SQLyzr: A Comprehensive Benchmark and Framework for Evaluating Text-to-SQL Systems

by

Sepideh Abedini

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2026

© Sepideh Abedini 2026

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Natural language–to–SQL (text-to-SQL) systems aim to enable users to interact with relational databases using natural language instead of SQL. Recent advances in large language models have significantly improved the performance of these systems, making them increasingly practical for real-world applications. With the rapid pace of progress and the growing adoption of text-to-SQL systems, robust benchmarking has become essential. However, existing benchmarks typically rely on a single correctness metric, lack alignment with real-world query usage patterns, and do not evaluate the scalability of generated queries, which limits their ability to provide realistic and practically meaningful evaluation.

This thesis introduces **SQLyzr**, a comprehensive text-to-SQL benchmark and evaluation framework designed to address these limitations. SQLyzr incorporates a fine-grained taxonomy of SQL queries and reports evaluation results at the level of query categories and subcategories, enabling detailed insights into system performance across different query types. In addition, SQLyzr extends traditional evaluation by introducing complementary metrics that assess not only the correctness but also the efficiency and structural complexity of generated SQL queries. To better reflect real-world usage, SQLyzr aligns the distribution of query categories with empirical SQL workload distributions and supports dataset scaling to enable evaluation on larger databases.

Building on these ideas, we also introduce a configurable text-to-SQL benchmarking framework that allows users to customize and extend benchmark components such as workloads, datasets, and evaluation metrics. The framework further provides novel features such as detailed error analysis for identifying incorrect queries with minor issues and workload augmentation for synthesizing additional question-SQL pairs that target weaknesses of a given text-to-SQL system.

We use SQLyzr to evaluate two state-of-the-art text-to-SQL systems with similar overall correctness scores. Our results demonstrate that SQLyzr enables clearer comparison between systems and reveals deeper insights into their relative strengths and weaknesses.

To support reproducibility and future research, the SQLyzr benchmark and framework are publicly available on GitHub at: https://github.com/sepideh-abedini/SQLyzr.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Professor Tamer Özsu, for his guidance, kindness, and trust throughout my master's journey. He always believed in me and gave me the opportunity to grow, develop my skills, and focus on this work under his supervision. His support, patience, and thoughtful advice were instrumental in shaping my academic path and helping me find my own direction. I am truly grateful for the opportunity to learn from him and to work under the mentorship of one of the most respected scholars in the field.

I would also like to thank the members of my examining committee, Professor Lukasz Golab and Professor Grant Weddell, for taking the time to review this thesis and provide valuable feedback.

Last but not least, I would like to thank my family for their unwavering love and support. While writing this thesis, my home country was going through a difficult time of war. During many parts of this process, my family had no internet access and no way to contact me, and the uncertainty of not knowing whether they were safe was one of the hardest challenges I faced day and night. Despite the distance and the silence, their strength and support have always been the foundation that carried me forward. I promised myself to push my boundaries in this field to make them proud and honor the sacrifices they made for me.

Dedication

I dedicate this thesis to my family and to all the brave people in my home country who continue to show resilience, courage, and hope in the face of adversity.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

Dedication v

List of Figures x

List of Tables xiii

1 Introduction 1

2 Background and Related Work 8

2.1 text-to-SQL Translation Task 8

2.1.1 Background and Terminology 8

2.1.2 Evolution of text-to-SQL Systems 9

2.2 Text-to-SQL Benchmarking 12

2.2.1 Background and Terminology 12

2.2.2 Benchmarking Datasets 12

2.2.3 Benchmarking Workloads 13

2.2.4 Evaluation Metrics 15

2.2.5 Limitations of Existing Benchmarks 16

3 SQLyzr Benchmark **18**
3.1 SQLyzr Workload . . . 19
3.1.1 Workload Composition . . . 19
3.1.2 Fine-Grained Taxonomy of SQL Queries . . . 20
3.1.3 Construction of Realistic Workload . . . 23
3.2 SQLyzr Dataset . . . 23
3.3 SQLyzr Evaluation Metrics . . . 25
3.3.1 Correctness Metrics . . . 26
3.3.2 Efficiency Metrics . . . 31
3.3.3 Aggregating Metrics Over a Workload . . . 33

4 SQLyzr Benchmarking Framework **35**
4.1 Overview and Design Goals . . . 35
4.2 Framework Pipeline . . . 36
4.2.1 Query Prediction . . . 36
4.2.2 Category Assignment . . . 37
4.2.3 Correctness and Efficiency Assessment . . . 39
4.2.4 Error Analysis . . . 39
4.2.5 Report Generation . . . 40
4.2.6 Workload Augmentation . . . 40
4.3 Error Analysis Module . . . 41
4.3.1 Error-Fixing Rules . . . 42
4.3.2 Result Equivalence under Fixing Rules . . . 45
4.3.3 Finding a Minimal Set of Error-Fixing Rules . . . 47
4.4 Framework Configurability . . . 48
4.5 System Optimizations . . . 49
4.6 Batch API Calls . . . 50
4.7 Database Systems Integration . . . 50
4.8 Graphical User Interface (GUI) . . . 51
4.9 Artifact Release and Deployment . . . 51

5 Experiments **53**
5.1 Experimental Setup . . . 54
5.1.1 Hardware Specification . . . 54
5.1.2 Text-to-SQL Systems . . . 55
5.1.3 Workload and Dataset . . . 57
5.1.4 SQLyzr Configuration . . . 57
5.2 Evaluation Results . . . 58
5.3 Fine-Grained Analysis . . . 60
5.3.1 Evaluation Metrics at Category-Level . . . 60
5.3.2 Subcategory-Level Analysis . . . 62
5.4 Aligning Workload with SQLShare . . . 64
5.4.1 Evaluation Results . . . 66
5.5 Dataset Scaling . . . 67
5.5.1 Construction of the Scaling Workload . . . 67
5.5.2 Evaluation Results . . . 67
5.6 Summary of Findings . . . 69

6 Conclusion & Future Work **76**
6.1 Conclusion . . . 76
6.2 Future Work . . . 78
6.2.1 Partial Correctness Metrics . . . 78
6.2.2 Semantic Correctness Assessment . . . 78
6.2.3 Human-in-the-Loop Evaluation . . . 78
6.2.4 Dynamic Taxonomy Construction . . . 79
6.2.5 LLM-Assisted Error-Fixing Suggestions . . . 79
6.2.6 Extensions Beyond SQL . . . 79
6.3 Final Remarks . . . 79

References **81**

APPENDICES **91**

A Workload Augmentation with LLM **92**

A.1 Workload Augmentation Prompt 92

B Error-Fixing Suggestions **97**

List of Figures

1.1 A single natural language question with multiple semantically equivalent queries. 4

1.2 An overview of the SQLyzr framework. 5

3.1 An example of a natural language question with evidence 20

3.2 Distribution of query categories in the SQLShare workload 24

3.3 An example where execution accuracy penalizes a semantically valid prediction 27

3.4 An example of different column ordering 28

3.5 An example of semantically equivalent queries that differ syntactically . . . 28

3.6 An example of letter-casing mismatch in string literals 30

3.7 An example of unnecessary structural complexity in the predicted query . . 32

4.1 Four stages of targeted workload augmentation in SQLyzr. Stage 1: Initial evaluation of a text-to-SQL model using category- and subcategory-based metrics. Stage 2: Identification of under-performing SQL subcategories using the fine-grained scores. Stage 3: Generation of new question-SQL for the identified subcategories on existing databases. Stage 4: Augmentation of the workload with the newly synthesized pairs. 40

4.2 An example of an incorrect `LIMIT` clause 43

4.3 An example of a missing column in the `SELECT` clause 43

4.4 An example of an extra column in the `SELECT` clause 43

4.5 An example of extra rows caused by a missing filtering condition 44

4.6 An example of mismatched column ordering in the `SELECT` clause 44

4.7 An example of a mismatched row ordering 45
4.8 Example of minor errors in the predicted queries that yield extra rows in the result. 47
4.9 Configuration panel of the SQLyzr GUI 51

5.1 Overall micro-average evaluation scores of DIN-SQL and DAIL-SQL. Higher values indicate better scores. 59
5.2 EA and REA scores per category 61
5.3 EM and REA scores per category 62
5.4 Complexity Consistency (CC) per category. Higher values indicate that the system performed better in keeping the complexity of the predicted queries close to ground truth queries. 63
5.5 Execution Time Consistency (ETC) per category. Higher values indicate that the system performed better in keeping the execution efficiency of the predicted queries close to ground truth queries. 64
5.6 Token usage per category 65
5.7 Correctness scores per subcategory 70
5.8 Category distribution comparison between SQLyzr and SQLShare 71
5.9 Micro-average evaluation scores for DIN-SQL and DAIL-SQL (aligned workload vs. original workload) 71
5.10 Macro-average evaluation scores for DIN-SQL and DAIL-SQL (aligned workload vs. original workload). 72
5.11 EA and REA scores per category (aligned workload). 72
5.12 Exact Match per category (aligned workload). 73
5.13 Complexity Consistency (CC) per category (aligned workload). Higher values indicate that the system performed better in keeping the complexity of the predicted queries close to ground truth queries. 73
5.14 Execution Time Consistency (ETC) per category (aligned workload). Higher values indicate that the system performed better in keeping the execution efficiency of the predicted queries close to ground truth queries. 74
5.15 Token Usage per category (aligned workload) 74

5.16 Average execution time of ground truth and predicted queries across scaling factors. 75
5.17 Overall evaluation scores of the scaling workload on original and ×1000 scaled datasets. 75

List of Tables

3.1 Train/test split of SQLyzr workload 20
3.2 Taxonomy of SQL Queries 22
3.3 Train/test split of SQLyzr dataset 24

5.1 Experimental Environment and Hardware Specifications 54
5.2 Overall evaluation scores of DIN-SQL and DAIL-SQL. 59
5.3 Category distribution for SQLyzr, SQLShare, and aligned variants of SQLyzr 65
5.4 Overall evaluation scores of DIN-SQL and DAIL-SQL on the aligned workload 66
5.5 Category distribution for the Spider subset of SQLyzr, SQLShare, and scaling workload (aligned with SQLShare). 68

B.1 Mapping between error-fixing rules and SQLyzr's textual suggestions . . . 98

Chapter 1

Introduction

Relational databases are the most widely used form of data storage across many domains, including healthcare, finance, education, and enterprise systems [69]. However, accessing data and retrieving specific information typically requires knowledge of query languages such as SQL, which can be challenging for users without prior experience in database systems. Even for users with a technical background, formulating correct and efficient SQL queries is non-trivial and error-prone, especially when queries involve complex logic, multiple tables, or large-scale databases. In addition, it often requires familiarity with the underlying database schema and, in some cases, database-specific SQL dialects.

Text-to-SQL systems aim to address these challenges by translating natural language questions into executable SQL queries, allowing users to interact with databases using everyday natural language. With the help of these systems, users can focus primarily on what information they want to retrieve rather than how to express it in SQL, making data access and analysis available to a wider range of users.

Text-to-SQL translation has been a topic of active research since Codd's early efforts to provide natural language interfaces for databases [15]. Recent advancements in large language models have substantially improved the practicality of text-to-SQL systems, making them more robust and applicable in real-world settings, and as a result, text-to-SQL translation has become one of the active research directions in both the database and natural language processing communities [51].

Early approaches for text-to-SQL relied on rule-based systems or neural models trained for specific tasks, which required careful feature engineering and domain-specific design and lacked generalization. In contrast, recent approaches increasingly rely on large language models that are not bound to specific domains and can be used across different database

schemas with minimal or no domain-specific training [35]. With the significant performance improvements achieved by recent large language models, LLM-based text-to-SQL systems are increasingly becoming practical tools rather than remaining purely academic prototypes.

Despite this progress, text-to-SQL remains a challenging task due to several inherent complexities [35]. First, user utterances can be ambiguous and may not include all the information required for accurate translation into a SQL query. For instance, consider the question "Find the patients whose cholesterol is higher than normal". This question could be translated into a query such as `SELECT * FROM patients WHERE TCHO > K`. However, unless a definition of what "normal" means is provided, it is not possible to determine the value of `K` and generate the correct query. As a result, accurately understanding the intent of the natural language question and resolving underspecified expressions is essential for text-to-SQL translation. Second, text-to-SQL systems must identify the relevant tables and columns needed to answer a given question, even when references to the underlying database schema are not always explicitly stated. For example, consider a table named `Problems` with columns `name` and `type`. In the question "How many Combinatorics problems are there?", it is not possible to determine whether "Combinatorics" refers to the problem name or its type without extra information. Accurately mapping natural language expressions to the correct schema elements is therefore a critical challenge in text-to-SQL translation. Finally, many questions require complex query structures and non-trivial reasoning that goes beyond simple translation from text to SQL. Queries involving nested subqueries, multiple joins, aggregation, or conditional logic introduce additional challenges and further increase the complexity of the text-to-SQL task.

To assess the performance of text-to-SQL systems, many benchmarks have been proposed. Two of the most widely used benchmarks are Spider [83] and BIRD [50]. Text-to-SQL benchmarks usually include a *workload* consisting of a collection of natural language questions paired with ground truth SQL queries (question-SQL pairs), a *dataset* composed of populated databases, and a set of *evaluation metrics* designed to assess the correctness and other qualities of the translation. While these benchmarks are useful, they have several limitations.

First, there is a lack of fine-grained evaluation of text-to-SQL systems. Most existing benchmarks report a single aggregated score over the entire workload without any fine-grained assessment provided. While such a score is useful as a high-level indicator of overall performance, it does not reveal much insight into which types of queries are difficult or how text-to-SQL systems behave across different SQL patterns. More importantly, text-to-SQL systems are increasingly being used in critical domains such as finance and healthcare. In these settings, understanding system behaviour and its limitations is as

important as measuring correctness. Users need to know which types of queries a system handles accurately and where it tends to fail in order to use these systems with confidence. However, current benchmarks and evaluation practices provide limited support for answering these questions.

Second, commonly used evaluation metrics can be overly strict and may not precisely reflect the actual performance of a text-to-SQL system. For example, execution accuracy (EA) [67] evaluates queries based on the equivalence of their execution results. Since EA is a binary metric, even small differences in the output, such as the presence of an extra column, can result in a score of zero, which can underestimate the actual accuracy of the text-to-SQL system. Moreover, existing benchmarks primarily focus on the correctness of generated queries and typically ignore other important aspects, such as query complexity and execution efficiency. To illustrate the limitations of such metrics, consider a natural language question posed over a database containing a large volume of records about employees and departments. Figure 1.1 shows different SQL queries corresponding to the same natural language question. While all of these queries return the correct answer, they differ in structure, complexity, and execution cost. For instance, Query B in Figure 1.1 includes an unnecessary join, and Query C uses a subquery, which can make both of them less efficient than Query A. Even after query plan optimizations (e.g., in SQLite), Queries B and C can still exhibit higher execution time overhead compared to Query A. Although these queries are all considered correct under standard evaluation metrics such as EA, no insight is provided into other important aspects, such as efficiency. Such aspects are particularly crucial for the real-world deployment text-to-SQL systems.

Third, current benchmarks focus on small-scale databases and are typically evaluated on databases with a fixed number of tables and rows that do not reflect the scale of real-world enterprise use cases. While this setup simplifies benchmarking, it limits the ability to study how effectively text-to-SQL systems operate in large-scale settings. As a result, even when text-to-SQL systems achieve high performance on existing benchmarks, they may still fail to generate queries that are efficient at scale.

In addition, existing benchmarks rely on a single static workload, which provides limited diagnostic insights and does not support iterative system improvement. To illustrate, consider a benchmark on which a system achieves high correctness accuracy. Further evaluations on the same workload reveal limited insight into the system's weaknesses, since meaningful improvement requires more challenging data points that specifically target these weaknesses.

Taken together, these observations suggest that evaluating text-to-SQL systems requires more than reporting a single correctness score. Several prior works have attempted to

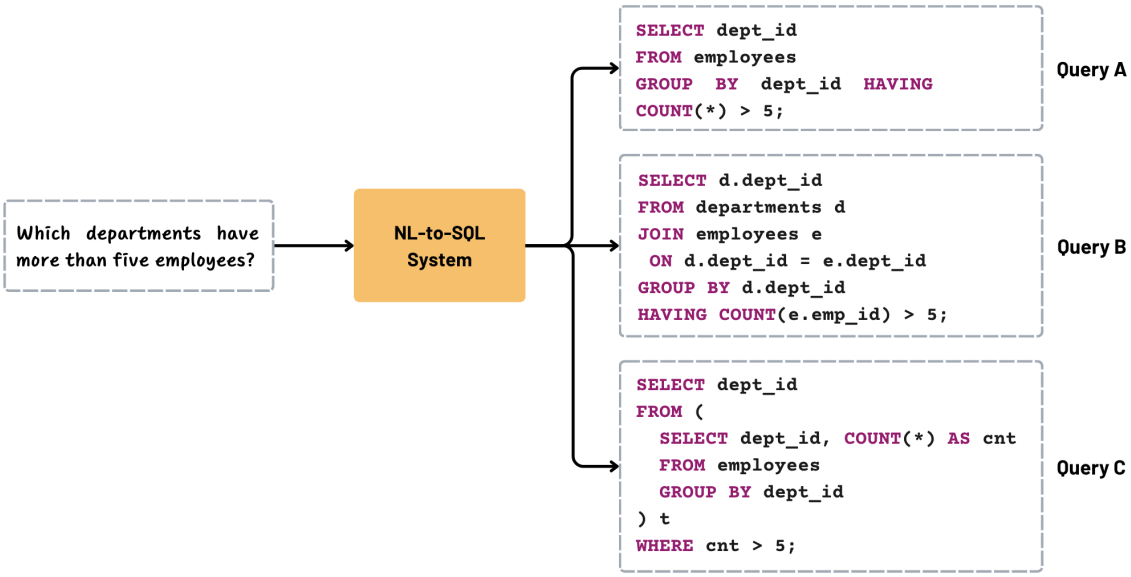


Figure 1.1: A single natural language question with multiple semantically equivalent queries.

address some aspects of these limitations. Some studies analyze system performance by classifying the workload queries according to their difficulty and provide a more fine-grained evaluation [83, 50]. However, these approaches often rely on simplistic classifications of query complexity and therefore provide limited insight into the system's behaviour across diverse SQL query types. As a result, there remains a gap between reported benchmark performance and a detailed understanding of text-to-SQL system strengths and weaknesses.

Several prior studies have explored different aspects of the limitations in evaluating text-to-SQL systems. Some works focused on analyzing the text-to-SQL errors [29, 39, 45, 32, 54]. However, they fall short of providing a detailed and comprehensive taxonomy of SQL query types that cause model failures. A few approaches [86, 83, 4] investigate alternative evaluation metrics beyond standard execution accuracy. Despite these efforts, most widely used benchmarks still remain constrained to the same traditional metrics and provide limited insight into system behaviour beyond aggregate performance.

In this thesis, we focus on addressing these limitations of text-to-SQL benchmarking by introducing a new comprehensive benchmark, called SQLyzr. The primary motivation behind SQLyzr is to enable more insightful, practical, and realistic evaluation of text-to-SQL systems beyond what is provided by existing benchmarks.

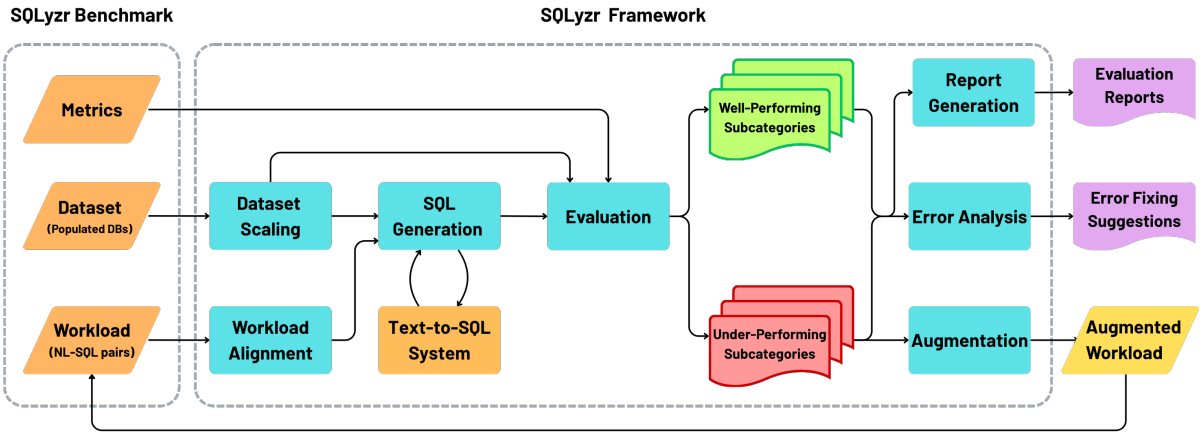


Figure 1.2: An overview of the SQLyzr framework.

The first contribution of this thesis is a text-to-SQL benchmark. SQLyzr introduces a fine-grained taxonomy of SQL queries based on their structural properties, and reports evaluation results at both the level of query categories and subcategories. In addition, SQLyzr incorporates a new correctness metric together with several efficiency-related metrics to capture aspects of system behaviour that are not reflected by standard evaluation measures. Moreover, using data synthesis, SQLyzr supports evaluation on databases of varying sizes, enabling analysis of system robustness and scalability. Together, these components aim to provide a more realistic assessment of text-to-SQL systems and to expose weaknesses that remain hidden under traditional evaluation settings.

The second contribution of this thesis is a text-to-SQL benchmarking framework that enables a configurable and extensible evaluation. Figure 1.2 shows an overview of the SQLyzr benchmarking framework.

SQLyzr takes as input a text-to-SQL workload together with its corresponding dataset. For each natural language question in the workload, the selected text-to-SQL system is invoked to generate a candidate SQL query based on the corresponding database schema. The generated queries are then evaluated by SQLyzr against the ground truth SQL queries using the defined evaluation metrics, with results reported at the level of query categories and subcategories. Based on the evaluation results, SQLyzr produces fine-grained evaluation reports and visualizations that show model behaviour across different SQL patterns and complexity levels. In addition, the framework identifies incorrect or suboptimal predictions and provides error-fixing suggestions for queries with minor errors. Moreover, SQLyzr identifies categories with scores lower than a certain threshold and generates ad-

ditional new question-SQL pairs targeting these weaknesses. The workload can then be augmented with these new pairs and used to re-evaluate the system. The augmentation enables an iterative evaluation process that focuses on the most challenging query types for a given text-to-SQL system and supports targeted system improvement.

To illustrate the capabilities of SQLyzr, this thesis presents an empirical evaluation of two state-of-the-art text-to-SQL systems. These results demonstrate how the features introduced in SQLyzr enable a more comprehensive evaluation of text-to-SQL systems and reveal that systems with similar overall correctness score can behave differently across query categories and complexity levels. Rather than replacing existing benchmarks, SQLyzr builds on them by providing additional structure, metrics, and analysis capabilities.

The contributions of this thesis are summarized as follows:

- **The SQLyzr Benchmark.** We introduce SQLyzr, a comprehensive benchmark for evaluating text-to-SQL systems, with the following characteristics:
  - A fine-grained SQL taxonomy for category- and subcategory-level analysis.
  - A realistic benchmarking workload aligned with real-world SQL usage patterns.
  - Evaluation metrics that provide more practical insights into system correctness and efficiency beyond traditional measures.
  - Scalable dataset generation to support evaluation on databases of varying sizes.
- **The SQLyzr Benchmarking Framework.** We design and implement an extensible benchmarking framework for evaluating text-to-SQL systems, with the following features:
  - A configurable evaluation pipeline that supports integration of new text-to-SQL systems, workloads, datasets, and metrics.
  - An error analysis module that provides actionable suggestions for correcting erroneous queries.
  - A workload augmentation mechanism that generates additional question-SQL pairs targeting identified system weaknesses.
  - A graphical user interface and reporting visualizations to improve the usability and interpretability of evaluation results.
- **Empirical Evaluation.** We conduct an empirical study using SQLyzr to evaluate state-of-the-art text-to-SQL systems, providing:

- Fine-grained performance analysis across query categories and subcategories.
- Demonstrate the effect of workload augmentation and dataset scaling on evaluation results.

We publicly release the SQLyzr benchmark and its framework implementation on GitHub[1].

The remainder of this thesis is organized as follows. Chapter 2 reviews related work on text-to-SQL systems and existing benchmarking approaches. Chapter 3 presents the SQLyzr benchmark, describing its workload, dataset, and evaluation metrics. Chapter 4 elaborates on the SQLyzr benchmarking framework and its main features. Chapter 5 reports experimental results obtained using SQLyzr to evaluate two state-of-the-art text-to-SQL systems and discusses the insights enabled by fine-grained evaluation. Chapter 6 concludes the thesis with a summary of findings and directions for future work.

[1] https://github.com/sepideh-abedini/SQLyzr

Chapter 2

Background and Related Work

In this chapter, we provide background on the text-to-SQL translation task and its benchmarking. Section 2.1 introduces the text-to-SQL translation task and reviews the evolution of text-to-SQL systems. Section 2.2 surveys existing benchmarks and describes their core components.

2.1 text-to-SQL Translation Task

2.1.1 Background and Terminology

Database systems are at the core of many software applications, with relational database management systems (RDBMS) being among the most widely used. In RDBMS, data is stored in structured tables defined by a rigid schema. This structured organization led to the development of Structured Query Language (SQL), which enables precise and expressive specification of operations on databases [66]. SQL remains one of the oldest programming languages still in active use, dating back to the 1970s [9].

As a technical language, SQL poses a barrier to non-expert users, who must either learn the language themselves or rely on database specialists to formulate queries [3]. Therefore, designing a more intuitive and user-friendly natural language interface for RDBMSs has been a long-standing research objective [15]. Text-to-SQL translation represents a specific formulation of this problem. Given a natural language question and a database schema, the goal is to generate a correct and executable SQL query. The translation is challenging

because it requires both syntactic correctness and precise semantic alignment with the user's intent and the underlying database schema.

The development of text-to-SQL systems has evolved alongside advances in natural language processing. Early systems relied on rule-based methods, whereas with the progress of machine learning and, in particular, language models, modern approaches shifted drastically [51].

In this section, we first provide a brief overview of text-to-SQL systems leading up to recent transformer-based approaches. Next, we discuss the current state-of-the-art text-to-SQL translation techniques that leverage large language models (LLMs).

2.1.2 Evolution of text-to-SQL Systems

Pattern Matching and Early Natural-Language Interfaces for Data Access

Early natural language interfaces for databases relied on rule-based and pattern-matching techniques. These methods handcrafted linguistic grammars and sets of rules to convert a given natural language text into a program or SQL query.

One of the earliest systems, BASEBALL [27], operated by detecting values within a user's question and retrieving matching rows from the database. Lunar Sciences Natural Language Information System (LUNAR) [76] allowed scientists to access NASA's chemical analysis data using English sentences. LUNAR relied on a sophisticated grammatical parsing of the natural language question and translated it directly into executable programs. Similarly, LADDER [34] employed hard-coded semantic grammar rules customized for a specific database schema of the US Navy. PRECISE [59] focused on reliable translation by validating whether a question could be deterministically mapped to SQL before generating a query. RENDEZVOUS [15] proposes a multi-turn conversation approach to clarify ambiguous questions with the help of the user, therefore, addressing some of the issues with the rigid grammars. Logic programming [41], which became popular in the 1980s, was adapted by systems such as CHAT-80 [74], where user questions were converted into Prolog programs.

Although these early approaches demonstrated the feasibility of natural language interfaces for databases, they were highly engineered for a specific domain and lacked generalization. Even slight deviations from their structure or grammar could cause them to fail. Their brittleness limited their adaptation to different domains.

Deep Learning and Transformer-Based Methods

With the promising performance of neural networks in natural language processing, systems began approaching the text-to-SQL translation problem using machine translation techniques. In these methods, text-to-SQL is formulated as a direct translation of natural language text into SQL queries. Using machine learning for translation allowed these methods to generalize beyond specific domains, providing more practical solutions.

Early attempts employed Sequence-to-Sequence (Seq2Seq) [70] architectures. Seq2SQL [87] was one of the earliest adaptations of Seq2Seq models for text-to-SQL translation. It decomposes a query into different components and uses Seq2Seq models for highly structured components such as `SELECT` columns and aggregation, while using reinforcement learning for the `WHERE` clause. However, it was limited to simple queries. SQLNet [77] improved on Seq2SQL by adapting a sketch-based method. The main idea was to use specific templates for the queries and then fill in the missing values with a model. In contrast to Seq2SQL, it does not use Seq2Seq and instead uses several task-specific neural networks. IRNet [30] introduces an intermediate representation, SemSQL, that sits between natural language and SQL query. A Seq2Seq model translates the text into SemQL, which is then deterministically converted into SQL.

With the introduction and success of transformer-based models, the performance of text-to-SQL translation improved significantly. Grappa [81] uses a synchronous context-free grammar (SCFG) to generate high-quality pairs of natural language questions and SQL queries accurately. This synthetic data is then used to fine-tune a RoBERTa [52] model for text-to-SQL translation. RAT-SQL [72] introduced relation-aware self-attention to encode database schemas as graphs. Graphix [49] similarly modifies the T5 architecture to add a graph-aware layer, improving the model's understanding of the schema and its relations. ResdSQL [48] employs a two-stage approach combining schema pruning with skeleton generation. It first utilizes a cross-encoder to extract the most relevant tables and columns for a question. Next, the query skeleton is generated and subsequently filled with values. NatSQL [23] builds on the idea of using an intermediate representation but simplifies and optimizes the representation for improved generalization.

LLM-Based Approaches

With the widespread adoption of Large Language Models (LLMs) and their advanced reasoning capabilities, they have become the dominant paradigm for translating natural language questions into SQL queries in recent text-to-SQL research. Current state-of-the-art text-to-SQL systems [62, 24, 61, 71, 72, 17, 47] rely heavily on in-context learning

and prompt engineering rather than task-specific training. Modern LLM-based text-to-SQL systems commonly incorporate several architectural components, which are discussed below.

Schema Linking. Schema Linking is the process of identifying which database elements (tables, columns, and relations) are relevant to a given natural language question and aligning them with the corresponding terms in the question. This process reduces noise from irrelevant schema elements, which is particularly important for databases with hundreds of tables, where irrelevant schema information can degrade the prompt quality or even exceed an LLM's context window. Knowledge retrieval complements schema linking by fetching domain-specific information, such as the actual data values stored in the rows or external metadata, to resolve ambiguities and improve the quality of the generated query.

DIN-SQL [62] includes a dedicated LLM prompt for schema-linking prior to SQL generation. C3-SQL [17] employs clear prompting, hint calibration, and consistent output formatting to guide the LLM toward producing the desired query structure. CHESS [71] adopts a multi-agent architecture consisting of an information retriever to extract relevant databases, a schema selector to prune unnecessary entities, a SQL generator, and a unit tester that evaluates the quality of the generated queries. RESDSQL [48] filters and ranks tables and columns before proceeding with SQL generation.

Example Extraction. Example extraction refers to the process of selecting relevant question-SQL pairs to include in the LLM prompt for in-context learning. Since LLMs are highly sensitive to the context provided in the prompt, supplying high-quality examples—particularly those with similar structure or logic—can significantly improve the generated output.

DIN-SQL [62] employs a fixed set of demonstration examples for each prompt, regardless of the input question. In contrast, DAIL-SQL [24] improves example selection by dynamically retrieving the most relevant question-SQL pairs based on their semantic similarity to the given natural language question.

Decomposition. Decomposition is another common technique in text-to-SQL systems, where the text-to-SQL translation task is divided into smaller and more manageable sub-tasks. Each sub-task focuses on a specific aspect of the translation process, which can improve the overall success rate by reducing the complexity handled at each step.

For example, CHASE-SQL [61] first prompts the LLM to decompose the query generation process into smaller components, then generates partial SQL fragments for each sub-task, and finally composes them into the final query.

Post-Processing. Post-processing refers to refining the SQL query generated by the LLM, such as correcting syntax errors or ensuring consistent use of table aliases. Some methods

additionally execute the generated query against the database. Then, if an execution error or an empty result is returned, the error message is fed back into the LLM to trigger a refinement step.

DIN-SQL [62] popularized the use of a self-correction module as a post-processing stage that checks for common semantic errors using the LLM itself. CHESS [71] employs a separate testing module that executes the query and, in case of errors, prompts the LLM to regenerate the query. CHASE [61] generates multiple candidate queries and uses a dedicated LLM agent to choose the most appropriate candidate as the final output.

2.2 Text-to-SQL Benchmarking

With the rapid advancement and deployment of text-to-SQL systems, many benchmarks have been introduced to evaluate and compare different approaches. In this section, we discuss some of the existing benchmarks and describe their key components.

2.2.1 Background and Terminology

A text-to-SQL benchmark typically consists of three main components:

- **Dataset:** A collection of populated databases.
- **Workload:** A set of natural language questions paired with the ground truth SQL queries.
- **Evaluation Metrics:** A set of metrics to assess the correctness and other properties of predicted queries.

2.2.2 Benchmarking Datasets

An essential component of text-to-SQL benchmarks is the dataset, which consists of populated databases used to execute queries and evaluate their correctness and efficiency. Existing benchmarks typically release these datasets as part of the benchmark package, covering databases of varying sizes and structural complexity.

WikiSQL [87] provides single-table databases extracted from Wikipedia pages. However, these databases are relatively simple and lack relational constraints such as primary

and foreign keys. Spider [83] significantly increased both the diversity and scale of datasets by including more than 200 multi-table databases across diverse domains. More recently, BIRD [50] further increased complexity by introducing 95 large and structurally complex databases, with a total size exceeding 33 gigabytes. The most recent and complex dataset is introduced by BEAVER [11], which includes millions of rows and enterprise-level schema characteristics.

2.2.3 Benchmarking Workloads

The workload of text-to-SQL benchmarks consists of natural-language questions paired with corresponding SQL queries that are manually annotated and verified, referred to as *"ground truth"* queries. The workload is typically the primary focus of the benchmark design and serves as the main distinguishing factor between different benchmarks.

Workload Complexity

As text-to-SQL systems evolved, benchmarks evolved accordingly. Early text-to-SQL systems were often specialized for specific domains and tasks. Therefore, early benchmarks focused on domain-specific workloads [33, 36, 60, 19, 64, 73, 32, 44]. With improvements in generalization of these systems, benchmarks expanded beyond single-domain settings. Instead of being restricted to a single database, newer benchmarks introduced cross-domain evaluation settings [79, 43, 83, 50, 11]. The primary objective of cross-domain benchmarks is to assess how a text-to-SQL system performs on databases and schemas that differ from those observed during training.

One of the most widely used benchmarks is Spider [83], which was among the first cross-domain benchmarks to incorporate complex SQL patterns across multiple databases. BIRD [50] further increased workload complexity by introducing questions that require advanced SQL constructs, including nested queries, multiple joins, and set operations. BEAVER [11] provides a comparatively smaller workload, consisting of approximately 200 question-SQL pairs. However, it uses significantly higher query complexity and enterprise-level schema characteristics.

Several variants of these benchmarks have been proposed to evaluate the specific properties of text-to-SQL systems. AmbiQT [6] introduces ambiguous questions to assess a system's ability to handle ambiguity. Extensions of Spider [21, 10, 20] introduce variations of original questions to evaluate robustness against linguistic perturbations such as

synonym substitution. Other works [22, 43] incorporate additional domain knowledge into questions to assess a system's ability to leverage implicit or external knowledge.

Taxonomy of Queries and Error Analysis

To provide deeper insight into text-to-SQL systems' performance, several benchmarks classify workload entries and analyze translation errors based on structural categories. Such categorization enables understanding which types of queries are easier or more challenging for models.

Spider [83] introduces a coarse-grained difficulty categorization—easy, medium, hard, and extra hard—based on the structural characteristics of the ground truth queries. Spider's classification has been adopted by subsequent benchmarks [43, 44]. BIRD [50] provides structural statistics such as the number of SQL tokens, keywords, and joins, offering additional descriptive insights. MimicSQL [73] decomposes queries into basic components, such as aggregation operations and condition columns, and reports the correctness at the component level. However, these components primarily cover simpler SQL constructs and do not extend to more advanced patterns. SEDE [32] extracts queries from real-world Stack Exchange usage and classifies them into seven categories, including window functions and case expressions. Although this approach covers more advanced constructs compared to earlier efforts, the categorization remains limited and does not provide a comprehensive taxonomy. FIBEN [64] focuses exclusively on nested query classification, following definitions from [40].

In a recent comparative study of existing text-to-SQL benchmarks [54], workloads are analyzed using quantitative attributes such as structural and operator variety. However, these metrics capture surface-level characteristics and do not fully account for nested or compositional constructs.

A more recent analysis [25] proposes 17 categories based on the structural properties of the resulting SQL queries. Although this taxonomy is more detailed than coarse-grained classifications used by Spider, it remains limited. Moreover, some categories are derived from observed model errors, indicating that the taxonomy does not follow a well-defined and systematic approach.

Several studies [29, 39, 45, 32, 54] investigate prediction errors, yet they do not provide a comprehensive and structured error analysis framework. While many works identify challenging query characteristics, a systematic and fine-grained taxonomy covering a broad range of SQL constructs remains lacking [51]. Developing such a taxonomy is a core motivation of this thesis.

Furthermore, a well-defined taxonomy enables systematic and fine-grained control over workload construction, ensuring coverage of diverse query patterns. Such control is essential for designing a comprehensive and practically meaningful text-to-SQL benchmark [25, 54].

2.2.4 Evaluation Metrics

text-to-SQL benchmark workloads consist of natural language questions paired with ground truth SQL queries. The most common approach for evaluating the correctness of predicted queries is to compare them against the corresponding ground truth query.

Two SQL queries are semantically equivalent if they produce the same result on every possible database instance. However, determining equivalence between arbitrary SQL queries is undecidable in general [14]. Consequently, practical evaluation methods approximate query equivalence using execution-based or syntactic comparisons.

The most widely used correctness metric is execution accuracy (EA) [87], which measures the proportion of predicted queries that produce the same result as the ground truth query on a single database instance. Execution accuracy may produce false positives when two semantically different queries yield identical results on a particular dataset. It may also produce false negatives when questions involve non-determinism (e.g., limiting results to a single row from a set of rows satisfying the constraint without specifying an ordering). In such cases, the execution results might differ even when the queries are semantically equivalent.

Test suite accuracy [86] is proposed to improve the robustness of execution accuracy by executing queries over multiple database instances instead of a single one. This reduces the likelihood of false positives caused by coincidental result equivalence.

Soft (or partial) execution accuracy [55] further relaxes strict equivalence by considering cases where the result of one query is a subset of the other. This approach is conceptually similar to the relaxed execution accuracy metric introduced in this work (Section 3.3).

Some approaches attempt to prove equivalence using denotational semantics within algebraic frameworks [13, 14, 28]. While theoretically rigorous, these methods are typically restricted to simplified subsets of SQL and do not support the full range of constructs used in realistic queries.

Another line of work evaluates the query equivalence using syntactic comparison. The simplest form is exact string matching, where two queries are considered equivalent only if

their strings are identical. However, syntactic matching can produce false negatives, since queries may differ syntactically while remaining semantically equivalent.

To mitigate this issue, exact match (EM) [83] compares individual SQL clauses separately and applies set-based comparisons within each clause. This way, different orderings within a component will not affect the evaluation result. ETM [4] further extends this idea by comparing the abstract syntax trees (ASTs) of queries.

Although correctness remains the dominant evaluation criterion for text-to-SQL systems, other properties of generated queries have also been considered. For example, the valid efficiency score [50] measures the relative execution efficiency of a correctly predicted query compared to the ground truth query, thereby reflecting both correctness and efficiency. Nevertheless, such non-functional attributes have received comparatively limited attention, and benchmark evaluation continues to emphasize correctness as the primary objective.

As text-to-SQL systems become increasingly integrated into practical applications, evaluating them solely based on binary correctness is insufficient. A comprehensive benchmark should assess correctness, structural adequacy, execution efficiency, scalability, and generation cost within a unified framework. Addressing this gap is a key objective of SQLyzr. Detailed definitions of the metrics used in SQLyzr are provided in Section 3.3.

2.2.5 Limitations of Existing Benchmarks

Despite substantial progress in existing benchmarks for text-to-SQL systems, several limitations remain.

First, most existing benchmarks only focus on correctness metrics, such as execution accuracy or exact match, without providing an assessment for other aspects of queries, such as efficiency or complexity.

Second, evaluation results are typically reported as a single aggregated score, which fails to provide detailed insights into the type of queries a text-to-SQL system can successfully translate and those it struggles with.

Third, the workloads used in existing benchmarks are often constructed artificially and do not reflect the real-world usage patterns, which prevents these benchmarks from effectively evaluating text-to-SQL systems in realistic scenarios.

Finally, these benchmarks rely on datasets of fixed size, which prevents systematic assessments of query behaviour under large-scale database settings.

Our primary motivation in this thesis is to address these limitations by providing a comprehensive and configurable text-to-SQL benchmark.

Chapter 3

SQLyzr Benchmark

This chapter introduces the SQLyzr benchmark, which is designed to enable fine-grained, realistic, and scalable evaluation of text-to-SQL systems. SQLyzr addresses the shortcomings of existing benchmarks by introducing a SQL query taxonomy that enables the identification of a system's difficulties across different query types. In addition, SQLyzr's workload better reflects real-world SQL usage patterns. It also uses a dataset whose size can be adjusted to enable scalability analysis. Finally, SQLyzr incorporates benchmark metrics that go beyond what existing benchmarks offer, allowing for more comprehensive evaluation.

This chapter focuses exclusively on the *benchmarking artifact* itself and presents the details of its workload, dataset, and evaluation metrics. It explains how these components address the identified limitations of existing benchmarks. SQLyzr comes with an extensible and configurable framework for benchmarking text-to-SQL systems. The design and implementation of the framework are presented in Chapter 4.

The remainder of this chapter is organized as follows. Section 3.1 describes the construction of the SQLyzr's workload, including the fine-grained taxonomy of SQL queries and its alignment with real-world query distributions. Section 3.2 presents the dataset used in SQLyzr and its scaling mechanism. Section 3.3 introduces the evaluation metrics used in the benchmark, including those introduced in SQLyzr that enable more effective assessment of both correctness and efficiency.

3.1 SQLyzr Workload

The SQLyzr workload consists of a collection of data points, each containing the following components:

- **Natural Language Question:** A question written in natural language that represents a user's request to retrieve information from a relational database.
- **Ground truth SQL Query:** The ground truth SQL query corresponding to the natural language question. Each ground truth query is manually reviewed by human annotators to ensure its correctness.
- **Database Identifier:** An identifier specifying the database for which the SQL query is targeted. text-to-SQL systems are provided with a schema of this database in order to generate a valid and executable query.
- **Evidence:** Auxiliary information associated with the question that facilitates alignment between expressions in the natural language question and elements of the underlying database schema.

Evidence is a useful component for the accurate translation of certain complex natural language questions. Such questions may contain domain-specific terms or phrases that do not directly correspond to elements of the underlying database schema. For example, consider the data point shown in Figure 3.1, extracted from the BIRD benchmark. In this question, the phrase "act in their own self-interest" carries a specific domain meaning, and corresponds to the `alignment` column. Without this additional contextual information, and relying solely on the natural language question and the schema, it would not be possible to generate the correct SQL query. Therefore, evidence plays a critical role in disambiguating user intent and guiding the alignment between natural language expressions and schema elements in the database. However, not all workloads require evidence. For instance, in Spider, questions are generally simple and self-contained, so no extra information or evidence is necessary for the query generation.

3.1.1 Workload Composition

The SQLyzr workload is curated by aggregating queries from multiple existing benchmarks, including Spider, BIRD, and BEAVER. These benchmarks provide complementary

```sql
-- Question
-- How many superheroes act in their own self-interest?

-- Evidence
-- superheroes who act in their own self-interest refers to alignment = 'Bad'

-- Gold SQL
SELECT COUNT(*) FROM superheros WHERE alignment = 'Bad';
```

Figure 3.1: An example of a natural language question with evidence

coverage of SQL query structures, database schemas, and natural language formulations, enabling SQLyzr to incorporate a diverse set of query patterns and domains.

SQLyzr provides separate training and test workloads and datasets. Given that modern text-to-SQL systems—particularly those based on large language models—require only a limited number of task-specific examples for in-context learning [7], the training split is intentionally kept small. In total, SQLyzr includes 20,979 question-SQL pairs, and we allocate ∼11% of the entries for the training split and the remainder for the test split. Table 3.1 summarizes the number of entries in each split.

Split	# Entries
Train	2,586
Test	18,393
Total	20,979

Table 3.1: Train/test split of SQLyzr workload

3.1.2 Fine-Grained Taxonomy of SQL Queries

A key feature of the SQLyzr benchmark is a comprehensive taxonomy of SQL queries that enables fine-grained evaluation of text-to-SQL systems. Each query in the workload is classified into one of six high-level categories and thirty-six subcategories according to the SQL constructs and structural patterns it contains. This contrasts with existing benchmarks that use much coarser difficulty labels such as “simple,” “moderate,” and “hard”. SQLyzr enables deeper analysis of where text-to-SQL systems may struggle.

Unlike prior benchmarks that rely on surface-level string matching to categorize queries, SQLyzr analyzes the abstract syntax tree (AST) of each SQL query to detect the presence of specific patterns or constructs such as joins, aggregations, nested subqueries, window functions, or other advanced operators. By operating on ASTs rather than raw query text, this classification approach can more accurately capture complex patterns that are difficult to detect through textual pattern matching alone.

Furthermore, AST-based categorization provides a flexible and extensible foundation. It allows users to define new categories or refine the criteria of existing ones to accommodate additional SQL constructs, thereby providing support for a broader range of query patterns.

By reporting evaluation results at the level of categories and subcategories, SQLyzr enables fine-grained analysis of the system's behaviour across different types of SQL queries. This fine-grained evaluation makes it possible to identify specific SQL patterns or constructs that are particularly challenging for the system under evaluation. Isolating these patterns allows researchers to target improvements toward the most problematic query types, making the refinement process of text-to-SQL systems more targeted and efficient.

The full taxonomy used in SQLyzr is presented in Table 3.2. The first category focuses on basic `SELECT` statements without additional clauses or complex structural constructs. The second category introduces fundamental SQL clauses such as `WHERE`, `LIMIT`, `GROUP BY`, and simple `JOIN`s. Category C_3 extends this by incorporating combinations of these clauses together with more advanced expressions such as `IS NULL` and `LIKE`. Category C_4 introduces an additional dimension of complexity by including compound queries and nested subqueries, as well as non-trivial joins (e.g., `LEFT JOIN`, `INNER JOIN`). Category C_5 further increases complexity by covering two-level nested queries and combinations of nested queries with other structural constructs. Finally, the last category C_6 is devoted to the most advanced SQL queries, including constructs such as `RANK`, common table expressions (`CTE`), and window functions with `PARTITION BY`.

We define a hierarchical set of categories ordered by increasing query complexity. The main intuition behind this ordering is that, as queries become structurally more complex, they require more reasoning steps, a deeper understanding of schema relationships, and the correct use of more advanced SQL constructs. Such queries are generally harder for humans to formulate correctly, and we anticipate that text-to-SQL systems will also encounter similar difficulties as complexity increases. This hierarchy follows the gradual introduction of SQL concepts and structures in standard database textbooks [53, 18]. Accordingly, each category gradually incorporates more advanced SQL concepts and constructs.

Table 3.2: Taxonomy of SQL Queries

Category	Subcategory	Rule
C1	S1	Multiple Columns in `SELECT`, Single Table in `FROM`
	S2	Single Star (`*`) Column in `SELECT`
C2	S3	Sorting with `ORDER BY`
	S4	Single Binary Expression in `WHERE`
	S5	Arithmetic Expressions
	S6	Aggregate Functions (e.g., `COUNT`, `AVG`)
	S7	Restrict Row Count with `LIMIT`
	S8	Removing Duplicates with `DISTINCT`
	S9	`GROUP BY` without `HAVING`
	S10	Equality `JOIN`
C3	S11	Non-Equality `JOIN`
	S12	`BETWEEN` Expressions
	S13	Single Binary Expression in `WHERE` + `JOIN`
	S14	Single Binary Expression in `WHERE` + `GROUP BY` without `HAVING`
	S15	`JOIN` + `GROUP BY` without `HAVING`
	S16	`GROUP BY` with `HAVING`
	S17	`LIKE` Expression
	S18	`IS NULL` Predicates
	S19	Multiple Expressions in `WHERE`
C4	S20	Single-Level Subquery
	S21	Compound Query (e.g., `UNION`)
	S22	Single-Level Subquery + `IN` Predicate
	S23	`INNER JOIN`
	S24	Complex `JOIN` + `ORDER BY`
	S25	Complex `JOIN` + `GROUP BY`
	S26	Complex `JOIN` + Aggregate Functions
	S27	Multiple-Table `JOIN` ($\geq$ 3 tables)
C5	S28	Single-Level Subquery + `GROUP BY` without `HAVING`
	S29	Single-Level Subquery + `JOIN`
	S30	Single-Level Subquery + Aggregate Functions

Continued on next page

Table 3.2 – continued from previous page

Category	Subcategory	Rule
C5	S31	Two-Level Subquery
	S32	`CASE` Expression
C6	S33	Subquery + Compound Query
	S34	Multi-Level Subquery ($\geq$ 3 Levels)
	S35	Common Table Expression
	S36	Window Function

3.1.3 Construction of Realistic Workload

One limitation of existing text-to-SQL benchmarks is the lack of alignment between the distribution of query types in their workloads and SQL usage patterns observed in real-world scenarios and applications. To reflect real-world usage, we align the distribution of query categories in the SQLyzr workload with the empirical distribution observed in the SQLShare dataset [37]. SQLShare is a large-scale collection of SQL queries gathered from a multi-year deployment of a database-as-a-service platform used by scientists and data analysts. Over four years, the platform accumulated approximately 24,000 queries spanning 7,900 datasets, providing a realistic snapshot of how SQL is used in practice.

To perform this alignment, we first categorized SQL queries from both workloads according to the taxonomy defined in Section 3.1.2. We then adjusted the SQLyzr workload by selectively filtering queries so that the proportion of queries in each category matches the proportions observed in SQLShare.

By ensuring that the distribution of query categories reflects real-world usage patterns, SQLyzr enables a more practically grounded evaluation of text-to-SQL systems. As a result, users can expect less deviation between system performance observed during evaluation and that observed in real-world deployment.

3.2 SQLyzr Dataset

As discussed later in Section 3.3, several evaluation metrics for text-to-SQL system rely on comparing the SQL query generated by the system with an annotated SQL query that serves as the ground truth. This comparison involves executing both queries on the corresponding databases and measuring metrics such as the correctness of the execution results

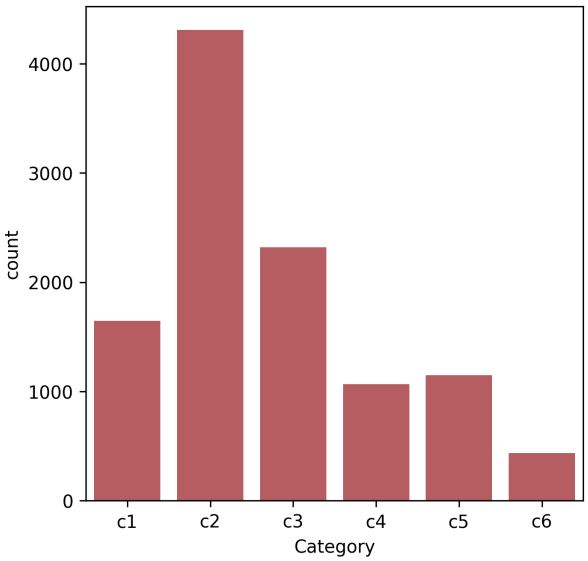


Figure 3.2: Distribution of query categories in the SQLShare workload

or efficiency-related properties of the execution. Therefore, text-to-SQL benchmarks typically provide populated databases to enable execution-based evaluation. SQLyzrś dataset is a superset of existing benchmark datasets. Specifically, the dataset is composed of the populated databases released as part of the Spider, BIRD, and BEAVER benchmarks. These databases are standardized into a unified representation to ensure consistency across evaluation runs and compatibility with the SQLyzr's evaluation pipeline.

Table 3.3 summarizes the statistics of the training and test datasets included in SQLyzr.

Split	# DBs	# Tables	# Columns	# Rows
Train	32	322	1,992	11,874,899
Test	254	1,750	11,738	54,615,247
Total	286	2,072	13,730	66,490,146

Table 3.3: Train/test split of SQLyzr dataset

Dataset Scaling. Unlike most existing benchmarks that rely on fixed-size databases, SQLyzr supports database scaling, enabling evaluation on databases of varying sizes. This capability allows the assessment of generated query latencies when a text-to-SQL system is deployed in large-scale, real-world environments. To implement dataset scaling, we integrate the Synthetic Data Vault (SDV) [57] into SQLyzr to expand existing databases using

synthetic data generation. Given a baseline populated database, users specify a scaling factor, and SDV synthesizes additional rows to increase the database size accordingly.

A key property of SDV is that it preserves the statistical characteristics of the original dataset, ensuring that the scaled database reflects the same value distributions and structural relationships. More specifically, SDV constructs a generative database model that encompasses the distribution of values in each column and the covariance between columns within a row. Inter-table relationships are modelled using Gaussian copulas, allowing the synthesis process to respect cross-table correlations. By sampling from this generative model, SDV produces synthetic records that are statistically consistent with the original data while increasing the scale of the database. This enables controlled experiments on scalability without altering the statistical characteristics of the databases.

As a proof of concept, we applied SDV to scale a subset of Spider databases and evaluate system behaviour under larger data volumes. Synthesizing data for more complex database schemas requires scalable algorithms from SDV, which are available only through its enterprise license. We plan to integrate these advanced scaling mechanisms in future releases of the SQLyzr.

3.3 SQLyzr Evaluation Metrics

A comprehensive evaluation of text-to-SQL system requires more than measuring the proportion of correctly generated queries, which is the primary evaluation metric in most existing benchmarks. While correctness is a necessary requirement, relying solely on correctness metrics obscures important differences between systems and provides limited insight into their practical performance and behaviour. To address this limitation, SQLyzr incorporates a set of complementary evaluation metrics designed to capture additional aspects of the text-to-SQL task beyond correctness alone.

The metrics in SQLyzr are organized into two main groups. The first group focuses on correctness and evaluates whether the predicted query produces the intended result under execution-based comparison. The second group captures non-functional properties of the generated query and the generation process, including structural complexity, execution efficiency, and generation cost. Together, these metrics provide a more comprehensive and practically meaningful assessment of system behaviour.

Notation

Let $\mathbb{Q}$ denote the domain of all possible SQL queries, $\mathcal{D}$ denote the dataset (the set of all databases), and $\mathcal{W}$ denote the workload. Each data point $w \in \mathcal{W}$ is represented as a tuple (s, q, d), where s is a natural-language question, $q \in \mathbb{Q}$ is the corresponding ground truth SQL query, and $d \in \mathcal{D}$ is the associated database. For simplicity, we omit the evidence field from the notation below. Let f denote a text-to-SQL system. For a given data point $w = (s, q, d)$, the system generates a predicted query $q' = f(s, d)$ based on the natural language question s and the schema of the database d. For any SQL query $q \in \mathbb{Q}$ and database $d \in \mathcal{D}$, let $R = exec(q, d)$ denote the execution result of q on d. We define metrics as functions of the form $m : \mathcal{W} \times \mathbb{Q} \to \mathbb{R}$ where $m(w, q')$ returns the score of the predicted query q' for the data point w. For binary evaluation metrics, we assume the score is either zero or one, i.e., we have $m : \mathcal{W} \times \mathbb{Q} \to \{0, 1\}$.

3.3.1 Correctness Metrics

Execution Accuracy (EA)

Execution accuracy (EA) is a common metric for assessing the semantic equivalence of two queries. This metric treats two queries as equivalent when they produce the same result on the given database instance. However, because verifying semantic equivalence is undecidable [14], execution accuracy over-approximates this notion by comparing the execution results on a single database.

Execution accuracy may produce both false positives and false negatives with respect to the intended semantics of the question. For example, two queries may return identical results on the provided test database while yielding different results on another database (false positive). Conversely, semantically equivalent queries may produce different results due to nondeterminism in query execution (e.g., missing ordering constraints).

Formally, for a data point $w = (s, q, d) \in \mathcal{W}$ and a predicted query q', execution accuracy is defined as:

$$EA(w, q') = \mathbb{I}\big[R' \equiv R\big],$$

where $\mathbb{I}[\cdot]$ denotes the indicator function, $R = exec(q, d)$, $R' = exec(q', d)$, and $\equiv$ denotes equivalence relation between execution results.

A limitation of execution accuracy is that it is a binary metric and does not differentiate between a completely incorrect query and one that contains only minor errors. For instance, consider the queries shown in Figure 3.3. The ground truth query returns the `lastname`

```sql
-- Question
-- List the full name of all students who are older than 10.

-- Gold
SELECT lastname, firstname FROM student WHERE age > 10

-- Prediction
SELECT firstname, lastname FROM student WHERE age > 10
```

Figure 3.3: An example where execution accuracy penalizes a semantically valid prediction

and `firstname` columns, whereas the predicted query returns these columns in a different order. In this case, both queries can be considered correct representations of the given question. However, since the predicted query has a different column ordering and yields a different result, EA assigns a score of zero because $R' \not\equiv R$. This strict equivalence criterion can therefore underestimate the actual performance of a text-to-SQL system. Despite these limitations, EA is a common metric used in this domain, and we also include it in SQLyzr.

Exact Match (EM)

Exact Match (EM) checks whether the structural components of a predicted SQL query match those of the ground truth query, following the definition introduced in the Spider benchmark [83]. EM performs set-based comparison over query components and therefore tolerates variations within a component, such as different column orderings in the `SELECT` clause. For example, consider the queries shown in Figure 3.4. Although the order of columns differs between the ground truth and predicted queries, EM ignores column ordering within the `SELECT` clause and therefore considers the prediction correct.

Using EM, however, a predicted query is considered incorrect if it differs structurally from the ground truth query, even when the two queries are semantically equivalent. Figure 3.5 illustrates this limitation of EM. In this example, the ground truth and predicted queries implement the same logical operation but use different syntactic structures. Despite being semantically equivalent, EM assigns a zero score to the predicted query. This demonstrates that EM may underestimate the correctness of the text-to-SQL system when semantically equivalent queries differ syntactically.

EM is also a commonly used metric for evaluating text-to-SQL systems, and therefore we include it in SQLyzr.

```sql
-- Gold
SELECT name, age FROM student WHERE age > 10

-- Prediction
SELECT age, name FROM student WHERE age > 10
```

Figure 3.4: An example of different column ordering

```sql
-- Gold
SELECT name FROM singer WHERE age BETWEEN 10 AND 20

-- Prediction
SELECT name FROM singer WHERE age >= 10 AND age <= 20
```

Figure 3.5: An example of semantically equivalent queries that differ syntactically

Exact Match Implementation. The existing implementation of EM provided by Spider relies on simple string matching and supports only basic SQL constructs. As a result, it cannot handle the more complex queries included in the SQLyzr's workload, such as nested sub-queries or complex query compositions. To address this limitation, we introduce a new mechanism for EM computation based on the AST representation of queries. Our approach compares queries by checking the equivalence of corresponding AST nodes, starting from the root and recursively traversing the tree to check the equivalence of each node. This method extends EM evaluation to support more complex SQL constructs while also providing a foundation for future extensions.

Relaxed Execution Accuracy (REA)

SQLyzr introduces Relaxed Execution Accuracy (REA) as a new metric to mitigate the strictness of EA. REA is designed to account for near-correct predictions whose execution results differ from those of the ground truth query only due to minor discrepancies, such as column ordering or extra columns. To compute REA, the predicted query is first normalized and then compared against the ground truth query under a relaxed equivalence relation, denoted by $\cong$.

Formally, let q denote the ground truth query and q' the predicted query for a workload

instance w. We define relaxed execution accuracy as

$$REA(w, q') = \mathbb{I}[q \cong q'],$$

where $\cong$ represents a relaxed query equivalence between the execution outputs of the predicted and ground truth queries as defined below.

Notation. To define this relation precisely, we introduce a notation for query execution results. Let $\mathbb{V}$ denote the domain of all possible cell values across all columns in the databases. The execution result of a query is defined as an ordered list of rows:

$$R = (r_1, r_2, \ldots, r_n)$$

where each row r is an ordered tuple

$$r = (v_1, v_2, \ldots, v_m), \quad \forall 1 \leq i \leq m,\ v_i \in \mathbb{V}$$

Definition 1 *Relaxed Row Equivalence. For two rows* $r = (v_1, v_2, \ldots, v_m)$ *and* $r' = (v'_1, v'_2, \ldots, v'_m)$*, we define relaxed row equivalence as:*

$$r \cong r' \iff |r| = |r'| \ \wedge\ \exists \sigma, \forall 1 \leq i \leq |r|,\ v_i = v'_{\sigma(i)}$$

where σ *is a permutation of* $\{1, 2, \ldots, m\}$*. This relation allows results with different column ordering to match.*

Definition 2 *Relaxed Result Equivalence. Let* R *and* R' *denote the execution results of* q *and* q'*, respectively. We define relaxed equivalence between results* $R \cong R'$ *as follows:*

$$R' \cong R \iff |R| = |R'| \ \wedge\ \exists \sigma,\ \forall\, 1 \leq i \leq |R|,\ r_i \cong r'_{\sigma(i)}$$

This relation allows results with different row-level ordering to match.

Definition 3 *Ordered Relaxed Result Equivalence. Let* R *and* R' *denote the execution results of* q *and* q'*, respectively. We define the ordered version of relaxed equivalence between results* $R \cong R'$ *as follows:*

$$R' \cong_{ord} R \iff |R| = |R'| \ \wedge\ \forall\, 1 \leq i \leq |R|,\ r'_i \cong r_i.$$

This relation lifts the row-level $\cong$ relation to the result level while preserving row ordering.

```sql
-- Gold
SELECT Country FROM AIRLINES WHERE Airline = "JetBlue";

-- Prediction
SELECT Country FROM AIRLINES WHERE Airline = "Jetblue";
```

Figure 3.6: An example of letter-casing mismatch in string literals

Definition 4 *Relaxed Query Equivalence. Finally, we define relaxed query equivalence as:*

$$q \cong q' \iff \begin{cases} R \cong_{ord} R' & \text{if } \texttt{ORDER BY} \in q, \\ R \cong R' & \text{otherwise.} \end{cases}$$

Intuitively, if the ground truth query does not enforce an explicit ordering, e.g., it does not contain an `ORDER BY` clause, REA ignores the ordering of rows. Furthermore, when comparing individual rows, REA ignores the ordering of columns within each row by treating rows as multisets of values rather than strictly ordered tuples.

Additionally, before executing the predicted query to compute REA, a set of transformations is applied to the query to correct minor issues. One such transformation is the *letter-casing matcher*.

To perform this transformation, we first compute a one-to-one mapping between string literal values appearing in the predicted query and those in the ground truth query. If a perfect match is found, we align the letter casing of each string literal in the predicted query with the one used in the ground truth query.

As an example, consider the queries shown in Figure 3.6. If the database contains the value "`JetBlue`" but not "`Jetblue`", the predicted query yields an empty result because no row satisfies the condition, and consequently, an EA score of zero is achieved. By applying the letter-casing transformation, the literal value used in the predicted query is corrected, and we can achieve the same result as the ground truth query. In this case, REA assigns a score of one, reflecting that the predicted query was semantically correct.

Currently, we only use the letter-casing transformation when computing REA. However, the design of REA is not limited to this specific transformation, and additional ones can be incorporated to address other minor discrepancies between the predicted and ground truth queries.

Another important advantage of REA is that it provides a more realistic assessment for ambiguous questions. For instance, consider the example shown in Figure 3.3. The

question-SQL question does not specify the order of columns to be included in the final result. However, any predicted query that uses a different column ordering than the ground truth query results in an EA score of zero. In these cases, the REA calculation ignores such minor deviations from the ground truth query and can reduce some of the cases where EA underestimates the actual performance of a text-to-SQL translation.

3.3.2 Efficiency Metrics

Correctness alone does not capture whether a predicted query is efficient or whether a text-to-SQL system introduces unnecessary complexity into the generated query. Although correctness remains the primary criterion for evaluating text-to-SQL system, deploying such systems in real-world settings requires consideration of additional non-functional properties of the generated queries. In practical applications, factors such as execution efficiency, resource consumption, and structural simplicity can significantly affect the usability and performance of the generated queries. SQLyzr therefore includes complementary metrics that capture execution cost, generation cost, and query complexity.

Token Usage (TU)

Large language models encode both input and output sequences as tokens, and the generation cost is typically proportional to the number of tokens consumed. Therefore, token usage can be used to estimate the generation cost of language-model-based text-to-SQL system.

For a specific language model M, we define $TU_M(s, d)$ as the total number of tokens consumed by M when generating the predicted query for the natural language question s given the schema of the database d.

Complexity Consistency (CC)

Complexity Consistency (CC) measures whether the predicted query introduces additional structural complexity compared to the ground truth query.

Formally, let $Cat(q)$ be a function that assigns a category to a given SQL query based on the taxonomy, and let the categories be ordered by increasing complexity:

$$C_1 < C_2 < \cdots < C_m,$$

```sql
-- Gold
SELECT dept_id FROM employees GROUP BY dept_id HAVING COUNT(*) > 5;

-- Prediction
SELECT dept_id FROM ( SELECT dept_id, COUNT(*) AS cnt
FROM employees GROUP BY dept_id ) t WHERE cnt > 5;
```

Figure 3.7: An example of unnecessary structural complexity in the predicted query

where C_i is considered less complex than C_j for $i < j$.

For a workload instance $w = (s, q, d)$, let $C = Cat(q)$ denote the category of the ground truth query and $C' = Cat(q')$ denote the category of the predicted query. We define complexity consistency as follows:

$$CC(w, q') = \begin{cases} \mathbb{I}[C' \leq C] & \text{if } REA(w, q') = 1 \\ 0 & \text{otherwise} \end{cases}$$

Intuitively, CC assigns a score of one if the correctly predicted query is no more complex than the ground truth query, and zero otherwise. This metric penalizes predictions that introduce unnecessary constructs, such as redundant joins or nested subqueries, which are not required for the correctness of the query.

For example, consider the queries shown in the Figure 3.7. Suppose the ground truth query is classified in category C_3, while the predicted query is classified in category C_5 due to the inclusion of an unnecessary nested subquery. In this case, we have $C' > C$, and therefore $CC(w, q') = 0$.

Execution Time Consistency (ETC)

Execution Time Consistency (ETC) evaluates whether the predicted query executes within an acceptable relative time overhead compared to the ground truth query. While correctness ensures semantic validity, ETC captures the execution efficiency of the query.

Formally, let q and q' denote the ground truth and predicted queries, respectively. Let $et(\cdot, d)$ be a function that returns the execution time of a query on a specific database instance d. We define $t = et(q, d)$ and $t' = et(q', d)$ as the execution times of the ground

truth and predicted queries, respectively. Execution time consistency is defined as:

$$ETC(w, q') = \begin{cases} \mathbb{I}\left[\frac{t'}{t} \leq \alpha\right] & \text{if } REA(w, q') = 1, \\ 0 & \text{otherwise,} \end{cases}$$

where α is a threshold that determines the acceptable execution-time overhead. The value of α is derived from the empirical execution-time distribution of queries in the benchmark workload.

To compute α, we first measure execution time variability across the workload. Let $\mathbf{t}_q = (t_1, \ldots, t_K)$ denote the vector of K repeated execution-time measurements for a ground truth query q. We define the coefficient of variation as

$$CV(\mathbf{t}_q) = \frac{\sigma(\mathbf{t}_q)}{\mu(\mathbf{t}_q)},$$

where $\sigma(\cdot)$ and $\mu(\cdot)$ denote the standard deviation and mean, respectively. The threshold α is then defined as follows:

$$\alpha = P_p\big(\{\, 1 + CV(\mathbf{t}_q) \mid q \in Q \,\}\big),$$

where Q is the set of ground truth queries in the workload and $P_p(\cdot)$ denotes the p-th percentile.

This percentile-based formulation ensures that ETC tolerates some degree of variation in execution time.

3.3.3 Aggregating Metrics Over a Workload

For a workload $\mathcal{W}$, a metric m, and a text-to-SQL system f, we use two aggregation methods to compute evaluation scores: the micro-average and macro-average.

Definition 5 *Micro-Average Let each $w \in \mathcal{W}$ be of the form $w = (s_w, q_w, d_w)$. The predicted query for w is $q'_w = f(s_w, d_w)$. The micro-average of metric m is calculated as the average over all data point in the workload:*

$$m_{micro}(\mathcal{W}, f) = \frac{1}{|\mathcal{W}|} \sum_{w \in \mathcal{W}} m(w, q'_w). \tag{3.1}$$

Definition 6 *Macro-Average Assume that there are n categories $C_1, \ldots, C_n$, and the workload is partitioned into a set of smaller workloads $\mathcal{W} = \bigcup_{i=1}^{n} \mathcal{W}_i$ where each $\mathcal{W}_i \subseteq \mathcal{W}$ contains only those data points that belong to category C_i.*

The macro-average of a metric m is then calculated by averaging the metric values across the categories:

$$m_{macro}(\mathcal{W}, f) = \frac{1}{n} \sum_{i=1}^{n} m_{micro}(\mathcal{W}_i, f) \tag{3.2}$$

Chapter 4

SQLyzr Benchmarking Framework

This chapter presents the SQLyzr benchmarking framework, a configurable and extensible evaluation framework for text-to-SQL systems, built upon the components defined in Chapter 3. While the SQLyzr benchmark comes with a predefined workload, dataset, and a set of evaluation metrics, the framework allows users to configure and extend these components according to their requirements. It also provides mechanisms for interacting with different text-to-SQL systems for query generation tasks and supports multiple RDBMS backends for query execution. Similar to the SQLyzr benchmark, fine-grained evaluation is a core feature of the framework. The framework further supports targeted error analysis and workload augmentation, allowing users to automatically generate new question-SQL pairs that focus specifically on query patterns where a system struggles.

In the remainder of this chapter, we describe the core components of the framework and elaborate on its novel features and design choices.

4.1 Overview and Design Goals

The SQLyzr framework is designed around three core principles. First, it emphasizes *fine-grained evaluation*, enabling users to analyze the text-to-SQL system performance at the level of query categories and subcategories instead of relying solely on aggregate metrics. Second, it is designed to be *modular and extensible*. It provides well-defined interfaces that allow users to integrate new systems, workloads, datasets, and evaluation metrics with minimal modification to the core functionalities of the framework. This modularity allows the framework to evolve alongside rapid advancements in language models and benchmarking methodologies. Third, the framework supports *iterative evaluation and targeted*

improvement. By identifying weaknesses of the system and using workload augmentation focused on challenging query patterns, the framework facilitates iterative refinement of text-to-SQL systems.

4.2 Framework Pipeline

The SQLyzr framework uses a multi-stage evaluation pipeline. An overview of the pipeline is illustrated in Figure 1.2, which consists of the following stages:

1. Query prediction
2. Category assignment
3. Correctness and efficiency assessment
4. Error analysis
5. Report generation
6. Workload augmentation

The inputs to the pipeline are a workload, a dataset, a text-to-SQL system under evaluation, and a set of evaluation metrics. For each data point in the workload, the framework generates a predicted SQL query using the configured text-to-SQL system, executes both the predicted and the ground truth queries on the corresponding database, computes the evaluation metrics, and finally aggregates the results into structured reports. Optionally, based on the evaluation results, the framework synthesizes new data points to augment the existing workload with more challenging queries for the system under evaluation.

In the remainder of this section, we describe each stage of the pipeline in detail.

4.2.1 Query Prediction

In the query prediction stage, the framework invokes the configured text-to-SQL system to generate a SQL query for each natural language question in the input workload. The system is treated as a black-box component that receives a natural language question together with the corresponding database schema and produces a predicted SQL query.

The current release of SQLyzr includes implementations of DIN-SQL and DAIL-SQL systems. However, the framework is not restricted to these systems. Users can integrate custom text-to-SQL systems by implementing a simple software interface provided by the framework, without modifying the core evaluation pipeline.

Multi-Iteration Evaluation

Due to the stochastic nature of large language models, the output of a text-to-SQL system may vary across different runs, even when provided with the same inputs. To mitigate this variability, SQLyzr supports multi-iteration evaluation, where the query prediction stage is executed multiple times. Rather than reporting the result of a single prediction, the framework aggregates evaluation metrics across iterations and provides statistically more stable and reliable scores.

Hyperparameter Tuning

SQLyzr supports enumerating values for configurable LLM and text-to-SQL system hyperparameters. Users specify a set of candidate values for each parameter, and SQLyzr automatically performs the evaluation runs across all specified parameter combinations and reports the corresponding results.

For example, in our experiments, we analyzed the impact of LLM generation randomness, controlled by the temperature parameter, on correctness and efficiency to find the optimal value.

4.2.2 Category Assignment

To enable fine-grained evaluation at the level of SQL query categories, SQLyzr assigns a category and subcategory to each data point based on its ground truth SQL query. Category assignment follows the taxonomy defined in Section 3.1.2 and is implemented through traversal of the query's abstract syntax tree (AST).

Tag Extraction.

Category assignment begins by first extracting a set of structural tags from each query. Tags represent specific SQL constructs or structural patterns. We define two types of tags: *ordered* and *unordered*. Ordered tags are related through a partial order relation $\preceq$, reflecting relative complexity. For two ordered tags $t_1 \preceq t_2$ if and only if t_2 is equally or

more complex than t_1 according to the taxonomy discussed previously. For example, if t_1 and t_2 correspond to unconditional and conditional `GROUP BY` clauses, respectively, then $t_1 \preceq t_2$, since a conditional `GROUP BY` introduces additional constraints and is therefore more complex.

Let s and s' be two sets of tags. We define the matching relation $\subseteq_M$ as:

$$s \subseteq_M s' \iff \forall t \in s,\ \exists t' \in s' \text{ such that } t \preceq t'$$

Intuitively, a set of tags s matches s' if for each SQL pattern (tag) in s, an equally complex or more complex pattern exists in s'.

Let $\mathbb{T}$ denote the set of all possible tags. Each subcategory is defined as a subset of tags, i.e., $s \subseteq \mathbb{T}$. To assign a subcategory to a query q, we first parse q into its AST representation and extract the corresponding tag set T_q by traversing the AST.

Among all subcategories that match the query,

$$\{\, s \in \mathcal{S} \mid s \subseteq_M T_q \,\},$$

(where $\mathcal{S} \subseteq 2^{\mathbb{T}}$ denotes the set of all subcategories), the framework assigns the *most complex* matching subcategory. This ensures that queries are not trivially assigned the simplest subcategories that matches every query.

Additionally, subcategories are partitioned into a set of disjoint categories. Formally, we define categories $C_1, C_2, \ldots, C_n$ such that each subcategory belongs to exactly one category:

$$\forall C, C',\ (s \in C \wedge s \in C') \Rightarrow C = C'.$$

We further define a total ordering $<$ among categories such that we have:

$$C_i < C_j \iff i < j,$$

which reflects increasing structural complexity.

By relying on AST traversal rather than surface-level string matching (as commonly done in prior benchmarks), SQLyzr can robustly recognize any patterns or query structures. This approach also enables flexible extensibility and supports the addition of new SQL constructs without modifying the core categorization logic.

Algorithm 1 presents the category assignment procedure. Given a SQL query q, the algorithm first parses the query and constructs its AST (line 2). It then extracts a set of tags $T_q \subseteq \mathbb{T}$ from the AST (line 3). Next, the algorithm iterates over all subcategories, starting

from S_{36} and proceeding down to S_1, to find the first subcategory that matches the query (lines 4–8). Since each subcategory is associated with exactly one category, identifying the matching subcategory also determines the corresponding category of the query. Traversing subcategories in descending order effectively corresponds to traversing categories from C_6 to C_1. This ordering ensures that more complex patterns are matched before simpler ones. If no matching subcategory is found, the algorithm returns an undefined subcategory, denoted by $S_\perp$.

Algorithm 1 Category Assignment

```
1: procedure AssignCategory(q)                          ▷ q : SQL query
2:     ast ← parse(q)
3:     T_q ← extract_tags(ast)
4:     for S ∈ {S_36, S_35, ..., S_1} do
5:         if S ⊆_M T_q then        ▷ Check whether the tags of the subcategory match the extracted tags
6:             return S
7:         end if
8:     end for
9:     return S_⊥
10: end procedure
```

4.2.3 Correctness and Efficiency Assessment

SQLyzr evaluates the generated SQL query using the correctness and efficiency metrics defined in Section 3.3. Each metric is computed for each data point and subsequently aggregated across subcategories, categories, and the entire workload (see Section 3.3.3 for formal definitions of aggregations).

4.2.4 Error Analysis

SQLyzr includes a dedicated error analysis module aimed at identifying and characterizing incorrectly predicted queries that contain minor, potentially fixable errors. Rather than treating all incorrect predictions uniformly, this module provides users with actionable feedback for fixable queries. The technical details of error analysis are described in Section 4.3.

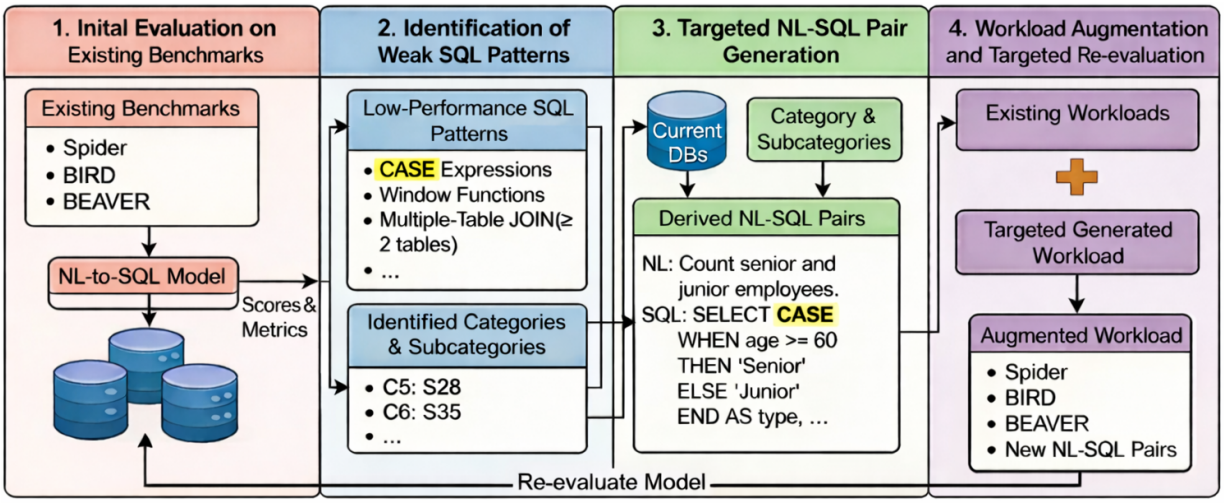


Figure 4.1: Four stages of targeted workload augmentation in SQLyzr. Stage 1: Initial evaluation of a text-to-SQL model using category- and subcategory-based metrics. Stage 2: Identification of under-performing SQL subcategories using the fine-grained scores. Stage 3: Generation of new question-SQL for the identified subcategories on existing databases. Stage 4: Augmentation of the workload with the newly synthesized pairs.

4.2.5 Report Generation

Based on the results produced in the assessment stage, SQLyzr generates evaluation reports at multiple levels of granularity. In addition to reporting aggregate scores over the entire workload, the framework provides detailed score breakdowns across query categories and subcategories. These fine-grained reports allow users to systematically analyze performance variations across different SQL patterns and identify constructs that are particularly challenging for a given text-to-SQL system.

Reports can be exported in tabular form to support further statistical analysis or rendered as visual charts to facilitate interpretation, comparison between systems, and presentation of results.

4.2.6 Workload Augmentation

When a text-to-SQL system consistently underperforms on specific categories or subcategories, augmenting the workload with additional data points from those patterns can target

the system's weaknesses and enable iterative refinement of the system to improve its correctness. To support this process, SQLyzr provides a workload augmentation module that synthesizes new question-SQL pairs targeting identified weaknesses.

The primary motivation for adopting workload augmentation in the framework is to customize the workload for a specific system. A benchmark with a fixed category distribution may not provide informative results for all systems. For instance, if most of the queries fall into categories that the system can reliably handle, the benchmark may fail to expose the system's limitations. Customizing the workload to increase the proportion of data points in categories where the system underperforms provides a more challenging and targeted workload for that specific system.

Moreover, augmenting the workload this way is not solely a matter of quantity. Increasing the proportion of data points in underperforming categories also introduces a broader range of queries within those categories. This diversity increases the likelihood of revealing the root cause of the system's weaknesses.

Figure 4.1 illustrates the workload augmentation procedure implemented in SQLyzr. The process begins by users specifying a threshold on the EA score, below which a subcategory is considered under-performing. After each evaluation cycle, SQLyzr identifies such subcategories and synthesizes additional question-SQL pairs for them using an LLM. To guide LLM generation, the framework provides example pairs from the training data points belonging to the identified subcategory. The LLM prompt used for this task is described in Appendix A.

The newly synthesized pairs are then added to the existing workload, enriching it with more challenging queries for the specific text-to-SQL system under evaluation. This iterative workflow allows users to re-evaluate the system, identify remaining weaknesses, and progressively refine the system.

4.3 Error Analysis Module

The error analysis module determines whether an incorrectly predicted query q' can be transformed into a query that produces the same execution result as the ground truth query q, using a predefined set of abstract error-fixing rules.

Such errors indicate that the predicted query is partially correct but contains specific issues whose correction would yield the correct result. Identifying these cases and reporting the corresponding errors provides developers with concrete guidance for improving the system.

Instead of attempting system improvements without insight and simply observing whether overall correctness increases, developers can focus on addressing the specific errors reported by SQLyzr. This targeted feedback enables a more efficient and systematic improvement process.

Conceptually, this mechanism can be seen as extending relaxed execution accuracy (Section 3.3.1) by permitting a more flexible notion of execution-result equivalence.

4.3.1 Error-Fixing Rules

The error-fixing rules are classified into two categories:

- **Pre-execution rules**, which transform the query before execution.
- **Post-execution rules**, which relax the equivalence conditions over execution results.

Pre-Execution Rules

Limit Clause Fixing Rule. This rule addresses discrepancies in the `LIMIT` clause between the predicted query and the ground truth query. For example, the predicted query may omit the `LIMIT` clause, use an incorrect limit value, or include a `LIMIT` clause when the ground truth query does not. In such cases, even if the overall structure of the query is correct, the query returns an incorrect number of rows, resulting in a zero EA score.

When this rule is triggered, SQLyzr suggests modifying the `LIMIT` value in the predicted query to match that of the ground truth query. After applying the correction, both queries contain the same `LIMIT` clause and can be re-evaluated accordingly.

Figure 4.2 shows an example in which the predicted query uses `LIMIT 5` while the ground truth query specifies `LIMIT 10`. Applying this rule replaces 5 with 10 in the predicted `LIMIT` clause.

Post-Execution Rules

Missing Columns Fixing Rule. This rule applies to cases where the predicted query omits one or more columns that are present in the ground truth query. Such omissions typically occur in the `SELECT` clause and lead to an EA score of zero, even when the remaining parts of the query are correct.

```sql
-- Gold
SELECT name FROM student LIMIT 10

-- Prediction
SELECT name FROM student LIMIT 5
```

Figure 4.2: An example of an incorrect `LIMIT` clause

```sql
-- Gold
SELECT name, age FROM student

-- Prediction
SELECT name FROM student
```

Figure 4.3: An example of a missing column in the `SELECT` clause

When this pattern is detected, SQLyzr suggests adding the missing columns from the ground truth query to the predicted query so that both queries project the same columns in the `SELECT` clause. In the example shown in Figure 4.3, the predicted query does not include the `age` column. By adding the missing column into the `SELECT` clause, the predicted query achieves an EA score of one.

Extra Columns Removal Rule. This rule handles cases where the predicted query includes additional columns in the `SELECT` clause that are not present in the ground truth query. Figure 4.4 shows an example in which the predicted query incorrectly includes the column `id`. Removing `id` from the `SELECT` clause results in matching the execution result of the predicted query with the ground truth query.

Extra Rows Removal Rule. This rule targets predicted queries that return extra rows compared to the execution result of the ground truth query, typically due to missing

```sql
-- Gold
SELECT name FROM student

-- Prediction
SELECT name, id FROM student
```

Figure 4.4: An example of an extra column in the `SELECT` clause

```sql
-- Gold
SELECT name FROM student WHERE age > 10

-- Prediction
SELECT name FROM student
```

Figure 4.5: An example of extra rows caused by a missing filtering condition

```sql
-- Gold
SELECT name, age FROM student

-- Prediction
SELECT age, name FROM student
```

Figure 4.6: An example of mismatched column ordering in the `SELECT` clause

filtering conditions. In such cases, SQLyzr suggests that the query should be modified to remove extra rows in the result set.

Figure 4.5 presents an example where the ground truth query includes a `WHERE` clause, while the predicted query omits it and therefore returns extra rows. Without the `WHERE` clause, the result of the predicted query includes extra rows that are not present in the ground truth result.

Column Order Fixing Rule. This rule addresses cases where the predicted query produces the same set of columns, but with a different ordering compared to the ground truth query. SQLyzr suggest using the same order of columns as the ground truth query to fix the error. Figure 4.6 shows an example of a mismatch between the ordering of columns in the `SELECT` clause.

Row Order Fixing Rule. This rule is applied when the predicted query produces the same set of rows as the ground truth query, but with a different ordering. When this error is detected, SQLyzr suggests modifying the query's `ORDER BY` clause to produce the same row ordering as the ground truth query. Figure 4.7 shows an example in which the queries differ only in the ordering of rows.

The textual feedback provided to users for each of the mentioned error-fixing rules is included in Appendix B.

```
-- Gold
SELECT name FROM student ORDER BY age ASC

-- Prediction
SELECT name FROM student ORDER BY age DESC
```

Figure 4.7: An example of a mismatched row ordering

4.3.2 Result Equivalence under Fixing Rules

In this section, we provide a formal formulation of execution result equivalence, considering the set of error-fixing rules defined in the previous section.

We follow the notation introduced in Section 3.3.1, representing each row r as a sequence of values $(v_1, \ldots, v_m)$ and a result R as a sequence of rows $(r_1, \ldots, r_n)$. We use $[n]$ to denote an index set $\{1, \ldots, n\}$. Let $\mathcal{G}$ denote the set of post-execution rules defined above.

Let $\phi : [n] \to [m]$ be a partial function between two index sets. We define the following properties for a function ϕ:

1. **Total:** $\forall i \in \text{dom}(\phi), \exists j \in \text{codom}(\phi)$ such that $\phi(i) = j$.
2. **Surjective:** $\forall j \in \text{codom}(\phi), \exists i \in \text{dom}(\phi)$ such that $\phi(i) = j$.
3. **Injective:** $\forall i_1, i_2 \in \text{dom}(\phi), \phi(i_1) = \phi(i_2) \implies i_1 = i_2$.
4. **Monotonic:** $\forall i_1, i_2 \in \text{dom}(\phi), i_1 \leq i_2 \implies \phi(i_1) \leq \phi(i_2)$.

Definition 7 (Row Matching $\simeq_\phi$) *Let r and r' be two rows and let $\phi : [|r|] \to [|r'|]$ be a partial injective function mapping indices of columns in r to indices in r'. We define matching between the rows as follows:*

$$r \simeq_\phi r' \iff \forall i \in \text{dom}(\phi), v_i = v'_{\phi(i)}$$

Definition 8 (Result Equivalence $\cong_g$) *Let R and R' be two results and let $g \subseteq \mathcal{G}$ be a subset of post-execution rules defined in the previous section. We lift row matching $\simeq$ to the result level with respect to the set of rules g as follows:*

$$R \cong_g R' \iff \exists \eta,\, P_g(\eta) \wedge \exists \phi,\, p_g(\phi),\, \forall i \in \mathrm{dom}(\phi), r_i \simeq_\phi r'_{\eta(i)}$$

Here:

- $\phi : [|r|] \to [|r'|]$ *is a partial injective mapping of row-level indices,*
- $\eta : [|R|] \to [|R'|]$ *is a partial injective mapping of result-level indices,*
- P_g *and* p_g *are predicates determined by the rule set* g *as defined below.*

Definition 9 (Row-Level Predicate $p_g(\phi)$) *Assuming* ϕ *is a row mapping,* p_g *is a predicate under the set of rules* g *defined as follows:*

$$\begin{aligned} p_g(\phi) \iff & (\textit{Extra Columns Removal} \in g \Rightarrow \phi \textit{ is total}) \\ & \wedge (\textit{Missing Columns Fixing} \in g \Rightarrow \phi \textit{ is surjective}) \\ & \wedge (\textit{Column Order Fixing} \notin g \Rightarrow \phi \textit{ is monotonic}) \end{aligned}$$

Definition 10 (Result-Level Predicate $P_g(\phi)$) *Let* ϕ *be a result mapping.* P_g *is a predicate under the set of rules* g *defined as follows:*

$$\begin{aligned} P_g(\phi) \iff & (\textit{Extra Rows Removal} \in g \Rightarrow \phi \textit{ is total}) \\ & \wedge (\textit{Row Order Fixing} \notin g \Rightarrow \phi \textit{ is monotonic}) \end{aligned}$$

Definition 11 (Query Matching) *Let* q *and* q' *denote the ground truth and predicted queries, and* g *be a set of rules. We define the matching between the queries under the set of rules* g *as follows:*

$$match(q, q', g) = \mathbb{I}[R \cong_g R']$$

where $R = exec(q, d)$ *and* $R' = exec(q', d)$.

4.3.3 Finding a Minimal Set of Error-Fixing Rules

The proposed error-fixing rules provide abstract suggestions for correcting predicted queries. However, identifying the concrete, query-specific modification that resolves the errors is non-trivial. This is because each abstract rule may correspond to multiple possible concrete fixes. For example, removing extra rows may require modifying predicates, adjusting an arithmetic constraint, or eliminating an unnecessary join. Therefore, directly enumerating all possible concrete modifications would create a large search space.

Figure 4.8 shows an example in which both predicted queries produce extra rows, although the underlying causes differ substantially. The first prediction involves an incorrect comparison operator, whereas the second one introduces a redundant join. Both errors fall under the same abstract rule, extra rows removal, yet require different concrete corrections.

SQL Queries

```sql
-- Ground Truth
SELECT name FROM student WHERE gpa > 10



-- Prediction #1
SELECT name FROM student WHERE gpa >= 10




-- Prediction #2
SELECT s.name FROM student s
JOIN enrollment e ON s.id = e.student_id
WHERE s.gpa > 10;
```

Execution Results

```
-- Ground Truth
+-------+
| Alice |
+-------+

-- Prediction #1
+-------+
| Alice |
| Bob   |
+-------+

-- Prediction #2
+-------+
| Alice |
| Alice |
+-------+
```

Figure 4.8: Example of minor errors in the predicted queries that yield extra rows in the result.

We formulate error-fix identification as a two-stage procedure. In the first stage, the framework identifies minimal subsets of abstract error-fixing rules that could fix the discrepancy between the predicted query q' and the ground truth query q. In the second stage, one would search within the scope of the identified rule sets to synthesize a concrete modification that, when applied to q', yields the same execution result as q. The first stage reduces the space of candidate fixes and thereby enables a more efficient search for concrete modifications.

In this work, we implement the first stage, leaving the second stage for future work.

Algorithm 2 shows the procedure for identifying the minimal set of error-fixing rules that can match the predicted and ground truth queries according to the relation in Definition 11. Note that post-execution rules do not modify the query itself. Instead, they relax the equivalence conditions over execution results. Line 7 performs the equivalence checking under a specific set of rules g.

The algorithm enumerates all valid subsets of rules in increasing order of cardinality to identify a minimal set of rules that fix the predicted query. Subsets containing incompatible rules (defined by I) are excluded from the search. For example, among the rules defined in Section 4.3.1, the extra columns removal and the missing columns rules are considered incompatible. Next, for each set of rules g, the algorithm first applies the pre-execution transformations to the queries (line 6) and then checks the matching of execution results with respect to the set of post-execution rules g_{post}.

Algorithm 2 Find a Minimal Fixing Rule Set

1: **procedure** FINDFIXINGRULES($q, q', \mathcal{G}, \mathcal{I}$) ▷ q: Gold Query, q': Predicted Query, $\mathcal{G}$: Set of All Rules, I: Incompatible Rules
2: **for** $k \leftarrow 1$ to $|\mathcal{G}|$ **do**
3: $G \leftarrow \{g \subseteq \mathcal{G} \mid |g| = k \wedge \forall i \in I,\, i \not\subseteq g\}$ ▷ All subsets of size k
4: **for** $g \in G$ **do**
5: $g_{pre}, g_{post} \leftarrow g$
6: $q, q' \leftarrow transform(q, q', g_{pre})$ ▷ Transform queries using the pre-execution rules
7: **if** $match(q, q', g_{post})$ **then** ▷ Check if EA = 1 if we apply fixes from groups in g
8: **return** g
9: **end if**
10: **end for**
11: **end for**
12: **end procedure**

4.4 Framework Configurability

The SQLyzr framework exposes a set of configuration options that allow users to customize evaluation runs according to their experimental objectives. In this section, we briefly describe the available configuration parameters.

Number of Iterations. SQLyzr allows users to specify the number of query prediction iterations per evaluation run. Since LLM-based text-to-SQL systems exhibit stochastic behaviour, running multiple iterations and reporting the mean score across runs provides statistically more stable and reliable evaluation results.

Text-to-SQL Systems. Users can choose one or more text-to-SQL systems for evaluation. When multiple text-to-SQL systems are configured, SQLyzr can produce comparative evaluation reports of these systems, including tabular summaries and visualizations.

Workloads. Users may select predefined workloads provided by SQLyzr or compose custom workloads by combining subsets of existing workloads.

Error Threshold for Workload Augmentation. For workload augmentation, users can specify a threshold on the EA score below which a subcategory is considered underperforming. SQLyzr then automatically synthesizes new question-SQL pairs for those subcategories whose performance falls below the specified threshold, enabling targeted refinement of the workload.

Due to the modular design of the framework, users can also modify or extend other components, such as category assignment and evaluation metrics, based on their requirements.

4.5 System Optimizations

To ensure scalability and cost-effectiveness, several optimizations are implemented in SQLyzr.

Parallelization. SQLyzr leverages multi-core CPU architectures to parallelize independent tasks within the evaluation pipeline. Pre-processing steps (performed by DAIL-SQL [24]), score computation, and error analysis are examples of CPU-bound tasks that benefit from parallel execution. This significantly reduces evaluation runtime, particularly for large workloads or multi-iteration evaluations.

Asynchronous Calls. The evaluation pipeline heavily relies on I/O-bound operations, such as HTTP requests to LLM APIs. To maximize throughput and resource utilization, SQLyzr employs asynchronous execution for these operations. The degree of concurrency for asynchronous calls can be configured based on user requirements.

In addition, SQLyzr incorporates a local rate limiter to constrain the number of in-flight tokens and requests, preventing violations of OpenAI rate limits. The rate limiter can be configured by specifying the OpenAI usage tier, which determines the allowable limits on

concurrent tokens and requests. This design removes the need for users to manually handle errors caused by rate limiting.

Caching. Repeated execution of identical database queries can introduce unnecessary computational overhead, particularly during error analysis, where the same query might be executed multiple times. To mitigate this, SQLyzr incorporates a caching mechanism for query execution that stores results on disk or in memory. This significantly reduces redundant computations and improves overall evaluation efficiency.

4.6 Batch API Calls

Several LLM providers, including OpenAI, offer batch APIs that process requests in bulk and provide a lower cost per token compared to synchronous APIs.

However, existing libraries for interacting with LLM APIs are typically designed around synchronous, single-request APIs. Therefore, adopting batch APIs typically requires substantial modifications to application code, since the programming model differs from that of standard APIs.

To address this limitation, SQLyzr provides a unified LLM client that abstracts away the differences between synchronous and batch APIs. The client exposes a single interface for LLM interactions, regardless of the underlying API mode. As a result, users can switch between synchronous and batch execution through a configuration option at runtime, without modifying application-level code.

The synchronous mode is suitable for smaller workloads where immediate responses are required, whereas batch mode is more appropriate for larger workloads, where it can significantly reduce the overall evaluation cost.

4.7 Database Systems Integration

SQLyzr performs all database interactions through a dedicated abstraction layer that provides a uniform interface for operations such as query execution, schema retrieval, and execution-time measurement, independent of the underlying RDBMS.

Currently, SQLyzr supports MySQL and SQLite. However, the framework is designed to be extensible and users can implement additional database drivers to support other RDBMSs without modifying the core evaluation pipeline.

4.8 Graphical User Interface (GUI)

To improve usability and accessibility, SQLyzr provides a graphical user interface (GUI) that allows users to configure, execute, and analyze benchmarking runs without directly interacting with configuration files or command-line interfaces.

The SQLyzr GUI supports workload selection, text-to-SQL system configuration, parameter tuning, execution monitoring, live logging, and visualization of evaluation results.

Figure 4.9 shows the main configuration panel of the GUI. This panel allows users to configure benchmark options such as workload selection, text-to-SQL system, number of iterations, temperature values, and the workload augmentation threshold.

We also provide a live deployment of SQLyzr that allows users to interact with the framework through the GUI[1].

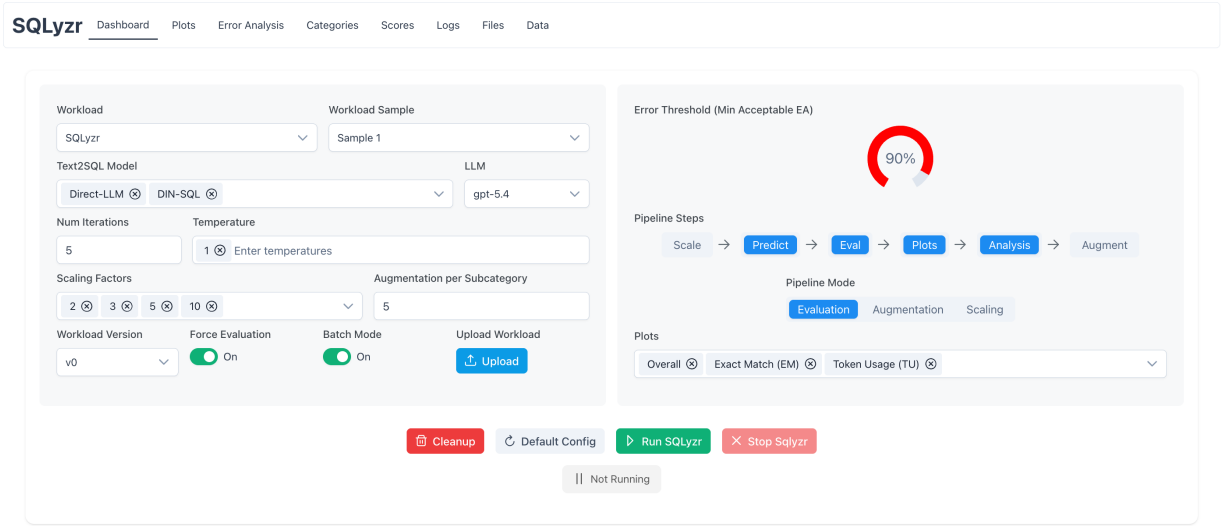


Figure 4.9: Configuration panel of the SQLyzr GUI

4.9 Artifact Release and Deployment

SQLyzr is released as a standalone Docker image to facilitate reproducibility and ease of deployment. New system updates are automatically tested, packaged, and released

[1] https://demo.sqlyzr.site/

through a continuous integration (CI) pipeline. In addition, SQLyzr supports automated deployment through a continuous deployment (CD) pipeline. These approaches ensure reproducibility and enable easy adaptation and deployment of the framework.

The SQLyzr benchmark and framework are publicly available on GitHub[2].

[2] https://github.com/sepideh-abedini/SQLyzr

Chapter 5

Experiments

This chapter presents the empirical evaluation of two state-of-the-art text-to-SQL systems, DIN-SQL [62] and DAIL-SQL [24], using the SQLyzr benchmarking framework. Our goal is to demonstrate how SQLyzr's fine-grained taxonomy and extended evaluation metrics enable a more comprehensive understanding of system behaviour beyond traditional benchmarks. In addition, we present experimental results on workload alignment with SQLShare and database scaling, analyzing their impact on evaluation results.

Key Findings. Our experimental results highlight several notable observations:

- **DAIL-SQL outperforms DIN-SQL:** Evaluation of these systems using existing benchmarks indicates that DAIL-SQL marginally outperforms DIN-SQL. However, a more comprehensive assessment enabled by SQLyzr reveals that DAIL-SQL consistently outperforms DIN-SQL across additional aspects of text-to-SQL translation as well, including query complexity, generation cost, and execution efficiency.
- **Macro-average scores are more reliable:** Our evaluation results demonstrate that macro-averaged scores across categories provide a more reliable assessment compared to micro-averaging, which can be biased toward categories that contain a larger number of queries.
- **DIN-SQL performs better on queries in C_1:** Fine-grained correctness analysis shows that DIN-SQL achieves higher accuracy than DAIL-SQL on the simplest category (C_1), while falling behind in more complex categories.

- **Certain query types are challenging for both systems:** Subcategory-level analysis reveals specific query types that are particularly challenging for both systems, including arithmetic expressions and advanced constructs such as window functions.

The remainder of this chapter is organized as follows. Section 5.1 describes the experimental setup, including the hardware specification, the evaluated text-to-SQL systems, and the configuration options of SQLyzr used in these experiments. Section 5.2 presents the overall evaluation results. Section 5.3 provides fine-grained analysis at the level of categories and subcategories. Section 5.4 reports the results of workload alignment with SQLShare. Section 5.5 presents the experimental results under dataset scaling. Finally, Section 5.6 summarizes the key findings from the experiments.

5.1 Experimental Setup

In this section, we describe the experimental setup, including the hardware specifications used for the evaluation, the evaluated text-to-SQL systems, and the configuration details of the SQLyzr framework used in our experiments.

5.1.1 Hardware Specification

Table 5.1 summarizes the hardware specifications and software environment used in our experiments. We used a native Python installation to run the text-to-SQL systems and perform the evaluation. SQL queries were executed using a native installation of SQLite and a Docker container of the MySQL DBMS.

Table 5.1: Experimental Environment and Hardware Specifications

Component	**Specification**
CPU	2 × Intel Xeon Platinum 8380 (40 cores each)
RAM	1 TB
GPU	NVIDIA A100 (80 GB VRAM)
Operating System	Ubuntu 22.04
Python Version	3.11
MySQL Version	8.0 (Official Docker image)

5.1.2 Text-to-SQL Systems

We evaluated two of the highest-performing text-to-SQL systems with publicly available source code. These systems were selected according to their execution accuracy (EA) ranking on the Spider benchmark leaderboard[1].

DIN-SQL

DIN-SQL [62] uses few-shot LLM prompting [7] to generate SQL queries from natural language questions. Its main novelty lies in decomposing SQL query generation into smaller sub-tasks, described below.

Schema Linking. This step aims to identify the condition values and references to database columns within the input question and map them to the corresponding elements in the underlying database schema. To perform this step, DIN-SQL prompts the LLM using the Chain-of-Thought (CoT) technique [75]. The prompt for this sub-task includes a fixed set of examples extracted once from the training data and reused for all questions.

Query Classification and Decomposition. In this task, each question is classified into one of three classes: easy, non-nested complex, and nested complex. The first class includes simple questions that do not require joins or nested queries. The second class includes questions that require joins but no sub-queries. The third class includes queries that require both joins and nested sub-queries. Similar to schema linking, this task is also performed using few-shot LLM prompting with a fixed set of demonstration examples provided in the LLM prompt.

SQL Generation. In the subsequent task, the LLM is prompted to generate the final SQL query for a given question using the schema links and the classification results obtained in the previous steps. A separate prompt and set of examples is designed for each query class, and the classification determines which prompt is used.

Self-Correction. The final task aims to correct potential issues in the generated SQL, such as missing `DISTINCT` or `DESC` keywords. This step is also implemented via LLM prompting.

One notable limitation of DIN-SQL is its reliance on a fixed set of demonstration examples within its LLM prompts. As discussed later in this chapter, the fine-grained analysis enabled by SQLyzr helps reveal how limited demonstration diversity can negatively affect performance, particularly for more complex queries.

[1] https://yale-lily.github.io/spider

Based on the publicly available implementation of DIN-SQL[2], we implemented a version compatible with SQLyzr. This implementation enables the use of framework features such as parallelization and asynchronous calls to accelerate evaluation (Section 4.5), as well as batch execution to reduce evaluation costs (Section 4.6).

DAIL-SQL

DAIL-SQL [24] focuses on improving the demonstration examples provided to LLMs for SQL generation by enhancing both example selection and example organization.

Example Selection. In contrast to DIN-SQL, which uses a fixed set of demonstration examples in its LLM prompts independent of the input question, DAIL-SQL dynamically selects the most relevant examples based on their similarity to the given natural language question.

To identify top candidates, it first masks the schema-specific tokens in both the input question and all questions in the examples pool. It then generates a *pre-prediction* SQL for the input question, which is an approximate query generated using a smaller, local model. The example pool is then filtered to retain only those examples that meet a specific similarity threshold relative to the pre-predicted SQL. Within this filtered subset, masked questions are ranked based on their semantic similarity to the masked input question, and the top k examples are selected. Semantic similarity between masked questions is computed using the Euclidean distance between their embeddings, which are generated by a pre-trained transformer model `all-mpnet-base-v2` [68]. The pre-prediction SQL is generated using the Graphix [49] model. Similarity between SQL skeletons is calculated using Jaccard similarity over the remaining SQL keywords after removing all schema-specific references and values.

Example Organization. Example organization refers to how demonstration examples are encoded within the prompt context. The format determines which components of each example are included—namely, the natural language question, the ground truth SQL query, and the database schema. Common strategies include the *full* format, where all three components are included, and the *SQL-only* format, where only SQL queries are provided. DAIL-SQL adopts a third approach by including the natural language question and the SQL query while excluding the database schema. This strategy allows DAIL-SQL to achieve strong execution accuracy while reducing token consumption.

This example selection and organization strategy gives DAIL-SQL a key advantage

[2] https://github.com/MohammadrezaPourreza/Few-shot-NL2SQL-with-prompting

over DIN-SQL. As demonstrated in our analysis, dynamic example selection is particularly effective for improving performance on more complex query categories.

Similar to DIN-SQL, we implemented a version of DAIL-SQL compatible with SQLyzr based on the official implementation[3].

5.1.3 Workload and Dataset

Details of the workload and dataset used in the benchmark are provided in Section 3.1. Here, we present a brief overview.

Both the workload and dataset of SQLyzr are constructed from the Spider, BIRD, and BEAVER benchmarks. The workload consists of 20,979 question-SQL pairs in total, with 18,393 allocated for testing. These pairs span 286 databases and 2,072 tables, with the populated databases containing over 66,000,000 rows in total. The workload is classified into 6 categories and 36 subcategories following the taxonomy defined in Section 3.1.2.

Aligning Workload with SQLShare. As discussed in Section 3.1.3, we align the SQLyzr workload with the SQLShare corpus [37]. The goal is to ensure that the SQLyzr workload reflects real-world usage patterns. We separately report and analyze the experimental results of using the aligned workload in Section 5.4.

Dataset Scaling. As discussed in Section 3.2, we use the Synthetic Data Vault (SDV) framework [57] to synthesize additional data for the databases, enabling experiments that simulate large-scale database environments. The results of dataset scaling experiments are presented in Section 5.5.

5.1.4 SQLyzr Configuration

The SQLyzr framework provides a wide range of configuration options. The settings used in our experimental evaluation are summarized below.

Iterations. To mitigate the stochastic behaviour of LLM-based SQL generation, SQLyzr supports multiple execution iterations. In our experiments, we configured the framework to perform three independent iterations to improve the robustness and stability of the reported results.

LLM Temperature. Using SQLyzr's hyperparameter tuning, we evaluated the sensitivity of the text-to-SQL systems to the LLM temperature parameter. We experimented with

[3] https://github.com/BeachWang/DAIL-SQL

temperature values of 0.2, 0.5, and 0.8. However, our results indicate that the impact of temperature on correctness and efficiency metrics is negligible (less than 1%). Therefore, we report all results using the fixed temperature of 0.2.

Language Model. GPT-4o-mini was used as the underlying LLM model for both DIN-SQL and DAIL-SQL.

Batch Processing. All LLM calls were made through OpenAI's batch API using SQLyzr's batch mode (see Section 4.6 for details), reducing experimental costs by 50%.

Parameters for ETC Calculation. As defined in Section 3.3, execution time consistency is satisfied when the ratio between the execution times of the predicted and ground truth queries is below the threshold α. The constant α is calculated according to Equation (3.3.2) with $K = 20$ and $p = 95$.

5.2 Evaluation Results

We begin by presenting the overall evaluation results, reporting the following metrics (see Section 3.3 for detailed definitions):

- **Execution Accuracy (EA)**: Measures whether the execution result of the predicted query matches that of the ground truth query.
- **Relaxed Execution Accuracy (REA)**: Measures a relaxed variant of EA using a less strict matching criterion.
- **Exact Match (EM)**: Measures component-wise equivalence between the predicted and the ground truth SQL.
- **Execution Time Consistency (ETC)**: Indicates whether the execution time of a correctly predicted query remains close to that of the ground truth SQL query.
- **Complexity Consistency (CC)**: Indicates whether the structural complexity of a correctly predicted query is no greater than that of the ground truth SQL query.
- **Token Usage (TU)**: Measures the number of tokens consumed by the LLM during text-to-SQL translation.

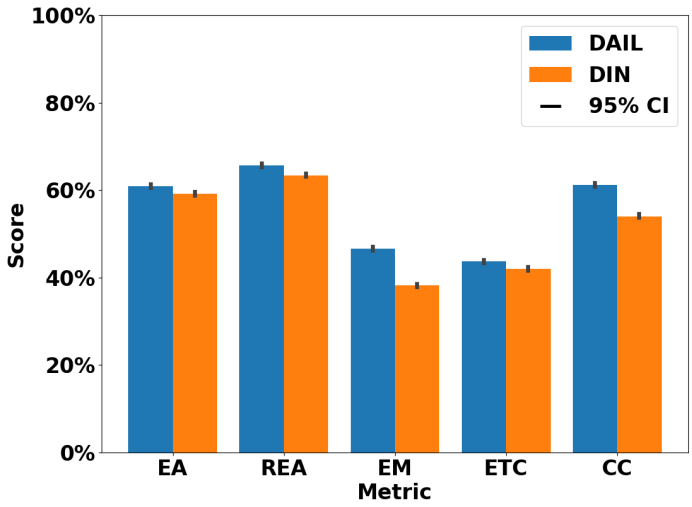


Figure 5.1: Overall micro-average evaluation scores of DIN-SQL and DAIL-SQL. Higher values indicate better scores.

Model	EA	REA	EM	ETC	CC
DAIL-SQL	60.88%	65.66%	46.58%	43.66%	61.20%
DIN-SQL	59.15%	63.42%	38.18%	42.06%	54.03%

Table 5.2: Overall evaluation scores of DIN-SQL and DAIL-SQL.

Figure 5.1 presents the micro-average scores, as defined in Equation (3.1), for DIN-SQL and DAIL-SQL, with 95% confidence intervals indicated by error bars. Table 5.2 presents the corresponding numerical values.

While existing benchmarks report that DAIL-SQL only marginally outperforms DIN-SQL in terms of correctness, the more comprehensive evaluation enabled by SQLyzr reveals that DAIL-SQL also outperforms DIN-SQL in other aspects of query generation. In addition to achieving higher EA and REA scores, DAIL-SQL demonstrates better performance on efficiency-related metrics such as ETC and CC. Furthermore, as discussed later in this chapter, DAIL-SQL consumes significantly fewer tokens on average, making it more cost-effective than DIN-SQL.

Taken together, these results indicate that DAIL-SQL provides a more robust, efficient, and cost-effective solution overall.

5.3 Fine-Grained Analysis

In this section, we analyze the experimental results through category-level evaluation scores.

5.3.1 Evaluation Metrics at Category-Level

Correctness of Queries. We begin the fine-grained analysis by comparing the systems with respect to correctness. Figure 5.2 presents the EA and REA scores of DIN-SQL and DAIL-SQL across all categories. The overall score corresponds to the macro-average, as defined in Equation (3.2), computed across the categories. As shown in the figure, DAIL-SQL consistently outperforms DIN-SQL across most categories.

Notably, DIN-SQL performs better in the simplest category, C_1, and achieves comparable results in C_2 and C_3, where the performance gap with DAIL-SQL is negligible ($\leq 1.5\%$). However, starting from C_4, a clear gap emerges between the two systems. This observation suggests that DIN-SQL struggles as query complexity increases. We hypothesize that this trend is primarily due to a key limitation of DIN-SQL: its reliance on a fixed set of demonstration examples for in-context learning regardless of the input question. However, as query complexity increases, the diversity of possible SQL patterns within a category also grows. A fixed demonstration set, therefore, cannot adequately cover this expanded space of patterns, leading to lower-quality demonstration examples for the LLM. In contrast, DAIL-SQL dynamically selects examples specifically chosen for each input question, which results in higher-quality and more relevant examples. This difference in example selection mechanisms helps explain why DIN-SQL performs competitively in simpler categories but falls behind in more complex categories.

Figure 5.3 presents the exact match (EM) scores per category, with shaded bars showing the corresponding REA scores. Consistent with the EA and REA trends, DAIL-SQL outperforms DIN-SQL across all categories in terms of EM. In this case, DIN-SQL no longer maintains an advantage in C_1 over DAIL-SQL. Moreover, the gap between the two systems is more pronounced for EM than for EA and REA, suggesting that DAIL-SQL produces queries that are more closely aligned with the human-annotated ground truth.

The figure also shows that the gap between REA and EM widens as query complexity increases. This observation highlights the brittleness of EM, particularly for complex categories. This can be attributed to the fact that, as queries become more complex, the space of possible solutions expands, increasing the likelihood that generated queries differ structurally from the ground truth SQL query and resulting in lower EM scores.

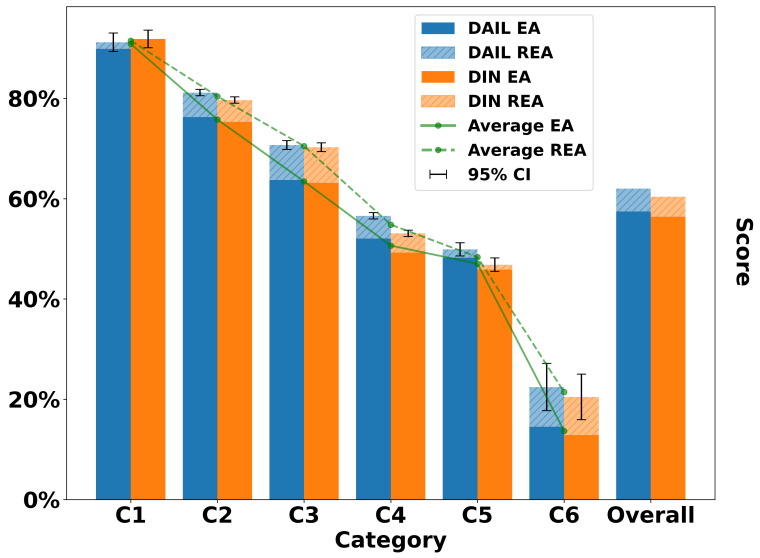


Figure 5.2: EA and REA scores per category

Correctness and Complexity Correlation. As shown in Figures 5.2 and 5.3, all correctness scores (EA, REA, and EM) exhibit a clear negative correlation with category complexity. This monotonic decreasing trend across categories validates the hierarchical structure of our taxonomy, where categories are ordered by increasing complexity.

Complexity of Queries. Figure 5.4 presents category-level complexity consistency (CC) scores. A higher CC score indicates that the system avoids introducing unnecessary structural complexity to the predicted query compared to the ground truth query.

DAIL-SQL consistently achieves higher CC scores than DIN-SQL across categories. Additionally, CC scores decline as query complexity increases. This trend is expected: as queries become more complex, the number of alternative queries grows, and systems are more likely to introduce redundant joins, nested subqueries, or unnecessary constructs.

Efficiency of Queries. Figure 5.5 compares category-level execution time consistency (ETC). Higher ETC scores indicate that a larger fraction of correctly predicted queries have execution times close to those of the ground truth queries.

As shown in the figure, DAIL-SQL outperforms DIN-SQL in most categories, with the exception of C_1 and C_5. However, the overall differences between the two systems remain negligible.

Another notable trend is the decreasing gap between ETC and its upper bound (REA)

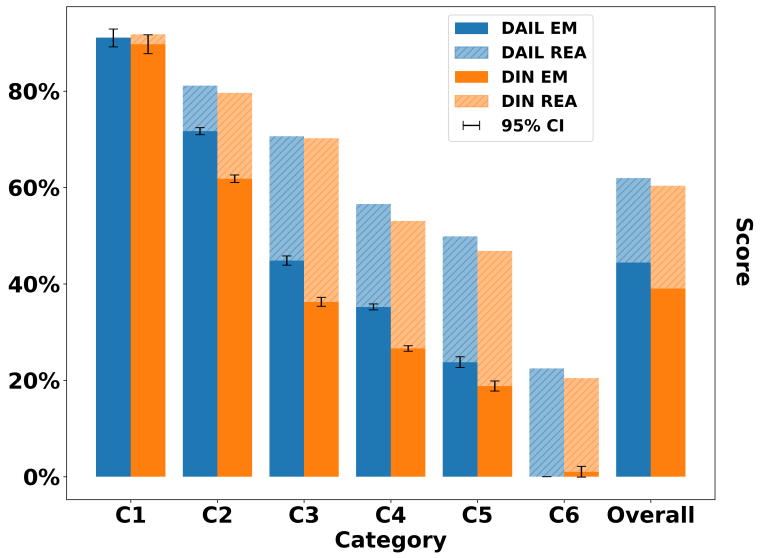


Figure 5.3: EM and REA scores per category

as category complexity increases. Since more complex queries generally incur higher execution times, there is greater tolerance for variations in the execution time of predicted queries. Consequently, small execution time differences are less likely to result in a zero ETC score.

Generation Cost of Queries. Figure 5.6 reports token consumption across categories. As expected, token usage increases with query complexity, as more complex questions require longer prompts and additional reasoning steps, leading in higher token consumption. Additionally, DAIL-SQL significantly outperforms DIN-SQL in terms of token efficiency. On average, DIN-SQL consumes approximately four times more tokens than DAIL-SQL. This is largely attributable to DIN-SQL's decomposition strategy, which involves multiple sequential LLM calls for schema linking, classification, SQL generation, and self-correction. However, in category C_6, DAIL-SQL consumes slightly more tokens than DIN-SQL. This may indicate inefficiencies in DAIL-SQL's prompt construction for highly complex queries, suggesting potential room for optimization.

5.3.2 Subcategory-Level Analysis

Figure 5.7 presents the correctness scores at the level of subcategories. The overall score corresponds to the macro-average across subcategories.

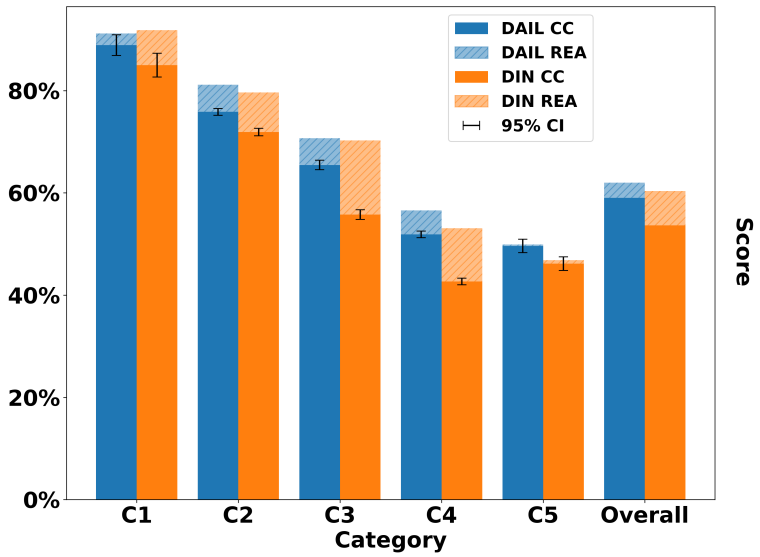


Figure 5.4: Complexity Consistency (CC) per category. Higher values indicate that the system performed better in keeping the complexity of the predicted queries close to ground truth queries.

Within each category, we expect the scores of its subcategories to be aligned with the corresponding category-level score. In this section, we identify subcategories that exhibit significant deviations from their parent category averages. This analysis allows us to identify specific query patterns that are particularly challenging for these systems.

Arithmetic Expressions (S_5). Subcategory S_5 corresponds to queries involving arithmetic expressions. Despite the relatively high mean REA for category C_2, both DIN-SQL and DAIL-SQL exhibit a noticeable drop in performance on S_5. This suggests that the additional reasoning required for arithmetic expressions makes these queries more error-prone, even when their structural complexity remains relatively low, as indicated by their classification in C_2. Subcategory S_5 corresponds to queries involving arithmetic expressions.

Null Expressions (S_{18}). Subcategory S_{18} also shows a substantial deviation from the overall score of category C_3. However, this subcategory contains only 10 question-SQL pairs, as indicated by the large confidence intervals.

Manual inspection of the predicted queries reveals that both systems generally handle `NULL` expressions correctly. The observed failures, however, are typically caused by issues not related to the `NULL` expressions, such as missing projection columns in the `SELECT`

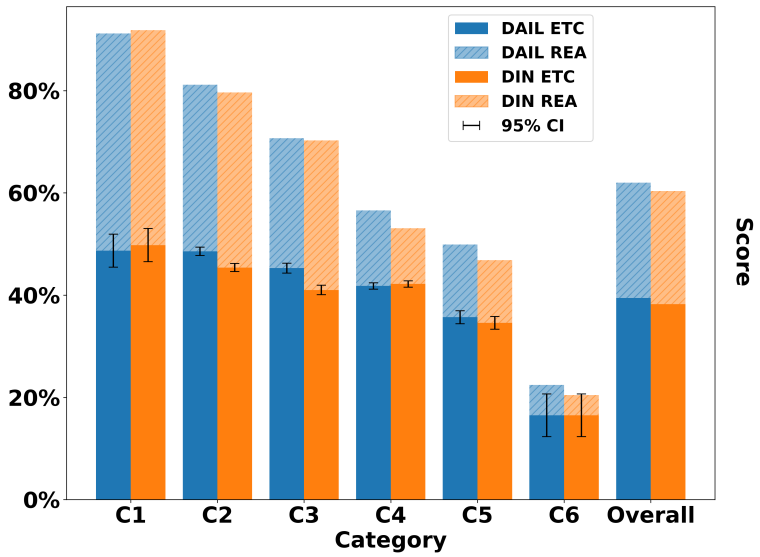


Figure 5.5: Execution Time Consistency (ETC) per category. Higher values indicate that the system performed better in keeping the execution efficiency of the predicted queries close to ground truth queries.

clause or incorrect filtering conditions. This suggests that the difficulty is not inherent in `NULL` expressions themselves. Given the small size of this subcategory, the deviation is therefore not sufficient to draw strong conclusions about a specific weakness in handling `NULL` expressions.

CTEs and Window/Ranking Functions (S_{35}, S_{36}). Subcategories corresponding to common table expressions (CTEs) and window or ranking functions consistently exhibit the lowest performance across both systems. In our evaluation, queries involving these constructs are rarely generated correctly.

A likely explanation is the lack of training data on such patterns. These subcategories use the most advanced constructs and are relatively rare. Therefore, language models are likely exposed less frequently to these advanced constructs.

5.4 Aligning Workload with SQLShare

In this section, we report evaluation results for a subset of the SQLyzr workload aligned with the category distribution of SQLShare. Figure 5.8 compares the percentage of data

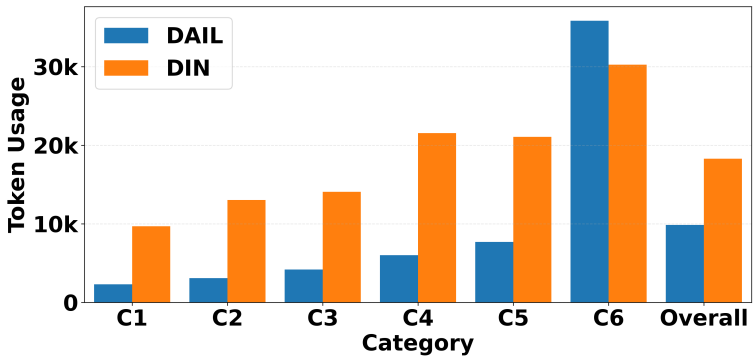


Figure 5.6: Token usage per category

points per category in SQLyzr and SQLShare. To align the SQLyzr workload with SQLShare, we adjusted the category distribution by down-sampling categories that were over-represented relative to SQLShare until the distributions matched. Table 5.3 shows the number and percentage of data points per category. The *Aligned SQLyzr* column shows the distribution of categories after alignment.

To increase the size of the aligned workload while preserving the target distribution, we repeated the alignment procedure multiple times using independently drawn random samples. The *Aligned SQLyzr*×5 column corresponds to the union of five independently aligned samples. This strategy allows us to use a larger portion of the original workload while maintaining the SQLShare distribution. In the remainder of this section, we report evaluation results using the *Aligned SQLyzr*×5 workload.

Category	SQLyzr		SQLShare		Aligned SQLyzr		Aligned SQLyzr× 5	
	#	%	#	%	#	%	#	%
C1	306	1.66%	1,644	15.05%	306	15.03%	1,530	15.03%
C2	5,018	27.28%	4,309	39.45%	803	39.44%	4,015	39.44%
C3	3,472	18.88%	2,319	21.23%	432	21.22%	2,160	21.22%
C4	7,667	41.68%	1,067	9.77%	199	9.77%	995	9.77%
C5	1,834	9.97%	1,148	10.51%	214	10.51%	1,070	10.51%
C6	96	0.52%	437	4.00%	82	4.03%	410	4.03%
Total	**18,393**		**10,924**		**2,036**		**10,180**	

Table 5.3: Category distribution for SQLyzr, SQLShare, and aligned variants of SQLyzr

5.4.1 Evaluation Results

Overall Scores. Figure 5.9 presents micro-average evaluation scores for the aligned workload compared with the original workload. As shown in the figure, all correctness and most efficiency-related metrics improve under the aligned workload. Specifically, EA increases by approximately 8%, REA by 7%, and EM by 10%. For efficiency-related metrics, CC increases by 8%, while ETC shows only a marginal change.

The main reason for this improvement is the difference in category distributions between SQLShare and the original SQLyzr workload. In SQLShare, 75.72% of data points belong to categories C_1, C_2, and C_3, whereas in the original SQLyzr workload, only 45.82% fall into these categories. Since C_1–C_3 correspond to lower-complexity queries, increasing their proportion naturally leads to higher overall scores for both systems.

Model	EA	REA	EM	ETC	CC
DAIL-SQL	68.08%	72.67%	57.26%	45.16%	68.70%
DIN-SQL	66.92%	70.79%	50.03%	42.38%	62.51%

Table 5.4: Overall evaluation scores of DIN-SQL and DAIL-SQL on the aligned workload

Micro- vs Macro-Average. Figure 5.10 presents the macro-average scores for the aligned workload compared with the original workload. In contrast to the micro-average results (Figure 5.9), the macro-average scores show only negligible differences between the aligned and original workloads.

This observation suggests that macro-averaging across categories provides a more robust aggregation strategy than micro-averaging, as it is less sensitive to the distribution of data points across categories.

Moreover, this result indicates that an aligned workload, when evaluated using macro-average scores, can reproduce the overall behaviour observed on a larger and differently distributed workload.

Correctness and Efficiency of Queries. Figures 5.11, 5.12 show the REA, EA, and EM scores per category for the aligned workload. Per-category trends remain consistent with those observed in the original workload.

In particular, the aligned results preserve the relative advantage of DAIL-SQL over DIN-SQL, as well as the increasing performance gap between the systems as query complexity grows. This consistency further validates the comparative conclusions drawn earlier.

Efficiency and Complexity of Queries. Figures 5.13, 5.14, and 5.15 present the complexity consistency, execution time consistency, and token usage results for the aligned

workload. Similar to correctness metrics, the trends in efficiency and complexity scores show no significant deviation from those observed in the original workload, indicating that the aligned workload can reliably reproduce the key results obtained with the larger workload.

5.5 Dataset Scaling

In this section, we present experimental results demonstrating SQLyzr's dataset scaling capability. The experiments are conducted on a subset of the SQLyzr workload, referred to as the *scaling workload*. These experiments serve as a proof-of-concept for evaluating text-to-SQL systems' behaviour under large-scale database sizes.

5.5.1 Construction of the Scaling Workload

Due to limitations of the data synthesis library used for scaling, we applied dataset scaling only to the Spider subset of the workload. We then constructed the scaling workload by aligning the remaining data points with SQLShare. Within the aligned workload a few databases could not be scaled by SDV due to the complexity of their schema. Therefore, we also excluded those databases and their corresponding data points from the scaling workload. Table 5.5 presents the category distributions for the original Spider subset, SQLShare, and the resulting scaling workload. The scaling workload contains 931 data points spanning 54 databases. Extending the scaling procedure to BIRD and BEAVER databases is left for future work.

To generate scaled databases, we used the Synthetic Data Vault (SDV) framework [57] to synthesize and insert additional rows into each database. The number of synthesized rows is controlled by a scaling factor parameter. In our experiments, we evaluated scaling factors of 10, 20, 50, 100, and 1000.

In the following, we report evaluation results for the scaling workload for each of the scaling factors.

5.5.2 Evaluation Results

Figure 5.16 presents the average execution time of the ground truth and predicted queries across scaling factors. As expected, execution time increases as the database size grows.

Category	Spider		SQLShare		Scaling workload	
	#	%	#	%	#	%
C1	304	3.32%	1,644	15.05%	143	15.36%
C2	3,907	42.71%	4,309	39.45%	375	40.28%
C3	2,790	30.50%	2,319	21.23%	202	21.70%
C4	1,652	18.06%	1,067	9.77%	93	09.99%
C5	456	4.99%	1,148	10.51%	100	10.74%
C6	38	0.42%	437	4.00%	18	1.93%*
Total	**9,147**		**10,924**		**931**	

* A few databases and corresponding data points from this category were excluded because they could not be scaled by SDV due to the complexity of their schema.

Table 5.5: Category distribution for the Spider subset of SQLyzr, SQLShare, and scaling workload (aligned with SQLShare).

The increase is particularly noticeable for queries involving joins and aggregations, which are more sensitive to dataset size.

Figure 5.17 compares overall evaluation scores for the scaling workload on the original databases and on databases scaled by a factor of 1000. Both EA and REA decrease under large-scale databases by approximately 5% for DAIL-SQL and 4% for DIN-SQL. This observation demonstrates that larger databases expose certain execution discrepancies that remain undetected at smaller scales. In particular, execution accuracy measured on small datasets may produce false positives when semantically incorrect queries coincidentally yield correct results. Increasing the database size reduces the likelihood of such accidental equivalence.

Furthermore, ETC decreases by approximately 10% for both systems under database scaling. This reduction is partly attributable to the current formulation of execution time consistency, which relies on a linear ratio between the execution times of predicted and ground truth queries. However, as the database size increases, execution time may grow non-linearly. For example, when queries involve joins, execution time grows polynomially. Under such conditions, a linear ratio may not adequately capture acceptable relative performance differences between predicted and ground truth queries. Consequently, exploring non-linear formulations of the execution time consistency metric is a promising direction for future work.

5.6 Summary of Findings

In this chapter, we evaluated two state-of-the-art text-to-SQL systems and demonstrated how SQLyzr provides insights beyond traditional correctness metrics. We further showed how fine-grained evaluation and analysis enable a deeper understanding of system behaviour and help identify specific system weaknesses.

We conclude this chapter with the following key findings from the experiments:

- While DAIL-SQL only marginally outperforms DIN-SQL in terms of execution accuracy, a comprehensive and fine-grained evaluation reveals that DAIL-SQL also generates more efficient queries and does so at a lower generation cost. Such a comprehensive evaluation provides a more reliable basis for selecting one system over another (Section 5.2).
- The macro-average of evaluation metrics across categories is a more robust aggregation strategy, as it shows consistent results across workloads with varying category distributions (Section 5.4).
- Combining macro-average aggregation with an aligned workload yields results that closely approximate those obtained on the original and larger workload. This observation indicates that a carefully designed, smaller workload can serve as an efficient and cost-effective alternative without compromising reliability (Section 5.4).
- Database scaling experiments reveal measurable drops in both correctness and efficiency metrics under large-scale settings. These results suggest that scaling not only enables realistic efficiency evaluation but also improves the reliability of approximate correctness measures such as execution accuracy by reducing the number of false positives (Section 5.5).
- Category-level analysis reveals that DIN-SQL achieves a higher correctness score in C_1, which contains the simplest forms of queries, but falls behind DAIL-SQL in more complex categories. This behaviour is likely attributable to DIN-SQL's reliance on a fixed set of demonstration examples for in-context learning (Section 5.3).
- Subcategory-level analysis shows that both systems struggle with specific query patterns, particularly arithmetic expressions and advanced constructs such as window functions. Identifying these patterns provides clear directions for improving the correctness of these systems (Section 5.3).

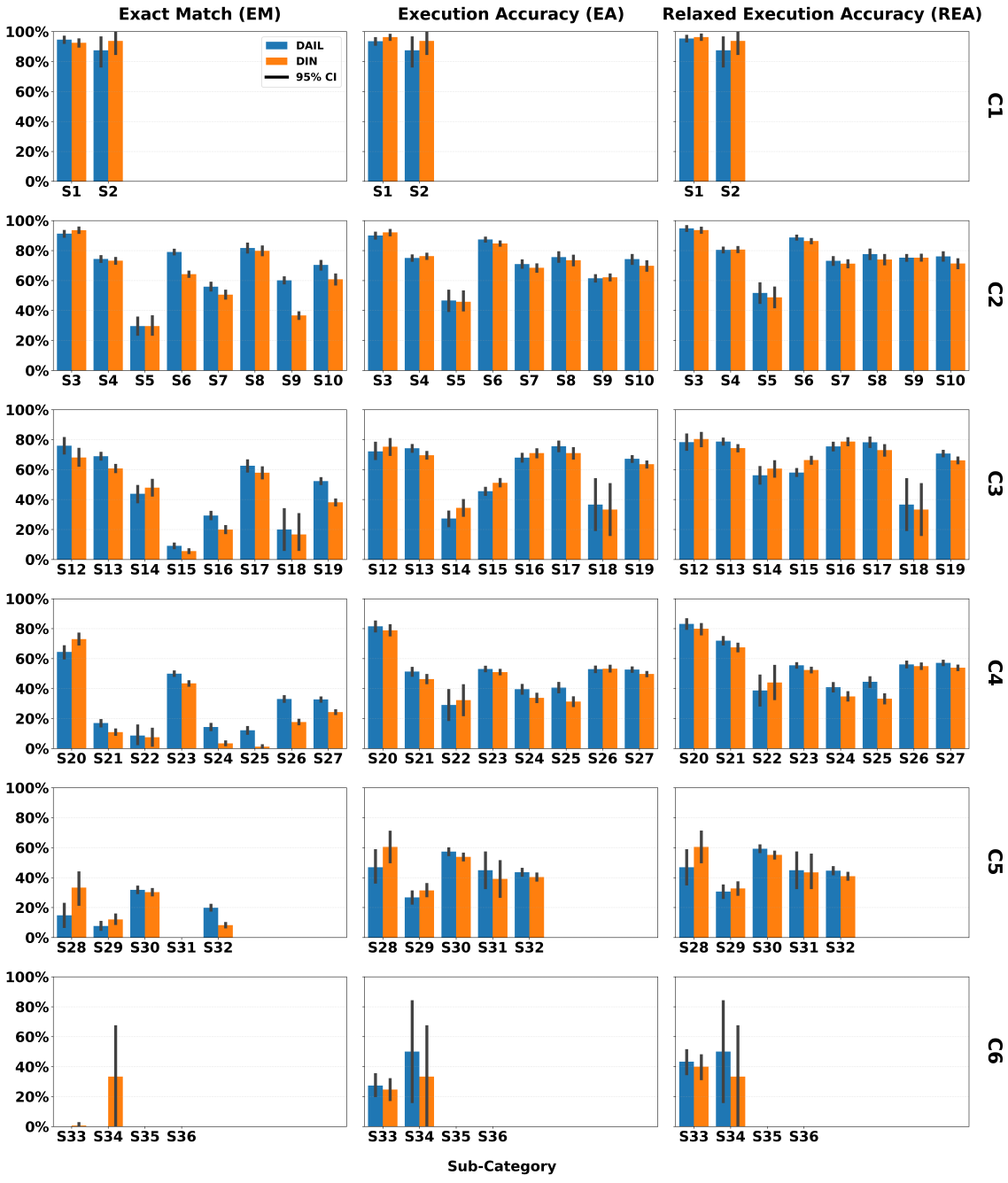


Figure 5.7: Correctness scores per subcategory

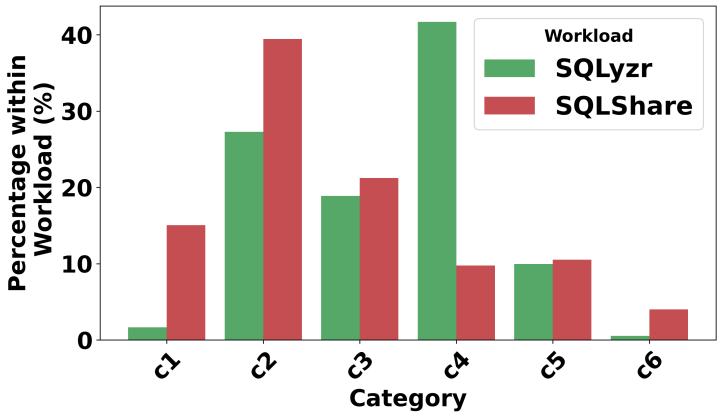


Figure 5.8: Category distribution comparison between SQLyzr and SQLShare

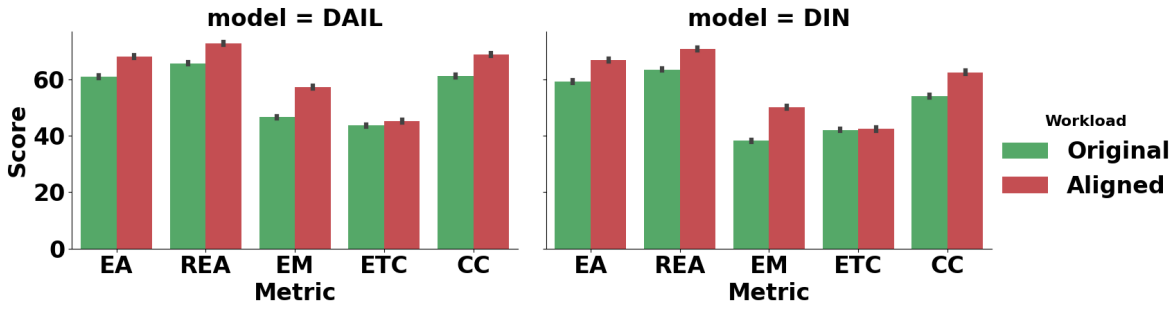


Figure 5.9: Micro-average evaluation scores for DIN-SQL and DAIL-SQL (aligned workload vs. original workload)

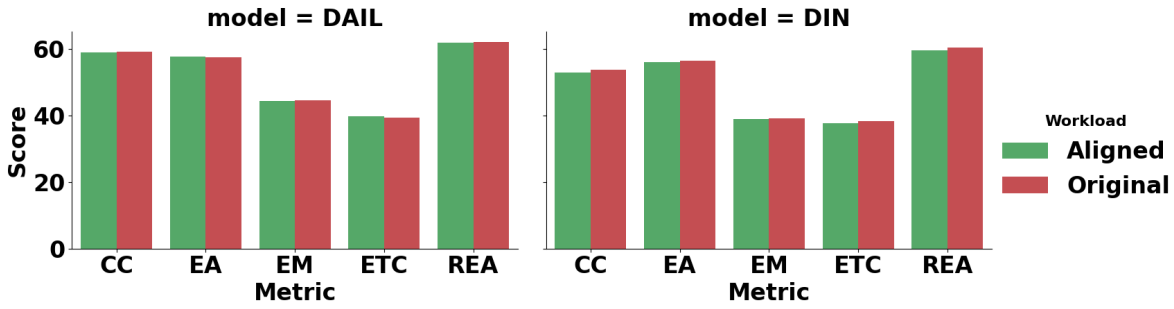


Figure 5.10: Macro-average evaluation scores for DIN-SQL and DAIL-SQL (aligned workload vs. original workload).

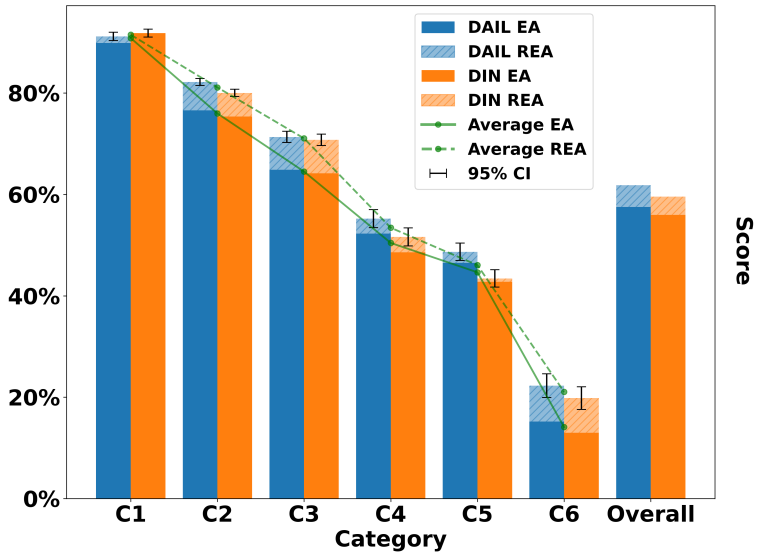


Figure 5.11: EA and REA scores per category (aligned workload).

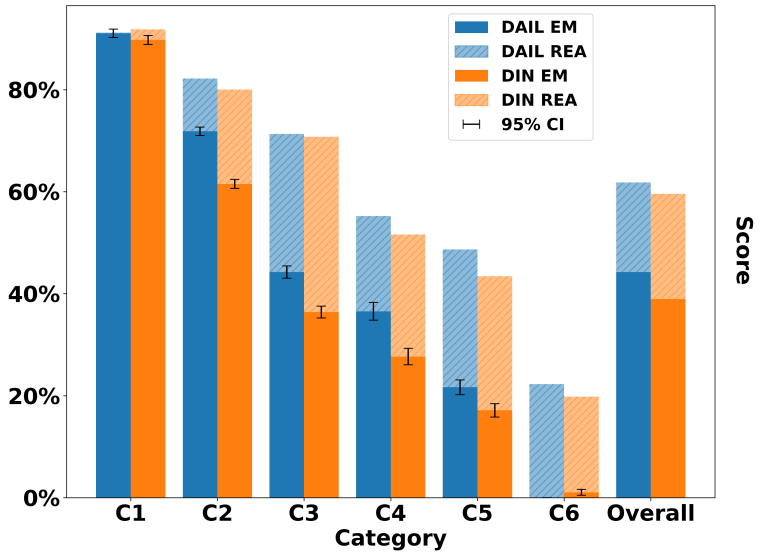


Figure 5.12: Exact Match per category (aligned workload).

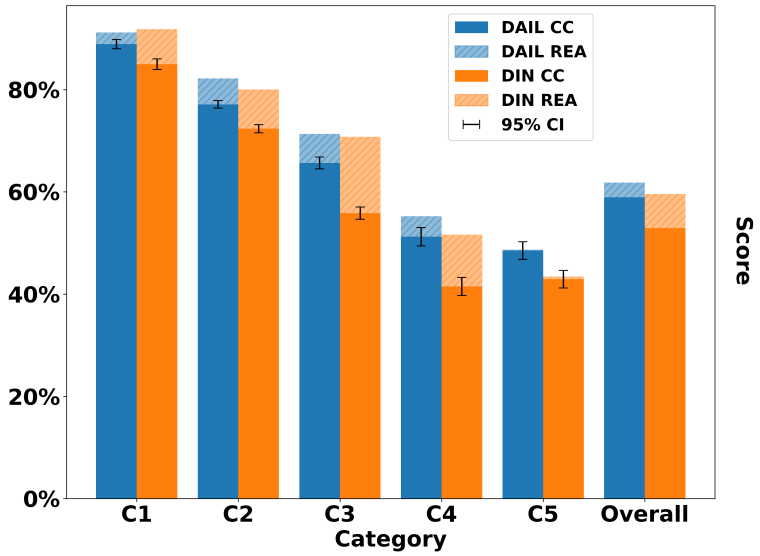


Figure 5.13: Complexity Consistency (CC) per category (aligned workload). Higher values indicate that the system performed better in keeping the complexity of the predicted queries close to ground truth queries.

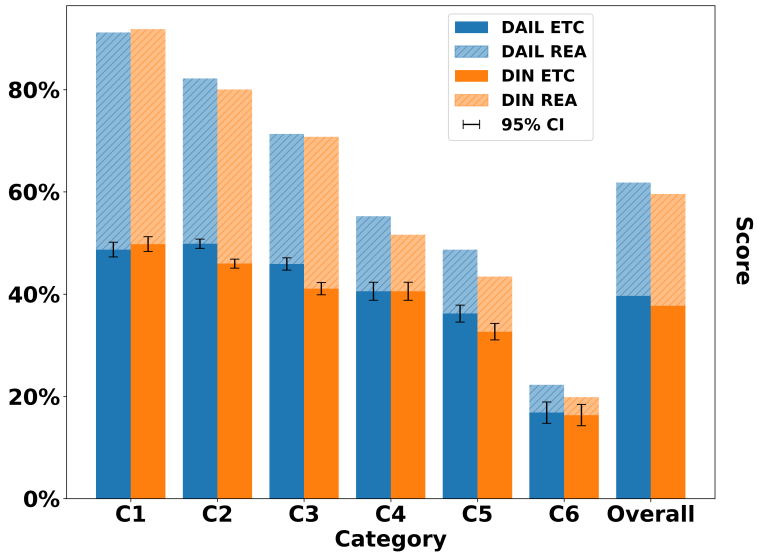


Figure 5.14: Execution Time Consistency (ETC) per category (aligned workload). Higher values indicate that the system performed better in keeping the execution efficiency of the predicted queries close to ground truth queries.

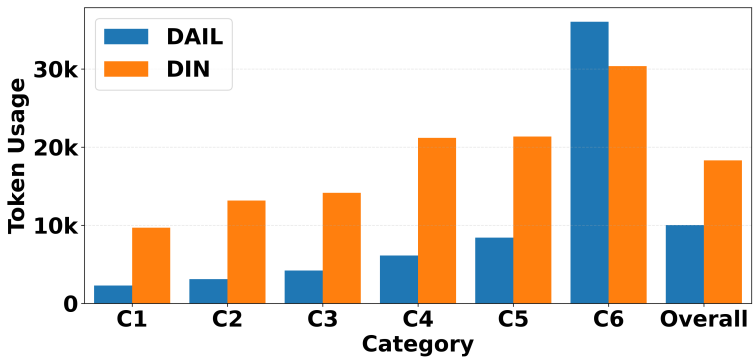


Figure 5.15: Token Usage per category (aligned workload)

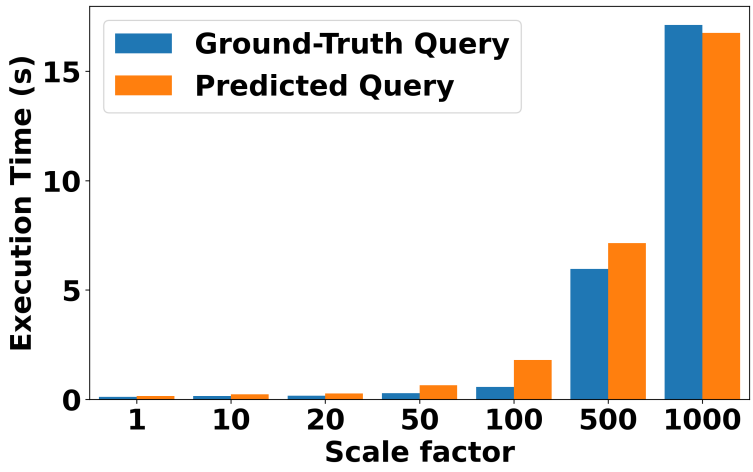


Figure 5.16: Average execution time of ground truth and predicted queries across scaling factors.

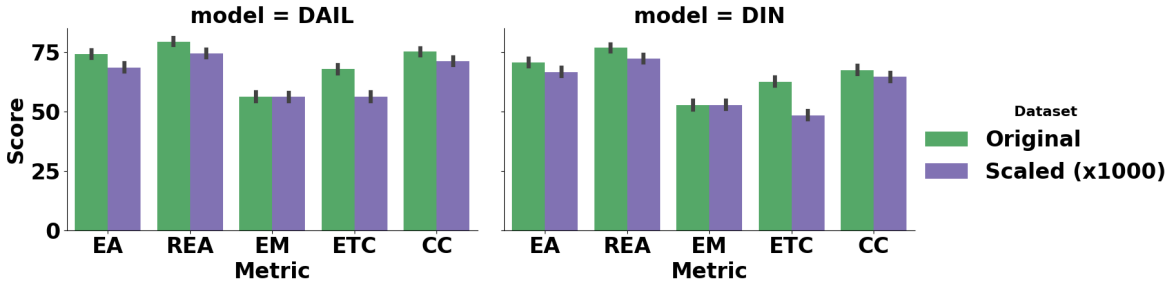


Figure 5.17: Overall evaluation scores of the scaling workload on original and $\times 1000$ scaled datasets.

Chapter 6

Conclusion & Future Work

6.1 Conclusion

Existing text-to-SQL benchmarks have several limitations. First, their evaluation metrics solely focus on the correctness of the generated queries and fail to capture other critical aspects, such as complexity, execution efficiency, and generation cost. Moreover, they incorporate synthetic workloads that do not align with real-world usage patterns, and use small, fixed-size datasets, which do not reflect enterprise and large-scale database settings.

In this work, we propose the SQLyzr benchmark to address these limitations of existing works. The primary contribution of this work is the design of a comprehensive benchmark, summarized as follows:

- **Fine-Grained analysis:** SQLyzr introduces a comprehensive SQL taxonomy and reports evaluation scores at the level of query categories and subcategories (Section 3.1.2).
- **Realistic workload:** the SQLyzr workload is aligned with empirical SQL usage patterns observed in the SQLShare corpus (Section 3.1.3).
- **Dataset scalability:** SQLyzr supports scaling databases to varying sizes, allowing researchers to evaluate the efficiency of generated queries under large-scale settings (Section 3.2).
- **Comprehensive evaluation metrics:** SQLyzr incorporates a suite of evaluation metrics that extends traditional measures and covers additional aspects of query generation beyond correctness (Section 3.3).

The second contribution of this work is the design and implementation of a benchmarking framework with the following capabilities:

- **Configurability and extensibility:** The SQLyzr framework is designed with modularity and extensibility in mind. Various components of the framework, including workload, dataset, and metrics, can be configured or extended to support different benchmarking scenarios (4.1).
- **Workload augmentation:** Based on the evaluation results of a text-to-SQL system, SQLyzr synthesizes new question-SQL pairs for underperforming subcategories to augment the workload with additional data points that target the system's weaknesses (Section 4.2.6).
- **Error analysis:** SQLyzr identifies incorrect queries that contain minor, potentially fixable errors to guide improvements to the system correctness (Section 4.3).

To demonstrate the analytical capabilities of SQLyzr and the insights it provides, we conducted an experimental evaluation of two state-of-the-art text-to-SQL systems, DIN-SQL and DAIL-SQL (Chapter 5).

The key observations from our experiments are summarized as follows:

- **DAIL-SQL is the superior system overall:** Existing benchmarks show that DAIL-SQL marginally outperforms DIN-SQL in terms of correctness. However, a more comprehensive evaluation enabled by SQLyzr reveals that DAIL-SQL also outperforms DIN-SQL in other aspects of the translation task, such as efficiency, token usage, and the complexity of the predicted query (Section 5.2).
- **Challenging query patterns:** Subcategory-level analysis reveals that specific SQL constructs, such as arithmetic expressions and advanced constructs (e.g., window functions and common table expressions), remain challenging for both text-to-SQL systems. Identifying these patterns provides clear directions for future improvements in text-to-SQL translation accuracy (Section 5.3).
- **Category-level scores are more robust:** Our experiments show that category-level scores are more stable across different workloads. Consequently, a macro-average score over the categories is a more robust aggregation strategy than micro-averages (Section 5.4).

6.2 Future Work

In this section, we outline future directions to further enhance both the benchmark and the framework.

6.2.1 Partial Correctness Metrics

While relaxed execution accuracy (REA) mitigates some of the strictness of execution accuracy, it remains a binary metric. Therefore, it does not quantify the extent to which a predicted query deviates from the ground truth. In practice, however, predicted queries can exhibit partial correctness, e.g., they may contain correct joins and filtering conditions but miss a column in the `SELECT` clause. Future work could explore graded metrics that assign a score in the interval $[0, 1]$ that reflects the degree of semantic similarity between predicted and ground truth queries.

6.2.2 Semantic Correctness Assessment

Determining the semantic equivalence of arbitrary SQL queries is undecidable. Existing correctness metrics, such as execution accuracy (EA) and relaxed execution accuracy (REA), provide a practical approximation of semantic equivalence that relies on comparing execution results of queries on a single database instance. However, these methods could lead to false positives, where semantically different queries yield the same result on a specific database.

To further strengthen correctness assessment, future work can explore utilizing language models for determining the semantic equivalence of queries. A principled approach is to formulate semantic equivalence checking as a supervised classification problem. Given a pair (q, q') and the associated schema, a classifier can be trained to predict whether the queries are semantically equivalent.

6.2.3 Human-in-the-Loop Evaluation

Future extensions of the SQLyzr framework could include interactive interfaces that allow domain experts to inspect predicted queries and validate correctness. Such a human-in-the-loop evaluation paradigm would bridge automated benchmarking with expert validation and could improve the reliability of the evaluations.

6.2.4 Dynamic Taxonomy Construction

The taxonomy used in SQLyzr is currently manually designed based on systematic analysis of SQL constructs. However, this approach may not optimally reflect the characteristics of every workload and benchmarking scenario. Different domains may require different categories or finer-grained subcategories. Future work could explore automated or semi-automated taxonomy construction.

6.2.5 LLM-Assisted Error-Fixing Suggestions

Currently, SQLyzr can only identify abstract error-fixing rules that indicate how a predicted query could be corrected. However, synthesizing a concrete fix tailored to a specific query has not yet been implemented. The main challenge of generating such concrete fixes is the large space of possible modifications, which makes enumerating them impractical. A possible direction for future work is to generate a large pool of deterministic candidate fixes and utilize LLMs to pick the most suitable fix for a given query. Such a hybrid approach could avoid enumerating all possible fixes while benefiting from the deterministically crafted fixes.

6.2.6 Extensions Beyond SQL

The SQLyzr benchmark focuses on evaluating text-to-SQL systems. However, many modern applications require translating natural language into other structured query languages or programming languages, such as SPARQL [31] and GraphQL [26].

Several core ideas introduced in SQLyzr, including fine-grained analysis, relaxed correctness measurements, workload alignment, and workload augmentation, are not limited to SQL and can be applied to benchmarking of natural language translation systems for other query languages. Utilizing these features in benchmarks enables a more detailed and practical evaluation and facilitates the improvement of such translation systems.

6.3 Final Remarks

As natural language interfaces to databases become increasingly practical and widely deployed, it is becoming essential to understand the true capabilities and limitations of modern text-to-SQL systems.

In this work, we proposed a comprehensive text-to-SQL benchmark to enable more realistic and insightful evaluation of text-to-SQL systems. Moreover, we introduced a benchmarking framework to provide a flexible evaluation platform that allows researchers to configure it according to their requirements with new workloads, datasets, metrics.

We hope that SQLyzr facilitates future research on text-to-SQL translation by providing practical and useful analytical capabilities to researchers.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 technical report. arXiv 2303.08774, 2023.

[2] Alfred V Aho and Jeffrey D Ullman. The theory of languages. *Mathematical Systems Theory*, 2(2):97–125, 1968.

[3] Ion Androutsopoulos, Graeme D. Ritchie, and Peter Thanisch. Natural language interfaces to databases–an introduction. *Natural Language Engineering*, 1(1):29–81, 1995.

[4] Benjamin G. Ascoli, Yasoda Sai Ram Kandikonda, and Jinho D. Choi. ETM: Modern insights into perspective on text-to-SQL evaluation in the age of large language models. *Information*, 16(2):123, 2025.

[5] Chandra Sekhar Bhagavatula, Thanapon Noraset, and Doug Downey. TabEL: Entity linking in web tables. In *Proc. International Semantic Web Conference*, pages 425–441. Springer, Cham, 2015.

[6] Adithya Bhaskar, Tushar Tomar, Ashutosh Sathe, and Sunita Sarawagi. Benchmarking and improving text-to-SQL generation under ambiguity. In *Proc. Conference on Empirical Methods in Natural Language Processing*, pages 10036–10058, 2023.

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information*

Processing Syst., Proc. 34th Annual Conf. on Neural Information Processing Syst., NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[8] Ursin Brunner and Kurt Stockinger. Valuenet: A natural language-to-sql system that learns from database information. In *Proc. 37th IEEE Intl. Conf. on Data Engineering*, pages 2177–2182, 2021.

[9] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A structured English query language. In *Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control*, pages 249–264, 1974.

[10] Shuaichen Chang, Jun Wang, Mingwen Dong, Lin Pan, Henghui Zhu, Alexander Hanbo Li, Wuwei Lan, Sheng Zhang, Jiarong Jiang, Joseph Lilien, Steve Ash, William Yang Wang, Zhiguo Wang, Vittorio Castelli, Patrick Ng, and Bing Xiang. Dr.Spider: A diagnostic evaluation benchmark towards text-to-SQL robustness. arXiv 2301.08881, 2023.

[11] Peter Baile Chen, Fabian Wenz, Yi Zhang, Devin Yang, Justin Choi, Moe Kayali, Nesime Tatbul, Michael Cafarella, Çağatay Demiralp, and Michael Stonebraker. BEAVER: An enterprise benchmark for text-to-SQL. arXiv 2409.02038, 2024.

[12] David Chiang. A hierarchical phrase-based model for statistical machine translation. In *Proc. Annual Meeting of the Association for Computational Linguistics*, pages 263–270, 2005.

[13] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries. *Proc. VLDB Endowment*, 11(11):1482–1495, 2018.

[14] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. Cosette: An automated prover for SQL. In *Proc. Conference on Innovative Data Systems Research*, pages 1–7, 2017.

[15] E. F. Codd. Seven steps to rendezvous with the casual user. In J. W. Klimbie and K. L. Koffeman, editors, *In Proc. IFIP Working Conf. Database Management*, pages 179–200, 1974.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proc. 2019*

Conf. North American Chapter of the Association for Computational Linguistics: Human Lang. Tech., volume 1, pages 4171–4186. Association for Computational Linguistics, 2019.

[17] Xuemei Dong, Cao Zhang, Ran Ge, Yuzhao Yang, Cheng Luo, Huanyong Xu, and Jiaqi Fu. C3-SQL: Clear, concise, and consistent prompting for text-to-SQL. arXiv 2307.07306, 2023.

[18] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. 6 edition, 2011.

[19] Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. Improving text-to-SQL evaluation methodology. arXiv 1806.09029, 2018.

[20] Yujian Gan, Xinyun Chen, Qiuping Huang, and Matthew Purver. Measuring and improving compositional generalization in text-to-SQL via component alignment. arXiv 2205.02054, 2022.

[21] Yujian Gan, Xinyun Chen, Qiuping Huang, Matthew Purver, John R. Woodward, Jinxia Xie, and Pengsheng Huang. Towards robustness of text-to-SQL models against synonym substitution. arXiv 2106.01065, 2021.

[22] Yujian Gan, Xinyun Chen, and Matthew Purver. Exploring underexplored limitations of cross-domain text-to-SQL generalization. In *Proc. 2021 Conference on Empirical Methods in Natural Language Processing*, pages 10326–10331, 2021.

[23] Yujian Gan, Xinyun Chen, Jinxia Xie, Matthew Purver, John R. Woodward, John Drake, and Qiaofu Zhang. Natural SQL: Making SQL easier to infer from natural language specifications. In *Proc. Findings of the Association for Computational Linguistics*, volume EMNLP 2021, pages 2030–2042, 2021.

[24] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-SQL empowered by large language models: A benchmark evaluation. arXiv 2308.15363, 2023.

[25] Orest Gkini, Theofilos Belmpas, Georgia Koutrika, and Yannis E. Ioannidis. An in-depth benchmarking of text-to-SQL systems. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 632–644, 2021.

[26] GraphQL Foundation. GraphQL specification. Specification, 2021. Available at: https://spec.graphql.org/.

[27] Bert F. Green, Alice K. Wolf, Carol Chomsky, and Kenneth Laughery. Baseball: An automatic question-answerer. In *Proc. Western Joint Computer Conference*, pages 219–224, 1961.

[28] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *Proc. 26th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 31–40, 2007.

[29] Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. Towards complex text-to-SQL in cross-domain database with intermediate representation. arXiv 1905.08205, 2019.

[30] Jiaqi Guo, Zuoquan Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. IRNet: A neural approach for cross-domain text-to-SQL. In *Proc. Annual Meeting of the Association for Computational Linguistics*, volume 1, pages 2238–2248, 2019.

[31] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 query language. W3C Recommendation, 2013. Available at: http://www.w3.org/TR/sparql11-query/.

[32] Moshe Hazoom, Vibhor Malik, and Ben Bogin. Text-to-SQL in the wild: A naturally-occurring dataset based on Stack Exchange data. arXiv 2106.05006, 2021.

[33] Charles T. Hemphill, John J. Godfrey, and George R. Doddington. The ATIS spoken language systems pilot corpus. In *Proc. Speech and Natural Language: Workshop Held at Hidden Valley, PA*, pages 96–101. Association for Computational Linguistics, 1990.

[34] Gary G. Hendrix, Earl D. Sacerdoti, Daniel Sagalowicz, and Jonathan Slocum. Developing a natural language interface to complex data. *ACM Transactions on Database Systems*, 3(2):105–147, 1978.

[35] Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. Next-generation database interfaces: A survey of LLM-based text-to-SQL. arXiv 2406.08426, 2024.

[36] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. Learning a neural semantic parser from user feedback. arXiv 1704.08760, 2017.

[37] Shrainik Jain, Dominik Moritz, Daniel Halperin, Bill Howe, and Ed Lazowska. Sqlshare: Results from a multi-year sql-as-a-service experiment. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 281–293, 2016.

[38] Ronak Kaoshik, Rohit Patil, Shaurya Agarawal, Naman Jain, and Mayank Singh. ACL-SQL: Generating SQL queries from natural language. In *Proc. 3rd ACM India Joint International Conference on Data Science and Management of Data (8th ACM IKDD CODS and 26th COMAD)*, page 423, 2021.

[39] Hyeonji Kim, Byeong-Hoon So, Wook-Shin Han, and Hongrae Lee. Natural language to SQL: where are we today? *Proc. VLDB Endowment*, 13(10):1737–1750, 2020.

[40] Won Kim. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems (TODS)*, 7(3):443–469, 1982.

[41] Robert A. Kowalski. The early years of logic programming. *Communications of the ACM*, 31(1):38–43, 1988.

[42] Wuwei Lan, Zhiguo Wang, Anuj Chauhan, Henghui Zhu, Alexander Li, Jiang Guo, Sheng Zhang, Chung-Wei Hang, Joseph Lilien, Yiqun Hu, et al. UniTE: A unified benchmark for text-to-SQL evaluation. arXiv 2305.16265, 2023.

[43] Chia-Hsuan Lee, Oleksandr Polozov, and Matthew Richardson. KaggleDBQA: Realistic evaluation of text-to-SQL parsers. In *Proc. 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*, pages 2261–2273, 2021.

[44] Gyubok Lee, Hyeonji Hwang, Seongsu Bae, Yeonsu Kwon, Woncheol Shin, Seongjun Yang, Minjoon Seo, Jong-Yeup Kim, and Edward Choi. EHRSQL: A practical text-to-SQL benchmark for electronic health records. In *Proc. Advances in Neural Information Processing Systems*, volume 35, pages 15589–15601. Curran Associates, Inc., 2022.

[45] Wenqiang Lei, Weixin Wang, Zhixin Ma, Tian Gan, Wei Lu, Min-Yen Kan, and Tat-Seng Chua. Re-examining the role of schema linking in text-to-SQL. In *Proc. Conference on Empirical Methods in Natural Language Processing*, pages 6943–6954. Association for Computational Linguistics, 2020.

[46] Fei Li and Hosagrahar V Jagadish. Nalir: An interactive natural language interface for querying relational databases. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 709–712, 2014.

[47] Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. Codes: Towards building open-source language models for text-to-SQL. *Proc. ACM Management of Data*, 2(3), 2024.

[48] Haoyang Li, Jingping Zhang, Yanzhao Cao, Lidong Bing, and Rui Yan. ResdSQL: Decoupling schema linking and skeleton parsing for text-to-SQL. In *Proc. AAAI Conference on Artificial Intelligence*, volume 37, pages 13066–13074, 2023.

[49] Jinyang Li, Binyuan Hui, Reynold Cheng, Bowen Qin, Chenhao Ma, Nan Huo, Fei Huang, Wenyu Du, Luo Si, and Yongbin Li. Graphix-T5: Mixing pre-trained transformers with graph-aware layers for text-to-SQL parsing. In *Proc. AAAI Conference on Artificial Intelligence*, volume 37, pages 13076–13084. Association for the Advancement of Artificial Intelligence (AAAI), 2023.

[50] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-SQLs. *Advances in Neural Information Processing Systems*, 36, 2024.

[51] Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. A survey of text-to-SQL in the era of LLMs: Where are we, and where are we going? *IEEE Transactions on Knowledge and Data Engineering*, 37(4):1954–1972, 2025.

[52] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A robustly optimized BERT pretraining approach. arXiv 1907.11692, 2019.

[53] J. Melton. *Advanced SQL: 1999 - Understanding Object-Relational and Other Advanced Features*. 2002.

[54] Anna Mitsopoulou and Georgia Koutrika. Analysis of text-to-SQL benchmarks: Limitations, challenges and opportunities. In *Proc. 28th International Conference on Extending Database Technology*, pages 618–630, 2025.

[55] Farhad Nooralahzadeh, Yi Zhang, Ellery Smith, Sabine Maennel, Cyril Matthey-Doret, Raphaël De Fondeville, and Kurt Stockinger. StatBot.Swiss: Bilingual open data exploration in natural language. In *Proc. Findings of the Association for Computational Linguistics*, volume ACL 2024, pages 5486–5507, 2024.

[56] Oracle Corporation. *MySQL 8.4 Reference Manual*. Oracle Corporation, Austin, TX, 2024. Accessed: 2024-11-15.

[57] Neha Patki, Roy Wedge, and Kalyan Veeramachaneni. The synthetic data vault. In *Proc. IEEE International Conference on Data Science and Advanced Analytics*, pages 399–410. IEEE, 2016.

[58] Xinyu Pi, Bing Wang, Yan Gao, Jiaqi Guo, Zhoujun Li, and Jian-Guang Lou. Towards robustness of text-to-SQL models against natural and realistic adversarial table perturbation. arXiv 2212.09994, 2022.

[59] Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. Towards a theory of natural language interfaces to databases. In *Proc. International Conference on Intelligent User Interfaces*, pages 149–157, 2003.

[60] Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. Towards a theory of natural language interfaces to databases. In *Proc. International Conference on Intelligent User Interfaces*, pages 149–157. Association for Computing Machinery, 2003.

[61] Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O. Arik. CHASE-SQL: Multi-path reasoning and preference optimized candidate selection in text-to-SQL. arXiv 2410.01943, 2024.

[62] Mohammadreza Pourreza and Davood Rafiei. DIN-SQL: Decomposed in-context learning of text-to-SQL with self-correction. In *Proc. Advances in Neural Information Processing Systems*, volume 36, pages 30557–30584, 2024.

[63] Cedric Renggli, Ihab F. Ilyas, and Theodoros Rekatsinas. Fundamental challenges in evaluating Text2SQL solutions and detecting their limitations. arXiv 2501.18197, 2025.

[64] Jaydeep Sen, Chuan Lei, Abdul Quamar, Fatma Özcan, Vasilis Efthymiou, Ayushi Dalmia, Greg Stager, Ashish Mittal, Diptikalyan Saha, and Karthik Sankaranarayanan. Athena++: natural language querying for complex nested sql queries. *Proc. VLDB Endowment*, 13(12):2747–2759, 2020.

[65] Tianze Shi, Chen Zhao, Jordan Boyd-Graber, Hal Daumé III, and Lillian Lee. On the potential of lexico-logical alignments for semantic parsing to SQL queries. arXiv 2010.11246, 2020.

[66] A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts*. 7 edition, 2019.

[67] Aditi Singh, Akash Shetty, Abul Ehtesham, Saket Kumar, and Tala Talaei Khoei. A survey of large language model-based generative AI for text-to-SQL: Benchmarks, applications, use cases, and challenges. In *Proc. IEEE Annual Computing and Communication Workshop and Conference*, pages 15–21. IEEE, 2025.

[68] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. MPNet: Masked and permuted pre-training for language understanding. In *Proc. Advances in Neural Information Processing Systems*, volume 33, pages 16857–16867, 2020.

[69] Michael Stonebraker and Andrew Pavlo. What goes around comes around... and around... *ACM SIGMOD Record*, 53(2):21–37, 2024.

[70] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proc. 27th International Conference on Neural Information Processing Systems*, pages 3104–3112, 2014.

[71] Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. CHESS: Contextual harnessing for efficient SQL synthesis. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, volume 235 of *Proceedings of Machine Learning Research*, pages 49375–49412. PMLR, 2025.

[72] Bailin Wang, Richard Shin, Xiaowei Liu, Oleksandr Polozov, and Matthew Richardson. RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers. In *Proc. Annual Meeting of the Association for Computational Linguistics*, pages 5735–5745, 2020.

[73] Ping Wang, Tian Shi, and Chandan K. Reddy. Text-to-SQL generation for question answering on electronic medical records. In *Proc. International World Wide Web Conference*, pages 350–361, 2020.

[74] David H. D. Warren and Fernando C. N. Pereira. An efficient easily adaptable system for interpreting natural language queries. *American Journal of Computational Linguistics*, 8(3–4):110–122, 1982.

[75] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V. Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. In *Proc. Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837, 2022.

[76] William A. Woods. The LUNAR sciences natural language information system: Final report. Technical Report BBN Report No. 2378, Bolt Beranek and Newman Inc., 1972.

[77] Xiaojun Xu, Chang Liu, and Dawn Song. SQLNet: Generating structured queries from natural language without reinforcement learning. arXiv 1711.04436, 2017.

[78] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. SQLizer: Query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):63:1–63:26, 2017.

[79] Ziyu Yao, Daniel S. Weld, Wei-Peng Chen, and Huan Sun. STAQC: A systematically mined question-code dataset from Stack Overflow. In *Proc. World Wide Web Conference*, pages 1693–1703. International World Wide Web Conferences Steering Committee, 2018.

[80] Tao Yu. Spider: A large-scale complex and cross-domain semantic parsing and text-to-SQL dataset. https://github.com/taoyds/spider, 2018. GitHub repository. Accessed: 2024-11-15.

[81] Tao Yu, Chien-Sheng Wu, Xi Victoria Lin, Bailin Wang, Yi Chern Tan, Xinyi Yang, Dragomir Radev, Richard Socher, and Caiming Xiong. GraPPa: Grammar-augmented pre-training for table semantic parsing. arXiv 2009.13845, 2020.

[82] Tao Yu, Rui Zhang, He Yang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, et al. CoSQL: A conversational text-to-SQL challenge towards cross-domain natural language interfaces to databases. arXiv 1909.05378, 2019.

[83] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proc. 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, 2018.

[84] Tao Yu, Rui Zhang, Michihiro Yasunaga, Yi Chern Tan, Xi Victoria Lin, Suyi Li, Heyang Er, Irene Li, Bo Pang, Tao Chen, et al. SParC: Cross-domain semantic parsing in context. arXiv 1906.02285, 2019.

[85] Xiaojing Yu, Tianlong Chen, Zhengjie Yu, Huiyu Li, Yang Yang, Xiaoqian Jiang, and Anxiao Jiang. Dataset and enhanced model for eligibility criteria-to-SQL semantic

parsing. In *Proc. 12th International Conference on Language Resources and Evaluation*, pages 5831–5838, 2020.

[86] Ruiqi Zhong, Tao Yu, and Dan Klein. Semantic evaluation for text-to-SQL with distilled test suites. In *Proc. Conference on Empirical Methods in Natural Language Processing*, pages 396–411. Association for Computational Linguistics, 2020.

[87] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2SQL: Generating structured queries from natural language using reinforcement learning. arXiv 1709.00103, 2017.

APPENDICES

Appendix A

Workload Augmentation with LLM

To augment the workload with synthetic question-SQL pairs (Section 4.2.6), SQLyzr employs few-shot prompting with a large language model. The goal of each prompt is to generate new question-SQL pairs for a specific SQL subcategory and database schema.

The subcategories targeted for augmentation are determined based on the EA scores of the evaluated system. Any subcategory whose EA score falls below a user-defined threshold is selected for workload augmentation. For each selected subcategory, SQLyzr retrieves question-SQL pairs from the training workload and includes them in the LLM prompt as demonstration examples.

The prompt instructs the LLM to generate a SQL query that satisfies the required subcategory constraint while avoiding constraints of other subcategories. As an example, the template of prompts used for synthesizing new question-SQL pairs for the subcategory S_8 (queries containing the `DISTINCT` keyword) is shown below. In this template, variables are represented using {}.

A.1 Workload Augmentation Prompt

```
You are an expert in SQLite and SQL query design.

Your task is to generate {num_pairs} synthetic text-SQL pairs.

Each pair must contain:
1. A natural language question.
```

```
2. A valid SQLite SQL query that answers the question regarding the given
↪  database schema.

GENERAL REQUIREMENTS

- Each SQL query must follow the provided database schema.
- Each SQL query must satisfy the REQUIRED constraint.
- Each SQL query must NOT satisfy any of the FORBIDDEN constraints.
- The natural language question must accurately describe the SQL query.
- Queries must be realistic and meaningful.
- Avoid trivial or duplicate questions.

------------------------------------------------
SUBCATEGORY CONSTRAINT DEFINITIONS

Each constraint corresponds to a specific SQL structure:
        s1: Contains multiple columns in the SELECT clause and a single
        ↪  table in the FROM clause
        s2: Contains a star column in the SELECT clause
        s3: Contains ORDER_BY keyword
        s4: Contains a single binary expression in the WHERE clause
        s5: Contains arithmetic expressions
        s6: Contains aggregate functions (e.g., COUNT, AVG, MIN)
        s7: Contains the LIMIT keyword
        s8: Contains the DISTINCT Keyword
        s9: Contains GROUP BY clauses without a condition (HAVING)
        s10: Contains natural JOINs
        s11: Contains a non-equi JOIN
        s12: Contains a BETWEEN expression
        s13: Contains JOIN clauses without a condition and contains a
        ↪  single expression in the WHERE clause
        s14: Contains a single expression in the WHERE clause and a GROUP
        ↪  BY clause without a condition (HAVING)
        s15: Contains a JOIN without a condition and contains a GROUP BY
        ↪  clause without a condition
        s16: Contains a GROUP BY clause with a condition
        s17: Contains a LIKE expression
        s18: Contains a NULL expression
```

```
        s19: Contains multiple expressions in the WHERE clause
        s20: Contains nested queries with one level of nesting
        s21: Contains a composition keyword (e.g., INTERSECT or UNION)
        s22: Contains IN expressions and single-level nested queries
        s23: Contains an INNER JOIN
        s24: Contains a non-simple JOIN (e.g., INNER, OUTER) and an ORDER
        ↪  BY keyword
        s25: Contains a non-simple JOIN (e.g., INNER, OUTER) with
        ↪  unconditional GROUP BY clauses
        s26: Contains a non-simple JOIN with aggregation keywords (e.g.,
        ↪  COUNT, AVG, MIN)
        s27: Contains more than one JOINs
        s28: Contains sub-queries with one level of nesting and GROUP BY
        ↪  clauses without a condition
        s29: Contains sub-queries with one level of nesting and JOINs
        s30: Contains sub-queries with one level of nesting and aggregate
        ↪  functions (e.g., COUNT, AVG, MIN)
        s31: Contains sub-queries with two levels of nesting
        s32: Contains a CASE expression
        s33: Contains sub-queries and composition keywords (e.g.,
        ↪  INTERSECT, UNION)
        s34: Contains a sub-query with more than two levels of nesting
        s35: Contains a Common-Table Expression
        s36: Contains a Window Function

------------------------------------------------
DATABASE SCHEMA
{database_schema}

------------------------------------------------
TARGET CONSTRAINT

The generated SQL queries MUST satisfy:
{target_subcategory}

------------------------------------------------
FORBIDDEN CONSTRAINTS
```

```
The generated SQL queries MUST NOT satisfy:
{forbidden_subcategories}

-------------------------------------------------
OUTPUT FORMAT

Return exactly {num_pairs} text-SQL pairs using the following format:

Pair #1
SQL: <SQL query>
Question: <natural language question>

Pair #2
SQL: <SQL query>
Question: <natural language question>
.
.
.
Pair #{num_pairs}
SQL: <SQL query>
Question: <natural language question>

-------------------------------------------------
********** Example **********

NUMBER of PAIRS: 10

DATABASE SCHEMA:

Tables:
  Table Name: list
    Columns:
      Column Name: lastname, Column type: text
      Column Name: firstname, Column type: text
      Column Name: grade, Column type: number
      Column Name: classroom, Column type: number
  Table Name: teachers
    Columns:
```

```
      Column Name: lastname, Column type: text
      Column Name: firstname, Column type: text
      Column Name: classroom, Column type: number

TARGET CONSTRAINT:

The generated SQL queries MUST satisfy:
s8

FORBIDDEN CONSTRAINTS:

The generated SQL queries MUST NOT satisfy:
s3, s4, s5, s6, s7, s9, s10, s10, s11, s12, s13, s14, s15, s16, s17, s18,
↪  s19, s20, s21, s22, s23, s24, s25, s26, s27, s28, s29, s30, s31, s32,
↪  s33, s34, s35, s36

OUTPUT:

Pair #1
SQL: SELECT DISTINCT grade FROM list WHERE classroom  =  105
Question: Find the grade of students studying in classroom 105.
.
.
.
Pair #10
SQL: SELECT DISTINCT classroom FROM list  WHERE grade < 4;
Question: Which classrooms host students in grades below 4?
**************************
```

Appendix B

Error-Fixing Suggestions

During the error-analysis stage (Section 4.3), SQLyzr identifies a set of error-fixing rules that can transform a predicted query so that it produces the same execution result as the ground truth query. For each detected rule, the framework produces a textual suggestion describing the required modification.

Table B.1 lists the mapping between error-fixing rules and the feedback messages produced by the system.

Error-Fixing Rule	**Produced Suggestion**
Limit Clause Fixing Rule	"The `LIMIT` clause in the predicted SQL query should be corrected."
Missing Columns Fixing Rule	"The predicted SQL query should be modified to include the missing columns in the `SELECT` clause."
Extra Columns Removal Rule	"The predicted SQL query should be modified to remove extra columns from the `SELECT` clause."
Extra Rows Removal Rule	"The predicted SQL query should be modified to remove extra rows from the result set."
Column Order Fixing Rule	"The predicted SQL query should be modified to match the column order of the ground truth query."
Row Order Fixing Rule	"The predicted SQL query should be modified to match the row ordering of the ground truth query."

Table B.1: Mapping between error-fixing rules and SQLyzr's textual suggestions