SQLBarber: A System Leveraging Large Language Models to Generate Customized and Realistic SQL Workloads

JIALE LAO, Cornell University, USA
IMMANUEL TRUMMER, Cornell University, USA

Database research and development often require a large number of SQL queries for benchmarking purposes. However, acquiring real-world SQL queries is challenging due to privacy concerns, and existing generation methods offer limited options for customization and for satisfying realistic constraints. To address this issue, we present SQLBarber, a system based on Large Language Models (LLMs) to generate customized and realistic SQL workloads. SQLBarber (1) eliminates the need for users to manually craft SQL templates in advance, while providing the flexibility to accept natural language specifications to constrain SQL templates, (2) scales efficiently to generate large volumes of queries matching any user-defined cost distribution (e.g., cardinality and execution plan cost), and (3) uses execution statistics from production environments to extract SQL template specifications and query cost distributions that reflect real-world query characteristics. SQLBarber introduces (1) a declarative interface for users to effortlessly generate customized SQL templates, (2) an LLM-powered pipeline augmented with a self-correction module that profiles, refines, and prunes SQL templates based on query costs, and (3) a Bayesian Optimizer to efficiently explore predicate values and identify a set of queries that satisfy the target cost distribution. We construct and open-source ten benchmarks of varying difficulty levels and target query cost distributions based on real-world statistics from Snowflake and Amazon Redshift. Extensive experiments on these benchmarks show that SQLBarber is the only system that can generate customized SQL templates. It reduces query generation time by one to two orders of magnitude and significantly improves alignment with the target cost distribution, compared with existing methods.

CCS Concepts: • **Information systems → Structured Query Language**; **Database performance evaluation**.

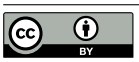
Additional Key Words and Phrases: SQL Generation, Large Language Models, Testing, Benchmarking

ACM Reference Format:
Jiale Lao and Immanuel Trummer. 2026. SQLBarber: A System Leveraging Large Language Models to Generate Customized and Realistic SQL Workloads. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 85 (February 2026), 27 pages. https://doi.org/10.1145/3786699

1 Introduction

Leading companies such as Amazon and Snowflake have recently released workload statistics to facilitate the construction of realistic benchmarks [34, 37]. However, due to privacy constraints, the actual queries and underlying databases are unavailable; only high-level workload characteristics are shared. These typically include: (1) properties of SQL templates (e.g., the number of accessed tables and the distributions of query operators such as joins and aggregations), and (2) cost distributions of instantiated queries (e.g., the distributions of cardinalities, execution times, and CPU usages). This raises an important research direction: transforming high-level workload information into realistic and executable benchmarks.

Authors' Contact Information: Jiale Lao, Cornell University, Ithaca, New York, USA, jiale@cs.cornell.edu; Immanuel Trummer, Cornell University, Ithaca, New York, USA, itrummer@cornell.edu.



ACM 2836-6573/2026/2-ART85
https://doi.org/10.1145/3786699

Recent methods have been proposed for generating SQL queries, but they struggle to produce customized queries that satisfy realistic constraints. SQLSmith [1], SQLancer [2], and SQLStorm [28] focus on generating diverse queries for testing and debugging. However, these methods do not allow users to control SQL templates or query costs, since their generation processes rely on randomization to ensure diversity. HillClimbing [4, 24] and LearnedSQLGen [43] support constraints on query costs but not on SQL templates, and their cost constraints are not derived from real-world statistics. CAB [35] and RedBench [9] synthesize workloads based on real-world statistics [34, 37]. However, they do not generate new queries; instead, they reuse queries from existing benchmarks, which have been shown to differ from real-world workloads [34, 36].

Thus we present SQLBarber, a system based on Large Language Models (LLMs) to generate customized and realistic SQL workloads. SQLBarber (1) removes the need for users to manually create SQL templates in advance, while allowing them to specify high-level natural language constraints for customized SQL template generation (e.g., number of tables, joins, aggregations, or presence of certain operators and complex scalar expressions), (2) scales efficiently to produce a large number of queries that match any user-defined cost distribution (e.g., cardinality and execution plan cost), and (3) uses execution statistics collected from Amazon Redshift [34] and Snowflake [37] to derive both the specifications for SQL templates and the cost distributions of SQL queries, ensuring that the generated SQL workloads reflect production environments.

Example 1.1. A database researcher is developing a novel caching method and wants to benchmark it on realistic workloads from industry. Having high-level information about workload properties, including information on query template structure as well as query cost distribution, as released by companies such as Snowflake, the researcher can generate one or multiple sets of templates with associated queries using SQLBarber. SQLBarber ensures that the generated workloads are close to the target workloads, meaning that benchmark results obtained for the novel caching method and baselines are meaningful.

SQLBarber employs an LLM-powered SQL Template Generator to create customized SQL templates for a target database based on user-defined natural language specifications. It connects to the target database to generate a textual summary of the database schema and identify possible join paths. The generator then constructs a prompt to combine the above information and invoke LLMs to generate SQL templates. Due to the hallucination problem of LLMs (i.e., LLMs can generate plausible but factually incorrect or nonsensical information [6, 40]), the generated SQL templates could have syntax errors and do not satisfy user specifications. Therefore, we propose using the reasoning and self-correction ability of LLMs to make this generation process reliable. Specifically, the generator uses LLMs to judge whether an SQL template satisfies the constraints and provide the reasoning process. If the template fails, the generator invokes LLMs to iteratively rewrite this template based on the reasoning results and error messages from the DBMS. Finally, the generator ensures that the generated SQL templates are executable after instantiation and compliant with user specifications.

SQLBarber relies on a Cost-Aware Query Generator to efficiently produce a large number of queries constrained to a target cost distribution. It takes as input the generated SQL templates and any user-specified cost distribution (e.g., cardinality, execution plan cost, or execution time). Firstly, it employs a profiling stage to evaluate the capability of each SQL template to produce queries of different costs. This is achieved by instantiating the previously generated SQL templates with different predicate values and evaluating these queries on the DBMS. Secondly, based on the profiling statistics, it refines these seed SQL templates to create new templates that can fill the cost ranges that are not covered by existing templates, and prunes templates that cannot contribute to

the target cost distribution. Finally, it utilizes Bayesian Optimization (BO) [20] to efficiently explore the predicate value space and identify a set of SQL queries that satisfy the target cost distribution.

SQLBarber leverages real-world execution statistics to ensure that the generated SQL workloads accurately reflect production environments. To the best of our knowledge, it is the first system to do so. Specifically, SQLBarber analyzes the statistics collected from Amazon Redshift [34] and Snowflake [37] to derive both the specifications for SQL templates and the cost distributions of SQL queries. The specifications define different features for different SQL templates, including the number of tables to access, the number of joins and aggregations, and the presence of certain operators. These specifications are combined with user-provided customized requirements to constrain the SQL template generation process of SQLBarber. Moreover, SQLBarber can generate SQL queries whose cost distribution closely matches the cost distribution observed in real-world workloads. We derive cardinality and execution time distributions from the query logs published by Amazon Redshift and Snowflake, and use these distributions to control the query generation process of SQLBarber. It is important to note that SQLBarber is not restricted to these specific distributions, and can generate queries that follow any user-specified cost distribution.

We extensively evaluate SQLBarber and compare it to state-of-the-art SQL generation methods. We construct and open-source ten realistic benchmarks for SQL workload generation, where each benchmark is characterized by the constraints on SQL templates and the cost distributions of SQL queries. All constraints are extracted from real-world statistics published by Amazon Redshift and Snowflake. We use TPC-H and IMDB as the underlying databases and deploy PostgreSQL as the query execution engine. We classify these benchmarks into different difficulties based on the dataset source, the number of queries to generate, and the number of intervals to split the target cost range. Extensive experiments on these benchmarks show that SQLBarber achieves one to two orders of magnitude improvement in query generation time, and shows significantly better alignment between the target cost distribution and the cost distribution of generated queries, compared to existing methods [4, 43]. Moreover, SQLBarber is the only system that can create customized SQL templates based on natural language instructions by users. We also conduct thorough scalability, ablation, and cost studies to evaluate different aspects of SQLBarber. The code is available at https://github.com/SolidLao/SQLBarber and a demonstration is available at [10].

Our original scientific contributions are the following:

- We present SQLBarber, a system based on LLMs to generate customized and realistic SQL workloads.
- We develop a reliable SQL template generator with a self-correction mechanism to create customized SQL templates based on natural language instructions and DBMS feedback.
- We propose a cost-aware query generator that efficiently produces a large number of queries to fill any cost distribution by profiling, refining, pruning, and instantiating SQL templates under the guidance of Bayesian Optimization.
- We construct and release ten realistic benchmarks for SQL workload generation by analyzing real-world execution statistics collected from Amazon Redshift and Snowflake.
- We evaluate SQLBarber experimentally on the proposed benchmarks and compare against state-of-the-art baselines.

The remainder of this paper is organized as follows. We discuss related work in Section 2. We formally define the SQL workload generation problem in Section 3 and give an overview of SQLBarber in Section 4. Section 5 describes the customized SQL template generator, and Section 6 discusses the cost-aware query generator. We conduct extensive experiments to evaluate the performance, scalability, and robustness of SQLBarber with an ablation study, a cost analysis, and extensive further analysis in Section 7. We conclude in Section 8.

2 Related Work

Diversity-Oriented Query Generation. SQLBarber relates to prior work on generating diverse SQL queries for testing and debugging [1, 2, 28]. SQLSmith [1] recursively and randomly expands grammar rules while ensuring schema and type consistency. SQLancer [2] employs a feedback-guided test generation approach that aims to produce diverse query execution plans, based on the idea that exploring a wide range of plans helps reveal bugs. SQLStorm [28] directly prompts LLMs to generate SQL queries with a high temperature to increase diversity. While the aforementioned methods emphasize query diversity for testing without constraints on SQL templates or query costs, SQLBarber integrates constraints on both templates and costs to turn high-level workload information into realistic, executable benchmarks.

Constraint-Aware Query Generation. SQLBarber is also related to prior work on cost-constrained SQL query generation [4, 24, 43]. Prior work [4, 24] uses the hill climbing algorithm to instantiate given SQL templates by adjusting predicate values so that the resulting queries meet target cost ranges (e.g., cardinalities within $[1000, 2000]$). LearnedSQLGen [43] extends this idea by applying reinforcement learning (RL) to explore both SQL templates and predicate values. However, none of these approaches generate new templates according to user specifications. In contrast, SQLBarber can generate new and customized SQL templates guided by real-world workload statistics [34, 37].

Analysis of Industry Workloads. SQLBarber builds on prior work that analyzes industrial workloads [34, 37, 38]. Redset [34] and Snowset [37] present detailed statistics on production workloads from Amazon Redshift and Snowflake, respectively. These datasets provide statistics for both SQL templates and query costs. Prior studies further show that query repetition is a key characteristic of real-world workloads [34, 38]. For example, Amazon Redshift reports that approximately 85%–90% of executed queries originate from repeated templates in 50% of its clusters [34], while Alibaba's logs show that 4.5 million queries correspond to only 205 templates [38]. Motivated by these findings, SQLBarber adopts templated query generation by first creating new templates from scratch using LLMs and then instantiating queries through Bayesian Optimization (BO). Both steps are guided by real-world statistics [34, 37] to ensure that the generated workloads remain realistic.

Real-World Statistics-Guided Workload Synthesis. SQLBarber relates to prior works that leverage real-world statistics (Redset [34] and Snowset [37]) to construct SQL workloads. CAB [35] uses the original 22 TPC-H templates (with insert and delete streams), deploys multiple databases, and schedules query arrivals based on a real-world Snowflake trace [37] to model multi-tenancy and workload fluctuations in cloud databases. RedBench [9] samples queries from existing benchmarks according to the number of normalized joins while preserving the query repetition patterns observed in Amazon Redshift workloads [34]. However, these approaches do not generate new queries; they reuse or sample existing templates and are thus limited by the diversity and structure of available benchmarks, which differ from real-world workloads [34, 36]. They primarily reproduce execution statistics with limited control over query templates—CAB does not model template structures, and RedBench only considers the number of normalized joins. In contrast, SQLBarber generates new and customizable SQL queries, constraining both template structures and query costs using real-world workload statistics [34, 37], ensuring both realism and flexibility.

LLM-Enhanced DBMS. More broadly, SQLBarber relates to prior works on leveraging LLMs to enhance DBMS, including DBMS tuning [5, 7, 11–13, 17, 32], query rewrite [18, 29, 31], natural language interface [16, 19, 33], and semantic query processing [8, 14, 22].

3 Problem Formulation

Since the actual queries and underlying databases are inaccessible due to privacy concerns, the customized and realistic SQL workload generation problem is to transform high-level workload

information into an executable workload whose SQL template properties and query cost distributions closely match those of the original production workload. Both template generation and query instantiation are guided by production workload statistics. Next, we formally define the problem.

Definition 3.1 (Database). The user must specify a target database $\mathbf{D} = \{R_1, R_2, \ldots, R_n\}$ on which to generate queries. Depending on privacy requirements, this can be a production database with a public schema and masked data, or a database that differs from the production database from which constraints are derived. The SQL generation method should produce SQL templates and instantiate queries whose workload statistics match the target as closely as possible, even when operating on a different database. Following prior work on query or benchmark generation [4, 9, 28, 35, 43], this problem does not constrain the underlying database, and existing methods typically use one from available benchmarks. Although we adopt the same assumption, our experiments in Section 7.5 show that (1) SQLBarber can adjust its generation process to adapt to different database types and sizes, and (2) the selected database affects the diversity and structure of the generated queries. This finding suggests an interesting direction for future work: integrating learned database generation [25, 39, 41] with query generation to enable end-to-end benchmark customization that jointly models both databases and queries.

Definition 3.2 (SQL Template). SQL templates $\mathbf{T} = \{t_1, t_2, \ldots, t_n\}$ are predefined SQL statements in which certain components of each template t_i are placeholders $\mathbf{P_i} = \{p_1, p_2, \ldots, p_m\}$ to be filled with predicate values. SQL templates alone cannot be executed directly by the execution engine.

Example 3.1. Given the schema: `Users(user_id, user_name)` and `Orders(order_id, user_id, order_amount)`, this template retrieves the ids of users who have orders that exceed a certain amount. There is a placeholder p_1 for this amount.

```
SELECT UNIQUE(user_id)
FROM orders
WHERE orders.order_amount > {p1};
```

Definition 3.3 (SQL Query). SQL queries are SQL statements that can be executed directly by the query execution engine. SQL templates **T** can be instantiated into executable SQL queries by replacing the placeholders **P** with predicate values.

Example 3.2. Given the SQL template used in Example 3.1, we instantiate this template by replacing the placeholder for amount with a predicate value of 500.

```
SELECT UNIQUE(user_id)
FROM orders
WHERE orders.order_amount > 500;
```

Definition 3.4 (Specifications for SQL Templates). Specifications $\mathbf{S} = \{s_1, s_2, \ldots, s_n\}$ control the structure and features of SQL templates. Each specification may include numerical constraints, such as limits on the number of tables, joins, or aggregations, as well as structural constraints, such as the presence of nested queries or the use of complex scalar expressions.

Example 3.3. Users can provide a natural language specification, such as: "I want a complex SQL template that accesses 30% of the tables, includes 5 joins, and performs 3 aggregations." Alternatively, for business intelligence use cases, a user might specify: "I want an SQL template with no joins but with complex scalar expressions," which is not supported by any existing benchmark [36].

Definition 3.5 (Customized SQL Template Generation). Given a database **D** and a collection of user specifications **S**, where s_i is a natural language requirement on the SQL template such as "This SQL template should have two joins and access three tables", we want to generate a set of SQL templates **T** such that t_i, after instantiation, is executable on **D** and satisfies s_i. Each SQL template t_i can be instantiated into SQL queries by replacing the placeholders **P** with predicate values.

Example 3.4. Given the schema used in Example 3.1, the user wants an SQL template that includes at least one JOIN and a nested query with an aggregation. This template has two placeholders p_1 and p_2 for two predicate values.

```
SELECT u.user_name, SUM(o.order_amount)
FROM users AS u
JOIN orders AS o ON u.user_id = o.user_id
WHERE u.user_id IN (
    SELECT  user_id
    FROM    orders
    GROUP BY user_id
    HAVING  COUNT(order_id) > {p1}
)
AND  o.order_amount >= {p2};
```

Definition 3.6 (Correctness of SQL Templates.). Correct SQL templates should satisfy two requirements. (1) SQL templates should not have syntax errors, and the instantiated SQL queries should be executable on the target database. (2) SQL templates should satisfy user-defined specifications. Note that since we are targeting generating SQL queries for testing and benchmarking purposes, we do not have requirements on the semantics of these templates.

Definition 3.7 (Cost-Aware Query Generation). Given a database **D**, an SQL template t_i with placeholders $\mathbf{P_i}$, and a target query cost **C**, we try to produce an SQL query by replacing the placeholders $\mathbf{P_i}$ with different predicate values, and identifying appropriate predicate values such that this query can have the target cost **C**. The query cost type could be cardinality, execution plan cost, execution time, or any user-defined one. These cost metrics can be obtained by estimations from the query optimizer or by actual execution.

Example 3.5. Given the database schema used in Example 3.1, the user provides that `Users` has 1000 rows and `Orders` have 500 rows. The user wants one query that returns 200 rows (cardinality=200) by joining the two tables with an appropriate predicate condition. By replacing the placeholder with the predicate value 50, this query successfully returns 200 rows.

```
SELECT u.user_name, o.order_date
FROM users u
JOIN orders o ON u.user_id = o.user_id
WHERE o.order_id > 50;
# Execute: succesfully returns 200 rows
```

Definition 3.8 (Cost Distributions of SQL Queries.). Each SQL query incurs a cost, and users can specify a target cost distribution that the generated queries are expected to follow. The Wasserstein Distance [26] can be used to measure the similarity between the user-specified target distribution and the cost distribution of the generated SQL queries. The system should generate SQL queries with costs that reduce this distance as much as possible within a given time budget.

Definition 3.9 (Customized and Realistic Workload Generation). Given a database **D**, a set of user specifications **S** for SQL templates, the number **N** of queries to generate, and a target cost distribution d^*, we want to generate an SQL workload where the underlying SQL templates satisfy **S** and the instantiated **N** queries match the target cost distribution d^*.

Example 3.6. Figure 1 presents an end-to-end example of the customized and realistic SQL workload generation. On the left side, users provide specifications for SQL templates, including both JSON-formatted numerical constraints and natural language instructions. These specifications guide the generation of SQL templates. Each SQL template contains placeholders for predicate values. By substituting these placeholders with different predicate values, the system generates queries of varying costs. The costs of all generated queries, or a selected subset, are expected to match the user-specified target cost distribution.

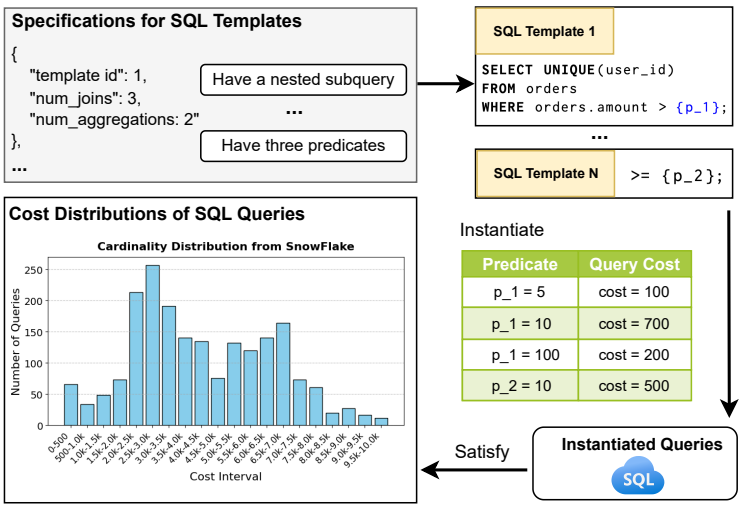


Fig. 1. A Running Example of SQL Workload Generation

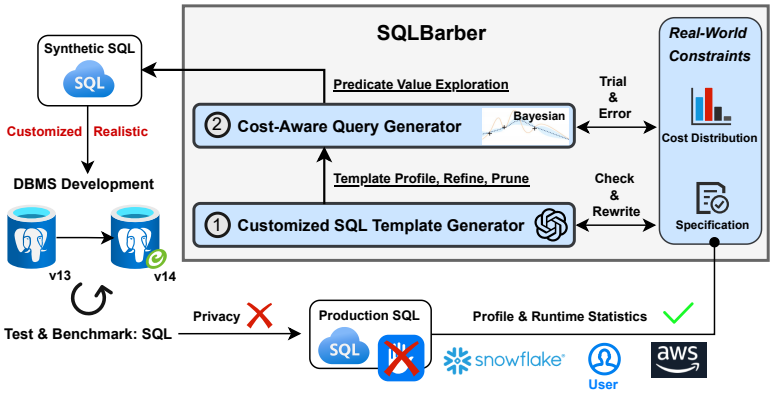


Fig. 2. System Overview of SQLBarber

4 System Overview

Figure 2 shows the two components of SQLBarber: ① a customized SQL template generator and ② a cost-aware query generator.

① The customized SQL template generator uses LLMs to create SQL templates based on user-provided natural language specifications (Definition 3.4). The specifications can define both the structure of the SQL template (e.g., the inclusion of nested subqueries) and its features (e.g., the number of tables accessed, the number of joins and aggregations, or the presence of certain operators such as groupby). First, the Template Generator connects to the target database and produces a text summary of the database schema and join paths. Subsequently, the Template Generator combines the user specification, the database schema, and the join paths to construct a prompt to invoke LLM to generate SQL templates. Due to the hallucination problem in LLMs (i.e., returning a response that is false or entirely made up), it is possible that the generated SQL templates have syntax errors or do not satisfy the user specifications. Since LLMs possess the capability to evaluate and self-correct their outputs, SQLBarber employs LLMs to reason about whether an SQL template satisfies user specifications and to identify the reasons behind it. If a template fails, LLM uses the reasoning results along with the error messages from the DBMS to iteratively rewrite this template, ensuring that it becomes both executable after instantiation and compliant with the user specification.

② The cost-aware query generator produces a large set of SQL queries that match any user-specified cost distribution. It follows a two-phase process that combines profiling with Bayesian Optimization (BO) to instantiate SQL templates using appropriate predicate values. First, the Query Generator profiles each template to estimate its capacity to produce queries across a range of costs. It replaces template placeholders with predicate values uniformly sampled from the target database, runs the resulting queries on the DBMS, and records metrics such as estimated cardinality and cost from the query optimizer. Using these observations, it refines existing SQL templates to create new templates that address uncovered cost ranges and discards templates that do not help cover the desired cost distribution. Third, the Query Generator applies BO to search the predicate value space, balancing the exploitation of predicate values already known to satisfy the cost targets with the exploration of unknown predicate values that may further improve coverage. Finally, the Query Generator outputs the SQL queries whose templates meet the user specifications and whose measured costs conform to the target distribution, making them suitable for realistic testing and benchmarking.

5 Customized SQL Template Generator

Step 1: Database Schema Summary. The Template Generator connects to the target database to gather necessary contexts for template construction and cost-aware query generation. Specifically, it extracts three categories of metadata: table-level, column-level, and constraint-level. For table-level information, we record table names, table sizes, and the number of tuples in a table. Table names tell LLMs which relations a query may access, table sizes allow LLMs to estimate the execution costs of a query (e.g., scanning large tables would take longer than scanning small tables), and the tuple counts provide information on the upper bound of the SQL cardinality. For column-level information, we supply column names, data types, and the number of distinct values, which guide LLMs to select appropriate filter predicates based on selectivity. Finally, constraint-level information includes primary and foreign keys, which indicate valid join pairs, and index metadata, which shows which columns can support index scans or index joins.

Step 2: Join Path Generation. The Template Generator generates all possible join paths based on the constraint-level information collected in Step 1. For each attempt to construct an SQL template, the Template Generator randomly selects a join path that satisfies the user-specified number of joins, and uses this join path to guide the SQL template construction. Currently, SQLBarber selects

join paths only for the main query. Extending this mechanism to handle nested queries would be an interesting direction for future work. Selecting only one join path offers several advantages. First, the random selection makes the template generation diverse enough to cover different join patterns and table combinations across many attempts. Second, this approach is cost-effective due to prompt compression: instead of including metadata for all tables and columns, only those involved in the sampled join path are included, which reduces the number of tokens processed by LLMs. Third, the method improves reliability by mitigating the challenges of long-context processing in LLMs [3], especially when the target database has many tables.

Step 3: Customized Prompt Construction. The Template Generator constructs the prompt for invoking LLMs by combining the database context from Step 1, the sampled join path from Step 2, and the user-provided specifications for SQL templates. These specifications can be expressed in various textual formats, including structured formats such as JSON and unstructured natural language descriptions. In addition to fully customized specifications, SQLBarber also supports using specifications derived from real-world workloads, such as those extracted from Amazon Redshift [34]. Each specification defines key properties of an SQL template, including the number of accessed tables, joins, scans, and aggregations. Taken together, the specifications across all templates describe the high-level characteristics of a real-world workload.

Step 4: SQL Template Generation. The Template Generator uses the prompt constructed in Step 3 to invoke LLMs and generate SQL templates. These templates can be instantiated into executable queries by replacing placeholders with concrete predicate values. It is important to note that this generation process is not cost-aware, and it is unclear whether the resulting queries will align with the target cost distribution. Therefore, the generated SQL templates are treated as seed templates, which will be further refined into template variants that better match the desired cost distribution.

Algorithm 1: Iterative Template Check and Rewrite

Input: SQL template T; User specifications S; Target database $\mathcal{D}$; LLM $\mathcal{M}$; Max iterations k.
Output: Valid SQL template T^* satisfying specifications S.

for $i \in 1..k$ **do**
  // Check specification compliance
  $\langle satisfied, violations \rangle \leftarrow \mathcal{M}.\textsc{ValidateSemantics}(T)$;
  if *not satisfied* **then**
    // Fix semantic violations
    $T^* \leftarrow \mathcal{M}.\textsc{FixSemantics}(T^*, S, violations)$;
  end
  // Check executability
  $\langle executable, errors \rangle \leftarrow \mathcal{D}.\textsc{ValidateSyntax}(T)$;
  if *not executable* **then**
    // Fix execution errors
    $T^* \leftarrow \mathcal{M}.\textsc{FixExecution}(T, errors)$;
  end
  if *satisfied and exetuable* **then**
    return T^*;
  end
end
return T^*;

Step 5: Template Check and Rewrite. The Template Generator iteratively checks and corrects the SQL templates generated in Step 4 to ensure these templates have no syntax errors and fulfill the user-provided specifications, addressing potential hallucination issues associated with the LLM. Algorithm 1 summarizes this process. It accepts an initial SQL template T, user specifications S, the target database $\mathcal{D}$, an LLM $\mathcal{M}$, and a maximum number of iterations k as input. The algorithm iterates until it either generates a valid template or reaches the maximum iteration limit (Line 2). Each iteration consists of two sequential validation phases:

Phase 1: Checking Specification Compliance (Lines 2–5). The algorithm first verifies if the current template T^* meets the provided user specifications by invoking LLM's ValidateSemantics function (Line 2). This function returns a boolean indicator *satisfied* and a list of *violations* indicating specification mismatches. If the template does not satisfy these specifications (Line 3), the LLM's FixSemantics function generates a corrected template based on these violations (Line 4). The iteration then continues with the updated template (Line 5).

Phase 2: Checking Database Executability (Lines 6–9). Next the algorithm validates the syntactic correctness of the template against the target database system by calling the ValidateSyntax function (Line 6). This function returns a boolean *executable* flag and any syntax *errors* encountered. If syntax errors are found (Line 7), the LLM's FixExecution function generates corrections based on the DBMS feedback (Line 8). Finally, we return the fixed SQL templates (Line 11).

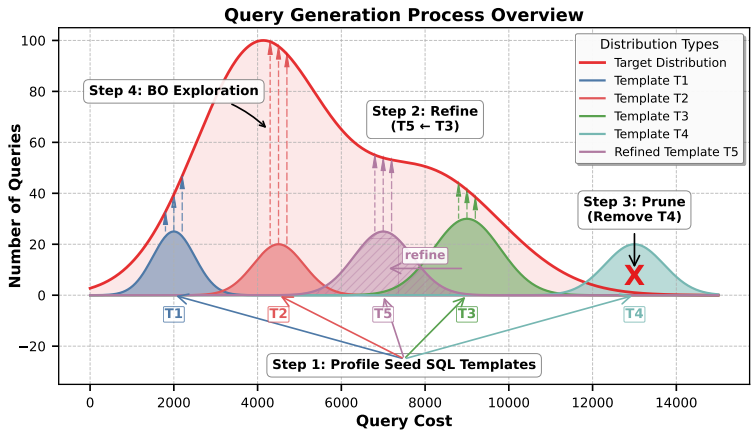


Fig. 3. Cost-Aware Query Generation

6 Cost-Aware Query Generator

Figure 3 illustrates the workflow of our cost-aware query generation process. The query generator takes as input the seed SQL templates produced by the customized SQL template generator (Section 5). First, we profile these seed templates to evaluate their ability to generate queries across different cost levels (Section 6.1). Based on the profiling results, we refine the templates to produce new ones that can better cover the target cost distribution and remove templates that do not contribute effectively (Section 6.2). Finally, we apply Bayesian Optimization (BO) to search for predicate values that yield a sufficient number of queries in each cost range (Section 6.3).

6.1 Template Profiling via Strategic Sampling

SQLBarber profiles all SQL templates generated by the customized SQL template generator to assess their ability to produce queries across different cost levels. This profiling step is important for two

reasons: (1) it identifies whether the existing SQL templates are sufficient to cover the entire cost range; if not, additional templates need to be generated, and (2) it determines which SQL templates are effective at generating queries in specific cost ranges, which in turn guides the optimization of the predicate search process.

SQLBarber profiles SQL templates by instantiating them into executable SQL queries with different predicate values. It then evaluates these queries on the target database to collect performance metrics. Depending on the user-specified optimization target, these metrics may include estimated cardinality and execution cost obtained from the query optimizer using the `EXPLAIN` statement, or actual measurements such as execution time and resource usage (e.g., CPU and memory) collected by executing the queries. In our experiments, we follow prior work [4, 24, 43] to use `EXPLAIN` to get estimated cost and cardinality from the optimizer, and further extend it by using estimated CPU usage as the optimization target.

SQLBarber samples predicate values for each SQL template using a space-filling method, Latin Hypercube Sampling (LHS) [23]. Each SQL template may include multiple placeholders, with each placeholder corresponding to a column in the database schema. Instead of independently sampling values for each column, SQLBarber applies LHS to generate sample values that are more evenly distributed across the joint value space. This helps improve the coverage of the multi-dimensional space and reduces the risk of generating poorly distributed samples, which can occur with independent random sampling along each dimension.

SQLBarber collects profiling results to support future optimizations. For each SQL template, it evaluates a small subset of sampled predicate values on the target database (e.g., 15% of the total number of queries to be generated) to keep the profiling overhead low. As shown in Figure 3 (Step 1), this process produces a cost distribution for each SQL template, which is used to identify the cost ranges that the template can cover and to determine which templates are effective for generating queries within specific cost intervals.

6.2 Adaptive Template Refinement and Pruning

SQLBarber adaptively refines existing SQL templates to generate new ones that cover cost ranges not represented by the original templates, and removes templates that do not contribute to the target distribution, enabling SQLBarber to meet arbitrary distributions.

As shown in Figure 3 (Step 2), given a target query cost range of 0 to 15,000, existing templates (T1 to T4) fail to produce queries with costs between 6,000 and 8,000. SQLBarber refines template T3 to create a new template T5 to generate queries within the 6,000 to 8,000 cost range. SQLBarber removes template T4, as it does not contribute effectively to the target cost distribution (Step 3).

SQLBarber employs a **cost-aware template refinement and pruning algorithm** to adapt initial SQL templates to better match the target cost distribution. First, it identifies low-coverage intervals, which are cost regions with fewer queries than the target requires. Second, it selects candidate templates for refinement based on their estimated utility in generating queries that fall into these low-coverage intervals. Third, it leverages LLMs to refine the most promising templates, producing new templates that are more likely to generate queries within the underrepresented intervals. Finally, it applies profiling to determine whether further refinement is necessary or if the current coverage is sufficient, in which case redundant templates can be pruned.

SQLBarber employs two processing strategies with different costs to process two types of underrepresented intervals: *missing intervals*, which receive little or no coverage, and *difficult intervals*, which fail to reach the target number of queries despite repeated refinements. This design is motivated by the observation that many missing intervals can be addressed efficiently by directly using LLMs to refine existing templates. In contrast, difficult intervals require a more costly strategy that leverages the in-context learning capabilities of LLMs. This involves iterative refinement based

on historical trials and feedback from the DBMS. By applying lightweight processing to missing intervals and reserving expensive refinement for difficult intervals, SQLBarber achieves better cost-effectiveness.

To distinguish underrepresented intervals, formally, let $\mathcal{I} = \{[l_1, u_1), [l_2, u_2), ..., [l_n, u_n)\}$ be the set of cost intervals specified by the user (e.g, cardinality intervals $\{[0, 1000), [1000, 2000)\}$), and let $\mathcal{P} = \{(T_i, C_i)\}_{i=1}^{|\mathcal{T}|}$ denote the profiling results, where T_i is an SQL template from $\mathcal{T}$ and $C_i = \{c_{i1}, c_{i2}, ..., c_{i|C_i|}\}$ is the set of observed execution costs of queries instantiated from T_i. We define the coverage of interval j as the number of queries falling within it:

$$c_j = \sum_{i=1}^{|\mathcal{T}|} |\{c \in C_i : l_j \leq c < u_j\}| \tag{1}$$

To estimate the utility of an SQL template T_i to produce queries in a given cost interval j, we define a *Closeness* metric as follows:

$$s_{ij} = \frac{1}{1 + \bar{d}_{ij}} \cdot v_i \tag{2}$$

where $\bar{d}_{ij}$ is the average distance between the template's historical query costs and interval j, and $v_i = \frac{|\text{unique}(C_i)|}{|C_i|}$, ranging from $\frac{1}{|C_i|}$ to 1, represents the ratio of distinct cost values to total queries, penalizing templates with limited cost diversity. The average distance is calculated as follows:

$$\bar{d}_{ij} = \frac{1}{|C_i|} \sum_{q \in C_i} \text{dist}(\text{cost}(q), [l_j, u_j]) \tag{3}$$

$\text{dist}(c, [l, u])$ returns 0 if $c \in [l, u]$, $(l - c)$ if $c < l$, and $(c - u)$ if $c > u$. This closeness score balances proximity to the target interval with the ability to generate diverse costs.

We use profiling to determine whether a newly refined template should be accepted. A refined template T_{new} with profiling results C_{new} is accepted if it satisfies the following condition:

$$\text{Prune}(T_{new}) = \begin{cases} \text{false} & \text{if } \exists j \in \mathcal{I}_{target} : |\{c \in C_{new} : c \in \mathcal{I}_j\}| > 0 \\ \text{false} & \text{if } D(\mathbf{d}^c + \mathbf{v}_{new}, \mathbf{d}^*) < D(\mathbf{d}^c, \mathbf{d}^*) \\ \text{true} & \text{otherwise} \end{cases} \tag{4}$$

where $\mathcal{I}_{target}$ represents the set of underrepresented intervals, $\mathbf{d}^c$ is the current distribution, $\mathbf{v}_{new}$ is the distribution contribution from C_{new}, and $D(\cdot, \cdot)$ is a distance metric (e.g., Wasserstein Distance) between distributions. Templates that do not generate queries within underrepresented intervals (do not satisfy the first condition in Eq. 4) or fail to improve the overall distribution coverage (do not satisfy the second condition in Eq. 4) are pruned.

The workflow is outlined in Algorithm 2. It takes as input the initial templates $\mathcal{T}$, profiling results $\mathcal{P}$, cost intervals $\mathcal{I}$, target distribution $\mathbf{d}^*$, and an LLM $\mathcal{M}$. The algorithm outputs both the refined templates $\mathcal{T}^*$ and their corresponding profiling results $\mathcal{P}^*$. The algorithm employs a two-phase iterative refinement strategy (Lines 2-11). In each phase, it computes the coverage vector $\mathbf{c}$ (Lines 4-5), where $\mathbf{c}[j]$ counts the number of cost values from all templates that fall within interval $\mathcal{I}_j$. Intervals with low coverage are identified as missing or difficult intervals based on the threshold $\tau \cdot d_j^*$ and based on the current phase (Line 2 and Line 6). The core refinement logic is encapsulated in the RefineForIntervals function (Lines 12-32). For each low-coverage interval j, the function computes closeness scores $\mathbf{s}_j$ using Equation (2) (Line 15) and selects the top-m templates with the highest scores (Line 16). Each selected template T is then refined by the LLM (Line 22), which generates a new template T_{new} targeted at the specific interval $\mathcal{I}_j$.

The two phases differ in their parameters and use of historical context. The first phase uses parameters ($\tau_1 = 0.2, k_1 = 3, m_1 = 3$) without history, performing standard refinement for missing intervals. The second phase employs stricter parameters ($\tau_2 = 0.1, k_2 = 5, m_2 = 5$) with history enabled, targeting persistently difficult intervals. When history is enabled (Lines 19-22), the LLM

Algorithm 2: Cost-Aware Template Refinement & Pruning

Input: Templates $\mathcal{T} = \{T_i\}$; Profiling results $\mathcal{P} = \{(T_i, \mathbf{C}_i)\}$ where $\mathbf{C}_i$ is a cost vector; Cost intervals $\mathcal{I}$; Target distribution $\mathbf{d}^*$; LLM $\mathcal{M}$.
Output: Refined templates $\mathcal{T}^*$; Refined profiling results $\mathcal{P}^*$.

Initialize: $\mathcal{T}^* \leftarrow \mathcal{T}, \mathcal{P}^* \leftarrow \mathcal{P}, \mathcal{H} \leftarrow \{\}$; `// History` $\mathcal{H}$`: interval → list of (template, cost vector)`
for $(\tau, k, m, use\_hist) \in [(\tau_1, k_1, m_1, false), (\tau_2, k_2, m_2, true)]$ **do**
  for $iter = 1$ ***to*** k **do**
    $\mathbf{c} \leftarrow \mathbf{array}(|\mathcal{I}|, 0)$; `// Coverage vector`
    $\mathbf{c}[j] = \sum_{i=1}^{|\mathcal{T}|} |\{c \in C_i : l(\mathcal{I}_j) \le c < u(\mathcal{I}_j)\}|$
    $\mathcal{I}_{low} \leftarrow \{j : \mathbf{c}[j] < \tau \cdot d_j^*\}$; `// Low coverage`
    $\mathcal{N} \leftarrow RefineForIntervals\,(\mathcal{I}_{low}, m, use\_hist)$;
    $\mathcal{T}^* \leftarrow \mathcal{T}^* \cup \mathcal{N}$;
  end
end
return $\langle \mathcal{T}^*, \mathcal{P}^* \rangle$;

Function *RefineForIntervals* $(\mathcal{I}_{target}, m, use\_history)$**:**
  $\mathcal{N} \leftarrow \emptyset$; `// New templates`
  for *each interval* $j \in \mathcal{I}_{target}$ **do**
    $\mathbf{s}_j \leftarrow \text{ComputeCloseness}(\mathcal{P}^*, j)$; `// Eq.(2)`
    $\mathcal{T}_j \leftarrow \text{SelectTopK}(\mathbf{s}_j, m)$;
    for *each template* $T \in \mathcal{T}_j$ **do**
      $history \leftarrow \text{Null}$;
      if $use\_history$ ***and*** $j \in \mathcal{H}$ ***and*** $|\mathcal{H}[j]| > 0$ **then**
        $history \leftarrow \mathcal{H}[j]$;
      end
      $T_{new} \leftarrow \mathcal{M}.\text{RefineTemplate}(T, \mathbf{C}_T, \mathcal{I}_j, history)$;
      $\mathbf{C}_{new} \leftarrow \text{Profile}(T_{new})$; `// Cost vector`
      if *Not Prune*$(\mathbf{C}_{new}, \mathcal{I}_{target}, \mathbf{d}^*)$ `// Eq.(4)`
      then
        $\mathcal{N} \leftarrow \mathcal{N} \cup \{T_{new}\}$;
        $\mathcal{P}^* \leftarrow \mathcal{P}^* \cup \{(T_{new}, \mathbf{C}_{new})\}$;
        $\mathcal{H}[j] \leftarrow \mathcal{H}[j] \cup \{(T_{new}, \mathbf{C}_{new})\}$
      end
    end
  end
  return $\mathcal{N}$;

receives both the current template and previous refinement attempts from $\mathcal{H}[j]$, enabling few-shot learning from past trials. This design is to optimize the trade-off between quality and cost. The parameter values are selected through simple tuning and have shown stable performance across ten benchmarks.

SQLBarber guides LLMs to refine existing SQL templates by leveraging both the historical cost ranges of these templates and the target cost range. It provides hints for three types of refinement operations in the prompt: (1) selecting alternative join paths that access larger or smaller tables to adjust the query cost, (2) modifying the SQL structure by adding or removing predicate conditions or changing filtering columns according to their selectivity, and (3) allowing the LLMs to refine templates freely without explicit constraints. The LLM then decides which operations to apply based on the current SQL template, its cost range, the target cost range, and the available table and column information, and outputs one or more refined SQL templates as candidates.

Each newly generated template undergoes profiling (using the same profiling method in Section 6.1) to obtain its cost vector $\mathbf{C}_{new}$ (Line 23) and is evaluated against the pruning criteria (Line 24). Templates that pass the pruning check—either by filling underrepresented intervals or reducing the overall distribution distance as defined in Equation (4)—are added to the refined template set $\mathcal{N}$ and their profiling results are recorded in $\mathcal{P}^*$ (Lines 26-27). The history $\mathcal{H}$ is also updated to track all refinement attempts (Line 28). This pruning ensures comprehensive generation across the entire cost range while minimizing unnecessary costs.

6.3 Predicate Search via Bayesian Optimization

In Figure 3, after template generation, refinement, and pruning (Steps 1–3), the cost range is fully covered along the horizontal dimension. However, the number of queries per cost interval (the vertical dimension) remains insufficient. SQLBarber uses Bayesian Optimization (BO) to search the predicate values of SQL templates, aiming to produce enough queries within each interval (Step 4).

SQLBarber employs an **Adaptive BO-Based Predicate Search** algorithm to systematically fill gaps between the current and target query cost distributions. As shown in Algorithm 3, it follows a four-step iterative process. First, it identifies the interval j^* with the largest gap between target and current distributions (Line 2). Second, it selects the most promising templates using multiple selection criteria (Line 3-4). Third, for each selected template, it explores the predicate value space using BO (Line 7-8). Finally, it updates the distribution and evaluates progress to adaptively adjust future exploration based on historical performance (Lines 9–14).

The algorithm selects promising SQL templates based on three key criteria (Lines 3-4) and uses closeness-based sampling to balance effectiveness with efficiency (Line 5). It filters for viable templates by checking: (1) sufficient remaining search space to generate enough queries for the gap size, (2) different predicate values can produce queries with diverse costs to avoid redundant explorations, and (3) exclusion of previously ineffective $\langle template, interval \rangle$ pairs (Line 3). Then it performs weighted sampling based on closeness scores (Eq. (2)) to select Top-K templates (e.g., 10) (Line 4), balancing effectiveness with computational efficiency.

The optimization phase leverages BO to efficiently explore predicate values for each selected SQL template. The BO budget is allocated proportionally to the gap size ($\mathbf{d}^*[j^*] - \mathbf{d}[j^*]$, Line 9), ensuring computational resources are allocated based on need. BO iteratively explores the predicate space by balancing exploitation of values meeting cost constraints with exploration of uncertain regions. The objective is to minimize the distance between the actual cost c of an instantiated query and the target cost interval $[c_l, c_r]$:

$$f(c) = \begin{cases} 0, & \text{if } c_l \le c \le c_r \\ 1 - \max\left(\min\left(\frac{c}{c_l}, \frac{c_l}{c}\right), \min\left(\frac{c}{c_r}, \frac{c_r}{c}\right)\right), & \text{otherwise} \end{cases} \tag{5}$$

We use Random Forest as the surrogate model and Expected Improvement as the acquisition function [21] to guide the search in the high-dimensional predicate space, with historical optimization runs reused to warm-start the model when applicable.

Algorithm 3: Adaptive BO-Based Predicate Search

Input: Templates $\mathcal{T}$, Profiling data $\mathcal{P}$, Number of templates to explore in one iteration K, Target distribution $\mathbf{d}^*$, Current distribution $\mathbf{d}$
Output: Updated distribution $\mathbf{d}'$, Generated queries Q

// Main loop: iteratively close distribution gaps
1 **while** $\exists j : \mathbf{d}^*[j] > \mathbf{d}[j]$ **do**
  // Step 1: Identify the most critical gap
2 $j^* \leftarrow \arg\max_j \{\mathbf{d}^*[j] - \mathbf{d}[j]\}$;
  // Step 2: Select most promising templates
3 $\mathcal{T}_{\text{viable}} \leftarrow \{T \in \mathcal{T} :$ HasSufficientSearchSpace$(T) \wedge$ HasDiverseCost$(T) \wedge$ NotPreviouslyIneffective$(T, j)\}$;
4 $\mathcal{T}_{cand} \leftarrow \text{TopK}_{T \in \mathcal{T}}$ (ClosenessScore$(T, j, \mathcal{P})$, K);
5 **if** $\mathcal{T}_{cand} = \emptyset$ **then**
6 Mark j^* as exhausted and continue;
7 **end**
  // Step 3: Explore predicates via BO
8 **for** *each template* $T \in \mathcal{T}_{cand}$ **do**
9 $Q_{new} \leftarrow$ BayesianOptimize$(T, j^*, \mathbf{d}^*[j^*] - \mathbf{d}[j^*])$;
  // Step 4: Update and evaluate progress
10 $\mathbf{d}' \leftarrow$ UpdateDistribution$(\mathbf{d}, Q_{new})$;
11 **if** *insufficient improvement* **then**
12 Mark (j^*, T) as ineffective;
13 **end**
14 $\mathbf{d} \leftarrow \mathbf{d}'$;
15 $Q \leftarrow Q \cup Q_{new}$;
16 **end**
17 **end**
18 **return** $\langle \mathbf{d}, Q \rangle$;

The algorithm employs adaptive learning to improve efficiency. After generating queries, it evaluates whether the produced queries yield sufficient improvement (Line 11). If a template fails to generate queries that improve the target interval, it is marked as ineffective for that interval (Line 12), preventing unnecessary future attempts. When no templates can improve a given interval, the interval is marked as exhausted (Lines 5–7), allowing the algorithm to concentrate on intervals with achievable optimization potential. This process continues until the generated cost distribution closely matches the target or until no further improvable intervals remain. The algorithm's adaptive behavior supports efficient resource usage and handles cases where some cost ranges cannot be improved using the current set of templates.

Example 6.1. Figure 4 presents a running example of Algorithm 3. First, the cost range $[3k–4k]$ lacks 400 queries, which we identify as the most critical gap. Second, the algorithm selects the most promising templates for generating queries in this interval. It applies three selection criteria to filter viable templates (T4 is excluded) and samples from the remaining ones based on their closeness to the gap (T2 and T5 are selected). Third, BO is used to explore the predicate value space of T2 and T5. It leverages observed queries, the surrogate model's predictions, and uncertainty estimates to

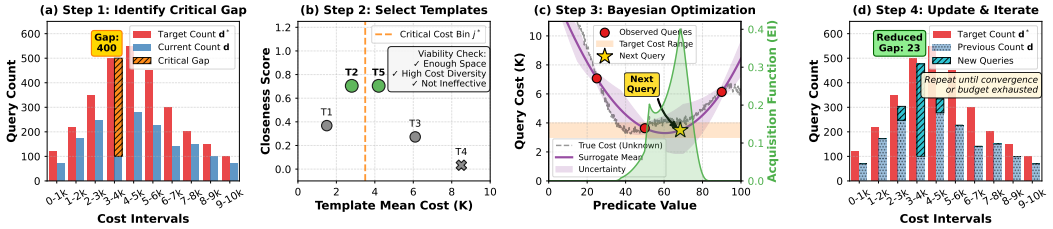


Fig. 4. A Running Example of the Adaptive BO-Based Predicate Search Algorithm

choose the next predicate value by maximizing the Expected Improvement acquisition function. Finally, after exploring and evaluating predicate values, the newly generated queries fill the gaps, reducing the largest gap from 400 to 23 queries. This loop continues until convergence is reached or the computational budget is exhausted.

7 Experimental Evaluation

7.1 Experimental Setup

Datasets. Following prior works [42, 43], we use two datasets as the underlying databases. (1) TPC-H [30] is a widely used benchmark to model a business analytics scenario with eight tables. We set the scale factor to 10. (2) IMDB [15] is a real-world dataset of 21 tables that contains information related to films and television programs.

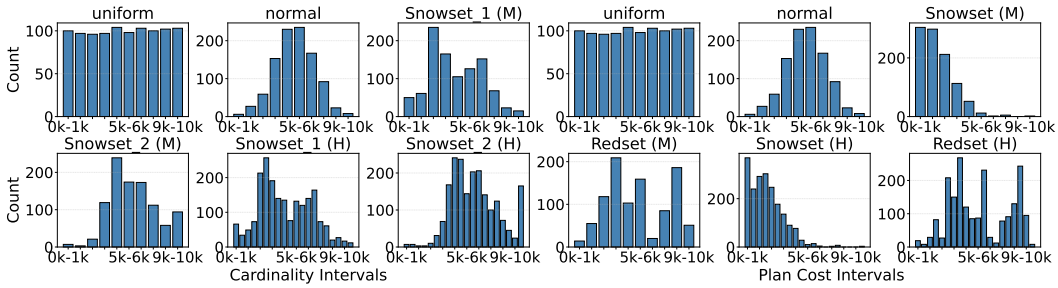


Fig. 5. Target Query Distributions (Left: Cardinality, Right: Execution Plan Cost)

Benchmarks. As shown in Figure 5, we create and open-source ten benchmarks for SQL workload generation. The uniform distribution measures the average ability of a method to generate queries in any given cost range, and the normal distribution simulates the distributions of existing benchmarks such as TPC-H and TPC-DS [34]. The eight real-world distributions are derived from the execution statistics provided by Snowflake [37] and Redshift [34]. For the uniform, normal, and real-world distributions with medium difficulty (denoted as "M"), we generate 1,000 queries across 10 cost intervals. For the more difficult distributions (denoted as "H"), we generate 2,000 queries across 20 cost intervals. Similarly to prior work [43], we set the target cost range to $[0, 10k]$ and actual values are linearly mapped to this range. For SQL template specifications, we use a randomly selected workload from Amazon Redshift [34], which contains 28 tables and 24 SQL templates. Each template is annotated with the attributes *num_tables_accessed*, *num_joins*, and *num_aggregations*. We also construct three natural language instructions to control (1) the presence of a nested subquery, (2) the number of predicate values, and (3) the use of the GROUP BY operator. Each template is randomly assigned at least one of these instructions.

Target Cost Metrics. Following prior studies [4, 24, 43], we focus on two types of SQL costs estimated by the query optimizer: (1) `Cost`, which represents the optimizer's estimated expense of executing a query, and (2) `Cardinality`, which is the number of rows returned by the query. SQLBarber also supports other cost functions beyond these two metrics, such as CPU usage, memory usage, and runtime. We demonstrate this capability by designing a CPU-based cost function that targets a real-world CPU usage distribution from Snowflake [37] in Section 7.5.

Evaluation Metrics. We use three metrics to evaluate SQL generation methods. (1) End-to-End Query Generation Time: This metric measures the total time required to generate a specified number of SQL queries (e.g., $1k$) that satisfy user-defined constraints. These constraints include the cost type (e.g., cardinality or execution cost), the target cost distribution (synthetic or real-world), the specified cost range (e.g., $[0, 10k]$), and the number of cost intervals (10 or 20 intervals). (2) Wasserstein Distance (Earth Mover's Distance [27]): This metric quantifies the difference between the target distribution provided by the user and the distribution of the generated queries. A smaller value indicates a closer match to the target, making it a useful measure for evaluating the quality of the generated queries. (3) Template Generation Accuracy: This metric measures the percentage of generated SQL templates that are executable and align with the user-specified template constraints. We report this metric only for SQLBarber in the ablation study, as it is the only method that supports template generation from natural language.

Baselines. We compare SQLBarber with other state-of-the-art SQL generation methods. (1) LearnedSQLGen [43] is a method that uses Reinforcement Learning (RL) to explore different SQL templates and predicate values to produce queries satisfying given cost constraints. (2) HillClimbing [4] is an approach that takes SQL templates as inputs, and uses heuristics to greedily tweak the predicate values in the given SQL templates to generate satisfied queries. We prepare about 16000 SQL templates as inputs by randomly adding or removing predicates in the SQL templates provided by the benchmarks, the same approach used in [43]. Since these two baseline methods can generate queries for only one cost range per iteration, and the order of generating queries for different ranges may affect performance, we apply two heuristics to each baseline: **Order**, which generates queries from the lowest to the highest cost ranges; and **Priority**, which generates queries for the cost range with the largest number of missing queries in each iteration. The number of optimization iterations is equal to the number of cost intervals in the benchmark. We set a time budget of one hour for each optimization iteration. (3) SQLStorm [28] directly prompts LLMs to generate diverse SQL queries by setting the model temperature to 1. We use the official implementation and follow the original experimental setup, employing `gpt-4o-mini` to generate 5,000 SQL queries for each of seven prompts through the OpenAI batch processing service. After each batch, we estimate query costs using the same procedure as other baselines and compute the Wasserstein Distance. Generation stops once the distance reaches 0 or all seven prompts have been used. For a fair comparison, we omit the steps for cross-DBMS template compatibility to reduce runtime and cost, as this work focuses on PostgreSQL.

Implementation. All methods use PostgreSQL v14.9. We implement SQLBarber in Python, and use OpenAI completion API to invoke the "o3-mini" model for the generation of SQL templates. We implement BO with the SMAC3 library [21].

Hardwares. All experiments are conducted on a Ubuntu server with two Intel Xeon Gold 5218 CPUs (2.3GHz with 32 physical cores), 384 GB of RAM, and two GeForce RTX 2080Ti GPUs. Only LearnedSQLGen [43] uses GPUs to train the neural networks used in RL. This is the same GPU type as reported by the authors.

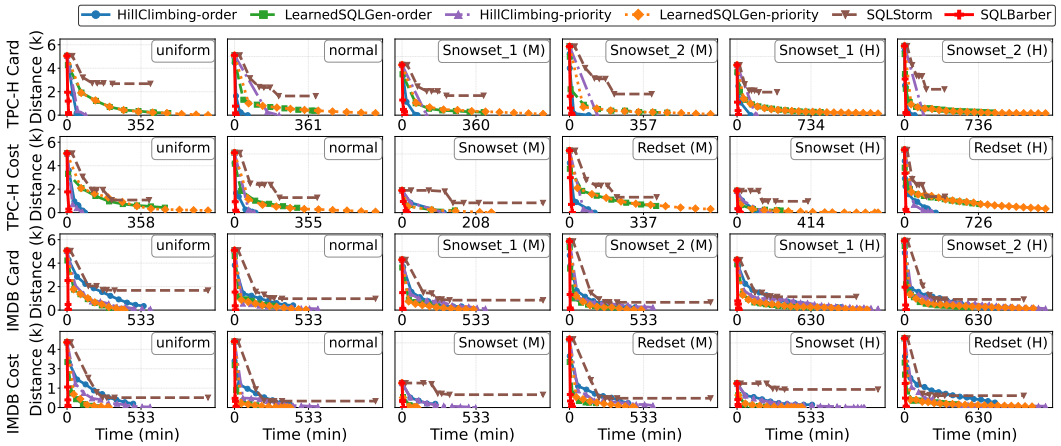


Fig. 6. Distance Between the Target Distribution and the Generated Distribution Over Time

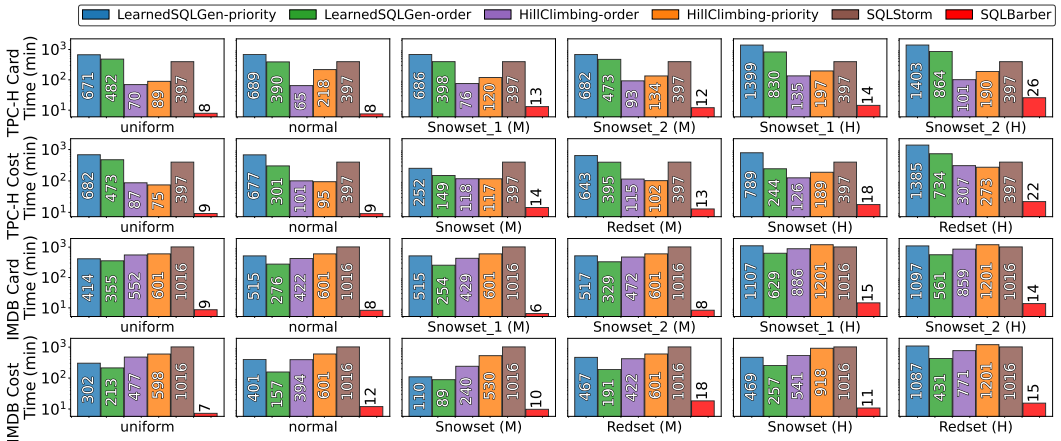


Fig. 7. SQL Workload Generation Time

7.2 Performance Study

Figure 6 and Figure 7 compares the effectiveness and efficiency of different SQL generation methods across all benchmarks, focusing on query cardinality (left) and execution plan cost (right). Each figure consists of three sub-figures arranged in rows for each benchmark: one distribution plot showing the target cost distribution, and two performance plots for the TPC-H and IMDB databases. Each performance plot includes (left) the Wasserstein distance between the target cost distribution and the cost distribution of the generated queries over time, and (right) the end-to-end query generation time.

SQLBarber significantly outperforms baseline methods in both the quality of the generated queries and the generation time. As shown in Figure 6 and Figure 7, SQLBarber consistently reduces the distance to zero across ten benchmarks, two underlying databases, and two optimization targets, with an average generation time of about 12 minutes. In contrast, baseline methods typically fail to reduce the distance to zero, often resulting in distances between 500 and 2000, even after hundreds or thousands of minutes. This gap is more pronounced on harder benchmarks. For instance, on the

"Snowset_Card_2_Hard" benchmark using IMDB as the underlying database, baseline methods require between 561 and 1201 minutes to generate queries with a similar cost distribution. In comparison, SQLBarber generates queries with an exact cost distribution match (i.e., distance of zero) in only 14 minutes. In summary, SQLBarber achieves exact cost distribution matches with one to two orders of magnitude less generation time, while baseline methods fail to reach a zero distance despite extensive computation.

SQLBarber achieves substantial performance gains by adapting to different cost distributions through the use of refined SQL templates and an efficient predicate value search algorithm. In contrast, LearnedSQLGen [43] requires a large number of samples for RL to capture the relationship among query cost, SQL templates, and predicate values, while HillClimbing [4] is limited by the quality of the input SQL templates. An interesting observation is that the baseline methods exhibit different trade-offs between generation time and query quality, depending on the heuristic used. The "order" heuristic results in shorter generation time but larger distances. This difference arises because the "priority" strategy selects cost intervals with the most missing queries, allowing important yet under-covered intervals to be revisited. Revisiting these intervals improves performance, as additional generation attempts are allocated to intervals with many missing queries, rather than discarding them after a single attempt as in the "order" heuristic.

Finally, we compare SQLStorm, the only non–cost-aware baseline, with other methods. SQLStorm takes 394 and 1016 minutes to generate 35,000 queries for the TPC-H and IMDB databases, respectively. These times are shared across all distributions because queries are generated once, and each batch generation for one prompt is treated as a step in the distance-over-time plot (up to seven steps, corresponding to seven prompts). The long generation time results from its batch processing, which exchanges runtime for lower query costs (a 50% discount from OpenAI). SQLStorm cannot reduce the distance to zero for any distribution, with final distances between 1000 and 3000. Failing to reduce the distance to zero is expected since SQLStorm aims for query diversity rather than cost alignment. Note that, as shown in Figure 10 (Section 7.5), SQLStorm indeed produces the most diverse queries among all compared methods and existing benchmarks (e.g., covering the largest query length range). These results demonstrate that diversity-oriented query generation and realistic, constraint-aware query generation address distinct yet complementary problem settings, both of which are essential for DBMS development.

7.3 Scalability Study

Figure 8 compares the scalability of different methods. A green check mark indicates that the method successfully reduces the distance to zero, whereas a red check mark indicates the failure.

Scaling with the number of queries. We evaluate how different methods scale as the number of queries to generate increases. We use "Redset_Cost_Hard" distribution with IMDB as the underlying database, fix the number of intervals to 10, and vary the number of queries from 50 to 500 and 5000. As shown in the first row of Figure 8, SQLBarber demonstrates significantly better scalability. Specifically, SQLBarber consistently reduces the distance to zero within tens of minutes, regardless of the number of queries. In contrast, baseline methods succeed only when the number of queries is small (e.g., 50), but fail to reduce the distance to zero when the number increases to 500 or 5000, even after hundreds of minutes. For baseline methods, query quality degrades as the number of queries increases, as indicated by larger final distances.

Scaling with the number of intervals. We evaluate how different methods scale as the number of intervals increases. We fix the number of queries to 1000, and vary the number of intervals from 5 to 10, 15, 20, and 25. As shown in the second row of Figure 8, SQLBarber again demonstrates significantly better scalability. In contrast, baseline methods are only effective when the number of intervals is as small as 5, and fail to converge as the number of intervals increases, even after

hundreds of minutes. We observe that for baseline methods, the distance does not always increase with more intervals. Since the total number of queries is fixed, adding more intervals reduces queries per interval, leading to smaller differences between the desired and actual counts, and thus lower distances.

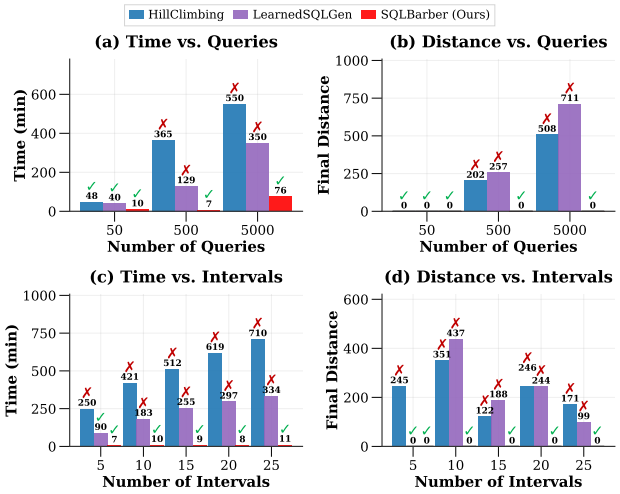


Fig. 8. Scalability Study on IMDB (Execution Plan Cost)

7.4 Ablation Study

Effect of Customized SQL Template Generator. SQLBarber verifies query syntax using DBMS error messages and ensures consistency between specifications and templates through LLM-based self-correction (Section 5, Step 5). This iterative rewrite process is automatically conducted, and to evaluate this process, we manually inspected whether the templates generated in different attempts satisfy their specifications. This manual inspection is only for evaluation purposes to produce Figure 9(a), which shows the cumulative number of templates that are correct with respect to the specification (blue) or syntax (orange) after each rewrite attempt. Initially, only 2 templates meet the specification and 8 are syntactically correct. By the fourth attempt, all 24 templates are both correct and error-free. This demonstrates the effectiveness of our Algorithm 1 in guiding the LLM to self-correct through iterative hybrid feedback.

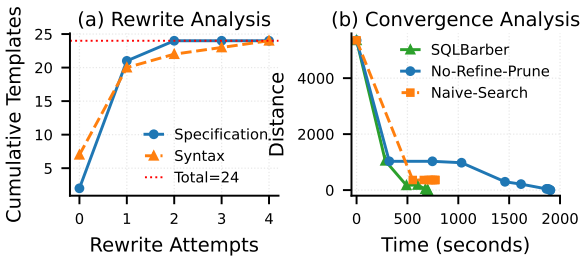


Fig. 9. Ablation Study of SQLBarber (IMDB, Redset_Cost)

Effect of Cost-Aware Query Generator. Figure 9 (b) shows how the Wasserstein distance changes over time when using different versions of SQLBarber. The variant "No-Refine-Prune" disables Algorithm 2, so it does not adaptively refine SQL templates. The variant "Naive-Search" replaces our BO-based predicate search (Algorithm 3) with a random search strategy. "No-Refine-Prune" takes approximately 3× longer to reduce the distance to zero because it cannot adapt SQL templates to complex distributions. "Naive-Search" fails to reduce the distance to zero because it cannot effectively select templates for different cost ranges or search for suitable predicate values. These results highlight the importance of both template refinement and BO-based predicate search.

7.5 Further Analysis

Cost Analysis. We report the number of tokens and the monetary cost incurred by SQLBarber and SQLStorm on the IMDB database in Table 1. On average, SQLBarber incurs costs of 1.36 USD to generate thousands of queries. The cost varies across benchmarks because SQLBarber adaptively generates queries according to the task characteristics. In contrast, SQLStorm generates queries randomly without feedback, so its cost is independent of the benchmark. Although SQLStorm consumes about 80 times more tokens, its monetary cost is only 3.56 times higher because `gpt-4o-mini` is cheaper than `o3-mini` and benefits from a 50% discount for batch processing provided by OpenAI. This difference also leads to a substantial increase in execution time: SQLStorm requires 1016 minutes, whereas SQLBarber takes approximately 18 minutes.

Table 1. Token Usage and Monetary Cost on IMDB

System	Benchmark	Tokens (K)	Cost (USD)
SQLBarber	uniform	416	1.20
SQLBarber	normal	514	1.50
SQLBarber	Snowset_Cost_Medium	387	1.22
SQLBarber	Redset_Cost_Medium	513	1.49
SQLBarber	Snowset_Cost_Hard	453	1.39
SQLBarber	Redset_Cost_Hard	452	1.35
SQLStorm	Benchmark-Independent	34 735	4.84

Comparison of Query Characteristics. Figure 10 compares the characteristics of queries from existing benchmarks and those generated by different systems. The top panel shows the distribution of query lengths, while the bottom panel presents the counts of structural metrics. In the bottom panel, colored bars represent mean values, black error bars indicate the minimum–maximum range, and blue bars denote the standard deviation. SQLBarber is not restricted to the shown distribution, as it can generate queries that satisfy user-specified characteristics. This capability is not available in other methods, whose generation processes are random and uncontrolled. Queries generated by all systems differ notably from those in existing benchmarks, which is desirable because such queries expose different challenges. HillClimbing and LearnedSQLGen produce the least diverse queries, while LLM-based approaches (SQLStorm and SQLBarber) achieve higher diversity, as indicated by the query length distributions and structural metric statistics. SQLBarber generates slightly less diverse queries than SQLStorm, as expected, because its generation process is guided by real-world distributions, whereas SQLStorm generates queries randomly.

Effect of Schema Complexity. As shown in Table 2, we synthesize IMDB-derived databases of different schema complexity. This is achieved by progressively expanding a core schema and controlling attribute retention to vary structural complexity, with detailed procedures provided in

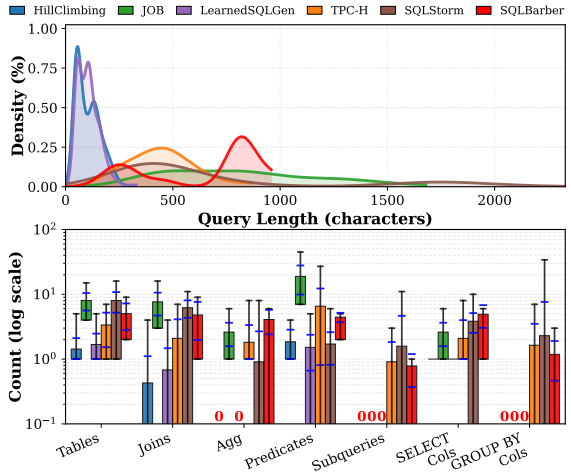


Fig. 10. Query Characteristics Comparison

Table 2. Schema Complexity of IMDB-Derived Databases

Databases	#Tables	#Columns	Avg Cols/Table	Join Range
IMDB-Core	9	27	3.0	1–7
IMDB-Extended	16	53	3.3	1–12
IMDB-Complete	21	81	3.9	1–16

the appendix due to space limitations. Figure 11 illustrates the effect of database schema complexity on SQLBarber when targeting three different cost distributions. SQLBarber consistently reduces the distance to zero through its adaptive, feedback-driven query generation. Although execution times vary across schemas, there is no clear correlation between schema complexity and runtime.

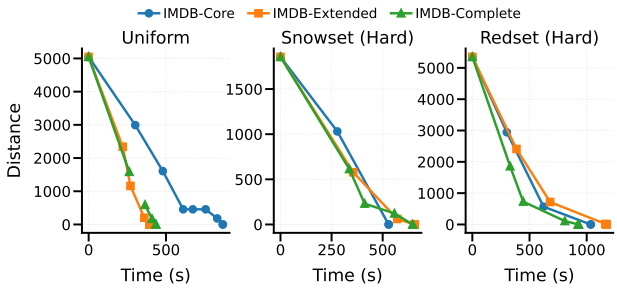


Fig. 11. Effect of Database Schema Complexity

More Target Cost Metrics. SQLBarber can be extended to optimize for additional cost metrics. As an example, we consider a CPU usage–based cost function. The total CPU cost is defined as the sum of CPU costs over all nodes, and we reuse the node-specific CPU cost formulas from PostgreSQL (`costsize.c`). More details about this function are provided in the Appendix. As shown in Figure 12, we use SQLBarber to generate queries on the IMDB and TPC-H databases based on a real-world

CPU usage distribution from Snowset [37], and compare the results against other cost metrics. SQLBarber reduces the distance to 0 within 30 minutes at a low monetary cost ($1.2 on IMDB and $1.7 on TPC-H), achieving comparable performance across different metrics. These results show that SQLBarber can effectively adapt to various cost functions through adaptive template generation and query instantiation.

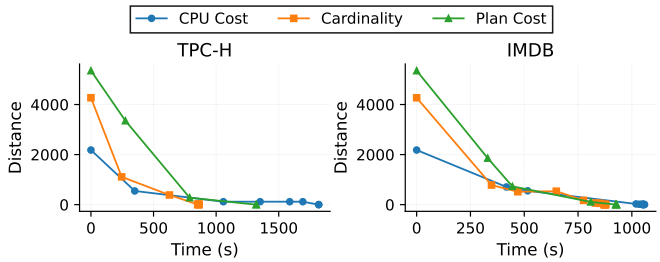


Fig. 12. SQLBarber Performance on Different Cost Metrics

Relation Between Template Specifications and Cost Ranges. Figure 13 shows a heatmap illustrating the cost ranges covered by each SQL template. We present 18 templates generated by SQLBarber from 18 template specifications in Redset [34]. Some templates span all cost ranges (14 and 16), while others cover only narrow ranges (template 12 covers 5k–6k). Certain intervals are covered by many templates (0–1k by 16 templates), whereas others are covered by only a few (9–10k by templates 14 and 16), making those templates crucial. This figure shows that (1) SQLBarber produces diverse templates that cover different cost ranges with varying frequencies, and (2) a fixed template set cannot cover all ranges, motivating SQLBarber's adaptive template refinement.

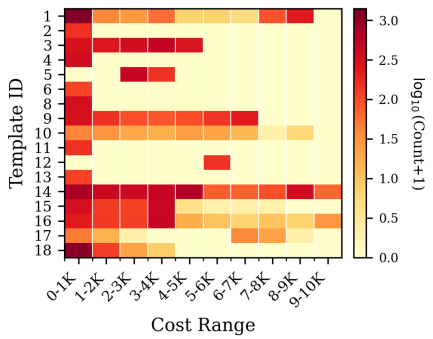


Fig. 13. Relation Between Templates and Cost Ranges

Effect of Database Types and Sizes. We experimented on two databases, IMDB and TPC-H, to show that SQLBarber can adapt to different database types. For database sizes, as shown in Figure 14, we vary the TPC-H SF from 1 to 50, including SF = 1, 10, 20, and 50. SQLBarber consistently reduces the distance to zero across all database sizes through its adaptive template generation and query instantiation. The optimization time, however, varies with database size because the database-independent seed templates exhibit different coverage over the target cost distribution. Specifically, after seed template generation and profiling, SQLBarber covers 17 and 14 out of 20 cost ranges for SF = 1 and SF = 10, but only 2 and 11 ranges for SF = 20 and SF = 50. For SF = 50, no queries appear in the small cost ranges (500–5000). This is expected, as generating low-cost

queries becomes more difficult on larger databases. These uneven initial coverages lead to varying optimization difficulty in the refinement phase. Nevertheless, SQLBarber adaptively adjusts its templates based on feedback from the target database, enabling it to gradually fill all cost ranges and ultimately reduce the distance to zero. Moreover, we observe that there are always some queries falling within the lowest cost range (0–500), regardless of the data scale. To understand this, we analyze the cost ranges produced by each SQL template and find that, due to our random table selection strategy for diverse template generation, some templates access constant-size tables in TPC-H (e.g., region and nation). These queries are therefore inherently low-cost, independent of the database size. However, these queries only account for about 40% of all queries in the smallest cost range. The remaining queries in this range either access tables with highly selective filters or combine constant tables with small tables under multiple predicate conditions. This diversity introduced by SQLBarber is beneficial, as it produces a wide variety of queries across different cost ranges rather than generating repetitive queries that only access constant-size tables.

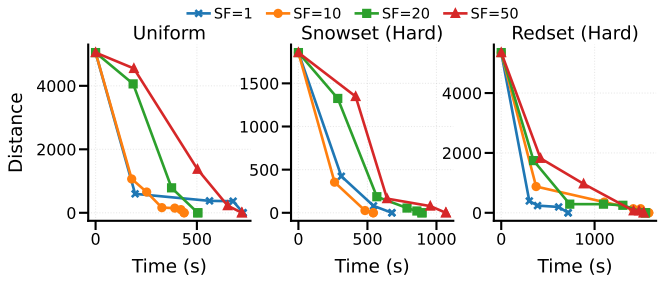


Fig. 14. Effect of Database Sizes (TPC-H, Query Plan Cost)

8 Conclusions and Outlook

We present SQLBarber, a system that uses Large Language Models (LLMs) to generate customized and realistic SQL workloads. Extensive experiments demonstrate its effectiveness, efficiency, and scalability, showing clear improvements over existing methods. We also perform an ablation study to analyze each algorithmic component and a cost study to assess its cost-effectiveness. Finally, we further analyze the characteristics of the generated queries, and evaluate the effects of schema complexity, database types, and database sizes on SQLBarber. We also study the use of CPU usage as a target cost metric and examine the relation between template specifications and cost ranges.

SQLBarber currently targets the OLAP scenario, but it can be extended to support OLTP workloads, including transaction generation and concurrency control. It can also be adapted to support cloud-native features such as multi-tenancy, query repetition, and temporal elasticity. Another interesting extension is enabling SQLBarber to generate semantic queries that extend SQL with semantic operators powered by LLMs or other machine learning models to process multi-modal data. Furthermore, our experiments show that the characteristics of the generated queries and their optimization difficulty are influenced by the types and sizes of the databases. This suggests that SQLBarber could be extended to jointly model and generate both databases and queries. Finally, a promising direction is to explore how to scale the system when using cost metrics that require the actual execution of SQL queries (e.g., execution time).

Acknowledgments

This material is based upon work supported by the National Science Foundation under Award No. 2239326.

References

[1] 2020. A. Seltenreich. Sqlsmith. https://github.com/anse1/sqlsmith

[2] Jinsheng Ba and Manuel Rigger. 2023. Testing database engines via query plan guidance. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2060–2071.

[3] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. 2024. LongBench: A Bilingual, Multitask Benchmark for Long Context Understanding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 3119–3137. doi:10.18653/v1/2024.acl-long.172

[4] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. 2006. Generating queries with cardinality constraints for dbms testing. *IEEE Transactions on Knowledge and Data Engineering* 18, 12 (2006), 1721–1725.

[5] Victor Giannakouris and Immanuel Trummer. 2024. Demonstrating lambda-Tune: Exploiting Large Language Models for Workload-Adaptive Database System Tuning. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD/PODS '24)*. Association for Computing Machinery, New York, NY, USA, 508–511. doi:10.1145/3626246.3654751

[6] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2025. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *ACM Trans. Inf. Syst.* 43, 2, Article 42 (Jan. 2025), 55 pages. doi:10.1145/3703155

[7] Xinmei Huang, Haoyang Li, Jing Zhang, Xinxin Zhao, Zhiming Yao, Yiyan Li, Tieying Zhang, Jianjun Chen, Hong Chen, and Cuiping Li. 2025. E2ETune: End-to-End Knob Tuning via Fine-tuned Generative Language Model. arXiv:2404.11581 [cs.AI] https://arxiv.org/abs/2404.11581

[8] Saehan Jo and Immanuel Trummer. 2024. ThalamusDB: Approximate Query Processing on Multi-Modal Data. *Proc. ACM Manag. Data* 2, 3, Article 186 (May 2024), 26 pages. doi:10.1145/3654989

[9] Skander Krid, Mihail Stoian, and Andreas Kipf. 2025. Redbench: A Benchmark Reflecting Real Workloads. *arXiv preprint arXiv:2506.12488* (2025).

[10] Jiale Lao and Immanuel Trummer. 2025. Demonstrating SQLBarber: Leveraging Large Language Models to Generate Customized and Realistic SQL Workloads. In *Companion of the 2025 International Conference on Management of Data* (Berlin, Germany) *(SIGMOD/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 151–154. doi:10.1145/3722212.3725101

[11] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. 2024. GPTuner: A Manual-Reading Database Tuning System via GPT-Guided Bayesian Optimization. *Proc. VLDB Endow.* 17, 8 (may 2024), 1939–1952. doi:10.14778/3659437.3659449

[12] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. 2025. GPTuner: An LLM-Based Database Tuning System. *SIGMOD Rec.* 54, 1 (April 2025), 101–110. doi:10.1145/3733620.3733641

[13] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Yuanchun Zhou, Mingjie Tang, and Jianguo Wang. 2024. A Demonstration of GPTuner: A GPT-Based Manual-Reading Database Tuning System. In *Companion of the 2024 International Conference on Management of Data* (Santiago, Chile) *(SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 4 pages. doi:10.1145/3626246.3654739

[14] Jiale Lao, Andreas Zimmerer, Olga Ovcharenko, Tianji Cong, Matthew Russo, Gerardo Vitagliano, Michael Cochez, Fatma Özcan, Gautam Gupta, Thibaud Hottelier, H. V. Jagadish, Kris Kissel, Sebastian Schelter, Andreas Kipf, and Immanuel Trummer. 2025. SemBench: A Benchmark for Semantic Query Processing Engines. arXiv:2511.01716 [cs.DB] https://arxiv.org/abs/2511.01716

[15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215. doi:10.14778/2850583.2850594

[16] 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. *Proc. ACM Manag. Data* 2, 3, Article 127 (May 2024), 28 pages. doi:10.1145/3654930

[17] Yiyan Li, Haoyang Li, Zhao Pu, Jing Zhang, Xinyi Zhang, Tao Ji, Luming Sun, Cuiping Li, and Hong Chen. 2024. Is Large Language Model Good at Database Knob Tuning? A Comprehensive Experimental Evaluation. arXiv:2408.02213 [cs.DB] https://arxiv.org/abs/2408.02213

[18] 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. *Proc. VLDB Endow.* 18, 1 (Sept. 2024), 53–65. doi:10.14778/3696435.3696440

[19] Meiyu Lin, Haichuan Zhang, Jiale Lao, Renyuan Li, Yuanchun Zhou, Carl Yang, Yang Cao, and Mingjie Tang. 2025. ToxicSQL: Migrating SQL Injection Threats into Text-to-SQL Models via Backdoor Attack. arXiv:2503.05445 [cs.CR]

https://arxiv.org/abs/2503.05445

[20] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. 2022. SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization. *Journal of Machine Learning Research* 23, 54 (2022), 1–9. http://jmlr.org/papers/v23/21-0888.html

[21] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. 2022. SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization. *Journal of Machine Learning Research* 23, 54 (2022), 1–9. http://jmlr.org/papers/v23/21-0888.html

[22] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, et al. 2025. Palimpzest: Optimizing ai-powered analytics with declarative query processing. In *Proceedings of the Conference on Innovative Database Research (CIDR)*. 2.

[23] Wei-Liem Loh. 1996. On Latin hypercube sampling. *The annals of statistics* 24, 5 (1996), 2058–2080.

[24] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. 2008. Generating targeted queries for database testing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 499–510. doi:10.1145/1376616.1376668

[25] Parimarjan Negi, Laurent Bindschaedler, Mohammad Alizadeh, Tim Kraska, Jyoti Leeka, Anja Gruenheid, and Matteo Interlandi. 2023. Unshackling Database Benchmarking from Synthetic Workloads. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 3659–3662. doi:10.1109/ICDE55515.2023.00292

[26] Victor M Panaretos and Yoav Zemel. 2019. Statistical aspects of Wasserstein distances. *Annual review of statistics and its application* 6, 1 (2019), 405–431.

[27] Victor M Panaretos and Yoav Zemel. 2019. Statistical aspects of Wasserstein distances. *Annual review of statistics and its application* 6, 1 (2019), 405–431.

[28] Tobias Schmidt, Viktor Leis, Peter Boncz, and Thomas Neumann. 2025. SQLStorm: Taking Database Benchmarking into the LLM Era. *Proc. VLDB Endow.* 18, 11 (Sept. 2025), 4144–4157. doi:10.14778/3749646.3749683

[29] Yuyang Song, Hanxu Yan, Jiale Lao, Yibo Wang, Yufei Li, Yuanchun Zhou, Jianguo Wang, and Mingjie Tang. 2025. QUITE: A Query Rewrite System Beyond Rules with LLM Agents. arXiv:2506.07675 [cs.DB] https://arxiv.org/abs/2506.07675

[30] Transaction Processing Performance Council (TPC). 2014. *TPC Benchmark^{TM} H (Decision Support) Standard Specification, Revision 2.17.1.* Technical Report. Transaction Processing Performance Council, San Francisco, CA, USA. https://www.tpc.org/tpch/

[31] Immanuel Trummer. 2022. CodexDB: synthesizing code for query processing from natural language instructions using GPT-3 codex. *Proc. VLDB Endow.* 15, 11 (July 2022), 2921–2928. doi:10.14778/3551793.3551841

[32] Immanuel Trummer. 2022. DB-BERT: A Database Tuning Tool That "Reads the Manual". In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 190–203. doi:10.1145/3514221.3517843

[33] Matthias Urban and Carsten Binnig. 2024. Demonstrating CAESURA: Language Models as Multi-Modal Query Planners. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 472–475. doi:10.1145/3626246.3654732

[34] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is Not Enough: An Analysis of the Amazon Redshift Fleet. *Proc. VLDB Endow.* 17, 11 (aug 2024), 3694–3706. doi:10.14778/3681954.3682031

[35] Alexander van Renen and Viktor Leis. 2023. Cloud Analytics Benchmark. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1413–1425. doi:10.14778/3583140.3583156

[36] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *Proceedings of the Workshop on Testing Database Systems* (Houston, TX, USA) *(DBTest '18)*. Association for Computing Machinery, New York, NY, USA, Article 1, 6 pages. doi:10.1145/3209950.3209952

[37] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 449–462. https://www.usenix.org/conference/nsdi20/presentation/vuppalapati

[38] Jiaqi Wang, Tianyi Li, Anni Wang, Xiaoze Liu, Lu Chen, Jie Chen, Jianye Liu, Junyang Wu, Feifei Li, and Yunjun Gao. 2023. Real-Time Workload Pattern Analysis for Large-Scale Cloud Databases. *Proc. VLDB Endow.* 16, 12 (Aug. 2023), 3689–3701. doi:10.14778/3611540.3611557

[39] Qingshuai Wang, Yuming Li, Rong Zhang, Ke Shu, Zhenjie Zhang, and Aoying Zhou. 2023. A Scalable Query-Aware Enormous Database Generator for Database Evaluation. *IEEE Trans. on Knowl. and Data Eng.* 35, 5 (May 2023), 4395–4410. doi:10.1109/TKDE.2022.3153651

[40] Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. 2025. Hallucination is Inevitable: An Innate Limitation of Large Language Models. arXiv:2401.11817 [cs.CL] https://arxiv.org/abs/2401.11817

[41] Jingyi Yang, Peizhi Wu, Gao Cong, Tieying Zhang, and Xiao He. 2022. SAM: Database Generation from Query Workloads with Supervised Autoregressive Models. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1542–1555. doi:10.1145/3514221.3526168

[42] Geoffrey X. Yu, Ziniu Wu, Ferdi Kossmann, Tianyu Li, Markos Markakis, Amadou Ngom, Samuel Madden, and Tim Kraska. 2024. Blueprinting the Cloud: Unifying and Automatically Optimizing Cloud Data Infrastructures with BRAD. *Proc. VLDB Endow.* 17, 11 (July 2024), 3629–3643. doi:10.14778/3681954.3682026

[43] Lixi Zhang, Chengliang Chai, Xuanhe Zhou, and Guoliang Li. 2022. LearnedSQLGen: Constraint-aware SQL Generation using Reinforcement Learning. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 945–958. doi:10.1145/3514221.3526155

Received July 2025; revised October 2025; accepted November 2025