Titre: Title:	Enhancing Text-to-SQL Systems Through LLM Routing and Effective Evaluation
Auteur: Author:	Mohammadhossein Malekpour
Date:	2025
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Malekpour, M. (2025). Enhancing Text-to-SQL Systems Through LLM Routing and Effective Evaluation [Master's thesis, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/71174/

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Enhancing Text-to-SQL Systems through LLM Routing and Effective Evaluation

MOHAMMADHOSSEIN MALEKPOUR
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Décembre 2025

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Enhancing Text-to-SQL Systems through LLM Routing and Effective Evaluation

présenté par **Mohammadhossein MALEKPOUR**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de:

Heng LI, président
Foutse KHOMH, membre et directeur de recherche
Amine MHEDHBI, membre et codirecteur de recherche
Lina MARSSO, membre

DEDICATION

To my father, Adel, for being my greatest motivation.

To my mother, Afsaneh, for her endless love and kindness.

To my sister, Fatemeh, for her constant pride and encouragement.

To my wife, Maryam, for being my safe shelter in every difficult moment.

To victims of Flight PS752 and to the people of Iran, whose courage will never be forgotten.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor, Professor Amine Mhedhbi, whose expertise, vision, and generous guidance shaped every stage of this work. Learning from him has been one of the most inspiring and rewarding parts of my master's journey, and his support, patience, and insight made this entire experience truly meaningful.

I am also sincerely grateful to Professor Foutse Khomh for welcoming me into his lab and giving me the opportunity to pursue my studies in Canada. His trust and support helped make this work possible.

I extend my sincere thanks to Professor Heng Li and Professor Lina Marsso for dedicating their valuable time and thoughtful effort to reviewing my thesis.

Finally, I would like to thank my lab mates Mohamed, Sunny, Anas, Nour, and Ela. Your friendship and support made this journey lighter and filled my time in the lab with warmth and great memories.

RÉSUMÉ

L'exploration des données par requêtes constitue un goulot d'étranglement opérationnel dans la prise de décision en entreprise. Les utilisateurs métier disposent souvent du contexte de domaine nécessaire, mais maîtrisent mal le SQL et les schémas de bases de données, tandis que les équipes data centralisées, qui traduisent les demandes en requêtes, fonctionnent comme une ressource partagée et contrainte, ce qui introduit des délais. Les systèmes Text-to-SQL visent à réduire cette friction en traduisant des questions en langage naturel (NL) en SQL exécutable, et les progrès récents des grands modèles de langage (LLM) ont rendu les premiers déploiements en entreprise de plus en plus envisageables. Malgré ces avancées, deux défis persistent: (i) le coût élevé des pipelines Text-to-SQL fortement dépendants des LLM pour traiter les requêtes les plus complexes, alors même que les charges de travail réelles couvrent un large éventail de complexité; et (ii) une évaluation reposant sur une métrique grossière et binaire, telle que l'exactitude d'exécution, ainsi que sur des benchmarks publics dont les schémas ne reflètent pas ceux des entreprises.

Cette thèse traite ces défis à travers de deux contributions complémentaires. Premièrement, nous introduisons un cadre de routage de LLM sensible au coût pour le Text-to-SQL, qui sélectionne, pour chaque requête, le modèle le moins coûteux susceptible de générer un SQL exact, au lieu de recourir systématiquement à un unique modèle de référence pour toutes les requêtes NL. Notre routeur est léger, construit à partir de signaux peu coûteux et de règles de décision simples (basées sur des scores et sur la classification), de sorte que la surcharge induite par le routage reste négligeable par rapport à l'étape de génération SQL qu'il optimise. Des expériences sur un benchmark récent montrent que le routage conserve une exactitude proche de celle du modèle le plus performant tout en réduisant le coût, ce qui le rend particulièrement adapté aux déploiements à fort volume.

Deuxièmement, nous renforçons l'évaluation à la fois du point de vue des métriques et des données d'évaluation. Du côté des métriques, nous proposons des mesures d'exécution fines opérant sur les tables de résultats exécutées: l'Execution Precision (EXP) quantifie la part de la sortie prédite qui est correcte, l'Execution Recall (EXR) quantifie la part de la sortie de référence récupérée, et leur score F1 en fournit une synthèse. Ces métriques mettent en évidence la correction partielle et distinguent la sur-prédiction de la sous-prédiction d'une manière que l'exactitude d'exécution binaire ne permet pas, offrant ainsi un signal plus diagnostique pour le débogage. Du côté des données, nous introduisons la Textual Query Augmentation (TQA), qui dé-naturalise systématiquement les identifiants de schéma et leurs

mentions en NL, par exemple en remplaçant des noms descriptifs par des abréviations de style entreprise, tout en préservant la sémantique. Appliquée à des systèmes Text-to-SQL de l'état de l'art, la TQA révèle une fragilité importante sous des régimes de nommage proches de ceux des entreprises et permet une évaluation plus réaliste, centrée sur la robustesse.

Notre cadre de routage, nos métriques fines et la TQA constituent des approches pratiques pour rendre les systèmes Text-to-SQL fondés sur des LLM à la fois plus efficaces et évalués de manière plus fiable, dans des conditions plus proches de l'usage réel en entreprise.

ABSTRACT

Exploratory data querying is an operational bottleneck in enterprise decision-making. Business users often have the necessary domain context but lack familiarity with SQL and database technology, while centralized data teams that translate requests into queries operate as a constrained shared resource, introducing delays. Text-to-SQL systems aim to reduce this friction by translating natural-language (NL) questions into executable SQL, and recent advances in large language models (LLMs) have made early enterprise deployments increasingly feasible. Despite this progress, two challenges persist: (i) the high cost of LLM-heavy Text-to-SQL pipelines to tackle the most complex queries, even though real workloads span a wide range of query complexities; and (ii) evaluation relying on a coarse, binary metric such as execution accuracy and on public benchmarks whose schemas fail to reflect enterprise ones.

This thesis addresses these challenges through two complementary contributions. First, we introduce a cost-aware LLM routing framework for Text-to-SQL that selects, on a per-query basis, the least expensive model expected to generate accurate SQL, rather than defaulting to a single strongest model for all NL queries. Our router is lightweight, built from inexpensive signals and simple score-based and classification-based decision rules, so that routing overhead remains negligible relative to the SQL generation stage it optimizes. Experiments on a modern benchmark demonstrate that routing preserves accuracy close to that of the strongest model while reducing cost, making it especially well-suited for high-volume deployments.

Second, we strengthen evaluation along both the metric and dataset dimensions. On the metric side, we propose fine-grained execution measures that operate on executed result tables: Execution Precision (EXP) quantifies how much of the predicted output is correct, Execution Recall (EXR) quantifies how much of the ground-truth output is recovered, and their F1 score summarizes both. These metrics expose partial correctness and distinguish over- from under-prediction in ways that binary execution accuracy cannot, providing a more diagnostic signal for debugging. On the data side, we introduce Textual Query Augmentation (TQA), which systematically de-naturalizes schema identifiers and their NL mentions, e.g., replacing descriptive names with enterprise-style abbreviations, while preserving semantics. Applied to state-of-the-art Text-to-SQL systems, TQA reveals substantial brittleness under enterprise-like naming regimes and enables more realistic, robustness-aware evaluation.

Our routing framework, fine-grained metrics, and TQA provide practical approaches for making LLM-based Text-to-SQL systems both more efficient and more reliably evaluated under conditions closer to real enterprise use.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

NL	Natural Language
SQL	Structured Query Language
DL	Deep Learning
PLM	Pre-trained Language Model
LLM	Large Language Model
EX	Execution Accuracy
EXP	Execution Precision
EXR	Execution Recall
TQA	Textual Query Augmentation

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

Exploratory data querying remains a major operational bottleneck for enterprise decision-making. Business users (e.g., product managers and analysts) often possess the necessary domain context but lack familiarity with the underlying physical data representation such as database schemas and SQL semantics. Conversely, centralized data teams understand the physical database structure and can translate questions into SQL queries and views, yet they operate as a constrained shared resource serving many stakeholders. This arrangement typically introduces friction: requests are clarified over multiple iterations, context is lost in handoffs, and delivery timelines stretch up to weeks. As a result, the time from question to actionable insight lengthens, slowing experimentation and delaying business decisions.

Text-to-SQL systems seek to relieve this operational bottleneck by translating natural-language (NL) queries into executable SQL. This creates a more direct path from user intent to answers, lowering the barrier to data access for non-experts. This also accelerates exploratory analysis for practitioners who would otherwise rely on manual query authoring. Recent advances in large language models (LLMs) have moved Text-to-SQL from a long-standing research problem toward practical adoption, with early enterprise deployments now emerging. In particular, LLMs have demonstrated strong SQL generation capabilities across complex schemas and the ability to produce arbitrarily nested and compositional queries that were difficult for earlier rule-based and task-specific neural methods.

Despite this progress, current deployments face two central challenges: (i) the high cost of Text-to-SQL systems and (ii) the lack of effective evaluation techniques on both public and private datasets alike.

Challenge 1 – High Cost. Modern Text-to-SQL systems are typically implemented as multi-stage pipelines, with several stages invoking state-of-the-art LLMs to maximize accuracy on complex inputs. While this design improves performance on difficult queries, it also incurs substantial inference overhead in both monetary cost and end-to-end latency. In practice, however, queries in enterprise workloads vary widely: the difficulty of user queries ranges from trivial lookups to multi-table joins and nested subqueries. Invoking the most expensive model uniformly across all queries is therefore inefficient.

To illustrate, consider a simple query such as *"Name any sports organization."* A suitable SQL translation is a straightforward single-table selection, a projection, and a LIMIT clause:

```
SELECT ORG_NAME FROM SPORTS_FINANCIALS LIMIT 1;
```

Now contrast this simple query with a substantially more complex one reported in the enterprise setting [1]: *"Identify our five sports organizations with the best and worst quarter-on-quarter financial performance in Canada for Q1 2024."* Here, the system must (i) ground the question in a larger portion of the schema (e.g., revenue, time, and geography attributes); (ii) interpret quarter boundaries and compute quarter-on-quarter deltas; (iii) rank entities by their deltas (improvement or deterioration); and (iv) return the top- and bottom-5 results. A plausible SQL translation is therefore far more involved. A suitable SQL query can be:

```
WITH FINANCIALS AS (
  SELECT
    COUNTRY,
    SPORT_CATEGORY,
    SUM(CASE WHEN TO_CHAR(FIN_MONTH, 'YYYY"Q"Q') = '2024Q1'
            THEN REVENUE ELSE 0 END) AS REVENUE_2024Q1,
    SUM(CASE WHEN TO_CHAR(FIN_MONTH, 'YYYY"Q"Q') = '2024Q2'
            THEN REVENUE ELSE 0 END) AS REVENUE_2024Q2
  FROM SPORTS_FINANCIALS
  WHERE TO_CHAR(FIN_MONTH, 'YYYY"Q"Q') IN ('2024Q1', '2024Q2')
    AND COUNTRY = 'CANADA'
  GROUP BY COUNTRY, SPORT_CATEGORY
), VIEWERSHIP AS (
  SELECT
    COUNTRY,
    SUM(CASE WHEN TO_CHAR(VIEW_MONTH, 'YYYY"Q"Q') = '2024Q1'
            THEN VIEWER_HOURS ELSE 0 END) AS VIEWER_HOURS_2024Q1,
    ...
    ) AS WORST_SPORT_RANK
  FROM FINANCIALS f
  JOIN VIEWERSHIP v ON f.COUNTRY = v.COUNTRY
)
SELECT SPORT_RANK, SPORT_CATEGORY, RPV, PRIOR_QTR_RPV, RPV_CHANGE, IMPACT
FROM CALCULATIONS
WHERE SPORT_RANK <= 5 OR WORST_SPORT_RANK <= 5
ORDER BY SPORT_RANK;
```

This second query demands substantially richer SQL structure: multiple common table expressions (CTEs), joins across tables, conditional aggregations, derived measures, and ranking clauses. Yet in many deployed Text-to-SQL systems, both the trivial lookup and the multi-step analytical query would be generated by the same state-of-the-art LLM, because the

system must be capable of handling the hardest cases. When a large fraction of exploratory workloads consists of simple queries, this one-size-fits-all strategy unnecessarily increases latency and inference cost, and can even degrade the interactive experience.

Challenge 2 – Effective Evaluation. This challenge has two complementary aspects: the metrics used to score system predictions, and the realism of the dataset schemas on which those metrics are computed.

Most Text-to-SQL evaluation relies on a single metric: *Execution Accuracy* (EX), a binary measure: if the output relation (i.e., the result table) produced by the predicted SQL matches the ground-truth relation, the query receives an EX of 1; otherwise, an EX of 0. This score is then aggregated across many input queries to compare systems. While EX is simple and easy to implement, it collapses a wide spectrum of outcomes into a single bit.

A prediction that recovers 99 out of 100 expected rows and one that returns a completely unrelated table both receive an EX of 0, since in both cases the predicted and ground-truth relations are not equal. Likewise, predictions that differ only by potentially negligible changes, such as returning extra columns, are penalized in the same way as predictions with severe semantic errors. For debugging and for model training or fine-tuning, this makes EX a very coarse signal: nearly correct results are indistinguishable from completely wrong ones. Moreover, EX provides no indication of how a prediction failed, e.g., whether the error arises from missing, mismatched, or extra rows or columns.

A second source of difficulty in effective evaluation comes from the evaluation datasets themselves. The schemas in public benchmarks often do not resemble the schemas encountered in practice. Benchmark datasets typically present small-to-moderate numbers of tables whose names, columns, and relationships are curated to be human-readable (e.g., `Student`, `major`, or `enrollment_date` as columns). Enterprise databases, by contrast, contain frequently abbreviated, overloaded, or encoded as internal identifiers column names (e.g., `mjr_desc` for "major description" or `enr_dt` for "enrollment date"). This mismatch makes the core grounding problem in Text-to-SQL, i.e., linking an NL entity, relationship, or attribute mention to the correct schema element, substantially harder in real deployments than in public benchmark settings. Consequently, systems can appear robust when evaluated on clean public benchmarks yet behave brittly on private enterprise schemas, where ambiguity and inconsistent naming conventions are the norm.

Research Questions. The two challenges above motivate the core research questions (RQs) that drive this thesis, addressed through two complementary lines of work.

RQ1: Given that enterprise NL queries yield SQL ranging from trivial lookups to multi-CTE

queries with joins and aggregates, can a Text-to-SQL system adaptively route each stage of a pipeline to the most "adequate" LLM, avoiding default reliance on the most capable (and most expensive) model?

RQ2: How can we evaluate Text-to-SQL systems in a way that is (i) diagnostic, moving beyond binary EX to expose the nature of errors (e.g., missing vs. extra rows/columns); and (ii) enterprise-representative, accounting for the gap between clean public benchmark schemas and the abbreviated or ambiguous schemas common in real deployments?

1.1 Contributions

LLM Routing for Optimizing SQL Generation (Chapter 3). To address the high cost incurred when a single state-of-the-art LLM is applied uniformly to queries of widely varying complexity, we design and evaluate an LLM routing framework for Text-to-SQL that selects, on a per-query basis, the least expensive model expected to produce accurate SQL, rather than defaulting to a single strongest model for all inputs. The router is intentionally lightweight: it uses inexpensive signals and simple decision rules, including score-based and classification-based strategies, so that routing overhead remains negligible relative to the LLM-based SQL generation it optimizes. We also examine when routing can improve accuracy in principle (through complementary model errors) and show empirically that general-purpose LLMs often fail on many of the same hard queries. In this setting, routing primarily enables a controlled cost–accuracy trade-off, keeping accuracy close to the best-performing LLM while reducing average inference cost.

Fine-grained Metrics and Query Mutation for Effective Evaluation (Chapter 4). To address both the limitations of binary Execution Accuracy and the mismatch between clean public benchmarks and enterprise schemas, we propose SQLMorph[1], an evaluation framework that combines fine-grained execution metrics with controlled query mutation to make Text-to-SQL evaluation more diagnostic and more enterprise-representative. On the metric side, we introduce Execution Precision (EXP) and Execution Recall (EXR), which quantify how much the output of the predicted SQL matches the ground truth's; their F1 score summarizes both. These metrics distinguish near-misses from major failures and separate over-prediction from under-prediction, providing a finer-grained signal for debugging and for training or fine-tuning than EX. On the data side, we introduce Textual Query Augmentation (TQA), a mutation approach to systematically de-naturalize schema identifiers and their NL mentions, e.g., moving from descriptive names like `enrollment_date` to enterprise-style

[1] https://github.com/dais-polymtl/sqlmorph

abbreviations like `enr_dt`, while preserving semantics. This mutation stress tests certain Text-to-SQL stages as such the ones grounding the query in the schema or ones resolving ambiguity through a human-in-the-loop approach. Together, these components enable more realistic and informative evaluation on both public benchmarks and enterprise-like variants.

1.2 Thesis Outline

Chapter 2 reviews the Text-to-SQL task and its evaluation methodology, and introduces *VortoSQL*, our pipeline used throughout this thesis for experimental study. Chapter 3 formulates the LLM routing problem, presents the proposed routing strategies for the SQL generation stage, and empirically characterizes their behavior. Chapter 4 introduces SQLMorph, an evaluation framework that provides execution metrics, analyzes their behavior under common error patterns, and presents Textual Query Augmentation (TQA) for producing enterprise-like schema variants while preserving semantics. Chapter 5 concludes with a discussion of findings, limitations, and directions for future work.

CHAPTER 2 BACKGROUND

This chapter provides the background needed to contextualize the contributions of this thesis. We first define the Text-to-SQL task and illustrate how natural-language questions are grounded to database schemas and translated into executable queries. We then summarize the main challenges that drive system design in practice, including linguistic ambiguity, schema complexity, domain knowledge requirements, and privacy constraints. Next, we review the evolution of approaches from rule-based and neural models to modern LLM-based pipelines, highlighting the key trade-offs in accuracy, generalization, and controllability. Finally, we describe standard benchmarks and evaluation metrics used in prior work, and introduce VortoSQL, the modular LLM-based pipeline that we use as our experimental testbed.

2.1 What is Text-to-SQL?

Text-to-SQL is the task of translating a natural language (NL) request into an executable SQL query over a target relational database, thereby bridging non-expert users and structured data access [2, 3]. As relational databases remain the backbone of many data-centric systems, direct SQL authoring can be a barrier for users without database expertise. A Text-to-SQL system interprets the user's intent, aligns mentions to the database schema (tables, columns, and values), and generates a logically equivalent query that can be executed to return the desired result. (e.g., Figure 2.1).

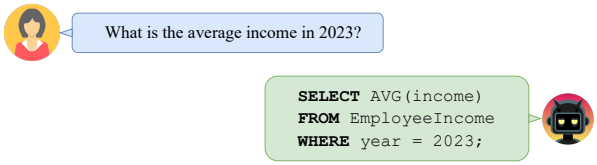


Figure 2.1 Text-to-SQL: mapping natural language to SQL.

Beyond simple lookups, practical deployments target complex information needs involving multi-table joins, nested predicates, aggregations, and ordering. By allowing users to ask questions in plain language, Text-to-SQL democratizes data access across domains such as business intelligence, customer support, and scientific analysis, reducing reliance on SQL specialists and pre-built dashboards [2]. This is especially helpful for onboarding analysts or time-sensitive, exploratory scenarios, where quick, interactive answers accelerate the path from idea to insight [1].

2.2 Challenges in Text-to-SQL

Text-to-SQL lies at the intersection of natural language understanding and program synthesis, and inherits difficulties from both. On the language side, user questions frequently contain *linguistic complexity and ambiguity*–nested clauses, coreference, ellipsis, and multiple plausible readings–which complicate intent resolution and the mapping from words to relational operators and attributes [4–6]. Robust systems must therefore integrate contextual cues and *domain knowledge*, including implicit business assumptions and schema conventions, to disambiguate intent and ground references to database contents and conventions [7].

A second class of challenges concerns *schema understanding and representation*. Accurate SQL generation requires faithful modeling of tables, columns, keys, and inter-table relationships, yet real schemas vary widely across domains and can be structurally intricate [8]. This complexity is amplified by *rare and complex SQL operations* (e.g., nested subqueries, outer joins, window functions) that are underrepresented in training data, as well as the need for *cross-domain generalization* when vocabulary and schema patterns shift between databases [9]. In practical deployments, schemas can be extremely large, stretching model context and attention and rendering naive schema linking brittle and slow [10].

A third set of issues arises from *knowledge and privacy*. Many real-world tasks require domain-specific terminology, abbreviations, and external definitions that are not captured by general pre-training. Retrieval-augmented prompting can help, but depends on high-quality knowledge bases and can introduce noise via imperfect retrieval [11]. Fine-tuning can embed domain knowledge but risks catastrophic forgetting and reduced adaptability, and is further complicated by privacy constraints when proprietary data cannot leave controlled environments [10]. These pressures motivate careful data governance–curating cleaner supervision and structured domain knowledge to improve retrieval, reduce ambiguity, and better align natural language with executable logic [10].

Finally, *evaluation* challenges persist. Recent work models human-like trial-and-error via *autonomous agents* that interleave reasoning, execution, and repair (e.g., ReAct-style controllers, REFORCE, and agentic pipelines in Spider 2.0), showing promising gains over single-shot approaches [12–14]. At the same time, standard evaluation faces persistent issues of *ambiguity* (multiple semantically distinct SQLs yielding acceptable answers) and *semantic mismatch* (questions partially unsupported by the database), which complicate the interpretation of scores and highlight the need for enterprise-scale, long-context benchmarks [10]. Moreover, the dominance of binary metrics such as Execution Accuracy (EX) leads to *diagnostic blindness*, obscuring partial correctness or over-/under-selection behaviors, while recent reinforcement

learning–based approaches face *reward–evaluation misalignment*, where proxy or binary rewards fail to capture true execution correctness or nuanced SQL behaviors.

2.3 Evolution of Approaches

Research on text-to-SQL has progressed through four major phases: early rule-based systems, neural sequence models, the adoption of pre-trained language models (PLMs), and, most recently, large language model (LLM)–based methods that rely on prompt engineering and/or fine-tuning. Each phase broadened coverage and robustness, while introducing new design trade-offs in generalization, data requirements, and controllability.

2.3.1 Rule-based Methods

Early systems mapped natural language to SQL with hand-crafted grammars, templates, and deterministic heuristics [15–18]. These pipelines enforced SQL syntax via production rules and slot-filling templates, yielding syntactically well-formed outputs for in-scope patterns. However, their reliance on manually curated rules limited portability new schemas and domains, and they struggled with linguistic phenomena such as ambiguity, ellipsis, and coreference. Encoding schema structure (e.g., multi-table joins and constraints) also proved labor-intensive and brittle, leading to rapid degradation outside curated scenarios [15–17].

2.3.2 DL-based Methods

With neural sequence-to-sequence modeling, encoder–decoder architectures (LSTMs and Transformers) became the dominant paradigm for mapping questions and schema representations to SQL [19–22]. Models explored intermediate representations and sketch-based slot filling to better handle composition and cross-domain variation (e.g., RYANSQL) [21, 22], and integrated schema graphs through graph neural networks to capture relational structure [23, 24]. Despite flexibility over rules, these approaches frequently produced incomplete or ill-formed queries (missing clauses, mis-nested subqueries) and underperformed on operations (e.g., window functions), reflecting data sparsity and limited structural priors [21–24].

2.3.3 PLM-based Methods

PLMs such as BERT and RoBERTa provided stronger linguistic priors via large-scale pre-training, and were fine-tuned on canonical text-to-SQL corpora to improve semantic understanding and SQL accuracy [8, 22, 25, 25–28]. Subsequent work made models schema-aware

by injecting structure and constraints from database metadata to enhance executability and schema grounding [29]. While PLMs generally surpassed earlier neural baselines, challenges remained: complex operations (e.g., outer joins, multi-stage aggregation) still induced syntax/-logic errors, and cross-domain generalization degraded when schema or vocabulary diverged from fine-tuning data, often necessitating costly adaptation [22, 25, 28].

2.3.4 LLM-based Methods

LLMs (e.g., GPT series) leveraged scaling laws and emergent capabilities to shift from task-specific training to prompt-driven workflows and lightweight adaptation [30–32]. Two streams predominate: prompt engineering that orchestrates retrieval-augmented prompts, few-shot demonstrations, and reasoning (e.g., chain-of-thought) [33–36], and fine-tuning of open models for domain constraints or privacy needs [34, 37]. Compared with traditional pipelines that mix diverse encoders/decoders (LSTM, Transformer, GNN), LLM-based systems emphasize a more uniform Transformer backbone [20] with modular pre-/inference/post-processing, while exploring domain knowledge integration and efficient adaptation [7, 38].

In-context Learning. In-context learning treats the LLM as a programmable interface: carefully constructed prompts bundle task instructions, schema snippets, and demonstrations, optionally enriched via retrieval-augmented generation to surface domain knowledge and value hints [31]. Reasoning additions such as chain-of-thought guide intermediate steps and decomposition for complex queries, improving faithfulness and coverage without parameter updates [36, 39, 40]. This pathway trades data/compute efficiency and rapid iteration for sensitivity to prompt design and context-window limits.

Fine-tuning. Fine-tuning follows the pre-train–fine-tune paradigm: curate or synthesize task data, select a base LLM, and adapt parameters (often with parameter-efficient techniques such as LoRA) to embed domain schemas, SQL patterns, and organizational constraints [34,41]. Compared with pure prompting, fine-tuning can yield stronger consistency, privacy control, and latency benefits, but introduces costs in data collection, training infrastructure, and maintenance; practical systems frequently combine selective fine-tuning with prompt-time retrieval and post-processing to balance accuracy, efficiency, and governance [37].

2.4 Benchmark and Evaluation Metrics

2.4.1 Datasets

Research benchmarks for text-to-SQL have evolved along two chronological waves. *Pre-LLM* resources established the core problem and evaluation protocols, led by cross-domain corpora such as WikiSQL [27], Spider 1.0 [8], and KaggleDBQA [42], with additional single- and cross-domain sets introduced subsequently (e.g., DuSQL [43]). Building on these foundations, a rich family of *post-annotated* variants stress-tested specific capabilities while reusing existing databases: robustness to schema/question perturbations (Spider-Realistic, Spider-SYN, ADVETA) [4, 44, 45], compositional generalization via clause-level recomposition (SPIDER-CG) [46], context-dependent interaction (SParC, CoSQL) [47, 48], and cross-lingual transfer (Spider-Vietnamese) [49]. Together, these datasets supported systematic analysis of schema grounding, compositionality, and generalization before the rise of LLMs.

With the emergence of instruction-following LLMs, *LLM-era* benchmarks shifted focus toward realism, scale, and long-context reasoning. SCIENCEBENCHMARK targets domain-shifted scientific databases curated with experts [50]. Dr.Spider measures robustness under 17 perturbation types spanning databases, questions, and SQL [51]. BIRD introduces large and noisy databases, evidence annotations for external knowledge grounding, and efficiency-aware evaluation [7]. Spider 2.0 frames enterprise-style, multi-query workflows across heterogeneous dialects and platforms [13]. Complementing these, BIRD-Critic 1.0 evaluates diagnostic and troubleshooting abilities on user-reported issues, while BULL offers a finance-specific testbed for specialized domains [52]. This newer suite emphasizes challenges underrepresented in earlier work, including knowledge augmentation, scalability, and execution efficiency.

In our studies, we adopt the BIRD benchmark as the primary evaluation benchmark, and we will explain the dataset details in the Experimental Setup section.

2.4.2 Metrics

Four widely used metrics for evaluating text-to-SQL systems are summarized below, organized into two categories: *content matching–based* and *execution–based* [53].

Content Matching–based Metrics. These metrics assess how closely the structure of the generated SQL matches the reference query, focusing on clause-wise correspondence rather than execution outcomes. They are particularly useful for diagnosing which parts of a query are incorrect while remaining robust to harmless ordering differences.

- **Component Matching (CM)** [8] evaluates the overlap between predicted and ground truth SQL at the clause level (`SELECT`, `WHERE`, `GROUP BY`, `ORDER BY`, and `KEYWORDS`). Each clause is decomposed into a set of sub-components (e.g., aggregated columns or conditions), and equality is checked on a set basis to ignore order. The final CM score is computed as the F1 score across exact set matches for all components.
- **Exact Matching (EM)** [8] measures the proportion of examples where the predicted SQL exactly matches the gold query across *all* components defined in CM. Because matching is performed on unordered sets, EM remains invariant to clause-internal ordering differences.

Execution–based Metrics. These metrics measure semantic correctness by comparing the execution results of the predicted and ground-truth queries on the database. They capture whether a system produces queries that yield the same answers or do so efficiently.

- **Execution Accuracy (EX)** [7, 8] evaluates correctness based on execution results. Let R_i and $\hat{R}_i$ denote the result sets of the ground-truth and predicted queries, respectively. The metric assigns a score of 1 if they match exactly and 0 otherwise:

$$EX = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}(R_i = \hat{R}_i) \quad \text{where} \quad \mathbf{1}(R_i, \hat{R}_i) = \begin{cases} 1 & \text{if } R_i = \hat{R}_i, \\ 0 & \text{otherwise.} \end{cases} \tag{2.1}$$

- **Valid Efficiency Score (VES)** [7] extends EX by also considering query runtime for valid predictions (i.e., those that yield identical results). For N examples, let Y_n and $\hat{Y}_n$ be the gold and predicted SQL, and V_n and $\hat{V}_n$ their executed outputs. It's defined as:

$$\text{VES} = \frac{1}{N}\sum_{n=1}^{N} \mathbf{1}(V_n, \hat{V}_n) \cdot R(Y_n, \hat{Y}_n) \text{ where } \mathbf{1}(V_n, \hat{V}_n) = \begin{cases} 1 & \text{if } V_n = \hat{V}_n, \\ 0 & \text{if } V_n \neq \hat{V}_n \end{cases} \tag{2.2}$$

  filters out invalid queries, and relative runtime efficiency is measured as

$$R(Y_n, \hat{Y}_n) = \sqrt{\frac{E(Y_n)}{E(\hat{Y}_n)}}, \tag{2.3}$$

  where $E(\cdot)$ denotes the execution time. Following the BIRD benchmark [1], efficiency values are averaged over multiple runs for stability.

2.5 VortoSQL: A Modular LLM-based Pipeline for Text-to-SQL

The system we implemented for our research, called **VortoSQL**, is a modular pipeline for In-context Learning (ICL)-based text-to-SQL generation. It is built in stages, where each stage is responsible for a specific part of the process. The pipeline follows a clear flow of retrieval, generation, and correction, reflecting the modular structure of typical multi-stage text-to-SQL systems, as shown in Figure 2.2.

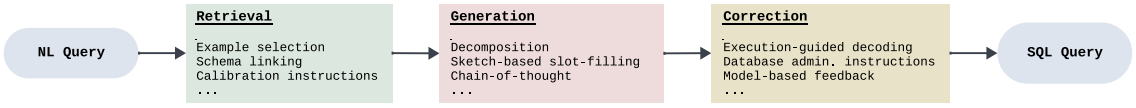


Figure 2.2 Text-to-SQL pipeline.

In the following sections, each stage is discussed in two parts: first, we provide an overview of related systems and techniques in the literature, and then we describe how the corresponding component is implemented in **VortoSQL**. At the end of this section, we provide a consolidated summary of all component configurations, as shown in Table 2.1.

2.5.1 Stage I – Retrieval

In the first stage, the system collects the information needed to ground SQL generation in the underlying database and task context. Modern Text-to-SQL pipelines treat retrieval broadly: it includes identifying the relevant subset of the database schema, retrieving informative example queries for in-context learning, and optionally incorporating auxiliary domain knowledge [54–56]. A central objective is to expose enough structure for accurate reasoning while avoiding unnecessary noise and token overhead.

Schema-oriented retrieval typically combines schema linking and schema filtering. Some systems use lightweight keyword or entity matching to map question terms to tables and columns [54, 55], while others enrich this step with value retrieval from the database using similarity search techniques such as Locality Sensitive Hashing (LSH) over vector embeddings [56–59]. These methods either *filter* the schema down to likely relevant elements or *augment* prompts with schema and value hints drawn from catalogs and metadata, thereby narrowing the space of plausible queries.

Recent approaches further integrate LLM-based reasoning into retrieval. Systems such as OpenSearch-SQL combine LLM schema linking with dense retrieval for values, then apply constraints and disambiguation rules before passing context to generation [60]. Others (e.g., N-rep, RSL-SQL, TA-SQL) explore multiple schema views or use provisional SQL sketches

to refine which entities matter [61–63]. Conversely, E-SQL questions aggressive filtering for advanced LLMs and instead performs *Query Enrichment*, using IR techniques (e.g., BM25) to select schema items, values, and instructions that are appended directly to the prompt [64, 65]. Across these designs, the retrieval stage serves as the interface between unstructured questions and structured database semantics: it defines what the model "sees" about the schema, which examples it can imitate, and which hints it receives for interpreting the question.

VortoSQL Schema Linker: The schema linker builds the database context that the LLM will see. Its goal is to present the amount of schema so the model can write a valid query. We support a *full schema* mode, where we pass the complete schema as text (all tables, columns, primary keys, and foreign keys). This avoids missing information, but increases tokens and cost. For filtering schema, inspired by prior study [66], we use two methods:

Table-then-Column Schema Linking (TCSL). The model first chooses a small set of tables that are relevant to the question, then chooses the columns inside those tables. The table step uses a table selection prompt (Appendix A) that takes the question, optional evidence, and a summary of the full schema. The column step uses a column selection prompt (Appendix A) over the reduced table list. The output is a set of relevant tables and columns. TCSL is a more aggressive filter and keeps prompts short.

Single-Column Schema Linking (SCSL). The model scores each column independently, without relying on table context. The output is a boolean relevance flag per column. This method is more cautious and is less likely to drop a needed column. We use a dedicated prompt for this method (Appendix A).

The input to schema linking is the user question and the database schema. We may also include an optional evidence string if domain notes are available. The output is a single schema description string that we insert into the LLM prompt. The text is human-readable and follows a stable format. It lists tables and columns with types, shows primary keys for each table, and shows foreign keys as join hints. When column descriptions or value notes exist in the metadata, we include them to help the model resolve ambiguous terms.

There is a trade-off between recall and noise. If filtering drops a needed column, the final SQL will be wrong. Passing many irrelevant columns can also hurt and raises cost. Recent evidence shows that newer LLMs handle extra schema better; when the full schema fits in context, it may be better to skip filtering. Schema linking still helps with smaller or older models, and it is necessary when schemas do not fit the context window [66].

VortoSQL Example Selector: The example selector prepares demonstrations for in-context learning. These demonstrations show the model how a question maps to an SQL query. We support zero-shot, one-shot, and few-shot prompting. In zero-shot, the prompt contains the instruction, the schema, and the user question. In one-shot and few-shot, we prepend k demonstrations. Each demonstration has a question, optional evidence text, and a gold SQL. The value of k controls the trade-off between cost and quality: more examples can help on complex cases, but they increase tokens.

The input to this stage is the user question and a pool of candidate examples. The output is a list of k demonstrations that will be inserted before the user question. We keep the format stable so it is easy to read and easy to compose into the final prompt. When evidence is available in an example, we include it. This can help the model connect natural language to schema patterns or common SQL idioms.

We use two selection techniques. The first is a random baseline. It draws k examples uniformly at random from the pool. We use it for ablation and sensitivity checks. A fixed seed makes it reproducible. The second is question-similarity selection. It measures how close the user question is to each example question in an embedding space and picks the top-k. The steps are simple. We pre-compute vector embeddings for all example questions and cache them. At run time, we embed the user question, compute cosine similarity with each cached vector, sort by similarity, and select the highest-scoring k. Caching reduces latency, cost and keeps the selection stable across runs. This method tends to pick examples that share intent or structure with the input, which improves few-shot prompting in practice.

2.5.2 Stage II – Generation

The second stage is responsible for turning the retrieved context into executable SQL. In contemporary pipelines, this stage typically combines candidate query generation with an implicit or explicit selection process, so we treat them jointly. Given the question, the schema (filtered or enriched), and possibly few-shot demonstrations [67], an LLM or specialized generator model produces one or more candidate queries [55, 56, 58–60]. Diversity is often encouraged through prompt variation, stochastic decoding, or alternative schema views, increasing the chance that at least one candidate is correct.

Many systems augment raw generation with structured reasoning. They may decompose complex questions into simpler sub-questions, use chain-of-thought explanations or intermediate representations, or first predict key SQL components (e.g., tables, joins, predicates) before forming the final query [36, 54, 55, 58, 61, 63]. Some approaches rely on powerful general-purpose LLMs guided by task-specific prompts and demonstrations, while others use

fine-tuned models combined with in-context examples to better capture SQL syntax and domain conventions [59]. Once candidates are generated, systems select a final query using heuristics, self-consistency, execution signals, or learned ranking: they may choose the most frequently produced candidate, the one whose behavior is most stable, or the one favored by a discriminator or pairwise comparison model [55, 56, 58, 59, 61, 68–70].

VortoSQL SQL Generator: The SQL generator turns the composed prompt into a single SQL query. It reads the schema text from the schema linker, optional demonstrations from the example selector, and the user question (plus optional evidence). The output is one SQL string formatted for the target dialect (SQLite in our setup). We keep this stage simple and deterministic to make results reproducible.

We support two prompt modes. In *zero-shot* mode, the prompt contains only the task instruction, the schema description, and the user question. The template is shown in Appendix B. This mode has low token cost and gives a clean baseline. In *few-shot* mode, we prepend k demonstrations before the user question. Each demonstration includes a question, optional evidence, and a gold SQL. The template is shown in Appendix B. This mode often helps on compositional queries and rare patterns, but it increases prompt length.

Generation uses a chat-completion model with a fixed temperature (we use 0) and an optional random seed. We use the same settings for zero-shot and few-shot to make the comparison fair. After generation, we apply a light cleanup step to remove markdown fences, newlines, and extra spaces, so the query is a single executable line.

2.5.3 Stage III – Correction

The third stage focuses on improving robustness by detecting and repairing issues in the generated SQL. Correction can be applied to a single chosen query or interleaved with candidate exploration, but the underlying idea is consistent: use feedback signals to move from a plausible candidate to an actually executable and faithful query. Most systems leverage execution-based feedback by running candidate queries and inspecting syntax errors, runtime failures, or suspicious outputs (such as empty or inconsistent result sets), then prompting an LLM or applying rules to fix them [10, 54, 55, 58–60, 62–64, 68, 69]. Others incorporate model-based feedback, where verifier models or specialized prompts check constraints, detect logical inconsistencies, or compare candidates before suggesting edits [56, 71].

Depending on the design, correction may: (i) act as a lightweight post-processing step focused on syntax and formatting, (ii) refine candidates prior to selection using error logs and schema hints, or (iii) run as an iterative loop that alternates execution and regeneration until a

satisfactory query is found or a retry budget is exhausted [56, 58, 59, 63, 68, 69, 72]. In all cases, the correction stage is a key mechanism for turning imperfect LLM outputs into reliable SQL programs by systematically exploiting feedback from the database or auxiliary models.

VortoSQL SQL Corrector: The SQL corrector fixes syntax problems in the generated query before we run execution-based checks. It takes the current SQL, the schema text, and the DBMS dialect. We first try to parse the query. If parsing fails, we ask the model to repair it using a small prompt that shows the schema, the broken SQL, and the parsing error. The prompt is given in Appendix C. The model returns a revised query. We flatten whitespace and try to parse again. This loop runs for a fixed number of attempts. If parsing succeeds, we keep the fixed query; otherwise we keep the last version and mark correction as failed.

Table 2.1 Summary of VortoSQL components configurations by pipeline stage.

Stage	Key	Values / Description
Retrieval	**Schema Linker**	
	`technique`	`full_schema`, `tcsl`, `scsl`
	`chat_completion_model_provider`	`openai`, `ollama`
	`chat_completion_model_name`	e.g., `gpt-4o`
	`temperature`	float $[0, 2]$ (default 0 for stable output)
	`random_seed`	integer or `null`
	Example Selector	
	`technique`	`random`, `question_similarity`
	`number_of_examples`	integer k (e.g., 5)
	`embedding_model_provider`	`openai`, `ollama`
	`embedding_model_name`	e.g., `text-embedding-3-small`
	`random_seed`	integer or `null`
Generation	**SQL Generator**	
	`prompt_template`	`zero_shot`, `few_shot`
	`chat_completion_model_provider`	`openai`, `ollama`
	`chat_completion_model_name`	e.g., `gpt-4o`
	`temperature`	float $[0, 2]$ (default 0)
	`random_seed`	integer or `null`
Correction	**SQL Corrector**	
	`max_correction_attempts`	integer (0 disables correction)
	`prompt_template`	`syntax_correction`
	`chat_completion_model_provider`	`openai`, `ollama`
	`chat_completion_model_name`	e.g., `gpt-4o`
	`temperature`	float $[0, 2]$ (0 for deterministic output)

CHAPTER 3 LLM ROUTING FOR OPTIMIZING SQL GENERATION

In recent years, Text-to-SQL has gained significant momentum with deployments within enterprise solutions to transform data accessibility [73, 74]. Text-to-SQL democratizes access to structured data, allowing non-experts to interact with databases directly without requiring data engineering expertise. For SQL analysts, it enhances their workflows by supporting query authoring, dataset exploration, and report generation. A major use case lies in iterative data exploration, where the *complexity of user queries can range widely—from simple row retrievals to multi-way joins with aggregations*. Fig. 3.1 illustrates this contrast between easy and hard queries, both in natural language and their corresponding SQL.

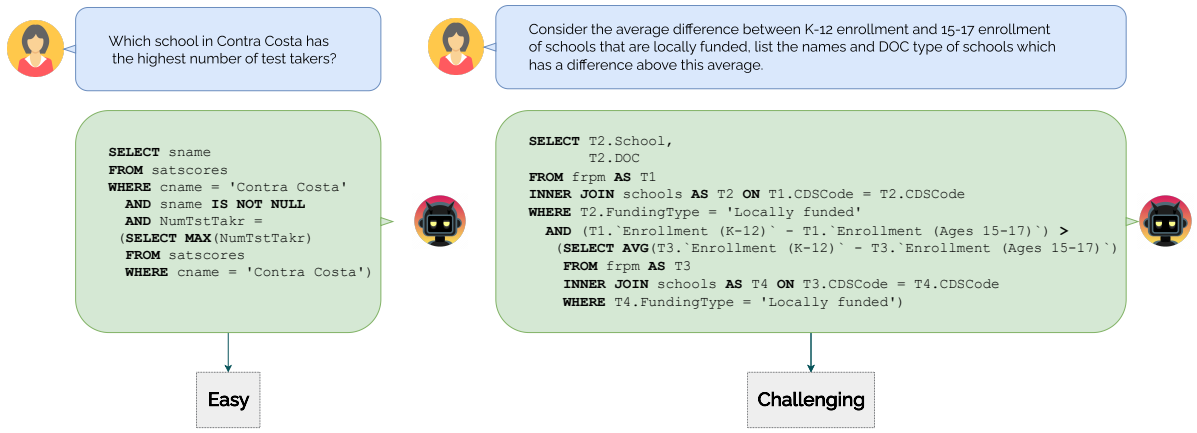


Figure 3.1 Examples of *easy* and *hard* queries in natural language and their SQL translations. Such variance in complexity motivates the need for routing rather than applying a single LLM to all queries.

State-of-the-art Text-to-SQL approaches follow a multi-stage pipeline [53, 75–77]. The pipeline consists of two main phases: (i) retrieval of contextual information–such as schema elements, examples, and instructions relevant to the query–and (ii) SQL generation. Enterprise solutions use capable LLMs for SQL generation to handle complex queries [1, 66]. While this is essential for complex queries, such capable LLMs introduce considerable latency and incur higher costs for simpler ones, such as inspecting a few rows from a table. This in turn, can negatively impact user experience and the average cost per query.

Leading Text-to-SQL benchmarks such as BIRD [7] rank submissions only based on accuracy while also indicating the model used on the leaderboard. Benchmarks typically assume a single-model approach, which mirrors the same inefficiencies found in enterprise deployment—namely, using the most capable models for all queries regardless of complexity. For instance, six of the top ten solutions on BIRD use `GPT-4o` or `Gemini`.

To handle the varying levels of complexity, *we investigate the implementation of LLM routers for Text-to-SQL* [78]. They route a query to the weakest, yet cheaper and faster model, capable of generating accurate SQL. We propose two routing approaches that achieve an accuracy close to that of the most capable LLM, always outperforming the second-best, and reducing costs by up to 1.4×. This cost reduction is substantial for enterprise deployments of analytics or SQL assistants with a high volume of queries. These routers are designed to be easy to train and efficient at inference. Fig. 3.2 depicts our pipelines with a router integrated within the SQL generation stage.

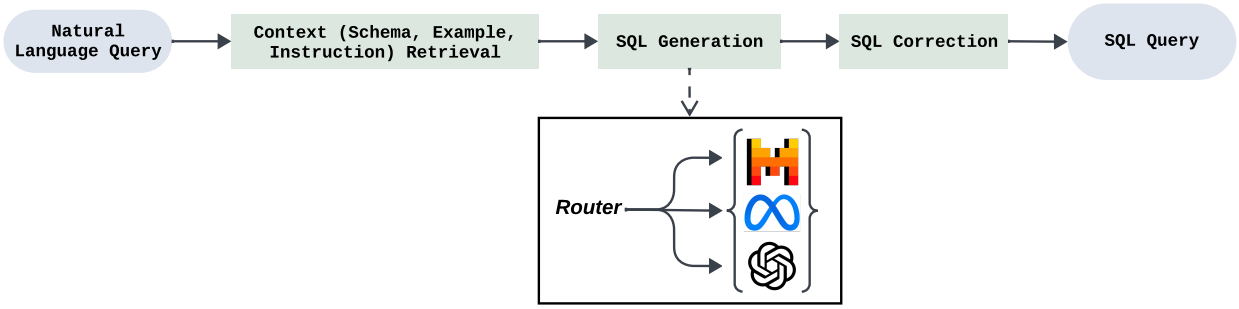


Figure 3.2 A multi-stage Text-to-SQL pipeline with an LLM router in the generation stage.

3.1 Preliminaries

3.1.1 Problem Formulation

Given a set of N models $\{ \mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_{N-1} \}$ and a natural language query Q, our objective is to identify the *weakest* model $\mathcal{M}_i$ that can accurately generate corresponding SQL for Q. Here, we define the weakest model as the one with the lowest SQL generation capability. In practice, this model typically exhibits the lowest latency and incurs the least dollar cost. We assume a total ordering of model strength, such that $\mathcal{M}_i$ is considered weaker than $\mathcal{M}_j$ if $i < j$. Therefore, we formulate the problem as an optimization task: select the smallest l such that $\mathcal{M}_l$ produces accurate SQL for Q.

Furthermore, we assume access to a dataset H, consisting of prior natural language (NL) queries, predicted SQL outputs, and corresponding ground-truth SQL for each of the N models. In practice, these examples can be collected from execution logs and manual labeling.

Accordingly, our goal is to learn an N-ary routing function that maps a query Q to the minimal label $l \in [0, N]$. Here, a label $l \in [0, N-1]$ indicates that $\mathcal{M}_l$ is the weakest model capable of generating accurate SQL for Q. A prediction of N denotes that no model within the set is capable of doing so.

3.1.2 Related Work

LLM Routing has been studied as a binary routing problem [79,80] for tasks such as question answering, summarization, and information extraction. To our knowledge, this is the first study of LLM N-ary routing and the first for the Text-to-SQL task.

Other prior work uses an ensemble of models for generation, as proposed by Jiang *et al.*, [81]. The approaches use multiple models to generate a set of answers instead of generating a single one and then use pairwise comparison and fusion of top-candidate to generate a final answer. While this approach is promising to maximize accuracy, it goes against the objective of cost minimization. A cascade approach has also been previously employed [82]. However, it cannot be used for Text-to-SQL. Starting with the weakest model, it is infeasible to determine whether the generated SQL is accurate or meets a quality threshold before falling back to the next available stronger model.

3.1.3 Metrics

To evaluate the effectiveness of generated SQL, we use *execution accuracy* (EX) as the primary metric [7,83]. Specifically, $EX(S)$ denotes the proportion of queries in the evaluation set S for which the output relation (from the predicted SQL) matches the ground-truth relation, where $0 \leq EX(S) \leq 1$. Here, matching relations are defined as those containing identical tuples, independent of attribute ordering. Our objective is to maximize $EX(S)$ while concurrently choosing the weakest model. In this work, we consider dollar cost as the cost we aim to minimize.

3.2 Methods

We consider two approaches. The first is a regression approach where we assign a score per model predicting the capability of generating accurate SQL for Q. We say a model is capable of generating accurate SQL if its score is above an input threshold. For all models above that threshold, we pick the weakest. The second is a classification approach where a router model predicts $l \in [0, N]$ as defined in the problem formulation.

If all scores of the regression approach are under the threshold or the classification approach predicts N, then we are predicting that no model is capable of generating accurate SQL. As such, dependent on the use case, we can decide to not attempt to generate SQL or we can route to one of the N models.

3.2.1 Score-based Routing

A regression-based router predicts for each model $\mathcal{M}_i$, a score $P(EX = 1_{\mathcal{M}_i}|Q)$, *i.e.*, the probability of generating accurate SQL for an input query Q. The router has the input parameter α: the score threshold such that if $P(EX = 1_i|Q) < \alpha$ then $\mathcal{M}_i$ is said to be incapable of generating the query. This router minimizes l such that $P(EX = 1_{\mathcal{M}_l}|Q) \geq \alpha$.

If the scores indicate that no model is capable of generating accurate SQL for Q, we can choose not to make an attempt or route to one of the models for a different accuracy-cost trade-off. To maximize accuracy in our implementation, we route to the strongest model.

We implement the scoring function using H for each of the N models. Given an input query Q, we first find $\mathcal{Q}_k$: the top-K similar queries in H. For each model $\mathcal{M}$, we set $P(EX = 1_{\mathcal{M}_i}|Q) = EX(\mathcal{Q}_k)_{\mathcal{M}_i}$. We then filter the models $(EX(\mathcal{Q}_k)_{\mathcal{M}_i} \geq \alpha)$ and pick the weakest. We denote this router based on its two parameters as R^K_α.

As summarized in Fig. 3.3, our score-based router computes per-model capability scores and routes to the weakest model above the threshold.

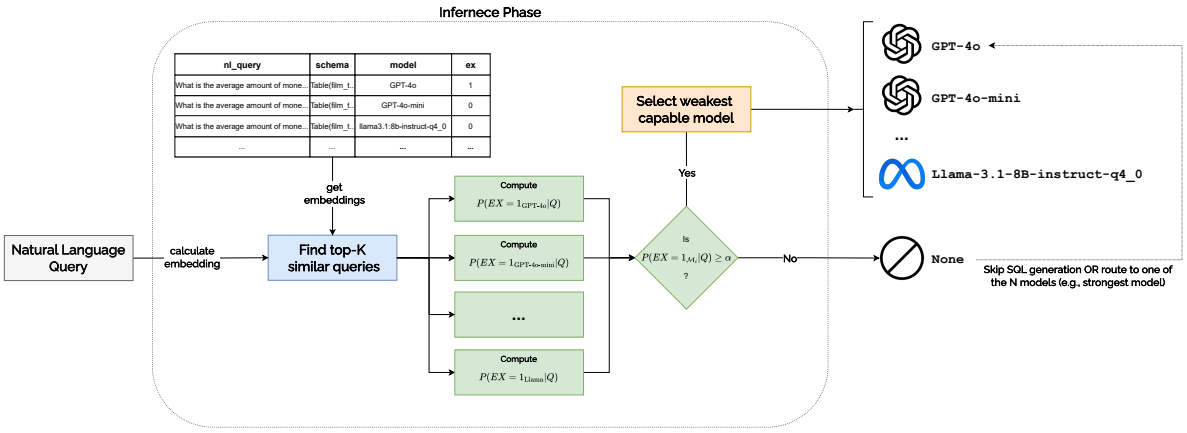


Figure 3.3 System design and data flow of the *score-based* router R^α_K. Given an input query Q, retrieve top-K neighbors from H, estimate $P(EX{=}1_{\mathcal{M}_i}|Q)$ for each model using neighbors' execution accuracy, filter by threshold α, and route to the *weakest* model meeting the threshold (or apply the None policy).

3.2.2 Classification-based Routing

We use a distilled `BERT`-style encoder model (`DistilBERT` [84]), which we train on H to learn the function R_{BERT}. Given an input query Q, we input the NL query and relevant retrieved schema to predict the label l indicating the weakest model $\mathcal{M}_l$ predicting $EX(Q) = 1_{\mathcal{M}_i}$. If R_{BERT} predicts N then it is predicting that none of the models can generate accurate SQL.

Similarly to the other router, to maximize accuracy in our implementation, we choose to route to the strongest model.

Fig. 3.4 outlines the classification-based router that predicts the weakest capable model.

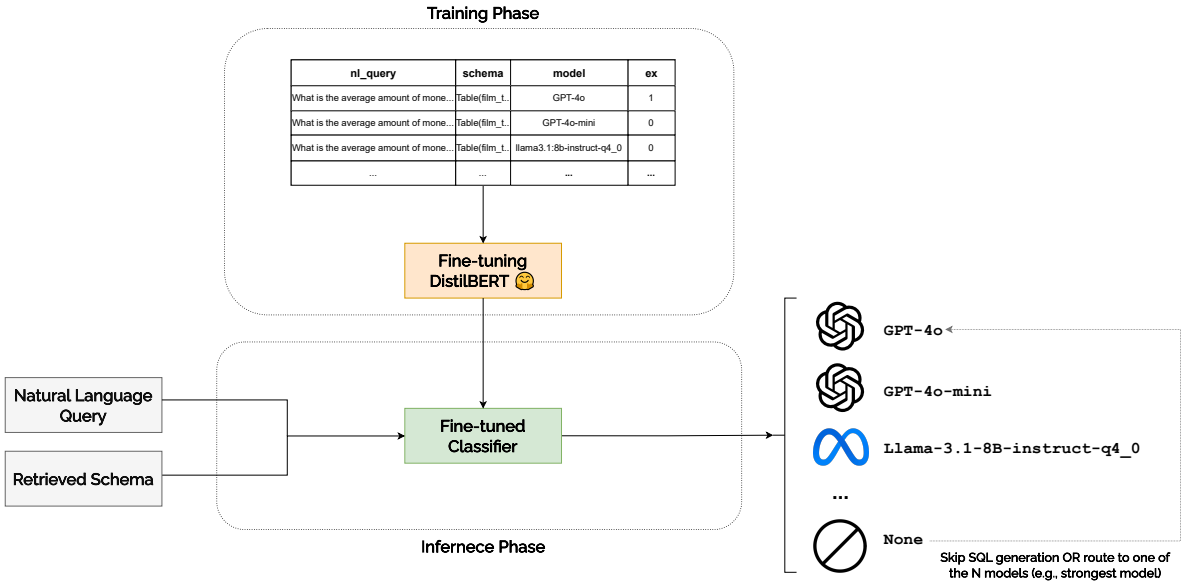


Figure 3.4 System design and algorithmic flow of the *classification-based* router R_{BERT}. During training, a distilled BERT encoder ingests (NL query, retrieved schema) with labels $l \in [0, N]$ derived from H. At inference, it predicts l indicating the *weakest* model $\mathcal{M}_l$ expected to achieve $EX{=}1$ (or N for None), after which the routing policy is applied.

3.3 Experiments

3.3.1 Datasets

We conducted our experiments using the BIRD dataset [7], which is widely considered to be the most challenging Text-to-SQL benchmark. Our evaluation set consisted of all 1534 queries in the dev dataset and our training set, acting as H, consisted of 9428 queries.

3.3.2 Models

Specific models and strength. We used three different LLMs for SQL generation: i) `GPT-4o`; ii) `GPT-4o-mini`; and iii) Llama: `Llama-3.1-8B-instruct-q4_0`. To order their SQL generation capability, we used a simplified Text-to-SQL pipeline consisting of a single attempt at generation while adding the whole schema to the LLM context. We evaluate the pipeline by selecting 10% of the queries of each database in BIRD uniformly at random.

We find `Llama` as the weakest model with EX 0.34, then `gpt-4o-mini` with EX 0.48, and `gpt-4o` as the strongest with EX 0.55.

Cost. In our experiments, we only analyzed dollar cost and forgo latency due to setup challenges. We use `Llama` as a local model and hence associate no dollar cost with it and use OpenAI services to access `gpt-4o` and `gpt-4o-mini`. Instead of using current token prices of OpenAI, we use a normalized unit cost where we set `gpt-4o-mini` input and output token cost to 1 and `gpt-4o`to 16.6× (multiplicative price difference).[1] We also used OpenAI's `text-embedding-3-small` model with cosine similarity to find the top K similar queries in H for the score-based router. The embedding cost is negligible per input NL query when compared with token usage and is therefore ignored.

Score-based Router Implementation. For a score-based router (R_k^α), we have to select the two parameters α and K. α (threshold score) is the minimum proportion out of the K similar queries for which a model had to generate accurate SQL to be considerate a candidate model for generation. We run a grid search over 10% of train queries as done before and for each input query, we remove the queries in the same database from being chosen as similar ones. We pick $\alpha > 0.5$ as it means intuitively some level of robustness where more than half of the K similar queries were generated successfully. For each $\alpha \in \{0.6, 0.7, 0.8, 0.9\}$, we do a search over $k \in [5, 50]$. We find that setting α to 0.9 is overly restrictive indicating N for every input query, *i.e.*, no model is capable of generate accurate SQL. As such, we only consider $\{0.6, 0.7, 0.8\}$. We choose the K that maximizes EX while having routed to each model at least once. We end up with three models with a difference less than 0.5% EX between them: $R_{24}^{0.6}$, $R_{25}^{0.7}$, and $R_{10}^{0.8}$.

Classification-based Router Implementation. We fine-tuned `DistilBERT` [84] on the fully labeled train set H. We removed from H, all queries in the retail_world database due to schema missing and all duplicate user questions. We split H 80-20 on db_id first (ensuring databases in train are not in validation and vis-versa). This led to a train set of 7346 queries and a validation set of 1697. When training on H and evaluating on the dev set, to retrieved the relevant schema using the TCSL schema linking approach [56,66]. We trained for 4 epochs, with a batch size of 32, and a learning rate of $1.00E{-}04$. We sampled each batch using a weighted random sampler, the weights correspond to the inverse of the frequency of each label within to the training data.

Table 3.1 EX%, model distribution, and cost compared to `gpt-4o` for each generation approach on the BIRD dev set, *i.e.*, three routers from above and the use of the three base models: (i) `gpt-4o`; (ii) `gpt-4o-mini`; and (iii) `Llama`: `llama3.1:8b-instruct-q4_0`.

Gen.	**EX%**	`gpt-4o`	`gpt-4o-mini`	`Llama`	`None`	**$ Reduc.**
`4o`	61.02	1534 (100%)	-	-	-	**1x**
`4o-mini`	49.22	-	1534 (100%)	-	-	**16.6x**
`Llama`	29.34	-	-	1534 (100%)	-	∞
$R_{25}^{0.7}$	60.14	197 (13%)	88 (6%)	5 (0%)	1243 (81%)	**1.1x**
$R_{10}^{0.8}$	59.42	160 (10%)	127 (8%)	26 (2%)	1220 (80%)	**1.1x**
$R_{24}^{0.6}$	57.92	324 (21%)	265 (17%)	54 (4%)	890 (58%)	**1.3x**
R_{BERT}	55.21	118 (8%)	311 (20%)	167 (5%)	938 (61%)	**1.4x**

3.4 Results

We evaluate our routers $R_{24}^{0.6}$, $R_{25}^{0.7}$, $R_{10}^{0.8}$, and R_{BERT} on all 1534 dev queries. Recall that all models, route to the strongest model (`gpt-4o`) in case N is predicted.

By analyzing the successful and failed query sets on the three base models (Details in Section 3.4.1), we find that each weaker model has a large subset of failed and successful queries with a stronger one. As such, we expect routing to lower the cost while at best keeping the best model's EX.

Table 3.1 summarizes the EX, model routing distribution and relative cost to the strongest model (`gpt-4o`) for each generation approach on BIRD's dev set. Our routers are up to 1.4x cheaper while being close in EX. With both routers, we lose some accuracy for a lower cost. The accuracy–cost trade-off on BIRD is easiest to explain through the parameters of the score-based router (Section 3.4.2).

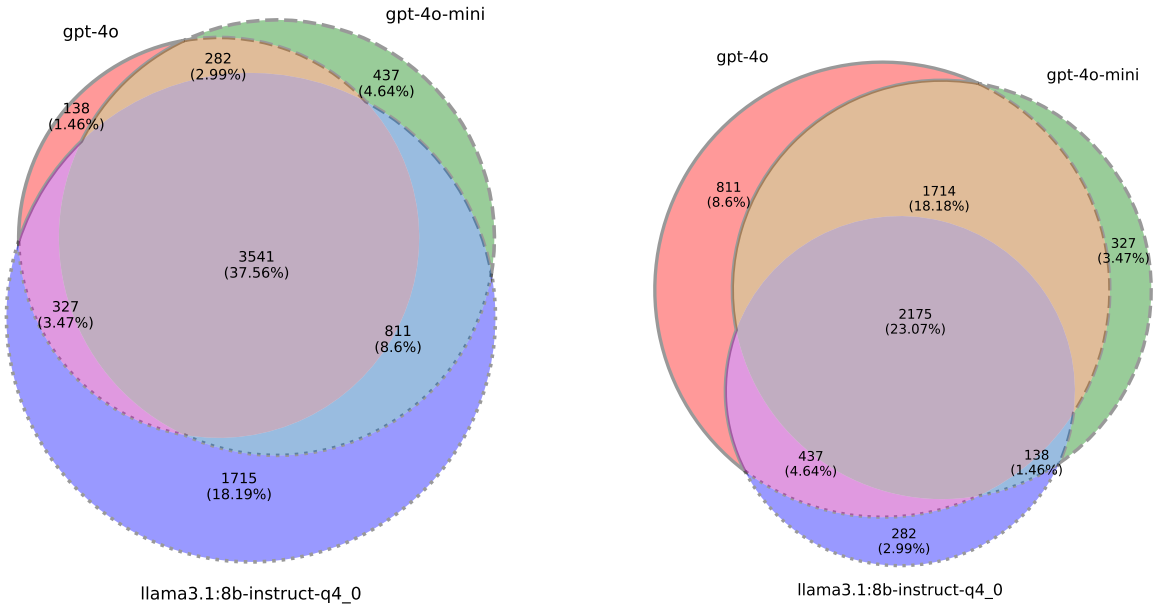
In this ongoing work, we aim next to assess whether an NL query with relevant schema is enough to classify its complexity as done in R_{BERT}. Furthermore, we plan to explore routing across more datasets and within more stages such as correction and schema linking instead of just generation.

3.4.1 Failure Case Analysis

By analyzing successful and failed query sets on the three base models, we find substantial overlap in both failures and successes, which constrains the headroom for EX improvements via routing (while still enabling cost reductions).

The difference in EX shows a clear gap in capability between models. We find that 37.56% of

[1]Numbers obtained based on OpenAI's offering as of 1 Oct. 2024



(a) Venn Diagram of Failure Cases

(b) Venn Diagram of Correct Cases

Figure 3.5 Distribution of Failure and Correct Cases across `gpt-4o`, `gpt-4o-mini`, and `llama3.1:8b-instruct-q4_0`.

Table 3.2 Failure Cases across Models

Failure Cases	**Count**	**Percentage**
`gpt-4o`	4288	45.49%
`gpt-4o-mini`	5071	53.79%
`llama3.1:8b-instruct-q4_0`	6394	67.83%
Intersection (all three)	3541	37.56%
Intersection (`gpt-4o`, `gpt-4o-mini`)	3823	40.55%
Intersection (`gpt-4o`, `llama`)	3868	41.03%
Intersection (`gpt-4o-mini`, `llama`)	4352	46.17%

the queries failed across all three models with a high common failed percentage of queries between every pair from 45.49%–67.83%. This indicates that when routing, it is unlikely to improve EX beyond that of the strongest model and that we expect a cost decrease only.

Our routers $R_{31}^{15.5}$ and R_{BERT} reach an EX 53.48 and 56.91, respectively. This is compared to a maximal EX of 61.02 when using only 4*o*. Generally, as the router distribution contains less of the strongest model, the EX decreases but also comes with more cost reduction. When we analyzed the failing query set, we found that the overlap between the three base models is rather large as well.

Table 3.3 indicates the input token usage as well as shows an associated cost units (in K). The number of output tokens is negligible in comparison to the input. We normalize the dollar cost as follows. We define based on relative cost such that `llama3.1:8b-instruct-q4_0` (`Llama`) tokens cost 0, and such that single input tokens for `GPT-4o-mini` (`gpt-4o-mini`) cost 1, and single input token for `GPT-4o` (`gpt-4o`) cost 33, respectively. Our routers are 1.4x and 1.8x cheaper while again being close in EX to `gpt-4o`. Beyond Table 3.1 and Table 3.3, we also report the latency in seconds in Table 3.4, showing the small added runtime overhead to the overall pipeline.

Table 3.3 Number of input tokens (in K) per model for each generation routing approach as number of output tokens is negligible in comparison. We define cost (in K) based on our deployment with relative costs such that `llama3.1:8b-instruct-q4_0` (`Llama`) tokens cost 0, single input tokens for `GPT-4o-mini` (`gpt-4o-mini`) cost 1, and single input token for `GPT-4o` (`gpt-4o`) cost 33. We also show in parenthesis the relative decrease compared with `gpt-4o`.

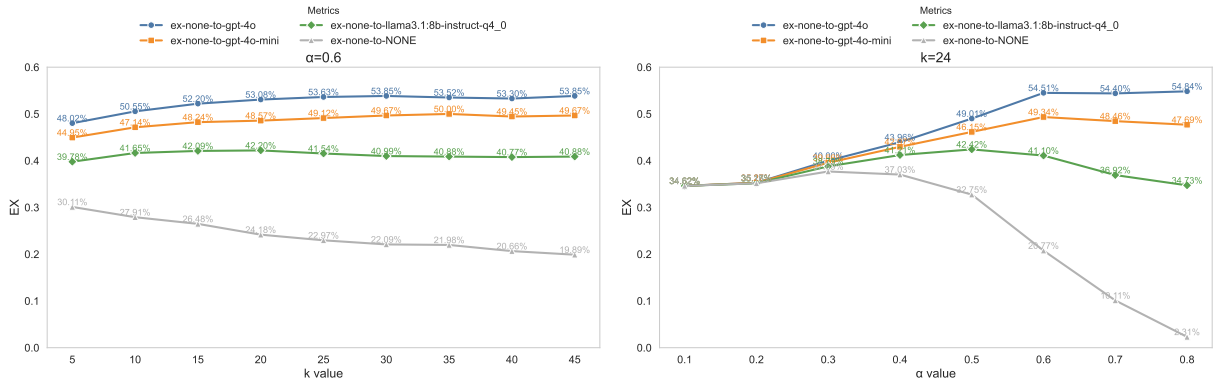
	# Queries, # (1K) Input Tokens			
Generation	`gpt-4o`	`gpt-4o-mini`	`Llama`	**Cost Units (in K)**
`gpt-4o`	1534, 4720	-	-	155.76
`gpt-4o-mini`	-	1534, 4720	-	4.72 **(33.3x)**
`Llama`	-	-	1534, 4720	0 (∞)
$R_{0.8}^{15.5}$(`gpt-4o`)*	2560	1739	531	86.22 **(1.80x)**
$R_{0.7}^{15.5}$(`gpt-4o-mini`)	1208	3091	531	42.96 **(3.62x)**
$R_{0.6}^{24}$(`Llama`)	1208	1739	1932	41.60 **(3.74x)**
$R_{31}^{15.5}$(`None`)	1208	1739	531	41.60 **(3.74x)**
R_{BERT}(`gpt-4o`)*	3317	1164	239	110.63 **(1.4x)**
R_{BERT}(`gpt-4o-mini`)	629	3853	239	24.61 **(6.3x)**
R_{BERT}(`Llama`)	629	1164	2927	22.92 **(6.8x)**
R_{BERT}	629	1164	239	22.92 **(6.8x)**

Table 3.4 Average added latency (in seconds) for score-based and classification-based routers on BIRD dev set. The reported values include standard deviation (±).

Router	Avg. Latency (s)	Std. Dev. (s)
Score-based (R_K^α)	0.47	±0.022
Classification-based (R_{BERT})	1.16	±0.349

3.4.2 Effect of K and α on EX and Model Distribution

We analyze the impact of varying parameters K (number of similar queries) and α (threshold) in the score-based router R_K^α on EX and distribution. This shows the cost–accuracy trade-off.



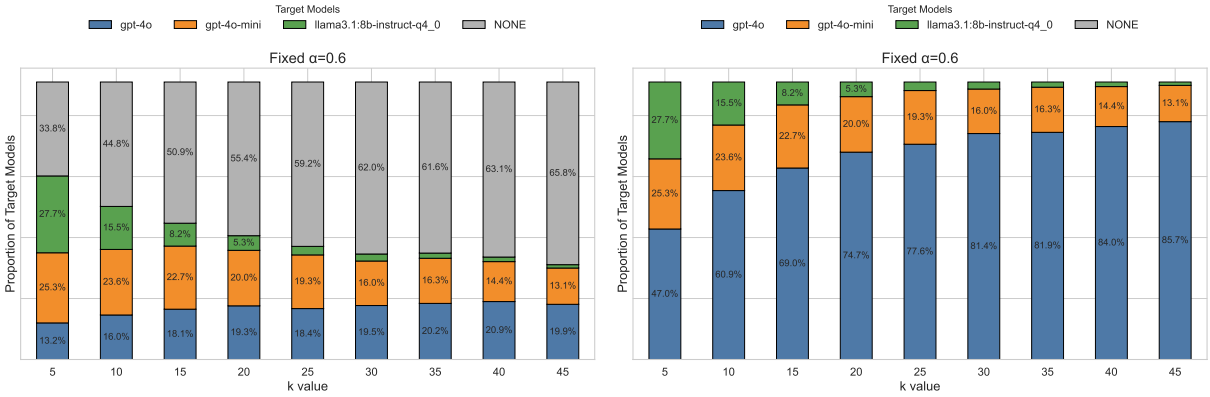
(a) EX as K changes

(b) EX as α changes

Figure 3.6 Execution Accuracy for a score-based router with different `None` routing strategies.

Our empirical observations are based on 10% of BIRD's train set, summarized in Figures 3.6, 3.7, and 3.8. Higher values of K or α lead to an increase in execution accuracy (EX) but result in a higher proportion of queries being routed to the strongest model `gpt-4o`, thereby increasing overall cost. Conversely, lower values of K or α favor routing to cheaper models like `gpt-4o-mini` and `Llama`, reducing cost but potentially decreasing EX due to their weaker performance.

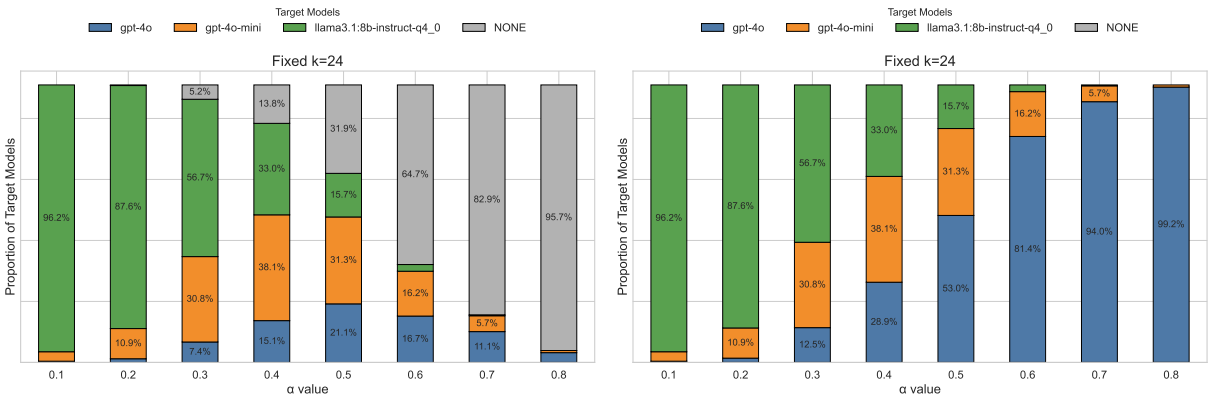
In practical applications, selecting appropriate values for K and α is crucial to balance the need for high execution accuracy with cost efficiency. By adjusting these parameters, practitioners can tailor this score-based router to meet their specific requirements.



(a) With None prediction

(b) `None` routed to `GPT-4o`

Figure 3.7 Effect of varying K on execution accuracy and on model distribution.



(a) With None prediction

(b) `None` routed to `GPT-4o`

Figure 3.8 Effect of varying α on execution accuracy and on model distribution.

CHAPTER 4 FINE-GRAINED METRICS AND QUERY MUTATION FOR EFFECTIVE EVALUATION

Natural language interfaces to databases aspire to democratize data access. They allow users to express information needs in natural language (NL) and obtain answers directly from structured data. At the core of this vision lies the *Text-to-SQL* task, which translates NL queries into executable SQL. While the task has witnessed remarkable progress [74], the advent of large language models (LLMs) has substantially improved translation accuracy and spurred the development of new techniques. This momentum has, in turn, driven the rapid emergence of benchmarks such as Spider [8] and BIRD [7], and enabled early, controlled production deployments.

Despite this progress, current evaluation practices remain far from capturing the complexity of real-world enterprise workloads. Enterprise queries often span multiple tables, exhibit intricate join structures, and reference schema elements with domain-specific terminology and abbreviations. In contrast, public benchmarks typically feature short, structurally simple queries defined over intuitive schema names. This leads to an overly optimistic assessment of system capabilities, thereby distorting cross-system comparisons. Recent efforts such as the Beaver benchmark [85] attempt to mitigate these limitations by introducing more complex schema. Nevertheless, progress remains incremental and labor-intensive, advancing only through the development of new benchmarks one at a time.

Evaluation methodology is further constrained by the metrics used to measure system performance. The dominant standard, *Execution Accuracy* (EX), assigns a score of 1 when the predicted query's output relation exactly matches that of the ground-truth query and 0 otherwise. While intuitive, this binary measure collapses a wide spectrum of outcomes into a single judgment, failing to capture partial correctness or provide diagnostic feedback. A query that retrieves nearly all correct tuples but misses one condition is indistinguishable from one that returns an irrelevant result. Similarly, structural differences such as column reordering or inclusion of irrelevant columns are penalized equally. Consequently, EX provides only a coarse, outcome-based view of correctness. This coarseness leaves evaluation misaligned with real-world requirements, where understanding the degree to which a system errs, and which components of a Text-to-SQL system succeed or fail, is essential.

Evaluating Text-to-SQL systems on private datasets offers a more realistic alternative. The final choice of a production system should ultimately depend on its accuracy over the target workload. However, constructing high-quality evaluation sets is costly: it demands extensive

manual annotation, domain expertise, and, increasingly, reliance on LLMs for automation. These processes are expensive, non-deterministic, and offer limited control over output, making reproducible and effective evaluation difficult. As a result, evaluation in both public and private settings remains fragmented and resource-intensive.

These challenges stem from a common limitation: current evaluation practices remain static, relying on fixed benchmark queries in the form of NL–SQL pairs and coarse binary metrics. To address these challenges, we reconceptualize Text-to-SQL evaluation as a dynamic process and introduce a new evaluation framework that automatically generates diverse, semantically valid query variants to target *choke points* across key system components, and introduces new execution-level metrics that move beyond binary correctness to capture both over- and under-prediction behaviour. By enabling controlled query mutations and richer evaluation signals, our method transforms Text-to-SQL evaluation into a more active process of systematic stress-testing and diagnosis.

4.1 Experimental Setup

Dataset

We use the BIRD benchmark [7], a large-scale, cross-domain dataset that has become a de facto standard for Text-to-SQL evaluation. BIRD contains 12,751 NL–SQL pairs across 95 databases spanning 37 domains. Each NL query may include *evidence*, *i.e.*, external text that explains schema attributes, values, or computations, offering richer context for queries within specific domains.

The dataset is divided into three splits: training (9,428 queries over 69 databases), development (dev) (1,534 queries over 11 databases), and hidden test (1,789 queries over 15 databases). We conduct all analyses on the development set, and to reduce experimental cost we also use BIRD's *mini-dev* subset, consisting of 500 high-quality NL–SQL pairs uniformly sampled from the dev set.

Models

Unless otherwise specified, we use `GPT-4o` and `text-embedding-3-small` from OpenAI as the large language and embedding models of choice, respectively.

Metrics

We adopt *Execution Accuracy (EX)* as the primary evaluation metric. EX compares the output relation of a predicted query $\hat{R}_i$ with that of the ground-truth query R_i, assigning a score of 1 for an exact match and 0 otherwise. The overall score across N queries is computed as the average over all evaluated queries:

$$EX = \frac{1}{N}\sum_{i=1}^{N}\mathbf{1}(R_i = \hat{R}_i) \text{ where } \mathbf{1}(R_i, \hat{R}_i) = \begin{cases} 1 & \text{if } R_i = \hat{R}_i \\ 0 & \text{otherwise.} \end{cases}$$

While this metric is standard, later sections (§4.2) introduce SQLMorph's fine-grained alternatives that capture partial correctness and aid in error diagnosis.

Text-to-SQL Systems

For our experiments, we rely on three representative open-source text-to-SQL systems that differ in architecture, retrieval strategies, and correction mechanisms. Table 4.1 summarizes their performance on the BIRD leaderboard at the time of submission. Their approaches can be summarized as follows:

1. DIN-SQL [54]. It performs schema retrieval to align NL mentions with tables, columns, and values, then classifies queries as easy, non-nested complex, or nested complex to adapt prompting strategies. It employs the intermediate representation NatSQL [44] and applies a final self-correction stage via LLM prompting.
2. MAC-SQL [55]. It follows the three-stage pipeline from Section 2.2, while adding a *decomposer* to break complex NL queries into subproblems, generate subqueries for each at generation, and then merge them.
3. CHESS [56]. It augments the pipeline with an information retriever (*IR*) that extracts keywords, values, and contextual hints from the NL query and schema. Its candidate generator (*CG*) repairs syntax errors, while a *unit tester (UT)* executes candidates and selects the one passing the most validation checks. We adopt $\text{CHESS}_{\text{IR+CG+UT}}$, the strongest reported configuration.

This setup allows us to evaluate the three primary contributions of SQLMorph: Fine-Grained Execution Metrics, and Textual Query Augmentation.

Table 4.1 Performance of evaluated systems on the BIRD.

System	**EX (%)**	
	Dev	**Test**
Human (upper bound)	–	93.0
$\text{CHESS}_{\text{IR+CG+UT}}$	68.3	71.2
MAC-SQL	57.6	59.6
DIN-SQL	50.7	55.9

4.2 Fine-Grained Evaluation Metrics

4.2.1 Overview

The standard *Execution Accuracy* (EX) metric is binary: the predicted and reference result relations either match exactly or they do not. This collapses a spectrum of outcomes into a single bit and obscures essential signal. Queries that recover most rows but miss a boundary condition receive the same score as queries that return irrelevant results; structural differences (*e.g.*, harmless extras columns) are penalized as harshly as semantic errors. As a consequence, EX masks partial correctness, blurs error diagnosis, and offers little guidance on where a Text-to-SQL system fails (under- or over-prediction of rows, columns, or values).

Our proposal We introduce a family of *fine-grained and relaxed, execution-level* metrics that quantify the deviation from the ground-truth, not just *whether* it exactly matches. Two complementary measures lie at the core: *Execution Precision* (EXP), the fraction of predicted cells that are correct, and *Execution Recall* (EXR), the fraction of ground-truth cells that are recovered. F1 summarizes both, while separate reporting of EXP and EXR shows over-prediction with lower precision) and under-prediction with lower recall.

4.2.2 Approach

Given a ground-truth query q and a predicted query q', we execute both to obtain column name sets $c(q), c(q')$ and row multisets $GRows, PRows$. If $GRows$ and $PRows$ are identical over their full schemas, then EX = 1 and we set EXP = EXR = F1 = 1. Otherwise, we proceed with relaxed matching, which first optionally matches the columns then the rows and cells. Finally, a choice is made on whether to penalize or not extra columns.

Column Matching We compare only over columns the evaluator deems possible to match; we offer three matching regimes:

- **Exact-Column (EC).** Match by exact column name: $c_\cap = c(q) \cap c(q')$.
- **Semantic-Column (SC).** Build a textual descriptor for each column (name plus top-k values), embed descriptors, restrict to type-compatible pairs, and compute a maximum-weight bipartite matching. Keep pairs above a similarity threshold (*e.g.*, 0.7).
- **No-Column (NC).** Skip matching entirely; For we rely on partial cell matching (part of a row) below.

Row/Cell Matching Let $c_\cap = c(q) \cap c(q')$ denote the matched column set (EC/SC) or implicit value set (NC):

- **Exact-Cell (EC).** After establishing $c_\cap$, project both Pred and GT rows onto these aligned columns. Treat projected rows as multisets. For each unique row pattern r, let $f_G(r)$ and $f_P(r)$ be its multiplicities. Then

$$|\text{MatchedRows}| = \sum_r \min(f_G(r), f_P(r)),$$
$$|\text{MatchedCells}| = |\text{MatchedRows}| \cdot |c_\cap|.$$

  Intuition: Only exactly identical rows (after projection) receive credit, and each such row contributes one full row of cells.

- **Partial-Cell (PC).** Project both GT and Pred rows onto $c_\cap$. Matching is performed in two phases:
  - Phase 1 - exact matching: For each row pattern r over $c_\cap$, match exactly $\min(f_G(r), f_P(r))$ copies.

$$|\text{MatchedRows}| = \sum_r \min(f_G(r), f_P(r)),$$
$$|\text{ExactMatchedCells}| = |\text{MatchedRows}| \cdot |c_\cap|.$$

    Remove matched multiplicities.
  - Phase 2 - partial matching: For each remaining predicted row p and ground-truth row g, compute similarity

$$\text{sim}(p, g) = \frac{\#\{i \in c_\cap : p_i = g_i\}}{|c_\cap|}.$$

Greedily select the pair with the highest similarity, add $\{i \in c_\cap : p_i = g_i\}$ to PartialMatchedCells, and remove both rows. Stop when maximum similarity is 0.

Finally, the total number of matched cells (|MatchedCells|) is |ExactMatchedCells| + |PartialMatchedCells|

Accounting for extra predicted columns We define $|GCells| = |GRows| \cdot |c(q)|$ as the total number of ground-truth cells and account for extra predicted columns by making a choice between:

- **Penalize Extras (PE) Pred Cols.** $|PCells| = |PRows| \cdot |c(q')|$. All predicted columns are counted, so extra columns reduce precision if they do not match ground-truth.
- **Ignore Extras (IE) Pred Cols.** $|PCells| = |PRows| \cdot |c_\cap|$. Only intersection columns are counted, under the assumption that the predicted table already contains the needed data, so extra columns can be ignored. This option is not available if you did not perform column matching (NC), as there are no common columns $c_\cap$.

Metrics

The final metrics are:

$$\text{EXP} = \frac{|\text{MatchedCells}|}{|PCells|}$$

$$\text{EXR} = \frac{|\text{MatchedCells}|}{|GCells|}$$

$$\text{F1} = \frac{2 \cdot \text{EXP} \cdot \text{EXR}}{\text{EXP} + \text{EXR}}$$

Summary: Precision is the fraction of predicted cells that are correct, recall is the fraction of ground-truth cells recovered, and $F1$ balances the two. We can choose to penalize extra predicted columns or ignore such predictions.

4.2.3 Experimental Analysis

Table Shape Sensitivity

Research Question – Do granular metrics consistently respond to structured expansions and reductions in predicted SQL outputs along both table dimensions?

This experiment serves as a mechanical sanity check: we perturb the shape of the predicted result along the row and column axes while holding the ground-truth (GT) table fixed, and we verify that EXP and EXR respond in the expected directions. The design is inspired by [86] but offers finer control over both axes.

We began with `question_id=3` in the `california_schools` database from the BIRD benchmark, a simple SQL query returning a single-row, single-column result: the mailing address (`MailStreet`) of the school with the highest number of FRPM-eligible K–12 students. To increase controllability over both rows and columns, we restructured this query using a subquery-based design. The subquery performs the original join and selects 9 columns and 9 rows, creating a rich intermediate result table. The outer query (which we manipulate) then selects a subset of this table. Our ground-truth query selects 5 columns and applies `LIMIT 5`, giving us a 5×5 ground-truth result table.

```
SELECT sub.col1, sub.col2, ..., sub.colN
FROM (
  SELECT
    T2.MailStreet AS col1,
    T2.School AS col2,
    T2.MailCity AS col3,
    T2.MailState AS col4,
    T2.Zip AS col5,
    T1.`FRPM Count (K-12)` AS col6,
    T1.`Enrollment (K-12)` AS col7,
    T1.`Free Meal Count (K-12)` AS col8,
    T1.`Percent (%) Eligible Free (K-12)` AS col9
  FROM frpm AS T1
  JOIN schools AS T2 ON T1.CDSCode = T2.CDSCode
  ORDER BY T1.`FRPM Count (K-12)` DESC
  LIMIT 9
) AS sub
LIMIT M;
```

The inner subquery always produces a fixed 9×9 result (see Fig. 4.1). The ground-truth query selects 5 columns and applies `LIMIT 5`, giving a 5×5 GT result table. The outer query controls shape by selecting N columns and applying `LIMIT M`.

From this setup, we constructed seven groups of variants, each manipulating the number of columns, rows, or both, across a range of 1 to 9, while holding the other dimension fixed (or co-varying it). This design lets us study both precision (EXP) and recall (EXR) dynamics as

predictions deviate from the ground-truth answer. The set includes 63 variant queries across 7 groups (Table 4.2). Figures 4.2a and 4.2b show metrics under the SC-PC evaluation setting.

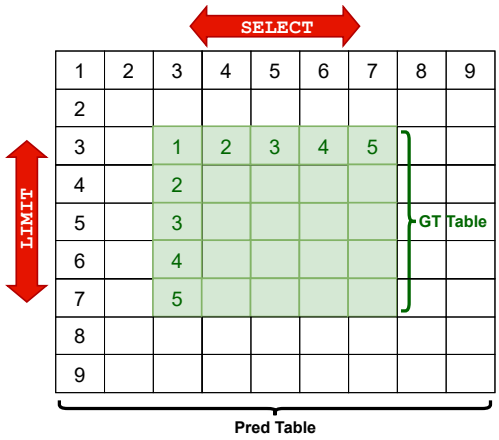


Figure 4.1 Schematic of the table shape manipulation.

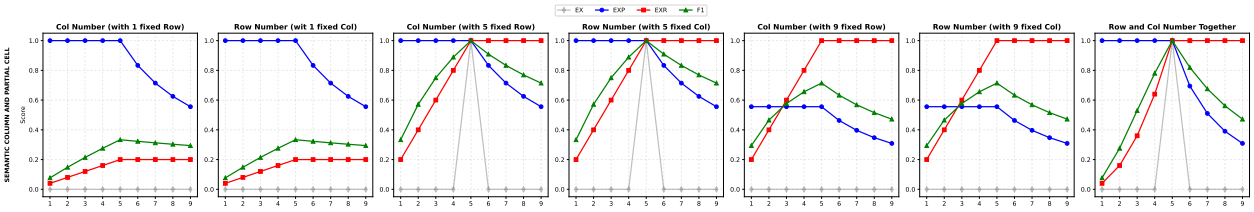
Table 4.2 Tracks in the Table Shape Sensitivity experiment, with axes and query recipes.

#	Track	What we vary	Outer-query recipe (Axis stressed)
1	Col Number (1 fixed Row)	$1 \rightarrow 9$ cols (rows=1)	keep `LIMIT 1`; add one col each step
2	Row Number (1 fixed Col)	$1 \rightarrow 9$ rows (cols=1)	keep single col; raise `LIMIT`
3	Col Number (5 fixed Row)	$1 \rightarrow 9$ cols (rows=5)	widen schema
4	Row Number (5 fixed Col)	$1 \rightarrow 9$ rows (cols=5)	raise `LIMIT`
5	Col Number (9 fixed Row)	$1 \rightarrow 9$ cols (rows=9)	widen schema
6	Row Number (9 fixed Col)	$1 \rightarrow 9$ rows (cols=9)	raise `LIMIT`
7	Row and Col Number Together	$1/1 \rightarrow 9/9$	grow both axes simultaneously

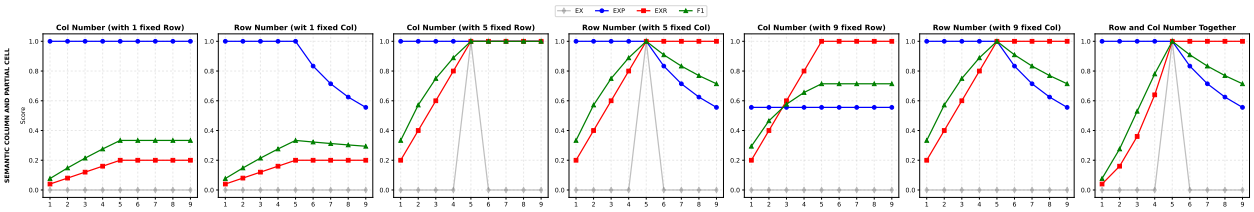
Analysis: This experiment changes the number of predicted rows ($|PRows|$) and columns ($|c(q')|$) while keeping the ground-truth fixed, to see whether EXP and EXR behave as expected under controlled table-shape variations.

Row axis (schema fixed). When $|PRows| < |GRows|$, the prediction misses some ground-truth rows. In this case, EXR grows with $|PRows|$ as more GT rows are recovered, while EXP ≈ 1 since the included rows are correct. When $|PRows| \geq |GRows|$, all GT rows are covered and EXR levels off 1. Extra rows lower EXP roughly as $|GRows|/|PRows|$, so F1 peaks when $|PRows| \approx |GRows|$ and then declines.

Column axis (rows fixed). When fewer than $|c(q)|$ columns are predicted, EXR drops in proportion to the missing GT cells. When more columns are predicted than in $c(q)$, the effect depends on the setting: under PE, EXP decreases because $|PCells| = |PRows| \cdot |c(q')|$



(a) Penalizing extra predicted columns (PE).



(b) Ignoring extra predicted columns (IE).

Figure 4.2 Graphs showing the trends of metrics for systematic table-shape variations.

grows; under IE, extra columns are ignored, so EXP remains unchanged. The "knees" occur at $|PRows| = |GRows|$ and $|c(q')| = |c(q)|$.

Across all seven tracks (Fig. 4.2), the expected patterns appear: – In row-only tracks, F1 peaks when $|PRows| = |GRows|$. – In column-only tracks, the gap between PE and IE emerges when $|c(q')| > |c(q)|$, while EXR stays aligned if all GT columns are present. – In the combined track, recall first improves as missing rows are added, then precision drops once surplus rows/columns appear. Because the fixed axis varies across tracks (sometimes set to 1, sometimes to $|GRows|$, sometimes to 9), the knees shiscft accordingly: when the fixed value equals the ground-truth, the knee falls exactly at the gold point; when fixed smaller or larger, the curves are offset but still follow the same piecewise-monotone shape. In this experiment, all configurations produce identical curves because after the column alignment and exact-cell matching phases, no further overlaps remain for partial matches; so we only plot one representative curve.

Under column alignment and exact-cell matching, each matched row contributes a full set of $|c(q)|$ cells. Thus, at the row knee ($|PRows| = |GRows|$), EXR $= 1$ and for $|PRows| > |GRows|$, EXP $= |GRows|/|PRows|$ with F1 $= 2|GRows|/(|GRows| + |PRows|)$. In column widening, PE vs. IE only diverges in EXP once $|c(q')| > |c(q)|$. Curve areas also align with intuition: EXR grows until coverage is complete, while EXP declines once spillover starts.

Takeaway. The metrics are *shape-aware*: on rows they distinguish *coverage* from *spillover*, on columns they separate *schema coverage* from *schema spillover*. This captures graded behavior that binary EX misses.

Controlled Error Sensitivity

Research Question – Can granular metrics capture and differentiate the impact of isolated, single-operator errors in predicted SQL queries?

To test metrics, we construct a benchmark of single-error SQL variants. Each mutant is derived by applying exactly one atomic mutation operator to a ground-truth query from the BIRD dev set and are summarized in Table 4.3. In mutant generation code, first we enforce depth-1 mutation by exhaustively attempting each operator and retaining only those that modify the AST and yield syntactically valid SQL. All mutations target only the outer query layer in case the query contains subqueries, ensuring controlled and isolated perturbations *across query structures*. Our mutation strategy uses a curated set of atomic operators covering key SQL components as follows:

Table 4.3 Single-error mutation operators and their expected effects on rows (R), columns (C), values (V), and execution metrics (EXP, EXR). Symbols: ↑ increase, ↓ decrease, = unchanged, × context-dependent.

Operator	Description	(R, C, V)	(EXP, EXR)
`projection_drop`	Remove one column from the projection list (if > 1 remain).	(=, ↓, =)	(=, ↓)
`add_star_wildcard`	Append `*` or `alias.*`/`table.*` if no star exists.	(=, ↑, =)	(PE ↓; IE =,=)
`distinct_toggle`	Remove the `DISTINCT` keyword.	(×, =, =)	(↓, ×)
`where_predicate_delete`	Remove one predicate from a compound condition.	(↑, =, =)	(↓, ↑)
`where_condition_flip`	Flip comparison direction (=↔≠, >↔<, ≥↔≤).	(×, =, =)	(×, ×)
`where_strengthen`	Make boundary stricter (`<`→`<=`, `>`→`>=`).	(↓, =, =)	(=, ↓)
`where_weaken`	Make boundary looser (`<=`→`<`, `>=`→`>`).	(↑, =, =)	(↓, ↑)
`where_remove`	Remove the entire `WHERE` clause.	(↑, =, =)	(↓, ↑)
`having_condition_flip`	Flip aggregate comparison (=↔≠, >↔<, ≥↔≤).	(×, =, =)	(×, ×)
`having_remove`	Remove the `HAVING` clause.	(↑, =, =)	(↓, ↑)
`join_break`	Remove the `ON` condition (cartesian product).	(↑, =, =)	(↓, ×)
`join_type_to_left`	Convert non-`LEFT` joins to `LEFT JOIN`.	(↑, =, =)	(↓, ↑)
`limit_increase`	Double the `LIMIT` and add a random `OFFSET`.	(↑, =, =)	(↓, ↑)
`limit_decrease`	Halve the `LIMIT` (min 1).	(↓, =, =)	(=, ↓)
`aggregation_swap`	Swap aggregates (`SUM`↔`AVG`, `MIN`↔`MAX`, `COUNT`→`SUM`).	(=, =, ×)	(↓, ↓)

Table 4.3 groups operators by how increasing their "strength" affects the metrics. Row *increase* (*e.g.*, `limit_increase`, `where_remove`) lowers precision while recall rises toward 1, a monotonic pattern. Row *decrease* (*e.g.*, `limit_decrease`, `where_strengthen`) has the inverse effect: recall drops while precision stays near 1. Schema edits isolate cover-

age effects: `projection_drop` cuts recall; `add_star_wildcard` reduces precision only under PE. Value errors like `aggregation_swap` degrade both. Boundary or dedup changes (`where_condition_flip`, `distinct_toggle`) behave non-monotonically, showing over- or under-selection that EX cannot capture.

Table 4.4 Impact of single-injected errors on evaluation metrics (%).

	Evaluation Metrics (%)											
	EC-EC-IE			**EC-PC-IE**			**SC-EC-IE**			**SC-PC-IE**		
Injected Error	ΔEX	ΔEXP	ΔEXR	ΔEXP	ΔEXR	ΔF1	ΔEXP	ΔEXR	ΔF1	ΔEXP	ΔEXR	ΔF1
`projection_drop`	-100	0	-43	0	-43	-28	0	-43	-28	0	-43	-28
`add_star_wildcard`	-100	-93	-10	-92	-4	-87	-82	0	-71	-82	0	-71
`distinct_toggle`	-100	-57	-57	-57	-57	-57	-57	-57	-57	-57	-57	-57
`where_predicate_delete`	-100	-85	-60	-85	-60	-81	-85	-64	-82	-85	-64	-82
`where_condition_flip`	-100	-96	-61	-97	-69	-96	-96	-61	-94	-97	-69	-96
`where_remove`	-100	-96	-54	-96	-56	-95	-96	-54	-95	-96	-56	-95
`where_strengthen`	-100	-65	-60	-67	-62	-66	-65	-60	-63	-67	-62	-66
`where_weaken`	-100	-65	-75	-65	-75	-73	-65	-75	-73	-65	-75	-73
`having_condition_flip`	-100	-64	-100	-67	-100	-100	-64	-100	-100	-67	-100	-100
`having_remove`	-100	-61	0	-62	0	-53	-61	0	-51	-62	0	-53
`join_break`	-100	-98	-42	-95	-47	-94	-98	-46	-98	-95	-54	-94
`join_type_to_left`	-100	-53	-50	-55	-52	-56	-53	-50	-54	-55	-52	-56
`limit_increase`	-100	-76	0	-76	0	-62	-76	0	-62	-76	0	-62
`limit_decrease`	-100	-4	-61	-4	-61	-45	0	-60	-43	0	-60	-43
`aggregation_swap`	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100

We test each mutant to see if metrics degrade proportionally to error severity. Strong errors (*e.g.*, dropping `WHERE` clauses) lowers both precision and recall; mild ones (*e.g.*, `limit_increase`) affect only one. We report ΔEX, ΔEXP, ΔEXR, and ΔF1, defined as $\Delta = \text{metric} - 1$, where more negative values indicate stronger penalties and higher sensitivity. Table 4.4 summarizes our results. It confirms the core limitation of binary **EX**: it is −100% for every single injected error, so it cannot distinguish easy mistakes from severe ones. Our relaxed metrics spread out the outcomes and show *how* a query is wrong.

Row count operators behave as designed. With `limit_increase` (pure over-selection), precision drops sharply while recall stays at the ceiling (EXP↓, EXR= 1), the classic false-positive pattern. With `limit_decrease` (under-selection), precision remains near intact while recall falls (EXP≈, EXR↓), the false-negative pattern. This clean split between EXP and EXR is exactly the diagnostic signal we need.

Schema operators separate coverage from penalty. `projection_drop` removes gold columns, so recall declines while precision is unchanged (EXR↓, EXP=). `add_star_wildcard` adds extra predicted columns: under exact-column accounting (EC) this lowers precision (EXP↓) with little change to recall; semantic-column matching (SC) softens the precision hit, showing

robustness to schema noise. `distinct_toggle` reduces both precision and recall amounts, consistent with duplicate handling shifting both predicted and covered cells.

Filter and grouping operators form a severity ladder. Mild edits like `where_predicate_delete` already depress both axes; stronger edits like `where_remove` or boundary `where_condition_flip` hurt more, with flips among the worst. For aggregates, `having_remove` behaves like row inflation: precision drops while recall stays at the ceiling (EXR= 1), whereas `having_condition_flip` collapses recall entirely (EXR= 0) and also lowers precision.

Join operators have distinct fingerprints. `join_break` (cartesian product) devastates precision and also reduces recall; partial-cell matching (PC) gives limited relief because many near-miss rows share some values. `join_type_to_left` yields a more balanced, mid-range drop on both axes, as expected for a milder change in join semantics.

SC (semantic columns) usually matches or improves upon **EC**, particularly for added noise like `add_star_wildcard`. **PC** (partial-cell rows) helps only when there are genuine near-misses (*e.g.*, `join_break`); otherwise it behaves like EC.

Takeaway. EXP and EXR turn a binary verdict into actionable diagnostics: they separate over- from under-prediction, distinguish column and row prediction mistakes and are further implemented to handle challenges of column aliasing under semantic alignment.

Table 4.5 System-level comparison on the BIRD dev set.

		Evaluation Metrics (%)														
		EC-EC-IE			**EC-PC-IE**			**SC-EC-IE**			**SC-PC-IE**			**NC-PC**		
System	EX	EXP	EXR	F1	EXP	EXR	F1	EXP	EXR	F1	EXP	EXR	F1	EXP	EXR	F1
BIRD Shared Failures Subset																
CHESS	0.00	28.76	19.66	18.44	29.06	19.85	18.82	47.80	16.62	15.14	51.44	19.11	17.82	N/A	N/A	N/A
DIN-SQL	0.00	29.01	21.12	18.42	29.09	21.25	18.51	63.04	19.54	17.18	66.26	22.05	19.71	N/A	N/A	N/A
MAC-SQL	0.00	31.09	16.07	14.63	31.34	16.30	14.84	54.47	13.18	11.91	56.97	15.10	13.82	N/A	N/A	N/A

System-Level Comparison

Research Question – Do granular metrics reveal finer distinctions in how systems fail, beyond the binary EX?

We conducted a comparative analysis across three Text-to-SQL systems: CHESS, DIN-SQL, and MAC-SQL on the BIRD dev set. Among the $1,534$ examples, we identified 410 queries for which all three systems produced $EX = 0$, representing $\sim$26.7% of the benchmark. These

shared failures pose a particular challenge: traditional binary metrics offer no information for ranking systems or analyzing their behaviour, despite potential partial correctness in the predictions. Table 4.5 summarizes the results on our metrics for the three systems.

The ranking of systems (CHESS > DIN-SQL > MAC-SQL) remains the same across EX and our relaxed metrics. However, EXP and EXR expose meaningful differences inside the shared-failure slice: e.g., CHESS recovers nearly a quarter of ground-truth cells while MAC-SQL captures closer to 15%. Moreover, precision/recall asymmetries reveal why systems fail, CHESS maintains higher recall, while DIN-SQL balances precision and recall more evenly. Thus, while the leaderboard order is stable, our metrics enrich it with diagnostic resolution that EX alone cannot provide. Fig. 4.3 showcases the EXP and EXR distribution across the failed queries how close systems can be to the right answer.

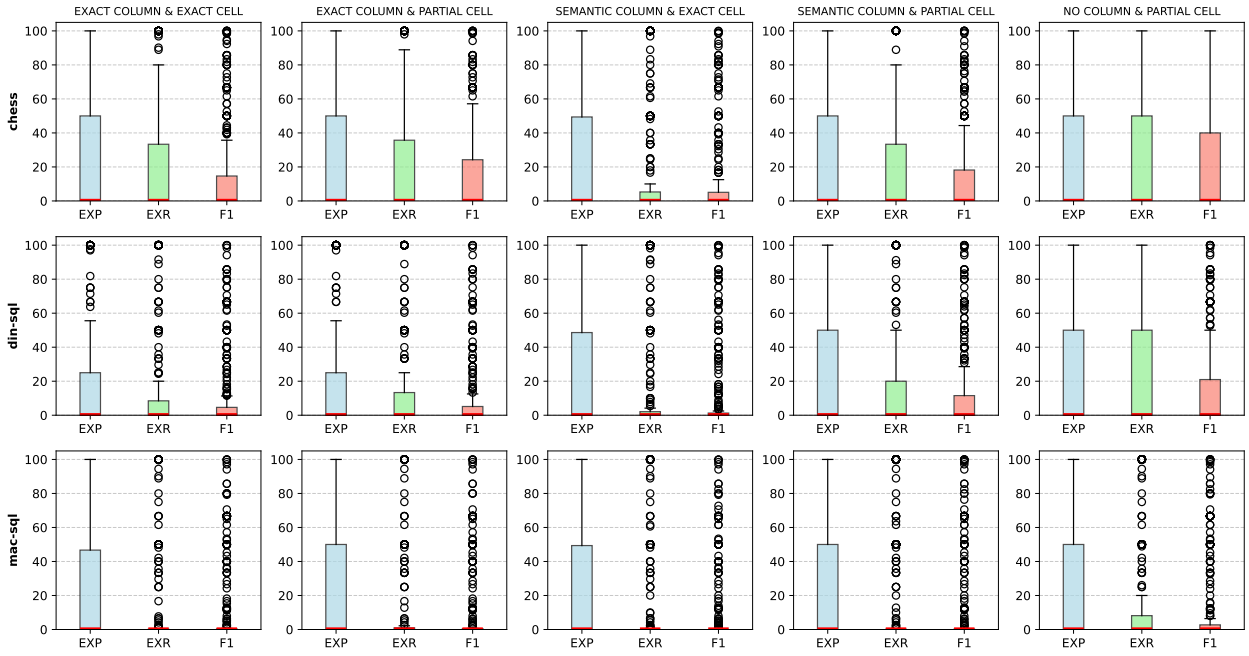


Figure 4.3 System-level comparison on the BIRD dev set's shared failures subset on combinations of fine-grained metrics.

Qualitative Example

Consider the following NL query: 'List the names of schools with more than 30 difference in enrollments between K–12 and ages 5–17. Please also give the full street address of the schools.' with the SC-EC-PE Metrics as follows:

SC-EC-PE Metrics (DIN-SQL vs MAC-SQL).
EX: 0.0 vs 0.0
EXP: 0.393 vs 0.5 (*gap:* 0.11)
EXR: 0.982 vs 1.0 (*gap:* 0.02)
F1: 0.561 vs 0.667 (*gap:* 0.11)

This highlights how big the variations can be among the failed queries. Next, we report the predicted and golden SQL queries and provide an explanation.

Gold SQL (Ground-truth):

```
SELECT T1.School, T1.Street
  FROM schools AS T1
  JOIN frpm AS T2 ON T1.CDSCode = T2.CDSCode
 WHERE T2.`Enrollment (K-12)` - T2.`Enrollment (Ages 5-17)` > 30;
```

DIN-SQL SQL (Predicted):

```
SELECT frpm.`School Name`, schools.Street,
  schools.City, schools.Zip, schools.State
  FROM frpm
  JOIN schools ON frpm.CDSCode = schools.CDSCode
 WHERE (frpm.`Enrollment (K-12)` - frpm.`Enrollment (Ages 5-17)`) > 30;
```

MAC-SQL SQL (Predicted):

```
SELECT T2.School, T2.Street, T2.City, T2.Zip
  FROM frpm AS T1
  JOIN schools AS T2 ON T1.CDSCode = T2.CDSCode
 WHERE T1.`Enrollment (K-12)` - T1.`Enrollment (Ages 5-17)` > 30;
```

Explanation:

- Both systems predict the correct logical condition for filtering schools with large enrollment differences, but they differ in their projections.
- One would expect both to obtain an EXR of 1.0, however DIN-SQL does not. It projects frpm.'School Name' instead of schools.School. frpm contains some corrupted values leading to not having a perfect match.
- DIN-SQL projects 5 attributes while MAC-SQL projects 4, *i.e.*, 3 and 2 extra attributes, respectively. The penalty is reflected in their EXP and therefore F1.

Takeaway: This example highlights a case with a large **EXR** (*i.e.*, retrieving most of the expected rows) and a small **EXP** (∼half for adding unnecessary cells). In some cases the metrics can also showcase the opposite. Binary EX marks both systems as equally wrong even though they are very close to the correct answer. Fine-grained metrics shows that both systems capture retrieve almow5cqll expected rows (> 0.98 EXR) and get penalized just for irrelevant columns.

4.3 Textual Query Augmentation

4.3.1 Overview

Schema naming exerts a strong influence on Text-to-SQL performance. Yet, while enterprise databases frequently employ domain-specific terminology and abbreviations, public benchmarks predominantly feature more natural names. To alleviate this issue, *Textual Query Augmentation (TQA)* targets the *robustness* of Text-to-SQL systems by systematically altering the natural language (NL) query and schema identifiers in evaluation sets while preserving semantics and executability.

TQA serves as a controlled mechanism to test the retrieval stage, known to be sensitive to naming conventions and lexical overlap. Prior studies establish an inverse relationship between naming *naturalness* and model accuracy, categorizing identifiers into three levels: regular (N1), low (N2), and least (N3) [87]. TQA challenges the retrieval stage by transforming benchmark data into harder, less-natural variants that maintain the same semantics.

Figure 4.4 illustrates the distribution of naming naturalness across the BIRD dev set, where roughly 80% of schema identifiers exhibit natural (N1) forms. This imbalance motivates the need for augmentation techniques that can generate realistic, enterprise-like naming conditions to challenge retrieval stages and human-in-the-loop modules.

4.3.2 Design Principles

Part of TQA's design principle is ensuring that performance changes observed arise solely from textual robustness and not from semantic variation. As such, TQA transforms schema elements and their references within queries and evidence while keeping the same executable SQL and only replacing the element name in the SQL if necessary. This ensures that only lexical surfaces, not query logic or intent, are modified. Furthermore, TQA's perturbations are targeted: Values, constants, and operators are fixed, guaranteeing minimal changes.

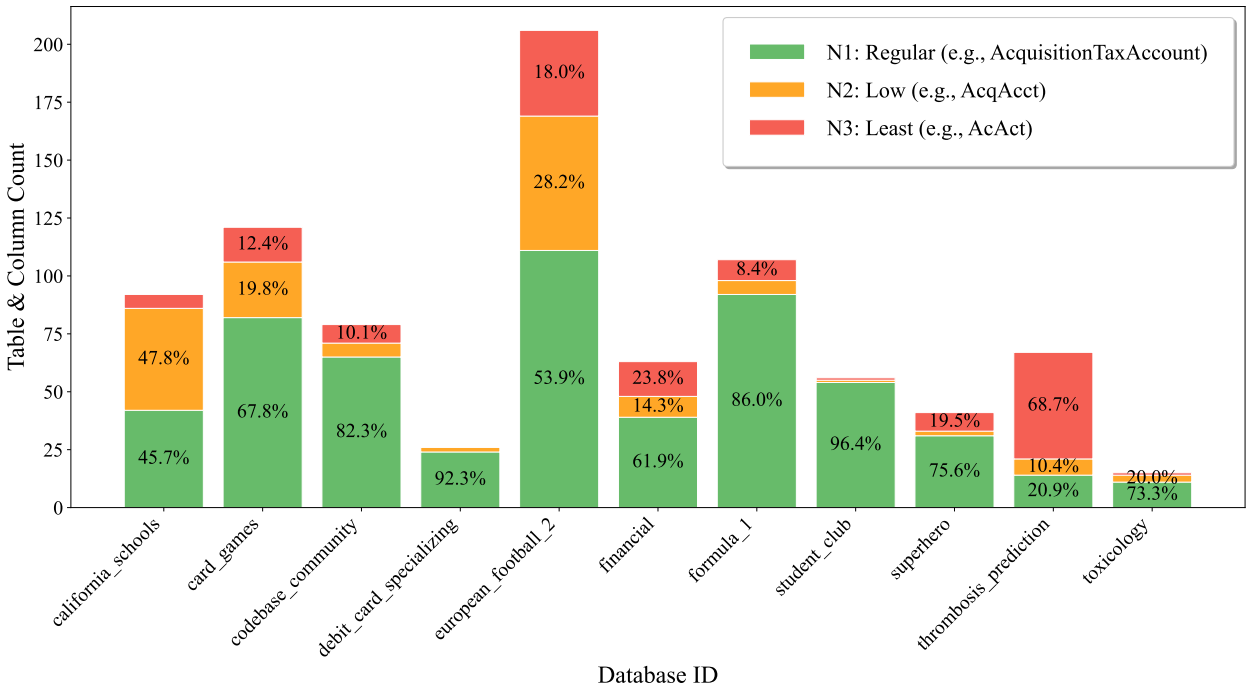


Figure 4.4 Distribution of naming naturalness across BIRD dev databases.

4.3.3 Approach

Our approach to TQA is divided into three parts:

Input

- **Seed query.** NL–SQL pair with accompanying evidence and relevant schema.
- **Naturalness classifier.** Fine-tuned over `google/canine-s` and classifies naturalness as regular (N1), low (N2), or least (N3) [87].
- **User preferences (optional).** De-naturalization aggressiveness (N1→N3 vs. N1→N2), and target references location ({NL, schema, both}).

Output

A set of minimally changed, *less-natural* references that are semantically faithful and executable leading to possibly: (i) modified schemas (via `ALTER`), (ii) updated gold SQL (identifiers rewritten), and (iii) updated NL/evidence (only schema mentions rewritten). Each item passes execution equivalence checks (EX= 1 relative to the original query).

Technique

TQA runs as follows:

- **Classify Naturalness.** Extract table and column identifiers from database schema and label them with SNAILS' classifier: N1, N2, or N3 with a confidence score.
- **Decrease Naturalness.** Given labels, transform N1 and N2 identifiers toward N3 using a few-shot LLM prompt, with temperature 0, and a fixed seed, yielding a renaming, *e.g.*, *WaterTemperature*→*WtTp*. Prompts are avaible at Appendix D and D.
- **Alter Schema.** Clone each database and apply table and column renames via SQL `ALTER`. Validate the rename was successful (*e.g.*, using `PRAGMA table_info()`).
- **Alter SQL Query.** Update golden SQL query by replacing old identifiers with the new ones using regex patterns while respecting quotes and avoiding SQL keywords/functions. Validate by executing both original and transformed queries on their respective schemas and requiring EX= 1, *i.e.*, both lead to the same output relation.
- **Alter NL Query.** Rewrite the NL query and evidence by substituting only schema mentions with least-natural forms using a fixed few-shot prompt (Appendix D). We avoid changes to values (numbers, dates, entities).

4.3.4 Experimental Analysis

To evaluate TQA, we follow the experimental setup in Section 4.1. We evaluate four augmentation settings which rename either the NL side (NL query and evidence), the SQL side (Schema definition and SQL query), or both:

- **O/O**: Original NL + Original SQL
- **L/O**: Less-natural NL + Original SQL
- **O/L**: Original NL + Less-natural SQL
- **L/L**: Less-natural NL + Less-natural SQL

Applying TQA to BIRD mini-dev (500 high-quality NL–SQL pairs from 11 dev databases), 73% of items remain executable and semantically preserved after the validation step in 'Alter SQL Query' leading to 366 augmented queries.

System-level analysis We evaluate Chess, Mac-SQL, and Din-SQL using EX. For each system, we only consider queries for which they had a successful generation, *i.e.*, under O/O

Table 4.6 TQA results on BIRD mini-dev showing %ΔEX vs. O/O.

	ΔEX (%)		
NL/Schema	**CHESS**	**MAC-SQL**	**DIN-SQL**
O/O	0.0	0.0	0.0
L/O	**-9.2**	**-14.9**	-6.8
O/L	-6.7	-9.5	**-17.1**
L/L	-7.9	-12.9	-16.6

has EX= 1, then measure changes under (L/O), (O/L), and (L/L). This leads to initial query counts per system as follows: 240 for Chess, 201 for Mac-SQL, and 205 for Din-SQL. Table 4.6 reports ΔEX (%) relative to O/O (negative is a drop). All systems degrade consistently. Schema and NL de-naturalization both reduce EX; DIN-SQL shows the largest degradation overall, followed by MAC-SQL and CHESS. Thus, TQA can increase benchmark difficulty while preserving semantics. We next analyze the impact of the three augmentation settings:

- **L/O.** De-naturalizing only the NL side yields drops of −9.2%, −14.9%, and −6.8% for CHESS, MAC-SQL, and DIN-SQL, respectively. The larger losses for CHESS and MAC-SQL indicates reliance on surface lexical overlap. This augmentation requires no database changes.
- **O/L.** Making only the SQL side less natural *in isolation* causes drops of −6.7%, −9.5%, and −17.1% for CHESS, MAC-SQL, and DIN-SQL, respectively. DIN-SQL is the most affected, and this is consistent with prior work [87]: reducing naming naturalness increases retrieval difficulty.
- **L/L.** Using less-natural NL and SQL leads to the largest overall degradation, but only slightly worse than O/L: −7.9%, −12.9%, and −16.6% for CHESS, MAC-SQL, and DIN-SQL, respectively. When NL and schema share the same abbreviated names, retrieval can re-align, limiting effects, which explains the boost for DIN-SQL.

Takeaway. TQA can make queries harder while preserving semantics and executability. We find that regardless of the augmentation setting: O/L, L/O, and L/L, all lead to degradation. For production settings where schema cannot change, L/O provides a natural setting to challenge systems retrieval. This further highlights the importance of query expansion and data expansion techniques in Text-to-SQL evaluation.

Stage-level analysis To conduct stage-level analysis, we adopt the schema retrieval formulation of prior work [66] and consider three retrieval approaches: **Full-Schema** (no filtering; retrieves over the full schema), **TCSL** (Table-then-Column), and **SCSL** (Single-Column). We evaluate them across O/O, O/L, L/O, and L/L on the 366-augmented queries We report **FPR** (False Positive Rate; lower is better) and **SLR** (Schema Linking Recall; higher is better), where SLR measures whether all required columns are retrieved for each query. Results are summarized in Table 4.7.

Table 4.7 Schema retrieval performance across naturalness settings. FPR (%) = False Positive Rate (lower is better), SLR (%) = Schema Linking Recall (higher is better).

	Full-Schema		**SCSL**		**TCSL**	
NL/Schema	**FPR**	**SLR**	**FPR**	**SLR**	**FPR**	**SLR**
O/O	89.2	99.7	24.0	15.3	10.4	28.4
L/O	86.1	30.3	25.0	9.8	11.8	23.5
O/L	89.2	96.7	27.8	7.7	12.4	23.2
L/L	86.5	25.7	28.4	11.7	13.5	19.4

- **L/O.** Relative to O/O, SLR falls substantially for Full-Schema (−69.4%) and decreases for SCSL/TCSL (−5.5/−4.9%). FPR rises slightly for SCSL/TCSL, indicating noisier contexts but both trends reduce EX.
- **O/L.** SLR drops (−3.0, −7.6, −5.2% for Full-Schema/SCSL/TCSL) and FPR increases (SCSL/TCSL: +3.8, +2.0%), consistent with harder schema linking.
- **L/L).** From L/O to L/L, additional changes are small: SLR (−4.6, +1.9, −4.1%) with slight FPR increases. This mirrors Table 4.6: when NL and schema share abbreviated names, retrieval can partially re-align.

Qualitative Example To illustrate augmentation and its downstream effects, we show an example from the `toxicology` database (ID 245) in the BIRD dev set. The original and less-natural versions differ only in schema naming and its propagation to NL and SQL, while preserving semantics and execution results. We next showcase the original and less-natural: NL query, evidence, and the required changes to the relevant schema. We then analyze the behaviour of schema retrieval on this query.

Original NL: *What is the average number of bonds the atoms with the element iodine have?*
Less-natural NL: *What is the average number of bnds the atms with the Elmt iodine have?*

Original Evidence:

atoms with the element iodine refers to element = 'i'; average = DIVIDE(COUNT(bond_id), COUNT(atom_id)) where element = 'i'

Less-natural Evidence:

atms with the Elmt iodine refers to Elmt = 'i'; average = DIVIDE(COUNT(b_id), COUNT(atmId)) where Elmt = 'i'

Original Relevant Schema:

`atom.atom_id`, `atom.element`, `connected.atom_ id`, `connected.bond_id`

Less-natural Relevant Schema:

`atm.atmId`, `atm.Elmt`, `conn.atmId`, `conn.b_id`

Schema Retrieval. Under **O/L** and **L/O** settings, systems frequently fail to retrieve the correct columns, *e.g.*, missing `atom_id` or `connected.atom_id`, leading to downstream execution errors. In contrast, when both NL and schema share the same abbreviated identifiers (**L/L**), retrievers re-align and correctly retrieve all relevant columns across all linking strategies (Full-Schema, TCSL, SCSL). This observation aligns with the aggregate analysis in Table 4.7, confirming that matching abbreviations across modalities partially mitigates the negative impact of reduced naturalness.

Takeaway. TQA can stress test query augmentation techniques or system query reformulation and understanding. It can help identify failures such as misalignment between a schema and an NL query.

CHAPTER 5 CONCLUSION

This thesis investigated how to make LLM-based Text-to-SQL systems more practical for enterprise use by addressing two challenges: the cost of LLM-powered pipelines and the difficulty of evaluating systems in a way that is both diagnostic and representative of enterprise conditions. Building on a multi-stage pipeline approach, we showed that (i) generation can be made cost-aware through routing, i.e., adaptive model selection, and (ii) evaluation can move beyond a single binary metric over clean schemas by combining fine-grained execution metrics with controlled schema and query mutations.

Our first contribution targets efficiency. We introduced an LLM routing framework that selects, for each query, the least expensive model expected to generate accurate executable SQL. By designing lightweight score-based and classification-based model routers, we ensured that routing overhead remains negligible relative to the SQL generation stage it optimizes. Empirically, the router maintained accuracy close to that of the strongest model while reducing average cost, demonstrating that cost-aware Text-to-SQL can preserve high performance without uniformly using on all queries a state-of-the-art LLM.

Our second contribution targets effective evaluation. We proposed fine-grained execution metrics: Execution Precision (EXP) and Execution Recall (EXR), summarized by F1 score, which exposes partial correctness and error patterns that binary Execution Accuracy (EX) collapses. Complementing these metrics, we introduced Textual Query Augmentation (TQA), which perturbs natural language queries and schema identifiers while preserving semantics, thereby emulating enterprise-style naming and stress-testing pipeline stages that depend on schema linking or retrieval and ambiguity resolution (including human-in-the-loop settings). Together, these metrics and mutations enable more informative comparisons across systems, and experiments highlight systematic brittleness as naming naturalness is reduced.

5.1 Limitations

LLM Routing. Our routing experiments are conducted in a limited setting: we use a single dataset and a single Text-to-SQL pipeline (VortoSQL), which threatens validity and may limit generalization to other architectures, datasets, and schema characteristics. We also route over only three LLMs and rely on fixed, normalized dollar costs, abstracting away deployment-specific effects such as infrastructure overheads, variable latency, and model-serving constraints; consequently, the observed cost-accuracy trade-offs may differ under

alternative pricing schemes or serving environments. Finally, both routing methods assume access to a historical log H with ground-truth labels, which can be costly to collect and may bias the router toward query patterns overrepresented in past traffic. In addition, realistic class imbalance in routing labels is not explicitly handled in our study and may affect router training and calibration.

Evaluation. Our fine-grained metrics rely on semantic matching when aligning attribute values, which cannot guarantee correctness in all cases and may introduce false matches or misses. Moreover, the use of a specific embedding model introduces an additional source of nondeterminism; fair cross-paper comparisons on public benchmarks would therefore require agreement on a standard embedding model, preprocessing, and thresholding configuration. In addition, TQA relies on LLMs for natural-language generation and augmentation, which introduces prompt sensitivity, the possibility of hallucinated or stylistically inconsistent rewrites, and reproducibility challenges, even though we aim to keep outputs deterministic.

5.2 Future Work

LLM Routing. An interesting research direction is to broaden routing beyond the SQL generation stage to additional pipeline stages, e.g., schema linking, candidate selection, and correction, where systems already invoke multiple state-of-the-art LLM calls per query. Finally, a larger empirical study across more datasets and workloads, including online measurements, will investigate whether routing can deliver simultaneous gains in execution accuracy and cost under realistic serving constraints.

Evaluation. Future work can extend beyond TQA and naming perturbations by incorporating controlled, semantics-preserving structural changes to queries, for example, introducing additional join and aggregation patterns. A further direction is to evaluate empirically whether the fine-grained metrics, EXP and EXR, are useful not only for diagnosis, but also as training rewards or as model-selection signals.

REFERENCES

[1] K. Maamari and A. Mhedhbi, "End-to-end text-to-sql generation within an analytics insight engine," *CoRR*, vol. abs/2406.12104, 2024.

[2] P. Ma and S. Wang, "Mt-teql: evaluating and augmenting neural nlidb on real-world linguistic and schema variations," *Proc. VLDB Endow.*, vol. 15, no. 3, p. 569–582, Nov. 2021. [Online]. Available: https://doi.org/10.14778/3494124.3494139

[3] G. Katsogiannis-Meimarakis and G. Koutrika, "A survey on deep learning approaches for text-to-sql," *The VLDB Journal*, vol. 32, no. 4, p. 905–936, Jan. 2023. [Online]. Available: https://doi.org/10.1007/s00778-022-00776-8

[4] X. Deng, A. H. Awadallah, C. Meek, O. Polozov, H. Sun, and M. Richardson, "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*. Association for Computational Linguistics, 2021. [Online]. Available: http://dx.doi.org/10.18653/v1/2021.naacl-main.105

[5] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "SQuAD: 100,000+ questions for machine comprehension of text," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, J. Su, K. Duh, and X. Carreras, Eds. Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 2383–2392. [Online]. Available: https://aclanthology.org/D16-1264/

[6] P. Rajpurkar, R. Jia, and P. Liang, "Know what you don't know: Unanswerable questions for SQuAD," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, I. Gurevych and Y. Miyao, Eds. Melbourne, Australia: Association for Computational Linguistics, Jul. 2018, pp. 784–789. [Online]. Available: https://aclanthology.org/P18-2124/

[7] J. Li, B. Hui, G. Qu, J. Yang, B. Li, B. Li, B. Wang, B. Qin, R. Geng, N. Huo, X. Zhou, C. Ma, G. Li, K. C. Chang, F. Huang, R. Cheng, and Y. Li, "Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, ser. NIPS '23. Red Hook, NY, USA: Curran Associates Inc., 2023.

[8] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. Radev, "Spider: A large-scale human-labeled dataset for

complex and cross-domain semantic parsing and text-to-SQL task," 2018. [Online]. Available: https://aclanthology.org/D18-1425/

[9] H. Yang, Y. Zhang, J. Xu, H. Lu, P.-A. Heng, and W. Lam, "Unveiling the generalization power of fine-tuned large language models," in *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, K. Duh, H. Gomez, and S. Bethard, Eds. Mexico City, Mexico: Association for Computational Linguistics, Jun. 2024, pp. 884–899. [Online]. Available: https://aclanthology.org/2024.naacl-long.51/

[10] A. Floratou, F. Psallidas, F. Zhao, S. Deep, G. Hagleither, W. Tan, J. Cahoon, R. Alotaibi, J. Henkel, A. Singla, A. V. Grootel, B. Chow, K. Deng, K. Lin, M. Campos, K. V. Emani, V. Pandit, V. Shnayder, W. Wang, and C. Curino, "Nl2sql is a solved problem... not!" in *Conference on Innovative Data Systems Research*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:266729311

[11] W. Fan, Y. Ding, L. Ning, S. Wang, H. Li, D. Yin, T.-S. Chua, and Q. Li, "A survey on rag meeting llms: Towards retrieval-augmented large language models," in *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, ser. KDD '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 6491–6501. [Online]. Available: https://doi.org/10.1145/3637528.3671470

[12] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," 2023. [Online]. Available: https://arxiv.org/abs/2210.03629

[13] F. Lei, J. Chen, Y. Ye, R. Cao, D. Shin, H. Su, Z. Suo, H. Gao, W. Hu, P. Yin, V. Zhong, C. Xiong, R. Sun, Q. Liu, S. Wang, and T. Yu, "Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows," 2025. [Online]. Available: https://arxiv.org/abs/2411.07763

[14] M. Deng, A. Ramachandran, C. Xu, L. Hu, Z. Yao, A. Datta, and H. Zhang, "Reforce: A text-to-sql agent with self-refinement, consensus enforcement, and column exploration," 2025. [Online]. Available: https://arxiv.org/abs/2502.00675

[15] F. Li and H. V. Jagadish, "Constructing an interactive natural language interface for relational databases," *Proc. VLDB Endow.*, vol. 8, no. 1, p. 73–84, Sep. 2014. [Online]. Available: https://doi.org/10.14778/2735461.2735468

[16] T. Mahmud, K. M. Azharul Hasan, M. Ahmed, and T. H. C. Chak, "A rule based approach for nlp based query processing," in *2015 2nd International Conference on Electrical Information and Communication Technologies (EICT)*, 2015, pp. 78–82.

[17] T. Yu, C.-S. Wu, X. V. Lin, B. Wang, Y. C. Tan, X. Yang, D. Radev, R. Socher, and C. Xiong, "Grappa: Grammar-augmented pre-training for table semantic parsing," *ICLR*, 2021.

[18] J. M. Zelle and R. J. Mooney, "Learning to parse database queries using inductive logic programming," *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2*, p. 1050–1055, 1996.

[19] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[20] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[21] D. Choi, M. C. Shin, E. Kim, and D. R. Shin, "RYANSQL: Recursively applying sketch-based slot fillings for complex text-to-SQL in cross-domain databases," *Computational Linguistics*, vol. 47, no. 2, pp. 309–332, Jun. 2021. [Online]. Available: https://aclanthology.org/2021.cl-2.12/

[22] J. Guo, Z. Zhan, Y. Gao, Y. Xiao, J.-G. Lou, T. Liu, and D. Zhang, "Towards complex text-to-SQL in cross-domain database with intermediate representation," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, A. Korhonen, D. Traum, and L. Màrquez, Eds. Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 4524–4535. [Online]. Available: https://aclanthology.org/P19-1444/

[23] B. Hui, X. Shi, R. Geng, B. Li, Y. Li, J. Sun, and X. Zhu, "Improving text-to-sql with schema dependency learning," 2021. [Online]. Available: https://arxiv.org/abs/2103.04399

[24] X. Xu, C. Liu, and D. Song, "Sqlnet: Generating structured queries from natural language without reinforcement learning," *ICLR*, 2017. [Online]. Available: https://arxiv.org/abs/1711.04436

[25] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," 2019.

[26] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," 2019. [Online]. Available: https://arxiv.org/abs/1907.11692

[27] V. Zhong, C. Xiong, and R. Socher, "Seq2sql: Generating structured queries from natural language using reinforcement learning," 2017. [Online]. Available: https://arxiv.org/abs/1709.00103

[28] L. Dou, Y. Gao, X. Liu, M. Pan, D. Wang, W. Che, D. Zhan, M.-Y. Kan, and J.-G. Lou, "Towards knowledge-intensive text-to-sql semantic parsing with formulaic knowledge," 2023. [Online]. Available: https://arxiv.org/abs/2301.01067

[29] H. Li, J. Zhang, C. Li, and H. Chen, "Resdsql: decoupling schema linking and skeleton parsing for text-to-sql," in *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence*, ser. AAAI'23/IAAI'23/EAAI'23. AAAI Press, 2023. [Online]. Available: https://doi.org/10.1609/aaai.v37i11.26535

[30] A. Radford and K. Narasimhan, "Improving language understanding by generative pre-training," 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:49313245

[31] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS '20. Red Hook, NY, USA: Curran Associates Inc., 2020.

[32] OpenAI, "Gpt-4 technical report," 2024. [Online]. Available: https://arxiv.org/abs/2303.08774

[33] N. Rajkumar, R. Li, and D. Bahdanau, "Evaluating the text-to-sql capabilities of large language models," 2022. [Online]. Available: https://arxiv.org/abs/2204.00498

[34] D. Gao, H. Wang, Y. Li, X. Sun, Y. Qian, B. Ding, and J. Zhou, "Text-to-sql empowered by large language models: A benchmark evaluation," *Proc. VLDB Endow.*, vol. 17, no. 5, p. 1132–1145, Jan. 2024. [Online]. Available: https://doi.org/10.14778/3641204.3641221

[35] S. Chang and E. Fosler-Lussier, "How to prompt llms for text-to-sql: A study in zero-shot, single-domain, and cross-domain settings," 2023. [Online]. Available: https://arxiv.org/abs/2305.11853

[36] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS '22. Red Hook, NY, USA: Curran Associates Inc., 2022.

[37] H. Li, J. Zhang, H. Liu, J. Fan, X. Zhang, J. Zhu, R. Wei, H. Pan, C. Li, and H. Chen, "Codes: Towards building open-source language models for text-to-sql," *Proc. ACM Manag. Data*, vol. 2, no. 3, May 2024. [Online]. Available: https://doi.org/10.1145/3654930

[38] Z. Hong, Z. Yuan, H. Chen, Q. Zhang, F. Huang, and X. Huang, "Knowledge-to-SQL: Enhancing SQL generation with data expert LLM," in *Findings of the Association for Computational Linguistics: ACL 2024*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 10 997–11 008. [Online]. Available: https://aclanthology.org/2024.findings-acl.653/

[39] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," 2022. [Online]. Available: https://arxiv.org/abs/2109.01652

[40] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: deliberate problem solving with large language models," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, ser. NIPS '23. Red Hook, NY, USA: Curran Associates Inc., 2023.

[41] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," 2021. [Online]. Available: https://arxiv.org/abs/2106.09685

[42] C.-H. Lee, O. Polozov, and M. Richardson, "KaggleDBQA: Realistic evaluation of text-to-SQL parsers," 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), C. Zong, F. Xia, W. Li, and R. Navigli, Eds. Online: Association for Computational Linguistics, Aug. 2021, pp. 2261–2273. [Online]. Available: https://aclanthology.org/2021.acl-long.176/

[43] L. Wang, A. Zhang, K. Wu, K. Sun, Z. Li, H. Wu, M. Zhang, and H. Wang, "DuSQL: A large-scale and pragmatic Chinese text-to-SQL dataset," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, B. Webber, T. Cohn, Y. He, and Y. Liu, Eds. Online: Association for Computational Linguistics, Nov. 2020, pp. 6923–6935. [Online]. Available: https://aclanthology.org/2020.emnlp-main.562/

[44] Y. Gan, X. Chen, J. Xie, M. Purver, J. R. Woodward, J. Drake, and Q. Zhang, "Natural sql: Making sql easier to infer from natural language specifications," *EMNLP*, 2021.

[45] X. Pi, B. Wang, Y. Gao, J. Guo, Z. Li, and J.-G. Lou, "Towards robustness of text-to-SQL models against natural and realistic adversarial table perturbation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 2007–2022. [Online]. Available: https://aclanthology.org/2022.acl-long.142/

[46] Y. Gan, X. Chen, Q. Huang, and M. Purver, "Measuring and improving compositional generalization in text-to-SQL via component alignment," in *Findings of the Association for Computational Linguistics: NAACL 2022*, M. Carpuat, M.-C. de Marneffe, and I. V. Meza Ruiz, Eds. Seattle, United States: Association for Computational Linguistics, Jul. 2022, pp. 831–843. [Online]. Available: https://aclanthology.org/2022.findings-naacl.62/

[47] T. Yu, R. Zhang, M. Yasunaga, Y. C. Tan, X. V. Lin, S. Li, H. Er, I. Li, B. Pang, T. Chen, E. Ji, S. Dixit, D. Proctor, S. Shim, J. Kraft, V. Zhang, C. Xiong, R. Socher, and D. Radev, "SParC: Cross-domain semantic parsing in context," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, A. Korhonen, D. Traum, and L. Màrquez, Eds. Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 4511–4523. [Online]. Available: https://aclanthology.org/P19-1443/

[48] T. Yu, R. Zhang, H. Y. Er, S. Li, E. Xue, B. Pang, X. V. Lin, Y. C. Tan, T. Shi, Z. Li, Y. Jiang, M. Yasunaga, S. Shim, T. Chen, A. Fabbri, Z. Li, L. Chen, Y. Zhang, S. Dixit, V. Zhang, C. Xiong, R. Socher, W. S. Lasecki, and D. Radev, "Cosql: A conversational

text-to-sql challenge towards cross-domain natural language interfaces to databases," 2019. [Online]. Available: https://arxiv.org/abs/1909.05378

[49] A. Tuan Nguyen, M. H. Dao, and D. Q. Nguyen, "A pilot study of text-to-SQL semantic parsing for Vietnamese," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, and Y. Liu, Eds. Online: Association for Computational Linguistics, Nov. 2020, pp. 4079–4085. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.364/

[50] Y. Zhang, J. Deriu, G. Katsogiannis-Meimarakis, C. Kosten, G. Koutrika, and K. Stockinger, "Sciencebenchmark: A complex real-world benchmark for evaluating natural language to sql systems," *Proc. VLDB Endow.*, vol. 17, no. 4, p. 685–698, Dec. 2023. [Online]. Available: https://doi.org/10.14778/3636218.3636225

[51] S. Chang, J. Wang, M. Dong, L. Pan, H. Zhu, A. H. Li, W. Lan, S. Zhang, J. Jiang, J. Lilien, S. Ash, W. Y. Wang, Z. Wang, V. Castelli, P. Ng, and B. Xiang, "Dr.spider: A diagnostic evaluation benchmark towards text-to-sql robustness," 2023. [Online]. Available: https://arxiv.org/abs/2301.08881

[52] C. Zhang, Y. Mao, Y. Fan, Y. Mi, Y. Gao, L. Chen, D. Lou, and J. Lin, "Finsql: Model-agnostic llms-based text-to-sql framework for financial analysis," in *Companion of the 2024 International Conference on Management of Data*, ser. SIGMOD '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 93–105. [Online]. Available: https://doi.org/10.1145/3626246.3653375

[53] Z. Hong, Z. Yuan, Q. Zhang, H. Chen, J. Dong, F. Huang, and X. Huang, "Next-generation database interfaces: A survey of llm-based text-to-sql," *CoRR*, vol. abs/2406.08426, 2024.

[54] M. Pourreza and D. Rafiei, "Din-sql: Decomposed in-context learning of text-to-sql with self-correction," *NeurIPS*, 2023.

[55] B. Wang, C. Ren, J. Yang, X. Liang, J. Bai, L. Chai, Z. Yan, Q.-W. Zhang, D. Yin, X. Sun, and Z. Li, "Mac-sql: A multi-agent collaborative framework for text-to-sql," *COLING*, 2025.

[56] S. T. et al., "Chess: Contextual harnessing for efficient sql synthesis," *CoRR*, vol. abs/2405.16755, 2024.

[57] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proceedings of the 25th International Conference on Very Large Data Bases*, ser. VLDB '99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, p. 518–529.

[58] M. Pourreza, H. Li, R. Sun, Y. Chung, S. Talaei, G. T. Kakkar, Y. Gan, A. Saberi, F. Ozcan, and S. O. Arik, "Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql," *CoRR*, 2024.

[59] Y. Gao, Y. Liu, X. Li, X. Shi, Y. Zhu, Y. Wang, S. Li, W. Li, Y. Hong, Z. Luo, J. Gao, L. Mou, and Y. Li, "A preview of xiyan-sql: A multi-generator ensemble framework for text-to-sql," *CoRR*, vol. abs/2411.08599, 2025.

[60] X. Xie, G. Xu, L. Zhao, and R. Guo, "Opensearch-sql: Enhancing text-to-sql with dynamic few-shot and consistency alignment," *CoRR*, vol. abs/2502.14913, 2025.

[61] Y. D. Dönder, D. Hommel, A. W. Wen-Yi, D. Mimno, and U. E. S. Jo, "Cheaper, better, faster, stronger: Robust text-to-sql without chain-of-thought or fine-tuning," *CoRR*, vol. abs/2505.14174, 2025.

[62] Z. Cao, Y. Zheng, Z. Fan, X. Zhang, W. Chen, and X. Bai, "RSL-SQL: robust schema linking in text-to-sql generation," *CoRR*, vol. abs/2411.00073, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2411.00073

[63] G. Qu, J. Li, B. Li, B. Qin, N. Huo, C. Ma, and R. Cheng, "Before generation, align it! A novel and effective strategy for mitigating hallucinations in text-to-sql generation," in *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, L. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, 2024, pp. 5456–5471. [Online]. Available: https://doi.org/10.18653/v1/2024.findings-acl.324

[64] H. A. Caferolu and O. Ulusoy, "E-sql: Direct schema linking via question enrichment in text-to-sql," *arXiv*, 2024.

[65] S. Robertson and H. Zaragoza, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends in Information Retrieval*, vol. 3, no. 4, pp. 333 – 389, 2009.

[66] K. Maamari, F. Abubaker, D. Jaroslawicz, and A. Mhedhbi, "The death of schema linking? text-to-sql in the age of well-reasoned language models," *CoRR*, vol. abs/2408.07702, 2024.

[67] X. D. et al., "C3: Zero-shot text-to-sql with chatgpt," *CoRR*, vol. abs/2307.07306, 2023.

[68] L. Sheng and S. Xu, "CSC-SQL: corrective self-consistency in text-to-sql via reinforcement learning," *CoRR*, vol. abs/2505.13271, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2505.13271

[69] D. Lee, C. Park, J. Kim, and H. Park, "MCS-SQL: leveraging multiple prompts and multiple-choice selection for text-to-sql generation," *COLING*, 2025.

[70] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. H. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," *ICLR*, 2023.

[71] A. Askari, C. Poelitz, and X. Tang, "Magic: Generating self-correction guideline for in-context text-to-sql," *CoRR*, vol. abs/2406.12692, 2024.

[72] K. Maamari, C. Landy, and A. Mhedhbi, "Genedit: Compounding operators and continuous improvement to tackle text-to-sql in the enterprise," 2025. [Online]. Available: https://arxiv.org/abs/2503.21602

[73] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch, "Natural language interfaces to databases - an introduction," *CoRR*, vol. abs/cmp-lg/9503016, 1995.

[74] A. Quamar, V. Efthymiou, C. Lei, and F. Özcan, "Natural language interfaces to data," *Found. Trends Databases*, vol. 11, no. 4, 2022.

[75] B. Li, Y. Luo, C. Chai, G. Li, and N. Tang, "The dawn of natural language to sql: Are we fully ready?" *CoRR*, vol. abs/2406.01265, 2024.

[76] X. Liu, S. Shen, B. Li, P. Ma, R. Jiang, Y. Luo, Y. Zhang, J. Fan, G. Li, and N. Tang, "A survey of nl2sql with large language models: Where are we, and where are we going?" *CoRR*, vol. abs/2408.05109, 2024.

[77] W. Zhang, Y. Wang, Y. Song, V. J. Wei, Y. Tian, Y. Qi, J. H. Chan, R. C.-W. Wong, and H. Yang, "Natural language interfaces for tabular data querying and visualization: A survey," *CoRR*, vol. abs/2310.17894, 2024.

[78] M. Malekpour, N. Shaheen, F. Khomh, and A. Mhedhbi, "Towards optimizing SQL generation via LLM routing," *CoRR*, vol. abs/2411.04319, 2024.

[79] D. Ding, A. Mallick, C. Wang, R. Sim, S. Mukherjee, V. Rühle, L. V. S. Lakshmanan, and A. H. Awadallah, "Hybrid LLM: cost-efficient and quality-aware query routing," *CoRR*, vol. abs/2404.14618, 2024.

[80] I. Ong, A. Almahairi, V. Wu, W. Chiang, T. Wu, J. E. Gonzalez, M. W. Kadous, and I. Stoica, "Routellm: Learning to route llms with preference data," *CoRR*, vol. abs/2406.18665, 2024.

[81] D. Jiang, X. Ren, and B. Y. Lin, "Llm-blender: Ensembling large language models with pairwise ranking and generative fusion," *CoRR*, vol. abs/2306.02561, 2023.

[82] L. Chen, M. Zaharia, and J. Zou, "Frugalgpt: How to use large language models while reducing cost and improving performance," *CoRR*, vol. abs/2305.05176, 2023.

[83] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. R. Radev, "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task," *CoRR*, vol. abs/1809.08887, 2018.

[84] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter," *CoRR*, vol. abs/1910.01108, 2019.

[85] P. B. Chen, F. Wenz, Y. Zhang, D. Yang, J. Choi, N. Tatbul, M. Cafarella, Ç. Demiralp, and M. Stonebraker, "Beaver: an enterprise benchmark for text-to-sql," *CoRR*, vol. abs/2409.02038, 2024.

[86] G. Pinna, Y. Perezhohin, L. Manzoni, M. Castelli, and A. De Lorenzo, "Redefining text-to-SQL metrics by incorporating semantic and structural similarity," *Sci. Rep.*, 2025.

[87] K. Luoma and A. Kumar, "Snails: Schema naming assessments for improved llm-based sql inference," *SIGMOD*, 2025.

APPENDIX A PROMPTS FOR SCHEMA LINKER

Table-then-Column Schema Linking (TCSL)

Table Linking Prompt

```
### OBJECTIVE ###
You are tasked with identifying which tables from a schema are related to the
provided user question.
You will be provided:
- An input user question{% if evidence and evidence|length > 0 %} and hint
(knowledge evidence){% endif %}
- The schema

### FORMATTING ###
Your output should be of the following valid JSON format!
{
"rationale": <str: the reasoning behind decision>,
"tables": <list[str]: relevant tables>
}

### OUTPUT ###
<USER QUESTION>: {{ user_question }}
{% if evidence and evidence|length > 0 -%}
<HINT>: {{ evidence }}{% endif %}
<SCHEMA>: {{ full_schema }}
<OUTPUT JSON>:
```

Column Linking Prompt

```
### OBJECTIVE ###
You are tasked with identifying which columns from a schema are related to the
provided user question.
You will be provided:
- An input user question{% if evidence and evidence|length > 0 %} and hint
(knowledge evidence){% endif %}
- The schema

### FORMATTING ###
Your output should be of the following valid JSON format!
{
"rationale": <str: the reasoning behind decision>,
<str: table name 1>: <list[str]: relevant columns>,
<str: table name 2>: <list[str]: relevant columns>,
...
}

### OUTPUT ###
<USER QUESTION>: {{ user_question }}
{% if evidence and evidence|length > 0 -%}
<HINT>: {{ evidence }}{% endif %}
<SCHEMA>: {{ filtered_tables_schema }}
<OUTPUT JSON>:
```

Single-Column Schema Linking (SCSL)

```
### OBJECTIVE ###
You are tasked with identifying whether or not a candidate column from a schema
is related to the provided input request.
You will be provided:
- An input NL query (and potentially a hint)
- A candidate column to evaluate

### FORMATTING ###
Your output should be of the following json format
{
    "rationale": <str: the reasoning behind decision>,
    "relevant": <bool: relevant or not>,
}

### OUTPUT ###
<INPUT QUERY>: {{ user_question }}
<HINT>: {{ hint }}
<CANDIDATE COLUMN>: {{ candidate_column }}
<OUTPUT>:
```

APPENDIX B PROMPTS FOR SQL GENERATION

Zero Shot Prompt

```
### Objective ###
You are a database expert.
Your task is to read the database schema, understand the user question{% if
evidence and evidence|length > 0 %} and hint (knowledge evidence){% endif %},
and create a valid SQLite query to answer the question.

### Database Schema ###
This schema provides a detailed overview of the database structure, including
tables, columns, primary/foreign keys, and relevant information regarding
relationships or constraints. Focus on the description beside each column, they
directly hint which columns are relevant for our question.
{{ schema_linker_db_schema }}

### Instructions ###
1. When you need to find the highest or lowest values based on a certain
condition, using ORDER BY + LIMIT 1 is prefered over using MAX/MIN within sub
queries.
2. If predicted query includes an ORDER BY clause to sort the results, you
should only include the column(s) used for sorting in the SELECT clause if the
question specifically ask for them. Otherwise, omit these columns from the
SELECT.
3. If the question doesnt specify exactly which columns to select, between name
column and id column, prefer to select id column.
4. Make sure you only output the information that is asked in the question. If
the question asks for a specific column, make sure to only include that column
in the SELECT clause, nothing more.
5. Predicted query should return all the information asked in the question
without any missing or extra information.
6. No matter of how many things the question asks, you should only return one
SQL query as the answer having all the information asked in the question.
7. Never use || to concatenate columns in the SELECT. Rather output the columns
as they are.
```

```
8. If you are joining multiple tables, make sure to use alias names for the
tables and use the alias names to reference the columns in the query. Use T1,
T2, T3, ... as alias names.
9. If you are doing a logical operation on a column, such as mathematical
operations and sorting, make sure to filter null values within those columns.
10. Only output the SQL query, no explanations or markdown punctuation needed.

### User Question (input){% if evidence and evidence|length > 0 %}, Hint
(knowledge evidence){% endif %} and SQL Query (output) ###
Question: {{ user_question }}
{% if evidence and evidence|length > 0 -%}Hint: {{ evidence }}{% endif %}
SQL:
```

Few Shot Prompt

```
### Objective ###
You are a database expert.
Your task is to read the database schema, understand the user question{% if
evidence and evidence|length > 0 %} and hint (knowledge evidence){% endif %},
and create a valid SQLite query to answer the question.

### Examples ###
Some example user questions and corresponding SQL queries are provided based on
similar problems from different databases.
{%- for example in example_selector_examples %}
Question: {{ example[0] }}
{% if evidence and evidence|length > 0 %}Hint: {{ example[1] }}{%- endif %}
SQL: {{ example[2] }}
{% endfor %}

### Database Schema ###
This schema provides a detailed overview of the database's structure, including
tables, columns, primary/foreign keys, and relevant information regarding
relationships or constraints. Focus on the description beside each column, they
directly hint which columns are relevant for our question.
{{ schema_linker_db_schema }}

### Instructions ###
1. When you need to find the highest or lowest values based on a certain
condition, using ORDER BY + LIMIT 1 is prefered over using MAX/MIN within sub
queries.
2. If predicted query includes an ORDER BY clause to sort the results, you
should only include the column(s) used for sorting in the SELECT clause if the
question specifically ask for them. Otherwise, omit these columns from the
SELECT.
3. If the question doesn't specify exactly which columns to select, between
name column and id column, prefer to select id column.
4. Make sure you only output the information that is asked in the question. If
the question asks for a specific column, make sure to only include that column
in the SELECT clause, nothing more.
5. Predicted query should return all the information asked in the question
without any missing or extra information.
```

```
6. No matter of how many things the question asks, you should only return one
SQL query as the answer having all the information asked in the question.
7. Never use || to concatenate columns in the SELECT. Rather output the columns
as they are.
8. If you are joining multiple tables, make sure to use alias names for the
tables and use the alias names to reference the columns in the query. Use T1,
T2, T3, ... as alias names.
9. If you are doing a logical operation on a column, such as mathematical
operations and sorting, make sure to filter null values within those columns.
10. Only output the SQL query, no explanations or markdown punctuation needed.

### User Question (input){% if evidence and evidence|length > 0 %}, Hint
(knowledge evidence){% endif %} and SQL Query (output) ###
Question: {{ user_question }}
{% if evidence and evidence|length > 0 -%}Hint: {{ evidence }}{% endif %}
SQL:
```

APPENDIX C PROMPTS FOR SQL CORRECTION

Syntax Correction Prompt

```
### Instruction ###
Fix the given unparsable SQLite SQL Query based on the provided database schema
and parsing error.
Only reply with the fixed SQL Query, without any comments or markdown
punctuation.

### Database Schema ###
{{ schema_linker_db_schema }}

### Unparsable SQL Query###
{{ sql_query }}

### Parsing Error ###
{{ parsing_error }}

### Fixed SQL Query ###
SQL Query:
```

APPENDIX D PROMPTS FOR TEXTUAL QUERY AUGMENTATION

N1-to-N3 Prompt

```
Abbreviate the database schema identifier to make it significantly shorter:
Protocol_Name -> P_Nm

Abbreviate the database schema identifier to make it significantly shorter:
WaterTemperature -> WtTp

Abbreviate the database schema identifier to make it significantly shorter:
Vehicle_Identification_Number -> VIN

Abbreviate the database schema identifier to make it significantly shorter:
NationalAeronauticalSpaceAgency -> NASA

Abbreviate the database schema identifier to make it significantly shorter:
Banking_Table -> B_Tbl

Abbreviate the database schema identifier to make it significantly shorter:
{{ _IDENTIFIER_ }} ->
```

N2-to-N3 Prompt

```
Abbreviate the database schema identifier to make it significantly shorter:
AirBagLocCurtain -> AbgLocCtn

Abbreviate the database schema identifier to make it significantly shorter:
WaterTemp-> WtTp

Abbreviate the database schema identifier to make it significantly shorter:
Veh_ID_Num -> VIN

Abbreviate the database schema identifier to make it significantly shorter:
Natnl_Aero_Space_Agncy -> NASA

Abbreviate the database schema identifier to make it significantly shorter:
Bnk_Tbl -> B_Tbl

Abbreviate the database schema identifier to make it significantly shorter:
{{ _IDENTIFIER_ }} ->
```

Decrease NL Naturalness Prompt

```
You are tasked with making natural language questions and evidence less natural
by replacing table and column names with their abbreviated forms. Only replace
words that refer to database schema elements (tables and columns), not values
or other content.

### FORMATTING ###
Your output should be of the following json format:
{
  "updated_question": <str>,
  "updated_evidence": <str>
}

## Schema Mapping (Original -> Abbreviated):
{{ _SCHEMA_MAPPING_ }}

## Examples:

### Example 1:
**Original Question:** Which restaurant has the highest rating and how many
reviews does it have?
**Original Evidence:** Restaurant information is in the restaurants table;
highest rating refers to MAX(average_rating); review count is stored in the
review_count column
**Output:**
{
  "updated_question": "Which rest has the highest rating and how many rvws does
it have?",
  "updated_evidence": "Rest information is in the rests table; highest rating
refers to MAX(avg_rtng); rvw count is stored in the rvw_cnt column"
}

### Example 2:
**Original Question:** Show me the names of students who have taken all
available courses in the Computer Science department.
**Original Evidence:** Student names are in the student table; courses taken
are in the enrollment table; department filter is department_name = 'Computer
Science'
```

```
**Output:**
{
  "updated_question": "Show me the names of stdnts who have taken all available
crses in the Comp Sci dpt.",
  "updated_evidence": "Stdnt names are in the stdnt table; crses taken are in
the enrl table; dpt filter is dpt_nm = 'Computer Science'"
}

### Example 3:
**Original Question:** Which medical center had the most patient visits during
the winter months of 2022?
**Original Evidence:** Medical center information is in the hospital table;
winter months are from December to February; visit counts are tracked in the
patient_visit table
**Output:**
{
  "updated_question": "Which med ctr had the most pt visits during the winter
months of 2022?",
  "updated_evidence": "Med ctr information is in the hosp table; winter months
are from December to February; visit counts are tracked in the pt_vst table"
}

## Task:
Now apply the same approach to the following:

**Original Question:** {{ _QUESTION_ }}
**Original Evidence:** {{ _EVIDENCE_ }}
**Output:**
{
  "updated_question": <str>,
  "updated_evidence": <str>
}
```