[Experiments & Analysis] Evaluating SQL Understanding in Large Language Models

Ananya Rahaman
University of Western Ontario
London, Ontario
arahaman@uwo.ca

Anny Zheng
University of Western Ontario
London, Ontario
azheng45@uwo.ca

Mostafa Milani
University of Western Ontario
London, Ontario
mostafa.milani@uwo.ca

Fei Chiang
McMaster Unviersity
Hamilton, Ontario
fchiang@mcmaster.ca

Rachel Pottinger
University of British Columbia
Vancouver, British Columbia
rap@cs.ubc.ca

ABSTRACT

The rise of large language models (LLMs) has significantly impacted various domains, including natural language processing (NLP) and image generation, by making complex computational tasks more accessible. While LLMs demonstrate impressive generative capabilities, there is an ongoing debate about their level of "understanding," particularly in structured domains like SQL. In this paper, we evaluate the extent to which LLMs "understand" SQL by testing them on a series of key SQL tasks. These tasks, such as syntax error detection, missing token identification, query performance prediction, query equivalence checking, and query explanation, assess the models' proficiency in recognition, context awareness, semantics, and coherence—skills essential for SQL understanding. We generate labeled datasets from well-known workloads, and evaluate the latest LLMs, focusing on how query complexity and syntactic features influence performance. Our results indicate that while GPT4 excels at tasks requiring recognition and context, all models struggle with deeper semantic understanding and coherence, especially in query equivalence and performance estimation, revealing the limitations of current LLMs in achieving full SQL comprehension.

1 INTRODUCTION

The rise of LLMs is having a significant impact across all domains, making computational and data science tasks more accessible and efficient. For example, in areas such as NLP and image generation, LLMs are able to generate human-like text and realistic images. While LLMs clearly do not have the same level of "understanding" as humans, their ability to solve problems (for which they are not directly trained) has alluded to some degree of "understanding" [15, 26, 31]. Thus, for LLMs, "understanding" refers to the model's ability to perform fundamental tasks at least as proficiently as humans, and potentially even better, across different contexts.

This level of proficiency can be measured against a set of characteristic *skills* to assess understanding. *Recognition* involves identifying the intended object/entity of interest, e.g., identifying and differentiating between objects and scenes in image generation. *Semantics* involves identifying how meaning is constructed and interpreted, e.g., the meaning of a red octagon is to stop, grasping the meaning of words and phrases. *Context* defines the scope and setting in which the semantics are interpreted, e.g., in NLP, resolving ambiguities when there exist multiple meanings based

on context, comprehending intent (understanding the purpose behind a speaker's words, such as detecting sarcasm or politeness), and handling out-of-distribution elements (identifying when an object or scenario doesn't fit familiar patterns). Lastly, *coherence* identifies the logical interconnection between objects, e.g., object coherence ensures that objects are placed in realistic positions relative to each other in image generation and identifies the logical links between words, sentences, and paragraphs sharing the same meaning. Achieving "understanding" requires models to demonstrate (increasing) proficiency in these skills and to complete task-specific operations accurately and meaningfully. Developing a deeper insight into LLM's "understanding" is crucial for reliable performance in real-world applications where accuracy is essential.

Toward this goal, we study how LLMs can be used in data management applications, particularly, their ability to perform SQL-related tasks. Our study goes beyond simply content generation; it evaluates specific SQL tasks that exhibit the aforementioned skills. We pose the question: *How well do LLMs "understand" SQL?*

SQL Tasks. We propose a series of core SQL tasks designed to probe the depth of LLMs' SQL "understanding". Novice to advanced SQL users perform tasks ranging from syntax error identification to query performance estimation to query equivalence and explanation. We evaluate LLMs' ability to perform such tasks, in increasing order of difficulty to reflect increasing skill proficiency.

Syntax error identification. Detecting advanced syntax errors that violate structural and semantic requirements vs. basic errors (e.g., missing parentheses) reflect varying levels of SQL "understanding". For example, detecting the misalignment of attributes, aggregation functions among SELECT, GROUP BY, HAVING clauses, incompatible attribute types between outer and inner queries, and invalid join operations require a complex "understanding" of the queries.

Missing token identification. Identifying missing tokens is a crucial pre-step for applications such as query recommendation, where missing token imputation and query auto-completion are key functionalities [14, 39]. We evaluate the ability to not only recognize a missing token but to identify the precise location and the type of missing token (e.g., missing keywords (e.g., SELECT or WHERE), table names, aliases used in joins or conditions, or literal values.

Query performance estimation. Given only the SQL query text, accurately estimating its runtime performance is challenging, as multiple factors such as the database schema, specific data instances,

Skill	syntax error	missing token	Q.perf. estimate	Q.equiv.	Q.explain.
Recognition	✓	✓			
Semantics				✓	✓
Context		✓	✓		✓
Coherence	✓		✓	✓	

Table 1: Skill-to-SQL task mapping

and the query workload all play a role [39]. Using publicly available query workloads, recent work has shown that more complex, longer queries with multiple joins and multiple predicate conditions incur higher execution costs [10, 16, 35]. We evaluate LLM "understanding" of query complexity for performance estimation, going beyond surface-level syntax.

Query equivalence. Two syntactically different queries are equivalent if they return the same result for all database instances. This is important for query optimization [4, 13], and query recommendation [39], where simpler query representations facilitate faster execution times. We evaluate query equivalence using labeled, equivalent (positive), and non-equivalent (negative) query pairs. While generating equivalent pairs is subtle, the negative case requires careful consideration. If we pair random, non-equivalent queries and label them as such, then the task becomes overly simplistic, as superficial differences will often identify non-equivalence, without testing the model's ability to understand deeper query semantics.

Query explainability. We evaluate LLMs to explain SQL queries by describing the query output. This task is similar to assessments in code and image understanding, to generate code documentation [22] and image captions, respectively, to measure understanding. We evaluate over a wide range of complex queries, including multiple tables, nested subqueries, and intricate logical conditions.

Skill to task proficiency. Each SQL task is associated to a set of skills to assess (SQL) understanding, as summarized in Table 1. In syntax error identification, we assess recognition and coherence to identify syntactic violations, and ensure logical consistency between clauses, such as mismatches between aggregation functions and GROUP BY clauses. Missing token identification requires recognition and context, to detect missing tokens and to determine the correct type and location (e.g., keyword, table name, or value). Query performance estimation evaluates context and coherence, as the model considers the query complexity, the database schema, and the workload to estimate performance. In query equivalence, we assess the model's semantics and coherence abilities to interpret different syntactic query formats but representing the same functional output. Lastly, in query explanation, we evaluate the model's semantics and context to describe the query's purpose, and it's result within the context of the schema and the data.

Paper Contributions. We present an experimental study evaluating the performance of the major LLMs over core SQL tasks.

SQL task-driven data benchmark. Many of these tasks require labeled data, which we generate by modifying raw queries from popular SQL workloads, such as the Sloan Digital Sky Survey (SDSS) [35], SQLShare [10], and Join Order [16]. For syntax error and missing token identification tasks, we create semi-synthetic datasets by randomly selecting queries from workloads and injecting errors or by removing tokens. For each task, we select an appropriate type, such as the type of syntax error to inject or the type of missing token

(e.g., keyword, table name, column name). For query performance tasks, we rely on the SDSS workload, which includes log information from past query evaluations. We classify queries based on their runtime, where high runtime represents computationally expensive queries. For query equivalence tasks, we manually modify selected queries to generate equivalent and non-equivalent pairs, ensuring that the modifications reflect realistic query transformations, such as rewriting nested queries using joins. Our SQL task-driven data benchmark is publicly available.[1]

Prompt-to-SQL task performance. Prompt tunning is key in ensuring consistent results from LLMs. We experiment with various prompts, testing them in small-scale trials using a subset of labeled data to identify the best prompt per task. However, interaction with LLMs goes beyond prompt design. Processing their responses is complex, and in our work, we addressed this by using a combination of automated scripts and manual checks to extract the labels.

SQL task evaluation framework. We systematically evaluate the factors influencing LLM performance across SQL tasks. Our evaluation framework considers three key dimensions. First, we compare SQL task performance across different LLMs. Second, we analyze the properties of the query workloads, particularly the syntactic complexity of the SQL queries, such as the number of tables, conditions, nested subqueries, and overall query length. We investigate how these syntactic properties affect the LLMs' ability to process and understand queries. Finally, we evaluate the performance of specific SQL tasks across varying parameters, e.g., how different types of missing tokens or query transformations affect LLM performance, and whether certain forms of query equivalence or error detection are more challenging to recognize. By considering these three dimensions: LLM performance comparison, workload properties, and task types, we aim to provide a comprehensive evaluation of the factors that influence LLM performance in SQL tasks.

Extensive comparative evaluation. Our experiments show that GPT4 performs best across most tasks, while no other model consistently ranks second. Although most models demonstrate strong performance in binary class tasks such as identifying syntax errors or missing tokens, all LLMs face challenges and suffer reduced accuracy in multi-class tasks, such as identifying the type of missing token or syntax error. LLMs generally struggle with longer and more complex queries, particularly involving logical reasoning or numerical computations, consistent with prior results [5, 8, 33].

Our experimental results demonstrate that LLMs perform well at tasks requiring recognition and context, such as syntax error detection and missing token identification. However, for tasks requiring coherence and semantic understanding, such as query equivalence and performance estimation, the models exhibit limitations. This suggests that while LLMs demonstrate proficiency at surface-level "understanding", they struggle to fully comprehend deeper semantic relationships, and logical coherence in SQL queries, underscoring the need for further improvements.

Paper Organization. In Section 2, we describe the workloads used in the study, including comprehensive statistics on the syntactic properties of the queries in our datasets. Section 3 outlines the experimental setup, covering the SQL tasks, data generation from the workloads, the LLMs, and our interaction with the LLMs, such

---

[1]https://github.com/AnanyaRahaman/LLMs_SQL_Understading

Workload	Number of Queries		Query Type		Aggregate		NestLvl	
	Original	Sampled	SELECT	CREATE	Yes	No	0	1
SDSS	5,081,188	285	Fig 1a		21	264	Fig 1e	
SQLShare	9,623	250	Fig 2a		59	192	Fig 2e	
Join-Order	157	157	113	44	119	38	-	-
Spider	4, 486	200	200	0	96	104	185	15

Table 2: Workload statistics overview

as prompt tuning. Section 4 presents the experimental results and analysis, Section 5 reviews related work, and we conclude the paper and discusses directions for future work in Section 6.

2 QUERY WORKLOADS

A query workload, or simply a workload, is a collection of SQL queries executed against a database, used to simulate real-world usage patterns for performance evaluation and optimization. We give an overview and detailed analysis of the four workloads used in our experimental study.

The Sloan Digital Sky Survey (SDSS) dataset [35]. SDSS consists of a relational database with data from a major astronomical survey providing detailed images and spectra of the sky and a SQL query workload used to interact with the SDSS database. The SDSS workload is characterized by its complexity and the need for precise astronomical data retrieval. The workload has been collected over two decades, containing millions of queries. In our study, we use queries recorded in 2023.

SQLShare [10]. SQLShare is an open data platform designed to make data sharing and querying more accessible. The SQLShare workload consists of a diverse set of user-generated SQL queries, ranging from simple data retrieval to complex data manipulation tasks. Unlike our other workloads, SQLShare consists of query statements over several databases with different schemas.

Join-Order [16]. The Join-Order Benchmark is a synthetic workload designed to evaluate the performance of database systems in optimizing join queries. The benchmark includes complex SQL queries to test the optimizer's ability to find efficient join orders.

Spider [36]. Spider is a large-scale, complex, cross-domain Text-to-SQL benchmark used to evaluate a model's natural language understanding, and SQL generation capabilities. It includes a wide range of databases, to evaluate generalization across different database schemas. Spider is used extensively in NLP to benchmark model performance to translate natural language queries to SQL. We use the Spider dataset exclusively for query explanation, while the other three workloads are used for the remaining tasks.

All our workloads, with the exception of Join-Order, contain a large number of queries; making it impractical to utilize all queries in our study. Therefore, we created smaller datasets by sampling a limited number of queries. Table 2 shows the "original" number of queries in the workloads, and "sampled" shows the sampled number of queries we use in our experiments. We describe the dataset generation process in Section 3.2. Next, we examine the syntactic properties of our sampled queries, to provide context to interpret our experimental results. Henceforth, we will use SDSS, SQLShare, Join-Order, and Spider to refer to the datasets created from the sampled queries of the original workloads.

2.1 Syntactic Properties of SQL Queries

For each SQL query, we assess the following properties:
- `char_count` and `word_count`, respectively, refer to the number of characters and the number of words in the query.
- `query_type` refers to the type of the query, e.g., SELECT, UPDATE, and CREATE.
- `table_count` and `join_count` refer to the number of distinct tables referenced in the query and the total number of joins, respectively. Joins include both explicit joins (using join keywords such as INNER JOIN) and implicit joins (tables in the FROM clause with join conditions).
- `column_count` refers the number of distinct columns used or referenced in the SELECT clause of the query.
- `function_count` refers to the total number of functions in the query, including built-in (like min, avg) and user-defined functions. `predicate_count` is the number of conditions specified in the WHERE clause.
- `nestedness` is the nested depth of subqueries within the query.
- `aggregate` refers to whether the query uses aggregate functions.

Table 2 provides a statistical overview of all four workloads, including the number of SELECT and CREATE queries and a breakdown of aggregate vs. simple queries. Figures 1-3 illustrate additional properties. Each figure is a histogram showing query counts on the y-axis and query properties on the x-axis, where the x-values represent a range of properties. For example, Figure 1b shows the number of queries (y-axis) across different ranges of query lengths (`word_count`). The figures highlight that SDSS and SQLShare contain more complex queries, with multiple tables and a wider variety of predicates. In contrast, Join-Order has simpler, less nested queries. For query length (`word_count`), SDSS and Join-Order have longer queries compared to SQLShare.

As pairs of properties may exhibit strong correlations, leading to redundancy and inefficiency, we examine the correlations between pairwise query properties using Pearson coefficients [24], and we use a threshold of 0.7 to indicate strong correlation. Figure 4 show the following observations:
- `char_count` and `word_count` are highly correlated, as longer queries generally contain more words.
- `table_count` and `join_count` are also highly correlated since queries with more tables usually involve more joins, a common pattern in multi-table SQL queries.

We consider correlations unique to specific workloads:
- In SDSS, `column_count` and `char_count` are strongly correlated as longer queries often involve selecting more columns or adding conditions. Additionally, `nestedness` and `join_count` are correlated, as deeply nested queries tend to include multiple joins.
- SQLShare and Join-Order exhibit a high correlation between `function_count` and `predicate_count` due to the frequent use of functions in conditions.
- In Join-Order, `char_count`, and `word_count` are correlated with `table_count` and `join_count`, indicating longer queries involve more joins and tables.

3 EXPERIMENTAL SETUP

We introduce our SQL tasks in Section 3.1, and our data preparation steps to inject errors, missing tokens, and derive equivalent and
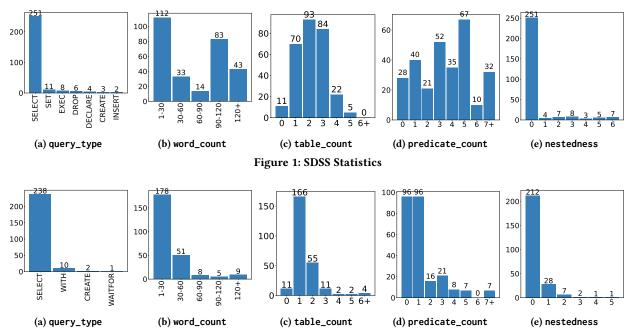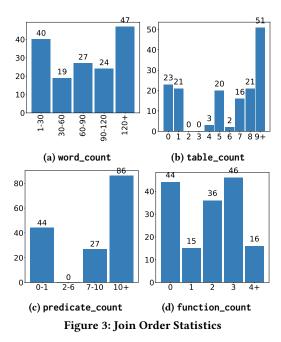
Figure 1: SDSS Statistics

Figure 2: SQLShare Statistics

Figure 3: Join Order Statistics

non-equivalent queries in Section 3.2. We then give an overview of the evaluated LLMs (Section 3.3), and how we prompt and respond to the LLMs in Section 3.4.

3.1 SQL Tasks

3.1.1 Binary Tasks. We begin with binary classification tasks that identify syntactic errors, missing tokens, and query equivalence.

syntax_error. We evaluate the LLM's ability to identify the presence of a syntax error. We study six types of errors as descrbed below. Listing 1 shows sample errors for each type.

- `aggr-attr`. Aggregate functions are used without properly grouping non-aggregated columns.
- `aggr-having`. Misusing the HAVING clause to filter non-aggregated columns instead of using WHERE.
- `nested-mismatch`. The inner query in a nested query returns multiple rows, which is not correctly handled in the outer query.
- `condition-mismatch`. Operations with incompatible data types, e.g., comparing numeric columns to strings.
- `alias-undefined`. An alias is used in a query but is not defined.
- `alias-ambiguous`. The same column appears in multiple tables, but its usage in a query does not specify the table reference.

```sql
-- Q1: Aggregation without GROUP BY (aggr-attr)
SELECT plate,mjd,COUNT(*), AVG(z)
FROM SpecObj WHERE z > 0.5;
-- Q2: Incorrect Use of HAVING (aggr-having)
SELECT plate,COUNT(*) AS NumSpectra
FROM SpecObj GROUP BY plate HAVING z > 0.5;
-- Q3: Type mismatch in subquery (nested-mismatch)
SELECT p.ra,p.dec,s.z
FROM PhotoObj AS p JOIN SpecObj AS s
ON s.bestobjid = (SELECT bestobjid FROM SpecObj);
-- Q4: Type mismatch in condition (condition-mismatch)
SELECT plate,mjd,fiberid FROM SpecObj WHERE z = 'high';
-- Q5: Undefined alias (alias-undefined)
SELECT s.plate,s.mjd,z
FROM SpecObj AS s JOIN PhotoObj AS p
ON s.bestobjid = photoobj.bestobjid;
-- Q6: Ambiguous alias (alias-ambiguous)
SELECT plate,fid FROM SpecObj AS s JOIN PhotoObj AS p
ON s.bestobjid = p.bestobjid WHERE bestobjid > 1000;
```
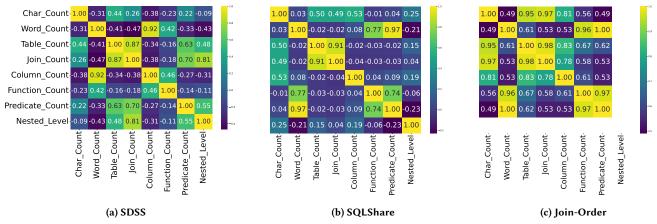
Listing 1: SQL syntax error examples

Figure 4: Pairwise correlations between query properties for each workload

miss_token. We probe the LLM to determine whether a SQL query is missing any tokens. Although this can be seen as a type of syntax_error, we consider it separately due to its importance in applications. We consider six types of tokens: keyword, table, column, value, alias, and predicate (comparisons).

query_equiv. Determines whether two SQL queries are equivalent, i.e., whether they have the same schema and produce the same results. We study ten types of equivalences and eight types of non-equivalences. Listings 2 shows a few examples of these types using the SDSS workload. For the full list of equivalence and non-equivalence types, along with detailed explanations and examples, we refer the reader to our GitHub repository.

- swap-subqueries. Swapping inner and outer sub-queries in nested queries.
- join-nested. Converting a join into a subquery or vice versa.
- cte. Rewriting a query using common table expressions (CTEs), a temporary result set defined using WITH, which simplifies complex queries and is referenced within the main query.
- reorder-conditions. Re-arranging the order of conditions in a WHERE clause.
  We study four types of non-equivalent transformations:
- agg-function. Modifying an aggregate function, e.g., updating to SUM from AVG.
- change-join-condition. Modifying the type of join, such as switching from an INNER JOIN to a LEFT JOIN.
- logical-conditions. Altering logical operators, such as changing AND to OR.
- value-change. Updating a filtering condition, e.g., altering the comparison value.

```
-- Q7: swap-subqueries (Equivalent)
SELECT s.plate,s.mjd FROM SpecObj AS s WHERE s.plate IN
    (SELECT p.plate FROM PhotoObj AS p WHERE p.ra > 180);
-- Equivalent Query:
SELECT p.plate,p.mjd FROM PhotoObj AS p
WHERE p.ra > 180 AND p.plate IN
    (SELECT s.plate FROM SpecObj AS s);
-- Q8: join-nested (Equivalent)
SELECT s.fiberid FROM SpecObj AS s JOIN PhotoObj AS p
ON s.bestobjid = p.objid WHERE p.ra > 180;
-- Equivalent Query:
SELECT fiberid FROM SpecObj WHERE bestobjid IN
```

```
    (SELECT objid FROM PhotoObj WHERE ra > 180);
-- Q9: cte (Equivalent)
SELECT plate,mjd FROM SpecObj WHERE z > 0.5;
-- Equivalent Query:
WITH HighRedshift AS
    (SELECT plate,mjd FROM SpecObj WHERE z > 0.5)
SELECT plate,mjd FROM HighRedshift;
-- Q10: reorder-conditions (Equivalent)
SELECT * FROM SpecObj WHERE plate = 1000 AND mjd > 55000;
-- Equivalent Query:
SELECT * FROM SpecObj WHERE mjd > 55000 AND plate = 1000;
-- Q11: agg-function (Non-Equivalent)
SELECT plate,AVG(z) FROM SpecObj GROUP BY plate;
-- Non-Equivalent Query:
SELECT plate,SUM(z) FROM SpecObj GROUP BY plate;
-- Q12: change-join-condition (Non-Equivalent)
SELECT s.plate,s.mjd FROM SpecObj AS s
JOIN PhotoObj AS p ON s.bestobjid = p.objid;
-- Non-Equivalent Query:
SELECT s.plate,s.mjd FROM SpecObj AS s
LEFT JOIN PhotoObj AS p ON s.bestobjid = p.objid;
-- Q13: logical-conditions (Non-Equivalent)
SELECT plate,mjd,fiberid
FROM SpecObj WHERE z > 0.5 AND ra > 180;
-- Non-Equivalent Query:
SELECT plate,mjd,fiberid
FROM SpecObj WHERE z > 0.5 OR ra > 180;
-- Q14: value-change (Non-Equivalent)
SELECT plate, mjd, fiberid FROM SpecObj WHERE z > 0.5;
-- Non-Equivalent Query:
SELECT plate,mjd,fiberid FROM SpecObj WHERE z > 5;
```

Listing 2: Examples of SQL equivalence and non-equivalence

performance_pred. We evaluate the model's ability to predict query runtime performance. Only the SDSS workload contains ground truth query execution times. Figure 5 shows a clear separation between short running (low-cost) vs. long running queries (costly), which we pose as a binary classification task, and consider costly queries as the positive class.

3.1.2 Multi-class Tasks. We extend the binary tasks towards multi-class tasks by probing LLMs to indicate the *type* of syntax error (**syntax_error**), type of missing token (**miss_token_type**), and type of query equivalence (**query_equiv_type**). We also evaluate the task of identifying a missing token's location (**miss_token_loc**).
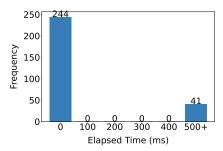
Figure 5: Elapsed time of sampled SDSS queries

3.1.3 Query explanation. This task (`query_exp`) explains what a SQL query does. It is the reverse of the text-to-SQL task, where existing benchmarks for text-to-SQL, such as WikiSQL [38], provide natural language descriptions of queries. However, many of these benchmarks contain relatively simple SQL queries compared to the more complex workloads in SDSS and SQLShare. Thus, we chose the Spider dataset [36] that includes more complex queries, and we further sampled longer and more complex queries.

Our analysis is qualitative rather than quantitative. We manually review the LLM generated explanations, and compare them with the ground truth descriptions provided in the workload. Our goal is to analyze and discuss when and why models fail to provide accurate, meaningful explanations (Section 4.5). While this task does not strictly require existing explanations, we use Spider's explanations to help with validation, and to streamline the evaluation process.

3.2 Data Preparation and Label Generation

We describe how task-specific labels are generated for each dataset.

`syntax_error` and `miss_token`. We generate semi-synthetic datasets by randomly selecting queries from each workload and injecting errors. For `syntax_error`, we randomly select the type of error to inject, including cases with no error (error-free). We create binary labels indicating the presence (or not) of an error, and the error type for the multi-class (`syntax_error_type`) task. We follow a similar method for `miss_token`, where we randomly removed a specific token type. We use the SDSS, SQLShare, and Join-Order workloads for these tasks.

`query_equiv`. We use the SDSS, SQLShare, and Join-Order workloads to generate equivalent and non-equivalent pairs. We randomly select the type of equivalence, and the queries to modify. Generating non-equivalent pairs is non-trivial as we must balance sufficient similarity between the queries (to make the task challenging) while including functional differentiation. Each query pair thus has a label (equivalent/non-equivalent), and a type of equivalence.

`performance_pred`. For this task, we used only SDSS, which includes runtime data. We randomly selected 285 queries and analyzed their runtime to classify each as either high or low runtime, serving as a proxy for computational expense. Queries running longer than 200 ms are high cost, otherwise, they are low cost. We select this threshold based on our observations from Figure 5.

3.3 Large Language Models

We use several state-of-the-art LLMs, briefly described below.

GPT3.5. Released by OpenAI in late 2022, GPT3.5 consists of 175 billion parameters and is trained on a large corpus including Common Crawl, Wikipedia, and various books and academic texts. It is designed to handle diverse NLP tasks [3].

GPT4. OpenAI's GPT4, launched in 2023, boasts over 200 billion parameters, with significant improvements in contextual understanding and reasoning over its predecessor. It was trained on a larger and more diverse dataset, enhancing its performance across a variety of language tasks [23].

Gemini. Developed by Google and introduced in 2024, Gemini prioritizes both accuracy and safety in AI interactions, with a strong emphasis on ethical AI. It has an estimated parameter count of 50 billion and is trained on vast multimodal datasets, handling both text and visual data. This focus on multimodal capabilities and alignment with human values differentiates it from models primarily designed for text-based tasks [1].

Llama3. Meta's Llama3, released in 2023, is available in versions up to 70 billion parameters. Trained on trillions of tokens from general-purpose datasets, it is designed for efficiency and scalability. LLaMA's focus on broad, general-purpose language tasks makes it ideal for deployment in resource-constrained environments, where maintaining performance is crucial [30].

MistralAI. Launched in 2024, MistralAI is optimized for high accuracy with a smaller footprint of 16 billion parameters. It is trained on a wide range of datasets, with an emphasis on domain-specific content and multilingual capabilities. MistralAI is particularly suited for specialized tasks, that require deeper understanding in areas such as SQL and other structured data, offering a balance of computational efficiency and domain-targeted performance [21].

LLMs achieve high performance due to their extensive training on datasets ranging from hundreds of billions to trillions of tokens. The trend in LLM development is to leverage larger datasets and more complex architectures to continually improve generalization across diverse tasks [3, 21, 23, 30].

3.4 Refining LLM Interactions

Interacting with LLMs requires careful attention to both input prompts and processing of their responses. By tuning prompts, we guide the models toward generating more accurate and relevant outputs. However, the responses also require post-processing because they are often not provided in a straightforward format necessary for our tasks, such as a simple label. Post-processing involves extracting the necessary information from potentially verbose or complex responses, and ensuring that it fits the specific format required for evaluation.

Prompt Tuning. Designing and refining input prompts to guide LLMs toward accurate responses is particularly important for complex tasks, where well-crafted prompts can significantly improve model performance [20, 27, 34]. In our study, prompt tuning is essential to effectively handle the intricacies of SQL syntax and semantics. Our tuning process involved two key steps:

(1) *Prompt Generation and Refinement.* We used LLMs to generate a variety of prompt candidates, which were then manually refined to ensure clarity and alignment with our task objectives [2, 32].

(2) *Mock Experiments.* We conducted mock experiments with a subset of data to evaluate the effectiveness of each prompt. The top-performing prompts from these tests were selected for full-scale experiments.

Following this approach, we developed a set of task-specific prompts to extract meaningful responses from the models. These prompts were tailored to each experimental task and varied in complexity, addressing challenges such as syntax error detection, query equivalence, and runtime estimation:

- `syntax_error` and `syntax_error_type`. Does the following query contain any syntax errors? If so, explain the error. [query]
- `miss_token`, `miss_token_type`, and `miss_token_loc`. Does the following query have any syntax errors? (yes/no) If yes, is there a missing word? (yes/no) If yes, what is the type of the missing word? If yes, what is the missing word? If yes, what is the position of the missing word? (Provide the word count position where the word is missing.) [query]
- `query_equiv` and `query_equiv_type`. Are the following two queries equivalent (do they produce the same results on the same database schema)? If yes, why are they equivalent? [query 1, query 2]
- `performance_pred`. Does the following query take longer than usual to run? [query]
- `query_exp`. Provide a single statement describing this query: [query].

The prompts listed above reflect the outcomes of our prompt tuning approach, which was specifically designed to address the SQL tasks in our study.

Handling LLM Output. Despite providing clear instructions, it is necessary to post-process the LLM output results. LLMs often produce lengthy and verbose responses that require careful extraction of relevant information. For example, in the prediction task described above, while most LLMs respond with a binary "yes" or "no," they often provide explanations about why the query takes a long or short time to execute. Similarly, in the `miss_token` task, the responses are not always formatted in a structure that aligns with our evaluation criteria. To address this, we rely on a combination of manual processing and automated scripts. Manual processing involves reading through the responses to extract the specific information required, e.g., isolating the "yes" or "no" from the rest of the explanation. To expedite this, we use scripts to detect common response patterns and automatically extract the relevant portions when the responses follow predictable structures. However, for more complex or less structured outputs, manual intervention is still necessary to ensure accuracy. This allows us to format the LLM outputs consistently, and to evaluate their performance effectively.

Zero-Shot, Few-Shot, and Fine-Tuning. Zero-shot learning refers to a model's ability to perform a task without seeing any specific examples, relying solely on its pre-trained knowledge. This approach is valuable for evaluating the model's inherent understanding of a domain. In our experiments, we focused exclusively on zero-shot learning to assess a model's ability to detect syntax errors, evaluate query equivalence, and predict query runtime costs. Our goal was to study the models in their raw form, without introducing additional task-specific information, which reflect real-world scenarios where such data may not always be available.

Case	Model	SDSS			SQLShare			Join-Order		
		Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
Syntax Error	GPT4	**0.98**	**0.95**	**0.97**	<u>0.94</u>	**0.93**	**0.93**	**0.95**	<u>0.91</u>	**0.93**
	GPT3.5	0.94	0.85	0.89	0.91	0.86	0.89	<u>0.93</u>	0.81	0.86
	Llama3	<u>0.95</u>	0.76	0.84	0.92	0.81	0.86	**0.95**	0.65	0.77
	MistralAI	0.93	<u>0.91</u>	<u>0.92</u>	0.92	<u>0.91</u>	<u>0.92</u>	0.85	**0.94**	<u>0.89</u>
	Gemini	0.94	0.70	0.80	**0.97**	0.53	0.68	0.84	0.61	0.70
Syn. Error Type	GPT4	**0.96**	**0.95**	**0.95**	**0.89**	**0.88**	**0.88**	**0.90**	**0.89**	**0.89**
	GPT3.5	0.87	0.85	0.85	<u>0.85</u>	<u>0.82</u>	<u>0.83</u>	0.83	0.78	0.78
	Llama3	0.83	0.79	0.79	0.79	0.76	0.76	0.78	0.67	0.64
	MistralAI	<u>0.90</u>	<u>0.88</u>	<u>0.89</u>	0.81	0.80	0.79	<u>0.86</u>	<u>0.81</u>	<u>0.82</u>
	Gemini	0.81	0.74	0.73	0.73	0.60	0.58	0.68	0.53	0.52

Table 3: Accuracy in `syntax_error` and `syntax_error_type`

Few-shot learning provides a model with few examples to improve performance, and fine-tuning further trains a model on task-specific datasets to improve accuracy. Although both approaches can help in situations where a model's initial performance is lacking, we did not use either method in our study. Our goal was to evaluate LLMs with minimal additional training to reflect their performance in environments where limited labeled data are available.

4 EXPERIMENTAL RESULTS

We present our results and analysis, with each subsection focusing on a primary SQL task, and its related secondary tasks.

Across all experiments, GPT4 consistently outperforms other models, with no clear runner-up in most cases. This dominance may be because of the larger model size, as we outlined in Section 3.3, and possibly the model being trained on a larger corpus of SQL queries. To avoid repetition, this general observation will not be restated in the individual result discussions.

4.1 Syntax Error Tasks

In this section, we present results for the two related tasks of `syntax_error` and `syntax_error_type`.

`syntax_error`. Table 3 (top) shows the comparative accuracy on the `syntax_error` task. The best-performing model is highlighted in bold, and the second-best is underlined. GPT4, GPT3.5, and MistralAI perform well, while Llama3 and Gemini struggle. This may be because Llama3 is trained on general-purpose text, and Gemini focuses more on AI ethics and multimodal tasks, meaning both have less specific knowledge of SQL compared to the other models.

Across all models, recall tends to be lower than precision, suggesting that the models are more conservative in detecting errors, missing some existing syntax errors (lower recall) but making fewer incorrect claims about errors (higher precision). One possible explanation is that these models may have been trained more extensively on correct SQL queries, with less exposure to syntactically incorrect examples. This precision-recall imbalance is particularly pronounced in Llama3 and Gemini, which exhibit significantly lower recall, resulting in reduced F1 scores as well.

An important question is when and why LLMs fail in `syntax_error`. To explore this, we examined two hypotheses: first, that failures are related to the syntactic properties of queries, such as `word_count` or `table_count`; and second, that they are linked to specific types

of syntax errors, such as `aggr-attr` or `nested-mismatch`, as discussed in Section 3.1. We applied the same analysis to other tasks while testing these hypotheses.

For the first hypothesis, we analyzed the distribution of syntactic properties across four categories: true positive (TP), true negative (TN), false positive (FP), and false negative (FN). Figure 6 illustrates this for `syntax_error` in SDSS, showing query distributions by `word_count`. The numbers below each category represent the average (top), median (middle), and total number of queries (bottom). Figures 6a and 6b show similar data for Llama3 and Gemini.
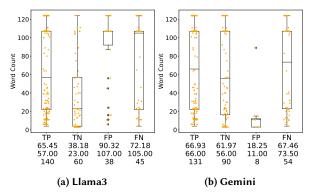

(a) Llama3 **(b) Gemini**

Figure 6: Relationship between `word_count` and model failure in `syntax_error` for SDSS. The three numbers (e.g., 65.45, 57.00, 140) represent the average and median query length and the number of queries in the category (TP). The orange scatter points represent queries plotted on the y-axis by their length, showing the length distribution per category.

To explore the correlation between query length (`word_count`) and model failure, we compared TP and FN (for queries with errors) as well as TN and FP (for queries without errors) while concluding when there are significant queries in each category. For example, in Figure 6a, the TP and FN categories have sufficient queries (140 and 45, respectively) to observe a pattern: the FP queries tend to be significantly longer (average 65.45 vs 72.18, median 57 vs 105). A similar trend is seen when comparing TN and FP, where FP queries are longer (average 38.18 vs 90.32, median 23 vs 107). This trend is also observed while comparing TP and FN in Gemini in Figure 6b, but there are not enough queries in FP to draw a conclusion for TN and FP. Overall, these findings suggest a correlation between query length (`word_count`) and failure in `syntax_error`, with longer queries being more prone to misclassification. We did not observe a similar pattern for any other syntactic properties across models or datasets, indicating that `word_count` is the most significant factor influencing failure likelihood in this task.

For the second hypothesis, Figure 7 presents the proportion of queries in FN for each type of syntax error, where a larger bar indicates that detecting errors of that type has been more challenging for the models. The results for SDSS (Figure 7a) suggest that type mismatch errors (`nested-mismatch` ■ and `condition-mismatch` ■) are particularly difficult for all models to detect. This is expected as the workload involves queries with many conditions for which the type of operands could be difficult to tell. For SQL-Share (Figure 7b), ambiguous alias (`alias-ambigous` ■) errors are

Case	Model	SDSS			SQLShare			Join-Order		
		Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
Missing Token	GPT4	**0.99**	**0.97**	**0.98**	**0.98**	**0.96**	**0.97**	**1.00**	**0.97**	**0.99**
	GPT3.5	0.92	0.92	0.92	<u>0.97</u>	0.88	<u>0.93</u>	<u>0.98</u>	<u>0.94</u>	0.96
	Llama3	<u>0.96</u>	<u>0.94</u>	<u>0.95</u>	0.91	<u>0.92</u>	0.91	0.97	<u>0.94</u>	0.96
	MistralAI	**0.99**	0.86	0.92	0.96	0.87	0.91	**1.00**	<u>0.94</u>	<u>0.97</u>
	Gemini	**0.99**	0.76	0.86	**0.98**	0.68	0.80	0.97	0.69	0.81
Token Type	GPT4	**0.94**	**0.94**	**0.94**	**0.91**	**0.89**	**0.90**	**0.98**	**0.97**	**0.98**
	GPT3.5	0.76	0.75	0.75	0.75	0.71	0.73	0.84	0.82	0.82
	Llama3	0.88	<u>0.85</u>	<u>0.86</u>	0.78	0.69	0.72	0.87	0.82	0.84
	MistralAI	<u>0.89</u>	<u>0.85</u>	<u>0.86</u>	<u>0.82</u>	<u>0.75</u>	<u>0.78</u>	<u>0.93</u>	<u>0.88</u>	<u>0.90</u>
	Gemini	0.63	0.63	0.54	0.75	0.53	0.57	0.44	0.60	0.39

Table 4: Accuracy for `miss_token` and `miss_token_type`

more problematic, which is expected given the large number of schemas and varied table aliases used in these queries. Lastly, in Join-Order (Figure 7c), errors involving mismatch in using nested queries (`nested-mimatch` ■) are the most frequently missed by the models since similar to SDSS the queries in Join-Order also have lengthy conditions in the `WHERE` clauses.

`syntax_error_type`. Table 3 (bottom) presents the weighted accuracy for `syntax_error_type`, which considers the six types of syntax errors (see Section 3.1). The strong performance of GPT4 and MistralAI, and GPT3.5 in detecting syntax errors also extends to identifying error types, while Llama3 and Gemini continue to perform less effectively, as expected. Overall, results for `syntax_error_type` are lower than for `syntax_error`, reflecting the increased difficulty of this task. Another key observation is that all models show lower performance on the SQLShare dataset, likely due to its more complex schema, which makes identifying the type of syntax errors more challenging.

<u>Takeaways:</u> The analysis of `syntax_error` and `syntax_error_type` shows that GPT4, MistralAI, and GPT3.5 outperform Llama3 and Gemini, likely due to differences in training focus. Longer queries are more prone to errors, and the types of syntax errors the models struggle with largely depend on the specific dataset.

4.2 Missing Token Tasks

Regarding missing token, we start by `miss_token`, and then present results related to `miss_token_type` and `miss_token_loc`.

`miss_token`. Table 4 (top) presents the accuracy of various LLMs in `miss_token`. Accuracy is higher compared to `syntax_error`, as `miss_token` is a simpler task. A notable change is Llama3 's improved performance in this task. This can be attributed to the fact that detecting missing tokens relies more on general pattern recognition, which is less specialized for SQL. Llama3 's broader training in recognizing patterns likely helps it improve in this context. Overall, recall remains lower than precision in `miss_token`, similar to `syntax_error`, likely because the models are more conservative in detecting errors, as explained previously.

We investigated the relationship between LLM failures in the `miss_token` task and the syntactic properties of queries. Figure 8a shows that, for GPT3.5 on the SQLShare dataset, query length (`word_count`) is correlated with failures, with an average `word_count`
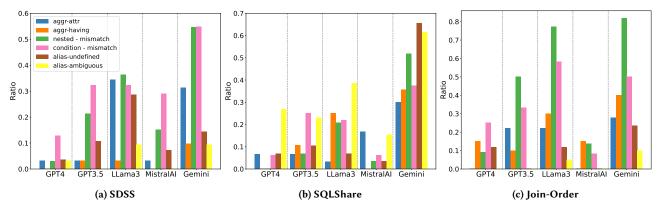
(a) SDSS   **(b) SQLShare**   **(c) Join-Order**

Figure 7: Relationship between syntax error type and FN in `syntax_error`.

(a) `word_count` in GPT3.5   **(b) `predicate_count` in Gemini**

(c) `nestedness` in Gemini   **(d) `table_count` in MistralAI**

Figure 8: LLMs' failure in `miss_token` for SQLShare.

Model	SDSS		SQLShare		Join-Order	
	MAE	HR	MAE	HR	MAE	HR
GPT4	**4.69**	**0.56**	**3.96**	**0.63**	**3.45**	**0.57**
GPT3.5	17.71	0.25	7.71	0.42	14.31	0.39
Llama3	15.60	0.33	7.57	0.40	13.11	0.39
MistralAI	18.09	0.36	8.58	0.42	9.92	0.40
Gemini	19.78	0.34	9.79	0.38	20.22	0.32

Table 5: MAE and Hit Rate (HR) for `miss_token_loc`

occurrence of keywords compared to SQLShare and Join-Order. In SQLShare, the most challenging missing token types are aliases and tables (■ and ■), which can be attributed to the presence of many small databases with numerous tables and various aliases in their queries. Finally, in Join-Order, there is no single token type with a notably higher failure rate, likely due to the simpler nature of the queries and the relatively low number of failures.

`miss_token_type`. We reported the weighted average accuracy values in Table 4 (bottom), with weights based on the number of queries for each type. The results indicate that `miss_token_type` is more challenging than `miss_token` across all LLMs, as evidenced by the reduced accuracy. The lowest accuracy is observed in SQL-Share, which is expected due to its complex schema compared to SDSS and Join-Order. Conversely, Join-Order shows the highest accuracy, reflecting its simpler schema. Notably, MistralAI consistently achieves the second-best performance. This is interesting as Llama3 was the second in `miss_token`, which suggests although Llama3 is better at detection due to its strength in detecting general patterns, MistralAI is better at SQL-related pattern recognition, correctly deciding the error type for more queries.

`miss_token_loc`. Table 5 compares the performance of various LLMs in predicting the location of the missing token across SDSS, SQLShare, and Join-Order. The key metrics are Mean Absolute Error (MAE) and Hit Rate (HR), where lower MAE and higher HR indicate better performance.

GPT4 consistently achieves the best results with the lowest MAE and highest HR across all datasets. Llama3 performs well in SQL-Share but shows weaker results elsewhere. GPT3.5 and MistralAI provide reasonable performance but with higher MAE and lower HR, reflecting less precision and accuracy.
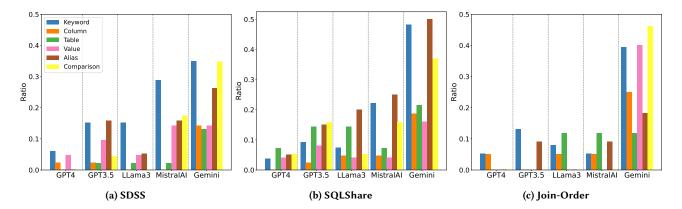
of 57 in FN compared to 27 in TP. We also examined other properties such as predicate count (`predicate_count`), nestedness level (`nestedness`), and table count (`table_count`), as seen in Figures 8b, 8c, and 8d. In all cases, the average values for FN are significantly higher than for TP (1.80 vs 0.90 in 8b, 0.44 vs 0.05 in 8c, and 1.92 vs 1.33 in 8d). However, due to the small number of FP queries, no definitive conclusions can be drawn for that category.

We now shift our analysis to the impact of the missing token type on the performance of LLMs in `miss_token`. We examined the breakdown of FN by token type, as shown in Figure 9, similar to our analysis for `syntax_error`. A key observation in SDSS is that the most frequent type of failure occurs for keyword (■). This is likely because SDSS contains a diverse set of query types with a higher

(a) SDSS **(b) SQLShare** **(c) Join-Order**

Figure 9: Relationship between missing token type and FN in `miss_token`.

Model	Prec.	Rec.	F1
GPT4	**0.88**	**0.93**	**0.90**
GPT3.5	<u>0.81</u>	0.83	<u>0.85</u>
Llama3	0.76	<u>0.90</u>	0.82
MistralAI	0.47	<u>0.90</u>	0.62
Gemini	0.71	0.73	0.72

Table 6: Acc. for `performance_pred`

Most models correctly predict the exact location at least 30% of the time, except GPT3.5 in SDSS, where the HR drops to 25%. Longer queries, especially in SDSS, contribute to higher MAE values, making precise location prediction more difficult.

<u>Takeaways:</u> All models perform better over the missing token tasks than syntax error detection, as missing token identification seems to be a simpler task related to learning frequent patterns. Llama3 shows improved performance due to its broad training in pattern detection. More complex queries tend to increase prediction errors, where complexity is related to different syntactic properties, such as `word_count`, `predicate_count`, `nestedness`, and `table_count`.

4.3 Query Performance Prediction

Table 6 shows the performance metrics for the SDSS dataset on the `performance_pred` task. GPT4 achieves the best results, followed by GPT3.5 and Llama3, which perform similarly. MistralAI and Gemini show lower overall performance. Across all models, recall is generally higher than precision, likely due to positive bias. LLMs tend to produce overly optimistic responses, in this case predicting that queries will take longer to run. Additionally, the queries are selected from more complex, lengthy queries in SDSS, which increases the likelihood of being labeled as costly.

As with `miss_token`, we examined the relationship between syntactic properties and failure rates for this task. The models show strong correlations between `word_count` and failure, with longer queries leading to more FP, as shown in Figure 10a for MistralAI. A similar trend is seen with `column_count` in Figure 10b. This suggests that the models mistakenly associate longer queries or those with more columns with higher execution time.

<u>Takeaways:</u> In the query performance prediction task, GPT4 consistently shows the highest accuracy. However, all models tend to

overestimate runtimes, leading to higher recall but lower precision, especially for longer and more complex queries. This suggests that improving model training with diverse query types could reduce this bias and enhance prediction accuracy.
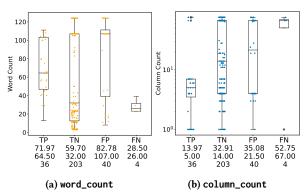

(a) `word_count` **(b) `column_count`**

Figure 10: MistralAI's failure in `performance_pred`

Case	Model	SDSS			SQLShare			Join-Order		
		Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
Equivalence	GPT4	**0.98**	**1.00**	**0.99**	**0.97**	**1.00**	**0.99**	**0.91**	**1.00**	**0.95**
	GPT3.5	0.87	<u>0.99</u>	0.93	<u>0.96</u>	**1.00**	<u>0.98</u>	0.83	<u>0.99</u>	0.90
	Llama3	0.88	**1.00**	0.93	0.94	0.98	0.96	<u>0.87</u>	**0.99**	<u>0.93</u>
	MistralAI	<u>0.95</u>	0.95	<u>0.95</u>	0.95	0.93	0.94	0.86	0.89	0.88
	Gemini	0.84	0.97	0.90	0.92	<u>0.99</u>	0.95	0.85	0.96	0.90
Equiv. Type	GPT4	**0.99**	**0.99**	**0.99**	**0.98**	**0.98**	**0.98**	**0.95**	**0.85**	**0.83**
	GPT3.5	<u>0.97</u>	<u>0.91</u>	<u>0.91</u>	<u>0.96</u>	<u>0.92</u>	<u>0.94</u>	0.90	0.78	0.77
	Llama3	<u>0.97</u>	0.85	0.86	0.93	0.88	0.89	<u>0.93</u>	<u>0.81</u>	<u>0.80</u>
	MistralAI	0.85	0.76	0.80	0.92	0.88	0.89	0.84	0.68	0.68
	Gemini	0.86	0.72	0.71	0.91	0.85	0.87	0.87	0.77	0.75

Table 7: Accuracy in `query_equiv` and `query_equiv_type`

4.4 Query Equivalence

Table 7 presents the results for `query_equiv` (top) and `query_equiv_type` (bottom). For both tasks, GPT4 achieves the best performance, with

GPT3.5 and Llama3 following closely but slightly less consistently. MistralAI and Gemini show more variability and generally lower scores across datasets. A positive bias (higher recall than precision) is noticeable in `query_equiv` but not in `query_equiv_type`, likely due to the latter being a multiclass task rather than binary.

Overall, `query_equiv_type` proves more challenging, with lower performance across LLMs and datasets, except for GPT4, which maintains near-perfect accuracy. This is expected, as determining equivalence is simpler than identifying the type of equivalence. Lower performance in Join-Order and SDSS compared to SQLShare suggests that longer queries make `query_equiv` more difficult.

Across all datasets, most LLMs show very few or no FN, reflected in the high recall. For example, GPT4 records FP in SDSS (5), SQL-Share (4), and Join-Order (9) but has no FN. Thus, we focus on FP to identify where models fail. A common feature of FP queries is that they involve modified conditions, such as changing values in conditions. For instance, altering "WHERE run = 756 AND field = 103" to "WHERE run = 756 AND field = 200" or "WHERE run = 756 OR field = 103." This indicates that LLMs struggle with logical reasoning and numerical manipulation, a limitation extensively discussed in the literature [5–8, 12, 33] and we confirm in our study.

In addition to logical reasoning and numerical issues, these problems become more pronounced in more complex queries, such as those with longer lengths or more tables and predicates. For GPT4 in SDSS, all 5 FP involve queries over 100 words, a pattern also observed in GPT3.5 (Figure 11). In Join-Order, where most queries are lengthy, both FP and FN occur more frequently across all LLMs. We report only Llama3 for Join-Order, as other LLMs exhibit similar trends. Considering `table_count` as a complexity parameter, in Join-Order, all FP occur in queries with more than 8 tables. Figure 12 shows that in SDSS, FP occurs in queries with 5 or more predicates. Similarly, in Join-Order (Figure 12), all FP across models are caused by queries with over 19 predicates. We include the figure for MistralAI, as all LLMs exhibit the same pattern. These suggest that `query_equiv` and `query_equiv_type` are more difficult for complex queries (longer queries with more predicates and tables).

Takeaways: In the query equivalence task, GPT4 performs best across datasets. However, distinguishing between different types of equivalence (`query_equiv_type`) proves more difficult, especially for Gemini and MistralAI. The errors mainly stem from challenges in understanding complex query conditions, highlighting the need for better SQL logic comprehension in LLMs.

4.5 Case Study: Query Explanation

We study the `query_exp` task and analyze several cases where LLMs failed to provide accurate explanations for SQL queries. The queries are presented in Listing 3. Here we present the correct ground truth descriptions from Spider, the erroneous explanations generated by the models, and briefly provide our analysis of the case:

Q15. The query finds the number of students who participate in the tryout for each college, ordered by descending count. Gemini incorrectly describes the query as: "Counts the occurrences of each unique value in the `cName` column." This explanation reduces the query to a simple counting of values in the `cName` column, ignoring the fact that the query specifically searches for students in tryouts, which is essential to convey its full meaning.
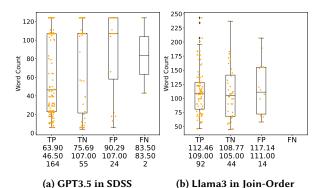

(a) GPT3.5 in SDSS   **(b) Llama3 in Join-Order**

Figure 11: `word_count` and LLM failures in `query_equiv`.

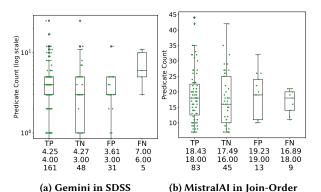**(a) Gemini in SDSS**   **(b) MistralAI in Join-Order**

Figure 12: `predicate_count` and LLM failure in `query_equiv`.

Q16. The query identifies the maximum number of times a course enrollment result can appear in different transcripts and displays the course enrollment ID. Gemini's explanation is: "Finds the student course ID with the highest number of occurrences." While this partially captures the query's purpose, it misses the query context of searching in transcripts.

Q17. The query finds the name and location of stadiums where concerts took place in both 2014 and 2015. GPT4 explains it as: "The query identifies stadiums that hosted concerts in both 2014 and 2015." This only partially explains the query and does not include the selected attributes. This issue occurs because LLMs often focus on capturing the overall semantics of a query but overlook specific details, such as selected attributes, especially in more complex tasks.

Q18. The query retrieves the number of cylinders for the Volvo car with the least acceleration. Llama3 incorrectly explains: "This SQL query retrieves the number of cylinders of the Volvo car with the fastest acceleration." The models misinterpret the "`ORDER BY ...` `ASC LIMIT 1`" clause, misunderstanding that the query is looking for the slowest car (lowest acceleration) rather than the fastest. Only MistralAI correctly explains this query.

Takeaways: These examples highlight a common issue with LLMs when explaining SQL queries: they often miss or misinterpret key details, particularly in tasks requiring context retention. While models may capture parts of a query, they frequently fail to provide complete and accurate explanations. This reflects known limitations

of LLMs in retaining context and applying knowledge to specific scenarios [19, 25, 28].

```
-- Q15:
SELECT count(*),cName FROM tryout
GROUP BY cName ORDER BY count(*) DESC
-- Q16:
SELECT count(*),student_course_id FROM Transcript_Cnt
GROUP BY student_course_id ORDER BY count(*) DESC LIMIT 1
-- Q17:
SELECT S.name,S.loc
FROM concert AS C JOIN stadium AS S
ON C.stadium_id = S.stadium_id WHERE C.Year = 2014
INTERSECT
SELECT S.name,S.loc FROM concert AS C JOIN stadium AS S
ON C.stadium_id = S.stadium_id WHERE C.Year = 2015
-- Q18:
SELECT C.cylinders FROM CARS_DATA AS C
JOIN CAR_NAMES AS T ON C.Id = T.MakeId
WHERE T.Model = 'volvo'
ORDER BY C.accelerate ASC LIMIT 1;
```

Listing 3: Query statements with inaccurate explanations

4.6 Reflections on SQL Understanding

We designed our SQL tasks to probe basic skills in understanding (as described in Section 1): recognition, semantics, context, and coherence.

Demonstrated Skills. Our results show that LLMs, particularly GPT-4, perform well in tasks requiring recognition and context. For instance, in syntax error detection, the models showed high precision in recognizing violations of SQL syntax, indicating a strong ability to identify and interpret the structure of SQL queries. Similarly, missing token identification rely on the model's context-awareness, as the ability to predict missing elements in a query depend on understanding the surrounding tokens and their relationships. This success highlights that LLMs effectively interpret the context within a query to identify missing or incorrect components.

Limitations and Shortcomings. The models were less successful at tasks requiring deeper coherence and semantic understanding. For example, query equivalence tasks proved to be challenging, especially for longer and more complex queries. This difficulty suggests that while LLMs understand surface-level structures, they struggle with deeper semantic coherence, and logical connections within queries. For query performance estimation, the models often overestimated runtimes, indicating that they lack a nuanced understanding of how query complexity, and database-specific factors interact to affect performance.

These observations suggest that while LLMs are adept at recognizing patterns and context, their ability to fully "understand" the deeper semantics and logical coherence of SQL queries is still developing. Future work should focus on refining these models to address these shortcomings, improving their overall SQL proficiency.

5 RELATED WORK

Recent advancements in LLMs have led to innovative approaches in data management, tackling tasks such as data wrangling, entity matching, table manipulation, and text-to-SQL generation.

Li et al. [18] propose an LLM-based approach for data wrangling that leverages code generation for structured data transformations.

This method significantly reduces computational costs compared to row-by-row processing, which is common in traditional LLM approaches. Their work highlights the importance of deterministic transformations to enhance model interpretability and reliability for data tasks like unit conversion and error detection.

In entity matching, Zhang et al. [37] introduce AnyMatch, a zero-shot entity matching model that achieves competitive performance using a small, specialized LLM. By utilizing efficient data selection techniques, this model performs comparably to larger models like GPT-4, while requiring fewer computational resources. Complementing this, Steiner et al. [29] explore the benefits of fine-tuning LLMs for entity matching, showing significant performance improvements but also noting that fine-tuning may reduce cross-domain generalization.

LLMs have been used for table manipulation as shown in Li et al. [17] with Table-GPT. This fine-tuned model is designed for tasks such as data cleaning and table-based question answering. The study demonstrates that LLMs trained on natural language text face limitations when handling two-dimensional tabular data, and that table-specific fine-tuning is necessary to overcome these challenges.

LLMs have also made significant progress in text-to-SQL tasks. Surveys by Hong et al. [11] and Gao et al. [9] provide overviews of how LLMs handle complex and cross-domain SQL generation. These studies highlight that while LLMs perform well on simpler queries, their accuracy drops with more complex structures involving nested queries, joins, and aggregations.

These works underscore the expanding role of LLMs in data management, from entity matching and data wrangling to text-to-SQL. Despite their potential for automating complex tasks, further research is needed to overcome challenges in efficiency, scalability, and generalization across domains.

6 CONCLUSION AND FUTURE WORK

In this paper, we study the proficiency of state-of-the-art LLMs towards "understanding" SQL. We evaluate their performance on key SQL tasks such as syntax error identification, missing token identification, query equivalence, query performance estimation, and query explanation. Our evaluation revealed that all models perform well on tasks requiring recognition and context. GPT4 consistently outperformed other models, particularly in handling complex SQL queries, while GPT3.5, MistralAI, and Llama3 showed strong capabilities in pattern recognition. In contrast, Gemini struggled with all SQL-specific tasks, particularly error detection. Despite these strengths, LLMs faced challenges with long and complex queries, an inability to pinpoint the exact location of missing tokens, and struggled with tasks requiring semantic coherence and logical connections within queries. As next steps, we will explore fine-tuning to handle query complexity and dynamic prompt tuning (to improve accuracy), and barriers to using LLMs for query recommendation and query optimization. We anticipate that targeted fine-tuning and dynamic prompt adjustment could significantly mitigate current limitations in handling complex queries and improve task-specific performance. This enhancement of LLMs is expected to bridge the gap between AI capabilities and real-world SQL needs, enabling better integration with database systems.

REFERENCES

[1] Anthropic. 2024. Gemini: A Safe and Ethical Large Language Model. *Anthropic Research* (2024).

[2] Ian Arawjo, Chelse Swoopes, Priyan Vaithilingam, Martin Wattenberg, and Elena L Glassman. 2024. ChainForge: A Visual Toolkit for Prompt Engineering and LLM Hypothesis Testing. In *CHI*. 1–18.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Few-Shot Learners. *arXiv preprint arXiv:2005.14165* (2020).

[4] Surajit Chaudhuri. 1998. An overview of query optimization in relational systems. In *PODS*. 34–43.

[5] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research* 24, 240 (2023), 1–113.

[6] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training Verifiers to Solve Math Word Problems. *ArXiv* abs/2110.14168 (2021). https://api.semanticscholar.org/CorpusID:239998651

[7] Antonia Creswell, Murray Shanahan, and Irina Higgins. [n.d.]. Selection-Inference: Exploiting Large Language Models for Interpretable Logical Reasoning. In *The Eleventh International Conference on Learning Representations*.

[8] Simon Frieder, Luca Pinchetti, Ryan-Rhys Griffiths, Tommaso Salvatori, Thomas Lukasiewicz, Philipp Petersen, and Julius Berner. 2024. Mathematical capabilities of chatgpt. *NeurIPS* 36 (2024).

[9] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. [n.d.]. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. ([n. d.]).

[10] Alon Y Halevy et al. 2014. SQLShare: A Platform for Structured Data Sharing. *CIDR* (2014).

[11] Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2024. Next-Generation Database Interfaces: A Survey of LLM-based Text-to-SQL. *arXiv preprint arXiv:2406.08426* (2024).

[12] Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. 2022. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International conference on machine learning*. PMLR, 9118–9147.

[13] Won Kim. 1982. On optimizing an SQL-like nested query. *ACM TODS* 7, 3 (1982), 443–469.

[14] Eugenie Yujing Lai, Zainab Zolaktaf, Mostafa Milani, Omar AlOmeir, Jianhao Cao, and Rachel Pottinger. 2023. Workload-Aware Query Recommendation Using Deep Learning. In *EDBT*. 53–65.

[15] Bruce W Lee and JaeHyuk Lim. 2024. Tasks That Language Models Don't Learn. *arXiv preprint arXiv:2402.11349* (2024).

[16] Viktor Leis et al. 2015. Join Order Benchmark. *VLDB* (2015).

[17] Peng Li, Yeye He, Dror Yashar, Weiwei Cui, Song Ge, Haidong Zhang, Danielle Rifinski Fainman, Dongmei Zhang, and Surajit Chaudhuri. 2024. Table-GPT: Table Fine-tuned GPT for Diverse Table Tasks. *ACM SIGMOD* 2, 3 (2024), 1–28.

[18] Xue Li and Till Döhmen. 2024. Towards Efficient Data Wrangling with LLMs using Code Generation. In *DM4ML*. 62–66.

[19] Xing Liu et al. 2023. Attention Mechanisms in Large Language Models: A Critical Review. *JAIR* 69 (2023), 1021–1050.

[20] Ggaliwango Marvin, Hellen Nakayiza, Daudi Jjingo, and Joyce Nakatumba-Nabende. 2023. Prompt engineering in large language models. In *Springer DICI*. 387–402.

[21] MistralAI. 2023. Mistral: New Model Architecture and Training. *Company Blog* (2023). Accessed: 2024-08-14.

[22] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *ICSE*. 1–13.

[23] OpenAI. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023).

[24] Karl Pearson. 1895. Note on Regression and Inheritance in the Case of Two Parents. *Proceedings of the Royal Society of London* 58 (1895), 240–242.

[25] Fabio Petroni et al. 2020. Contextualizing Knowledge for LLMs: Challenges and Opportunities. In *EMNLP*. 123–136.

[26] Jing Qian, Hong Wang, Zekun Li, Shiyang Li, and Xifeng Yan. [n.d.]. Limitations of Language Models in Arithmetic and Symbolic Induction. ([n. d.]).

[27] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. 2024. A systematic survey of prompt engineering in large language models: Techniques and applications. *arXiv preprint arXiv:2402.07927* (2024).

[28] Noah Shinn, Shuyuan Kuo, et al. 2023. Context Forgetting in Large Language Models. *arXiv preprint arXiv:2303.12345* (2023).

[29] Aaron Steiner, Ralph Peeters, and Christian Bizer. 2024. Fine-tuning Large Language Models for Entity Matching. *arXiv preprint arXiv:2409.08185* (2024).

[30] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothee Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. [n.d.]. LLaMA: Open and Efficient Foundation Language Models. ([n. d.]).

[31] Thinh Hung Truong, Timothy Baldwin, Karin Verspoor, and Trevor Cohn. 2023. Language models are not naysayers: an analysis of language models on negation benchmarks. In **SEM*. 101–114.

[32] Li Wang, Xi Chen, XiangWen Deng, Hao Wen, MingKe You, WeiZhi Liu, Qi Li, and Jian Li. 2024. Prompt engineering in consistency and reliability with the evidence-based guideline for LLMs. *npj Digital Medicine* 7, 1 (2024), 41.

[33] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[34] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. [n.d.]. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. ([n. d.]).

[35] Donald G York et al. 2000. The Sloan Digital Sky Survey. *AJ* (2000).

[36] Tao Yu et al. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. *EMNLP* (2018).

[37] Zeyu Zhang, Paul Groth, Iacer Calixto, and Sebastian Schelter. 2024. AnyMatch–Efficient Zero-Shot Entity Matching with a Small Language Model. *arXiv preprint arXiv:2409.04073* (2024).

[38] Victor Zhong et al. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *arXiv preprint arXiv:1709.00103* (2017).

[39] Zainab Zolaktaf, Mostafa Milani, and Rachel Pottinger. 2020. Facilitating SQL query composition and analysis. In *SIGMOD*. 209–224.