

Text-to-(Single-or-Multi)-SQL: Bridging Real-World Needs with SpiderS

Yujian Gan^{1,2,*} Xuan Ouyang^{3,*} Yuxi Lin^{4,*}
Zhilin Zhang⁵ Jinxia Xie⁶ Vassilis Routsis^{2,†}

¹Queen’s University Belfast ²University College London

³University of Wisconsin–Madison ⁴University of Hong Kong

⁵Pennsylvania State University ⁶Nanjing University of Science and Technology

*Equal contribution †Corresponding author

y.gan@qub.ac.uk, xouyang7@wisc.edu, v.routsis@ucl.ac.uk

Abstract

Text-to-SQL systems allow users to interact with databases using natural language, but most assume that each question must be answered by a single SQL query. In practical settings, some information needs are more reliably satisfied by generating multiple simpler SQL queries, which is not supported by existing benchmark or evaluation protocols. To address this gap, we introduce SpiderS, which, to the best of our knowledge, is the first benchmark for the Text-to-(Multi)-SQL task, supporting both single- and multi-query solutions within a unified evaluation framework. To enable this setting, we propose a flexible evaluation protocol that accepts any prediction covering the required information, regardless of query number or structure. Under our evaluation protocol, we evaluate several large language models on SpiderS and find that allowing multi-query outputs improves execution accuracy. However, current models demonstrate fragility: even a minor prompt change from generating one SQL to generating one or more SQL can significantly degrade performance. To improve robustness, we introduce a lightweight router that decides whether to generate a single or multiple SQL for a given question, leading to substantial performance gains. The data and code are publicly available at <https://github.com/ygan/SpiderS>.

1 Introduction

Natural Language Interfaces to Databases (NLIDBs) allow users to query relational data using everyday language, removing the need to master formal query languages like SQL. Decades of progress in both the database and NLP communities have enabled NLIDBs to successfully convert user questions into structured queries (Qin et al., 2022; Pourreza and Rafiei, 2023; Hong et al., 2024; Zhang et al., 2024).

At the core of NLIDBs lies the Text-to-SQL task, where a natural language utterance is translated

Question: List students’ names, GPAs, and average GPA.

Solution 1: `SELECT name , GPA , (SELECT avg(GPA) FROM student) FROM student;`
(Single SQL)

Solution 2: `SELECT name , GPA FROM student
UNION ALL
SELECT ‘Avg GPA’ , avg(GPA) FROM student;`
(Single SQL)

Solution 3: `SELECT ‘Average GPA’ , avg(GPA) FROM student
UNION ALL
SELECT name , GPA FROM student;`
(Single SQL)

Solution 4: `SELECT name , GPA FROM student;
(Multi SQL) SELECT avg(GPA) FROM student;`

Figure 1: A single user request may correspond to multiple valid SQL solutions.

into one SQL query that returns the correct data (Iacob et al., 2020; Abbas et al., 2022; Katsogiannis-Meimarakis and Koutrika, 2023; Gao et al., 2023; Pourreza et al., 2024; Zhu et al., 2024). However, in real-world settings, especially in education, business analytics, and reporting, users care more about whether their information need is satisfied, and less about whether this happens through one SQL or several. For example, a teacher summarizing student performance may want to retrieve each student’s GPA as well as the average GPA across all students. While this goal can be achieved via a single SQL query using nested subselects or UNION constructs, a much simpler and clearer approach is to execute two separate queries: one for listing names and GPAs, and another for computing the average. As shown in Figure 1, there are multiple correct SQL solutions to this request, including both single-query and multi-query forms.

Although all four SQL solutions in Figure 1 return exactly the same information, existing Text-to-SQL benchmarks typically mark only one of them as correct (the so-called “gold” query). Suppose the gold annotation is Solution 1; then if a model predicts Solution 2 or 3 (both syntactically different but semantically equivalent), it would still be judged incorrect. Worse still, Solution 4 adopts a multi-SQL strategy that splits the task into two simpler queries. This approach is equally valid and

often easier to interpret, yet it falls entirely outside the scope of current benchmarks, which assume every question should be answered by a single SQL query. As a result, the current evaluation setting fails to capture valid alternative realizations and overlooks natural use cases where multiple SQLs are more appropriate.

The First Text-to-(Multi)-SQL Benchmark. To ensure that all four solutions in Figure 1 are treated as positive predictions under evaluation, we advocate a Text-to-(Multi)-SQL paradigm, where models may output either a single SQL query or multiple SQL queries, and correctness is determined by information coverage rather than query form. Following this paradigm, we construct SpiderS, the first benchmark that explicitly supports both single-SQL and multi-SQL solutions under a unified evaluation framework. SpiderS is built upon the Spider dataset (Yu et al., 2018). Each question in SpiderS can be satisfied by either a single SQL query or a set of multiple SQL queries. This design is not only semantically motivated but also practically useful: while single-SQL is often faster on small result sets, as is common in Spider, multi-SQL can be more efficient for large-output cases, as analyzed in Appendix G. Accordingly, we annotate multi-SQL solutions for all questions, and additionally include equivalent single-SQL alternatives whenever needed.

Although all questions in SpiderS admit a correct single-SQL solution and existing training data only provide supervision for text-to-single-SQL, we observe that multi-SQL generation consistently outperforms single-SQL generation across several widely used large language models (LLMs). This suggests that allowing models to decompose complex requests into multiple simpler SQL queries can substantially improve reliability, even without explicit Text-to-(Multi)-SQL supervision. Nevertheless, existing methods still exhibit a significant performance drop on SpiderS compared to Spider, with execution accuracy decreasing by 0.1 to 0.4. Moreover, a minimal prompt change: from “generate one SQL” to “generate one or more SQL”, can noticeably degrade performance on the Spider Text-to-single-SQL benchmark, revealing a robustness gap when models are asked to reason beyond a fixed single-query format.

Evaluation Protocol. Existing metrics typically require an exact match between the predicted SQL

and the gold SQL, either in their syntactic structure or in their execution results. This is especially limiting in scenarios where multiple correct SQL solutions exist. We introduce a new evaluation protocol where a prediction is accepted as correct if it contains all the information required by any of the gold queries, regardless of whether it is expressed as a single or multiple SQL queries. For example, as shown in Figure 1, even though SpiderS only annotates Solution 1 and Solution 4, our evaluation correctly accepts Solution 2 and Solution 3 as well, since they return the same information.

Modeling Strategy. To support multi-SQL generation while avoiding overly complex SQL queries, we explore two modeling strategies. The first is a direct generation approach, where we fine-tune a language model to generate either single or multiple SQL queries based on the input question. The second approach introduces a lightweight *router* that determines whether a question should be handled with single or multiple SQL queries, and guides the downstream models accordingly. Both approaches demonstrate strong empirical performance. In particular, the router is compact, easy to integrate, and significantly enhances overall performance.

Contributions. Our contributions are threefold:

1. We introduce SpiderS, the first Text-to-(Multi)-SQL dataset, featuring multiple valid SQL solutions, including both single- and multi-query forms when applicable.
2. We propose a flexible evaluation protocol that accepts predictions as correct if they cover the information required by any gold solution, enabling robust evaluation without exhaustively annotating all valid variants.
3. We explore two modeling strategies: (1) supervised generation, where the model is trained to output single- or multi-SQL queries as needed; and (2) a lightweight router that decides whether to use single- or multi-query prompting. Both perform well in practice.

2 Related Work

Text-to-SQL is a central task in Natural Language Interfaces to Databases (NLDBs), aiming to map natural language questions to executable SQL queries (Katsogiannis-Meimarakis and Koutrika, 2023). A number of benchmarks have facilitated this line of research, including WikiSQL (Zhong et al., 2017), Spider (Yu et al., 2018), and BIRD (Li

et al., 2023b). However, these datasets and their evaluations predominantly assume that each question should be answered by a single SQL query, which overlooks realistic cases where multiple simpler SQL queries better satisfy the same information need. Recent work has also examined the limitations of Text-to-SQL systems from system design and evaluation perspectives (Biswal et al., 2024; Ma et al., 2025; Renggli et al., 2025). More details of related work are given in Appendix A.

3 Theoretical Motivation

We provide an idealized argument for why generating multiple simpler SQL queries can be easier than composing one monolithic query. Further analysis is provided in Appendix B.

Let $C(q) \geq 0$ denote the complexity of a SQL query, such as the number of joins or nesting depth. We use an idealized surrogate model in which the probability of correctly generating a query decreases exponentially with its complexity:

$$P(\text{correct } q) = e^{-\alpha C(q)}, \quad \alpha > 0. \quad (1)$$

For cases in which a question can be answered either by a multi-SQL set $\{q_1, \dots, q_m\}$ or a single combined query $q^{(\text{single})}$, composing the single query may introduce additional structural overhead, such as nesting or UNION. We model this overhead as

$$C(q^{(\text{single})}) \geq \sum_{i=1}^m C(q_i) + \Delta, \quad \Delta > 0. \quad (2)$$

Assuming independence across sub-queries, this yields

$$\frac{P_{\text{multi}}}{P_{\text{single}}} \geq e^{\alpha \Delta} > 1, \quad (3)$$

suggesting that, within this idealized model, multi-SQL generation can be easier when composing a single query introduces non-trivial structural overhead. This intuition is consistent with our empirical results.

4 Dataset Construction

4.1 Annotation and Data Generation

Question and SQL Annotation. To construct the SpiderS benchmark, we extend the original Spider dataset by rewriting existing user requests to increase their complexity and creating multiple SQL queries for each one. Three undergraduate

Question 1: Find the name of the district which has the largest area and the name of spokesmen who do not speak for any district.
Solution: `SELECT name FROM district ORDER BY Area_km DESC LIMIT 1;`
(Multi SQL) `SELECT name FROM spokesman WHERE Spokesman_ID NOT IN ( SELECT Spokesman_ID FROM spokesman_district );`

Question 2: What is the average price for products with type Clothes, and how many customers use each payment method?
Solution: `SELECT avg(product_price) FROM Products WHERE product_type_code = 'Clothes';`
(Multi SQL) `SELECT payment_method_code , count(*) FROM Customers GROUP BY payment_method_code;`

Figure 2: Examples based on the same pattern.

and graduate students in computer science were responsible for the data annotation. After analyzing 300+ initially annotated examples, we found that all examples could be categorized into 14 typical data generation patterns, such as rewriting user request or merging multiple related ones. The details of the 14 data generation patterns are provided in the Appendix E. We then aimed to annotate at least one representative example for each pattern in each database, which served as guidance for GPT-4o-based data generation. In total, we manually annotated 890 high-quality examples, each consisting of a clear, composite natural language request and its corresponding SQL queries.

Data Generation. Based on these identified patterns, we designed prompt templates to guide GPT-4o in generating large-scale data. All generated examples underwent a three-stage validation process: (1) automatic checks for syntactic correctness, pattern conformity, and schema alignment; (2) a GPT-4o-based semantic check to verify that the question matches the SQL behavior; and (3) manual inspection of flagged cases. Any example that failed any check was discarded. Details of the validation process are provided in the Appendix D.2.

4.2 Sources of Data Diversity

Although we defined only 14 generation patterns, the generated dataset exhibits strong diversity for two main reasons:

First, the inputs used for generation vary widely in both content and structure. Some examples are derived from a single user request and SQL query, while others are created by combining multiple inputs in different ways. This variation in input configurations leads to diverse outputs even under the same pattern. For example, the two examples shown in Figure 2 are generated using the same pattern (merging two simple requests), but their dif-

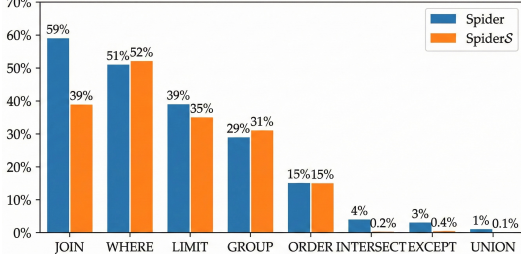


Figure 3: Percentage of SQL keywords.

fering input results in output request and SQL that vary significantly in both structure and semantics.

Second, we applied constraints during generation to minimize repetition at the output level, resulting in a duplication rate below 15%. For comparison, around 50% of examples in the original Spider dataset share identical SQL outputs.

4.3 Dataset Statistics

Our dataset follows the same structure as the Spider dataset, with 8,475 training samples, 1,124 validation samples, and 2,147 test samples. Of these, 890 examples were manually created, and the remaining 10,856 were automatically generated. On average, each question requires 2.097 SQL queries to answer. Specifically, 1,134 samples need 3 SQL queries, while 10,612 samples require 2.

Figure 3 compares the frequency of SQL keywords between the two datasets. Compared to the original Spider, SpiderS has similar frequencies for keywords like WHERE, GROUP, ORDER, and LIMIT. However, JOIN, UNION, INTERSECT, and EXCEPT are less frequent in SpiderS because we intentionally keep per-query SQL complexity moderate, so the benchmark primarily stresses multi-query decomposition rather than heavy set operations under the Spider schema.

5 Evaluation Protocol and Metrics

Execution Match. Prior work typically uses execution match as the evaluation criterion, which requires the predicted SQL to return the exact same result as the gold SQL, ignoring the order in which columns appear. However, this strict comparison often misjudges semantically correct SQLs as incorrect. To address this, we mark a prediction as correct if it contains all information required by the gold query with no extra rows; extra columns are allowed, and correctness is checked after projecting the predicted result onto the essential columns. For example, in Figure 4, predictions that add extra

GOLD	NAME GPA		NAME GPA		
	Jack	3.5	Jack	3.5	
	Andy	3.8	Andy	3.8	
			Lucy	3.3	
PREDICT	NAME GPA		NAME AGE GPA		
	Andy	3.8	Jack	20	3.5
	Jack	3.5	Andy	21	3.8

Figure 4: Examples of accepted and rejected predicted SQL results under our evaluation protocol.

columns are accepted, while those with extra rows or incorrect ordering are rejected. Our protocol encourages minimal gold annotations that capture the essential columns needed to satisfy the user’s request, allowing other logically correct SQL queries to be judged positive even if they are not explicitly annotated.

We further formalize this with two evaluation metrics: Execution Score (ES) and Execution Accuracy (EX), which are defined below.

Execution Score (ES). To evaluate the correctness of multi-SQL predictions, we introduce a metric called Execution Score (ES). For the i^{th} example, Let G_i denote the collection of valid gold solution sets, where each $g \in G_i$ contains one or more SQL queries that jointly provide a valid answer to the question. Let P_i be the set of predicted SQL queries.

If a predicted SQL contains composite constructs such as UNION or UNION ALL, we apply two forms of processing: (1) using the original predicted SQLs as-is, and (2) splitting any composite queries into their subqueries. Each of these two variants yields a candidate predicted set, and we compute ES scores for both.

The final ES_i is defined as the maximum score obtained by comparing each candidate predicted set with every valid gold solution set:

$$ES_i = \max_{\tilde{P}_i \in \{P_i, \text{split}(P_i)\}} \left(\max_{g \in G_i} \frac{m(\tilde{P}_i, g)}{\max(|\tilde{P}_i|, |g|)} \right), \quad (4)$$

where:

- P_i : The set of original predicted SQL queries.
- $\text{split}(P_i)$: The set of subqueries obtained by decomposing any composite queries in P_i .
- G_i : The set of valid gold solution sets for the i^{th} question, where each $g \in G_i$ contains one or more gold SQL queries that jointly provide a valid answer.
- $\tilde{P}_i$: Either P_i or $\text{split}(P_i)$, used during scoring.

- $m(\tilde{P}_i, g)$: The cardinality of a maximum one-to-one matching between queries in $\tilde{P}_i$ and g , where a predicted query p can be matched with a gold query g_j if their execution results match under the execution-match criterion described above. Each predicted query and each gold query can participate in at most one matched pair.
 - $|\cdot|$: The cardinality (number of queries) in a set.
- According to the ES computation process, Solution 2 and Solution 3 in Figure 1 can be split and then successfully matched with Solution 4, ultimately achieving the maximum ES score of 1.

Execution Accuracy (EX). EX is formally defined as the proportion of examples in the evaluation set for which the executed results of both the predicted and ground-truth SQL for a single question are identical, relative to the overall number of examples. However, in our task definition, we require multiple SQL queries to address one single question. Therefore, following the previous definition of Execution Score (ES), the i^{th} question in the evaluation set is considered solved only when ES_i is equal to 1. That is, the execution result of each of the predicted SQL can match the one of a corresponding ground-truth SQL. The computation of EX can be expressed as follows:

$$\text{EX} = \frac{\sum_{i=1}^N \mathbb{M}(ES_i)}{N}, \quad (5)$$

where function $\mathbb{M}(\cdot)$ returns 1 only if ES_i is equal to 1, indicating that every executed result in predicted result set P_i can match a corresponding one in ground-truth result set G_i . The expression of $\mathbb{M}(\cdot)$ is shown below:

$$\mathbb{M}(ES) = \begin{cases} 1, & \text{if } ES_i = 1, \\ 0, & \text{otherwise.} \end{cases} \quad (6)$$

6 Methodology

Task. Text-to-(Multi)-SQL maps a natural language question Q and database schema D to a set of one or more SQL queries $y = \{y_1, \dots\}$ that together satisfy the information need:

$$y = f(Q, D \mid \phi). \quad (7)$$

We provide the full formalization in Appendix C.1.

Baseline 1: SQL Generator. We fine-tune a language-model-based generator to output either a single SQL or multiple SQL queries serialized in a structured format (e.g., a JSON object with keys `sql_1`, `sql_2`, ...). We adopt LoRA for

parameter-efficient adaptation (Hu et al., 2021); training objectives and implementation details are deferred to Appendix C.2.

Baseline 2: Router + SQL Generation. To enable existing models to better handle multi-SQL generation, we introduce a lightweight router R that predicts whether a query should be answered with a single or multiple SQL queries, and accordingly selects an appropriate prompt for the downstream generator. The router is not designed to directly serve end users; instead, it is intended to enhance the capabilities of existing models. We instantiate R with RoBERTa-base and Qwen2.5-Instruct variants (0.5B, 7B); model/training details are in Appendix C.3.

7 Experiments

7.1 Experimental Setup

Prompt. We use three prompt templates: `Prompt1` asks for exactly one SQL query, `Prompt1VN` allows one or multiple SQL queries as needed, and `PromptN` explicitly asks for multiple SQL queries. `Prompt1VN` is our main unified setting for testing whether models can decide the number of queries themselves, while `PromptN` is used only in the router-based setting after the input is classified as multi-SQL.

Dataset. We conducted our experiments in both the Spider and SpiderS test sets. For efficient and low-cost evaluation, we independently sampled 1,000 examples from the complete Spider and SpiderS test sets (2,147 examples each), resulting in two distinct small test sets.

The evaluation tasks are defined below, where each task corresponds to a specific dataset and prompt type.

- **TaskSM:** Full SpiderS test set with `Prompt1VN`. $\mathbb{M}$ indicates `Prompt1VN`.
- **TaskSM_{small}:** The small SpiderS test set (1,000-item) with `Prompt1VN`.
- **TaskSO_{small}:** The small SpiderS test set, with `Prompt1`. $\mathbb{O}$ represents the use of `Prompt1`.
- **TaskRO:** The complete Spider test set (2,147-item) with `Prompt1`. $\mathbb{R}$ is derived from the last letter of “Spider”.
- **TaskRO_{small}:** The small Spider test set (1,000-item) with `Prompt1`.
- **TaskRM:** The complete Spider test set (2,147-item) with `Prompt1VN`.

Question: Which engine is most commonly used by drivers, and how many drivers are there per constructor?

Solution 1: (Single SQL)

```
SELECT 'Most Common Engine' AS Result, (
  SELECT Engine FROM driver GROUP BY Engine
  ORDER BY COUNT(*) DESC LIMIT 1 ) AS Engine_Info,
  NULL AS Constructor, NULL AS Driver_Count
UNION ALL
SELECT 'Driver Count Per Constructor' AS Result, NULL
AS Engine_Info, Constructor AS Constructor, COUNT(*)
AS Driver_Count FROM driver GROUP BY Constructor;
```

Solution 2: (Multi SQL)

```
SELECT Engine FROM driver GROUP BY Engine
ORDER BY COUNT(*) DESC LIMIT 1;
SELECT Constructor, COUNT(*) FROM driver
GROUP BY Constructor;
```

Figure 5: For tasks that require multiple SQL queries, forcing a single-SQL solution is often more complex.

Models	TaskSM _{small}		TaskSO _{small}	
	EX	ES	EX	ES
GPT-4o	0.519	0.669	0.229	0.373
Gemini-2.5-Flash	0.662	0.779	0.460	0.571
Qwen2.5-Coder-7B	0.457	0.617	0.126	0.250
XiYanSQL-32B	0.685	0.802	0.255	0.377

Table 1: EX and ES scores of different models on TaskSM_{small} and TaskSO_{small}.

- **TaskRM_{small}:** The small Spider test set (1,000-item) with Prompt_{1vN}.

Models. We evaluated Text-to-(Multi)-SQL capabilities across open-source, fine-tuned, closed-source LLMs, and multi-agent Text-to-SQL models. For open source models, we study Llama3.1 (Grattafiori et al., 2024), Qwen 2.5 (Qwen Team et al., 2025), CodeLlama (Rozière et al., 2024). Among the closed-source model, we evaluate Gemini-2.5-Flash (Comanici et al., 2025), GPT-4o (OpenAI et al., 2024), Claude-3.5-Sonnet, and o3-mini. For Text-to-SQL models, we choose CHESS (Talei et al., 2024) and XiYanSQL-32B (Gao et al., 2024). For ease of discussion, we refer to Qwen2.5-Coder-Instruct as Qwen2.5-Coder in this paper. For implementation details, please refer to Appendix F.

7.2 Performance: Single-SQL vs. Multi-SQL

As shown in Table 1, constraining models to generate a single SQL query leads to a substantial performance drop across all evaluated models. This supports our theoretical analysis in Section 3: a monolithic SQL often needs to compose multiple user intents through nesting, aggregation, or set operations, making generation more difficult. Although Gemini-2.5-Flash shows the smallest drop, stronger reasoning ability only partially mitigates this difficulty. The gap is especially clear in complex examples. For instance, while Figure 1 shows a case where the single- and multi-SQL solutions

are comparable, Figure 5 illustrates a case where the single-SQL solution requires substantially more structural composition.

We further analyze execution efficiency in Appendix G. On Spider, most databases and result sets are small, so the single-statement form is faster on most examples. However, for large-output cases, merging independent SQL queries into one statement inevitably introduces NULL padding, and wider results through the UNION ALL wrapper; these few cases dominate the total execution time, making multi-SQL faster overall. Thus, single-SQL is suitable for small-output settings, whereas multi-SQL is more efficient for large databases or large-result queries. Since multi-SQL is also simpler and more reliable to generate, we relax the single-query constraint in subsequent experiments and exclude TaskSO_{small} from further evaluation.

7.3 Main Results

Table 2 compares the performance of the models in the Spider and SpiderS datasets. Among models that have not been fine-tuned on SpiderS, XiYanSQL-QwenCoder-32B, which is specifically optimized for SQL generation, performs the best. Among the other models, Gemini-2.5-Flash achieves the best results, with Llama-3.1-405B-Instruct close behind. The model with the lowest performance is CodeLlama-7B-Instruct, mainly due to its inability to generate correct JSON output and the significant hallucinations it produces.

Small Test Sets Provide Reliable Evaluation.

When comparing the full test set to its subset, we find the results are highly consistent, with the smaller dataset achieving very similar performance to the full one. This suggests that evaluation on the smaller set is both reliable and representative.

Performance Scales with Model Size. There is a clear positive correlation between model size and performance. For example, in the Llama-3.1 series, the 8B-parameter achieves an EX score of 0.414 on the TaskSM_{small}, while the 405B-parameter model achieves a score of 0.554. A similar trend is seen in the Qwen2.5 series, where the 32B model outperforms the 7B model by 0.059 in EX score on the TaskSM_{small}.

Effectiveness of Fine-Tuning. We fine-tuned three models: Qwen2.5-Coder-7B-Instruct, CodeLlama-7B-Instruct, and Llama-3.1-8B-Instruct, each on the combined training data of

Models	Task $\mathbb{R}\mathbb{O}$		Task $\mathbb{R}\mathbb{M}$		Task $\mathbb{S}\mathbb{M}$		Task $\mathbb{R}\mathbb{O}_{small}$		Task $\mathbb{R}\mathbb{M}_{small}$		Task $\mathbb{S}\mathbb{M}_{small}$	
	EX	ES	EX	ES	EX	ES	EX	ES	EX	ES	EX	ES
Closed-source Models												
GPT-4o	-	-	-	-	0.499	0.653	0.786	0.786	0.667	0.720	0.519	0.669
Claude-3.5-Sonnet	-	-	-	-	-	-	0.721	0.721	0.712	0.712	0.317	0.490
*o3-mini	-	-	-	-	-	-	0.773	0.773	0.772	0.772	0.535	0.662
*Gemini-2.5-Flash	-	-	-	-	-	-	<u>0.803</u>	<u>0.803</u>	<u>0.794</u>	<u>0.795</u>	0.662	0.779
Open-source Models												
Qwen2.5-Coder-7B-Instruct	0.773	0.773	0.706	0.726	0.447	0.609	0.774	0.774	0.709	0.728	0.457	0.617
Qwen2.5-Coder-32B-Instruct	0.802	0.802	0.753	0.773	0.506	0.654	0.809	0.809	0.769	0.785	0.516	0.665
CodeLlama-7B-Instruct	0.141	0.141	0.095	0.103	0.042	0.079	0.144	0.144	0.084	0.092	0.037	0.074
CodeLlama-34B-Instruct	0.196	0.196	0.253	0.256	0.094	0.154	0.190	0.190	0.252	0.254	0.095	0.156
Llama-3.1-8B-Instruct	0.653	0.653	0.653	0.660	0.395	0.560	0.645	0.645	0.648	0.653	0.414	0.579
Llama-3.1-405B-Instruct	-	-	-	-	-	-	0.788	0.788	0.750	0.752	0.554	0.714
Mistral-Large-Instruct-2411	-	-	-	-	-	-	0.730	0.730	0.504	0.572	0.453	0.636
Fine-tuned Models												
Qwen2.5-Coder-7B-Instruct	-	-	0.788	0.789	0.725	0.831	-	-	0.787	0.788	0.731	0.837
CodeLlama-7B-Instruct	-	-	0.730	0.730	0.656	0.781	-	-	0.730	0.731	0.651	0.781
Llama-3.1-8B-Instruct	-	-	0.762	0.762	0.706	0.811	-	-	0.755	0.755	<u>0.724</u>	<u>0.822</u>
Text-to-SQL Models												
CHES (GPT-4o-mini)	-	-	-	-	-	-	0.775	0.775	0.494	0.618	0.568	0.714
XiYanSQL-32B	0.852	0.852	0.813	0.819	0.671	0.791	0.854	0.854	0.826	0.830	0.685	0.802

Table 2: Performance comparison of various models on Spider and Spider $\mathbb{S}$. * denotes reasoning LLMs. CHES is a model-agnostic multi-agent Text-to-SQL system. Entries marked with “-” denote full-test-set evaluations omitted due to computational and API-cost constraints. Fine-tuned models are evaluated only under the Prompt_{1VN} format used during training, so Task $\mathbb{R}\mathbb{O}$ and Task $\mathbb{R}\mathbb{O}_{small}$ with Prompt_1 are excluded.

Spider and Spider $\mathbb{S}$ using Prompt_{1VN} . This joint-dataset fine-tuning yields substantial performance improvements. For example, CodeLlama-7B’s EX on Task $\mathbb{S}\mathbb{M}$ increases significantly from 0.042 to 0.656, representing a relative improvement of 1461%. Besides, Qwen2.5-Coder-7B achieves the best overall results, attaining the highest EX and ES scores on Task $\mathbb{S}\mathbb{M}$ among all evaluated models.

7.4 Robustness Analysis

We observe that many existing Text-to-SQL models exhibit limited robustness when facing prompt variations and increasing task complexity. Nearly all models exhibit notable performance drops when transitioning from Task $\mathbb{R}\mathbb{O}$ to Task $\mathbb{R}\mathbb{M}$, with some experiencing dramatic declines. CHES, which adopts a multi-agent architecture, suffers the largest drop in EX score (0.281). The next two largest drops are observed in Mistral-Large-Instruct (0.226) and GPT-4o (0.119), respectively. The only change between these two tasks is a small modification to the prompt: from Prompt_1 to Prompt_{1VN} . Although syntactically minor, this prompt change increases output flexibility, often causing models to generate multiple plausible SQL queries when one is sufficient, thereby reducing EX under the single-SQL Spider setting; see Appendix J for a targeted error analysis.

The most robust models are o3-mini and Gemini-2.5-Flash, both of which exhibit minimal perfor-

mance degradation. We observe that these two reasoning models tend to first determine the number of required SQL queries before generation, which makes them less sensitive to prompt modification. A notable exception is CodeLlama-34B-Instruct, which shows a slight performance improvement on the Task $\mathbb{R}\mathbb{M}$. However, given its overall weak performance, this gain is unlikely to be meaningful.

When comparing Task $\mathbb{R}\mathbb{O}$ and Task $\mathbb{S}\mathbb{M}$, we observe a substantial performance drop across models, suggesting that Task $\mathbb{S}\mathbb{M}$ introduces additional challenges. Since Spider $\mathbb{S}$ is derived from Spider, we further examine the effect of Spider overlap in Appendix I. Claude-3.5-Sonnet shows the largest decline, with a 0.404 drop in EX. Even reasoning LLMs like Gemini-2.5-Flash experience a 0.14 drop, while the Text-to-SQL model XiYanSQL-32B sees a decline of 0.169. These results underscore the challenges in Spider $\mathbb{S}$ and reveal that current models are limited in their ability to reliably perform multi-SQL generation.

7.5 Execution Score (ES)

Compared to Execution Accuracy (EX), Execution Score (ES) provides a more nuanced view of model performance. Take GPT-4o and CHES as examples: when comparing their results on Task $\mathbb{R}\mathbb{O}$ and Task $\mathbb{R}\mathbb{M}$, we observe that the drop in EX is much larger than the drop in ES. This indicates that although these models fail to generate a fully correct

Metric	RoBERTa	Qwen2.5-0.5B	Qwen2.5-7B
Accuracy	0.8672	0.9909	0.9960
Precision	0.8273	0.9839	0.9931
Recall	0.9962	0.9981	0.9991
F1 Score	0.9039	0.9910	0.9961

Table 3: Results of three router models on the combined test set of Spider and SpiderS (2,147 each). Qwen2.5 models are from the Qwen2.5-Instruct family.

set of SQL queries in TaskRM (resulting in an EX of 0), they are still able to generate some individual SQLs that are correct, which leads to a non-zero ES. In contrast, Mistral-Large experiences substantial drops in both EX and ES, suggesting that it is not robust to prompt modifications and often fails to generate any correct SQL queries in TaskRM.

A similar pattern emerges in TaskSM: for most models, ES is notably higher than EX. This suggests that while models still struggle to generate all required SQL correctly, they are often capable of producing some correct queries. These partial successes demonstrate an initial grasp of multi-SQL generation, though significant progress is still needed to achieve complete correctness.

7.6 Router Effectiveness

Table 3 presents the performance of three router models trained to determine whether a given question should be answered using a single SQL query (label 0) or multiple SQL queries (label 1). Notably, even the lightweight Qwen2.5-0.5B model performs remarkably well, achieving an accuracy of 0.9909 and an F1 score of 0.9910, which is only marginally lower than that of its much larger 7B counterpart. We provide further analysis of the router’s decision basis in Appendix H.

All models exhibit exceptionally high recall, indicating that true multi-SQL cases (label 1) are rarely misclassified as single-SQL. This distinction holds significant practical importance. Misclassifying a multi-SQL query as single-SQL can impair downstream model performance by forcing complex tasks into a single-query framework. In contrast, misclassifying a single-query task as multi-query is generally more tolerable, as it mainly increases generation and execution cost. Moreover, even when multiple queries are not strictly necessary, multi-query outputs enable correctness to be assessed at the level of individual queries: producing some correct SQL queries is more informative than producing none or only incorrect ones.

Models	Without Router		With Router	
	EX	ES	EX	ES
GPT-4o	0.593	0.695	0.626	0.712
o3-mini	0.653	0.717	0.659	0.732
Qwen2.5-Coder-7B	0.583	0.673	0.639	0.718
CHESS	0.531	0.666	0.650	0.728
Qwen2.5-Coder-7B [†]	<u>0.759</u>	<u>0.813</u>	-	-
XiYanSQL-32B	0.756	0.802	0.771	0.831

Table 4: Performance of different models with and without the Qwen2.5-0.5B-Instruct router on the combined test set of Spider_{small} and SpiderS_{small} (1,000 examples each). [†] denotes models that have been jointly fine-tuned on both Spider and SpiderS.

The router does not generate SQL and is not exposed to end users. Instead, it serves as an auxiliary component that helps existing models better handle both single- and multi-SQL generation. Table 4 shows how the addition of the lightweight Qwen2.5-0.5B-Instruct router affects model performance. In this setup, models are not told in advance whether to generate a single or multiple SQL queries. Without the router, all models rely on a generic prompt ($P_{\text{prompt}_{1VN}}$), which can be ambiguous. When the router is used, it predicts whether the input should be answered with a single or multiple SQL queries, and selects the appropriate prompt (P_{prompt_1} or P_{prompt_N}) accordingly.

Across all models, the router brings consistent performance gains. The largest improvements are observed in CHESS and Qwen2.5-Coder-7B, which struggle under the $P_{\text{prompt}_{1VN}}$ but benefit substantially when given explicit guidance from the router. With the help of a clear prompt, their performance approaches that of o3-mini without a router. The multi-agent-based CHESS model, when equipped with a router, nearly matches the performance of the router-augmented o3-mini. For reasoning LLMs such as o3-mini, which are already effective at handling $P_{\text{prompt}_{1VN}}$, the router still yields smaller gains.

Compared to direct fine-tuning, the router-based approach is more adaptable: it can be seamlessly combined with different base models and leverage their respective strengths. For example, XiYanSQL-32B, when paired with the router, achieves the best performance, outperforming the directly fine-tuned Qwen2.5-Coder-7B. In particular, Qwen2.5-Coder-7B was fine-tuned exclusively using $P_{\text{prompt}_{1VN}}$, making it incompatible with router-guided prompting; as a result, we did not apply the router to it.

8 Conclusion

We introduce the Text-to-(Multi)-SQL task and the SpiderS benchmark, which break the limitation of single-query assumptions by supporting multiple SQLs. We propose a more flexible evaluation protocol that accepts any prediction covering the required information. Experiments show that most models are sensitive to prompt variations, lack robustness, and struggle with multi-SQL generation. To address this, we introduce a lightweight router that accurately decides whether a question requires one or multiple SQL, and significantly improves model performance. Text-to-(Multi)-SQL better aligns with real-world needs and opens up a promising direction for future research.

Limitations

SpiderS is constructed by extending the Spider benchmark and therefore inherits Spider’s database schemas and question style. We do not construct a full SpiderS-style extension for other datasets such as BIRD or Spider-2. As a cross-dataset diagnostic, we evaluate router generalization on constructed BIRD examples in Appendix H; however, the multi-intent BIRD instances are constructed rather than naturally collected from real users.

Due to API and token cost constraints, some full-test-set evaluations are not executed. We therefore report small-set results for all comparable configurations and verify that the 1,000-example subset preserves model ranking and performance trends. In addition, fine-tuned models are evaluated primarily under their training prompt format, leaving a broader cross-prompt robustness analysis for future work.

A large portion of the training data is generated using GPT-4o and filtered through automatic checks, model-based semantic validation, and selective manual review. Despite conservative filtering and the additional analyses in Appendix D, the generated data may still contain artifacts or distributional biases, and some residual noise may remain.

Ethical Considerations

This work constructs a research benchmark for text-to-(single-or-multi)-SQL by extending the public Spider benchmark.

Data provenance and privacy. The benchmark is built from existing benchmark databases rather than real user logs. We do not collect personal user

data. Any release of derived artifacts is intended to follow the original dataset’s licensing and redistribution constraints.

Synthetic generation and bias. Some examples are produced with a large language model and then filtered/verified. Synthetic data may contain artifacts or distributional biases and may not fully reflect real user intents. We mitigate these risks through multi-stage verification (automated checks and human review) and discuss remaining limitations in the paper.

Human review and misuse. Human reviewers validate generated examples with schema-consistency and correctness guidelines. Like other text-to-SQL systems, models trained on this benchmark could be misused if connected to sensitive databases; deployments should apply standard access control, auditing, and least-privilege safeguards.

Acknowledgments

This work was supported in part by the Economic and Social Research Council (ESRC) through the CORDIAL-AI project (grant ES/Z502911/1). We also thank Microsoft Research - Accelerating Foundation Models Research program, for the provision of Azure resources to run some of the LLMs used in the experiments in this paper. The funders had no role in the study design, data collection and analysis, decision to publish, or preparation of the manuscript.

References

- Shanza Abbas, Muhammad Umair Khan, Scott Uk-Jin Lee, Asad Abbas, and Ali Kashif Bashir. 2022. [A review of nlidb with deep learning: Findings, challenges and open issues](#). *IEEE Access*, 10:14927–14945.
- Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.
- Adithya Bhaskar, Tushar Tomar, Ashutosh Sathe, and Sunita Sarawagi. 2023. [Benchmarking and improving text-to-SQL generation under ambiguity](#). In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7053–7074, Singapore. Association for Computational Linguistics.

- Asim Biswal, Liana Patel, Siddarth Jha, Amog Kamsetty, Shu Liu, Joseph E. Gonzalez, Carlos Guestrin, and Matei Zaharia. 2024. [Text2sql is not enough: Unifying ai and databases with tag](#).
- Shuaichen Chang, Jun Wang, Mingwen Dong, Lin Pan, Henghui Zhu, Alexander Hanbo Li, Wuwei Lan, Sheng Zhang, Jiarong Jiang, Joseph Lilien, Steve Ash, William Yang Wang, Zhiguo Wang, Vittorio Castelli, Patrick Ng, and Bing Xiang. 2023. [Dr.spider: A diagnostic evaluation benchmark towards text-to-SQL robustness](#). In *The Eleventh International Conference on Learning Representations*.
- Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, et al. 2025. [Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities](#).
- Xiang Deng, Ahmed Hassan Awadallah, Christopher Meek, Oleksandr Polozov, Huan Sun, and Matthew Richardson. 2021. [Structure-grounded pretraining for text-to-SQL](#). In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1337–1350, Online. Association for Computational Linguistics.
- Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186.
- Mingwen Dong, Nischal Ashok Kumar, Yiqun Hu, Anuj Chauhan, Chung-Wei Hang, Shuaichen Chang, Lin Pan, Wuwei Lan, Henghui Zhu, Jiarong Jiang, Patrick Ng, and Zhiguo Wang. 2025. [PRACTIQ: A practical conversational text-to-SQL dataset with ambiguous and unanswerable queries](#). In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 255–273, Albuquerque, New Mexico. Association for Computational Linguistics.
- Yujian Gan, Xinyun Chen, Qiuping Huang, and Matthew Purver. 2022. [Measuring and improving compositional generalization in text-to-SQL via component alignment](#). In *Findings of the Association for Computational Linguistics: NAACL 2022*, pages 831–843, Seattle, United States. Association for Computational Linguistics.
- Yujian Gan, Xinyun Chen, Qiuping Huang, Matthew Purver, John R. Woodward, Jinxia Xie, and Pengsheng Huang. 2021. [Towards robustness of text-to-SQL models against synonym substitution](#). In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 2505–2515, Online. Association for Computational Linguistics.
- Yujian Gan, Matthew Purver, and John R. Woodward. 2020. [A review of cross-domain text-to-SQL models](#). In *Proceedings of the 1st Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 10th International Joint Conference on Natural Language Processing: Student Research Workshop*, pages 108–115, Suzhou, China. Association for Computational Linguistics.
- Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023. [Text-to-sql empowered by large language models: A benchmark evaluation](#).
- Yingqi Gao, Yifu Liu, Xiaoxia Li, Xiaorong Shi, Yin Zhu, Yiming Wang, Shiqi Li, Wei Li, Yuntao Hong, Zhiling Luo, Jinyang Gao, Liyu Mou, and Yu Li. 2024. [A preview of xian-sql: A multi-generator ensemble framework for text-to-sql](#). *arXiv preprint arXiv:2411.08599*.
- Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, et al. 2024. [The llama 3 herd of models](#).
- 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](#).
- Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. [Lora: Low-rank adaptation of large language models](#).
- Nan Huo, Xiaohan Xu, Jinyang Li, Per Jacobsson, Shipei Lin, Bowen Qin, Binyuan Hui, Xiaolong Li, Ge Qu, Shuzheng Si, Linheng Han, Edward Alexander, Xintong Zhu, Rui Qin, Ruihan Yu, Yiyao Jin, Feige Zhou, Weihao Zhong, Yun Chen, Hongyu Liu, Chenhao Ma, Fatma Ozcan, Yannis Papakonstantinou, and Reynold Cheng. 2026. [BIRD-INTERACT: Re-imagining text-to-SQL evaluation via lens of dynamic interactions](#). In *The Fourteenth International Conference on Learning Representations*.
- Radu Cristian Alexandru Iacob, Florin Brad, Elena-Simona Apostol, Ciprian-Octavian Truică, Ionel Alexandru Hosu, and Traian Rebedea. 2020. [Neural approaches for natural language interfaces to databases: A survey](#). In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 381–395, Barcelona, Spain (Online). International Committee on Computational Linguistics.
- George Katsogiannis-Meimarakis and Georgia Koutrika. 2023. A survey on deep learning approaches for text-to-sql. *The VLDB Journal*, 32(4):905–936.

- Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023a. Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 13067–13075.
- Jinyang Li, Binyuan Hui, Ge Qu, Jiayi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2023b. 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*, 36.
- Jinyang Li, Xiaolong Li, Ge Qu, Per Jacobsson, Bowen Qin, Binyuan Hui, Shuzheng Si, Nan Huo, Xiaohan Xu, Yue Zhang, Ziwei Tang, Yuanshuai Li, Florensia Widjaja, Xintong Zhu, Feige Zhou, Yongfeng Huang, Yannis Papakonstantinou, Fatma Ozcan, Chenhao Ma, and Reynold Cheng. 2025. [SWE-SQL: Illuminating LLM pathways to solve user SQL issues in real-world applications](#). In *Advances in Neural Information Processing Systems*, volume 38, pages 97085–97120.
- Haoran Luo, Haihong E, Zichen Tang, Shiyao Peng, Yikai Guo, Wentai Zhang, Chenghao Ma, Guanting Dong, Meina Song, Wei Lin, Yifan Zhu, and Anh Tuan Luu. 2024. [Chatkbqa: A generate-then-retrieve framework for knowledge base question answering with fine-tuned large language models](#). In *Findings of the Association for Computational Linguistics ACL 2024*, page 2039–2056. Association for Computational Linguistics.
- Rui Ma, Kai Zhang, Zhenying He, Yinan Jing, X Sean Wang, and Zhenqiang Chen. 2025. Chase: A native relational database for hybrid queries on structured and unstructured data. *arXiv preprint arXiv:2501.05006*.
- Farhad Nooralahzadeh, Yi Zhang, Ellery Smith, Sabine Maennel, Cyril Matthey-Doret, Raphaël De Fondeville, and Kurt Stockinger. 2024. [StatBot.Swiss: Bilingual open data exploration in natural language](#). In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 5486–5507, Bangkok, Thailand. Association for Computational Linguistics.
- OpenAI, Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, et al. 2024. [Gpt-4o system card](#).
- Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O. Arik. 2024. [Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql](#).
- Mohammadreza Pourreza and Davood Rafiei. 2023. [Din-sql: Decomposed in-context learning of text-to-sql with self-correction](#).
- Jiexing Qi, Jingyao Tang, Ziwei He, Xiangpeng Wan, Yu Cheng, Chenghu Zhou, Xinbing Wang, Quanshi Zhang, and Zhouhan Lin. 2022. Rasat: Integrating relational structures into pretrained seq2seq model for text-to-sql. *arXiv preprint arXiv:2205.06983*.
- Bowen Qin, Binyuan Hui, Lihan Wang, Min Yang, Jinyang Li, Binhua Li, Ruiying Geng, Rongyu Cao, Jian Sun, Luo Si, Fei Huang, and Yongbin Li. 2022. [A survey on text-to-sql parsing: Concepts, methods, and future directions](#).
- Qwen Team, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiayi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. [Qwen2.5 technical report](#).
- Cedric Renggli, Ihab F Ilyas, and Theodoros Rekatsinas. 2025. Fundamental challenges in evaluating text2sql solutions and detecting their limitations. *arXiv preprint arXiv:2501.18197*.
- Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. [Code llama: Open foundation models for code](#).
- Irina Saparina and Mirella Lapata. 2024. [Ambrosia: A benchmark for parsing ambiguous questions into database queries](#).
- Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. Picard: Parsing incrementally for constrained auto-regressive decoding from language models. *arXiv preprint arXiv:2109.05093*.
- Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. Chess: Contextual harnessing for efficient sql synthesis. *arXiv preprint arXiv:2405.16755*.
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.
- Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921.

Weixu Zhang, Yifei Wang, Yuanfeng Song, Victor Jun-qiu Wei, Yuxing Tian, Yiyan Qi, Jonathan H. Chan, Raymond Chi-Wing Wong, and Haiqin Yang. 2024. [Natural language interfaces for tabular data querying and visualization: A survey](#).

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyang Luo. 2024. [LlamaFactory: Unified efficient fine-tuning of 100+ language models](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 400–410, Bangkok, Thailand. Association for Computational Linguistics.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.

Yuhang Zhou, Yu He, Siyu Tian, Yuchen Ni, Zhangyue Yin, Xiang Liu, Chuanjun Ji, Sen Liu, Xipeng Qiu, Guangnan Ye, et al. 2024. r3-nl2gql: A model coordination and knowledge graph alignment approach for nl2gql. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 13679–13692.

Xiaohu Zhu, Qian Li, Lizhen Cui, and Yongkang Liu. 2024. [Large language model enhanced text-to-sql generation: A survey](#).

A Extended Related Work

A.1 Natural Language Interface to Databases

Research on Natural Language Interfaces to Databases (NLIDBs) aims to let users query structured data in natural language, with Text-to-SQL (Katsogiannis-Meimarakis and Koutrika, 2023) being the most prominent task. It focuses on mapping questions to SQL by interpreting intent, aligning with schema, and handling complex constructs like joins and subqueries (Gan et al., 2020). Recent advancements in deep learning, particularly transformer (Vaswani et al., 2017) and large language models (Devlin et al., 2019; Achiam et al., 2023), have significantly improved performance (Scholak et al., 2021; Qi et al., 2022; Li et al., 2023a; Gao et al., 2024) in this area. In addition to Text-to-SQL, other tasks such as Text-to-SPARQL (Luo et al., 2024) aim to extend NLIDBs systems to semantic web databases. There is a growing emphasis on improving related models’ ability to handle more complex domain-specific queries and to provide accurate context-sensitive responses.

A.2 Popular Text-to-SQL Datasets

Text-to-SQL is a key topic in human-computer interaction, prompting widespread research. Sev-

eral benchmark datasets have been introduced to support the conversion of natural language into SQL. WikiSQL (Zhong et al., 2017) is a large-scale dataset for converting natural language questions into SQL queries, containing 80,000 question-SQL pairs over tables extracted from Wikipedia. Spider (Yu et al., 2018) is a large-scale, cross-domain dataset for SQL query generation, with a focus on multi-table queries, complex conditions, and aggregation. BIRD (Li et al., 2023b) is a benchmark for evaluating Text-to-SQL models on large-scale, real-world databases, emphasizing value retrieval, complex reasoning, and execution efficiency. NL2GQL (Zhou et al., 2024) is a dataset that pairs natural language questions with corresponding GQL queries, designed to help develop models for querying graph databases, including tasks like node selection, graph traversal, and pathfinding. However, all of the above datasets focus on converting a single natural language request into a single SQL query. They overlook realistic scenarios where a single question is better answered through multiple SQL queries.

A.3 Extensions of Spider and BIRD

Several studies extend Spider and BIRD to evaluate more realistic or challenging Text-to-SQL settings. Spider-Syn (Gan et al., 2021) evaluates robustness to synonym substitution, while Spider-SS/Spider-CG (Gan et al., 2022) focuses on compositional generalization. Spider-Realistic (Deng et al., 2021) reduces explicit schema mentions, and Dr.Spider (Chang et al., 2023) systematically evaluates robustness under question, schema, and SQL perturbations. Extensions of BIRD include SWE-SQL (BIRD-CRITIC) (Li et al., 2025) for realistic SQL debugging and BIRD-Interact (Huo et al., 2026) for dynamic database interactions. However, these benchmarks do not explicitly formulate the decision of whether a single user request should be answered by one or multiple SQL queries as a dedicated routing problem.

A.4 Ambiguity and Disambiguation vs. Text-to-(Multi)-SQL

Several recent NL2SQL benchmarks focus on *ambiguity* or underspecification, where a single natural language question admits multiple plausible interpretations and thus multiple alternative SQL programs, such as AmbiQT (Bhaskar et al., 2023), AMBROSIA (Saparina and Lapata, 2024), and PRACTIQ (Dong et al., 2025). In this line of work,

the core challenge is disambiguation, either by generating multiple candidate SQLs or by interacting with users to clarify intent.

Another related direction studies relaxed execution-based evaluation. For example, Stat-Bot.Swiss (Nooralahzadeh et al., 2024) argues that strict execution accuracy may falsely penalize predictions that do not exactly match the gold execution result but still satisfy the user’s information need. It therefore introduces relaxed variants such as EA_{soft} and $EA_{partial}$, which accept certain subset relations between a single gold result set and a single predicted result set. Our protocol shares this high-level motivation: both aim to avoid penalizing semantically useful predictions solely because they differ from a single gold SQL execution.

However, the two settings differ in their evaluation unit and scope. When both the gold and prediction contain a single SQL query, our execution match is conceptually close to EA_{soft} , since both can accept predictions with additional non-essential columns. In contrast, SpiderS targets clear composite requests that may require multiple information units. Therefore, our protocol evaluates a set of predicted SQL executions against one or more valid gold SQL solution sets, rather than comparing only one predicted result set with one gold result set. This enables us to score single-SQL, multi-SQL, and decomposable composite-SQL predictions under the same framework. ES measures partial information coverage over query sets, while EX counts an example as correct only when all required information units are covered. Thus, SpiderS generalizes relaxed execution matching from single-result tolerance to decomposition-aware information coverage.

SpiderS addresses a fundamentally different setting that is *not related to ambiguity*. All SpiderS questions are designed to be clear and unambiguous, but may contain multiple information requirements that cannot be naturally expressed within a single SQL query. During annotation, potentially ambiguous questions are explicitly revised or disambiguated using cues such as “first ... then ...” or “separately list ...”, ensuring each question has a well-defined target. The goal of SpiderS is coverage via decomposition: producing one SQL or a set of SQLs that jointly retrieve all required information, rather than generating multiple competing interpretations.

A.5 Limitations of Text-to-SQL Systems

Recent studies have advanced Text-to-SQL systems from several complementary angles. However, it’s still hard for these methods to address complex tasks. Biswal et al. (2024) integrate Retrieval-Augmented Generation (RAG) with database tables to propose a unified, general-purpose paradigm for answering natural language questions over databases, aiming to provide a better user experience than traditional Text-to-SQL approaches. Likewise, CHASE (Ma et al., 2025) enhances hybrid query execution by bridging structured and unstructured data. Renggli et al. (2025) analyze the limitations of Text-to-SQL from an evaluation perspective, which can lead to both prediction and evaluation errors. However, none of these papers involved Text-to-(Multi)-SQL.

B Detailed Theoretical Analysis

We provide a qualitative sequence-probability intuition followed by an idealized complexity-based analysis to explain why multi-SQL generation can be easier than composing a single, complex SQL query. Together, they capture two observations that align with our empirical results: (i) forcing multiple, weakly related sub-tasks into a single SQL query can introduce additional structural complexity, and (ii) this increased complexity may produce less common local token compositions that are more difficult for language models to generate reliably.

Sequence-probability intuition. As a qualitative intuition, complex single-SQL programs often combine local SQL patterns in less frequently observed ways than several simpler queries. For example, a single statement may need to intertwine aggregation, filtering, nested subqueries, or UNION-style constructs in order to represent multiple information needs within one result structure. Such combinations may be less familiar to a language model than the simpler local structures appearing in independently generated SQL queries.

For many questions in SpiderS, including the example in Figure 5, a multi-SQL solution naturally decomposes the user request into several relatively simple and weakly coupled queries. By contrast, the corresponding single-SQL solution may require additional composition to combine heterogeneous objectives into a single statement.

However, token probabilities in an autore-

gressive language model are strongly context-dependent. We therefore do not assume a general monotonic relationship between the number of rare or unseen local patterns and the probability of an entire SQL sequence. We treat this observation only as a qualitative intuition and rely on the following complexity-based analysis for our formal argument.

Complexity-based view. We next formalize the intuition that composing a single complex SQL query can be structurally harder than composing multiple simpler ones.

Let $C(q)$ be a non-negative complexity measure for a SQL query q , for example a weighted combination of abstract syntax tree size, number of joins, nesting depth, and the presence of advanced constructs such as `GROUP BY` and `HAVING`.

For the following idealized analysis, we use an exponential surrogate model in which the probability of correctly generating a query decreases with its complexity:

$$P(\text{output } q \text{ correctly}) = e^{-\alpha C(q)}, \quad (8)$$

for some $\alpha > 0$.

Consider a question whose intent can be satisfied either by a multi-SQL solution

$$Q^{(\text{multi})} = \{q_1, q_2, \dots, q_m\}, \quad (9)$$

or by a single-SQL solution $q^{(\text{single})}$ that combines the same information requirements into one statement.

For many SpiderS examples, each q_i is a relatively simple query, whereas $q^{(\text{single})}$ may require additional joins, nested subqueries, `UNION`-style constructs, or other composition mechanisms to encode the same information in one result structure. We model this additional composition overhead using the assumption

$$C(q^{(\text{single})}) \geq \sum_{i=1}^m C(q_i) + \Delta, \quad (10)$$

for some $\Delta > 0$.

If we further assume that the correctness events for the individual queries q_i are independent under this idealized model, then under the surrogate model in Equation (8), the probability of correctly

generating the complete multi-SQL solution is

$$P_{\text{multi}} = \prod_{i=1}^m P(\text{output } q_i \text{ correctly}) \quad (11)$$

$$= \prod_{i=1}^m e^{-\alpha C(q_i)} \quad (12)$$

$$= e^{-\alpha \sum_{i=1}^m C(q_i)}. \quad (13)$$

The probability of correctly generating the corresponding single-SQL solution is

$$P_{\text{single}} = e^{-\alpha C(q^{(\text{single})})}. \quad (14)$$

Using the complexity-gap assumption in Equation (10), we obtain

$$\frac{P_{\text{multi}}}{P_{\text{single}}} = \exp\left(-\alpha \sum_{i=1}^m C(q_i) + \alpha C(q^{(\text{single})})\right) \quad (15)$$

$$\geq \exp(\alpha \Delta) \quad (16)$$

$$> 1. \quad (17)$$

Proposition 1. *Under the idealized model in Equation (8), the complexity-gap assumption in Equation (10), and the independence assumption above, the probability of correctly generating a complete multi-SQL solution is strictly higher than that of correctly generating the corresponding single-SQL solution:*

$$P_{\text{multi}} \geq e^{\alpha \Delta} P_{\text{single}} > P_{\text{single}}. \quad (18)$$

Intuitively, when a single-SQL solution must encode multiple aspects of a request in one statement, it may incur additional composition overhead beyond the summed complexities of the corresponding simpler queries. Under the idealized model above, this additional complexity translates into a lower probability of successful generation. This analysis is not intended as a calibrated probabilistic model of modern LLMs, but rather as a formalization of the structural intuition motivating the Text-to-(Multi)-SQL setting.

C Additional Method Details

C.1 Task Definition

Text-to-(Multi)-SQL aims to convert a natural language question Q into one or multiple SQL queries $y_1, y_2, \dots$ that retrieve the correct results from a given database D . A database is formally defined as $D = \langle C, T \rangle$, where $C = \{c_1, c_2, \dots, c_m\}$

represents the set of column names and $T = \{t_1, t_2, \dots, t_m\}$ represents the set of table names. Given D and Q , the objective is to generate the most appropriate SQL query (or queries set), such that executing $y_1, y_2, \dots$ on D yields the expected answer. The query generation process can be expressed as

$$y = \{y_1, y_2, \dots\} = f(Q, D|\phi), \quad (19)$$

where $f(\cdot|\phi)$ is a model parameterized by ϕ , which maps the natural language question Q and database schema D to a set of SQL queries $\{y_1, y_2, \dots\}$.

C.2 SQL Generator Details

We fine-tune a SQL generator G built on language models to produce either a single SQL query or a sequence of queries. The model takes a natural language question Q and a database schema D as input, and produces an output y , as defined in Equation 19, where y is a JSON object containing one or more SQL queries, each stored under a distinct key (e.g., `sql_1`, `sql_2`, ...).

The SQL generator is fine-tuned using Low-Rank Adaptation (LoRA) (Hu et al., 2021) applied to all eligible linear projection layers in the model. The training objective is the standard autoregressive loss:

$$\mathcal{L}_{\text{LLM}}(\phi) = - \sum_{t=1}^{|y|} \log p_{\phi}(y_t | y_{<t}, Q, D), \quad (20)$$

where ϕ are the parameters of the LLM.

C.3 Router Details

To make existing models capable of handling Text-to-(Multi)-SQL generation, we introduce a lightweight router model R_{θ} that determines whether a natural language query requires single- or multi-SQL generation.

Training Data. The training set contains labeled triples:

$$\{(Q_i, D_i, g_i^*)\}_{i=1}^N, \quad g_i^* \in \{\text{single}, \text{multi}\}, \quad (21)$$

where g_i^* is the minimal generation strategy that yields a correct answer for input (Q_i, D_i) .

Model. We use RoBERTa-base and two Qwen2.5-Instruct variants (0.5B, 7B) for R_{θ} .

- **RoBERTa-base.** A classification head is placed over the [CLS] embedding to predict a binary label. The output logits $h_{\theta}(Q, D) \in \mathbb{R}^2$ are passed through a softmax. We minimize the cross-entropy between the predicted probabilities and the one-hot ground truth:

$$\mathcal{L}(\theta) = - \sum_{i=1}^N \sum_{g \in \{0,1\}} \mathbb{1}[g = g_i^*] \log R_{\theta}(g | Q_i, D_i). \quad (22)$$

- **Qwen2.5-Instruct.** A generative LLM is fine-tuned to output token "0" (for single query) or "1" (for multi-query) via next-token prediction. The training procedure and loss are the same as in Baseline 1.

D Extra Information about Dataset Construction

D.1 Question and SQL Annotation

Annotation Procedure. The annotation process began with an open-ended exploration: three computer science students (both undergraduate and graduate, and all proficient in SQL) were asked to freely annotate natural language questions and their corresponding SQL queries across multiple databases. At this stage, no predefined categories or patterns were enforced—annotators simply aimed to create complex, diverse, and realistic examples. After approximately 300 examples had been annotated, we analyzed the dataset and identified 14 recurring SQL construction patterns, such as user query rewrites, multi-intent merging, and schema-driven condition variations. These patterns were then formalized into a structured annotation guide, with each pattern accompanied by a definition, example questions, and corresponding SQLs. The guide ensured consistency in the remaining annotation process.

To promote diversity and coverage, annotators were instructed to label one representative example for each pattern in every target database. These high-quality examples later served as in-context demonstrations for GPT-4o during the data generation phase.

Question Clarity. Throughout the annotation process, we emphasized the importance of writing clear and unambiguous user questions, consistent with the original Spider dataset. Ambiguous questions—those that lacked sufficient constraints or

could yield multiple interpretations—were revised or explicitly disambiguated.

For example, the question “Find all the teacher names and also all the classrooms for math courses” can be read as either a combined query for teachers and classrooms in math courses, or as two separate queries: one for all teacher names, and one for classrooms specific to math courses. To prevent confusion, we rephrased such questions using disambiguating cues like “first ..., then ...” or “separately list ...”, clarifying intent and ensuring that each question had a well-defined SQL counterpart.

Annotation Validation. All manually annotated examples underwent a multi-step validation pipeline:

1. **Manual cross-checking.** Annotators reviewed each other’s examples to catch inconsistencies between the user request and SQL.
2. **Automatic script validation.** We used a script to check SQL syntax, keyword usage, pattern adherence, and schema consistency.
3. **GPT-4o semantic check.** Each example was further verified using GPT-4o to ensure that the natural language question accurately matched the SQL output.

If an issue was identified in any stage, the example was returned to the annotators for revision. Crucially, no examples were discarded during annotation; all flagged data underwent revision cycles. Only in the later data generation stage were outputs filtered for quality.

Through this process, we created a seed set of 890 high-quality annotated examples that were structurally diverse, semantically aligned, and representative of the 14 identified patterns.

D.2 Data Review

Automatic Validation Script. To ensure annotation consistency and reduce human error, we developed an automatic validation script that iteratively refined data quality through several systematic checks:

- **Syntactic correctness:** Ensured that SQL queries were syntactically correct and executable without errors.
- **Pattern matching:** Verified each SQL query strictly conformed to one of the predefined 14 SQL generation patterns without extraneous or missing clauses.

- **Schema alignment:** Confirmed that all table names and column references in the SQL queries matched exactly with those defined in the corresponding database schema.

Entries that failed any check were returned to annotators for correction or discarded if not repairable.

GPT-4o-Based Validation. In addition to rule-based checks, GPT-4o served as an advanced semantic validation tool. Given each user query and its corresponding SQL statement, GPT-4o performed the following validations:

1. Assessed whether the natural language query was clearly articulated and free from ambiguity.
2. Ensured logical consistency between the natural language query and its corresponding SQL query.
3. Identified potential discrepancies or ambiguities.

Any data flagged by GPT-4o underwent manual review for final resolution. This hybrid validation approach combined computational precision with human judgment, significantly enhancing the overall dataset robustness.

D.3 Data Generation with GPT-4o

To scale beyond the manually annotated seed set, we leveraged GPT-4o to generate additional question-SQL pairs. When a relevant in-context example—i.e., from the same database and pattern—was available, we adopted a **one-shot prompting** strategy using it as a demonstration; otherwise, we used **zero-shot prompting** to promote diversity and generalization.

Since not all generated examples would meet our quality standards, we deliberately over-generated data during this stage. All examples were passed through the same three-stage validation pipeline described in the annotation section—comprising automatic structural checks, GPT-4o-based semantic alignment, and selective manual review. Only those examples that successfully passed all validation stages were retained. Unlike in manual annotation, where problematic cases were revised and refined, generated samples that failed validation were discarded directly to streamline the process. This conservative filtering allowed us to balance efficiency with quality control.

After filtering, we selected a final subset of high-quality generated examples such that the overall

dataset size closely matched that of the original Spider benchmark, while preserving comparable complexity, coverage, and clarity.

D.4 Additional Data Quality Analysis

Human audit. All generated examples in the test set that passed GPT-4o validation were subsequently reviewed by human annotators. Approximately 1% were removed during this review. This should be interpreted as the observed human-review rejection rate rather than the residual error rate, since some examples were removed because they were overly complex, difficult to understand, or insufficiently clear rather than semantically incorrect. Before human review, the conservative GPT-4o validation step filtered out approximately 25% of candidate examples, including examples containing even minor ambiguity.

Source-identification analysis. To examine possible linguistic artifacts, we conducted a blinded source-identification experiment on 200 balanced human-written and GPT-4o-generated examples with two independent annotators. The annotators achieved accuracies of 61.0% (95% CI: 54.1%–67.5%) and 45.5% (95% CI: 38.7%–52.4%), respectively. Inter-annotator agreement was close to chance (48.5%, Cohen’s $\kappa = 0.004$), while mean item-level accuracy was 53.2% (bootstrap 95% CI: 48.5%–58.0%). These results suggest that the human-written and generated questions were not reliably distinguishable by human annotators.

E Data Generation Patterns

To support structured and scalable data generation, we formalized a set of 14 data generation patterns based on the initial 300 manually annotated examples. Each pattern includes a clear definition, representative user questions, and corresponding SQL queries, and was used to guide both subsequent manual annotation and GPT-4o-based generation. These patterns serve as a principled framework for expanding the dataset in a controlled, interpretable, and high-coverage manner. We detail the 14 generation patterns below, illustrating how each introduces distinct forms of linguistic or structural variation. These pattern IDs are also used in Section K to analyze model performance and identify which question types are more error-prone.

1. Reverse Selection in One-to-Many Relationships

To introduce compositional complexity

into text-to-SQL pairs, we employ reverse selection on one-to-many relationships. This approach ensures that a given natural language query is naturally decomposed into two simpler SQL queries to retrieve the requested information, as opposed to a single SQL query.

In relational databases, a one-to-many relationship exists when an entity in one table (e.g., departments) is associated with multiple entities in another table (e.g., teachers). The department table serves as the parent (one), while the teacher table acts as the child (many), with the foreign key in the teacher table referencing the primary key in the department table.

Consider the following example, where the original query is:

Find the names of teachers who are older than 40.

This query can be answered using a single SQL statement:

```
SELECT name FROM teacher WHERE age > 40;
```

To enforce multi-query execution, we introduce a reverse selection constraint by modifying the query as follows:

Find the names of teachers who are older than 40 and the departments of the remaining teachers.

Here, the teacher name and the department name of other teachers are semantically incompatible in a single SQL query, necessitating at least two queries:

```
SELECT name FROM teacher WHERE age > 40;
```

```
SELECT department_name FROM teacher WHERE age <= 40;
```

This transformation ensures that a single query is insufficient to retrieve all requested information, thereby enriching the complexity of text-to-(Multi)-SQLs pairs.

2. Reverse Selection in One-to-One Relationships

Unlike the one-to-many relationships discussed previously, a one-to-one relationship occurs when each record in a table corresponds to a single record in another table. In this case, reverse selection cannot involve direct inclusion of attributes from the same table,

as such queries can always be answered using a single SQL statement. Therefore, to enforce compositional complexity, reverse selection in one-to-one relationships must ensure that the two queries retrieve different types of information.

A critical constraint of one-to-one reverse selection is that the secondary query must retrieve different attributes. If the modified query asks for the same attribute from different records, it can always be rewritten as a single SQL query, rendering the transformation invalid. So we need to ensure the queries ask for logically distinct attributes. Consider the following transformation:

Find the names of teachers who are older than 40 and retrieve the average age of the remaining teachers.

This results in the following SQL queries:

```
SELECT name FROM teacher WHERE age > 40;
```

```
SELECT AVG(age) FROM teacher WHERE age <= 40;
```

Here, the first query retrieves individual teacher names, while the second retrieves an aggregate statistic, making them more difficult to combine naturally into a single SQL statement.

- 3. Reverse Selection in One-to-One Relationships Without Aggregation** One-to-one reverse selection can be further constrained by eliminating aggregation functions, making the compositional SQL queries more structurally distinct while still requiring multiple SQL queries to fully answer the question. This ensures that query decomposition is necessary but does not rely on aggregate statistics to enforce multi-query generation.

4. Distinguishing Separate Entities for Independent Queries

In the process of constructing diverse SQL queries, one effective approach is to generate questions that target distinct entities within the database. At first glance, it may seem intuitive that retrieving different types of information always necessitates multiple SQL queries. However, this is not necessarily the case. When two entities are inherently linked

through a foreign key relationship, it is often possible to retrieve both pieces of information using a single query. For instance, if the original query asks for the names of all teachers, it can be extended to also request their department names without requiring an additional SQL statement. Since teachers are linked to departments via a foreign key, a simple join operation suffices to bring both sets of information together within a single query.

However, to ensure that two distinct queries are required rather than one combined query, the entities being retrieved must not only belong to different tables but also be subject to independent constraints. Consider the case where we modify the original question from "Find the names of teachers older than 40" to "Find the names of teachers older than 40 and the names of departments with more than 40 members." This slight modification forces the retrieval of two independent sets of information—one about teachers filtered by age, and another about departments filtered by the number of members. Since these conditions apply to separate tables and do not share a common join path that would allow them to be merged into a single SQL query, it becomes necessary to generate two distinct SQL statements.

Find the names of teachers who are older than 40 and the names of departments with more than 40 people.

This question introduces conditions on different attributes from separate tables, making it impossible to construct a single SQL query that answers both parts. Instead, two independent queries must be generated:

```
SELECT name FROM teacher WHERE age > 40;
```

```
SELECT name FROM department WHERE num_people > 40;
```

5. Asymmetric Query Expansion: One Condition, One Open Query

In contrast to the previous method, which ensured that two distinct objects were queried with their own independent constraints, this method introduces an asymmetry: one of the SQL queries retains a specific condition while the other remains unrestricted. This subtle shift in design changes the way queries

are structured and how they interact with the database schema. Instead of ensuring that both queries are filtered by independent constraints, one query maintains a filtering condition while the other remains a general retrieval of all records from a related table.

To illustrate, consider an original query asking for "Find the names of all teachers." A straightforward extension would be to also retrieve the names of all departments, a task that could easily be achieved using a single SQL query through a simple join operation. However, if we modify this to "Find the names of teachers who are older than 40 and all department names," we introduce a structural asymmetry—one query applies a condition (teachers older than 40), while the other query retrieves all departments without filtering. This distinction ensures that the two queries must remain separate rather than being merged into a single SQL statement.

The process of generating such queries relies on selecting an appropriate example from existing data, ensuring that one query retains a condition while the other does not. The system systematically scans the dataset to identify candidate queries that fit this asymmetric pattern, filtering out those that contain multiple constraints or already require multiple SQL statements. Once an appropriate pair of queries is selected, the prompt is carefully constructed to emphasize the independence of the two queries while maintaining clarity in natural language.

6. Blending Aggregates with Individual Attributes: Merging Statistical Insights with Specifics

In natural language database queries, there is often a need to balance between querying individual records and summarizing trends. A common way to expand a query is by introducing an aggregation function alongside a standard selection. This method ensures that both specific details and overarching insights are retrieved simultaneously, making the query richer and more informative.

For example, consider the original query:

Find the names of teachers who are older than 40.

This query retrieves a list of individual teachers who meet the condition, focusing on granular details. However, by expanding the question to include an aggregation function, we can introduce a statistical perspective. The modified query becomes:

Find the names of teachers who are older than 40 and their average age.

Here, in addition to retrieving individual teacher names, the query also calculates the overall average age of teachers above 40. This transformation creates a dual perspective—while one part of the query extracts direct information from the database, the other part summarizes the information using an aggregation function.

To construct such queries correctly, it is crucial to ensure that both components operate on the same filtered dataset. This prevents inconsistencies, such as computing an average age across all teachers instead of just those older than 40. The corresponding SQL query would look like this:

```
SELECT name FROM teachers WHERE age > 40;
```

```
SELECT AVG(age) FROM teachers WHERE age > 40;
```

Alternatively, a single SQL statement can often achieve the same result using a GROUP BY clause:

```
SELECT name, (SELECT AVG(age) FROM teachers WHERE age > 40) AS avg_age FROM teachers WHERE age > 40;
```

Since this data generation method can also be completed with a SQL query, we introduced the `sql_0` statement to store all queries integrated into one SQL.

7. Querying Different Grouping Dimensions

Grouping data is a fundamental aspect of SQL queries, allowing us to derive meaningful insights by categorizing and summarizing records. While most queries focus on a single way to group data, an alternative approach introduces multiple GROUP BY clauses, effectively broadening the scope of the query. By shifting the lens through which data is aggregated, we can uncover deeper relationships

within the dataset that a single GROUP BY might miss.

This method is especially useful when dealing with datasets where different categorical attributes can provide valuable insights when analyzed separately. For example, consider a dataset of department management. The original question:

Tell me which states have at least 3 heads born there?

The corresponding SQL query would look like this:

```
SELECT born_state FROM head GROUP BY  
born_state HAVING COUNT(*) >= 3;
```

However, what if we want to explore a different way of grouping the data? We could also ask:

Tell me which states have at least 3 heads born there and which age groups have at least 3 heads?

The result is that we have to use 2 different SQL queries to answer this question:

```
SELECT age FROM head GROUP BY age  
HAVING COUNT(*) >= 3;
```

```
SELECT born_state FROM head GROUP BY  
born_state HAVING COUNT(*) >= 3;
```

8. Switching Between HAVING and LIMIT for Diverse Filtering

This method modifies SQL queries by ensuring that one query filters aggregated results using HAVING, while the other applies ORDER BY and LIMIT. Unlike the previous approach, which changed the GROUP BY column to introduce variation, this method focuses on switching between filtering patterns to generate distinct yet complementary queries.

If the original query includes LIMIT, the second query must replace it with a HAVING clause to introduce a different filtering method. Conversely, if the original query already uses HAVING, the new query must avoid it and instead adopt an ORDER BY and LIMIT structure. This ensures that the two queries cannot be merged into one, as they use fundamentally different ways of selecting data.

For example, if the original query retrieves the department with the most teachers using ORDER BY and LIMIT:

```
SELECT department FROM teachers GROUP  
BY department ORDER BY COUNT(*) DESC  
LIMIT 1;
```

The second query should instead filter using HAVING, such as selecting age groups where the number of teachers exceeds a threshold:

```
SELECT age FROM teachers GROUP BY age  
HAVING COUNT(*) > 2;
```

This variation ensures that both queries serve different analytical purposes while maintaining structural diversity. By alternating between HAVING and LIMIT, we create queries that offer different perspectives on the dataset, preventing them from being collapsed into a single SQL statement.

9. Reordering the Same Data from Different Perspectives

This method transforms a query that sorts data using ORDER BY into two separate queries, each sorting the same dataset by a different attribute. Unlike cases where LIMIT is involved, which restricts the number of results, this approach ensures that both queries return the full dataset, just presented in different orders.

Consider the original query:

Find the names of all teachers, sorted by age in descending order.

Original SQL Query:

```
SELECT name FROM teachers ORDER BY  
age DESC;
```

To introduce variety, we modify the query by adding an alternative sorting criterion. Instead of only ordering by age, we ask for two separate orderings—one by age and another by salary. The question can be transformed into:

Give me two separate lists of teacher names: one sorted by age in descending order and the other sorted by salary in descending order.

Corresponding SQL Queries:

```
SELECT name FROM teachers ORDER BY  
age DESC;
```

```
SELECT name FROM teachers ORDER BY  
salary DESC;
```

This transformation ensures that both results remain distinct and cannot be merged into a single SQL query.

10. **Sorting One Object While Leaving the Other Unordered**

This method builds upon the previous approach of modifying ORDER BY queries but introduces a crucial difference: instead of applying sorting criteria to both SQL queries, one query retains the ORDER BY clause while the other remains unsorted. The key idea is to introduce contrast—one dataset follows an explicit order while the other remains in its natural or default arrangement.

Consider the original query:

Find the names of all teachers, sorted by age in descending order.

Original SQL Query:

```
SELECT name FROM teachers ORDER BY age DESC;
```

To introduce variation while maintaining meaningful structure, we modify the query to introduce an additional unordered dataset. Instead of just listing teachers sorted by experience, we now also retrieve the departments they belong to, but without any explicit sorting applied to them.

Modified Question:

Find the names of all teachers, sorted by their ages in descending order, and separately list all department names.

Corresponding SQL Queries:

```
SELECT name FROM teachers ORDER BY age DESC;
```

```
SELECT department_name FROM departments;
```

This modification ensures that two separate results are produced, but unlike the previous approach (where both queries applied sorting in different ways), here only one result is ordered while the other remains unaffected. This distinction is important because it emphasizes that ORDER BY is not necessarily required for every SQL statement—it depends on the nature of the query and what needs to be compared or presented.

11. **Sorting One Object with LIMIT Condition While Leaving the Other Unordered**

This method is quite similar to the previous method (10) in that it requires an SQL query

with an ORDER BY clause and an extra query remaining unsorted. However, the ORDER BY query also includes an additional LIMIT condition. Furthermore, we intentionally select the additional SQL query to include a WHERE clause while minimizing the use of any aggregation functions. This is set to make the problem more challenging and to ensure the conditions of both 2 queries are clearly described to avoid any potential ambiguity.

Consider the original query:

Find the name of the oldest teacher

Original SQL Query:

```
SELECT name FROM teachers ORDER BY age DESC LIMIT 1;
```

To introduce variation while maintaining meaningful structure, we modify the query to introduce an additional unordered query with a WHERE clause. We intentionally select an additional query with a WHERE clause and avoid using any aggregation function.

Modified Question:

Find the name of the oldest teacher and the ages of all teachers whose names contain the letter "A".

Corresponding SQL Queries:

```
SELECT name FROM teachers ORDER BY age DESC LIMIT 1;
```

```
SELECT age FROM teachers WHERE name LIKE "%A%";
```

This modification is similar to the previous method, but introduces more conditions: One query is required to select only some results from the whole sorted list, while another query should return the unsorted results under the WHERE condition. This type of modification further challenges agents' ability to comprehend the nature of the questions.

12. **Dual Attribute Selection Sorted by the Same Attribute but in Reverse Orders with LIMIT Condition**

This method also requires an original SQL query to return results constrained by the LIMIT clause sorted by some attributes. We then add another query to select another attribute sorted by exactly the same attribute but in reverse order. Notice that the results are still

constrained by the LIMIT clause. Intuitively, one simple way to achieve this type of modification is to transform the statements in the original question that use superlative forms into their opposites (e.g., changing 'maximum' to 'minimum' and 'most' to 'least'). Also, we intentionally include some questions to change the number of results required by the LIMIT clause to make this type of modifications more variant and more challenging.

Consider the original query:

Show the name of the teacher with the highest salary.

Original SQL Query:

```
SELECT name FROM teachers ORDER BY salary DESC LIMIT 1;
```

Based on the the detailed explained pattern above, we introduce an additional query to select another attribute in the same table sorted by exactly the same attribute (e.g, salary) but in the reverse order (e.g, ASC). We can also change the number of results returned by changing the LIMIT clause (e.g, the lowest three salaries).

Modified Question:

Show the name of the teacher with the highest salary and the office numbers of the teachers who have the lowest three salaries.

Corresponding SQL Queries:

```
SELECT name FROM teachers ORDER BY salary DESC LIMIT 1;
```

```
SELECT office_no FROM teachers ORDER BY salary ASC LIMIT 3;
```

This modification requires the agent to output two distinct SQL queries to select two subsets of different attributes based on the sorted result of the same attribute but in reverse orders. This type of modification could pose challenges for some agents, as they must select two distinct attributes based on a condition and its nearly opposite form.

13. Dual Attribute Selection Sorted by the GROUP BY clause of the Primary Key but in Reverse Orders with LIMIT Condition

This method can be considered as a more challenging variant of the previous method (12) as

the main logic is quite similar. The only difference is that we use the GROUP BY clause of the primary key of the table to do sorting (e.g, GROUP BY id ORDER BY COUNT(*) DESC LIMIT 1). This task is considered more challenging as the original question requires the agent to understand the structure of the table and use the GROUP BY clause properly. A more complex task in the same setting to select two attributes sorted by some attributes of groups in reverse orders can result in greater confusion for agents.

Consider the original query:

Show the status of the city that has hosted the greatest number of competitions.

Original SQL Query:

```
SELECT T1.Status FROM city AS T1 JOIN farm_competition AS T2 ON T1.City_ID = T2.Host_city_ID GROUP BY T2.Host_city_ID ORDER BY COUNT(*) DESC LIMIT 1;
```

Based on the additional explanation of this variant, we select another proper attribute (e.g, the primary key itself) in the table and only changes the sorting order to the reverse one.

Modified Question:

Show the status of the city that has hosted the greatest number of competitions and the city ID with the fewest competitions.

Corresponding SQL Queries:

```
SELECT T1.Status FROM city AS T1 JOIN farm_competition AS T2 ON T1.City_ID = T2.Host_city_ID GROUP BY T2.Host_city_ID ORDER BY COUNT(*) DESC LIMIT 1;
```

```
SELECT T1.City_ID FROM city AS T1 JOIN farm_competition AS T2 ON T1.City_ID = T2.Host_city_ID GROUP BY T2.Host_city_ID ORDER BY COUNT(*) ASC LIMIT 1;
```

14. Question Generation with Three SQL Queries

We also introduce a small proportion of questions that require three SQL queries to address. These questions are generated based on some of the previous methods. As this could be a more challenging and difficult task for agents,

we only select some relatively simple methods to generate questions.

14.1 Combining One Original Question with One Already Generated Question Requiring Two SQL Queries

An intuitive way to generate questions requiring three SQL queries to address is to combine one original question from the dataset with one generated and well-evaluated question requiring two queries. To ensure that the final generated question is reasonable, relatively concise, and unambiguous, we intentionally select those original SQL queries that include a WHERE clause while excluding certain keywords that may form complex questions (such as LIMIT, GROUP, EXCEPT, INTERSECT, UNION, ORDER) and control the length of the selected SQL queries. Then we sample a well-evaluated question from our generated dataset and ask the agent to merge the two questions. The agent is also suggested to use some keywords like "separately" or "first, then," to avoid any potential ambiguous statements.

14.2 Querying Different Grouping Dimensions to Generate three queries

This method is quite similar to the previous method 7 as we still ask to group the table data based on different attributes separately, but now three different attributes are used instead of only two.

Two example generated questions are shown below:

1. *Show the number of teachers for each department, age, and country separately.*
2. *Find the department, the age group and the country group with the highest number of teachers.*

14.3 Reordering the Same Data from Three Different Perspectives

This method is quite similar to the previous method 9 as we still ask to provide lists of one single attribute sorted separately by different attributes, but this time we require three lists. Notice that each query sorted by each attribute can still be in ascending or descending order randomly.

An example generated questions are shown below:

Provide me with three lists of teacher names, sorted by age, department id and salary in descending order, respectively.

14.4 Two or Three Queries with DISTINCT

Previously our methods didn't consider the keyword DISTINCT to avoid any potential ambiguous statement. Here we ask the agent to generate a small subset of questions to ask for distinct values for selected columns only. The generated questions may require two or three queries to return only distinct/unique values for mentioned attributes.

A simple example question is shown below:

Provide me with distinct teacher name, unique department names, and distinct teacher ages, respectively.

F Implementation Details

All experiments were conducted using LLAMA Factory(Zheng et al., 2024), with LoRA-based fine-tuning utilizing the bf16 precision. The relevant hyperparameters were configured as follows:

- preprocessing_num_workers: 16
- per_device_train_batch_size: 1
- gradient_accumulation_steps: 4
- learning_rate: 1.0e-4
- num_train_epochs: 3.0
- lr_scheduler_type: cosine
- warmup_ratio: 0.1

Other parameters were set to the default values provided by LLAMA Factory. The experiments were conducted on single or multiple A100 GPUs (40GB/80GB). No hyperparameter tuning was performed, and each experiment was run only once.

G Efficiency Analysis of Multi-SQL Execution

A natural concern about multi-SQL is that executing multiple queries may introduce additional database-call overhead compared with a single SQL statement. To examine whether this overhead leads to a substantial efficiency penalty, we compare sequential gold multi-SQL execution with a functionally equivalent single-statement execution form.

For each example, we construct an equivalent single-statement form directly from the gold multi-SQL queries while preserving the original subquery

Metric	Value	Cells	#Cases	Single faster	Multi faster	Tie
Number of examples	2147	0–20	980	94.2%	3.6%	2.2%
Gold multi-SQL total time	2399.84 ms	21–100	929	94.0%	4.6%	1.4%
Equivalent single-statement total time	2540.61 ms	101–1000	180	55.6%	32.2%	12.2%
Relative speedup of equivalent form	0.9446×	1001–10000	35	0.0%	82.9%	17.1%
Gold multi-SQL mean time	1.1178 ms	>10000	23	0.0%	100.0%	0.0%
Equivalent single-statement mean time	1.1833 ms	Table 7: Efficiency comparison by the size of the Single-SQL output payload, measured as returned rows multiplied by returned columns. Cells represents the total number of cells returned by Single-SQL.				
Gold multi-SQL median time	0.0155 ms					
Equivalent single-statement median time	0.0121 ms					
Single faster cases / multi faster cases / tie	1861 / 223 / 63					
BIRD-style VES-like score (Li et al., 2023b)	114.86					

Table 5: Efficiency comparison between sequential gold multi-SQL execution and a functionally equivalent single-statement execution form. The VES-like score is computed as $\text{mean}(\sqrt{t_{\text{multi}}/t_{\text{single}}}) \times 100$, where values above 100 indicate that the single-statement form is faster on average after per-example normalization.

Returned rows	#Cases	Single faster	Multi faster	Tie
0–10	1495	95.5%	2.8%	1.7%
11–50	560	83.0%	13.2%	3.8%
51–1000	59	6.8%	67.8%	25.4%
>1000	33	0.0%	97.0%	3.0%

Table 6: Efficiency comparison by the number of returned rows.

logic. The constructed statement is then validated using the same evaluation metrics and execution procedure as the original multi-SQL queries, ensuring that it returns the same results as the gold execution. Therefore, this comparison controls for query semantics and focuses only on the physical execution strategy: executing multiple statements sequentially versus executing one equivalent statement. All efficiency experiments are conducted on a MacBook with an Apple M2 CPU.

Table 5 summarizes the results on the SpiderS test set (2,147 examples). Somewhat surprisingly, gold multi-SQL is slightly faster in total runtime: it takes 2399.84 ms, compared with 2540.61 ms for the equivalent single-statement form. Thus, multi-SQL is slightly faster overall, with the single-statement form requiring $1.06\times$ the total runtime. However, the per-example comparison shows the opposite trend for most cases: the single-statement form is faster on 1,861 examples, multi-SQL is faster on 223 examples, and 63 examples are tied. The median time also favors the single-statement form, with 0.0121 ms versus 0.0155 ms for multi-SQL. Consistent with this per-example trend, a BIRD-style VES-like score (Li et al., 2023b), computed as the mean of $\sqrt{t_{\text{multi}}/t_{\text{single}}} \times 100$, is 114.86, indicating that the single-statement form is faster on average after normalizing per example.

This mixed result suggests a heavy-tailed efficiency pattern. For most lightweight examples, packaging multiple subqueries into one statement can reduce repeated statement and cursor boundaries, making the single-statement form faster. However, a small number of expensive cases dominate the total runtime, where the single-statement packaging may introduce additional cost or lead to less favorable execution behavior. Thus, these results do not show that either form is uniformly more efficient. Instead, they indicate that execution efficiency depends on the physical execution strategy and the distribution of queries.

G.1 When Should We Use Multi-SQL Instead of Single-SQL?

We further analyze when executing multiple SQL queries separately is preferable to merging them into a single SQL query. In our setting, the single-query formulation combines the outputs of multiple SQL queries using `UNION ALL`, together with extra `NULL` padding cells to align heterogeneous result schemas. Therefore, the efficiency trade-off is not simply between “one query” and “multiple queries”, but between reducing query invocation overhead and increasing the size and complexity of the returned result.

Table 6 shows a clear pattern with respect to the number of returned rows. When the total result size is small, Single-SQL is strongly preferred. For queries returning at most 10 rows, Single-SQL is faster in 95.5% of the cases; for queries returning 11–50 rows, Single-SQL is still faster in 83.0% of the cases. This suggests that, for small outputs, the dominant cost is the fixed overhead of executing multiple SQL statements. By merging the subqueries into one statement, Single-SQL avoids repeated SQL execution overhead and thus achieves better latency.

However, the trend reverses as the result size increases. When the result contains 51–1000 rows,

Feature	Single route	Multi route	Diff.	ρ
Question length	12.388	21.780	9.393	0.753
Sub-intent markers	0.447	1.702	1.255	0.686
Contains “and”	0.357	0.963	0.606	0.641
Contains “separately”	0.000	0.185	0.185	0.317
Aggregation cues	0.542	0.955	0.413	0.210
Ranking cues	0.190	0.345	0.155	0.109
Comparison cues	0.143	0.241	0.098	0.100
Grouping cues	0.272	0.480	0.208	0.076

Table 8: Feature analysis of router decisions. “Single route” and “Multi route” denote the average feature values among examples routed to single-SQL and multi-SQL generation, respectively. ρ denotes Spearman correlation with the binary routing decision.

Method	Accuracy	Precision	Recall	F1
Length-only	0.858	0.848	0.872	0.860
Qwen2.5-0.5B	0.991	0.984	0.998	0.991

Table 9: Comparison between the Qwen2.5-0.5B router and a length-only baseline. The length-only baseline predicts multi-SQL if the question length is at least 17 tokens, which is the best threshold selected by F1 score. Although length is a strong proxy for multi-SQL requirements, it is insufficient to explain the router’s behavior.

Multi-SQL becomes faster in 67.8% of the cases. When the result contains more than 1000 rows, Multi-SQL is faster in 97.0% of the cases. This indicates that, once the output becomes large, the additional cost introduced by the Single-SQL wrapper dominates the saved execution overhead.

The reason is that Single-SQL must return a unified table for all sub-query outputs. This requires padding non-applicable cells with `NULL`. As a result, each output row can become wider than the original result produced by the corresponding sub-query. For small results, this additional payload is negligible. For large results, especially on large databases or queries that return large lists, the widened output increases materialization, memory traffic, and result transfer cost. In contrast, Multi-SQL preserves the original schema of each sub-query and avoids padding every row to a common schema.

Table 7 further supports this explanation by measuring the number of returned cells. Here, `single_cells` is computed as the number of rows returned by the Single-SQL query multiplied by its number of output columns. When `single_cells` is no more than 100, Single-SQL is faster in around 94% of the cases. In the intermediate range of 101–1000 cells, the behavior

becomes mixed. Once `single_cells` exceeds 1000, Multi-SQL becomes dominant; when it exceeds 10000, Multi-SQL is faster in all cases in our evaluation. This suggests that the absolute output payload, rather than the number of SQL statements alone, is the key factor determining the efficiency trade-off.

SQL keywords alone are not reliable indicators of which execution strategy is faster. Since the Single-SQL form is constructed from the same decomposed sub-queries, operators such as `COUNT`, `DISTINCT`, `ORDER BY`, and `NOT IN` appear in both execution strategies. Their effect is therefore indirect: they may correlate with the expected output size or with expensive intermediate operations, but they do not by themselves determine whether Single-SQL or Multi-SQL is preferable. In our analysis, output size is a much stronger signal than the presence of any individual SQL operator.

Overall, Single-SQL is preferable when the decomposed sub-queries produce small result lists, where the saved execution overhead outweighs the cost of the `UNION ALL` wrapper. Multi-SQL is preferable when one or more sub-queries produce large outputs, especially on large databases where queries can generate many rows. In such cases, executing the sub-queries separately avoids the extra identifier column, `NULL` padding, and widened result schema introduced by the Single-SQL formulation.

H Router Decision Analysis

Although the router substantially improves downstream Text-to-(Multi)-SQL generation, its decision basis is not directly observable. To better understand what signals are associated with its routing decisions, we conduct a post-hoc analysis based on interpretable question-level features. Since the single-SQL and multi-SQL examples are

Length bin	#Ex.	Multi ratio	Router Acc.	Δ Sub-intent	Δ And	Δ Separately	Δ Agg.
4–11	932	0.011	1.000	1.189	0.802	0.400	0.256
12–15	940	0.184	0.988	1.107	0.558	0.306	0.228
16–19	869	0.613	0.988	0.613	0.348	0.201	0.421
20–23	773	0.882	0.987	0.614	0.295	0.164	0.555
24–45	780	0.960	0.997	0.419	0.153	0.168	0.736

Table 10: Length-bin analysis of router decisions. #Ex. denotes the number of examples in each question-length bin. Multi ratio denotes the proportion of multi-SQL examples. Router Acc. denotes router accuracy within the bin. Δ denotes the feature mean of gold multi-SQL examples minus that of gold single-SQL examples in the same bin. Sub-intent, And, Separately, and Agg. correspond to sub-intent markers, the occurrence of “and”, the occurrence of “separately”, and aggregation cues, respectively.

Question type	#Ex.	Coverage	Multi ratio	Router Acc.
Explicit decomposition	468	0.109	1.000	1.000
Long composite	2210	0.515	0.848	0.991
Multi-intent conjunction	2874	0.669	0.723	0.989
Comparison / filtering	717	0.167	0.605	0.987
Aggregation	2056	0.479	0.589	0.992
Ranking / superlative	962	0.224	0.587	0.992
Grouping / per-each	920	0.214	0.558	0.991

Table 11: Rule-based question type breakdown. Types are not mutually exclusive. Coverage denotes the proportion of examples belonging to each type. Multi ratio denotes the proportion of gold multi-SQL examples within each type. Router Acc. denotes router accuracy for examples of that type.

constructed over the same set of Spider schemas, global schema statistics such as the number of tables, columns, and foreign keys are largely controlled and are not expected to be the primary factors behind routing decisions. We therefore focus on question-side signals, including question length, sub-intent markers, conjunctions, aggregation cues, ranking cues, grouping cues, and comparison expressions.

Table 8 summarizes the feature distributions for examples routed to single-SQL and multi-SQL generation. Multi-SQL routing decisions are strongly associated with longer questions and linguistic decomposition cues. For example, questions routed to multi-SQL generation are substantially longer on average, contain more sub-intent markers, and are much more likely to include conjunctions such as “and”. Explicit decomposition markers such as “separately” also correlate positively with multi-SQL routing. Overall, these results indicate that routing decisions are primarily associated with question complexity and linguistic decomposition cues.

To further examine whether the router simply applies a question-length heuristic, we compare it with a length-only threshold classifier. As shown in Table 9, the best length-only classifier achieves an F1 score of 0.860, which is substantially lower

than the Qwen2.5-0.5B router’s F1 score of 0.991. This indicates that question length is a useful proxy for multi-intent structure, but it is insufficient to explain the router’s behavior.

We further divide the examples into five question-length bins, as shown in Table 10. Although the proportion of multi-SQL examples increases with question length, the middle bins still contain a mixture of single- and multi-SQL questions. The router maintains over 98.7% accuracy in every bin, suggesting that it can distinguish routing decisions even among questions of similar length. Moreover, within each length bin, multi-SQL examples consistently contain more sub-intent markers, conjunctions, and explicit decomposition cues such as “separately”. These results suggest that the router’s decisions are not merely driven by length, but are closely associated with linguistic signals of multiple information needs.

Finally, we analyze which question types are most associated with multi-SQL decomposition. We perform a rule-based breakdown using interpretable linguistic cues. Since the per-question benefit of decomposition depends on the downstream generator, we use the gold single/multi label as a proxy for whether a question structurally requires decomposition. As shown in Table 11, explicit decomposition markers provide the strongest sig-

nal: all questions in this category are labeled as multi-SQL. Long composite questions and questions with multi-intent conjunctions also show high multi-SQL ratios, suggesting that multi-SQL decomposition is especially relevant when a question combines multiple information needs. Aggregation, ranking, grouping, and comparison cues show moderate multi-SQL ratios, indicating that these operations alone do not necessarily require decomposition but often appear in questions with multiple sub-goals.

To further examine whether the router generalizes beyond generated SpiderS examples, we evaluate it on 175 manually constructed examples in the SpiderS test set written by human annotators. The router achieves an F1 score of 0.99.

We additionally evaluate the router on BIRD without using BIRD examples for training. Original BIRD questions are treated as single-SQL instances, while multi-intent instances are constructed by combining two questions from the same database using 12 unseen composition formats rather than the 14-pattern SpiderS generation pipeline. The router achieves an F1 score of 0.97. These results provide further evidence that the router generalizes across human-written examples, cross-dataset inputs, and unseen composition formats. However, the multi-intent examples remain human-constructed rather than naturally collected from real users.

I Overlap-Aware Analysis

Since SpiderS is built by extending Spider, many examples naturally share schemas, question fragments, or SQL patterns with the original benchmark. As Spider has been a widely used benchmark for years, modern LLMs may have been exposed to Spider examples directly or indirectly during pretraining or instruction tuning. This raises an important question: within SpiderS, are examples that are closer to the original Spider benchmark easier for current models, or do they still pose substantial challenges due to the multi-intent and multi-SQL requirements?

To answer this question, we conduct an overlap-aware analysis. For each SpiderS example, we compute its similarity to Spider examples from the same database using both question-level and sub-intent-level overlap. We first attempted a strict filtering procedure that removes examples with high full-question or sub-intent overlap with Spi-

der. This leaves only 56 examples, indicating that a purely low-overlap split is not representative of SpiderS. We therefore use a diagnostic less-vs-high overlap split. With a threshold of 0.75, this produces a less-overlap group of 222 examples and a high-overlap group of 1,925 examples.

As shown in Table 12, the raw less-overlap subset achieves higher performance than the high-overlap subset for both GPT-4o and XiYanSQL-32B. At first glance, this may seem counterintuitive: if overlap with Spider were the main factor, one might expect high-overlap examples to be easier. However, the raw comparison is confounded by difficulty distribution. The less-overlap subset contains a much larger proportion of easy examples, while the high-overlap subset contains many more hard and extra-hard examples. Therefore, the raw gap should not be interpreted as evidence that higher overlap makes examples harder.

To control for this confounder, we construct matched high-overlap subsets by sampling from the high-overlap group so that the sampled subset has the same hardness distribution as the less-overlap group. Specifically, each matched subset contains the same number of easy, medium, hard, and extra-hard examples as the less-overlap subset. We repeat this sampling process with 50 random seeds and report the mean and standard deviation.

After difficulty matching, the effect of overlap becomes model-dependent. For GPT-4o, the matched high-overlap subset achieves 0.557 ± 0.024 EX, which is still lower than the less-overlap result (0.694). For XiYanSQL-32B, the matched high-overlap subset achieves 0.775 ± 0.024 EX, slightly higher than the less-overlap result (0.752). These mixed trends suggest that similarity to Spider components is not the dominant factor determining SpiderS performance. It may help SQL-specialized models such as XiYanSQL-32B, but it does not consistently explain model success or failure across models.

This observation is consistent with our prompt-robustness results. On the original Spider benchmark, changing the prompt from generating one SQL query to generating one or more SQL queries already causes performance degradation for many models. This indicates that the Text-to-(Multi)-SQL setting changes the reasoning process itself: models must not only generate SQL, but also decide whether decomposition is needed, determine the number of required SQL queries, and ensure

Model	Less-overlap EX	High-overlap EX	Matched high-overlap EX	Full SpiderS EX
GPT-4o	0.694	0.476	0.557 ± 0.024	0.499
XiYanSQL-32B	0.752	0.662	0.775 ± 0.024	0.671

Table 12: Overlap-aware analysis on SpiderS. The less-overlap subset contains 222 examples, the high-overlap subset contains 1,925 examples, and each difficulty-matched high-overlap sample contains 222 examples. Full SpiderS EX is reported for reference.

that all information needs are covered. For harder examples, this additional reasoning burden is further amplified.

J Error Analysis for Prompt Robustness

In Section 7.4, we observe that several models exhibit performance degradation when evaluated on Spider under Prompt_{1VN} , compared with Prompt_1 . The purpose of this analysis is to clarify whether this degradation is primarily caused by a misunderstanding of the prompt instructions, or by another type of generation behavior induced by the more flexible output format.

We conduct a targeted error analysis based on GPT-4o’s generated outputs and their evaluation results. Specifically, we inspect cases where GPT-4o’s prediction is correct under Prompt_1 , but the corresponding prediction is judged incorrect under Prompt_{1VN} . These paired cases isolate errors associated with changing the prompt from requiring exactly one SQL query to allowing one or more SQL queries, while keeping the underlying dataset, database schema, and question unchanged.

The analysis shows that the majority of these failures are not caused by instruction-following or formatting errors. In the inspected cases, the model outputs remain syntactically valid and consistent with the required JSON format. We do not observe cases where the model fails to follow the prompt structure or produces outputs that are unrelated to the input question. Instead, the dominant failure mode is *multi-SQL over-generation*: the model generates multiple SQL queries even when a single SQL query is sufficient for the original Spider task.

Table 13 summarizes the error distribution. Among the cases where GPT-4o is correct under Prompt_1 but incorrect under Prompt_{1VN} , 94.5% correspond to multi-SQL over-generation. Within these over-generation cases, 75.4% contain two generated SQL queries, and 24.6% contain three generated SQL queries. In many such examples, at least one generated SQL query matches the gold SQL or returns the correct execution result, but the

Error category	Percentage
Multi-SQL over-generation	94.5%
Two generated SQL queries	75.4%
Three generated SQL queries	24.6%
Other errors	5.5%

Table 13: Error distribution for GPT-4o cases that are correct under Prompt_1 but incorrect under Prompt_{1VN} .

prediction is still judged incorrect by exact execution accuracy because additional SQL queries are also produced.

This explains why the drop in execution accuracy can be larger than the drop in execution score. Execution accuracy assigns zero credit when the complete predicted set is not fully correct, even if one generated SQL query is correct. By contrast, execution score can still assign partial credit when some generated queries match the gold execution result. Therefore, the performance degradation under Prompt_{1VN} should not be interpreted mainly as a failure to understand the prompt instructions. Rather, it reflects a robustness issue under a more flexible output space: when the model is allowed to generate one or more SQL queries, it sometimes generates unnecessary additional queries for questions that require only one.

K Analysis by Question Type

Table 14 reports model performance across the 14 question types defined in Appendix E. Overall, the results show that errors are not uniformly distributed across patterns. Instead, models are more likely to fail on question types that require preserving multiple independent constraints, handling complementary selections, or producing complete multi-query outputs.

The most difficult category is Type 1, i.e., reverse selection in one-to-many relationships. It obtains the lowest average EX across the three models. This suggests that models often struggle to correctly separate the selected subset from the remaining records while also retrieving attributes from a related entity. The gap between EX and ES

Table 14: Execution match and execution score across question types. XIYAN refers to XiYanSQL-32B, and Qwen2.5 refers to the fine-tuned Qwen2.5-Coder-7B-Instruct model.

Type	Count	GPT4o		XIYAN		Qwen2.5	
		EX	ES	EX	ES	EX	ES
1	275	0.36	0.60	0.48	0.71	0.60	0.78
2	111	0.44	0.64	0.70	0.82	0.73	0.80
3	64	0.66	0.77	0.86	0.93	0.88	0.91
4	278	0.54	0.70	0.66	0.80	0.57	0.75
5	176	0.77	0.87	0.73	0.84	0.76	0.87
6	229	0.61	0.73	0.76	0.82	0.76	0.82
7	212	0.40	0.50	0.70	0.79	0.81	0.89
8	49	0.08	0.47	0.82	0.88	0.86	0.91
9	103	0.78	0.80	0.93	0.94	0.94	0.95
10	108	0.53	0.68	0.67	0.80	0.81	0.89
11	204	0.35	0.60	0.57	0.74	0.63	0.78
12	6	0.33	0.58	1.00	1.00	1.00	1.00
13	25	0.36	0.42	0.60	0.70	0.80	0.80
14	307	0.49	0.59	0.65	0.74	0.77	0.87

is also large for this type, indicating that models can often solve part of the request but fail to cover all required information units.

Type 11 is another error-prone category. This type combines an ordered top- k query with an additional unordered query under a separate filtering condition. Its EX scores are 0.35, 0.57, and 0.63 for GPT-4o, XiYanSQL-32B, and Qwen2.5-Coder-7B, respectively. The difficulty likely comes from the need to simultaneously handle ranking, the LIMIT constraint, and an independent WHERE condition in another query. This makes Type 11 more challenging than Type 10, where only one query is ordered while the other remains unordered, but no LIMIT condition is involved.

Types 13 and 14 also reveal important failure modes. Type 13, which requires reverse-order ranking over grouped primary-key-based results, has low performance, especially for GPT-4o, with only 0.36 EX and 0.42 ES. This indicates that models may confuse the grouping key, the sorted aggregate, or the direction of the ranking condition. However, this result should be interpreted cautiously because Type 13 contains only 25 examples. Type 14 contains many more examples and requires three SQL queries. Its EX scores are 0.49, 0.65, and 0.77, showing that increasing the number of required queries makes complete coverage harder. The higher ES values, especially 0.87 for Qwen2.5-Coder-7B, suggest that models often generate some correct subqueries but miss at least one component,

leading to an EX failure.

Type 8 shows a model-specific weakness. GPT-4o performs very poorly on this type, with only 0.08 EX, while XiYanSQL-32B and Qwen2.5-Coder-7B achieve 0.82 and 0.86 EX. Since Type 8 requires switching between HAVING-based filtering and ORDER BY/LIMIT-based ranking, this result suggests that general-purpose LLMs may have difficulty distinguishing different aggregation-filtering structures, whereas SQL-specialized or fine-tuned models handle this pattern more reliably.

In contrast, some question types are relatively easy. Type 9 achieves the highest and most stable performance, with EX scores of 0.78, 0.93, and 0.94. This type only asks for the same data to be reordered from different perspectives, so the core selection target remains unchanged and only the ORDER BY attribute varies. Type 3 and Type 5 are also comparatively easier, suggesting that models are more reliable when the decomposition is explicit and each subquery follows a familiar selection pattern. Type 12 obtains perfect scores for XiYanSQL-32B and Qwen2.5-Coder-7B, but its count is only 6, so the result is not sufficient to draw a strong conclusion.

Overall, the main error-prone types are Type 1, Type 11, Type 13, Type 8, and Type 14. These patterns share a common property: they require models not only to generate valid SQL, but also to correctly decompose the question, preserve the intended constraint for each subquery, and cover all requested information units. The frequent gap between EX and ES further shows that many failures are partial rather than complete: models often generate at least one correct SQL query, but fail to produce the full set needed to satisfy the composite request.

L Experiments Prompts

Generate one SQL Prompt

You are an experienced database expert.

Now you need to generate **one SQL query** given the database schema and a question.
The database schema is defined as follows:

{DATABASE_SCHEMA}

Requirements:

1. Each SQL query must strictly adhere to the database schema provided.
2. Use precise JOINS, filters, and aggregations based on the schema.
3. Decompose the question into sub-queries if necessary and generate separate SQL queries.
4. Ensure all queries are syntactically correct and executable in an SQLite-compatible environment.

Your task:

Generate one SQL query to answer the given question.

Question:

{QUESTION}

Output Format:

Respond only with the following JSON structure:

```
{
  "sql_1": "Your SQL query here."
}
```

Generate Multiple SQL Prompt

You are an experienced database expert.

Now you need to generate **one or multiple SQL queries** given the database schema and a question.
The database schema is defined as follows:

{DATABASE_SCHEMA}

Requirements:

1. Each SQL query must strictly adhere to the database schema provided.
2. Use precise JOINS, filters, and aggregations based on the schema.
3. Decompose the question into sub-queries if necessary and generate separate SQL queries.
4. Ensure all queries are syntactically correct and executable in an SQLite-compatible environment.

Your task:

Generate up to three SQL queries to answer the given question. Use as many queries as needed, from one to three. If fewer than three queries are required, leave the unused query fields empty.

Question:

{QUESTION}

Output Format:

Respond only with the following JSON structure:

```
{
  "sql_1": "Your first SQL query here.",
  "sql_2": "Your second SQL query here, or leave it empty if not needed.",
  "sql_3": "Your third SQL query here, or leave it empty if not needed."
}
```