	1	5.491	1	8.421	1	8.390
Q3	0.139	1	0.136	1	0.101	1	0.134	1	0.133
Q4	0.181	1	0.201	1	0.151	1	0.199	1	0.204
Q5	0.108	1	0.108	1	0.085	1	0.109	1	0.109
Q6	0.072	1	0.097	1	0.072	1	0.097	1	0.097
Q7	0.093	1	0.134	1	0.106	1	0.130	1	0.142
Q8	0.086	1	0.119	1	0.090	1	0.120	1	0.120
Q9	0.181	1	0.252	1	0.187	1	0.263	1	0.247
Q10	0.117	1	0.162	1	0.122	1	0.167	1	0.160
Q11	0.029	1	0.040	1	0.032	1	0.043	1	0.040
Q12	0.111	1	0.168	1	0.112	1	0.151	1	0.150
Q13	0.235	1	0.276	1	0.186	1	0.278	1	0.275
Q14	0.091	1	0.098	1	0.075	1	0.100	1	0.099
Q15	0.008	1	0.105	1	0.079	1	0.103	1	0.199
Q16	0.083	1	0.086	1	0.077	1	0.086	1	0.084
Q17	209.713	1	212.518	0	0.142	1	205.344	1	204.554
Q18	1.614	1	1.592	1	1.176	1	1.622	1	1.605
Q19	0.132	1	0.137	0	0.000	1	0.135	1	0.142
Q21	0.443	1	0.456	1	0.351	1	0.471	0	0.445
Q22	0.054	1	0.058	1	0.042	1	0.057	1	0.059
EX		**100.00%**		**90.48%**		**100.00%**		**95.24%**	
VES		**82.40%**		**95.29%**		**80.56%**		**76.06%**	

8.2.3.2 Model: GPT-5.1

GPT Zero-Shot

Query	Correct	Time (s)	Ref Time (s)
Q1	0	0.696	0.499
Q2	1	7.871	5.236
Q3	1	0.147	0.139
Q4	1	0.142	0.181
Q5	1	0.106	0.108
Q6	1	0.094	0.072
Q7	1	0.125	0.093
Q8	0	0.456	0.086
Q9	1	0.266	0.181
Q10	0	0.163	0.117
Q11	1	0.038	0.029
Q12	1	0.148	0.111
Q13	1	0.676	0.235
Q14	1	0.095	0.091
Q15	1	0.101	0.008
Q16	1	0.153	0.083
Q17	0	202.399	209.713
Q18	1	2.600	1.614
Q19	1	0.132	0.132
Q21	0	0.286	0.443
Q22	0	0.054	0.054
Execution Accuracy (EX)		**71.43%**	
Valid Efficiency Score (VES)		**53.12%**	

GPT Zero-Shot with Reasoning

Query	Correct	Time (s)	Ref Time (s)
Q1	1	0.700	0.499
Q2	1	7.914	5.236
Q3	1	0.137	0.139
Q4	1	0.145	0.181
Q5	1	0.114	0.108
Q6	1	0.100	0.072
Q7	1	0.148	0.093
Q8	0	0.324	0.086
Q9	0	0.198	0.181
Q10	0	0.200	0.117
Q11	1	0.029	0.029
Q12	1	0.165	0.111
Q13	1	0.291	0.235
Q14	1	0.101	0.091
Q15	0	0.111	0.008
Q16	0	0.157	0.083
Q17	0	0.001	209.713
Q18	1	2.658	1.614
Q19	1	0.181	0.132
Q21	0	4.443	0.443
Q22	0	0.049	0.054
Execution Accuracy (EX)		**61.90%**	
Valid Efficiency Score (VES)		**50.72%**	

GPT Few-Shot

Query	Ref Time(s)	2Ex, λ=0.4		2Ex, λ=0.6		5Ex, λ=0.4		5Ex, λ=0.6	
		Correct	t(s)	Correct	t(s)	Correct	t(s)	Correct	t(s)
Q1	0.499	0	0.719	0	0.707	0	0.704	1	0.717
Q2	5.236	1	6.204	1	0.050	1	8.339	1	10.842
Q3	0.139	1	0.137	1	0.136	1	0.145	1	0.137
Q4	0.181	1	0.142	1	0.153	1	0.147	1	0.144
Q5	0.108	1	0.108	1	0.159	1	0.112	1	0.109
Q6	0.072	1	0.096	1	0.102	1	0.095	1	0.097
Q7	0.093	1	0.129	1	0.128	1	0.341	1	0.134
Q8	0.086	0	0.131	0	0.137	0	0.097	0	0.001
Q9	0.181	1	0.253	1	0.267	1	0.271	0	0.225
Q10	0.117	0	0.210	0	0.199	0	0.168	0	0.164
Q11	0.029	1	0.044	1	0.040	1	0.051	1	0.040
Q12	0.111	0	0.151	1	0.150	1	0.148	1	0.146
Q13	0.235	1	0.273	1	0.273	1	0.283	1	0.281
Q14	0.091	1	0.104	1	0.098	1	0.099	1	0.099
Q15	0.008	0	0.206	0	0.209	1	0.222	0	0.206
Q16	0.083	1	0.153	1	0.152	1	0.157	1	0.168
Q17	209.713	0	0.191	0	0.263	0	0.256	0	0.150
Q18	1.614	1	1.533	1	1.739	1	1.525	1	2.562
Q19	0.132	0	0.205	0	0.164	0	0.142	0	0.148
Q21	0.443	0	0.455	0	0.579	0	0.581	1	0.479
Q22	0.054	0	0.047	0	0.048	0	0.103	0	0.051
EX		**57.14%**		**61.90%**		**66.67%**		**66.67%**	
VES		**49.09%**		**51.15%**		**48.44%**		**53.18%**	

GPT Few-Shot with Reasoning

Query	Ref	2Ex, λ=0.4		2Ex, λ=0.6		5Ex, λ=0.4		5Ex, λ=0.6	
	Time (s)	Correct	t(s)	Correct	t(s)	Correct	t(s)	Correct	t(s)
Q1	0.499	1	0.500	1	0.743	1	0.704	1	0.697
Q2	5.236	1	7.369	1	8.115	1	8.163	1	8.097
Q3	0.139	1	0.130	1	0.128	1	0.130	1	0.128
Q4	0.181	1	0.278	1	0.144	1	0.146	1	0.142
Q5	0.108	1	0.121	0	0.135	0	0.000	1	0.107
Q6	0.072	1	0.100	1	0.094	1	0.094	1	0.094
Q7	0.093	1	0.137	1	0.125	1	0.125	1	0.125
Q8	0.086	0	0.136	0	0.001	0	0.365	0	0.096
Q9	0.181	0	0.232	0	0.253	1	0.275	1	0.258
Q10	0.117	0	0.212	0	0.199	0	0.249	0	0.160
Q11	0.029	1	0.038	1	0.053	1	0.079	1	0.038
Q12	0.111	1	0.145	1	0.147	1	0.168	1	0.146
Q13	0.235	0	0.274	1	0.223	0	0.299	1	0.272
Q14	0.091	1	0.097	1	0.094	1	0.096	1	0.095
Q15	0.008	0	0.201	1	0.100	1	0.227	1	0.113
Q16	0.083	1	0.150	1	0.084	1	0.157	1	0.150
Q17	209.713	0	0.182	0	0.261	0	0.307	0	0.133
Q18	1.614	1	1.488	1	1.484	1	2.648	1	2.548
Q19	0.132	0	0.111	0	0.112	0	0.148	0	0.103
Q21	0.443	0	0.467	1	0.439	0	0.000	0	0.699
Q22	0.054	0	0.052	0	0.049	0	0.048	0	0.054
EX		**57.14%**		**66.67%**		**61.90%**		**71.43%**	
VES		**46.79%**		**55.52%**		**42.75%**		**54.99%**	

8.2.3.3 Model: DeepSeek V3.1

DeepSeek Zero-Shot

Query	Correct	Time (s)	Ref Time (s)
Q1	1	0.817	0.499
Q2	1	8.531	5.236
Q3	1	0.134	0.139
Q4	1	0.149	0.181
Q5	0	0.172	0.108
Q6	1	0.096	0.072
Q7	1	0.132	0.093
Q8	0	471.540	0.086
Q9	0	0.211	0.181
Q10	0	0.167	0.117
Q11	1	0.041	0.029
Q12	1	0.148	0.111
Q13	0	0.275	0.235
Q14	1	0.095	0.091
Q15	0	0.142	0.008
Q16	1	0.152	0.083
Q17	0	0.323	209.713
Q18	1	2.576	1.614
Q19	0	0.115	0.132
Q21	0	0.750	0.443
Q22	0	0.050	0.054
Execution Accuracy (EX)		**52.38%**	
Valid Efficiency Score (VES)		**40.57%**	

DeepSeek Zero-Shot with Reasoning

Query	Correct	Time (s)	Ref Time (s)
Q1	1	0.818	0.499
Q2	1	5.452	5.236
Q3	1	0.132	0.139
Q4	1	0.143	0.181
Q5	0	0.139	0.108
Q6	1	0.095	0.072
Q7	1	0.129	0.093
Q8	0	38.788	0.086
Q9	0	0.203	0.181
Q10	0	0.160	0.117
Q11	1	0.038	0.029
Q12	1	0.148	0.111
Q13	1	0.268	0.235
Q14	0	0.099	0.091
Q15	0	0.142	0.008
Q16	1	0.152	0.083
Q17	0	0.184	209.713
Q18	1	2.549	1.614
Q19	0	0.138	0.132
Q21	0	1.152	0.443
Q22	0	0.057	0.054
Execution Accuracy (EX)		**52.38%**	
Valid Efficiency Score (VES)		**42.56%**	

DeepSeek Few-Shot

Query	Ref Time(s)	2Ex, λ=0.4		2Ex, λ=0.6		5Ex, λ=0.4		5Ex, λ=0.6	
		Correct	t(s)	Correct	t(s)	Correct	t(s)	Correct	t(s)
Q1	0.499	1	0.608	1	0.750	1	0.701	1	0.708
Q2	5.236	0	5.623	0	8.258	1	5.674	0	0.021
Q3	0.139	0	0.099	0	0.132	1	0.132	1	0.149
Q4	0.181	1	0.107	1	0.144	1	0.157	1	0.146
Q5	0.108	0	0.103	1	0.107	1	0.110	0	0.137
Q6	0.072	1	0.075	1	0.096	1	0.096	1	0.096
Q7	0.093	1	0.096	1	0.130	1	0.131	1	0.127
Q8	0.086	0	93.399	0	0.109	0	0.118	0	0.115
Q9	0.181	1	0.257	1	0.256	1	0.264	0	0.226
Q10	0.117	0	0.162	0	0.162	0	0.141	0	0.142
Q11	0.029	1	0.039	1	0.038	1	0.038	0	0.000
Q12	0.111	1	0.148	1	0.148	1	0.148	1	0.155
Q13	0.235	0	0.001	0	0.001	1	0.275	1	0.302
Q14	0.091	1	0.096	1	0.115	1	0.098	1	0.102
Q15	0.008	0	0.268	1	0.341	0	0.289	0	0.180
Q16	0.083	1	0.157	0	0.157	0	0.156	0	0.157
Q17	209.713	0	0.246	0	0.247	0	0.277	0	0.246
Q18	1.614	1	2.597	1	2.632	1	2.657	1	2.643
Q19	0.132	0	0.137	0	0.135	0	0.115	0	0.592
Q21	0.443	0	1.369	0	0.453	0	0.686	0	1.140
Q22	0.054	0	0.052	0	0.049	0	0.054	0	0.059
EX		**47.62%**		**52.38%**		**61.90%**		**42.86%**	
VES		**41.60%**		**38.21%**		**51.35%**		**35.03%**	

DeepSeek Few-Shot with Reasoning

Query	Ref Time(s)	2Ex, λ=0.4		2Ex, λ=0.6		5Ex, λ=0.4		5Ex, λ=0.6	
		Correct	t(s)	Correct	t(s)	Correct	t(s)	Correct	t(s)
Q1	0.499	1	0.706	1	0.703	1	0.498	1	0.513
Q2	5.236	1	8.370	1	8.351	1	5.656	1	7.767
Q3	0.139	1	0.134	1	0.131	1	0.108	1	0.129
Q4	0.181	1	0.146	1	0.144	1	0.138	1	0.143
Q5	0.108	1	0.107	1	0.116	1	0.107	1	0.105
Q6	0.072	1	0.096	1	0.111	1	0.097	1	0.094
Q7	0.093	1	0.126	1	0.127	0	0.000	1	0.126
Q8	0.086	1	0.118	1	0.118	0	0.215	1	0.118
Q9	0.181	1	0.255	1	0.245	1	0.246	1	0.243
Q10	0.117	0	0.012	0	0.140	0	0.160	0	0.180
Q11	0.029	1	0.050	1	0.053	0	0.000	1	0.051
Q12	0.111	1	0.149	1	0.148	0	-	1	0.149
Q13	0.235	1	0.201	1	0.196	1	0.270	1	0.273
Q14	0.091	1	0.097	1	0.096	1	0.100	1	0.094
Q15	0.008	1	0.152	1	0.150	0	0.138	1	0.151
Q16	0.083	1	0.084	1	0.084	1	0.086	1	0.085
Q17	209.713	0	0.246	0	0.132	0	0.183	0	0.138
Q18	1.614	1	2.565	1	2.566	1	2.560	1	2.541
Q19	0.132	1	0.117	0	0.142	0	0.073	0	0.114
Q21	0.443	0	31.187	0	0.730	0	1.658	0	0.656
Q22	0.054	0	0.052	0	0.050	0	0.080	0	0.000
EX		**80.95%**		**76.19%**		**52.38%**		**76.19%**	
VES		**65.60%**		**59.76%**		**49.47%**		**60.97%**	

8.2.3.4 Model: Llama 4 Maverick

Llama Zero-Shot

Query	**Correct**	**Time (s)**	**Ref Time (s)**
Q1	1	0.846	0.499
Q2	0	0.025	5.236
Q3	1	0.146	0.139
Q4	0	0.001	0.181
Q5	1	0.159	0.108
Q6	1	0.095	0.072
Q7	0	0.125	0.093
Q8	0	86.554	0.086
Q9	1	0.244	0.181
Q10	0	0.022	0.117
Q11	0	0.027	0.029
Q12	0	0.127	0.111
Q13	1	0.665	0.235
Q14	1	0.119	0.091
Q15	1	0.102	0.008
Q16	1	0.152	0.083
Q17	0	0.183	209.713
Q18	1	1.802	1.614
Q19	0	0.136	0.132
Q21	1	0.441	0.443
Q22	0	0.052	0.054
Accuracy		**52.38%**	
VES		**35.06%**	

Llama Zero-Shot with Reasoning

Query	**Correct**	**Time (s)**	**Ref Time (s)**
Q1	1	0.596	0.499
Q2	1	5.869	5.236
Q3	0	0.075	0.139
Q4	0	0.000	0.181
Q5	1	0.091	0.108
Q6	1	0.070	0.072
Q7	0	0.001	0.093
Q8	0	0.268	0.086
Q9	1	0.203	0.181
Q10	0	0.119	0.117
Q11	0	0.020	0.029
Q12	0	0.102	0.111
Q13	0	0.226	0.235
Q14	0	0.096	0.091
Q15	1	0.085	0.008
Q16	1	0.118	0.083
Q17	0	0.176	209.713
Q18	1	1.410	1.614
Q19	0	0.106	0.132
Q21	1	0.337	0.443
Q22	0	0.036	0.054
Execution Accuracy (EX)		**42.86%**	
Valid Efficiency Score (VES)		**38.54%**	

Llama Few-Shot

Query	**Ref Time(s)**	**2Ex, λ=0.4**		**2Ex, λ=0.6**		**5Ex, λ=0.4**		**5Ex, λ=0.6**	
		Correct	**t(s)**	**Correct**	**t(s)**	**Correct**	**t(s)**	**Correct**	**t(s)**
Q1	0.499	1	0.629	1	0.527	1	0.515	1	0.696
Q2	5.236	1	5.859	1	6.042	1	6.083	1	8.249
Q3	0.139	0	0.001	1	0.105	0	0.110	0	0.160
Q4	0.181	1	0.152	1	0.186	1	0.191	1	0.197
Q5	0.108	1	0.083	1	0.105	1	0.103	1	0.113
Q6	0.072	1	0.071	1	0.096	1	0.103	1	0.096
Q7	0.093	0	0.001	1	0.138	1	0.125	0	0.126
Q8	0.086	0	0.000	0	0.001	0	0.000	0	0.155
Q9	0.181	1	0.266	0	0.001	1	0.241	1	0.265
Q10	0.117	0	0.002	0	0.207	0	0.158	0	0.203
Q11	0.029	1	0.034	1	0.037	1	0.039	0	0.001
Q12	0.111	1	0.109	0	0.151	1	0.144	1	0.146
Q13	0.235	0	0.000	0	0.023	1	0.273	1	0.264
Q14	0.091	1	0.075	1	0.108	1	0.095	1	0.095
Q15	0.008	0	0.102	0	0.003	0	0.130	0	0.143
Q16	0.083	1	0.114	0	0.025	0	0.173	0	0.150
Q17	209.713	0	0.137	0	0.140	0	0.228	0	0.188
Q18	1.614	1	1.421	1	1.887	1	1.569	1	1.672
Q19	0.132	0	0.106	0	0.124	0	0.147	0	723.139
Q21	0.443	0	0.001	1	0.498	0	0.433	0	0.517
Q22	0.054	0	0.026	0	0.070	0	0.053	0	0.047
EX		**52.38%**		**52.38%**		**57.14%**		**47.62%**	
VES		**51.53%**		**47.31%**		**49.44%**		**39.20%**	

Llama Few-Shot with Reasoning

Query	Ref Time(s)	2Ex, λ=0.4		2Ex, λ=0.6		5Ex, λ=0.4		5Ex, λ=0.6	
		Correct	t(s)	Correct	t(s)	Correct	t(s)	Correct	t(s)
Q1	0.499	1	0.904	1	0.502	1	0.702	1	0.717
Q2	5.236	1	6.172	1	5.349	1	8.024	0	0.003
Q3	0.139	0	0.001	0	0.001	0	0.001	1	0.143
Q4	0.181	1	0.311	1	0.148	1	0.197	1	0.196
Q5	0.108	1	0.096	1	0.081	1	0.110	1	0.107
Q6	0.072	0	0.087	1	0.072	1	0.107	1	0.095
Q7	0.093	1	0.098	1	0.096	1	0.138	1	0.130
Q8	0.086	0	0.112	1	0.098	0	0.001	0	0.001
Q9	0.181	1	0.228	1	0.183	1	0.266	1	0.249
Q10	0.117	0	0.175	0	0.148	0	0.153	0	0.146
Q11	0.029	1	0.033	1	0.032	1	0.042	1	0.037
Q12	0.111	1	0.116	1	0.108	1	0.152	1	0.147
Q13	0.235	1	0.181	1	0.172	0	0.021	1	0.277
Q14	0.091	1	0.090	0	0.062	1	0.099	1	0.101
Q15	0.008	0	0.099	0	0.079	0	0.124	0	0.133
Q16	0.083	1	0.121	0	0.020	1	0.151	1	0.152
Q17	209.713	0	0.191	0	0.052	0	0.194	0	0.182
Q18	1.614	1	1.392	1	1.312	1	1.882	0	0.000
Q19	0.132	0	0.117	0	0.106	0	0.147	0	0.146
Q21	0.443	0	1.655	1	0.331	0	0.454	0	0.150
Q22	0.054	0	0.057	0	0.039	0	0.121	0	0.340
EX		**57.14%**		**61.90%**		**57.14%**		**57.14%**	
VES		**51.62%**		**67.78%**		**43.04%**		**45.88%**	

8.2.4 Level 3: Direct Data Access

Questions 21-23 are an addition to test some concepts and capabilities of the models. They are not part of the TPC-H benchmark.

LLama FT

Query	Exact Match	Numeric Match	Token-level F1	MAE
Q1	0	0	0.000	NaN
Q2	0	0	0.000	NaN
Q3	0	0	0.000	NaN
Q4	0	0	0.000	NaN
Q5	0	0	0.000	NaN
Q6	0	0	0.000	NaN
Q7	0	0	0.000	NaN

Query	Exact Match	Numeric Match	Token-level F1	MAE
Q8	0	0	0.000	NaN
Q9	0	0	0.000	NaN
Q10	0	0	0.000	NaN
Q11	0	0	0.000	NaN
Q12	0	0	0.000	NaN
Q13	0	0	0.000	NaN
Q14	0	0	0.000	NaN
Q15	0	0	0.000	NaN
Q16	0	0	0.000	NaN
Q17	0	0	0.000	NaN
Q18	0	0	0.000	NaN
Q19	0	0	0.000	NaN
Q20	0	0	0.000	NaN
Q21	0	0	0.000	NaN
Q22	0	0	0.000	NaN
Q23	0	0	0.054	NaN

Qwen 3B FT

Query	Exact Match	Numeric Match	Token-level F1	MAE
Q1	0	0	0.000	NaN
Q2	0	0	0.100	NaN
Q3	0	0	0.000	NaN
Q4	0	0	0.000	NaN
Q5	0	0	0.000	NaN
Q6	0	0	0.000	NaN
Q7	0	0	0.000	NaN
Q8	0	0	0.000	NaN
Q9	0	0	0.000	NaN
Q10	0	0	0.058	NaN
Q11	0	0	0.000	NaN
Q12	0	0	0.000	NaN
Q13	0	0	0.000	NaN
Q14	0	0	0.000	NaN
Q15	0	0	0.000	NaN
Q16	0	0	0.000	NaN
Q17	0	0	0.000	NaN
Q18	0	0	0.000	NaN
Q19	0	0	0.000	NaN
Q20	0	0	0.000	NaN

Query	Exact Match	Numeric Match	Token-level F1	MAE
Q21	0	0	0.000	NaN
Q22	0	0	0.000	NaN
Q23	0	0	0.109	NaN

Qwen 7B (3 epochs) FT

Query	Exact Match	Numeric Match	Token-level F1	MAE
Q1	0	0	0.000	NaN
Q2	0	0	0.100	NaN
Q3	0	0	0.000	NaN
Q4	0	0	0.000	NaN
Q5	0	0	0.000	NaN
Q6	0	0	0.000	NaN
Q7	0	0	0.000	NaN
Q8	0	0	0.000	NaN
Q9	0	0	0.000	NaN
Q10	0	0	0.058	NaN
Q11	0	0	0.000	NaN
Q12	0	0	0.000	NaN
Q13	0	0	0.000	NaN
Q14	0	0	0.000	NaN
Q15	0	0	0.000	NaN
Q16	0	0	0.000	NaN
Q17	0	0	0.000	NaN
Q18	0	0	0.000	NaN
Q19	0	0	0.000	NaN
Q20	0	0	0.000	NaN
Q21	0	0	0.000	NaN
Q22	0	0	0.000	NaN
Q23	0	0	0.109	NaN

Qwen 7B (1 epochs) FT

Query	Exact Match	Numeric Match	Token-level F1	MAE
Q1	0	0	0.000	NaN
Q2	0	0	0.010	NaN
Q3	0	0	0.000	NaN
Q4	0	0	0.143	NaN
Q5	0	0	0.000	NaN
Q6	0	0	0.000	75493.210
Q7	0	0	0.000	NaN

Query	Exact Match	Numeric Match	Token-level F1	MAE
Q8	0	0	0.000	NaN
Q9	0	0	0.060	NaN
Q10	0	0	0.000	NaN
Q11	0	0	0.000	NaN
Q12	0	0	0.167	NaN
Q13	0	0	0.000	NaN
Q14	0	0	0.000	7
Q15	0	0	0.000	NaN
Q16	0	0	0.000	NaN
Q17	0	0	0.000	NaN
Q18	0	0	0.000	NaN
Q19	0	0	0.000	NaN
Q20	0	0	0.000	NaN
Q21	0	0	0.000	1
Q22	0	0	0.000	19
Q23	0	0	0.000	NaN

Mistral FT

Query	Exact Match	Numeric Match	Token-level F1	MAE
Q1	0	0	0.000	NaN
Q2	0	0	0.000	NaN
Q3	0	0	0.000	NaN
Q4	0	0	0.000	NaN
Q5	0	0	0.000	NaN
Q6	0	0	0.000	NaN
Q7	0	0	0.000	NaN
Q8	0	0	0.000	NaN
Q9	0	0	0.000	NaN
Q10	0	0	0.017	NaN
Q11	0	0	0.000	NaN
Q12	0	0	0.000	NaN
Q13	0	0	0.000	NaN
Q14	0	0	0.000	NaN
Q15	0	0	0.000	NaN
Q16	0	0	0.000	NaN
Q17	0	0	0.000	NaN
Q18	1	0	0.000	NaN
Q19	0	0	0.000	NaN
Q20	0	0	0.000	NaN

Query	Exact Match	Numeric Match	Token-level F1	MAE
Q21	0	0	0.000	NaN
Q22	0	0	0.000	NaN
Q23	0	0	0.194	NaN

Qwen RAG

Query	Exact Match	Numeric Match	Token-level F1	MAE
Q1	0	0	0.068	NaN
Q2	0	0	0.000	NaN
Q3	0	0	0.000	NaN
Q4	0	0	0.500	NaN
Q5	0	0	0.000	NaN
Q6	0	0	0.000	72028.739
Q7	1	0	0.000	NaN
Q8	0	0	0.000	NaN
Q9	0	0	0.000	NaN
Q10	0	0	0.000	NaN
Q11	0	0	0.043	NaN
Q12	0	0	0.267	NaN
Q13	0	0	0.375	NaN
Q14	0	1	0.000	NaN
Q15	0	0	0.000	NaN
Q16	0	0	0.000	NaN
Q17	0	0	0.000	NaN
Q18	0	0	0.000	NaN
Q19	1	0	0.000	NaN
Q20	0	0	0.000	NaN
Q21	1	1	1.000	0
Q22	1	1	1.000	0

Mistral RAG

Query	Exact Match	Numeric Match	Token-level F1	MAE
Q1	0	0	0.100	NaN
Q2	0	0	0.000	NaN
Q3	0	0	0.000	NaN
Q4	0	0	0.444	NaN
Q5	0	0	0.000	NaN
Q6	0	0	0.000	NaN
Q7	1	0	0.000	NaN
Q8	1	0	0.000	NaN

Query	Exact Match	Numeric Match	Token-level F1	MAE
Q9	0	0	0.000	NaN
Q10	0	0	0.006	NaN
Q11	0	0	0.000	NaN
Q12	0	0	0.191	NaN
Q13	0	0	0.211	NaN
Q14	0	0	0.000	NaN
Q15	0	0	0.000	NaN
Q16	0	0	0.000	NaN
Q17	1	0	0.000	NaN
Q18	1	0	0.000	NaN
Q19	1	0	0.000	NaN
Q20	0	0	0.000	NaN
Q21	1	1	1.000	0
Q22	0	0	0.000	1

LLama RAG

Query	Exact Match	Numeric Match	Token-level F1	MAE
Q1	0	0	0.068	NaN
Q2	0	0	0.064	NaN
Q3	0	0	0.098	NaN
Q4	0	0	0.400	NaN
Q5	0	0	0.000	NaN
Q6	0	0	0.000	81598.689
Q7	0	0	0.000	NaN
Q8	0	0	0.000	NaN
Q9	0	0	0.140	NaN
Q10	0	0	0.000	NaN
Q11	0	0	0.003	NaN
Q12	0	0	0.267	NaN
Q13	0	0	0.400	NaN
Q14	0	0	0.000	42
Q15	0	0	0.000	NaN
Q16	0	0	0.008	NaN
Q17	1	0	0.000	NaN
Q18	0	0	0.000	NaN
Q19	0	0	0.000	NaN
Q20	0	0	0.000	NaN
Q21	1	1	1.000	0
Q22	1	1	1.000	0

Gemini RAG

Query	Exact Match	Numeric Match	Token-level F1	MAE
Q1	0	0	0.068	NaN
Q2	0	0	0.000	NaN
Q3	0	0	0.000	NaN
Q4	0	0	0.444	NaN
Q5	0	0	0.000	NaN
Q6	0	0	0.000	NaN
Q7	1	0	0.000	NaN
Q8	1	0	0.000	NaN
Q9	0	0	0.000	NaN
Q10	0	0	0.000	NaN
Q11	0	0	0.000	NaN
Q12	0	0	0.000	NaN
Q13	0	0	0.000	NaN
Q14	0	1	0.000	0
Q15	0	0	0.000	NaN
Q16	0	0	0.000	NaN
Q17	1	0	0.000	NaN
Q18	0	0	0.000	NaN
Q19	1	0	0.000	NaN
Q20	0	0	0.000	NaN
Q21	1	1	1.000	0
Q22	1	1	1.000	0

DeepSeek RAG

Query	Exact Match	Numeric Match	Token-level F1	MAE
Q1	0	0	0.093	NaN
Q2	0	0	0.000	NaN
Q3	0	0	0.039	NaN
Q4	0	0	0.500	NaN
Q5	0	0	0.235	NaN
Q6	0	0	0.000	NaN
Q7	1	0	0.000	NaN
Q8	1	0	0.000	NaN
Q9	0	0	0.076	NaN
Q10	0	0	0.000	NaN
Q11	0	0	0.000	NaN
Q12	0	0	0.267	NaN
Q13	0	0	0.400	NaN

Query	Exact Match	Numeric Match	Token-level F1	MAE
Q14	0	0	0.000	NaN
Q15	0	0	0.000	NaN
Q16	0	0	0.000	NaN
Q17	1	0	0.000	NaN
Q18	1	0	0.000	NaN
Q19	1	0	0.000	NaN
Q20	0	0	0.000	NaN
Q21	0	0	0.000	1
Q22	0	0	0.000	NaN

8.3 Repositories

TPC-H benchmark

TPC-C benchmark

Basic Text-to-SQL TPC-H evaluation

Basic Text-to-SQL TPC-C evaluation

Context-Aware SQL Generation results for TPC-C and TPC-H

Fine-tuned models

Rag models

Bibliography

[1] D. D. Chamberlin, "Early history of SQL," *IEEE Annals of the History of Computing*, vol. 34, no. 4, pp. 78–82, 2012.

[2] T. Taipalus, "The effects of database complexity on SQL query formulation," *Journal of Systems and Software*, vol. 165, p. 110576, 2020, doi: https://doi.org/10.1016/j.jss.2020.110576.

[3] A. Mohammadjafari, A. S. Maida, and R. Gottumukkala, "From natural language to sql: Review of llm-based text-to-sql systems," *arXiv preprint arXiv:2410.01066*, 2024.

[4] S. S. Conn, "OLTP and OLAP data integration: a review of feasible implementation methods and architectures for real time data analysis," in *Proceedings. IEEE SoutheastCon, 2005.*, 2005, pp. 515–520.

[5] W. X. Zhao *et al.*, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, vol. 1, no. 2, 2023.

[6] Z. Hong *et al.*, "Next-Generation Database Interfaces: a Survey of LLM-based Text-to-SQL," *IEEE Transactions on Knowledge and Data Engineering*, no. , pp. 1–20, 2025, doi: 10.1109/TKDE.2025.3609486.

[7] M. Arslan, H. Ghanem, S. Munawar, and C. Cruz, "A Survey on RAG with LLMs," *Procedia Computer Science*, vol. 246, pp. 3781–3790, 2024, doi: https://doi.org/10.1016/j.procs.2024.09.178.

[8] X. Zhu, Q. Li, L. Cui, and Y. Liu, "Large Language Model Enhanced Text-to-SQL Generation: A Survey." [Online]. Available at: https://arxiv.org/abs/2410.06011

[9] M. Pourreza and D. Rafiei, "Din-sql: Decomposed in-context learning of text-to-sql with self-correction," *Advances in Neural Information Processing Systems*, vol. 36, pp. 36339–36348, 2023, [Online]. Available at: https://arxiv.org/abs/2304.11015

[10] J. Wei *et al.*, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models." [Online]. Available at: https://arxiv.org/abs/2201.11903

[11] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large Language Models are Zero-Shot Reasoners." [Online]. Available at: https://arxiv.org/abs/2205.11916

[12] H. Zhang, R. Cao, L. Chen, H. Xu, and K. Yu, "ACT-SQL: In-Context Learning for Text-to-SQL with Automatically-Generated Chain-of-Thought." [Online]. Available at: https://arxiv.org/abs/2310.17342

[13] N. Jin, J. Siebert, D. Li, and Q. Chen, "A Survey on Table Question Answering: Recent Advances." [Online]. Available at: https://arxiv.org/abs/2207.05270

[14] D. Min *et al.*, "Exploring the Impact of Table-to-Text Methods on Augmenting LLM-based Question Answering with Domain Hybrid Data." [Online]. Available at: https://arxiv.org/abs/2402.12869

[15] S.-A. Chen *et al.*, "Tablerag: Million-token table understanding with language models," *Advances in Neural Information Processing Systems*, vol. 37, pp. 74899–74921, 2024.

[16] P. Li *et al.*, “Table-gpt: Table fine-tuned gpt for diverse table tasks,” *Proceedings of the ACM on Management of Data*, vol. 2, no. 3, pp. 1–28, 2024.

[17] S. Zhang, A. T. Luu, and C. Zhao, “SynTQA: Synergistic Table-based Question Answering via Mixture of Text-to-SQL and E2E TQA.” [Online]. Available at: https://arxiv.org/abs/2409.16682

[18] N. Abhyankar, V. Gupta, D. Roth, and C. K. Reddy, “H-star: Llm-driven hybrid sql-text adaptive reasoning on tables,” *arXiv preprint arXiv:2407.05952*, 2024.

[19] G. K. Anna Mitsopoulou, “Analysis of Text-to-SQL Benchmarks: Limitations, Challenges and Opportunities,” *Proceedings of the 28th International Conference on Extending Database Technology (EDBT)*, 2025.

[20] F. Lei *et al.*, “Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows,” *arXiv preprint arXiv:2411.07763*, 2024.

[21] J. Li *et al.*, “Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls,” *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[22] L. C. Y. L. Xiaohu Zhu Qian Li, “Large Language Model Enhanced Text-to-SQL Generation: A Survey,” *arXiv:2410.06011v1 [cs.DB] 08 Oct 2024*, 2024.

[23] F. Liu, Y. Deng, Y. Xu, P. Lu, Y. Su, and Y. Cao, “Tree Edit Distance Guided Structural Evaluation for Text-to-SQL.” [Online]. Available at: https://openreview.net/forum?id=sgBso3EbW5

[24] J. Bulian, C. Buck, W. Gajewski, B. Börschinger, and T. Schuster, “Tomayto, tomahto. beyond token-level answer equivalence for question answering evaluation,” *arXiv preprint arXiv:2202.07654*, 2022.

[25] C. J. Willmott and K. Matsuura, “Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance,” *Climate research*, vol. 30, no. 1, pp. 79–82, 2005.

[26] F. Raab, W. Kohler, and A. Shah, “Overview of the TPC-C Benchmark: The Order-Entry Benchmark.” [Online]. Available at: https://www.tpc.org/tpcc/detail5.asp

[27] T. P. P. Council, “TPC-H Benchmark Homepage.” [Online]. Available at: https://www.tpc.org/tpch/