SQLStorm: Taking Database Benchmarking into the LLM Era

Tobias Schmidt
Technische Universität München
tobias.schmidt@in.tum.de

Viktor Leis
Technische Universität München
leis@in.tum.de

Peter Boncz
CWI
boncz@cwi.nl

Thomas Neumann
Technische Universität München
neumann@in.tum.de

ABSTRACT

In this paper, we introduce a new methodology for constructing database benchmarks using Large Language Models (LLMs), as well as SQLStorm v1.0, a concrete benchmark on a real-world dataset of three sizes (1 GB, 12 GB, 220 GB) consisting of over 18 K queries. This methodology of using AI to generate query workloads breaks new ground, not only in its ability to cheaply ($15) generate huge volumes (22 MB) of realistic queries but especially because it greatly expands the amount of SQL functionality and query constructions that is covered, compared to human-written SQL benchmarks such as TPC-H, TPC-DS, and JOB. The use cases of SQLStorm that we think will advance data systems most are: (i) improving SQL compatibility between systems, (ii) increasing system quality by identifying crashes/errors and fixing those, (iii) improving cardinality estimators and query optimizers, by identifying trends and opportunities (queries where other systems do much better), as well as (iv) overall system performance, both in terms of speed and robustness.

PVLDB Reference Format:
Tobias Schmidt, Viktor Leis, Peter Boncz, and Thomas Neumann. SQLStorm: Taking Database Benchmarking into the LLM Era. PVLDB, 18(11): 4144 - 4157, 2025.
doi:10.14778/3749646.3749683

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/SQL-Storm/SQLStorm.

1 INTRODUCTION

Benchmarks crystallize performance, providing an objective foundation for comparing database systems. Beyond mere measurement tools, they shape the evolutionary trajectory of database technologies by influencing which workloads vendors and researchers prioritize. In the OLAP space, TPC-H and TPC-DS have emerged as dominant benchmarks in industry and academia.

Both benchmarks rely on a limited set of handcrafted queries and synthetic data generators. While this controlled design ensures repeatability and comparability, it also comes with drawbacks. TPC benchmarks, for instance, have been criticized for their limited coverage of key real-world functionalities, such as string processing, and their reliance on unrealistic data distributions that fail to capture the complexities of production workloads [49, 56, 59, 60]. However, given the privacy concerns surrounding real-world query logs and database contents, synthetic OLAP benchmarks have remained the primary – if not the only – viable alternative.

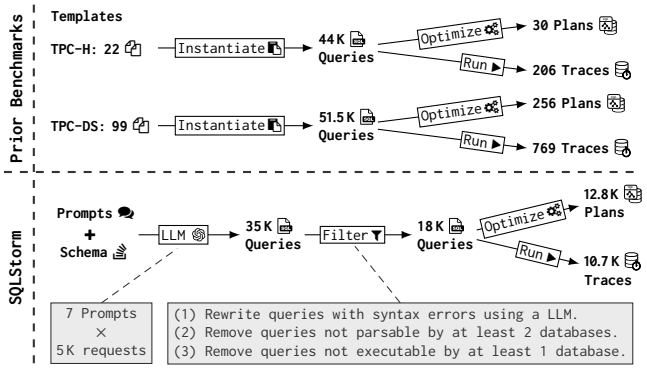

Figure 1: Comparison of query generation pipelines in template-based benchmarks (TPC-H/DS) and our LLM-based SQLStorm methodology. The generated queries are optimized and executed in Umbra and we report the number of distinct query plans and execution traces. The LLM-based approach yields a more diverse query set than templated benchmarks.

Generative AI, powered by Large Language Models (LLMs), presents a novel opportunity to rethink SQL benchmarking. LLMs from major AI vendors have been trained on vast open-source SQL corpora, implicitly capturing real-world SQL usage patterns. With appropriate prompting, these models can generate realistic SQL queries that better reflect practical database workloads. Moreover, the stochastic nature of LLMs enables random sampling from a distribution of real-world SQL queries, creating large, diverse, and representative query sets.

We propose a new benchmarking methodology called SQLStorm that leverages LLMs for SQL generation. By providing a database schema and carefully designed prompts, LLM APIs can generate hundreds of thousands of complex queries that more accurately reflect real-world SQL workloads than traditional handcrafted benchmarks. Figure 1 illustrates SQLStorm's generation pipeline for rewriting and filtering LLM-generated SQL queries. It removes erroneous queries and ensures compatibility with multiple systems by testing them in PostgreSQL, Umbra, and DuckDB. Our new approach facilitates automated large-scale benchmark generation.

Conventional OLAP benchmarks, like TPC-H and TPC-DS, rely on a small set of expert-curated query templates that, while useful

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749683

for comparability, are highly sanitized and often fail to capture the diversity and unpredictability of real-world workloads. Even when instantiated thousands of times, template-based benchmarks yield only a limited variety of query plans, and executions of the same template are similar. In contrast, LLM-generated benchmarks exhibit a much broader range of complexity, sometimes incorporating unexpected constructs observed in open-source SQL corpora, resulting in thousands of distinct query plans and execution traces. Furthermore, the stochastic behavior of LLMs generates a long-tailed distribution of queries, with some containing up to 38 operators – a characteristic common in production databases but rarely seen in traditional benchmarks.

Using our methodology, we also contribute SQLStorm v1.0, a concrete benchmark designed to be both comprehensive and challenging for modern database systems, built on the StackOverflow dataset, available in three sizes (1 GB, 12 GB, and 220 GB). For this first version, we employ OpenAI's GPT-4o-mini model to generate a diverse set of SQL queries tailored to the StackOverflow schema. This constitutes a rich, realistic dataset alongside an extensive query set that spans a wide array of SQL constructs. The sheer volume of queries (over 22 MB) not only ensure broad SQL pattern coverage but also makes it nearly impossible to optimize for individual queries. TPC-H, in particular, allows for per-query chokepoint analysis and hand-optimized implementations, as it contains only 22 query templates [10, 14, 16].

As the name suggests, SQLStorm is designed to stress-test database engines across multiple dimensions, including functionality, error handling, optimization, and execution. The benchmark supports several key use cases:

- **Optimizer Stress Testing:** The extensive query set exposes weaknesses in query optimizers by incorporating a broad mix of simple, complex, and edge-case queries.
- **Cardinality Estimation:** SQLStorm challenges cardinality estimation techniques on real-world data with complex queries and a large set of relational operators.
- **Performance Regression Testing:** SQLStorm's extensive and stable query set enables developers to track performance across versions, ensuring reliability as systems evolve.
- **Scalability Analysis:** With datasets spanning three orders of magnitude in size, the benchmark enables systematic evaluation of database scalability.
- **Comparative Benchmarking:** SQLStorm facilitates head-to-head comparisons of database systems.
- **Query Optimization Mining:** Large relative performance differences between systems often highlight missing query optimization rules in the slower system.
- **Bring your own data:** SQLStorm can be applied to any dataset, allowing users to generate custom benchmarks.

By leveraging LLMs for benchmark generation, SQLStorm represents a paradigm shift in SQL benchmarking – moving beyond static, expert-designed queries toward a scalable, adaptable methodology that better reflects real-world workloads.

The rest of this paper is organized as follows: We begin by reviewing related work in Section 2. Next, Section 3 introduces the SQLStorm methodology and details how LLMs can be used to efficiently generate large-scale benchmarks. We then present and analyze the SQLStorm v1.0 dataset and queries. Following that, Section 4 explores practical use cases for SQLStorm. Finally, we conclude Section 5 and discuss future directions.

2 RELATED WORK

There is a large body of work on human-expert crafted database benchmarks. On the industrial side, the most important have been the efforts of TPC [20]. Good performance on TPC-H/-DS and TPC-C are now primary targets to show that a new analytical respective transactional database system masters the basic techniques required to enter the (mature) database market. While these well-understood [10, 16, 53] benchmarks exercise execution engines reasonably well, they fail to address crucial optimizer problems like join ordering due to their small schemas, limited queries, and uniform data distributions. JCC-H [9] and DSB [15] extend existing benchmarks by incorporating complex correlations and skew to stress systems. From the academic side, JOB [30] and its variants [25, 63] introduced complex join query patterns on the real-life IMDB dataset with skewed and correlated data distributions, helping spur recent efforts to improve query optimization. However, the decision to keep the query sets in these benchmarks small and hence understandable for humans naturally limits the coverage of complex structures.

Previous benchmarks with a large set of realistic queries, most prominently, did so by distilling real workloads – consisting of data schemas and query logs – into synthetic equivalents [13, 26, 41, 57]. These methodologies, while valuable, only work for database teams that already have access to such customer-donated workloads. Also, real-life workloads [56, 60] are typically dominated by many simpler queries. Hence, a sample from those may not adequately cover the space of complex, large, or even outlandish queries that mature database systems are expected to handle robustly.

SQLStorm uses the Stack Overflow datasets – a decision also followed by STATS-CEB [18] and Stack [36], whose query set consists of 100 human-generated queries. These queries are relatively simple and cover only a small fraction of database functionality exercised by SQLStorm. The only existing database benchmark, to our knowledge, that exploits LLMs to generate queries, is the Surprise Benchmark [7] – however, it generates batches of only 5 queries incrementally. A final relevant field is database testing and database fuzzing [44]. We tried using SQLSmith [45], and while it can generate many queries quickly, these lack credible semantics and, consequently, relevant operator and expression patterns.

Large language models have been investigated for a variety of database-related tasks [31, 70]. Given their capabilities as coding assistants, they are naturally suited for text-to-SQL translation [19, 32, 54, 68]. However, researchers are also exploring deeper integration of LLMs into data systems, including applications such as database tuning [12, 27, 33, 55], query optimization [5, 50, 64], query rewriting [34, 35, 48], data cleaning [40, 58], and diagnostic tools [69].

3 SQLSTORM

Existing benchmarks like TPC-H and TPC-DS are widely used to evaluate the performance of database systems. However, these benchmarks have several limitations: (a) the query sets are relatively

small and contain at most a hundred query templates, (b) the queries only use a fraction of the SQL standard and do not include more complex constructs like recursion, arrays, or JSON, and (c) the benchmarks use synthetic data that fail to represent real-world scenarios. Our novel SQLStorm methodology uses a real-world dataset and large-scale query sets to address these limitations.

This section describes how SQLStorm generates database benchmarks using LLMs. The process can be divided into four steps:

(1) Query Generation: We use OpenAI's *GPT-4o-mini* model to generate SQL queries based on user-defined prompts.
(2) Query Rewriting: We rewrite the generated queries to increase compatibility and remove duplicates.
(3) Query Selection: SQLStorm relies on PostgreSQL, Umbra, and DuckDB to identify parsable and executable queries.
(4) Complexity Classification: We classify the generated SQL queries based on their complexity and the features they use.

3.1 The StackOverflow Dataset

The Stack Exchange network is a valuable knowledge base for technical topics with millions of questions, answers, and comments. The StackOverflow website, in particular, is a popular platform for asking and answering programming-related questions [4]. Stack Exchange makes data dumps of their network available under the Creative Commons BY-SA 4.0 license, allowing us to use the data for research purposes. Compared to other real-world datasets, like IMDB [30], the Stack Exchange data dumps contain more than 100 GB of data, and different sizes are available for download. This allows for a scalable and challenging benchmark for modern analytical database systems.

We downloaded three datasets in different sizes from the StackOverflow website [23], the Mathematics Stack Exchange (Math) [22], and the Database Architects Stack Exchange (DBA) [21]. The data dumps contain all the questions, comments, votes, and users up to the 3rd of October 2024. The extracted CSV from the StackOverflow website is the largest, with 240 GB; the other two datasets are smaller and contain only 13.8 GB and 1.26 GB, respectively. Instructions for downloading a newer version of the dataset can be found here [3]; we provide scripts to convert the dumps to CSV files. In addition, the database schema and documentation are also available online at [1]. We make the extracted datasets and the schemas available for download along with the SQLStorm benchmark.

The dataset consists of 13 tables, 5 of which encode enum types and are the same size for all datasets. The remaining 8 relations contain the actual data and scale with the dataset size. Table 1 lists their size and the number of records for SQLStorm-220 (StackOverflow) the largest of the three datasets with 220 GB. The two smaller datasets SQLStorm-1 (DBA) and SQLStorm-12 (Math) consist of 3.1 M rows with 1.2 GB and 39.6 M rows with 12.6 GB, respectively.

The *Users* and *Badges* tables model the users and the badges they have earned. The *Posts* and *Comments* tables contain the website's actual content, while the *PostHistory* table stores the edit history. The *PostLinks*, *Tags*, and *Votes* relations store further metadata for the posts. In addition, the five tables not listed — *PostTypes*, *PostHistoryTypes*, *CloseReasonTypes*, *VoteTypes*, and *LinkTypes* — model enum values. The database schema contains 19 foreign key relationships, including self-references, allowing complex join graphs. Every table has a 32-bit integer as primary key.

Table 1: The number of rows and (uncompressed) binary size of the Stack Overflow dataset. The last part of the table reports how many columns with the data type exists and the fraction of space they occupy on SQLStorm-220. Text columns consume more than 200 GB.

Table	#Rows	Size	Integer	Date	Text
Users	26.1 M	2.2 GB	6 (30.3 %)	2 (17.3 %)	5 (52.3 %)
Badges	52.9 M	1.5 GB	4 (37.1 %)	1 (27.0 %)	1 (36.0 %)
Posts	60.2 M	71.6 GB	11 (3.3 %)	5 (3.1 %)	6 (93.6 %)
Comments	91.0 M	16.5 GB	4 (8.2 %)	1 (4.1 %)	3 (87.7 %)
PostHistory	161.9 M	122.7 GB	4 (1.7 %)	1 (1.0 %)	5 (97.3 %)
PostLinks	6.5 M	0.2 GB	4 (69.2 %)	1 (30.8 %)	0 (0.0 %)
Tags	66.0 K	1.9 MB	6 (61.0 %)	0 (0.0 %)	1 (39.0 %)
Votes	242.4 M	5.9 GB	5 (69.2 %)	1 (30.8 %)	0 (0.0 %)
Total	641.1 M	220.5 GB	44 (5.1 %)	12 (3.1 %)	21 (91.8 %)

We chose the three datasets as their size is in different orders of magnitude. SQLStorm-12 is roughly 10× larger than SQLStorm-1, while SQLStorm-220 is almost 20 times larger than the Math dataset. Note that the datasets do not scale perfectly linearly; for example, the number of users grows only by 5× between the smallest scale factor and SQLStorm-12, while the number of posts and comments is 20× larger. This characteristic makes cardinality estimation and query optimization more interesting as the data's scale and distribution change. Alternatively, filtering records by creation date can reduce the 220 GB StackOverflow dataset to smaller scale factors. Over the last 17 years, the dataset has grown by 5 GB to 20 GB per year, scaling smoothly from a few gigabytes to two hundred while retaining real-world characteristics.

Next, we investigate which data types are used and their storage size. Table 1 reports the number of integer, date, and text columns, including their size on SQLStorm-220. 44 of the 77 columns are numerical, mostly 32-bit and 16-bit integers for encoding primary and foreign key relationships or counters. However, these columns account only for 5.1 % of the total data size. Most of the data is stored in text columns, which make up for almost 92 % of the consumed space, even though only 27 % of columns use a character data type. The longest text contains 116,293 characters, while the average string length is 338 characters. This distribution matches the observations from real-world data where variable-sized strings are dominant [39, 49, 56, 59]. Furthermore, the strings also contain Unicode characters; synthetic benchmarks like TPC-H and TPC-DS lack these characteristics.

3.2 Query Generation

Large language models are trained on a vast amount of data to produce human-like output and solve complex tasks. This includes public source code or questions from programming-related platforms like StackOverflow [8, 67]. The 2024 StackOverflow Developer Survey reports that SQL is the second most popular programming language among professional developers [2]. As a result, LLMs have seen many programming problems and solutions, including

Table 2: The prompts used to generate the query set on the StackOverflow dataset. Besides the prompt, we also include the database schema in the request. The table also shows the number of queries that pass each step of the selection process; queries from simple prompts are cheaper and more likely to be selected.

	Prompt (GPT-4o-mini)	Parse 1	Parse 2	Exec.	Yield	Cost
P1	`Generate an interesting and elaborate SQL query for performance benchmarking, potentially including constructs such as outer joins, (correlated) subqueries, CTEs, window functions, set operators, complicated predicates/expressions/calculations, string expressions, and NULL logic.`	1927	2690	2533	50.7 %	$2.38
P2	`Generate an interesting and elaborate SQL query for performance benchmarking.`	2472	3461	3196	63.9 %	$2.19
P3	`Generate a SQL query for performance benchmarking.`	3188	4205	3791	75.8 %	$1.83
P4	`Generate a simple SQL query for benchmarking.`	3343	3694	3690	73.8 %	$1.54
P5	`Generate an interesting and elaborate SQL query for performance benchmarking, potentially including constructs such as outer joins, (correlated) subqueries, CTEs, window functions, set operators, complicated predicates/expressions/calculations, string expressions, and NULL logic. Incorporate obscure semantical corner cases and unusual or even bizarre SQL semantics.`	1430	2060	1848	37.0 %	$2.72
P6	`Generate an interesting and elaborate SQL query for benchmarking string processing.`	1895	2676	1987	39.7 %	$2.41
P7	`Generate an interesting and elaborate SQL query for performance benchmarking, potentially including constructs such as outer joins, (correlated) subqueries, (recursive) CTEs, window functions, set operators, complicated predicates/expressions/calculations, string expressions, and NULL logic.`	951	1432	1206	24.1 %	$2.66
Total		15206	20218	**18251**	52.1 %	**$15.73**

SQL. Therefore, asking an LLM for random queries will sample the real-world distribution of queries it was trained on and produce a realistic workload.

OpenAI and Anthropic provide batch APIs, allowing to run the models in an asynchronous fashion and reduce cost by 50%. This allows for fast, scalable, and cost-effective generation of large sets of SQL queries. Not all queries will be syntactically correct or semantically meaningful, but the same is true for queries written by humans. Our selection process will find useful queries for benchmarking and remove the rest. The queries' semantics are of secondary importance in an analytical benchmarking context, provided they process a sufficient volume of data and incorporate various SQL features, such as complex joins, window functions, recursion, and other advanced operations.

Since LLMs are a black box, finding the right prompt is more of an art than a science. We tested several prompts and found that the following prompt provides good results and diverse SQL queries:

P1
```
Generate an interesting and elaborate SQL query for performance
benchmarking, potentially including constructs such as outer
joins, (correlated) subqueries, CTEs, window functions, set
operators, complicated predicates/expressions/calculations,
string expressions, and NULL logic.
```

From our experience, adding hints to the prompt can help the model explore a broader range of possibilities. However, sometimes hints may limit the search space, causing the LLM to over-commit on the explicitly enumerated features. It requires careful consideration of what information is given to the model. However, we must provide the LLM with enough details to generate correct and meaningful queries. To achieve this, we also included the database schema as `CREATE TABLE` statements as a suffix to the main prompt:

PS
```
Do not explain the query, only output one SQL query. Use
the following StackOverflow schema: CREATE TABLE ...
```

The schema includes foreign key relationships and primary keys, allowing the LLM to generate queries that join multiple tables and find the correct join conditions. Additionally, we added some comments to the schema, clarifying the structure of the database and the five enum tables.

While *P1* generates a diverse set of queries comparable to TPC-DS in terms of complexity, it has two significant drawbacks: (1) The queries are more complex than the majority observed in real-world workloads [56]. Hence, simple statements using basic SQL features and accessing only one or two tables should also be part of the benchmarks. (2) The prompt does not cover advanced SQL constructs like deeply correlated subqueries or recursive CTEs. While these constructions are not prevalent in actual workloads, they are still important for evaluating the system's robustness and testing edge cases that database engineers might not have considered.

We modified the prompt step-by-step to generate simpler and more complex queries. Table 2 lists the final seven prompts we used for creating SQLStorm. *P2* - *P4* gradually remove hints from the prompt to make the statements simpler. *P5* and *P7* instruct the LLM to generate unusual SQL constructs to test corner cases and recursion. As strings are prominent in real-world datasets and workloads, *P6* explicitly asks for queries focused on string operations. We call OpenAI's GPT-4o-mini model for each prompt with a batch of 5,000 requests. A batch is processed within one day (typically much faster) and costs $2 on average. The cost between batches differs, as simple prompts produce shorter queries and require fewer input and output tokens than complex prompts.

Next, let us discuss two example queries generated by the LLM using the prompts *P1* and *P4*. The first query in Figure 2 finds all users with posts from the last year and outputs statistics about their badges and closed posts. The second query retrieves the top 10 highest-scoring posts, along with the author and comments. We can observe a significant difference in the complexity: while the first query uses window functions, filtered aggregations, and multiple CTEs, the second only uses simple SQL constructs.

3.3 Query Cleaning & Rewriting

Using our generation process, we obtained 35,000 queries on the StackOverflow dataset. However, these queries include duplicates, references to the current time, or comments where the LLM tries to explain the query. Sometimes, the output also contains multiple

```sql
WITH RankedPosts AS (
       SELECT p.Id AS PostId, p.Title, p.CreationDate, p.Score,
              p.ViewCount, p.OwnerUserId, u.Reputation,
              ROW_NUMBER() OVER (PARTITION BY p.OwnerUserId
                                 ORDER BY p.CreationDate DESC) AS rn
       FROM Posts p JOIN Users u ON p.OwnerUserId = u.Id
       WHERE p.CreationDate >=
              (CAST('2024-10-01' AS DATE) - INTERVAL '1 year')),
    UserBadges AS (
       SELECT b.UserId,
              COUNT(*) FILTER (WHERE b.Class = 1) AS GoldBadges,
              COUNT(*) FILTER (WHERE b.Class = 2) AS SilverBadges,
              COUNT(*) FILTER (WHERE b.Class = 3) AS BronzeBadges
       FROM Badges b GROUP BY b.UserId),
    ClosedPostCount AS (
       SELECT ph.UserId, COUNT(*) AS ClosedPosts
       FROM PostHistory ph
       WHERE ph.PostHistoryTypeId = 10 GROUP BY ph.UserId)
SELECT rp.PostId, rp.Title, rp.CreationDate, rp.Score,
       rp.ViewCount, rp.Reputation,
       COALESCE(ub.GoldBadges, 0)   AS GoldBadges,
       COALESCE(ub.SilverBadges, 0) AS SilverBadges,
       COALESCE(ub.BronzeBadges, 0) AS BronzeBadges,
       COALESCE(cpc.ClosedPosts, 0) AS ClosedPosts,
       CASE WHEN rp.Score > 100 THEN 'High Score'
            WHEN rp.Score BETWEEN 50 AND 100 THEN 'Medium Score'
            ELSE 'Low Score' END    AS ScoreCategory
FROM RankedPosts rp LEFT JOIN UserBadges ub
       ON rp.OwnerUserId = ub.UserId
     LEFT JOIN ClosedPostCount cpc
       ON rp.OwnerUserId = cpc.UserId
WHERE rp.rn = 1
ORDER BY rp.Score DESC, rp.ViewCount DESC;
```

Figure 2: Complex query generated by prompt *P1*. As instructed, the prompt constructs outer joins, window functions, and complicated expressions, like filtered aggregates.

```sql
SELECT U.DisplayName AS UserName, P.Title AS PostTitle,
       P.Score AS PostScore, C.Text AS CommentText,
       C.CreationDate AS CommentDate
FROM Posts P JOIN Users U ON P.OwnerUserId = U.Id
     LEFT JOIN Comments C ON P.Id = C.PostId
WHERE P.PostTypeId = 1
ORDER BY P.Score DESC LIMIT 10;
```

Figure 3: Simple query generated by prompt *P4*. Without additional instructions, GPT-4o-mini generates queries with few joins and no complex SQL constructs.

queries or DML statements instead of the expected `SELECT` statements. In order to make the queries suitable for performance benchmarking, our next step is to clean and rewrite the LLM's output. We start by deleting duplicated queries; on the StackOverflow dataset, we found 1287 duplicates, all generated by the prompt *P4*. Next, we remove comments in the query text and extract the first `SELECT` statement if multiple are present. Furthermore, we replace `current_time` and similar functions with a fixed timestamp to ensure the queries are reproducible. For example, the query in Figure 2 originally used `current_date` in line 7; we replaced it with a fixed date, `2024-10-01` to prevent empty scans in the future.

After rewriting the queries, we test how many can be parsed by PostgreSQL, Umbra, and DuckDB. We consider a query parsable if at least two of the three systems can execute the query on an empty database. Table 2 reports the number of parsable queries for each prompt under Parse 1. Unsurprisingly, the simple prompts *P3* and *P4* yield more correct queries. More than 60 % of the queries

Table 3: The number of queries that are parseable and executable by PostgreSQL, Umbra, and DuckDB. Almost half of the initial queries (Parse 1) are compatible with two systems. After rewriting the queries (Parse 2), even more queries can be parsed by . The queries that pass the SQLStorm selection process are highlighted in green.

Parse 1	9427	5644	32	103	166	381	1875	16085
Parse 2	17016	2157	39	137	144	379	1138	12703
Exec.	10910	1164	24	2747	63	3312	31	1967

are parsable, while for complex prompts like *P5* and *P7*, only less than 30 % qualify. On average, 43 % of the queries can be parsed.

We noticed that GPT-4o-mini primarily generates queries in the PostgreSQL dialect. This may be explained by the fact that PostgreSQL is widely used in open-source projects and its dialect follows the SQL standard more closely than the dialects of some commercial systems. Table 3 indicates that all three systems together can parse 9,427 queries. However, several queries are incompatible with DuckDB: 5644 queries only run in PostgreSQL and Umbra. The two systems implement a relatively obscure feature from the SQL standard that allows them to omit columns from the group by clause if they are functionally dependent on another column in the clause, which is used by the LLM. DuckDB, in contrast, can parse 1875 queries that neither PostgreSQL nor Umbra can execute. For instance, DuckDB supports `SUM` aggregations on boolean values.

Ideally, we want to have queries that run in all three systems and avoid dialect-specific constructs. In order to make queries compatible between systems, we use the LLM again to rewrite the query text with the following prompt:

PF
```
Make the following PostgreSQL query more compatible with
different SQL dialects. The query might contain '::' casts,
rewrite them to standard SQL. Remember to put all ungrouped
columns and columns that appear in window functions into
the group by clause. Do not explain the query, only output
the converted query.
```

The prompt instructs the LLM to make the query compatible with standard SQL. We specifically address PostgreSQL's shorthand casts and missing attributes in `GROUP BY` clauses, as these are common patterns. After the prompt, the query text is appended.

In the second pass, GPT-4o-mini rewrites 24,692 queries; these queries could not be parsed by all three systems or contain a shorthand cast. It fixes several incompatibilities: adding missing columns into the group by clause, rewriting short hand casts to function-style casts, and converting boolean expressions in `SUM` aggregations to integer values using a `case` expression. In addition, the LLM was also able to fix several problems we did not mention in the prompt. As a result, after the second pass, 20,218 queries qualify, as shown in Table 2 (Parse 2), an increase of more than 5,000 queries. More importantly, the number of queries that all three systems can parse almost doubles to 17,016 queries (cf. Table 3), and queries not compatible with any system are reduced by 20 %. However, 869 queries became incompatible after the LLM attempted to fix them; we revert these queries to the original version.

Cleaning and rewriting makes the queries suitable for a benchmark. Rewriting queries with the same LLM significantly improves

the compatibility with different systems and fixes some errors in the queries. The cost of this step is similar to generating the queries: the first LLM pass costs $10, and the second pass $6.

3.4 Query Selection

After rewriting, we select the final set of queries that will be used for the benchmark. First, we remove all queries that cannot be parsed by at least two systems, then test if each query runs on the SQLStorm-1 dataset. The dataset contains roughly 1 GB of data, and a query is executable if one of the three systems finishes it in under 1 second. We require two systems to be able to parse the query, as we want to avoid system-specific language constructs. For execution, one system is sufficient: if a query runs in one system but times out in another, it indicates that the system does not optimize the query well and is interesting for performance evaluation. The timeout eliminates queries that are too expensive to execute, such as queries with huge intermediate results due to cross-products.

Table 2 lists the results of this step: 18,251 of the 20,218 queries from the parsing step qualify and remain in the benchmark. Yield rates in the different batches differ significantly, while from our first prompt, *P1*, 50 % of the queries remain, only 24 % of the queries from *P7* are parsable and executable. *P3* and *P4* have the highest yield rates; more than 70 % of the queries qualify. Recall that *P4* generated 1287 duplicated queries, which we eliminated during the cleaning step. If we do not consider these duplicates, the prompt yields more than 99 % of the queries.

Of the 17,016 queries that were parsable by all three systems, 10,910 queries run in all three systems, see Table 3. PostgreSQL's query engine is slower than Umbra's and DuckDB's, and more queries time out. Consequently, 2,747 queries can only be executed by Umbra and DuckDB, and 3,312 queries only by Umbra.

Admittedly, developing this benchmark had a major impact on Umbra's compatibility, prompting us to implement missing features and resolve several bugs. Section 4.3 analyzes our changes to Umbra caused by SQLStorm in more detail. To summarize, the SQLStorm methodology generates a large-scale analytical benchmark at a negligible cost. We spend less than $16 and obtain over 18,000 queries compatible with PostgreSQL, Umbra, or DuckDB.

3.5 Query Validation & Correctness

In this chapter, we evaluate the correctness of the generated queries and assess whether they are suitable for result validation. We focus on two main aspects: whether the generated queries address real-world scenarios and whether they are semantically correct. In addition, we offer a validation set to verify the correctness of the results, similar to benchmarks like TPC-H and -DS.

To evaluate the correctness, we randomly sampled 100 queries from the final set and manually categorized them into three groups: *(1) correct* queries accurately join and filter the data, producing semantically meaningful and logically sound results, *(2) almost correct* queries are accurate in terms of join and filter conditions but contain minor errors, e.g., using a count where a distinct count would be semantically more meaningful, and *(3) incorrect* queries are semantically flawed due to incorrect or missing join conditions, empty filter criteria, or erroneous computations. A manual inspection of the sampled queries revealed the following distribution: 53% of the queries are correct, 34% almost correct, and 13% incorrect.

Table 4: Number of queries with identical results in at least two systems. In some cases, one system differs from the other two; for example, DuckDB returns different results than PostgreSQL and Umbra for 1,932 queries (-). Additionally, 5,664 queries yield inconsistent outputs across all systems.

match in ≥ 2 systems			one system differs from the other			
			-	-	-	∅
9519	86	685	1932	257	108	5664

Manually classifying the queries is tedious; therefore, we use a heuristic to check the queries automatically. We inspect the query plan generated by Umbra and test if all joins use one of the 19 foreign key relationships or reasonable attributes such as user names. 69 % of the queries join only along foreign keys, and 11 % contain reasonable joins. The remaining 20 % of the queries contain incorrect or missing join conditions. However, this automatic classification reports some false positives as queries are semantically meaningful, even though they use an incorrect join attribute or perform a cross-join. For the manually reviewed sample, eight queries were falsely classified as incorrect by the heuristic.

Combining the two analyses, we conclude that the majority of the queries are correct or almost correct; however, a notable portion contains minor or, occasionally, significant errors. Nevertheless, we choose to retain these queries, as errors occur in real-world scenarios as well: Users interactively write and refine SQL queries step-by-step, leading to underspecified queries, e.g., missing join or filter conditions [59]. Moreover, AI assistants are becoming increasingly common in the industry, helping users generate SQL snippets, optimize existing queries, and automatically correct errors [6, 46, 51]. Recent research indicates that while human experts outperform LLMs in SQL query correctness, they still make mistakes on more than 7 % of tasks on the BIRD dataset [29, 32]. This error rate rises for less experienced users [56], suggesting that a query set containing mistakes more accurately reflects reality than a perfectly error-free set. Machine-generated SQL and non-expert written queries are common in practice and lead to erroneous or underspecified queries [56, 59].

A manual review of the queries shows that they cover various realistic analytical tasks. Common patterns include (a) ranking users, posts, or comments based on various scoring metrics, often requiring complex computations and multi-table joins; (b) retrieving recent records from different tables; (c) enriching tables with additional statistics or detailed information; and (d) aggregating data across multiple tables to produce summaries or reports. In our analysis, we found that many queries answer similar questions. However, they differ in part greatly in their implementation, the used attributes and tables, and the SQL constructs.

Lastly, we assess the suitability of the generated queries for result validation. We computed the result sets for all queries on the three systems: 12,587 produce identical results in at least two systems, while the remaining 5,664 exhibit inconsistent or nondeterministic behavior. Table 4 shows the number of queries with matching outputs for different combinations of the three systems.

We observe that DuckDB differs from PostgreSQL and Umbra more often because Umbra closely follows PostgreSQL's semantics [52]. This highlights subtle differences in SQL interpretation and implementation across systems; such queries are particularly valuable to database engineers as they expose bugs and incompatibilities.

SQLStorm generates semantically meaningful queries that are well-suited for result validation. Yet, it includes erroneous and underspecified queries, reflecting the reality of machine-generated and non-expert written SQL commonly found in practice [56, 59].

3.6 Complexity Classification

SQLStorm has diverse queries, including simple ones with only a few joins and complex queries with advanced SQL constructs. For developing new database systems or research prototypes, supporting all queries upfront will be challenging. We therefore classify the queries into three complexity classes: low, medium, and high. Low complexity queries can be supported by implementing basic operations and require only a small subset of the SQL standard. Researchers can use them to evaluate prototype systems with limited functionality. The medium complexity queries require additional operators such as window functions, set operations, and outer joins. The high complexity class include the remaining queries, demanding advanced features such as recursion, arrays, json, and complex query unnesting (e.g., mark joins [43]).

We use the matrix in Table 5 to determine the complexity of a query based on the execution traces collected by Umbra. For instance, medium complexity queries only use operators from the low complexity class in addition to Window and SetOperation. Similarly, the expressions must not use complex regex or json functions. We also limit the number of joins and aggregations, as deeper query trees require sophisticated join ordering and cardinality estimation for efficient execution. If a query does not fulfill one condition, it falls into a higher class.

We developed the assignment based on how many queries qualify in each category and our experience developing the required features. The medium complexity class is the largest with 10,195 queries, followed by 4,596 simple queries and 3,460 high complexity queries. Table 6 shows the average number of operators per query and SQL text length in the different complexity classes. As query complexity increases, both the average query text length and the number of operators and expressions grow accordingly. Low complexity queries are roughly 3 times smaller regarding code and plan size. The table also includes statistics for the TPC benchmarks: TPC-H is, on average, more complex than the low class and simpler than the medium class. TPC-DS is similar to the high complexity class, except that it uses more expressions per query.

Table 5: Matrix to determine the complexity of a query.

Complexity	low	medium	high
Operators	TableScan, Join, GroupBy, Select, Map	Window, SetOperation	Iteration, RegexSplit, ArrayUnnest
Expression Categories	comparison, cast, case, arithmetic (simple)	nulls, strings, date, array, arithmetic (complex)	regex, json
Types	bool, int, text, numeric, float, date, timestamp	arrays	record, json, timestamptz
Join Types	inner, outer	semi, anti, single	mark
#Joins	<= 3	<= 8	
#GroupBy	<= 1	<= 3	

Table 6: Average number of operators, expressions, and query text size. Basic relational operators are present in all benchmarks, whereas advanced features like recursion (Iteration), arrays, regexes, and JSON are unique to SQLStorm.

	SQLStorm			**TPC**		**Redshift [56]**	
	low	med.	high	-H	-DS	[60s,∞)	Total
Query Text	334	1168	1347	472	1370	n.a.	4396
Operators	6.42	14.41	22.08	9.09	19.27	10.74	3.50
TableScan	2.87	4.94	5.75	3.68	7.22	6.51	2.03
Join	1.87	4.17	6.20	2.82	6.30	2.28	0.53
Sort	0.99	1.03	1.10	0.82	0.83	0.34	0.12
GroupBy	0.68	2.33	4.01	1.32	2.08	0.72	0.52
Select	0.004	0.97	1.78	0.09	1.17	n.a.	n.a.
Window	0	0.64	1.01	0	0.20	0.2	0.04
SetOperation	0	0.004	0.003	0	0.28	n.a.	n.a.
ArrayUnnest	0	0	0.49	0	0	n.a.	n.a.
Iteration	0	0	0.05	0	0	n.a.	n.a.
RegexSplit	0	0	0.001	0	0	n.a.	n.a.
Expressions	6.42	29.12	35.61	12.86	37.51	n.a.	n.a.
string modif.	0	0.13	0.77	0.23	0.18	n.a.	n.a.
string match.	0	0.07	0.18	0.32	0.01	n.a.	n.a.
regex	0	0	0.004	0	0	n.a.	n.a.
json	0	0	0.003	0	0	n.a.	n.a.

The queries from the low complexity class use fewer operators as we limit the number of joins to 3. Sorts often appear as the last operator in the query tree to order the output. However, in more complex queries they are also used to select the top-k results on intermediate computations. The Iteration operator implements recursive CTEs in Umbra; 157 queries from *P7* in SQLStorm use this feature. ArrayUnnest and RegexSplit iterate through arrays/regex matches; besides set operations, these operators are the least common. SQLStorm also features regex and JSON expressions.

van Renen et al. also report numbers on the query complexity in their analysis of the Amazon Redshift fleet [56]. We extracted their results from the paper to compare with SQLStorm. The queries from the entire Redshift fleet are, on average, less complex than those in the low complexity class. However, Redshift is closer to the medium and high class if we only consider the long-running queries. The average number of table scans in long-running queries exceeds the high complexity class.

3.7 Query Diversity

SQLStorm contains a large number of queries, raising the important question: Is the query set genuinely diverse or minor variations of a few templates, like TPC-H and -DS? To assess the workload's diversity, we compute the number of unique query plans and execution traces produced in Umbra. A query plan is unique if its

Table 7: Number of unique query plans and execution traces recorded in Umbra on SQLStorm and TPC-H/DS.

Unique / Total	SQLStorm	TPC-H	TPC-DS
Plans	12,384 / 18,251	30 / 44,000	256 / 51,500
→ Trees	9,869 / 18,251	24 / 44,000	154 / 51,500
→ Operators	26,667 / 267,401	153 / 416,000	1,028 / 1,317,151
Traces	10,692 / 18,251	206 / 44,000	769 / 51,500

tree structure differs or contains a new instance of an operator. The tree's shape is defined by the exact layout of the operators and their operator type (e.g., join, map, groupby, etc.). We consider additional details for operator instances, such as the number of attributes used, the specific expressions evaluated, and the join or aggregation type. Umbra's optimizer produces 18,233 plans with 267,401 operators on SQLStorm. Among these are 9,869 distinct query trees and 26,667 unique operator instances, resulting in 12,384 unique queries. Compared to TPC-H and TPC-DS, SQLStorm offers greater query plan diversity. The template-based benchmark corpora cannot match the textual and structural diversity of LLM-generated queries.

For execution traces, the trend is similar: Umbra records 10,692 distinct traces, while TPC-H and TPC-DS have 10× fewer despite having almost 3× more queries (44 K and 51.5 K, respectively). The traces capture dimensions such as runtime, memory usage, number of scanned rows, allocated memory, and operator counts.

Snowflake and Redshift published execution traces for their cloud data warehouses [56, 60]. Figure 4 compares the runtime distribution of SQLStorm to these traces. Unlike artificial benchmarks, which lack a broad distribution and exhibit more pronounced spikes, SQLStorm's runtime distribution more closely resembles real-world patterns. Snowflake and Redshift report queries running for over 10 hours; for practicality, we limited execution time to 10 minutes in our measurements. Nevertheless, some SQLStorm queries hit this limit, so the workload also includes long-running queries.

The SQLStorm methodology generates queries with diverse query plans and execution traces. We observe complex queries with up to 61 operators and 72 expressions for the StackOverflow dataset. The queries cover a wider range of features than standard benchmarks like TPC-H or TPC-DS and stress query engines in multiple dimensions, resulting in more than 10,000 unique query plans and execution traces. The runtimes range from milliseconds to several minutes, similar to Redshift and Snowflake.

3.8 Exploring Other Large Language Models

The state of the art for LLMs is changing rapidly: new models are released monthly, and funding has increased eightfold between 2022 and 2023, reaching $2.52 billion [37]. Training data sets, the number of parameters, the context window, and underlying hardware all grow rapidly. We anticipate significant advancements over the next decade, and the models' capabilities will continue to improve. The concrete benchmark we present in this paper is just the first version, and we plan to extend it in future releases. Upcoming models and prompt tuning will yield new queries with features not yet covered and increase the query text size. The SQLStorm methodology is the first step into systematically exploiting LLMs and Foundation Models for database benchmarking.

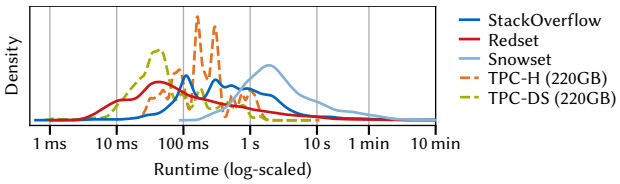

Figure 4: Runtime distribution of SQLStorm, compared to real-world workloads (Redset & Snowset) and artificial benchmarks (TPC-H & -DS).

Many companies offer closed and open-source models: OpenAI (GPT), Anthropic (Claude), Meta, and Google (Gemini), to name only a few. The models differ in their training data, number of parameters, and capabilities and are optimized for different tasks and price points. When we generated the queries (October 2024 through February 2025), OpenAI's GPT-4o-mini model was the most cost-effective LLM in terms of query yield. Table 8 compares LLMs for generating SQL benchmark queries.

The cheaper models, GPT-4o-mini, Claude 3.0 Haiku, and Claude 3.5 Haiku, produce query sets with similar complexity, i.e., the average number of operators is almost identical. The latest Haiku model achieves the highest query yield but is three times more expensive than GPT-4o-mini. We also include the older GPT-3.5-turbo model in the comparison. While many of the model's generated queries qualify, the queries are also significantly shorter and less complex; the results are comparable to *P3* or *P4* on GPT-4o-mini. Even though Anthropic's models are priced higher than OpenAI's GPT-4o-mini, the overall costs are comparable, as Claude supports prompt caching, reducing the price by roughly two-thirds.

On average, the GPT-4o queries use two more operators than the other models. Claude 3.5 Sonnet generates the highest number of operators yet. However, as the two models generate more complex queries, they are also more expensive. GPT-4o and Claude 3.5 Sonnet cost 10 times more than GPT-4o-mini. Therefore, we decided to use GPT-4o-mini since it is the most cost-effective model.

To our surprise, Anthropic's Claude 3.5 Sonnet model performs worst of all LLMs regarding the number of qualifying queries. Only 15.4 % of the 1,000 generated queries pass the selection process and can be executed by one of three database systems. We found that the queries generated by the model often contain the same error: Sonnet combines aggregations and window functions but "forgets" to add the columns used by the window to the `GROUP BY` clause.

Table 8: Comparison of OpenAI's GPT and Anthropic's Claude models for generating benchmark queries using *P1* (cf. Table 2). While GPT-4o-mini is the most cost-effective, newer and larger models generate more complex queries.

Model	Yield	Cost	Avg length	Avg ops.
GPT-3.5-turbo (2024-01-25)	66.7 %	$3.44	620 B	8.7
GPT-4o-mini (2024-07-18)	49.7 %	$2.38	1221 B	14.1
GPT-4o (2024-07-18)	52.3 %	$26.27	1342 B	16.2
Claude 3.0 Haiku (2024-03-07)	38.1 %	$3.29	1300 B	13.4
Claude 3.5 Haiku (2024-10-22)	57.9 %	$6.73	1110 B	13.6
Claude 3.5 Sonnet (2024-10-22)	15.4 %	$30.48	1575 B	20.9

Prompt *PF* attempts to resolve this issue in a second pass through GPT-4o-mini. However, the queries are too complex for the small model to fix the mistakes. Rewriting the queries with the Claude 3.5 Sonnet might be more effective but increases the cost significantly.

SQLStorm v1.0 solely relies on GPT-4o-mini for query generation. However, we observe a trend that newer and larger models are capable of generating more complex queries. For example, GPT-4o generates twice as many operators as GPT-3.5-turbo and doubles the code size. Claude 3.5 Sonnet gives an idea of the potential of the next generation of LLMs. We will explore these models in future releases of SQLStorm and optimize the prompts further.

3.9 Availability and Future Releases

SQLStorm serves as both a methodology and a new database benchmark. With this paper, we release version 1.0 of SQLStorm, including all associated artifacts – the query sets, benchmark results, and recorded traces – and provide scripts to run the generated queries in more than six database systems. Additionally, we also release the source code and prompts used to generate, rewrite, and select queries compatible with PostgreSQL, Umbra, and DuckDB. The entire process is fully automated, enabling the creation of large-scale benchmarks without human intervention.

For future releases of SQLStorm, we plan to incorporate more datasets, prompts, and new models to increase the workload's diversity and complexity. New LLMs and prompts are selected based on their cost, query quality, and whether they can generate queries with new features, e.g., query tree shapes and operator instances. We encourage other researchers and database developers to propose datasets, prompts, or query patterns to better reflect real-world workloads. For instance, SQLStorm can be extended to support graph workloads, streaming data analysis, and other domains.

4 EXAMPLE USE CASES

The SQLStorm benchmark is designed to evaluate the performance of database systems using a real-world dataset and a realistic workload. In this section, we will explore various use cases for SQLStorm.

We conduct all benchmarks on a server with an AMD EPYC 9454P CPU (48 cores, 96 threads) and 384 GB of RAM. We evaluate four database systems implementing the PostgreSQL dialect: Umbra, PostgreSQL, DuckDB, and Tableau Hyper. In addition, we also test two commercial systems, DBMS X and Y, which adopt a different SQL dialect. We list the different database systems and their versions in Table 9. The database systems run in privileged Docker containers to improve reproducibility. A Python script sends queries to the database and measures the end-to-end latency. Except for PostgreSQL, all systems use a columnar storage format and implement vectorized or compilation-based query execution.

4.1 Improving Compatibility and Robustness

First, we investigate the systems' compatibility and robustness when executing the queries. We run all queries on SQLStorm-1 and report the number of successful and failed queries in Table 9. Queries might fail for multiple reasons, such as syntax errors, execution failures (e.g., arithmetic overflows, scalar subqueries with multiple results, recursion depth exceeded, etc.), timeouts (we cancel a query if it does not finish within 10 seconds), out-of-memory (OOM), or fatal crashes. We consider a query fatal if it crashes the database system or the system is not responsible within 100 seconds after the query exceeds its time limit. Additionally, we report the percentage of queries that return incorrect results, considering only those with deterministic outputs (see Table 4).

Table 9: Database systems used for evaluating different use cases for SQLStorm. The table also reports the number of queries each system can execute on SQLStorm-1.

System (Version)	Success	Syntax Error	Exec. Error	Timeout + OOM	Crash	Differ. Result
Umbra (25.01)	18,165	18	34	34	0	2.0 %
→ before SQLStorm	16,068	2,053	9	5	116	3.2 %
PostgreSQL (17.0)	15,731	87	13	2,420	0	0.9 %
DuckDB (1.2.0)	15,208	2,164	168	711	0	15.5 %
Hyper (0.0.21200)	13,316	4,727	37	123	48	1.7 %
DBMS X	2,938	14,566	307	440	0	8.3 %
→ rewritten	12,451	3,646	1,093	1,061	0	10.3 %
DBMS Y	8,744	9,200	0	307	0	15.1 %
→ rewritten	13,138	4,647	20	442	4	25.2 %

None of the six systems can execute all queries successfully. Umbra has the highest success rate with 18,165 queries, followed by PostgreSQL with 15,731 queries. However, both systems still experience syntax errors as they do not support some features other systems do. Timeouts are common in PostgreSQL due to its slower performance on analytical queries.We have already improved the current version of Umbra with the SQLStorm queries. Before these changes, Umbra failed for over 2,000 queries and crashed for 116.

For the other systems, compatibility is a bigger issue. Hyper, for instance, fails for 15.9 % of the queries as it lacks the `string_agg` and `string_to_array` functions. More interestingly, 48 queries crashed Hyper, requiring a system restart to continue the benchmark. For the large SQLStorm-220 dataset, we also observe database crashes in DuckDB and Umbra. Only the commercial DBMS X and PostgreSQL are stable and do not experience any fatal queries.

While the first four systems can parse most queries, the commercial systems struggle with the SQLStorm. Their SQL differs from the PostgreSQL dialect. As a result, DBMS X has syntax errors on more than 14,500 queries. To evaluate these systems on a large set of queries, we rewrite the benchmark using GPT-4o-mini to the system-specific dialect using the following simple prompt:

PR
```
Convert the following PostgreSQL query to <DIALECT> syntax.
Remember to put all ungrouped columns and columns that
appear in window functions into the group by clause. Do
not explain the query, only output the converted query.
```

The prompt works well for both commercial systems; we replace `<DIALECT>` with the system dialect name and let the LLM translate the queries. With the rewritten queries, every system can execute at least two-thirds of SQLStorm successfully. Of course, the LLM might accidentally change the semantics of the query; for DBMS X, the number of incorrect queries increases by 2 %, while DBMS Y sees an increase of 10 %. Consequently, the rewritten queries are not guaranteed to perform the same computations as the original queries, and comparing the runtimes requires some caution.

4.2 Evaluating Cardinality Estimation

Cardinality estimation is a crucial part of query optimization. Accurate estimates are decisive for finding a cheap and robust query

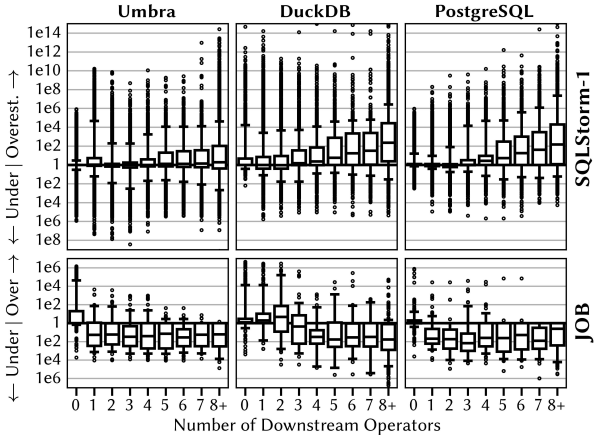

Figure 5: Estimation error in SQLStorm-1 and JOB. Each box plot shows the error distribution for operators with the given number of downstream operators. While underest. are more common on JOB, overest. dominate in SQLStorm as it combines different operators. Accurately estimating joins alone is insufficient for precise cardinality estimations.

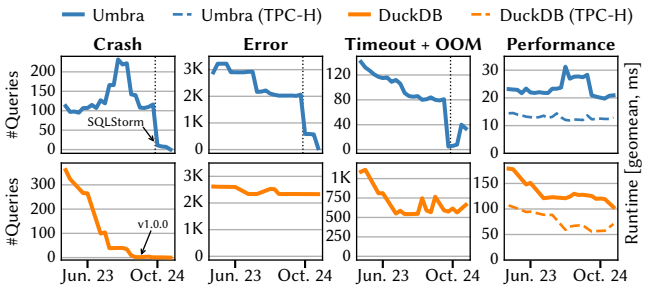

Figure 6: SQLStorm on Umbra and DuckDB over the last 2 years. In Oct. 24 we created SQLStorm and started fixing bugs in Umbra (dotted line). The dashed line in the last plot shows the geometric mean runtime on TPC-H 10 GB. SQLStorm allowed Umbra to catch up with DuckDB in terms of crashes and it reduced the number of errors significantly.

plan, as mistakes during query optimization can accumulate [11, 30, 38, 42]. In this section, we evaluate the accuracy of Umbra's, DuckDB's, and PostgreSQL's query optimizer by comparing the estimated cardinalities from the query plans with the actual cardinalities. Figure 5 shows the estimation error for different levels in the query tree for the three systems. We compute the factor by which the estimate and the exact cardinalities differ and distinguish between under- and overestimation. The number of nodes in all subtrees of an operator determines the number of downstream operators. For instance, while table scans have no children and their number of downstream operators is always 0, a join operator has two subtrees, and we sum up the number of operators in both trees.

All three systems perform reasonably well for up to three downstream operators, maintaining a median estimation error close to 1. However, as the number of downstream operators increases, the error grows as well (note the logarithmic scale). Misestimations propagate through the tree, making estimating the cardinality for operators higher up more challenging. For eight or more operators, DuckDB and PostgreSQL misestimate the cardinality by at least two orders of magnitude for the majority of queries. Umbra, in contrast, is more robust and keeps the median error below 3 for all depths. Nevertheless, a significant fraction of the queries are over- and underestimated by more than four orders of magnitude, even for a few downstream operators. DuckDB experiences errors as severe as 14 orders of magnitude.

We included the estimation errors on the Join Order Benchmark for comparison. A different pattern emerges in JOB: underestimations are more common than overestimations, and misestimations exceeding four orders of magnitude are less frequent. The combination of different operators, in particular, aggregations and joins, makes cardinality estimation more difficult on SQLStorm and results to more extreme errors. In addition, outer joins and incorrect join conditions skew the estimation error in Umbra and PostgreSQL towards overestimations. JOB lacks this characteristic; it only includes inner joins and aggregations at the end of long join pipelines, missing the complexity of real-world workloads [28].

For robust query optimization, more accurate cardinality estimate algorithms are needed. SQLStorm is a good starting point for investigating these aspects further. Large-scale benchmarks help to identify symptomatic weaknesses and edge cases in query optimizers and cardinality estimators. Furthermore, database developers can use SQLStorm to identify the impact of changes made in the query optimizer on the plan quality. For instance, several lines of recent work focus on making cardinality estimation and query processing more robust [24, 47, 62, 65, 66]; SQLStorm can be used to evaluate the effectiveness of these approaches.

4.3 Regression Testing

Given the complexity of modern database systems, regression testing plays a crucial role in maintaining the integrity and performance of database applications [17, 61]. For example, Umbra and DuckDB use TPC-H and TPC-DS to ensure reliability, efficiency, and functionality across versions. However, these benchmarks have limited scope, and the small query set may not reveal all issues. SQLStorm tests a broader range of features and unveils performance gains and losses on a large workload.

Figure 6 investigates how the robustness and compatibility of Umbra and DuckDB changed over time. In January 2023, Umbra could execute only 15,102 queries on SQLStorm-1. Until September 2024, the number of successful queries increased to 16,067 as we implemented more features and fixed bugs in Umbra. To our surprise, the number of fatal queries that cause system crashes temporarily increased to 232 at the start of 2024. We fixed these issues; however, it took several months to discover and resolve them. After introducing SQLStorm, we could fix 90 % of the crashes within one month, and until January 2025, we resolved all fatal queries and most of the timeouts and exceptions on SQLStorm-1. At the time of writing this paper, only 86 queries remain that Umbra cannot execute. SQLStorm greatly improves development speed and helps to identify and fix issues early.

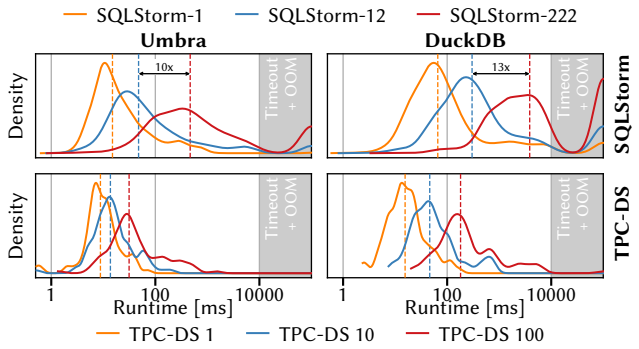

Figure 7: Scalability of Umbra and DuckDB on SQLStorm and TPC-DS. The dashed lines show the median runtime. Queries in TPC-DS are closer to the median and timeouts are almost non-existent. Scaling on SQLstorm is more challenging.

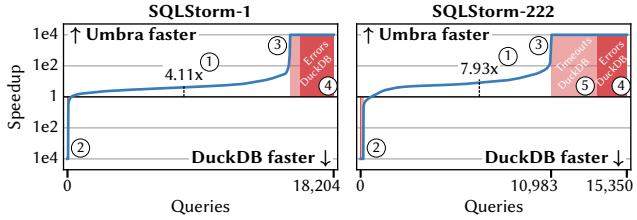

Figure 8: Umbra vs. DuckDB. We report per-query speedups of Umbra over DuckDB in ascending order. Umbra succedds on more queries and executes the majority of queries faster.

DuckDB started with 361 crashes in January 2023, and the developers constantly reduced this number as they prepared for the major 1.0.0 release in June 2024. At that point in time, some occasional crashes still occurred, and the last ones were addressed in the latest release, 1.2.0. DuckDB is supported by a large community that played a crucial role in identifying the bugs, even without SQLStorm. However, queries that return with error in DuckDB remain almost unchanged over time. Syntax errors dominate this category: there is no immediate need for DuckDB to implement missing PostgreSQL construct as users can adapt their queries. The number of timeouts in DuckDB fluctuates between 550 and 700 queries. SQLStorm aids in pinpointing regressions and addressing them before releasing a new version.

While Umbra does not see a significant change in runtime, the geometric mean of the end-to-end time improve by almost 2× in DuckDB of the last two years. For Umbra, we observe a performance regression of 11.4 % around January 2024 that was resolved in August of the same year. This regression was not evident in the TPC-H 10 GB benchmark, which explains why we did not notice it earlier. In DuckDB, the latest release improved the performance on SQLStorm considerably. Conversely, the same release caused a performance regression of 22 % on TPC-H. This highlights the limitations of TPC-H as a sole benchmark, as it may overlook performance regressions or improvements that become evident in more diverse, real-world workloads like SQLStorm.

4.4 Scaling with the Dataset Size

In contrast to other real-world datasets, such as IMDB [30], the StackOverflow dataset and therefore SQLStorm is available in different sizes: 1 GB (SQLStorm-1), 12 GB (SQLStorm-12), and 220 GB (SQLStorm-220). This substantially broadens the applicability of SQLStorm, e.g., making it possible to run the benchmark on both small machines and larger, e.g., distributed, deployments. Increasing the dataset size and comparing the performance for the same query furthermore provides insights into a query engine's scalability.

Figure 7 shows the runtime distribution of the queries on the three datasets for Umbra and DuckDB. The two systems behave broadly similarly. On the small dataset, most queries complete in less than 100 ms, and timeouts are rare. As the scale factor increases, the queries take longer to finish, and timeouts become more frequent. Nevertheless, Umbra and DuckDB scale well, and the median runtime only increases by 8× and 10×, respectively, when moving from SQLStorm-12 to SQLStorm-220. On SQLStorm-220, DuckDB times out on 4,938 queries, while Umbra exceeds the time limit of 10 s for 1,807 queries and the memory limit for 1,166 queries.

We also included the runtime distribution of TPC-DS, which consists of 100 query templates and is considered one of the most extensive and challenging analytical benchmarks. TPC-DS lacks outliers, and the runtimes are spread closer to the median. Like real-world workloads [56, 60], SQLStorm covers a broader spectrum and experiences more timeouts. Furthermore, the median runtime of TPC-DS increases by only 3× from TPC-DS-10 to TPC-DS-100 even though the data grows by a factor of 10. In SQLStorm, 20× more data leads to 10× longer runtimes in DuckDB. Therefore, scaling up on SQLStorm is more challenging than on TPC-DS. We believe that solving these scalability challenges involves not just better engineering but also innovations in query optimization and query processing. For example, one of the queries that succeeds on the small dataset but fails on the larger datasets performs a join with substring search as the join predicate of the form `R.a LIKE '%' || S.b || '%'`. In existing systems this semantically meaningful query results in quadratic runtime even though the query could be efficiently processed using a suffix array.

4.5 Comparing Systems

Existing database systems can only execute a subset of SQLStorm. Because this subset differs between systems, statistics such as average or geometric mean can be somewhat misleading, e.g., a system that fails on difficult queries might have good average performance. Users would probably care at least as much about queries that fail or take a very long time.

To illustrate this behavior, we compare the runtimes and failures for Umbra and DuckDB, shown in Figure 8. We consider only queries that run in one of the evaluated systems and calculate the speedup for every query. If a query timeouts or fails in one system but succeeds in the other, we set the speedup to 1e4 in the figure. The speedups are arranged in ascending order from left to right: first, the queries that fail or timeout in Umbra but succeed in DuckDB; next, the queries that run in both systems; and finally, the queries that fail or timeout in DuckDB but succeed in Umbra. This visualization provides a comprehensive but succinct overview between two systems, even for thousands of queries.

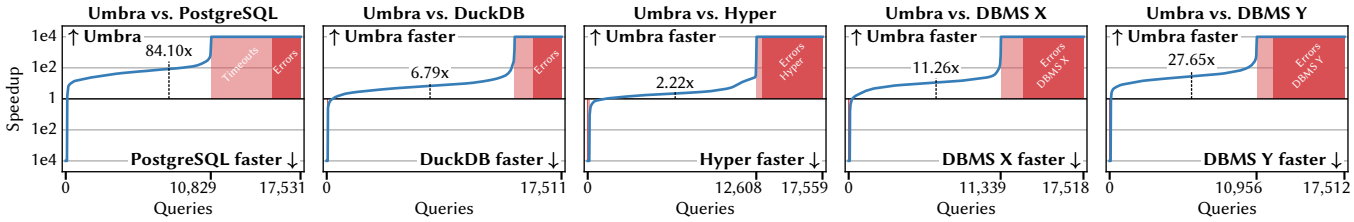

Figure 9: System comparison on SQLStorm-10. SQLStorm uncovers for every system promising queries for further optimizations.

Table 10: The query generation process, the number of executable queries per system, and the number of operators on the StackOverflow, the TPC-H and the JOB dataset.

Dataset	Query Selection			Systems			avg.
	Parse 1	Parse 2	Exec.				Ops.
StackOverflow	15206	20218	18251	12161	18133	13712	13.9
TPC-H	16964	21074	17036	6964	16777	15301	16.0
TPC-DS	13361	16037	15242	12309	15053	14461	12.9
JOB	14749	17425	11714	2720	10948	9495	16.8

The plot reveals some interesting findings: ① is the medium speedup of both systems' executable queries (success or timeout). Umbra is 4.11× faster than DuckDB on SQLStorm-1 and 7.93× on SQLStorm-220. While at ②, DuckDB performs significantly better than Umbra, at ③, Umbra outperforms DuckDB by up to 1,000×. These queries are particularly interesting for database developers as the other systems perform multiple orders of magnitude better. Such large performance differences are usually caused by differences in query optimizations such as missing optimization rules or large cost model errors.

The red area at ④ shows the queries that fail in DuckDB but succeed in Umbra. In ⑤, DuckDB encounter timeouts while Umbra finishes the queries in under 10 seconds. Even on the small SQLStorm-1 dataset, the number of unsuccessful queries in DuckDB is considerable. Umbra also has queries that time out or fail but succeed in DuckDB. However, the number of such instances is small and almost invisible in the figures.

In Figure 9, we repeat the analysis between Umbra and all other systems on SQLStorm-12. Interestingly, each system outperforms Umbra on certain queries. For instance, PostgreSQL, which is typically 84× times slower than Umbra due to its row-based storage and volcano-style query execution, still surpasses Umbra in performance on 108 out of 17,531 queries. These queries will help pinpoint weaknesses in Umbra and guide further performance improvements. Database developers can use SQLStorm to explore optimization opportunities in their systems and uncover missing features. While the median runtime indicates the performance benefits of different architectures, such as columnar versus row-based storage, the long tails reveal isolated performance bugs.

4.6 Generating New Benchmarks

We selected the StackOverflow dataset because it contains real-world data and features more and longer strings than synthetic datasets. Yet, the SQLStorm methodology can be applied to other (real-world or synthetic) datasets as well. We tested this using the well-known analytical benchmarks: TPC-H/-DS, and JOB [30]. The methodology remains the same: GPT-4o-mini generates 35,000 queries using the seven prompts from Table 2 for each dataset; only the schema is adjusted to match the dataset.

Table 10 shows the number of queries for all three datasets, how many are executable in PostgreSQL, Umbra, and DuckDB, and the average number of operators per query. Although the specific numbers differ, the broad picture is similar to the original SQLStorm. After pre-processing and filtering, more than 10,000 queries remain, enough to stress-test any database system and provide a comprehensive evaluation. On TPC-H and JOB queries have two operators more as they perform more joins than the queries from the other two datasets. Aside from this difference, GPT-4o-mini generates a similar distribution of operators across all datasets.

SQLStorm is not limited to the StackOverflow data and can be applied other datasets as well. The technique is a simple and cost-efficient way to generate large-scale benchmarks for database systems. We make all the queries publicly available, and database developers can use the SQLStorm prompts to generate new benchmarks on custom datasets.

5 SUMMARY AND FUTURE WORK

SQLStorm shifts database benchmarking from handcrafted queries to large-scale LLM-generated workloads. We demonstrate that large language models generate diverse, complex, and unpredictable queries that encompasses a broad range of SQL constructs. The initial SQLStorm release includes 18,251 queries on a real-world dataset. These challenging queries reveal weaknesses and optimization potential in database system, fostering new research directions in query optimization, cardinality estimation and robust data processing. SQLStorm complements existing benchmarks like TPC-H and TPC-DS, which are limited by a few query templates and small feature spaces. While these benchmarks contain high-quality expert-written queries, our generated queries occasionally include rare SQL constructs, semantic mistakes, and edge cases capable of stressing database systems.

SQLStorm v1.0 leverages OpenAI's GPT-4o-mini model. Future releases of this benchmark will incorporate next-generation LLMs, further enhancing complexity and diversity. Database engineers can refine the prompts and datasets to tailor the benchmark to their needs and generate queries to new SQL features such as property graph queries or ASOF joins. This paper takes the first step towards automated large-scale database benchmark generation in the LLM era; SQLStorm offers a scalable real-world dataset and a replicable, cost-effective query generation pipeline.

REFERENCES

[1] 2022. *Database schema documentation for the public data dump and SEDE.* Retrieved February 1, 2025 from https://meta.stackexchange.com/questions/2677/database-schema-documentation-for-the-public-data-dump-and-sede

[2] 2024. *2024 Developer Survey.* Retrieved February 1, 2025 from https://survey.stackoverflow.co/2024/

[3] 2025. *How do I access a data dump for a Stack Exchange site?* Retrieved February 1, 2025 from https://stackoverflow.com/help/data-dumps

[4] 2025. *Stack Exchange.* Retrieved February 1, 2025 from https://stackexchange.com/about

[5] Peter Akioyamen, Zixuan Yi, and Ryan Marcus. 2024. The Unreasonable Effectiveness of LLMs for Query Optimization. *CoRR* abs/2411.02862 (2024).

[6] Amazon. 2023. *Amazon Q generative SQL.* Retrieved May 1, 2025 from https://aws.amazon.com/about-aws/whats-new/2023/11/amazon-redshift-generative-sql-query-editor-preview/

[7] Lawrence Benson, Carsten Binnig, Jan-Micha Bodensohn, Federico Lorenzi, Jigao Luo, Danica Porobic, Tilmann Rabl, Anupam Sanghi, Russell Sears, Pinar Tözün, and Tobias Ziegler. 2024. Surprise Benchmarking: The Why, What, and How. In *DBTest@SIGMOD*. ACM, 1–8.

[8] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ B. Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri S. Chatterji, Annie S. Chen, Kathleen Creel, Jared Quincy Davis, Dorottya Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren E. Gillespie, Karan Goel, Noah D. Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark S. Krass, Ranjay Krishna, Rohith Kuditipudi, and et al. 2021. On the Opportunities and Risks of Foundation Models. *CoRR* abs/2108.07258 (2021).

[9] Peter A. Boncz, Angelos-Christos G. Anadiotis, and Steffen Kläbe. 2017. JCC-H: Adding Join Crossing Correlations with Skew to TPC-H. In *TPCTC (Lecture Notes in Computer Science, Vol. 10661)*. Springer, 103–119.

[10] Peter A. Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *TPCTC (Lecture Notes in Computer Science, Vol. 8391)*. Springer, 61–76.

[11] Surajit Chaudhuri, Vivek R. Narasayya, and Ravishankar Ramamurthy. 2009. Exact Cardinality Query Optimization for Optimizer Testing. *Proc. VLDB Endow.* 2, 1 (2009), 994–1005.

[12] Sibei Chen, Ju Fan, Bin Wu, Nan Tang, Chao Deng, Pengyi Wang, Ye Li, Jian Tan, Feifei Li, Jingren Zhou, and Xiaoyong Du. 2024. Automatic Database Configuration Debugging using Retrieval-Augmented Language Models. *CoRR* abs/2412.07548 (2024).

[13] Shaleen Deep, Anja Gruenheid, Kruthi Nagaraj, Hiro Naito, Jeffrey F. Naughton, and Stratis Viglas. 2020. DIAMetrics: Benchmarking Query Engines at Scale. *Proc. VLDB Endow.* 13, 12 (2020), 3285–3298.

[14] Jonathan Dees and Peter Sanders. 2013. Efficient many-core query execution in main memory column-stores. In *ICDE*. IEEE Computer Society, 350–361.

[15] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek R. Narasayya. 2021. DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems. *Proc. VLDB Endow.* 14, 13 (2021), 3376–3388.

[16] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *Proc. VLDB Endow.* 13, 8 (2020), 1206–1220.

[17] Florian Haftmann, Donald Kossmann, and Eric Lo. 2007. A framework for efficient regression tests on database applications. *VLDB J.* 16, 1 (2007), 145–164.

[18] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *Proc. VLDB Endow.* 15, 4 (2021), 752–765.

[19] Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2024. Next-Generation Database Interfaces: A Survey of LLM-based Text-to-SQL. *CoRR* abs/2406.08426 (2024).

[20] Karl Huppler. 2009. The Art of Building a Good Benchmark. In *TPCTC (Lecture Notes in Computer Science, Vol. 5895)*. Springer, 18–30.

[21] Stack Exchange Inc. 2024. *Database Administrators StackExchange.* Retrieved October 1, 2025 from https://dba.stackexchange.com/

[22] Stack Exchange Inc. 2024. *Mathematics StackExchange.* Retrieved October 1, 2025 from https://math.stackexchange.com/

[23] Stack Exchange Inc. 2024. *StackOverflow.* Retrieved October 1, 2025 from https://stackoverflow.com/

[24] Mahmoud Abo Khamis, Vasileios Nakos, Dan Olteanu, and Dan Suciu. 2024. Join Size Bounds using l_p-Norms on Degree Sequences. *Proc. ACM Manag. Data* 2, 2, Article 96 (2024).

[25] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR*. www.cidrdb.org.

[26] Skander Krid, Mihail Stoian, and Andreas Kipf. 2025. Redbench: A Benchmark Reflecting Real Workloads. In *aiDM@SIGMOD*. ACM.

[27] 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 (2024), 1939–1952.

[28] Kukjin Lee, Anshuman Dutt, Vivek R. Narasayya, and Surajit Chaudhuri. 2023. Analyzing the Impact of Cardinality Estimation on Execution Plans in Microsoft SQL Server. *Proc. VLDB Endow.* 16, 11 (2023), 2871–2883.

[29] Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida Wang, and Tao Yu. 2025. Spider 2.0: Evaluating Language Models on Real-World Enterprise Text-to-SQL Workflows. In *ICLR*. OpenReview.net.

[30] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.

[31] Guoliang Li, Xuanhe Zhou, and Xinyang Zhao. 2024. LLM for Data Management. *Proc. VLDB Endow.* 17, 12 (2024), 4213–4216.

[32] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chen-Chuan Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs. In *NeurIPS*.

[33] Yiyan Li, Haoyang Li, Pu Zhao, 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. *CoRR* abs/2408.02213 (2024).

[34] 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 (2024), 53–65.

[35] Jie Liu and Barzan Mozafari. 2024. Query Rewriting via Large Language Models. *CoRR* abs/2403.09060 (2024).

[36] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *SIGMOD Conference*. ACM, 1275–1288.

[37] Nestor Maslej, Loredana Fattorini, C. Raymond Perrault, Vanessa Parli, Anka Reuel, Erik Brynjolfsson, John Etchemendy, Katrina Ligett, Terah Lyons, James Manyika, Juan Carlos Niebles, Yoav Shoham, Russell Wald, and Jack Clark. 2024. Artificial Intelligence Index Report 2024. *CoRR* abs/2405.19522 (2024).

[38] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *Proc. VLDB Endow.* 2, 1 (2009), 982–993.

[39] Ingo Müller, Cornelius Ratsch, and Franz Färber. 2014. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *EDBT*. OpenProceedings.org, 283–294.

[40] Avanika Narayan, Ines Chami, Laurel J. Orr, and Christopher Ré. 2022. Can Foundation Models Wrangle Your Data? *Proc. VLDB Endow.* 16, 4 (2022), 738–746.

[41] Parimarjan Negi, Laurent Bindschaedler, Mohammad Alizadeh, Tim Kraska, Jyoti Leeka, Anja Gruenheid, and Matteo Interlandi. 2023. Unshackling Database Benchmarking from Synthetic Workloads. In *ICDE*. IEEE, 3659–3662.

[42] Thomas Neumann and César A. Galindo-Legaria. 2013. Taking the Edge off Cardinality Estimation Errors using Incremental Execution. In *BTW (LNI, Vol. P-214)*. GI, 73–92.

[43] Thomas Neumann and Alfons Kemper. 2015. Unnesting Arbitrary Queries. In *BTW (LNI, Vol. P-241)*. GI, 383–402.

[44] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *OSDI*. USENIX Association, 667–682.

[45] Andreas Seltenreich, Bo Tang, and Sjoerd Mullender. 2024. *SQLsmith: A random SQL query generator.* Retrieved February 1, 2025 from https://github.com/anse1/sqlsmith

[46] Snowflake. 2025. *Snowflake Copilot.* Retrieved May 1, 2025 from https://docs.snowflake.com/en/user-guide/snowflake-copilot

[47] Mihail Stoian, Andreas Zimmerer, Skander Krid, Amadou Latyr Ngom, Jialin Ding, Tim Kraska, and Andreas Kipf. 2025. Parachute: Single-Pass Bi-Directional Information Passing. *Proc. VLDB Endow.* 18, 10 (2025).

[48] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2024. R-Bot: An LLM-based Query Rewrite System. *CoRR* abs/2412.01661 (2024).

[49] Jan Vincent Szlang, Sebastian Bress, Sebastian Cattes, Jonathan Dees, Florian Funke, Max Heimel, Michel Oleynik, Ismail Oukid, and Tobias Maltenberger. 2025. Workload Insights From The Snowflake Data Cloud: What Do Production Analytic Queries Really Look Like? *Proc. VLDB Endow.* 18, 11 (2025).

[50] Jie Tan, Kangfei Zhao, Rui Li, Jeffrey Xu Yu, Chengzhi Piao, Hong Cheng, Helen Meng, Deli Zhao, and Yu Rong. 2025. Can Large Language Models Be Query

Optimizer for Relational Databases? *CoRR* abs/2502.05562 (2025).

[51] BIRD Team and Google Cloud. 2025. *BIRD-CRITIC: Can LLMs Fix User Issues in Real-World Database Applications?* Retrieved May 1, 2025 from https://bird-critic.github.io/

[52] DuckDB Team. 2025. *DuckDB Documentation: PostgreSQL Compatibility.* Retrieved May 1, 2025 from https://duckdb.org/docs/stable/sql/dialect/postgresql_compatibility.html

[53] Pinar Tözün, Ippokratis Pandis, Cansu Kaynak, Djordje Jevdjic, and Anastasia Ailamaki. 2013. From A to E: analyzing TPC's OLTP benchmarks: the obsolete, the ubiquitous, the unexplored. In *EDBT*. ACM, 17–28.

[54] Immanuel Trummer. 2022. CodexDB: Synthesizing Code for Query Processing from Natural Language Instructions using GPT-3 Codex. *Proc. VLDB Endow.* 15, 11 (2022), 2921–2928.

[55] Immanuel Trummer. 2022. DB-BERT: A Database Tuning Tool that "Reads the Manual". In *SIGMOD Conference*. ACM, 190–203.

[56] 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 (2024), 3694–3706.

[57] Alexander van Renen and Viktor Leis. 2023. Cloud Analytics Benchmark. *Proc. VLDB Endow.* 16, 6 (2023), 1413–1425.

[58] Alexander van Renen, Mihail Stoian, and Andreas Kipf. 2024. DataLoom: Simplifying Data Loading with LLMs. *Proc. VLDB Endow.* 17, 12 (2024), 4449–4452.

[59] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *DBTest@SIGMOD*. ACM, 1:1–1:6.

[60] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *NSDI*. USENIX Association, 449–462.

[61] Florian M. Waas, Leo Giakoumakis, and Shin Zhang. 2011. Plan space analysis: an early warning system to detect plan regressions in cost-based optimizers. In *DBTest*. ACM, 2.

[62] Qichen Wang, Bingnan Chen, Binyang Dai, Ke Yi, Feifei Li, and Liang Lin. 2025. Yannakakis+: Practical Acyclic Query Evaluation with Theoretical Guarantees. *Proc. ACM Manag. Data* 3, 3, Article 235 (2025).

[63] Johannes Wehrstein, Timo Eckmann, Roman Heinrich, and Carsten Binnig. 2025. JOB-Complex: A Challenging Benchmark for Traditional & Learned Query Optimization. In *AIDB@VLDB*.

[64] Zhiming Yao, Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2025. A Query Optimization Method Utilizing Large Language Models. *CoRR* abs/2503.06902 (2025).

[65] Zixuan Yi, Yao Tian, Zachary G. Ives, and Ryan Marcus. 2025. Low Rank Learning for Offline Query Optimization. *Proc. ACM Manag. Data* 3, 3, Article 183 (2025).

[66] Junyi Zhao, Kai Su, Yifei Yang, Xiangyao Yu, Paraschos Koutris, and Huanchen Zhang. 2025. Debunking the Myth of Join Ordering: Toward Robust SQL Analytics. *Proc. ACM Manag. Data* 3, 3, Article 146 (2025).

[67] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A Survey of Large Language Models. *CoRR* abs/2303.18223 (2023).

[68] Xinyang Zhao, Xuanhe Zhou, and Guoliang Li. 2024. Chat2Data: An Interactive Data Analysis System with RAG, Vector Databases and LLMs. *Proc. VLDB Endow.* 17, 12 (2024), 4481–4484.

[69] Xuanhe Zhou, Guoliang Li, Zhaoyan Sun, Zhiyuan Liu, Weize Chen, Jianming Wu, Jiesi Liu, Ruohang Feng, and Guoyang Zeng. 2024. D-Bot: Database Diagnosis System using Large Language Models. *Proc. VLDB Endow.* 17, 10 (2024), 2514–2527.

[70] Xuanhe Zhou, Zhaoyan Sun, and Guoliang Li. 2024. DB-GPT: Large Language Model Meets Database. *Data Sci. Eng.* 9, 1 (2024), 102–111.