Cracking Query Bottlenecks: Towards Efficiency-Oriented Text-to-SQL Generation

LI LIN, The State Key Laboratory of Blockchain and Data Security, College of Computer Science and Technology, Zhejiang University, China
YUNFENG SHEN, The State Key Laboratory of Blockchain and Data Security, College of Computer Science and Technology, Zhejiang University, China
LINGFENG BAO, The State Key Laboratory of Blockchain and Data Security, College of Computer Science and Technology, Zhejiang University, China
RONGXIN WU, School of Informatics, Xiamen University, China
YANG LIU, Nanyang Technological University, Singapore

Text-to-SQL models translate natural language questions into SQL, enabling non-technical users to access databases. However, most existing research focuses on correctness, neglecting query efficiency. In this paper, we address the challenge of evaluating the execution efficiency of generated SQL in Text-to-SQL by introducing EESQLBench, a novel benchmark designed to assess both correctness and efficiency. EESQLBench pairs each natural language question with an expert-optimized SQL query, providing a reliable efficiency baseline. We evaluate six representative large language models (LLMs), including four open-source models (SQLCoder, CodeLlama, DeepSeek-Coder, and DeepSeek-R1) and two closed-source models (GPT-5.2 and Gemini-2.5-Pro), using cost-based metrics including Cost Reachability (CR) and Acceptable Reachability at k (AR@k). Our results reveal that current LLMs, despite achieving high correctness, struggle to produce efficient queries. We observe substantial efficiency gaps between models and emphasize that semantic correctness alone does not guarantee query efficiency. Furthermore, we provide insights into common inefficiency patterns in LLM-generated SQL queries, such as missing access pruning and inefficient subquery logic.

CCS Concepts: • **Software and its engineering** → *Software maintenance tools.*

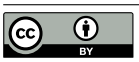
Additional Key Words and Phrases: Text-to-SQL, large language models, benchmark

ACM Reference Format:
Li Lin, Yunfeng Shen, Lingfeng Bao, Rongxin Wu, and Yang Liu. 2026. Cracking Query Bottlenecks: Towards Efficiency-Oriented Text-to-SQL Generation. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA178 (October 2026), 23 pages. https://doi.org/10.1145/3832269

1 Introduction

Text-to-SQL leverages recent advances in Natural Language Processing (NLP) and machine learning models to translate natural language (NL) questions into SQL queries [1, 23, 46]. This approach holds great potential for enabling non-technical users to access and interact with databases in an intuitive and convenient manner [1, 6, 46]. Consequently, Text-to-SQL has been widely applied in

Authors' Contact Information: Li Lin, The State Key Laboratory of Blockchain and Data Security, College of Computer Science and Technology, Zhejiang University, HangZhou, China, linli1210@zju.edu.cn; Yunfeng Shen, The State Key Laboratory of Blockchain and Data Security, College of Computer Science and Technology, Zhejiang University, HangZhou, China, shenyunfeng@zju.edu.cn; Lingfeng Bao, The State Key Laboratory of Blockchain and Data Security, College of Computer Science and Technology, Zhejiang University, HangZhou, China, lingfengbao@zju.edu.cn; Rongxin Wu, School of Informatics, Xiamen University, XiaMen, China, wurongxin@xmu.edu.cn; Yang Liu, Nanyang Technological University, , Singapore, Yangliu@ntu.edu.sg.



ACM 2994-970X/2026/10-ARTISSTA178
https://doi.org/10.1145/3832269

various domains, including Business Intelligence and data analytics systems, virtual assistants, and educational tools [16, 22, 36, 42, 50, 64]. Progress in Text-to-SQL has been driven by benchmark datasets that pair NL questions with SQL queries, such as Spider [69] and BIRD [35]. With the success of Large Language Models (LLMs), the state-of-the-art Text-to-SQL models are typically based on two paradigms: (1) fine-tuning on Text-to-SQL training datasets [32–34, 51, 53, 62], and (2) few-shot prompting using examples extracted from these datasets [19, 33, 47]. After training, researchers evaluate model performance and accuracy using corresponding test datasets.

Problem. While recent studies [19, 32–34, 47, 51, 53, 62] have primarily focused on improving the correctness of Text-to-SQL translation—ensuring that generated SQL queries produce results consistent with ground-truth queries—they have largely overlooked the efficiency of the generated queries. In practical applications, however, queries that are semantically correct may still exhibit vastly different execution times due to variations in join orders, predicate placement, or aggregation strategies [13, 27, 65]. Figure 1 illustrates this problem with a motivating example. Given the user question, an LLM-generated SQL query produces the correct result but contains redundant nested subqueries and repeated joins on the same tables. Such structural bloat causes multiple scans and materializations of intermediate results, significantly increasing I/O cost and confusing the optimizer. The LLM-generated query takes 640 ms to execute, while an equivalent but leaner version written by a database administrator (DBA) completes in only 78 ms—an improvement of more than **8**×. Such inefficiency exemplifies a deeper issue in current Text-to-SQL research: LLMs can generate semantically correct yet inefficient SQL queries that degrade system performance. This observation raises a fundamental question:

> ***Key Question.*** *To what extent can Text-to-SQL models not only produce correct queries but also generate efficient ones that align with query optimization principles?*

However, evaluating the efficiency of Text-to-SQL systems presents several challenges:

Challenge #1: Lack of Efficiency-Oriented Benchmarks. Existing datasets such as Spider [69] are designed to assess the semantic accuracy of Text-to-SQL models and do not differentiate among semantically equivalent queries with vastly different execution efficiency. These benchmarks neglect the efficiency dimension—they provide small schemas, limited data volume, and lack indexing or execution-cost annotations, making it difficult to distinguish fast from slow queries. Constructing a benchmark capable of evaluating query efficiency is inherently complex, as it requires coordinated design at the levels of *schema*, *instance*, and *query*. At the **schema level**, performance-sensitive designs are particularly suitable for revealing cost differences among equivalent queries. For example, schemas that contain multi-join chains (e.g., A–B–C–D), where join orders and predicate placements influence execution plans, tend to exhibit larger runtime variations [27, 30]. Redundant relationship paths (e.g., primary-key versus foreign-key joins) and a mixture of indexed and non-indexed tables can further diversify plan choices [27, 30]. At the **instance level**, heterogeneous data configurations help amplify performance differences [27]. Varying table scales, skewed value distributions, and partially selective indexes can make optimizer decisions more sensitive to query formulation, allowing inefficient queries—such as those with redundant joins, misplaced filters, or nested aggregations—to manifest measurable slowdowns. At the **query level**, benchmarks benefit from including semantically equivalent SQL variants that differ in structural complexity [27, 30]. Queries featuring nested subqueries, rich filtering conditions, and multi-join aggregations provide diverse opportunities for optimization and better reflect the efficiency issues commonly observed in LLM-generated SQL. Furthermore, current benchmarks do not provide expert-optimized SQL queries that could serve as reliable efficiency baselines. Without

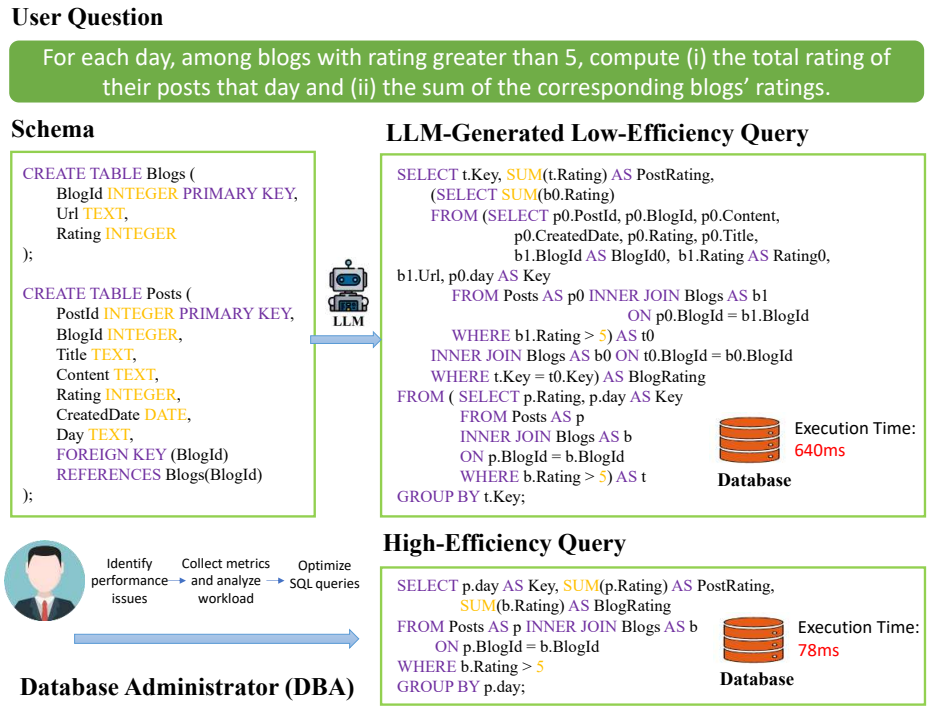


Fig. 1. **Motivating Example: Semantically Correct but Inefficient LLM-Generated Query**

such high-quality human references, it remains difficult to measure how efficiently model-generated SQLs approach expert-level performance.

Challenge #2: Unstable and Unreliable Efficiency Evaluation. Even with a well-constructed benchmark, evaluating the efficiency of generated SQL queries remains inherently unstable. Unlike correctness evaluation—where the output can be deterministically compared with a ground-truth SQL—there is no standardized "ground truth" runtime for how fast a query should execute. A query's runtime can fluctuate due to numerous external factors such as caching, buffer states, query plan caching, and concurrent system load, making execution time an unreliable indicator of intrinsic query efficiency [30]. Therefore, more stable and intrinsic metrics are needed to evaluate query efficiency independently of fluctuating runtime conditions.

EESQLBench. To address Challenge #1, we construct EESQLBench, an Efficiency-Oriented benchmark for Text-to-SQL evaluation. EESQLBench contains 213 efficiency-critical Text-to-SQL tasks, where each NL question is paired with a human-written canonical solution that represents an expert-level, efficiency-optimized SQL query. It should be noted that the term "optimized SQL" in our paper refers to expert-refined, efficiency-improved queries rather than globally optimal ones. Our goal is not to identify the global optimum, but to assess whether LLM-generated queries can achieve efficiency comparable to well-designed human queries. We build the benchmark upon real-world databases and enhance them through targeted schema- and instance-level interventions, such as enriching join structures and augmenting data distributions, to amplify performance-sensitive behaviors. Based on these schemas, we generate a diverse collection of structurally complex SQL queries inspired by DBMS fuzzing techniques, focusing on queries that admit substantial optimization opportunities. We then identify efficiency-critical cases by analyzing slow SQL queries and retain those that can be significantly optimized through automated rewriting tools and expert

refinement. Finally, for each retained query, we construct a corresponding NL question using a combination of LLM-assisted generation and human verification, ensuring fluency and alignment with realistic user intents.

Cost-based Efficiency Metrics. To address Challenge #2, we introduce a set of cost-based efficiency metrics that enable stable and reproducible evaluation of Efficiency-Oriented Text-to-SQL. Instead of relying on runtime-based execution latency, which is highly sensitive to caching effects, system load, and transient system states, our metrics are grounded in query plan costs. In particular, we adopt *Total Scanned Rows* as the primary cost proxy, which measures the total number of tuples processed across all operators in the executed query plan. Under identical database schemas, data instances, and query semantics, the cost is stable across repeated executions, as it reflects the logical amount of work induced by the execution plan rather than runtime-dependent factors. This property makes it a substantially more stable indicator of intrinsic query efficiency than execution time. Based on this cost formulation, we propose **Cost Reachability (CR)** and **Acceptable Reachability at k (AR@k)** to quantify how closely a LLM-generated SQL query approaches the execution efficiency of an expert-optimized canonical solution.

Evaluation. We evaluate six representative LLMs on EESQLBench, including four open-source models (SQLCoder, CodeLlama, DeepSeek-Coder, and DeepSeek-R1) and two closed-source models (GPT-5.2 and Gemini-2.5-Pro). The results show that current LLMs still struggle to generate both correct and efficient SQL queries. Even the strongest model (Gemini-2.5-Pro) attains only modest end-to-end efficiency under the strict threshold (k=1), with AR@1 reaching at most 18.2% (Simple Level) and 21.9% (Hard Level). Additionally, our qualitative analysis reveals that inefficiency mainly stems from four recurring failure modes—inefficient subquery logic, missing access-level pruning, suboptimal operator timing, and join-semantics issues—which highlights limited optimization awareness in current LLM-generated SQL.

Contributions. We summarize the contributions of this paper as follows:

- We introduce EESQLBench, the first benchmark for systematically evaluating efficiency in Text-to-SQL models, where each NL question is paired with an expert-optimized canonical SQL.
- We propose stable cost-based efficiency metrics grounded in execution-plan signals, including Cost Reachability (CR) and Acceptable Reachability at k (AR@k).
- We evaluate six representative LLMs on EESQLBench and show that semantic correctness does not necessarily imply high efficiency, revealing substantial efficiency gaps across models.

2 Background and Motivation

2.1 Preliminaries

Text-to-SQL Systems. Text-to-SQL aims to automatically translate NL questions into executable SQL queries, allowing non-expert users to interact with structured databases through NL interfaces. With the advent of LLMs, recent research has leveraged in-context learning and fine-tuning techniques to improve the semantic accuracy of SQL generation [47, 56, 58, 61, 66]. However, most existing work focuses solely on semantic correctness—whether the generated SQL produces the same result as the ground-truth query—while overlooking the equally critical dimension of execution efficiency. In real-world database systems, multiple semantically equivalent SQL queries can differ drastically in runtime due to variations in query structure, join ordering, and index utilization [35]. Such efficiency discrepancies directly affect user experience in latency-sensitive applications such as e-commerce, banking, and real-time analytics. Empirical evidence shows that an additional 500 milliseconds of query latency can reduce web traffic by up to 20% [39], and users frequently abandon websites that take longer than three seconds to load [43]. These findings

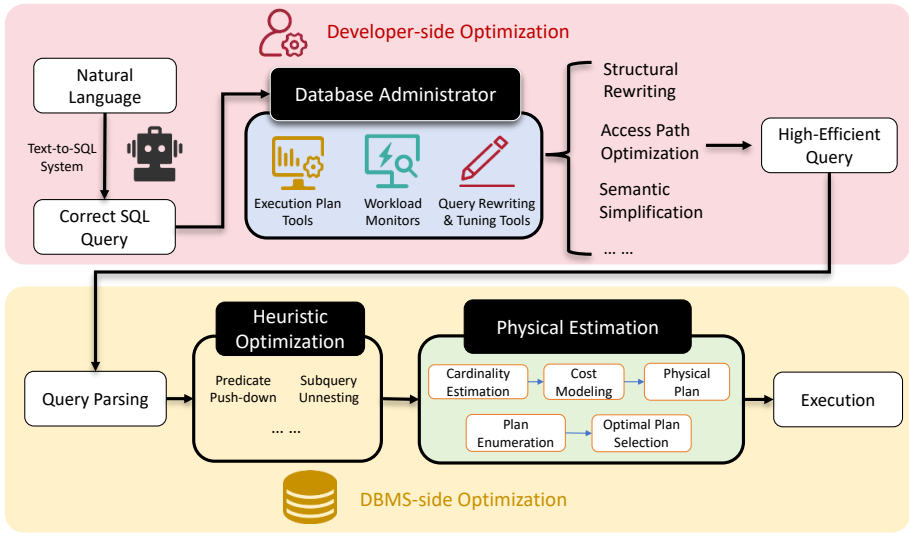


Fig. 2. **Overall Query Optimization Process**

highlight the necessity of moving beyond correctness and developing **Efficiency-Oriented Text-to-SQL** systems that not only ensure accurate query semantics but also generate SQL statements optimized for execution performance.

Query Optimization. Query optimization lies at the core of database performance, ensuring that SQL queries are executed with minimal resource consumption while preserving their semantic correctness. Traditionally, optimization is regarded as a task handled internally by the database management system (DBMS), where the optimizer transforms a parsed SQL query into the most efficient physical execution plan through a sequence of heuristic and cost-based decisions. However, the efficiency of query execution is not solely determined by the DBMS optimizer. The quality of the SQL query itself—how it is written or structured—plays a crucial role in defining the upper bound of optimization.

As illustrated in Figure 2, the overall optimization process can be divided into two complementary stages: developer-side optimization and DBMS-side optimization. At the developer side, SQL queries can be manually rewritten into more efficient forms through structural rewriting, access path optimization, and semantic simplification, often assisted by query tuning tools, workload monitors, and execution plan analyzers. Such rewritings eliminate redundant operations and provide the DBMS with a better starting point for internal optimization. On the DBMS side, query optimization proceeds through a sequence of heuristic and cost-based transformations. The heuristic (cost-independent) phase applies rule-based rewrites such as predicate push-down and subquery unnesting to simplify the logical plan. Subsequently, the cost-based physical optimization estimates cardinalities and operator costs, enumerates alternative execution plans, and selects the one with the lowest estimated cost. The resulting physical plan determines access methods, join orders, and parallelization strategies used during query execution. Figure 3 illustrates a motivating example in which two semantically equivalent SQL queries exhibit drastically different performance. In the costly query, the filtering predicate "`uuid LIKE '%0c'`" is placed in the `HAVING` clause, forcing the DBMS to first scan the entire table and perform aggregation before any filtering can be applied. As a result, a large number of unnecessary rows are aggregated and materialized into a temporary table, significantly increasing execution cost. Because such aggregation-level predicates are not guaranteed to be rewritten into early filters by the DBMS optimizer, this inefficiency persists despite semantic equivalence.

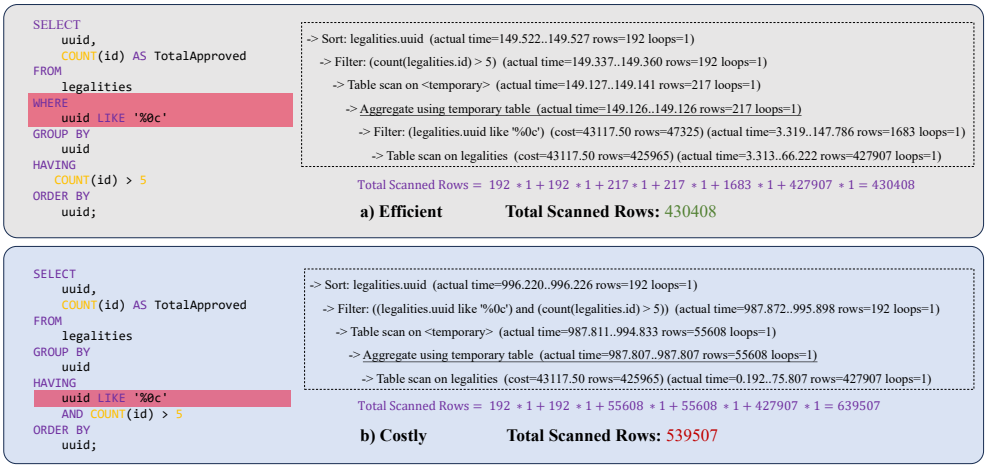


Fig. 3. **Semantically MySQL Equivalent Queries with Divergent Query Plans**

Query Plan. A query plan is essential for determining the efficiency of SQL query execution, as it defines the sequence of operations and data access methods used by the DBMS. The plan is generated by the query optimizer, which evaluates various execution strategies based on system resources and database state. The optimizer's goal is to select the most efficient plan by considering factors like query structure, data size, index availability, and operation costs. Typically, the optimization follows a cost-based model [5], where the plan with the lowest estimated cost is chosen [30].

A query plan is typically represented as a tree of operators, where each operator corresponds to an action such as scanning a table, filtering rows, or joining tables. Each node in the tree represents a relational operator, and the edges between nodes indicate the flow of data between operators. In practice, query plans can be inspected using the `EXPLAIN` or `EXPLAIN ANALYZE` command. Besides estimated costs, modern DBMSs report actual execution statistics, such as the number of rows processed by each operator, which reveal how much intermediate data is materialized and propagated through the plan. These statistics provide a more stable signal of query efficiency than end-to-end execution time, which can be affected by caching and system noise. As illustrated in Figure 3, the first query's query plan consists of a sequence of major operators, including a base table scan, an early filter on `uuid`, an aggregation operator that materializes a small temporary table, a post-aggregation filter, and a final sort. Because the filtering predicate is applied before aggregation, only a small number of rows are propagated to the aggregation and subsequent operators, resulting in limited intermediate data processing. In contrast, the second query's query plan contains a similar set of operators, but the absence of early filtering forces the aggregation operator to process and materialize a much larger intermediate result, which is then filtered and sorted at later stages. To quantify the efficiency difference between query plans, we use **Total Scanned Rows** as the cumulative number of actual rows processed by all operators in the execution plan. This metric captures the total amount of data examined and propagated during query execution, reflecting the cumulative cost of redundant scans and large intermediate results.

2.2 Efficiency-Oriented Text-to-SQL

Traditional Text-to-SQL aims to generate an executable SQL query q from a NL question x over a database schema $\mathcal{S}$, such that the execution result of q matches the ground-truth query q^*. Formally, the standard objective can be written as:

$$f_\theta(x, \mathcal{S}) \rightarrow q, \quad \text{s.t.} \quad \text{Exec}(q, \mathcal{S}) = \text{Exec}(q^*, \mathcal{S}),$$

Table 1. **Statistics of Representative Text-to-SQL Benchmarks.**

Dataset	Redundancy Measure		DB Complexity				Query Complexity		Evaluation Metrics		
	#Question–SQL	Annotation Type	#DBs	#Tables / DB	#Cols / Table	#Records / DB	Tables / Query	Selects / Query	EM	EX	VES
WikiSQL [72]	80,654	Synthetic	26,531	6.34	17	1	1.00	1.00	✓	✓	✗
Spider [69]	11,840	Manual	206	5.13	5.01	8,980	1.83	1.17	✓	✓	✗
SParC [70]	10,228	Manual	166	5.28	5.14	9,665	1.58	1.10	✓	✓	✗
CoSQL [68]	8,350	Manual	206	5.28	5.14	9,665	1.54	1.11	✓	✓	✗
CSpider [40]	11,840	Manual	206	5.13	5.01	8,980	1.83	1.17	✓	✓	✗
SQUALL [54]	11,276	Manual	2,108	1.91	9.18	71	1.22	1.29	✓	✓	✗
ViText2SQL [41]	9,693	Manual	166	5.28	5.14	9,665	1.17	1.12	✓	✓	✗
DuSQL [63]	25,003	Synthetic	208	4.04	5.29	20	1.49	1.25	✓	✓	✗
Spider–Syn [17]	1,034	Manual	166	5.28	5.14	9,665	1.68	1.17	✓	✓	✗
Spider–DK [18]	535	Manual	169	5.25	5.14	9,494	1.71	1.16	✓	✓	✗
Spider–Realistic [12]	508	Manual	166	5.28	5.14	9,665	1.79	1.21	✓	✓	✗
KaggleDBQA [29]	272	Manual	8	17	2.12	595,075	1.25	1.05	✓	✓	✗
PAUQ [3]	9,876	Manual	166	5.28	5.14	9,693	1.82	1.17	✓	✓	✗
BIRD [35]	10,962	Manual	80	7.64	7.14	549,335	2.07	1.09	✓	✓	✓
AmbiQT [4]	3,046	Synthetic	166	5.28	5.14	9,665	1.85	1.17	✓	✓	✗
BULL [71]	7,932	Manual	3	26	14.96	85,631	1.22	1.12	✓	✓	✗
EESQLBench (Ours)	213	Manual	69	7.75	7.45	3,432,599	3.55	2.88	✓	✓	✓

Note.

Redundancy Measure: indicates the size and diversity of question–SQL pairs; "Annotation Type" denotes whether SQLs are manually annotated (Manual) or automatically synthesized (Synthetic).
DB Complexity: measures the structural scale of databases, including number of databases (#DBs), average tables per DB, columns per table, and records per DB.
Query Complexity: describes query structural richness, including number of tables joined and average number of SELECT clauses.
Evaluation Metrics: EM (Exact Match), EX (Execution Accuracy), and VES (Valid Efficiency Score).

where f_θ denotes a Text-to-SQL model parameterized by θ, and $\text{Exec}(q, \mathcal{S})$ denotes the result of executing query q on schema $\mathcal{S}$.

However, this formulation only emphasizes **semantic correctness** without considering query efficiency. In practice, multiple semantically equivalent SQLs may lead to drastically different execution latencies due to join order, predicate placement, or index utilization. Therefore, we introduce the **Efficiency-Oriented Text-to-SQL** task, which extends the traditional objective by additionally optimizing for execution efficiency. Given a NL question x and schema $\mathcal{S}$, the goal is to generate a SQL query q_{opt} satisfying:

$$q_{\text{opt}} = \arg \min_{q \in Q(x,\mathcal{S})} \text{Cost}(q, \mathcal{S})$$
$$\text{s.t.} \quad \text{Exec}(q, \mathcal{S}) = \text{Exec}(q^*, \mathcal{S}),$$

where $Q(x, \mathcal{S})$ is the set of semantically correct SQL candidates produced by f_θ for $(x, \mathcal{S})$, and $\text{Cost}(q, \mathcal{S})$ represents the estimated or actual execution cost of query q on $\mathcal{S}$. Intuitively, the model should not only ensure that q_{opt} yields correct results but also that it achieves minimal execution time or resource consumption under the same semantic constraints.

2.3 Challenges

While existing Text-to-SQL benchmarks have significantly promoted progress in semantic correctness, evaluating Efficiency-Oriented Text-to-SQL models remains an open and underexplored challenge. Table 1 shows the statistics of representative Text-to-SQL benchmarks. We summarize two fundamental challenges in building and evaluating Efficiency-Oriented Text-to-SQL systems.

Challenge #1: Lack of Efficiency-Oriented Benchmarks. Most existing benchmarks are primarily designed to assess semantic correctness of generated SQL queries, rather than how efficiently they execute. As shown in Table 1, mainstream datasets such as Spider [69] and CoSQL [68] exhibit very limited structural complexity: each query involves on average fewer than two tables and barely one `SELECT` clause. Such simple patterns produce near-identical query plans across implementations, leaving no room to reveal efficiency differences among semantically equivalent SQLs. Even larger datasets, including BIRD [35] (80 databases with ~7.6 tables per database) primarily emphasize schema diversity and question coverage rather than query complexity. Although they contain millions of database records, their question–SQL pairs still favor simple retrieval forms—flat

joins or shallow filters—while rarely including multi-join, nested, or aggregation-heavy queries that are sensitive to join order, predicate pushdown, or index selection. To genuinely evaluate query efficiency, a benchmark must be designed with efficiency-aware constraints at three levels: (1) *Schema level* — large, interconnected schemas that permit multiple semantically equivalent join paths; (2) *Instance level* — heterogeneous table scales and skewed value distributions that amplify cost gaps among alternative plans; and (3) *Query level* — diverse SQL variants involving nested subqueries, redundant joins, and compound predicates that stress the optimizer. Without coordinating these three dimensions, efficiency cannot be meaningfully or stably reflected in benchmark evaluation. Moreover, current benchmarks lack human-optimized SQL references that can serve as efficiency baselines. Without manually crafted high-efficiency queries, it is impossible to quantify how far a model-generated SQL deviates from expert-level performance.

Challenge #2: Unstable and Unreliable Efficiency Evaluation. Recent work, such as the BIRD [35] benchmark, introduces the Valid Efficiency Score (VES), which measures the relative runtime of a model-generated SQL query against a reference query provided in the dataset. However, it is worth noting that the reference SQL in BIRD is not necessarily the result of careful, expert-level manual optimization, which limits its reliability as a gold standard for efficiency comparison. More fundamentally, runtime-based metrics remain highly sensitive to external factors such as caching effects, buffer states, query plan caching, and concurrent system load, which can significantly distort the observed efficiency relationship between two semantically equivalent queries. Therefore, we require more stable and intrinsic metrics to evaluate query efficiency independently of fluctuating runtime conditions.

2.4 Our Solutions

To address the challenges identified above, we propose a unified framework for Efficiency-Oriented Text-to-SQL evaluation that combines an efficiency-aware benchmark with stable, plan-based efficiency metrics.

Solution to Challenge #1: An Efficiency-Oriented Benchmark. To address the first challenge, we construct EESQLBench, which offers the first systematic evaluation of execution efficiency in Text-to-SQL systems. Each NL question is paired with a human-written canonical SQL statement that captures expert-level optimization. To make efficiency differences observable and measurable, EESQLBench is designed with explicit efficiency-aware constraints at the schema, instance, and query levels, ensuring that semantically equivalent SQL queries can induce substantially different execution plans and execution costs. At the schema level, we enhance real-world schemas by repairing missing foreign-key relationships and introducing mixed indexing strategies, enabling multiple semantically equivalent join paths and distinct execution plans. At the instance level, we apply controlled data augmentation with imbalanced table sizes and skewed value frequencies, making execution cost sensitive to predicate placement and join order. At the query level, we adopt a reverse generation strategy that first synthesizes complex SQL queries and then constructs corresponding NL questions. Specifically, we leverage DBMS fuzzing-inspired query generators to produce a large pool of structurally diverse and complex SQL queries, including nested subqueries, aggregation-heavy patterns, redundant joins, and alternative predicate placements that stress the optimizer. From this pool, we identify efficiency-critical queries whose execution costs can be substantially reduced through structural rewriting, and retain those cases after automated analysis and expert verification. Through these coordinated designs, EESQLBench exposes efficiency gaps that cannot be revealed by existing Text-to-SQL benchmarks and provides a principled testbed for evaluating efficiency-oriented Text-to-SQL systems.

Solution to Challenge #2: Stable Cost-Based Efficiency Metrics. To overcome the instability of runtime-based evaluation, we adopt query-plan-level cost metrics that reflect intrinsic execution

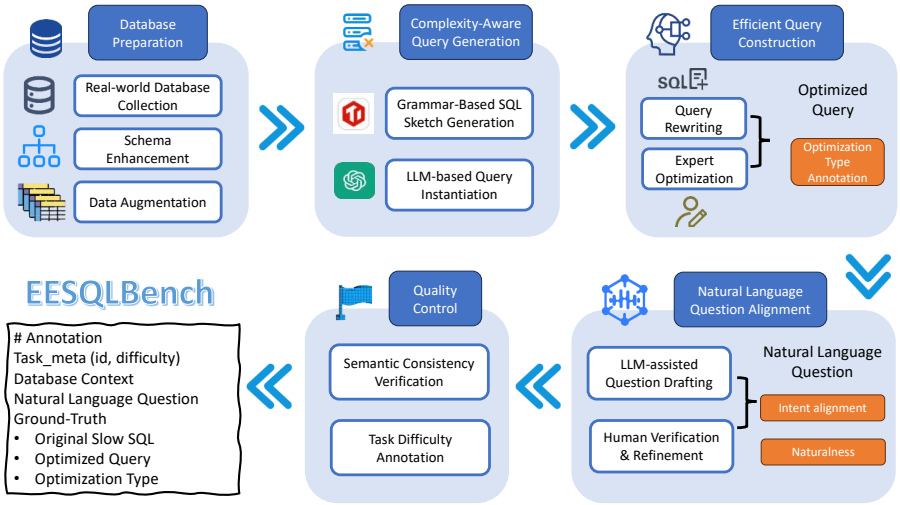


Fig. 4. **Overview of EESQLBench Construction Pipeline**

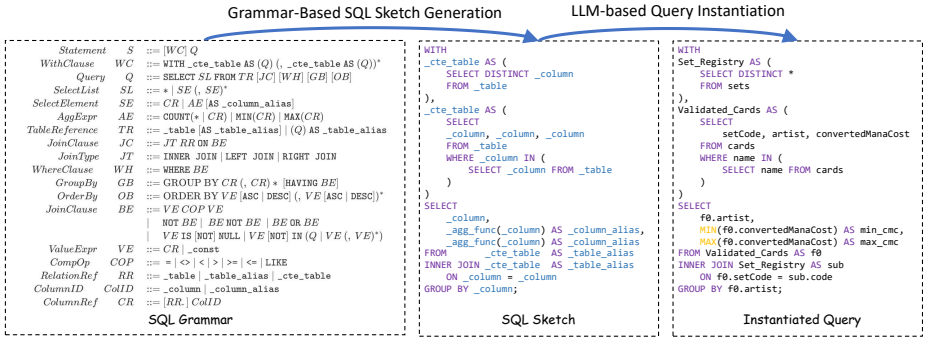
behavior while remaining insensitive to system-level noise. Instead of end-to-end latency, we measure query efficiency using **Total Scanned Rows**, defined as the sum of actual rows processed by all physical operators in the execution plan, as reported by `EXPLAIN ANALYZE`. This metric directly captures the amount of data examined and propagated during execution, providing a plan-grounded proxy for query execution cost. Figure 3 shows that, although the two semantically equivalent queries produce identical results, their execution plans process substantially different numbers of rows at intermediate operators, resulting in Total Scanned Rows of 430,408 and 539,507, respectively. Unlike execution latency, operator-level scanned row counts are deterministic under identical schemas and data instances, making Total Scanned Rows stable and reproducible across repeated executions. Based on this cost proxy, we further design **Cost Reachability (CR)** to quantify relative efficiency gaps against expert-optimized SQL, and **Acceptable Reachability at k (AR@k)** to measure how often model-generated queries meet practical efficiency thresholds.

3 EESQLBench

In this section, we introduce the construction of EESQLBench, the first benchmark for evaluating Efficiency-Oriented Text-to-SQL tasks. Figure 4 illustrates the entire process involved in building our benchmark. Starting from real-world databases, we enhance database schemas and data instances to expose performance-sensitive behaviors, and generate complex SQL queries that admit substantial optimization opportunities. For each generated query, we further curate a semantically equivalent but efficiency-optimized canonical SQL via automated rewriting and expert refinement, and align it with a natural language (NL) question. More details will be introduced in the following subsections.

3.1 Benchmark Construction

3.1.1 Database Preparation. We prepare the databases of EESQLBench by building upon real-world relational databases collected from the BIRD [35] benchmark. We choose BIRD [35] because it covers cross-domain scenarios spanning diverse application areas, including e-commerce, finance, and healthcare, and provides large-scale tables derived from real-world data with rich relational structures and realistic data distributions. Based on these databases, we enhance the original schemas to better support efficiency-oriented evaluation. In particular, we explicitly repair missing or implicit foreign-key relationships, which enables multiple alternative join paths among related



Note: We adopt a uniform naming convention where all placeholders are denoted by identifiers prefixed with an underscore (_), indicating schema-dependent holes to be instantiated later.

Fig. 5. **Two-Stage Query Generation via Grammar-Based SQL Sketching and LLM-Based Instantiation**

tables and allows semantically equivalent SQL queries to be expressed in structurally different forms. We further apply mixed indexing strategies by randomly indexing a subset of foreign-key and attribute columns, so that different join orders and access paths can lead to distinct execution plans. At the instance level, we enlarge the databases by systematically inserting additional rows into existing tables while strictly preserving the original schemas, data types, and foreign-key constraints. For each table, new tuples are generated by sampling valid key values from referenced tables and duplicating existing records with controlled variations on non-key attributes. For selected predicate-relevant attributes, we further introduce controlled value-frequency skew during tuple generation. Specifically, we randomly sample existing values from the original database and designate 20% of the distinct values as high-frequency values. When generating new tuples, these high-frequency values are oversampled so that the added tuples follow an approximate 80/20 split between frequent and non-frequent values. This strategy creates controlled predicate selectivity to amplify efficiency differences among semantically equivalent queries.

3.1.2 Complexity-Aware Query Generation. After preparing databases, we synthesize SQL queries that are both structurally complex and optimization-sensitive. We adopt a reverse construction strategy: instead of starting from NL and hoping a generator produces sufficiently challenging SQL, we first generate complex SQL candidates and then align them with NL questions. Our design is inspired by DBMS fuzzing, which can explore a wide spectrum of query structures (e.g., deep nesting, multi-join chains, and aggregation patterns) that are useful for stress-testing optimizer behavior. However, off-the-shelf fuzzers are not directly applicable: they often produce ill-typed, semantically invalid, or practically meaningless queries. On the other hand, prompting an LLM can improve semantic plausibility and executability, but the generated SQL is often overly "clean" and repetitive due to pretraining biases, favoring simple join patterns and shallow nesting. As a result, neither approach alone can simultaneously provide high structural diversity and semantic validity. To balance these two objectives, we introduce a *SQL Sketch* mechanism that decouples *query structure* from *schema-specific semantics*. As illustrated in Figure 5, query generation proceeds in two stages: (i) grammar-based SQL sketch generation, which generates the high-level relational structure of a query, and (ii) LLM-based instantiation, which binds schema-dependent elements under explicit constraints.

(1) Grammar-Based SQL Sketch Generation. We firstly generate SQL sketches by sampling from a context-free grammar that specifies optimization-sensitive query structures. We implement this grammar as shown in Figure 5 and use GO-RANDGEN [45] as a grammar-based generator to derive grammar-valid sketches, covering patterns such as multi-table joins, nested subqueries, common table expressions (CTEs), and aggregations with HAVING clauses. By construction, each

sketch fixes the high-level composition and nesting of relational operators, while abstracting away schema-specific details. Concretely, schema-dependent elements—including table names, column references, join predicates, and literal constants—are replaced with typed placeholders (denoted by identifiers prefixed with an underscore, e.g., `_table`, `_cte_table`, `_column`, `_agg_func`, `_const`). This sketch-based design ensures structural correctness and diversity at the query level, while deferring schema binding and semantic instantiation to a later stage.

(2) LLM-based Query Instantiation. Given a target database, we prompt an LLM to instantiate each sketch by filling placeholders with concrete tables, columns, predicates, and values. To improve semantic plausibility and controllability, the instantiation prompt includes the following components:

- **Task Instruction:** Instructs the LLM to fill the provided SQL sketch into an executable query that corresponds to a meaningful analysis intent, rather than producing arbitrary or trivial SQL.
- **Database Schema:** Provides the full `CREATE TABLE` statements (table names, columns, types, primary keys, and foreign keys) so that the LLM can select type-consistent columns and produce join conditions aligned with the schema constraints.
- **Database Values:** Samples representative values for a few columns (e.g., categorical values, typical timestamps, common IDs) to ground predicate generation, reducing ill-formed conditions (e.g., comparing strings to numeric columns) and improving contextual relevance.
- **Few-shot Examples:** Provides a small number of example queries following the same formatting and style, helping the LLM adhere to the desired SQL conventions (e.g., aliasing, indentation, and aggregation style).

The LLM is explicitly instructed to satisfy typing constraints and foreign-key connectivity when instantiating placeholders, producing executable SQL candidates.

Finally, we validate each instantiated query through execution. Queries that fail to parse, violate schema constraints, or produce empty results are discarded. The retained queries therefore form a large pool of complex and semantically meaningful SQL instances that admit substantial optimization opportunities and support subsequent canonicalization and question alignment. This hybrid construction also mitigates potential bias from LLM assistance. Since high-level query structures are determined by grammar-based sketches rather than free-form LLM generation, the model has limited influence over the structural distribution of benchmark queries. Together with execution-based validation, this design reduces the risk that the benchmark reflects superficial artifacts of LLM generation style, and instead grounds it in meaningful and executable SQL workloads.

3.1.3 Efficient Query Construction. For each synthesized SQL query, our goal is to construct a semantically equivalent but execution-efficient canonical query that represents expert-level optimization. To this end, we adopt a two-stage optimization pipeline that combines automated query rewriting with expert refinement, and explicitly annotate the applied optimization types. As a first step, we apply an automated query rewriting procedure to identify candidate efficiency improvements. Specifically, we leverage WeTune [65], a rule-based query rewriting framework, to generate semantically equivalent query variants through systematic application of rewriting rules. For each original query, we evaluate the rewritten variants and retain those that exhibit reduced execution cost. Building upon the automatically rewritten candidates, database experts further refine the queries to produce a final optimized canonical SQL. In this project, we engage three full-time Text-to-SQL experts, all of whom are Ph.D. students with formal training in hands-on experience in SQL optimization and query tuning. The entire expert refinement process for all queries takes approximately 80 person-hours in total. During this stage, experts manually inspect query structures and execution plans, and apply domain knowledge to introduce deeper optimizations that are difficult to fully capture by rule-based systems. In addition, we explicitly

annotate the primary optimization types applied to each query pair. Each query pair is labeled with one or more optimization categories indicating where efficiency gains are achieved, such as *join reordering*, *predicate pushdown*, *subquery elimination*, and *aggregation rewriting*. For example, if an optimized query achieves lower execution cost by pushing a filtering predicate from an outer query into an inner subquery before aggregation, it is annotated with *predicate pushdown*. These annotations enable fine-grained analysis of Text-to-SQL systems' optimization capabilities.

3.1.4 Natural Language Question Alignment. After constructing pairs of semantically equivalent original and optimized SQL queries, we align them with NL questions that faithfully capture the underlying analytical intent. The key design principle of this stage is to ensure fairness: each NL question should reflect the high-level user intent of the query, without encoding any SQL-specific structures or optimization decisions. To achieve this, we generate NL questions based on the shared semantics of each original–optimized query pair, rather than directly mirroring either concrete SQL formulation. Specifically, we adopt a back-translation strategy in which an LLM is prompted to translate a given SQL query into a concise and natural NL question. Importantly, the prompt provides both the original SQL and the optimized canonical SQL as semantic references, and explicitly instructs the model to generate a question that is simultaneously satisfied by both queries. Concretely, the prompt is constructed with the following components:

- **Task Instruction.** The model is instructed to generate a NL question that describes the analytical goal of the queries—i.e., what information is requested—rather than how the result is computed.
- **Dual Semantic References.** Both the original SQL query and its optimized canonical counterpart are provided as semantic references. The model is instructed to infer the common analytical intent underlying the two queries and generate a single NL question that is valid for both, treating the SQLs as black-box specifications of the result.
- **Implementation Abstraction Constraint.** The prompt explicitly prohibits mentioning SQL-specific constructs, such as joins, subqueries, aggregation operators, execution order, or optimization strategies, and requires the question to be phrased in natural, user-centric terms.
- **Semantic Equivalence Constraint.** The generated question must remain answerable by any semantically equivalent SQL formulation, ensuring neutrality with respect to different query structures and execution strategies.

Following generation, all NL questions undergo a human verification and refinement step. Three experts check each question for semantic alignment with both SQL variants, clarity, and naturalness, and revise ambiguous or overly verbose descriptions when necessary. As a result, each NL question in EESQLBench represents a realistic, user-facing query intent, and that can be correctly answered by multiple semantically equivalent SQL queries. This alignment strategy enables EESQLBench to simultaneously evaluate (i) semantic correctness, by checking whether a model-generated SQL answers the NL question correctly, and (ii) efficiency awareness, by comparing the execution efficiency of the generated SQL against the expert-optimized canonical query.

3.1.5 Quality Control and Annotations. In the final stage, we conduct quality control for each Text-to-SQL task to ensure the correctness, consistency, and reliability of the benchmark. This stage consists of semantic consistency verification and task difficulty annotation.

(1) Semantic consistency verification. We first verify the semantic consistency of each task across both SQL queries and the corresponding NL question. Specifically, we check that the original SQL query and the optimized SQL query are semantically equivalent by executing them under the same database configuration and comparing their result sets. Meanwhile, we require the optimized query to consistently achieve lower execution cost, ensuring the presence of a genuine and stable

efficiency gap. We further verify that the NL question can be correctly answered by both SQL queries, confirming that it accurately captures their shared analytical intent.

(2) Task difficulty annotation. After passing semantic consistency verification, we annotate the difficulty of each Text-to-SQL instance using a rule-based human scoring scheme along three complementary dimensions:

- **Schema Complexity:** the number of involved tables, join relationships, and foreign-key dependencies;
- **Question Understanding:** the presence of multiple conditions, implicit constraints, or required reasoning in the NL question;
- **SQL Complexity:** the structural and operator-level complexity of the query, such as subqueries, aggregations, `HAVING` clauses, and CTEs.

Each dimension is graded according to predefined criteria specified in the annotation guidelines [10], and the final difficulty label (e.g., Simple, Medium, or Hard) is determined by aggregating the scores across all dimensions.

All verification and annotation procedures are conducted by three domain experts. The use of three annotators follows standard practice in empirical software engineering to balance annotation quality and cost, and is consistent with prior benchmarks [38, 67]. Two experts independently evaluate each instance following a unified guideline, and disagreements are resolved by a third expert acting as an adjudicator. We measure inter-annotator agreement using Cohen's kappa coefficient, obtaining an overall kappa score of 0.92, which indicates high annotation consistency and reliable quality. As a result, each task in EESQLBench is accompanied by structured annotations, including the database context, paired original and optimized SQL queries, a shared NL question, explicit optimization-type labels, and a difficulty level. These annotations provide a reliable foundation for analyzing both semantic correctness and optimization awareness of Text-to-SQL systems. Due to the space limitations, a detailed annotation guideline, including annotator selection criteria and bias-mitigation protocols, is provided at [14].

3.2 Statistics

We present a statistical analysis of EESQLBench and compare it with representative Text-to-SQL benchmarks in Table 1. EESQLBench is constructed on MySQL as the execution backend. Overall, EESQLBench contains 213 Text-to-SQL tasks over 69 real-world databases. All tasks are manually curated and paired with expert-optimized canonical SQL queries, enabling direct efficiency comparison between model-generated SQL and expert-level solutions. Following a unified guideline (Section 3.1.5), we further label each task with a difficulty level, resulting in 55 Simple, 85 Medium, and 73 Hard tasks.

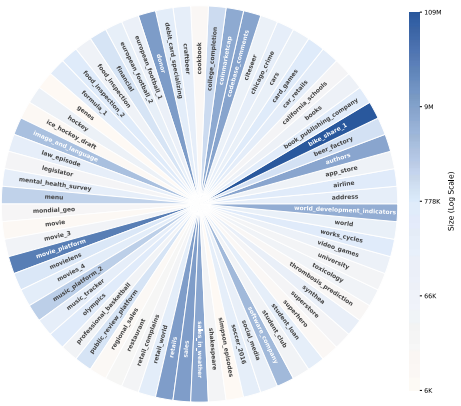


Fig. 6. **Distributions of Database Size in EESQLBench (Log Scale)**

Figure 6 shows the distribution of database sizes in EESQLBench on a log scale; for completeness, we provide a detailed breakdown of record counts in our online documentation [8]. EESQLBench covers databases with highly heterogeneous instance scales, spanning from 148kB to 4.4 GB. In addition, EESQLBench is explicitly designed to be efficiency-aware at three levels. At the *schema level*, each database contains 7.75 tables on average (and up to 65), with repaired foreign-key relationships that enable multiple semantically equivalent join paths. At the *instance level*, each database contains 3,432,599 records on average, and we further

introduce controlled table-scale imbalance and value-frequency skew to amplify cost gaps among semantically equivalent queries. At the *query level*, each task involves 3.55 tables on average (up to 11) and 2.88 subqueries on average (up to 8), resulting in a substantial portion of multi-join and nested queries that admit non-trivial optimization opportunities.

4 Efficiency-Oriented Evaluation Metrics

Evaluating Efficiency-Oriented Text-to-SQL requires metrics that not only assess semantic correctness but also reliably quantify query execution efficiency. While execution latency is a natural indicator of efficiency, it is highly sensitive to external factors, making it unstable and difficult to reproduce. Therefore, we adopt query-plan-based cost metrics, which provide a more stable proxy for intrinsic query efficiency.

Semantic Correctness. We first evaluate whether a generated query $\hat{q}$ is semantically equivalent to the canonical query q^*. Following prior works [35, 69], semantic equivalence is determined by comparing execution results:

$$\mathbf{1}(q^*, \hat{q}) = \begin{cases} 1, & \text{if } \mathrm{Exec}(\hat{q}, \mathcal{S}) = \mathrm{Exec}(q^*, \mathcal{S}), \\ 0, & \text{otherwise.} \end{cases}$$

Efficiency-related metrics are computed only on semantically correct queries (i.e., those with $\mathbf{1}(q^*, \hat{q}) = 1$), ensuring that efficiency evaluation is not confounded by incorrect query semantics.

Cost Reachability (CR). To measure a model-generated query's proximity to expert-level efficiency, we introduce *Cost Reachability (CR)*. For each semantically correct query pair $(q^*, \hat{q})$, we define:

$$CR_n = \frac{\mathrm{Cost}(q^*, \mathcal{S})}{\mathrm{Cost}(\hat{q}, \mathcal{S})},$$

where $\mathrm{Cost}(q, \mathcal{S})$ denotes the execution cost of query q on schema $\mathcal{S}$, obtained from the query execution plan. A larger CR_n indicates that the generated query $\hat{q}$ is more efficient relative to the canonical solution q^*, while $CR_n < 1$ implies inferior efficiency.

To mitigate the influence of extreme outliers, we follow prior cost-based evaluation practices [35] and apply a square-root transformation to obtain a stabilized score:

$$R_n = \sqrt{CR_n}.$$

The overall Cost Reachability is then computed as:

$$CR_{\mathrm{avg}} = \frac{1}{\sum_{n=1}^{N} \mathbf{1}(q_n^*, \hat{q}_n)} \sum_{n=1}^{N} \mathbf{1}(q_n^*, \hat{q}_n) \cdot R_n,$$

where N is the total number of evaluation instances. For brevity, we use CR to refer to the aggregated score CR_{avg} in the remainder of the paper.

Acceptable Reachability at k (AR@k). While CR_{avg} characterizes the efficiency gap *conditioned on semantic correctness*, it does not reflect how often a model produces queries that are both correct and practically efficient. To capture this end-to-end utility, we define *Acceptable Reachability at k* (AR@k) as:

$$AR@k = \frac{1}{N} \sum_{n=1}^{N} \mathbf{1}(q_n^*, \hat{q}_n) \cdot \mathbf{1}(CR_n \geq k),$$

where $k \in (0, 1]$ is a user-defined threshold. Intuitively, AR@k measures the fraction of all test instances for which the generated SQL is semantically correct and its execution cost does not

exceed $1/k$ times that of the canonical solution. For example, $k = 0.5$ corresponds to accepting queries that are at most twice as expensive as the expert-optimized SQL, while $k = 1.0$ only counts queries whose execution cost is no higher than the canonical solution (i.e., at least as efficient as the expert-optimized SQL).

Together, CR and AR@k provide complementary perspectives on efficiency: CR measures *how close* a model can get to expert-level efficiency on average (given correctness), while AR@k measures *how often* it produces SQL that is both correct and meets a practical efficiency threshold.

5 Experiment

We aim to answer the following key research questions (RQs) to evaluate the effectiveness of EESQLBench for assessing both correctness and efficiency in Text-to-SQL systems:

- **RQ1.** How well can existing LLMs generate Text-to-SQL queries in terms of both semantic correctness and execution efficiency under EESQLBench? (Section 5.2)
- **RQ2.** What are the major failure modes that lead to efficiency degradation in LLM-generated SQL, and what optimization patterns do experts apply in canonical solutions? (Section 5.3)

5.1 Experimental Setup

Models and Prompting Strategy. We evaluate six representative LLMs including four open-source models: SQLCoder-7B-2 [55], CodeLlama-7B-Instruct-hf [52], DeepSeek-Coder-6.7B-Instruct [21], and DeepSeek-R1-Distill-Llama-8B [11], as well as two closed-source models: GPT-5.2 [20] and Gemini-2.5-Pro [48]. These models are selected based on their strong performance on Text-to-SQL tasks; each has been pre-trained or fine-tuned with SQL-related data, equipping them with advanced SQL reasoning capabilities. Among the evaluated models, Gemini-2.5-flash additionally supports tool-use functionalities, allowing it to autonomously consult external resources such as documentation, schemas or reference implementations. All experiments are conducted in deterministic settings with temperature fixed to 0.0 to reduce randomness.

Since our primary objective is to establish a standardized capability benchmark that isolates and evaluates LLMs' intrinsic SQL generation abilities, we do not include full agentic systems (e.g., execution-feedback-driven or tool-augmented pipelines). Although agentic systems can be powerful, they introduce confounding factors such as multi-component coordination, tool orchestration strategies, and schema introspection mechanisms, which obscure the underlying generative capacity of the LLM and hinder fair comparison across models.

For prompting, we adopt the in-context learning strategy DAIL-SQL [19], which has achieved state-of-the-art performance on the BIRD benchmark as of October 2025. We strictly follow the original DAIL-SQL setup, including prompt templates and decoding configurations.

Metrics. For correctness, we adopt the widely used Execution Accuracy (EX) [35, 69], defined as the proportion of NL inputs whose generated SQL executes to the same result as the canonical SQL. For efficiency, we employ two cost-based metrics introduced in Section 4: Cost Reachability (CR) and Acceptable Reachability at k (AR@k), with k set to 0.8 and 1.0 in our experiments. In addition to cost-based evaluation, we also report the Valid Efficiency Score (VES) [35], a runtime-based metric. VES evaluates efficiency by comparing the execution speed of a valid generated SQL query against its canonical reference, and then averaging the relative efficiency scores across all instances.

Environment. We perform our experiments on a workstation an 8-core "11th Gen Intel(R) Core(TM) i7-11700@2.50GHz" processor, 64GB of memory, and NVIDIA RTX A6000 with 48GB of VRAM running Ubuntu 22.04.1 LTS. For open-source LLMs, we deploy a local API server based on vLLM [28] which is a unified library for LLM serving and inference. For GPT-5.2 and Gemini-2.5-Pro, we query the respective official APIs with matching decoding settings.

5.2 RQ1: Overall Results

Table 2 shows the overall performance of six LLMs on EESQLBench across three difficulty levels.

Correctness. Overall, correctness varies substantially across models, suggesting that EESQLBench remains challenging even for strong closed-source LLMs. Gemini-2.5-pro consistently achieves the highest EX across all levels (Simple: 74.5%, Medium: 56.5%, Hard: 65.8%), followed by GPT-5.2 (69.1%, 50.6%, 60.3%). Among open-source models, DeepSeek-Coder attains moderate EX (40.0%, 22.4%, 19.2%), while CodeLlama and DeepSeek-R1 lag behind, and SQLCoder fails on most instances (EX $\leq$ 1.4%).

Efficiency among correct outputs (CR) and its relation to EX. CR reveals a markedly different ranking from EX, confirming that semantic correctness is not a sufficient proxy for efficiency. For example, on Medium, CodeLlama achieves the highest CR (1.210) despite a low EX (12.9%), indicating that when it succeeds, it can produce queries that are even more efficient than the canonical solutions. A similar pattern holds on Simple where DeepSeek-R1 attains the best CR (0.815) with low EX (16.4%), while GPT-5.2 has relatively high EX but only mid-level CR (0.621). Notably, CR values above 1.0 (e.g., CodeLlama on Medium/Hard) suggest that the "canonical" query can be further optimized for a small subset of cases, highlighting the value of cost-based analysis for uncovering additional optimization opportunities.

End-to-end "correct & efficient" comparison. We use AR@k as the most direct and fair metric for comparing models on the Efficiency-Oriented Text-to-SQL task, since it measures the fraction of all test instances where a model produces a SQL query that is both semantically correct and practically efficient under a cost threshold. Gemini-2.5-pro is the strongest model in terms of AR across all levels under both thresholds: AR@1 reaches 18.2% (Simple), 20.0% (Medium), and 21.9% (Hard), and AR@0.8 reaches 21.8%, 27.1%, and 23.3%, respectively. For open-source models, AR@1 remains low overall (typically $\leq$ 9.1% on Simple and $\leq$ 12.9% on Medium), and several models collapse on Hard (e.g., DeepSeek-R1: AR@1=0.0%). Overall, these results indicate that producing *both* correct and near-expert efficient SQL remains challenging, especially for open-source LLMs.

Consistency between CR and VES. Due to space limitations, we do not include VES in Table 2 and instead report it in our online leaderboard. To further validate our plan-cost proxy, we conduct an additional reliability and consistency study comparing cost-based metrics (CR) with runtime-based VES. Specifically, for each evaluated model and each test instance, we repeat query execution for N=10 runs under a fixed DBMS configuration, and report (i) the *relative standard deviation* (RSD) to quantify metric stability across repeated runs (lower is more stable), and (ii) the Pearson correlation coefficient between runtime and plan-cost signals to quantify their linear agreement. Across all models, we observe that CR (cost-based) exhibits substantially lower variance than VES (runtime-based): the average RSD of CR is 0, whereas the average RSD of VES is 15.77%. Moreover, Total Scanned Rows shows a strong positive correlation with runtime across instances, with an average Pearson correlation of 0.83, indicating that our plan-cost proxy remains consistent with runtime behavior while being more reproducible.

Statistical significance. To further examine whether the observed efficiency differences are robust across tasks, we conduct paired statistical analysis on the main efficiency comparisons. Using the Wilcoxon signed-rank test over task-level efficiency results, we find that the observed differences are statistically significant across all settings ($p < 0.001$). We further compute 95% confidence intervals and Cliff's delta, obtaining a medium effect size (approximately $\delta = 0.52$), which indicates that the observed efficiency gaps are not only statistically reliable but also practically meaningful.

Table 2. **Evaluation Results of Each Model for Efficiency-Oriented Text-to-SQL on EESQLBench.**

Model	Simple				Medium				Hard			
	EX(%)	CR	AR@0.8(%)	AR@1(%)	EX(%)	CR	AR@0.8(%)	AR@1(%)	EX(%)	CR	AR@0.8(%)	AR@1(%)
GPT-5.2	69.1	0.621	10.9	5.5	50.6	0.745	21.2	12.9	60.3	0.767	21.9	17.8
Gemini-2.5-pro	**74.5**	0.683	**21.8**	**18.2**	**56.5**	0.832	**27.1**	**20.0**	**65.8**	0.739	**23.3**	**21.9**
CodeLlama	29.1	0.753	5.5	3.6	12.9	**1.210**	9.4	5.9	4.1	**1.076**	2.7	1.4
DeepSeek-R1	16.4	**0.815**	10.9	9.1	4.7	0.902	2.4	2.4	1.4	0.205	0.0	0.0
DeepSeek-Coder	40.0	0.519	9.1	3.6	22.4	0.949	14.1	12.9	19.2	0.881	12.3	9.6
SQLCoder	0.0	0.000	0.0	0.0	1.2	0.990	1.2	0.0	1.4	2.113	1.4	1.4

Note:
EX is Execution Accuracy (*higher is better*).
CR is Cost Reachability computed on semantically correct instances (*higher is better*; 1.0 indicates cost parity with the canonical SQL).
AR@k is the fraction of all test instances whose generated SQL is both correct and within the cost threshold (*higher is better*).

Summary: Gemini-2.5-pro delivers the best end-to-end utility, while CodeLlama and DeepSeek-R1 exhibit strong optimization behavior on the subset of correct queries despite low EX. The discrepancy between EX and CR/AR further confirms that higher semantic accuracy does not necessarily imply better efficiency, motivating the need for efficiency-oriented benchmarks.

5.3 RQ2: Efficiency Analysis

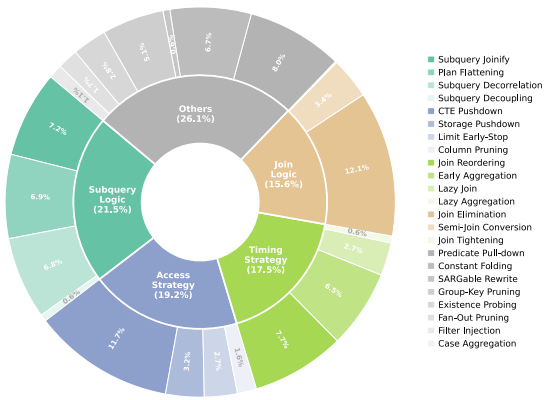


Fig. 7. **Inefficiency Patterns**

To understand why LLM-generated SQL becomes inefficient and how experts improve it, we conduct a qualitative efficiency analysis on semantically correct yet inefficient generations. Specifically, we select all instances with CR_n<1.0 (in total 878 cases), and ask three SQL experts to independently perform open coding [26] by jointly inspecting the generated SQL, the expert canonical SQL, and their execution plans. Following a standard open-coding protocol, each expert first independently proposes fine-grained labels for recurring inefficiency patterns, after which the experts iteratively compare, refine, and merge overlapping labels through discussion. After resolving disagreements, we consolidate recurring inefficiency patterns into 5 high-level themes (and 23 fine-grained labels), as shown in Figure 7. Due to space limitations, we summarize the categories and key observations below, and provide additional examples and case studies in our GitHub repository [9]. We also release the detailed coding protocol and label definitions online to improve transparency and reproducibility [15].

E1. Subquery Logic (21.5%), LLMs tend to overuse nested and correlated subqueries, producing procedural, row-by-row execution (or repeated materialization) instead of set-based plans; experts typically flatten plans, decorrelate subqueries, or "joinify" existence checks to unlock optimizer search space. **E2. Access Strategy (19.2%)**, models often miss opportunities to push predicates into scans/CTEs, prune unused columns, or propagate `LIMIT` for early stop, which turns otherwise selective queries into heavy I/O workloads. **E3. Timing Strategy (17.5%)**, models frequently place joins/aggregations in a suboptimal order (e.g., joining before selective filtering), causing cardinality explosion; experts reorder joins, apply early aggregation, or delay expensive operators

(lazy join/aggregation) until after pruning. **E4. Join Logic (15.6%)**, inefficiency often stems from suboptimal join semantics and unnecessary join work, such as retaining redundant join branches that could be removed, using overly permissive join types that block optimization, or failing to convert existence-only joins into semi-joins, which inflates intermediate results and increases join cost. **E5. Others (26.1%)**, this category captures a long tail of operator-level and plan-shape inefficiencies. Typical cases include missed existence probing (using costly `DISTINCT`/`GROUP BY` where `EXISTS`/`IN` suffices), lack of group-key pruning, missing case aggregation, and filter injection failures in complex nested logic. Although individually less frequent, these patterns often trigger unnecessary materialization and inflate intermediate results, indicating limited awareness of operator semantics and plan-shape simplification.

Summary: The dominant inefficiency patterns align with four themes—subquery logic (E1), access strategy (E2), timing strategy (E3), and join logic (E4). Experts mitigate them via subquery unnesting, early pruning, timing-aware reordering, and join tightening/elimination.

6 Discussion

Implications. For Text-to-SQL users and practitioners, our findings highlight that producing a semantically correct query is not sufficient for good user experience: correct SQL can still be unnecessarily slow on complex databases. Therefore, when deploying Text-to-SQL in latency-sensitive settings, users should prefer models that deliver higher end-to-end "correct & efficient" rates (e.g., Gemini-2.5-pro), and consider adding lightweight efficiency check before execution. For LLM developers, the gap between EX and efficiency metrics indicates that improving correctness alone will not automatically yield efficient SQL; models need explicit signals and objectives for optimization behavior. This suggests incorporating efficiency-aware supervision, cost-aware reranking/rewarding, and training-time augmentation with optimization-sensitive structures to better align generation with DBMS performance.

Benchmark Choice and Practical Relevance. EESQLBench is built on BIRD [35] as its real-world foundation, since BIRD provides cross-domain databases with realistic schemas and data distributions, making it a suitable starting point for studying efficiency-oriented Text-to-SQL. Compared with traditional DBMS benchmarks such as TPC-H [60], which focus on measuring engine-level performance on a single synthetic retail schema with a fixed set of queries, EESQLBench targets a different problem: optimization gaps in LLM-generated SQL across diverse real-world databases. This distinction is important because our goal is not to benchmark the raw execution capability of a DBMS, but to evaluate whether a Text-to-SQL system can generate SQL that is not only semantically correct, but also efficient under realistic schema and workload settings.

This design also reflects the practical relevance of EESQLBench. In real database-backed applications such as business intelligence and analytics platforms, semantically correct but inefficient SQL queries can become recurring bottlenecks when issued repeatedly, increasing latency and resource consumption. Such inefficiencies are often difficult to detect under correctness-only evaluation, because a query may produce the right answer while still imposing unnecessary overhead on the database engine. From this perspective, EESQLBench should not be viewed as merely another benchmark dataset, but rather as an efficiency-oriented performance-testing infrastructure for LLM-generated SQL. It combines efficiency-sensitive tasks, strong human-validated reference queries, and plan-based testing oracles, enabling systematic and reproducible detection of inefficiency issues that would otherwise remain invisible under conventional Text-to-SQL evaluation.

Extensibility. Although the current version of EESQLBench is instantiated on BIRD-derived databases and evaluated on MySQL, its construction is not tied to a specific benchmark or DBMS.

The overall pipeline is modular, consisting of database preparation, complexity-aware query construction, expert refinement, and plan-based evaluation. Therefore, adapting EESQLBench to other benchmarks or real-world databases mainly requires replacing the underlying schemas and data while reusing the same workflow. This design makes EESQLBench extensible beyond its current setting and provides a foundation for broader efficiency-oriented evaluation in future work.

Threats to Validity. The first potential external threat is the scope of the tested LLMs. In our experiments, we evaluate six representative LLMs, including both open-source and closed-source models, as well as general and code LLMs. While this selection covers the dominant model families currently used in practice, it may not fully represent all emerging LLMs. Extending the benchmark to multi-step or agent-based systems is left for future work. Another external validity concern is the choice of DBMSs. Our benchmark is instantiated on MySQL, and execution behavior may differ across DBMSs due to variations in optimizers, cost models, and execution engines. Nevertheless, MySQL is among the most widely deployed relational databases in real-world applications, making our benchmark practically relevant. Moreover, the design principles of EESQLBench are DBMS-agnostic, and we plan to extend the benchmark to additional DBMSs in future work to further enhance its generality.

Internal validity may affected by the involvement of human experts in query optimization and annotation. The construction of optimized SQL queries and the assignment of optimization-type and difficulty labels may introduce subjective bias. To mitigate this threat, we engage three domain experts with substantial experience in SQL optimization and Text-to-SQL systems. Two experts independently perform verification and annotation following detailed guidelines, while disagreements are resolved by a third expert acting as an adjudicator. We further measure inter-annotator agreement using Cohen's kappa coefficient, indicating a high level of annotation consistency.

7 Related Work

Text-to-SQL Systems and Benchmarks. Text-to-SQL systems aim to automatically translate natural language questions into executable SQL queries, bridging the gap between non-expert users and relational databases [1, 23, 46]. This capability enables a wide range of real-world applications, including intelligent database assistants, automated analytics, and natural language interfaces for data exploration. With the rapid progress of LLMs, recent approaches have achieved significant improvements in SQL generation accuracy through in-context learning, decomposition strategies, and instruction fine-tuning [47, 56, 58, 61, 66]. These LLM-based methods are typically evaluated on widely adopted benchmarks such as WikiSQL [25] and Spider [69], which measure the semantic correctness of generated SQL queries against reference ground truths.

However, existing benchmarks focus solely on semantic correctness, overlooking the execution efficiency of generated SQLs. To address this gap, our paper introduces an Efficiency-Oriented Text-to-SQL Benchmark EESQLBench that jointly evaluates both correctness and execution efficiency through newly designed performance-aware metrics.

Performance Benchmarks and Efficiency Evaluation. In the database community, traditional benchmarks such as TPC-H [60] and TPC-DS [59] are designed to evaluate DBMS performance under standardized decision-support workloads. These benchmarks measure engine-level execution capability using fixed schemas and query templates, focusing on throughput, latency, and system scalability. Other benchmarks, such as the Join Order Benchmark [31] and CH-benCHmark [7], further study query optimization challenges and mixed workload settings. In parallel, recent work has begun to study the efficiency of LLM-generated code. Benchmarks such as COFFE [44], EffiBench [24], and ENAMEL [49] evaluate whether generated programs are not only functionally correct but also efficient.

Our work differs from prior database performance studies (e.g., TPC-H [60]) and software performance engineering work (e.g., COFFE [44]) in that they mainly evaluate DBMS behavior or general program efficiency, whereas EESQLBench evaluates the execution efficiency of LLM-generated SQL under semantic equivalence.

Query Rewriting. Query rewriting is a fundamental component of database query optimization, which transforms an existing SQL query into a semantically equivalent but more efficient logical form. Most prior works follow a rule-based paradigm. For example, WeTune [65] employs a rule generator and an SMT solver to synthesize and verify new rewriting rules, though its efficiency decreases when handling complex queries. Learned Rewrite [73] leverages Calcite's existing rule set and applies Monte Carlo Tree Search (MCTS) to navigate the optimal rule application order for performance improvement.

In contrast, our work—Efficiency-Oriented Text-to-SQL—shifts this optimization objective into the generation stage. Instead of rewriting existing SQL statements, our framework learns to generate SQL queries that are both semantically correct and execution-efficient from the outset. It internalizes the cost-awareness and structural reasoning traditionally handled by rewriting systems into the natural language–to–SQL generation process. This paradigm not only eliminates the need for post-hoc rewriting overhead but also avoids the potential semantic drift introduced during rule-based transformations. Furthermore, because Efficiency-Oriented Text-to-SQL directly generates optimized SQL across different database dialects, it does not rely on dialect-specific rewriting rules or external equivalence verification. As a result, it achieves both higher robustness and broader generalization compared with pipelines that perform Text-to-SQL generation followed by separate query rewriting. In addition to evaluating generation-stage efficiency, EESQLBench can also serve as a complementary benchmark for query rewriting systems. By providing paired semantically equivalent SQL queries with explicit efficiency gaps and canonical optimized references, EESQLBench enables systematic evaluation of rewriting methods in terms of their ability to close the efficiency gap toward expert-level performance.

8 Conclusion

In this paper, we introduce EESQLBench, an efficiency-oriented Text-to-SQL benchmark designed to assess both the correctness and efficiency of generated queries. We use metrics such as Cost Reachability (CR) and Acceptable Reachability at k (AR@k) to evaluate six representative LLMs, and the results show that while current models perform well in terms of correctness, their efficiency often lags behind, especially for more complex queries. These findings highlight the need for efficiency-aware evaluation metrics in future Text-to-SQL systems.

Data Availability

All evaluation results are publicly available on our leaderboard https://eesqlbench.github.io/, and the code and dataset are released at https://github.com/EESQLBench/EESQLBench.

Acknowledgments

We sincerely thank the anonymous reviewers for their valuable and insightful feedback. This work is supported by the Xinmiao Project of ZheJiang Provincial Applied Basic Research Program (Grant No. 2026XMZD032), the Zhejiang Pioneer (Jianbing) Project (2025C01198), the National Science Foundation of China (No. 62372398, No. 72342025), Zhejiang Provincial Science Foundation of China (No. LQK26F020002), and the Fundamental Research Funds for the Central Universities (No. 226-2025-00067). Lingfeng Bao is the corresponding author, and is also affiliated with the Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security.

References

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

[2] Qiushi Bai, Sadeem Alsudais, and Chen Li. 2023. Querybooster: Improving SQL performance using middleware services for human-centered query rewriting. *arXiv preprint arXiv:2305.08272* (2023).

[3] Daria Bakshandaeva, Oleg Somov, Ekaterina Dmitrieva, Vera Davydova, and Elena Tutubalina. 2022. PAUQ: Text-to-SQL in Russian. In *Findings of the Association for Computational Linguistics: EMNLP 2022*. 2355–2376.

[4] Adithya Bhaskar, Tushar Tomar, Ashutosh Sathe, and Sunita Sarawagi. 2023. Benchmarking and improving text-to-sql generation under ambiguity. *arXiv preprint arXiv:2310.13659* (2023).

[5] Surajit Chaudhuri. 1998. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 34–43.

[6] Edgar F Codd. 1974. *Seven steps to rendezvous with the casual user*. IBM Thomas J. Watson Research Division.

[7] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, et al. 2011. The mixed workload CH-benCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*. 1–6.

[8] EESQLBench Contributors. 2025. EESQLBench Database Size and Record Counts. https://github.com/EESQLBench/EESQLBench/blob/main/data/DatabaseSize.md. Accessed: 2026-01-29.

[9] EESQLBench Contributors. 2025. EESQLBench Inefficiency Pattern Annotation. https://github.com/EESQLBench/EESQLBench/blob/main/task/InefficiencyPattern.md. Accessed: 2026-01-29.

[10] EESQLBench Contributors. 2026. EESQLBench Difficulty Annotation Guidelines. https://github.com/EESQLBench/EESQLBench/blob/main/task/DifficultyAnnotationGuideline.md. Accessed: 2026-01-28.

[11] DeepSeek-AI. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *arXiv preprint arXiv:2501.12948* (2025). https://arxiv.org/abs/2501.12948

[12] Xiang Deng, Ahmed Hassan Awadallah, Christopher Meek, Oleksandr Polozov, Huan Sun, and Matthew Richardson. 2020. Structure-grounded pretraining for text-to-sql. *arXiv preprint arXiv:2010.12773* (2020).

[13] Haoran Ding, Zhaoguo Wang, Yicun Yang, Dexin Zhang, Zhenglin Xu, Haibo Chen, Ruzica Piskac, and Jinyang Li. 2023. Proving query equivalence using linear integer arithmetic. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–26.

[14] EESQLBench. 2026. EESQLBench Annotation Guidelines. https://github.com/EESQLBench/EESQLBench/blob/main/task/EESQLBench%20Annotation%20Guidelines.md. Accessed: 2026-01-29.

[15] EESQLBench. 2026. Open-Coding Protocol for Inefficiency Pattern Analysis (RQ2). https://github.com/EESQLBench/EESQLBench/blob/main/task/Open-Coding%20Protocol%20for%20Inefficiency%20Pattern%20Analysis%20(RQ2).md. Accessed: 2026-01-29.

[16] Ahmed Elgohary, Saghar Hosseini, and Ahmed Hassan Awadallah. 2020. Speak to your parser: Interactive text-to-SQL with natural language feedback. *arXiv preprint arXiv:2005.02539* (2020).

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

[18] Yujian Gan, Xinyun Chen, and Matthew Purver. 2021. Exploring underexplored limitations of cross-domain text-to-sql generalization. *arXiv preprint arXiv:2109.05157* (2021).

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

[20] GPT-5.2. [n. d.]. OpenAI API Model Documentation (GPT-5.2). OpenAI. https://platform.openai.com/docs/models Accessed: 2026-01-22.

[21] Daya Guo, Dejian Yang, Haowei Zhang, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024). https://arxiv.org/abs/2401.14196

[22] Fadi H Hazboun, Majdi Owda, and Amani Yousef Owda. 2021. A natural language interface to relational databases using an online analytic processing hypercube. *AI* 2, 4 (2021), 720–737.

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

[24] Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie M Zhang. 2024. Effibench: Benchmarking the efficiency of automatically generated code. *Advances in Neural Information Processing Systems* 37 (2024), 11506–11544.

[25] Wonseok Hwang, Jinyeong Yim, Seunghyun Park, and Minjoon Seo. 2019. A comprehensive exploration on wikisql with table-aware word contextualization. *arXiv preprint arXiv:1902.01069* (2019).

[26] Shahedul Hasan Khandkar. 2009. *Open Coding*. Technical Report. University of Calgary. No. 2009.

[27] Jan Kossmann, Thorsten Papenbrock, and Felix Naumann. 2022. Data dependencies for query optimization: a survey. *The VLDB Journal* 31, 1 (2022), 1–22.

[28] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[29] Chia-Hsuan Lee, Oleksandr Polozov, and Matthew Richardson. 2021. KaggleDBQA: Realistic evaluation of text-to-SQL parsers. *arXiv preprint arXiv:2106.11455* (2021).

[30] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[31] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the join order benchmark. *The VLDB Journal* 27, 5 (2018), 643–668.

[32] Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023. Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 13067–13075.

[33] Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024. Codes: Towards building open-source language models for text-to-sql. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–28.

[34] Jinyang Li, Binyuan Hui, Reynold Cheng, Bowen Qin, Chenhao Ma, Nan Huo, Fei Huang, Wenyu Du, Luo Si, and Yongbin Li. 2023. Graphix-t5: Mixing pre-trained transformers with graph-aware layers for text-to-sql parsing. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 37. 13076–13084.

[35] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2023. 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 (2023), 42330–42357.

[36] Shuqin Li, Kaibin Zhou, Zeyang Zhuang, Haofen Wang, and Jun Ma. 2023. Towards text-to-SQL over aggregate tables. *Data Intelligence* 5, 2 (2023), 457–474.

[37] Zhaodonghui Li, Haitao Yuan, Huiming Wang, Gao Cong, and Lidong Bing. 2024. LLM-R2: A large language model enhanced rule-based rewrite system for boosting query efficiency. *arXiv preprint arXiv:2404.12872* (2024).

[38] Li Lin, Hongqiao Chen, Qinglin Zhu, Liehang Chen, Linlong Tang, and Rongxin Wu. 2025. DLBench: A Comprehensive Benchmark for SQL Translation with Large Language Models. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 854–866.

[39] Greg Linden. 2006. Marissa Mayer at Web 2.0. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html. Accessed: 2025-10-25.

[40] Qingkai Min, Yuefeng Shi, and Yue Zhang. 2019. A pilot study for Chinese SQL semantic parsing. *arXiv preprint arXiv:1909.13293* (2019).

[41] Anh Tuan Nguyen, Mai Hoang Dao, and Dat Quoc Nguyen. 2020. A pilot study of text-to-SQL semantic parsing for Vietnamese. *arXiv preprint arXiv:2010.01891* (2020).

[42] Parth Parikh, Oishik Chatterjee, Muskan Jain, Aman Harsh, Gaurav Shahani, Rathin Biswas, and Kavi Arya. 2022. Auto-Query-A simple natural language to SQL query generator for an e-learning platform. In *2022 IEEE Global Engineering Education Conference (EDUCON)*. IEEE, 936–940.

[43] Neil Patel. 2018. How Loading Time Affects Your Bottom Line. https://neilpatel.com/blog/speed-is-a-killer/. Accessed: 2025-10-25.

[44] Yun Peng, Jun Wan, Yichen Li, and Xiaoxue Ren. 2025. Coffe: A code efficiency benchmark for code generation. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 242–265.

[45] PingCAP. 2024. go-randgen: Grammar-based Random SQL Generator. https://github.com/pingcap/go-randgen. Accessed: YYYY-MM-DD.

[46] Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. 2003. Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th international conference on Intelligent user interfaces*. 149–157.

[47] Mohammadreza Pourreza and Davood Rafiei. 2023. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Advances in Neural Information Processing Systems* 36 (2023), 36339–36348.

[48] Gemini 2.5 Pro. [n. d.]. Gemini API Models Documentation (Gemini 2.5 Pro). Google. https://ai.google.dev/gemini-api/docs/models Accessed: 2026-01-22.

[49] Ruizhong Qiu, Weiliang Will Zeng, James Ezick, Christopher Lott, and Hanghang Tong. 2024. How efficient is llm-generated code? a rigorous & high-standard benchmark. *arXiv preprint arXiv:2406.06647* (2024).

[50] Abdul Quamar, Fatma Özcan, Dorian Miller, Robert J Moore, Rebecca Niehus, and Jeffrey Kreulen. 2020. Conversational BI: An ontology-driven conversation system for business intelligence applications. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3369–3381.

[51] Daking Rai, Bailin Wang, Yilun Zhou, and Ziyu Yao. 2023. Improving generalization in language model-based text-to-SQL semantic parsing: Two simple semantic boundary-based techniques. *arXiv preprint arXiv:2305.17378* (2023).

[52] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, et al. 2023. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950* (2023). https://arxiv.org/abs/2308.12950

[53] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. *arXiv preprint arXiv:2109.05093* (2021).

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

[55] SQLCoder. [n. d.]. SQLCoder-7B-2 Model Card. Hugging Face. https://huggingface.co/defog/sqlcoder-7b-2 Accessed: 2026-01-22.

[56] Ruoxi Sun, Sercan Ö Arik, Alex Muzio, Lesly Miculicich, Satya Gundabathula, Pengcheng Yin, Hanjun Dai, Hootan Nakhost, Rajarishi Sinha, Zifeng Wang, et al. 2023. Sql-palm: Improved large language model adaptation for text-to-sql (extended). *arXiv preprint arXiv:2306.00739* (2023).

[57] Zhaoyan Sun, Xuanhe Zhou, Guoliang Li, Xiang Yu, Jianhua Feng, and Yong Zhang. 2024. R-bot: An llm-based query rewrite system. *arXiv preprint arXiv:2412.01661* (2024).

[58] Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. Chess: Contextual harnessing for efficient sql synthesis. *arXiv preprint arXiv:2405.16755* (2024).

[59] Transaction Processing Performance Council (TPC). 2026. TPC-DS Homepage. https://www.tpc.org/tpcds/ Accessed April 17, 2026.

[60] Transaction Processing Performance Council (TPC). 2026. TPC-H Homepage. https://www.tpc.org/tpch/ Accessed: 2026-01-29.

[61] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, et al. 2023. Mac-sql: A multi-agent collaborative framework for text-to-sql. *arXiv preprint arXiv:2312.11242* (2023).

[62] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2019. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. *arXiv preprint arXiv:1911.04942* (2019).

[63] Lijie Wang, Ao Zhang, Kun Wu, Ke Sun, Zhenghua Li, Hua Wu, Min Zhang, and Haifeng Wang. 2020. DuSQL: A large-scale and pragmatic Chinese text-to-SQL dataset. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 6923–6935.

[64] Weiguo Wang, Sourav S Bhowmick, Hui Li, Shafiq Joty, Siyuan Liu, and Peng Chen. 2021. Towards enhancing database education: Natural language generation meets query execution plans. In *Proceedings of the 2021 International Conference on Management of Data*. 1933–1945.

[65] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. 2022. Wetune: Automatic discovery and verification of query rewrite rules. In *Proceedings of the 2022 International Conference on Management of Data*. 94–107.

[66] Yuanzhen Xie, Xinzhou Jin, Tao Xie, MingXiong Lin, Liang Chen, Chenyun Yu, Lei Cheng, ChengXiang Zhuo, Bo Hu, and Zang Li. 2024. Decomposition for enhancing attention: Improving llm-based text-to-sql through workflow paradigm. *arXiv preprint arXiv:2402.10671* (2024).

[67] Chengran Yang, Bowen Xu, Ferdian Thung, Yucen Shi, Ting Zhang, Zhou Yang, Xin Zhou, Jieke Shi, Junda He, Donggyun Han, et al. 2022. Answer summarization for technical queries: Benchmark and new approach. In *ProcEEDINGS OF The 37th IEEE/ACM international conference on automated software engineering*. 1–13.

[68] 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. 2019. Cosql: A conversational text-to-sql challenge towards cross-domain natural language interfaces to databases. *arXiv preprint arXiv:1909.05378* (2019).

[69] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887* (2018).

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

[71] Chao Zhang, Yuren Mao, Yijiang Fan, Yu Mi, Yunjun Gao, Lu Chen, Dongfang Lou, and Jinshu Lin. 2024. Finsql: Model-agnostic llms-based text-to-sql framework for financial analysis. In *Companion of the 2024 International Conference on Management of Data*. 93–105.

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

[73] Xuanhe Zhou, Guoliang Li, Jianming Wu, Jiesi Liu, Zhaoyan Sun, and Xinning Zhang. 2023. A learned query rewrite system. *Proceedings of the VLDB Endowment* 16, 12 (2023), 4110–4113.