

Automated Database Testing via LLM-Synthesized SQL Features

SUYANG ZHONG, National University of Singapore, Singapore

MANUEL RIGGER, National University of Singapore, Singapore

Database Management Systems (DBMSs) have been tested by various automated testing approaches. Many of them generate pairs of equivalent queries to identify bugs that cause DBMSs to compute incorrect results, and have found hundreds of bugs in mature, widely used DBMSs. Most such approaches are based on manually written SQL generators; however, their bug-finding capabilities remain constrained by the limited set of SQL features supported by the generators. In this work, we propose ShQveL, an approach that augments existing SQL test-case generators by leveraging Large Language Models (LLMs) to synthesize SQL fragments. Our key idea is to systematically incorporate SQL features gained through automated interactions with LLMs into the SQL generators, increasing the features covered while efficiently generating test cases. Specifically, ShQveL uses SQL sketches—SQL statements with incomplete code segments that LLMs fill—to integrate LLM-generated content into the generator. We evaluated ShQveL on 5 DBMSs and discovered 55 unique and previously unknown bugs, 50 of which were promptly fixed after our reports.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; • **Information systems** → **Database management system engines**; • **Computing methodologies** → *Natural language generation*; *Artificial intelligence*.

Additional Key Words and Phrases: Database systems, automated testing, test case generation, SQL testing, large language models

ACM Reference Format:

Suyang Zhong and Manuel Rigger. 2026. Automated Database Testing via LLM-Synthesized SQL Features. *Proc. ACM Manag. Data* 4, 4, Article 259 (September 2026), 27 pages. <https://doi.org/10.1145/3837097>

1 Introduction

Database Management Systems (DBMSs) are large, complex, and fundamental software systems. Unsurprisingly, they are prone to bugs. Various approaches have been proposed to detect *logic bugs* in them using automated testing [14, 31–33, 35, 40]. They primarily tackle the so-called *test-oracle problem* by validating whether a DBMS operates as expected, for example, by deriving an equivalent query from a given input query to check whether the DBMS produces consistent results [3, 31, 32, 50]. Such test oracles have successfully identified hundreds of bugs in popular traditional DBMSs like SQLite and MySQL, as well as emerging DBMSs like DuckDB and TiDB. Most automated DBMS testing tools automatically derive SQL test cases. They can be broadly classified into generation-based ones, which generate test cases from scratch, based on rule-based generators, while mutation-based ones mutate existing statements.

More than a thousand DBMSs exist,¹ and automatically applying existing testing tools to all of them is challenging. The key challenge is that DBMSs' dialects differ in both syntax and semantics,

¹<https://dbdb.io/>

Authors' Contact Information: Suyang Zhong, suyang@u.nus.edu, National University of Singapore, Singapore, Singapore; Manuel Rigger, rigger@nus.edu.sg, National University of Singapore, Singapore, Singapore.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2836-6573/2026/9-ART259

<https://doi.org/10.1145/3837097>

Listing 1. An example bug in CrateDB caused by scalar function `ARRAY_POSITION`. The query is expected to return an empty result; however, one redundant row is returned.

```

1 CREATE TABLE t0(c0 INT, c1 ARRAY(STRING));
2 INSERT INTO t0(c0, c1) VALUES (1, ['a', 'b']);
3
4 SELECT c0 FROM t0
5 WHERE (c0 != ARRAY_POSITION(t0.c1, 'c', 1));
6 -- {1} ✖ -- {} ✔

```

including operations and data types, which existing automated testing approaches utilize in their tests. To test dialect-specific features, both mutators and generators would ideally generate or mutate instances of these features. Mutation-based test case generation [17, 24] depends on high-quality seed inputs, which can be extracted from test suites. However, tests are often inapplicable to DBMSs other than those for which they were written due to significant dialect differences [53]. Besides, mutation-based testing tools are difficult to employ to find logic bugs since many test oracles impose strict input constraints. Conversely, generation-based methods [31–33] require significant human effort to create DBMS-specific generators.

SQLancer++ represents an initial attempt to address these challenges [54]. It consists of an SQL generator that infers which features are supported by a given DBMS. It can be used on different DBMSs regardless of SQL dialects without modifications to its source code. For a specific feature (e.g., a function or an operator), *SQLancer++* executes a number of test cases and then infers the probability of it being supported based on the execution status of the statements containing this feature. However, *SQLancer++*'s ability to find bugs is constrained by the initially supported set of manually implemented SQL features, potentially missing unique or newly released features of a DBMS. For example, it supports only 3 data types and 58 functions, and it would be difficult to maintain a generator that supports most features of most DBMSs. Listing 1 demonstrates one bug in CrateDB that *SQLancer++* failed to find since it lacks support for the nested data type `ARRAY(STRING)` and the incorrectly implemented function `ARRAY_POSITION`. Supporting new DBMS-specific features is a labor-intensive process that requires expertise in both the DBMS and the test case generator. Thus, an automatic method to integrate these features into the generator is essential.

Recent progress on *Large Language Models (LLMs)* suggests the possibility of using them as DBMS test-case generators. First, prior research has applied LLMs to data management, such as in the context of text-to-SQL [11, 21] and query rewriting [23, 26]. LLMs have also been applied to testing in a variety of domains, including general-purpose testing [43], compiler testing [44], and testing of deep learning libraries [6]. However, several challenges persist when applying LLMs to test DBMSs. First, the throughput of LLM-based generation is significantly lower than that of traditional query generators. For instance, LLM-R² [23] leverages LLMs to rewrite queries, resulting in an average latency exceeding one second per query on each of its benchmarks, while tools like *SQLancer* can generate thousands of queries per second. Second, the cost of employing LLMs for testing is high. Utilizing an LLM typically requires access to a powerful GPU server or incurs significant expenses through API services (e.g., GPT-4o costs 15 US dollars for every 1 million tokens). As a result, integrating LLMs into the CI/CD pipelines for DBMS development is neither straightforward nor cost-effective. Third, current LLMs suffer from hallucination [15, 51], producing unreliable outputs. In summary, the inherent inefficiency, expense, and unreliability of current LLMs pose significant challenges to their application in the testing domain.

Table 1. Comparison of an LLM-based generator and a manually written generator on DuckDB for 6 hours.

Method	Device	Unique Bugs Found	Branch Cov.	Generated Stmts.	Validity Rate
Fuzz4All	GPU	0	24.5%	183K	70.9%
SQLancer++	CPU	2	21.3%	41,976K	93.1%

In this paper, we propose *ShQveL*, a technique that enables existing SQL generators to incorporate features via LLMs. Our core insight to achieve *efficient* SQL generators that can generate DBMS-specific SQL features is to leverage LLMs by permanently integrating the LLM-generated content after automatically validating it. After an initial learning phase, we can disable LLM interactions and thus achieve an efficiency comparable to manually written generators. A key challenge is to design the approach such that LLMs can be used to determine DBMS-specific features, and subsequently validate and integrate this knowledge into the SQL generators. To bridge the gap between the LLM-generated content and generator source code, we propose the concept of *SQL sketching*, loosely inspired by the concept of sketch-based program synthesis [36, 42]. An SQL sketch is a template of SQL statements generated by the original SQL generator with incomplete segments as placeholders, or *holes*. During learning, an LLM fills the holes in the sketch based on its pre-trained knowledge and DBMS documentation through in-context learning. Subsequently, the LLM-generated fragments are integrated into the generator. In order to impose constraints on the LLM, we prompt it by providing a SQL sketch that implicitly imposes constraints on feature usage (e.g., a SQL sketch `SELECT 1 ?? 1` indicates that the operator should have two integer operands). After receiving the LLM's response, our approach validates the resulting filled sketch and subsequently uses the validated fragments during testing.

We implemented *ShQveL* based on SQLancer++ and evaluated it on CrateDB, CockroachDB, DuckDB, MonetDB, and TiDB. We found and reported 55 unique, previously unknown bugs, showing the effectiveness of our approach. 50 bugs have been fixed, demonstrating that the developers considered the bugs important. Our goal was not to outperform the manually written generators (e.g., *SQLancer*), but to find bugs in DBMSs of different dialects without investing any implementation effort. Despite this, *ShQveL* achieved comparable code coverage on PostgreSQL and SQLite, and, on DuckDB, observed increases of 45% compared with SQLancer++ and 30% compared with SQLancer. *ShQveL* can learn features efficiently and economically, as it learned around 400 fragments for each DBMS in six hours with a cost of less than 1 USD per DBMS.

We believe that our approach is both practical and scalable. The learned fragments can persist across executions and can be reused even across different systems. Once a sufficient number of features has been learned, the learning process can be stopped, leading to more efficient test-case generation, especially in resource-limited environments (e.g., CI runs). Additionally, our *SQLancer++*-based feedback mechanism validates LLM-generated fragments without human supervision. In general, this approach makes the testing process more efficient, economical, and manageable.

In summary, we make the following contributions:

- We propose *SQL sketching* as a general notion for integrating LLM-generated content with generation-based DBMS testing.
- We propose a testing approach, *ShQveL*, which leverages LLMs to persist and integrate DBMS-specific features into the generator.
- We implemented and evaluated the approach, which has found 55 unique, previously unknown bugs in widely used DBMSs.

2 Background and Motivation Study

To understand the rationale behind our approach, we first introduce the preliminaries for generating SQL test cases in a dialect-agnostic manner (see Section 2.1). Subsequently, we demonstrate the challenges of using LLMs in DBMS testing and the limitations of two naive approaches: (1) generating test cases directly with the LLM (see Section 2.2) and (2) generating test generators with the LLM (see Section 2.3). Given the LLMs' proficiency in SQL tasks like text-to-SQL, a straightforward approach is to have them generate SQL statements for fuzzing, as described in a recent blog post.² For example, we can prompt the LLM to generate SQL statements based on the corresponding documentation and execute the generated statements on the target systems. LLMs have also achieved high performance on code generation [13, 18], suggesting that they can be used to synthesize generators, a possibility that we thus explored. The throughput of executing synthesized generators that themselves eschew interactions with LLMs is expected to be higher, but any error in an LLM's output would render the whole generator unusable. We chose DuckDB v0.7.1, a historical version of the widely used embedded relational DBMS, as our target [9, 32].

2.1 Dialect-agnostic SQL Generation

SQLancer++ is an automated platform for testing DBMSs to find logic bugs, aiming to test a wide range of DBMSs with different SQL dialects. Its core component is an adaptive SQL generator that, during execution, dynamically infers which of a set of predefined SQL features—elements or properties in the query language that can be specified at different levels of granularity—are supported by a given DBMS. Specifically, a feature might be a keyword, such as `DISTINCT`, or an entire SQL statement, like `VACUUM`. It can also refer to operators or functions, for example, the null-safe operator `<=>` in MySQL. To infer whether this feature is supported by the system under test, during execution, *SQLancer++* generates a sufficient number of SQL statements containing the feature. Subsequently, *SQLancer++* calculates the estimated probability of the features being supported based on whether the DBMS executed a predefined percentage of instances of the features successfully in previous executions. *SQLancer++* supports only a limited set of SQL features across DBMSs, as it relies on a manually selected subset of basic, widely shared features across dialects. It is feasible to add more features for one specific DBMS; however, it requires a high implementation effort, and it would be infeasible to add and maintain dialect-specific features of many DBMSs, thus motivating the subsequent exploration.

2.2 LLM-based Test Generation

Methodology. We first demonstrate the potential and challenges of using LLMs to directly generate test cases by applying Fuzz4All [43] to test DuckDB. Fuzz4All is a universal state-of-the-art fuzzer that leverages LLMs (e.g., StarCoder [22]) to target a variety of input languages, including C/C++ and Java, although it was not specifically evaluated on SQL. We executed both Fuzz4All and *SQLancer++* continuously for 6 hours on a GPU server, comparing their test efficiency and test case validity rates. For Fuzz4All, we configured the system with a summary of the DuckDB documentation while keeping all other settings at their defaults (e.g., GPT-4 as distillation LLM and StarCoder as generation LLM). We executed both systems in a single-threaded manner for comparability.

Results. Table 1 summarizes the results of our experiments. First, *SQLancer++* generated 229× more SQL statements, as invoking the LLM for every SQL statement is inefficient. Second, we observed a higher validity rate for *SQLancer++*, although it also fails to achieve a 100% validity rate, because certain features can trigger expected errors due to semantic constraints (e.g., adding

²<https://celerdatablog.com/blog/chatgpt-is-now-finding-bugs-in-databases>

Listing 2. A random DuckDB CREATE TABLE generator synthesized by ChatGPT o1. We shortened variable names and omitted helper functions for simplicity.

```

1  def rand_constr(col_name, existing_cols):
2      constrs = []
3      col_constrs = [
4          f"UNIQUE",
5          # Issue 1: Value is fixed
6          f"CHECK ({col_name} > 0)",
7          f"CHECK ({col_name} <> ' ')",
8      ]
9      use_col_constraints = random.choice([True, False])
10     if use_col_constraints:
11         constrs.append(random.choice(col_constrs))
12     if existing_cols and random.choice([True, False]):
13         referenced_col = random.choice(existing_cols)
14         # Issue 2: SQL syntax is incorrect
15         # Issue 3: Table name is hardcoded
16         constrs.append(f"FOREIGN KEY ({col_name}) REFERENCES
17             some_other_table({referenced_col})")
18     return " ".join(constrs)
19
20 def generate_create_table_statement():
21     # Generate columns_sql with rand_constr
22     # ...
23     # Construct the SQL statement and return it
24     return f"CREATE TABLE {table_name} (\n    {columns_sql}\n);"
```

two very large integers may overflow and result in an error). Third, SQLancer++ reported 156 bug-inducing test cases with 2 unique bugs compared to none discovered by the LLM-based approach. However, the LLM-based method achieved higher branch coverage; one plausible explanation is that it can generate DuckDB-specific SQL features, while SQLancer++ was designed and implemented to support mostly common ones. While exercising code is a necessary condition for finding a bug, it is not sufficient; in practice, only specific test cases might trigger it, which is why high throughput is also necessary [4].

LLM-generated SQL test cases incorporating DBMS-specific features can achieve higher coverage, while manually written generators with higher throughput can detect more bugs.

2.3 LLM-based Generator Generation

Methodology. We employed LLMs to synthesize a statement generator for DuckDB to explore the potential of efficient test case generation. We used ChatGPT o1—one of the most advanced models for code at the time of writing—to synthesize a program capable of generating random CREATE TABLE statements with the following prompt: “Act as an advanced Python and database developer. Please generate a Python script that, upon execution, randomly creates a DuckDB CREATE TABLE statement.” Additionally, we provided relevant documentation to guide the model in generating DBMS-specific content.

Results. Listing 2 shows the code that ChatGPT synthesized. While ChatGPT’s output is almost correct, we observed that hallucinations in the LLM’s output can cause bugs. For example, lines

13–17 of Listing 2 incorrectly apply a FOREIGN KEY constraint as a column constraint in DuckDB, which may result in a syntax error when executing the generated SQL statement. Although we supplied DuckDB’s documentation, ChatGPT still produced incorrect code—likely due to difficulties in parsing highly structured languages [41]. Such errors could potentially render the whole testing tool unusable unless they are manually fixed.

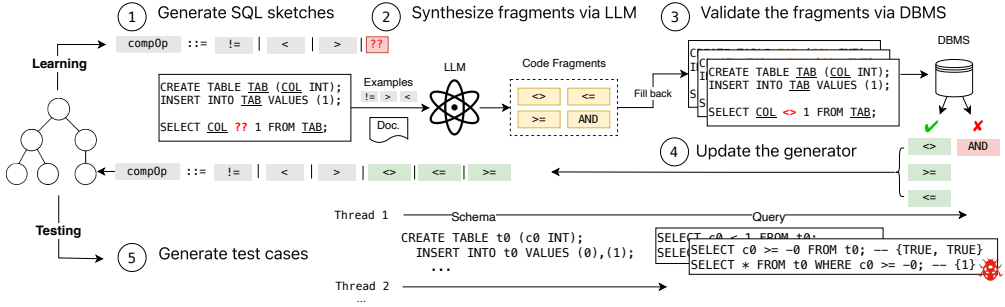
An additional challenge is how to assemble multiple such generators to form a complete automated testing tool, as well as how to address concerns such as schema management, value diversity, and statement dependencies. First, LLMs struggle to maintain consistent schema references. For example, LLMs might not reference tables created by the CREATE TABLE statements. In line 16 of Listing 2, the generator references a non-existent table, `some_other_table`. Second, generators synthesized using LLMs often explore a limited part of the search space; line 6 of Listing 2 compares a column’s value to 0, while it may overlook boundary values (e.g., the maximum integer value) that may trigger bugs. Third, statements must be executed in the correct order; for example, CREATE TABLE should precede INSERT. While advanced approaches in LLM agents for general coding tasks [46, 52] might address some of these issues, additional manual effort will likely still be required, and subtle errors could affect the effectiveness and efficiency.

LLMs struggle to synthesize SQL generators correctly, as they face internal hallucination issues and other challenges—schema handling, generating diverse and valid constants, and statement scheduling—which together lead to suboptimal accuracy and reliability in the generated queries.

3 ShQveL

We propose *ShQveL*, an approach to automatically test DBMSs, supporting a multitude of SQL dialects through LLM guidance. The core idea behind our approach is to use LLMs to extract knowledge of different dialects’ SQL features and corresponding DBMS documentation to then integrate this knowledge into SQL generators automatically. Consequently, unlike existing solutions like *SQLancer* or *SQLancer++*, *ShQveL* can potentially cover each DBMS’s SQL features even without requiring manual implementation effort. To avoid the high cost and low throughput of directly instructing the LLM to generate SQL tests (see Section 2.2), we persist knowledge gained through LLM interactions; after a user has decided that a sufficient number of features has been learned, no further LLM invocations are required. Due to hallucinations and other limitations (e.g., limited context length) of current LLMs, directly synthesizing SQL generators would result in errors, requiring manual intervention (see Section 2.3). Rather, we propose *SQL sketches* for generator augmentation, in which a manually written *base generator* generates basic, common SQL statements, some of which contain *holes* to be filled by the LLM with code fragments of DBMS-specific features. The fragments synthesized are validated by a self-validation mechanism to eliminate invalid fragments caused by LLM hallucinations. To address the challenge that LLM-synthesized fragments are unaware of valid tables, columns, or types from the current database schema, and may produce deterministic rather than diverse outputs, we provide the LLM with schema information and random literal generators as part of the prompt.

System overview. Figure 1 shows an overview of *ShQveL*. During the execution of *ShQveL*, the base generator is either instructed to initiate the learning phase, aiming to learn new features, in which case *ShQveL* generates a SQL sketch (see step ①), or *ShQveL* generates test cases aiming to find bugs in the DBMS (see step ⑤). In the learning phase, *ShQveL* invokes an LLM to fill the SQL sketch (see step ②). One SQL sketch can contain multiple incomplete segments. Each incomplete segment ??—referred to as a *hole*—can be filled with DBMS-specific feature fragments (e.g., SQL keywords, constants, or expressions). We refer to each LLM-synthesized string that can

Fig. 1. Overview of *ShQveL*Table 2. SQL sketch design for *ShQveL*

Feature	Generation Rule	Example Sketch	Example Filling
Statement	Insert ?? as a whole statement	CREATE TABLE <u>TAB</u> (<u>COL</u> INT); ??;	CREATE TABLE <u>TAB</u> (<u>COL</u> INT); ANALYZE;
Clause	Insert ?? into statements	CREATE TABLE ?? <u>TAB</u> (<u>COL</u> INT ??, ??)??;	CREATE TABLE IF NOT EXISTS <u>TAB</u> (<u>COL</u> INT NOT NULL , PRIMARY KEY (<u>COL</u>));
Data type	Insert ?? as data type name and value	CREATE TABLE <u>TAB</u> (<u>COL</u> ??); INSERT INTO <u>TAB</u> VALUES (??);	CREATE TABLE <u>TAB</u> (<u>COL</u> ARRAY); INSERT INTO <u>TAB</u> VALUES ([1, <RANDOM_INT>]);
Expression	Insert ?? as expression node	CREATE TABLE <u>TAB</u> (<u>COL</u> ARRAY); SELECT ??(<u>COL</u>) FROM <u>TAB</u> ;	CREATE TABLE <u>TAB</u> (<u>COL</u> ARRAY); SELECT ARRAY_SUM (<u>COL</u>) FROM <u>TAB</u> ;

be substituted into a hole as a *fragment*, and a sketch whose holes are all filled is a *filled sketch*. *ShQveL* then invokes an LLM to fill the SQL sketch, using the target DBMS documentation and few-shot examples as guidance. The LLM fills the holes with fragments, and the resulting filled sketch is then executed on the system for validation (see step ③). If execution is successful, we mark the fragments as *valid*; otherwise, we mark them as *invalid*. In step ④, *ShQveL* updates the generator using the validated fragments. Based on the updated generator, *ShQveL* keeps generating SQL statements to test the target system (see step ⑤).

3.1 Sketch Generation

In step ①, *ShQveL* generates multiple SQL sketches for learning and integrating specific SQL features into the generator. The underlying base generator generates concrete SQL statements by sequentially appending string fragments—SQL keywords, schema references, operators, and randomly generated constants. For example, a typical statement is assembled by concatenating the strings `SELECT`, `col0`, `>`, `42`, `FROM` and `t0`. To create a SQL sketch, *ShQveL* inserts a hole at developer-specified locations in the generated SQL statements for the feature to be learned. For example, as shown in Figure 1, the generator might generate a sketch that allows the LLM to specify a binary operator by outputting the placeholder `??` instead of a concrete operator. The generator might thus generate an incomplete `SELECT` statement (e.g., `SELECT COL ?? 1 FROM TAB`), which the LLM can subsequently fill by synthesizing one of many plausible operators.

Sketch design. Table 2 shows the example sketches and their corresponding fillings at different levels. One or more holes (denoted by `??`) can exist in one sketch. The four hole types—*statement*, *clause*, *expression*, and *data type*—mirror the SQL feature taxonomy that *SQLancer++* adopts to enumerate dialect-specific SQL features [54]. Automatically inferring new hole locations from feedback is left as future work (Section 6). For clause-level features, the developer can replace any segment of an SQL statement with a hole or insert a hole into the statement; the LLM then fills the

hole, and *ShQveL* integrates the fragment into the generator. For statement-level features, the entire SQL statement serves as a hole for the LLM to fill and incorporate, and for other feature levels, the developer specifies hole placement by annotating specific sites in the generator source (Section 4).

Context statements. To help the LLM synthesize fragments that depend on database state, each SQL sketch includes complete SQL statements that provide context in addition to statements with holes. These context statements convey schema information to the LLM. For example, a SELECT statement that queries an existing table typically requires preceding CREATE TABLE statements. In Figure 1, where *ShQveL* is learning the comparison operator feature, the SELECT statement with a hole appears below the CREATE TABLE and INSERT statements, which establish the database state as table TAB and insert sample data into column COL. Conversely, when LLM-synthesized fragments contain COL or TAB, the generator can identify them as database objects. Consequently, the generator can abstract concrete column references in a fragment so that, during testing, it instead references a column available in the current schema. As a result, fewer potentially ambiguous natural language instructions are needed in the prompt design, and the filled sketch can be executed to validate that the filled fragments are correct.

Fragment integration. Our sketch design enables *ShQveL* to reuse LLM-learned fragments across feature levels. After a filled sketch passes validation, *ShQveL* integrates each synthesized fragment by attaching it to the hole that generated it. Each hole samples from a validated fragment pool; data-type holes store paired type-value templates, while statement, clause, and expression holes store string fragments. For example, after learning ARRAY and corresponding insertable values from the data-type sketch in Table 2, row 3, the generator can use ARRAY as a column type in the context statements for expression-level sketches (row 4). The LLM can then synthesize fragments such as ARRAY_SUM(COL) that operate on the learned type.

3.2 Prompt Design and Fragment Synthesis

In step ②, *ShQveL* prompts the LLM to fill the holes in the sketches using in-context learning [5]. Two challenges for designing prompts limit this synthesis. First, the LLM’s training data might lack less common SQL dialects or recent SQL features. Second, LLMs tend to generate outputs based on common patterns seen in their training data. As further detailed in this section, *ShQveL* uses *Retrieval-Augmented Generation (RAG)* techniques to address the first challenge, and incorporates random literal generators to address the second challenge. Specifically, *ShQveL* prompts the synthesis LLM with up-to-date DBMS information through the documentation of DBMSs based on RAG techniques. Additionally, *ShQveL* exposes an interface to the LLM that allows it to generate random literals to increase the exploration of interesting and diverse test cases.

Prompt design. We subsequently elaborate on the prompt that *ShQveL* uses to interact with the LLMs (see Listing 3). Each synthesis prompt asks the LLM to fill the numbered placeholders {0}, {1}, ... in the generator-produced SQL sketch. At a high level, the prompt combines lightweight natural-language requirements (e.g., dialect constraints and determinism) with structured context, including the target DBMS, an up-to-date documentation summary via RAG, and the SQL sketch with its context statements. We implement this design by including a small few-shot set of previously validated fragments and instructing the LLM to output multiple candidates in a simple CSV-like format (*one fragment per line*: hole_id; fragment). This format enables deterministic parsing and direct substitution into sketches for validation (Section 3.3). We also explicitly instruct the LLM to avoid non-deterministic functions; any remaining non-deterministic fragments are handled by the keyword filter and false-alarm analysis described in Section 5.4.

Listing 3. An example prompt for fragment synthesis. **\$DBMS** represents the target DBMS name, **\$DOCUMENTATION** represents the summarized documentation, and **\$GENERATORS** represents the available literal generators (Table 3).

You are an expert in SQL dialects. Given the following SQL sketch in **\$DBMS**

```
CREATE TABLE TAB (COL ARRAY(STRING));
INSERT INTO TAB VALUES (['a', 'b']);
INSERT INTO TAB VALUES ({0});
INSERT INTO TAB VALUES ({1}(NULL));
INSERT INTO TAB VALUES ({2}(NULL, NULL));
SELECT {3}(COL) FROM TAB;
```

Task: For each placeholder ({0}, {1}, ...) in the SQL sketch, generate as many
↪ deterministic, rare, and complex concrete alternatives as possible...

Refer to examples from the documentation: **\$DOCUMENTATION**

When helpful, use generator identifiers from **\$GENERATORS** in your fragments.

Table 3. Examples of the random generators.

Generator Identifier	Description
RANDOM_INT	A random integer, e.g., 1
RANDOM_VARCHAR	A random string, e.g., abc
RANDOM_DATE	A random date, e.g., 2024/12/13
RANDOM_TABLE	A random table reference.
RANDOM_COLUMN	A random column reference.

Retrieval-augmented generation. *ShQveL* uses a RAG-based approach to incorporate up-to-date information from DBMS documentation into the synthesis prompt (see the **\$DOCUMENTATION** field in Listing 3). Although relying solely on the LLM’s internal knowledge is possible, its performance can be limited, especially for emerging DBMSs whose specifications would likely be missing in the model’s training data (see Section 5.2). LLMs are pre-trained and updated infrequently; thus, they cannot capture the latest knowledge of the target DBMS. Besides, their capacity limits their ability to provide detailed, accurate information for less common DBMSs.

To address the above issues, *ShQveL* uses a three-stage retrieval-augmentation process, as shown in function `rag` in Algorithm 1 (lines 15–19) and illustrated here with a running example $D = \text{CrateDB}$, $f = \text{ARRAY}$, $\ell = \text{data type}$. First, *ShQveL* concatenates D , f , and a short descriptor of the feature level into a natural-language query (e.g., “CrateDB ARRAY data type overview”). Second, *ShQveL* submits the query to a search engine and fetches the top N official DBMS documentation pages for f ($N = 3$ in our evaluation); the same component could alternatively be implemented using embedding-based retrieval by indexing the documentation in a vector database. Third, *ShQveL* invokes a lightweight LLM (e.g., GPT-4o-mini) to distill the retrieved pages into a compact reference summary (**\$DOCUMENTATION**) that lists, for every feature the page describes, a triple (name, description, concrete SQL example), reducing the reference size to fit within LLM context limits.

Random generators. *ShQveL* exposes a function interface to the LLM for generating random literals to explore a larger search space during testing (see \$GENERATORS in Listing 3, instantiated from Table 3). LLMs are likely to generate answers that are frequently seen in their training data, while in testing, using values that are not frequently seen may trigger bugs. See Listing 2 Issue 1 as an example. LLMs can return concrete fragments (e.g., `CHECK (col_name > 0)` for a column constraint clause); however, a bug may be triggered only when comparing `col_name` to a large integer or another column reference.

We incorporate random literal generators as callable functions within the code fragments. Table 3 shows examples of interface functions. They are designed to generate random literals—integers, strings, or identifiers. We follow the general methodology established by *SQLancer++* to bias the random generators toward fuzzing boundary values, which are more likely to trigger logic bugs. These values include integer values near `MAX_INT`, `MIN_INT`, and `0`; empty, long, and Unicode strings; extreme time values. We introduce the schema-identifier generators `RANDOM_TABLE` and `RANDOM_COLUMN` so that LLM-synthesized fragments can reference valid schema objects at test time. When *ShQveL*'s generator incorporates new fragments that include these keywords, it automatically invokes the matching random literal generators, dynamically generating randomized values to be included in the generated SQL statements. By prompting LLMs with literal generators, one potential fragment could be `CHECK (col_name > <RANDOM_INT>)`. During testing, the literal generator will be invoked, and a comparison with a random integer will be generated.

3.3 Sketch Validation

In step ③, *ShQveL* validates the synthesized code fragments using execution feedback from the target systems. Due to LLM hallucinations, unsupported SQL features may be returned. To tackle this, we propose a validation mechanism. Otherwise, even a small number of invalid features would result in many invalid test cases, rendering the testing process inefficient. Sketch validation determines only whether a filled sketch is accepted by the target DBMS; semantic equivalence between the query pairs that the test oracle compares is enforced by the test oracle itself (e.g., TLP [32]).

ShQveL validates the SQL fragments by executing the filled statements on DBMSs. *ShQveL* substitutes the hole in the sketch with the returned fragment and executes the filled statements to check whether the features learned are supported (see step ③). After filling the holes, the statements are expected to be a self-contained SQL test case, that is, given an initial clean database state, we can execute this test case to create a table, insert data, and query it without error. If execution of the statements is successful, we can infer that the fragments filled in have a high probability of being supported by the target system. Otherwise, the fragments filled in are marked as unsupported. In the following, we refer to a fragment as *valid* if it executes successfully when substituted into the sketch, and as *invalid* otherwise. For example, the unequal operator `<>` is valid, and the logical AND operator is invalid as shown in step ③. Note that such fragments may contain features such as functions for random data generation that pass the test in rare cases, while actually not being supported. We address this challenge during test execution (see Section 3.5). By integrating the fragments that contain no unsupported features, *ShQveL* ensures a high validity rate for test cases.

3.4 Feature Scheduling

ShQveL uses a feature scheduler to increase the likelihood of using newly learned fragments in the generated test cases. This addresses multiple issues. First, as the fragment set grows, it is increasingly unlikely that newly learned fragments will be sampled and incorporated into generated statements, which delays bug discovery in those new features. Besides, SQL features exhibit cross-level dependencies (Section 2.1): expression-level features (e.g., functions and operators) require

Algorithm 1: Feature-oriented testing in *ShQveL*.

```

1 repeat
2    $\mathcal{P} \leftarrow \text{initializeTypes}(D)$ ;
3    $\mathcal{L} \leftarrow \emptyset$ ;
4   repeat
5     // Pick next type  $\tau \in \mathcal{P} \setminus \mathcal{L}$ 
6      $\tau \leftarrow \text{nextType}(\mathcal{P}, \mathcal{L})$ ;
7     // Documentation summary via RAG (Section 3.2)
8      $\mathcal{R} \leftarrow \text{rag}(D, \tau, \text{datatype})$ ;
9     // Synthesize fragments for  $\tau$ 
10     $\mathcal{N} \leftarrow \text{synthesize}(\tau)$ ;
11    while  $\text{curQueries} < \text{maxQueries}$  do
12      // Biased generation toward  $\mathcal{N}$ 
13       $\mathcal{D} \leftarrow \text{generateDatabase}(\mathcal{N})$ ;
14       $Q \leftarrow \text{generateQueries}(\mathcal{D}, \mathcal{N})$ ;
15       $\text{oracleCheck}(\mathcal{D}, Q)$ ;
16     $\mathcal{L} \leftarrow \mathcal{L} \cup \{\tau\}$ ;
17  until  $\text{allTypesLearned}$ ;
18 until  $\text{timeout}$ ;

19 Function  $\text{rag}(D, f, \ell)$ :
20    $q \leftarrow \text{generateSearchQuery}(D, f, \ell)$ ;
21    $\mathcal{P}_{\text{doc}} \leftarrow \text{searchAndFetchTop}(q)$ ;
22    $\mathcal{R} : \langle \text{name}, \text{desc}, \text{example} \rangle \leftarrow \text{summarize}(\mathcal{P}_{\text{doc}})$ ;
23   return  $\mathcal{R}$ ;

```

prerequisite data types to be available in the generator. For example, a function like `ARRAY_CONCAT()` can only be meaningfully tested once the `ARRAY` data type has been learned.

Algorithm 1 shows the feature scheduler. It maintains a pool $\mathcal{P}$ of candidate dialect-specific data types and a set $\mathcal{L}$ containing all learned data types, where $\mathcal{P}$ is initialized by prompting an LLM to synthesize the DBMS’s supported data types through data type sketches. At each iteration (see line 5 in Algorithm 1), *ShQveL* selects the next unseen type $\tau \in \mathcal{P} \setminus \mathcal{L}$ and triggers a dedicated learning phase that synthesizes type-level fragments, including constants of τ , and related expression-level fragments that consume or produce τ (see line 7 in Algorithm 1). After integrating the validated fragments, *ShQveL* enters a type-focused testing phase of maxQueries queries (see lines 8–11 in Algorithm 1), during which it temporarily biases generation toward using the newly integrated fragments, for example by creating tables and views with columns of type τ and generating expressions over τ . Once this focused phase completes, *ShQveL* marks τ as learned by adding it to $\mathcal{L}$ and repeats the process by selecting the next unseen type; when $\mathcal{P} \setminus \mathcal{L}$ is empty, *ShQveL* re-initializes $\mathcal{P}$ and removes the already-learned types.

Suppose $\mathcal{P} = \{\text{JSON}, \text{ARRAY}, \text{DECIMAL}\}$ and $\mathcal{L} = \emptyset$. In one iteration, *ShQveL* selects $\tau = \text{ARRAY}$ and runs the dedicated learning phase to synthesize type-level fragments (e.g., `ARRAY` constants) as well as expression-level fragments that consume/produce `ARRAY` (e.g., functions such as `ARRAY_UNIQUE()` and `ARRAY_SUM()`). It then enters the type-focused testing phase and biases generation toward

using the new fragments, for example, by creating tables or views with ARRAY columns and inserting learned constant values into those columns.

3.5 Testing

In step ⑤, *ShQveL* generates test cases and applies the test oracles. Initially, it does so by using only manually implemented features. It subsequently expands the set of features (see step ④) by executing the learning phase. The learning incurs negligible overhead since the learning is decoupled from the test case generation as a separate thread, leaving the overall testing throughput essentially unchanged. During testing, *ShQveL* further validates the fragments at run time.

Run-time validation. *ShQveL* further validates learned fragments through a feedback mechanism during testing. This is because, during the self-validation in step ③, invalid fragments that fail non-deterministically can falsely pass the sketch validation. For example, the `<RANDOM_VARCHAR>` generator can generate a string `'0'`, which can be implicitly converted to an integer, meaning that we could incorrectly infer that, for example, an operator expecting only integer operands could process characters. To address this, we reuse the same Bayesian model in *SQLancer++* [54] to infer whether a feature is supported at run time. First, after each feature has been integrated, *ShQveL* generates test cases containing these features. Second, *ShQveL* calculates the estimated probability of each feature being executed successfully based on a statistical model [12]. Third, *ShQveL* ranks the features by their estimated probability and omits the features with low probability. This Bayesian feedback estimates only the probability of feature support—that is, whether executing a feature is accepted by the DBMS—and does not validate semantic equivalence between queries. The correctness of the generated test cases is still enforced by the test oracle.

4 Implementation

We introduce the implementation details of how *ShQveL* generates SQL sketches and integrates fragments into the generator.

Generator instrumentation. We modified the manually written generator to generate SQL sketches for features at four levels, following the sketch rules in Table 2. Specifically, we override the string construction logic of the generator and treat selected positions as *holes* that can be instantiated differently between learning and testing. In learning, the hole generates placeholders at these sites to form a sketch instance; in testing, it substitutes each placeholder with a candidate sampled from the corresponding validated fragment set. For example, in Table 2 (row 2), we insert a hole after `CREATE TABLE`: during learning, the hole becomes a placeholder, while during testing, it can be filled with `IF NOT EXISTS`. Each hole is introduced by overloading the generator's `append` method: a base call `append(default)` is replaced with `append(default, holeId)`, where `default` is the original fragment and `holeId` uniquely identifies the call site and serves as the key to bind the hole to its corresponding fragment pool. A global flag `isLearn` switches the method between learning and testing modes, which implements the two-phase process of Figure 1: during learning, it generates a numbered placeholder `{0}, {1}, ...`; during testing, it samples a validated fragment from the pool keyed by `holeId`. Listing 4 shows an excerpt at a function hole site, with the original line marked as removed and the instrumented call as added.

Fragment materialization. Fragments contain *dynamic components* that depend on the current database state, including schema object references (e.g., tables/columns) and random literal generators. To keep learned fragments reusable across schemas and executions, we represent such components using *unique identifiers* with a fixed name and description exposed to the LLM during learning (e.g., `RANDOM_INT`: generate a random integer value). During testing, the generator

Listing 4. Example excerpt of the generator instrumentation.

```

1 - sb.append("SIN");
2 + sb.append("SIN", Hole.FUNC);
3
4 void append(String def, Hole holeId) { // overloaded
5   if (isLearn) sb.append("{ " + nextHoleId() + "}");
6   else sb.append(fragmentPool.get(holeId).sample());
7 }

```

Table 4. *ShQveL* allowed us to find and report 55 bugs in 5 systems, of which 50 were fixed by the developers, and 39 were logic bugs. In terms of bug types, 8 were attributed to clause features, 26 to type features, and 21 to expression features. Notably, none of these bugs could be detected by SQLancer++ [54].

DBMS Names	All	Bug Status		Test Oracle		SQL Feature			Released	LOC	GitHub Stars	Source Language	Prior Tools
		Fixed	Conf.	Logic	Other	Clause	Type	Expr.					
CockroachDB	2	1	1	1	1	0	1	1	2014	2522k	31.7k	Go	[1, 31, 32, 38, 54]
CrateDB	21	21	0	20	1	7	8	6	2017	563k	4.3k	Java	[54]
DuckDB	16	15	1	13	3	0	15	1	2019	721k	35.2k	C++	[9, 10, 32, 50, 54]
MonetDB	11	11	0	2	9	1	1	9	2004	339k	0.5k	C	[9, 54]
TiDB	5	2	3	3	2	0	1	4	2016	1212k	39.6k	Go	[1, 32, 37, 38, 40, 50, 54]
Total	55	50	5	39	16	8	26	21					

materializes these identifiers into concrete values: it rebinds schema identifiers to valid objects in the current schema, and it evaluates random-generator identifiers into constants using a seeded generator.

5 Evaluation

We evaluated *ShQveL* from multiple perspectives. First, we evaluated the effectiveness of *ShQveL* in finding bugs over a long-term testing campaign, and analyzed the bugs we found from different levels of features (see Section 5.1). Second, we compared *ShQveL*—under different configurations—with state-of-the-art logic bug detection tools (see Section 5.2). Third, we measured the contribution of each component in *ShQveL* to bug finding effectiveness (see Section 5.3).

Baselines. We implemented *ShQveL* based on *SQLancer++*. It consists of 10.5K LOC written in Java for the base generator and validator, and 200 LOC of Python for building LLM agents. In comparison, *SQLancer* has 83K LOC and *SQLancer++* has 8.4K LOC. We compared *ShQveL* with *SQLancer++* and *SQLancer*. Our aim for *ShQveL* was to detect bugs in various DBMSs with different dialects. In addition, we sought to make it more effective and support more features than *SQLancer++*. We did not aim to outperform *SQLancer*’s manually written, dialect-specific generators; rather, our goal was to show the feasibility of fully automatic bug detection for dialect-specific features.

Setup. We conducted the experiments using a server with a 64-core EPYC 7763 at 2.45 GHz and 512 GB of memory running on Ubuntu 22.04. In *ShQveL*, we used the OpenAI APIs to query LLMs. Specifically, we used GPT-4o for SQL sketch synthesis and GPT-4o-mini for document summarization. During each testing iteration, we randomly created up to 2 tables, 1 view, and 20 inserts, and then executed 100K queries. These are the standard settings for *SQLancer* and *SQLancer++*. We also compared *ShQveL* under different LLM settings. “*ShQveL_{Model}*” denotes using only the LLM’s internal knowledge without DBMS documentation.

DBMS selection. We evaluated the bug-finding effectiveness of our approach on 5 DBMSs—CockroachDB, CrateDB, DuckDB, MonetDB, and TiDB (see Table 4). We used the latest development versions and reported bugs only when they could be reproduced on their latest versions. SQLancer provides support for all DBMSs except CrateDB, which is supported only by SQLancer++. Among the systems, CockroachDB, DuckDB, MonetDB, and TiDB have previously been the focus of both logic-bug [27, 38] and memory-bug [9] detection techniques. We also selected these five systems because their development communities actively investigate and resolve reported issues, enabling us to validate the uniqueness of any bugs we uncover. Although we also detected bugs in other popular DBMSs (e.g., MySQL and MariaDB), many previously reported bugs remain unfixed, making it challenging to determine whether any bug-inducing test cases trigger known bugs [14, 33, 38]. To avoid overloading the developers, we refrained from reporting any additional bugs for these systems. In our experiments on code coverage evaluation (see Section 5.2), we compared *ShQveL* with the baseline tools—*SQLancer*, *SQLancer++*, and *SQLRight*—on DuckDB, SQLite, and PostgreSQL, which differs from the DBMSs used for evaluating bug-finding effectiveness, since AFL-based tools (e.g., *SQLRight* [24]) cannot be applied to non-C/C++ systems—CockroachDB, CrateDB, and TiDB—and we can still trigger crashes on the latest version of MonetDB, which would cause any experiments to prematurely terminate.

5.1 Bug Finding Effectiveness

We continuously tested the five DBMSs during a six-month fuzzing campaign, followed by several months of intermittent testing. This methodology was also used to evaluate various other automated testing approaches, such as *SQLancer++* [54] and other testing approaches for DBMSs [1, 16, 33]. Because existing approaches have exhaustively tested the target systems [1, 9], any new bugs found by *ShQveL* indicate cases that existing tools missed. We ran *ShQveL* for several minutes to one day until it generated a number of bug reports. We employed the bug reduction and prioritization mechanisms of *SQLancer++*, and then we further processed the reduced and prioritized bug reports. After the bugs were fixed, we updated the DBMS to the latest version and started another testing run on it.

Bug statistics. Table 4 demonstrates the statistics of the bugs we reported. The full bug list is included in the artifact. In total, we reported 55 bugs, of which 50 bugs have been fixed, and the rest have been confirmed. 39 out of the 55 reported bugs were logic bugs, including 1 bug in CockroachDB, 20 bugs in CrateDB, 13 bugs in DuckDB, 2 bugs in MonetDB and 3 bugs in TiDB. *ShQveL* also found 16 other bugs causing hangs, internal errors, or crashes in the DBMS. This is expected, since automatically incorporating new features could help *ShQveL* detect issues in these previously untested or not well-tested features. *ShQveL* detected the logic bugs by the TLP [32] oracle, and detected other bugs (e.g., crash and hang) by monitoring the DBMS process status and response latency. These results are encouraging and demonstrate that *ShQveL* can find bugs that the DBMS developers are willing to fix.

Feature analysis. Table 4 presents bugs found by *ShQveL* augmented with SQL features at different levels. Most of the bugs found on DuckDB are caused by data type features, including BIT, TIMEZONE, and INET. These data types are not supported by *SQLancer*, which is why *ShQveL* could successfully find them. Most bugs in TiDB and MonetDB stem from expression-level features—functions and operators. Extending existing generators with new functions or operators might require less effort than adding clause- or statement-level features; however, manually covering hundreds of built-in or newly released features is impractical. *ShQveL* can automatically incorporate these new features.

Listing 5. A bug in TiDB when using a JSON function.

```

1 CREATE TABLE t0(c0 VARCHAR(500), c1 INT);
2 CREATE VIEW v0(c0) AS SELECT 'a' FROM t0;
3 INSERT INTO t0(c0) VALUES ('b');
4
5 SELECT * FROM t0 NATURAL RIGHT JOIN v0
6     WHERE (JSON_VALID(t0.c1)=0);
7 -- {} ✖ -- {'a' NULL} ✔

```

Listing 6. A large integer argument led to a CrateDB crash.

```

1 CREATE TABLE t0(c0 INT) WITH (number_of_replicas = 1000);
2 -- raise a SQLParseException
3 CREATE TABLE t0(c0 INT) WITH (number_of_replicas = 1925152226);
4 -- hang and crash

```

Listing 7. Column constraints cause CrateDB to incorrectly select a hash join plan.

```

1 CREATE TABLE t0 (c1 INT NOT NULL DEFAULT 1);
2 CREATE TABLE t1 (c1 INT);
3 INSERT INTO t0(c1) VALUES (1);
4 INSERT INTO t1(c1) VALUES (2);
5
6 SELECT * FROM t0, t1 WHERE
7     (t1.c1 >= 1) = ((t1.c1 = t1.c1) AND (t0.c1 <= t0.c1));
8 -- {} ✖ -- {1 2} ✔

```

Subsequently, we present the bugs we found at different feature levels. Our goal is to show the breadth of distinct bugs identified by each feature we learned. We present only reduced test cases that still contain the feature and highlight the core issue, rather than the original and semantically equivalent queries used to uncover these bugs.

Incorrect function result. Listing 5 demonstrates a bug in TiDB that was related to function-level features. The TiDB developers explained it was an inconsistent behavior for the `JSON_VALID` function when its argument is a column reference or a literal `NULL`. Surprisingly, the developers discovered that MySQL was also affected by this bug, and thus, they submitted an issue report to the MySQL developers. This bug was found by the TLP oracle; however, *SQLancer* failed to find it, since it lacked support for this function.

Table property clause crash. We found a bug in CrateDB (see Listing 6) in which a `CREATE TABLE` statement that set a large integer value as a table constraint led to a hang and crash of the system. The system crashed due to an integer overflow when validating the shards limit; however, CrateDB failed to catch the exception. The `WITH` clause is not defined in standard SQL and the attribute `number_of_replicas` is CrateDB-specific. *ShQveL* learned this statement-level fragment and used it while creating tables, which led to the crash. Although *ShQveL* was not explicitly designed for detecting crash bugs, our experiments demonstrate its capability to uncover several such issues. This bug also shows the benefit of the built-in random literal generator, since the LLM trained on the real-world dataset is unlikely to directly generate the code fragments with an overly large integer value.

Listing 8. DuckDB incorrectly evaluated a cast of a time value.

```

1 CREATE TABLE t1(c0 TIME WITH TIME ZONE);
2 INSERT INTO t1(c0) VALUES ('12:34:56');
3
4 SELECT (CAST(t1.c0 AS TIME) IN ('12:34:56')) FROM t1;
5 -- false ✖ -- true ✔

```

Table 5. Line and branch coverage on SQLite, PostgreSQL, and DuckDB after 24 hours, comparing *ShQveL* (with and without documentation summarization) against *SQLancer++*, *SQLRight*, *SQLancer*, and *Argus*.

Approach	SQLite		PostgreSQL		DuckDB	
	Line	Branch	Line	Branch	Line	Branch
<i>ShQveL</i>	41.3%	33.2%	31.6%	22.9%	44.6%	27.2%
<i>ShQveL_{Model}</i>	38.2%	30.6%	30.4%	21.9%	41.7%	24.9%
<i>SQLancer++</i>	30.3%	23.0%	25.4%	17.8%	31.6%	18.8%
<i>SQLRight</i>	47.5%	37.7%	44.9%	34.2%	-	-
<i>SQLancer</i>	46.6%	36.4%	32.3%	23.3%	33.4%	20.9%
<i>Argus</i>	-	-	30.1%	21.8%	37.1%	21.6%

False optimization by default clause. Listing 7 shows a bug caused by incorrectly selecting a hash join plan. As explained by the developers, the issue arises when joining tables whose filter involves multiple nested equality comparisons, and manifests only when the column constraints NOT NULL DEFAULT 1 are enabled. *ShQveL* learns the clause-level fragments and can generate a database schema with DBMS-specific features, which may trigger potential optimizations.

Data type. We found a bug in DuckDB (see Listing 8) where an IN expression was unexpectedly evaluated to false. We found this bug by the TLP oracle for which none of the partitioning queries of this expression predicate fetched any rows. As explained by the developers, this was due to a bug in the implementation of cast for TIME data types. They also mentioned that the support for this data type is not well-tested.³ Manually implementing these features into existing generators is labor-intensive. By automatically learning from an LLM, *ShQveL* can easily uncover untested features and trigger potential bugs.

5.2 Baseline Comparison

We measured the performance using multiple configurations of *ShQveL* and baselines. We compared the bug detection efficiency between *ShQveL* and *SQLancer++* on CrateDB 5.6, a historical version. Using a historical version for evaluating the efficiency is common practice [38, 50], as the uniqueness of bugs can be determined by identifying which commit fixed a bug and clustering all bug-inducing test cases by the commit that fixed them. We also evaluated the code coverage of *ShQveL* under different settings, *SQLancer*, *SQLRight*, *SQLancer++*, and *Argus* where applicable on three DBMSs written in C/C++: SQLite, PostgreSQL, and DuckDB. Although code coverage is not a crucial metric to measure the capabilities of finding logic bugs, it can help relatively compare how many features have been covered. We do not expect *ShQveL* to outperform *SQLancer* or *SQLRight*, since they require manually written generators or parsers, which are specific to each system, while *ShQveL* can be easily adapted to SQL dialects that are not supported by existing tools.

5.2.1 Coverage.

³<https://github.com/duckdb/duckdb/issues/13813>

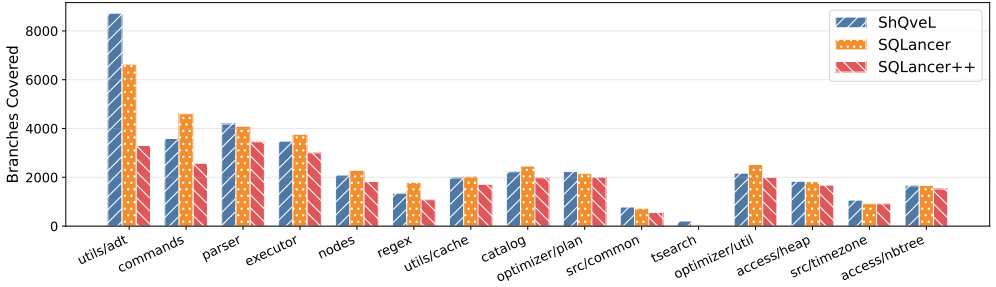


Fig. 2. Branches covered in PostgreSQL by top-level directory (from the lcov report), comparing *ShQveL*, *SQLancer*, and *SQLancer++*. We show the 15 directories with the largest coverage difference between *ShQveL* and *SQLancer++*.

Code coverage. Table 5 demonstrates the average line and branch coverage across 10 runs in 24 hours, which adheres to best practices [20]. *SQLRight*, which is based on *Squirrel*, achieves the highest coverage since it was designed to maximize code coverage. Besides, the coverage of initial seed inputs from each DBMS test suite already exceeds that of *SQLancer*. *SQLancer* achieves relatively higher coverage on PostgreSQL and SQLite. For *SQLancer* and *SQLRight*, significant manual effort was necessary (e.g., 9.7K LOC for the SQLite generator in *SQLancer* and an average of 7.9K LOC for *Squirrel*, on which *SQLRight* is based) to support these features, whereas *ShQveL* can achieve similar performance automatically. Note that on DuckDB, *ShQveL* outperformed *SQLancer*. One plausible explanation is that DuckDB, as an emerging and popular DBMS, evolves rapidly with new features, whereas *SQLancer* may not incorporate them as quickly. By incorporating LLM-derived features, *ShQveL* increases branch coverage over *SQLancer++* by 44.3% on SQLite, 28.7% on PostgreSQL, and 44.7% on DuckDB. Furthermore, the documentation summarization in *ShQveL* achieves 5%-9% more coverage than using only the LLM’s internal knowledge. *ShQveL* also achieves higher branch coverage than Argus [28], a concurrent LLM-based DBMS-testing approach, on both PostgreSQL and DuckDB.

Fine-grained coverage analysis. Figure 2 provides a finer-grained comparison of branch coverage in PostgreSQL across *ShQveL*, *SQLancer*, and *SQLancer++* in different source code directories. *ShQveL* significantly outperforms the baselines in the *utils/adt* directory—Abstract Data Type—and slightly outperforms the baselines in the parsing and planning pipeline (e.g., *parser*, *optimizer/plan*, and *optimizer/util*). However, *ShQveL* is less competitive in components like *commands*, *regex*, and some downstream execution paths (e.g., *access/heap*). One reason is that *SQLancer*’s customized generators have a higher validity rate—they more often satisfy prerequisite schemas, modes, and cross-clause dependencies due to manually specified rules or heuristics—so more generated inputs survive past parsing/planning and reach these components.

We observed that *SQLancer* can achieve higher coverage for *regex*-related functionality, because it includes a specialized *regex* expression generator. It reliably constructs valid patterns and generates operators or functions such as `~`, `regexp_match`, and `regexp_replace` with well-formed *regex* patterns as inputs. In contrast, *ShQveL* treats the *regex* pattern as an unconstrained string and may produce semantically invalid patterns that are rejected; learning such constraints might require a domain-specific grammar or specification for the generator. Conversely, the higher coverage of *ShQveL* in *utils/adt* is consistent with our feature-level dialect comparison in Table 6: *ShQveL* learns a wider variety of built-in data types and, together with our random literal generator,

Table 6. Feature-level dialect coverage comparison between *SQLancer* and *ShQveL*.

DBMS	<i>SQLancer</i>			<i>ShQveL</i>		
	Stmt.	Datatype	Expr.	Stmt.	Datatype	Expr.
SQLite	17	4	77	16	4	76
DuckDB	8	6	114	31	13	157
PostgreSQL	22	10	151	29	16	144

produces more diverse constants that stress type-specific parsing, coercions, and operators (e.g., NUMERIC/DECIMAL literals, nested ARRAY literals, and structured JSON/JSONB values).

Feature coverage. We quantify dialect coverage by counting distinct, deduplicated *features* at three levels: statements, data types, and expressions. We exclude clause-level fragments because embedded constants or identifiers make deduplication unreliable. For example, *ShQveL* may generate column-constraint clauses such as `CHECK COL > 1` and `CHECK COL < 1`. These differ only in the instantiated constant and comparison operator, not in the underlying core feature (e.g., a `CHECK` constraint over a column predicate). Counting them as separate clause features would therefore overcount what is essentially the same underlying clause feature. Parsing each clause-fragment into an AST may be a potential improvement, and we leave it as future work. For *ShQveL*, a feature is a validated, unique fragment learned from documentation and classified by the corresponding sketch level. For *SQLancer*, which lacks an explicit list of supported features, we approximate feature sets by inspecting its dialect-specific generators and grouping the covered AST node kinds into the same three levels.

Table 6 compares the resulting feature coverage on SQLite, DuckDB, and PostgreSQL. The results show that the feature coverage of *ShQveL* is comparable to *SQLancer* on SQLite and PostgreSQL, while achieving higher coverage at all three feature levels on DuckDB. On DuckDB, the additional features include more statement forms (e.g., bulk-load/export statements like `COPY`), built-in data types (e.g., JSON/JSONB, DECIMAL/NUMERIC, and nested types such as ARRAY), and expression nodes (e.g., JSON extraction, array/list constructors, and time-related functions). On PostgreSQL and SQLite, *SQLancer* achieves higher code coverage than *ShQveL*, because *SQLancer*'s handwritten generators are optimized to produce test cases that are less likely to be rejected by the DBMS, as described in the previous paragraph.

5.2.2 Bug Finding Efficiency. Figure 3 shows the bug detection efficiency of *ShQveL*, *ShQveL* without reference summarization (*ShQveL_{Model}*), and *SQLancer++* on CrateDB. In one hour, *ShQveL* detected more bugs than *ShQveL_{Model}*, as it generates features described in the documentation that would otherwise be missed. Listing 1 illustrates a class of bugs missed by *ShQveL_{Model}*; we found that LLMs often fail to provide dialect-specific SQL features for DBMSs that might be underrepresented in their training data. Both *ShQveL* and *ShQveL* without summarization can outperform *SQLancer++*, since *ShQveL* can discover new bugs by LLM-derived features. *ShQveL* found its first bugs slightly later than *SQLancer++* because it started learning from scratch, and summarizing the documentation takes time.

5.3 Ablation Study

We further measured the contributions and costs of individual components of *ShQveL*.

Feature importance. We measured the effectiveness of learning features at each level. We evaluated *ShQveL* on SQLite, PostgreSQL, and DuckDB by enabling learning only one type of feature during

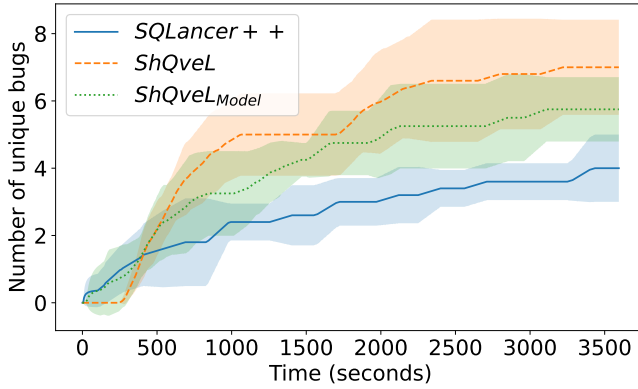


Fig. 3. Unique bugs found on CrateDB within one hour, averaged across 5 independent runs. The shaded area shows the standard deviation.

Table 7. Branch coverage achieved by *ShQveL* on SQLite, PostgreSQL, and DuckDB after 24 hours when learning exclusively from statement, clause, datatype, and expression features.

DBMS	Base	Statement	Clause	Datatype	Expression
SQLite	23.0%	28.6%	24.9%	23.6%	24.6%
PostgreSQL	17.8%	20.0%	18.0%	19.8%	19.8%
DuckDB	18.8%	22.9%	19.2%	22.4%	20.7%

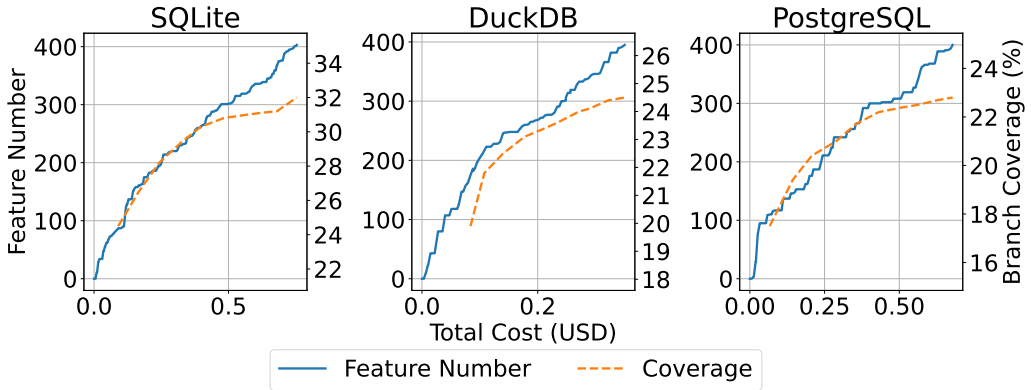


Fig. 4. The cost of LLM APIs to learn features on SQLite, DuckDB, and PostgreSQL over six hours.

each execution. Table 7 shows the incremental learning effect in terms of branch coverage of each individual level of feature. The base column represents using the base generator. Statement-level features lead to the greatest coverage increase across all three DBMSs, since the base generator supports only common SQL statements. *ShQveL* found no new logic bugs by learning new statement-level features, since most logic errors stem from issues in the query processor rather than the statement executor. Expression-level features are dependent on data types (see Section 3.1). However,

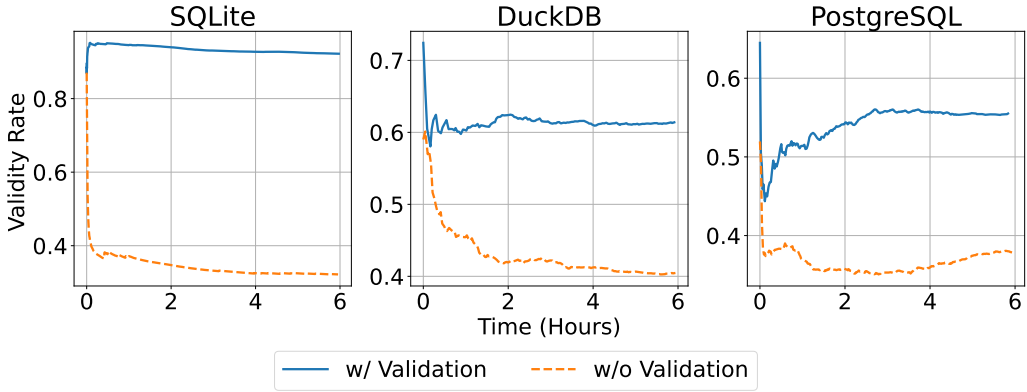


Fig. 5. The cumulative validity rate of test cases generated by *ShQveL* on SQLite, DuckDB, and PostgreSQL over six hours.

in our experimental setup, the functions and operators were learned without their associated data types, causing most of them to fail.

Feature learning costs. We measured the cost required by *ShQveL* to discover new features and improve branch coverage over a six-hour run on SQLite, DuckDB, and PostgreSQL. Each learning phase was triggered only after executing a sufficient number of test cases (e.g., 200K SQL statements). We incremented the feature count when *ShQveL* generated a valid, previously unseen code fragment for a given level of SQL sketch, filtering out duplicates and invalid fragments. We also recorded the branch coverage of the DBMSs during execution. Figure 4 demonstrates the cumulative number of learned features (solid line) and branch coverage (dashed line) against total API cost in USD. *ShQveL* learns over 400 features on all three DBMSs for under 1 US dollar, achieving branch coverage close to that of the original SQLancer. On DuckDB, it learns around 400 features for less than \$0.4 and outperforms SQLancer. Note that SQLite’s higher throughput triggers more frequent learning phases, and thus has a higher cost.

Validity rate. We measured the query and statement validity rates of *ShQveL* when learning new features over a six-hour run on SQLite, DuckDB, and PostgreSQL with and without the fragment validation mechanisms. Figure 5 shows the cumulative success rate of the generated SQL statements, including both DDL and DML statements, and queries over time. When fragment validation is disabled, the validity rate declines by 65.1%, 34.1%, and 31.8% on SQLite, DuckDB, and PostgreSQL, respectively. The validity rate remains above zero, since *ShQveL* generates common features besides the newly learned features. Among the three systems, SQLite achieves the highest overall validity, due to its dynamic typing and ability to coerce most values into the required types.

LLM usage is incurred once per DBMS during the initial learning phase: learning around 400 fragments costs at most \$1 per DBMS and completes in under six hours. After that, *ShQveL* invokes no LLM during test-case generation and its throughput matches *SQLancer++*. Learned fragments persist across runs and across DBMS versions, so the one-time cost is amortized over every subsequent CI run.

Table 8. False alarms observed at each step of the incremental filter construction on PostgreSQL 17.0.

Step	Filter size	Run time	False alarms
0	0	1 h	17.0
1	10	1 h	2.4
2	15	1 h	1.4
3	19	1 h	0.4
4	20	1 h	0
5	20	24 h	0

5.4 False Alarm Analysis

False alarms can arise when *ShQveL* learns fragments that violate the constraints of the test oracle (e.g., non-deterministic functions), leading to discrepancies in query results that may be misinterpreted as bugs. To reduce such alarms, *ShQveL* maintains a keyword filter that excludes any learned fragment containing such features. We sought to measure how often false alarms would arise without this filter, and the manual effort required to build it.

Results. We conducted the analysis on PostgreSQL 17.0, constructing the filter incrementally to observe how the number of false alarms decreased after each addition. At each step, we ran five *ShQveL* instances in parallel on PostgreSQL for one hour, collected all bug-inducing test cases, and manually classified them as true bugs or false alarms. For each false alarm, we added the responsible identifier to the filter. For the final step, in which no new entry was added, we extended the run to 24 hours to increase the chance of triggering any remaining false alarms. Table 8 reports the filter size and the average number of false alarms observed at each step. The resulting filter covers three types of fragments: non-deterministic functions, set-returning functions, and transaction-control statements.

6 Discussion

Design rationale. Our evaluation demonstrates that each component of *ShQveL* contributes to the approach’s overall effectiveness, thus justifying *ShQveL*’s design. First, to systematically cover the SQL features, we define four hole types (statement, clause, expression, data type) corresponding to the taxonomy of prior work [54]; the coverage study (see Table 7) shows that each level contributes incremental code coverage. Second, to address the LLM’s knowledge gap on emerging or less common DBMSs, we use a RAG-based approach to incorporate the information from the documentation of the target DBMS, which yields 5–9% additional branch coverage (see Table 5). Third, to prevent hallucinated fragments from reducing testing throughput, we filter fragments through an execution-based feedback mechanism; without it, the statement validity rate drops (see Figure 5), making testing inefficient. While each component adds implementation complexity, removing any one degrades effectiveness, as shown in our ablation study (Section 5.3).

Target systems. On PostgreSQL and SQLite, *ShQveL* improves branch coverage over *SQLancer++* (Table 5), but fails to detect new logic bugs. PostgreSQL and SQLite have been continuously fuzzed by tools like *SQLancer* (with 9.7K LOC of handwritten generators for SQLite alone) for several years, and are widely recognized as among the most thoroughly tested open-source DBMSs. Not finding new bugs in these systems is consistent with the extensive prior testing effort. For example, *DDLCheck*, *QPG* and *TQS* [1, 39, 40] likewise reported no new bugs on them.

Future work. Direct LLM-based SQL generation has limitations such as hallucination, low throughput, high cost, and incompatibility with established test oracles, yet interest in self-evolving agents that could synthesize SQL generators automatically is growing. We initially explored this direction (see Section 2.3), but at the time, LLMs struggled with internal DBMS state and the complex SQL grammar required for robust generator synthesis. With rapid progress in LLM capabilities, this direction merits renewed investigation: future agentic workflows could let an LLM synthesize, analyze, and continually refine test generators autonomously using execution feedback and external knowledge.

7 Related Work

LLM-aided testing. Recent work has leveraged LLMs for automated software testing across multiple domains. Fuzz4All [43] is a universal fuzzing framework that uses LLM-driven prompt generation and mutation to generate diverse test inputs in various programming languages. WhiteFox [44] is an LLM-based white-box compiler fuzzer where one model analyzes compiler optimization passes to derive input requirements and another model generates test programs. QTRAN [25] applies LLMs to translate query pairs across SQL dialects for metamorphic testing, fine-tuning a model to preserve oracle relations during mutation. These methods use LLMs to generate inputs directly, resulting in low throughput, and LLM hallucinations can lead to false bug reports, as reported in the QTRAN work. Conversely, *ShQveL* separates learning and testing, and generates inputs through traditional generators, making the process efficient and cost-effective. KernelGPT [45] is an approach to enhance OS kernel fuzzing by leveraging LLMs to synthesize system call specifications for fuzzers. MetaMut [30] is an LLM-guided mutation testing approach that integrates compiler domain knowledge into prompts to automatically generate high-quality mutators for compiler fuzzing. As in *ShQveL*, MetaMut invokes the LLM only during an initial phase and does not use it as a direct test case generator.

Test case generation for DBMSs. Existing methods for generating test cases for DBMSs can be divided into two categories: mutation-based testing and generator-based testing. Mutation-based testing for DBMSs is based on SQL-specific mutators to mutate existing SQL statements. Griffin [10] uses a grammar-free SQL fuzzing approach that replaces hand-written grammars by summarizing the database state in a lightweight metadata graph to guide semantically correct query mutations. BuzzBee [47] extends DBMS fuzzing to cover multiple database models. Sedar [9] improves the effectiveness of mutation-based testing by transferring SQL seeds across DBMS dialects, thus obtaining high-quality diversified inputs. None of the above mutation-based testing approaches can be applied to find logic bugs due to semantic constraints from test oracles. In contrast, generator-based approaches construct queries based on pre-defined rules. SQLsmith [34] is a representative tool, which utilizes schema metadata to generate well-formed random SQL queries. SQLancer [31–33] also generates queries based on various hand-written SQL generators. Both tools are built on manually implemented generators, and *ShQveL* can automatically augment these generators without human supervision.

DBMS test oracles. *ShQveL* can use existing test oracles. In this work, we implemented TLP and NoREC. TLP [32] detects logic bugs by executing a query and checking that its result matches the combined results of three queries that split the rows into separate parts. NoREC [31] detects logic bugs by executing a query that is receptive to optimizations and comparing its result to an equivalent version that is unlikely to be optimized. Various other test oracles could be integrated by *ShQveL*. EET [17] rewrites each query through equivalent transformations and checks that the rewritten query returns the same results as the original one. CODDTest [50] leverages compiler optimizations to find logic bugs in DBMSs, especially in advanced features like subqueries. APOLLO [19] finds

performance regressions by running random queries on different versions of the same DBMS, and flagging cases where the newer version runs much slower. Several other test oracles cannot be supported in a dialect-agnostic manner as they require DBMS-specific information. For example, CERT [2] requires manual effort to implement query plan parsers, with format usually differing across DBMSs, to find unexpected differences in the cardinality estimator. Radar [38] requires manual effort in designing and analyzing the metadata constraints for the raw database generation. Conceptually, *ShQveL* could be applied to augment the generators of all these oracles, thereby improving their bug-detection capabilities.

LLM- and AI-empowered systems. LLMs have recently been leveraged in database management for tasks such as query rewriting, SQL dialect translation, and text-to-SQL conversion. DB-GPT [56] treats the LLM as the “brain” of the DBMS that can adaptively handle tasks like automatic query reformulation and index recommendation. LLM-R² [23] is a rule-based rewrite framework that uses an LLM to recommend rewrite rules drawn from existing SQL rewriting platforms. CrackSQL [55] combines rule-based techniques with LLMs to translate between SQL dialects. FinSQL [49] is a model-agnostic, LLM-based text-to-SQL framework for financial data analysis. Similar to the above systems, *ShQveL* also leverages LLM knowledge in database management, but focuses on enhancing DBMS testing tools.

CatSQL [8] combines rule-based SQL sketch templates with deep learning models to fill in query details, improving the accuracy and reliability of NL2SQL translation. *ShQveL* shares a similar idea with CatSQL in that both leverage a language model to fill SQL sketches; however, *ShQveL* differs in the following respects. First, CatSQL focuses on SELECT queries, whereas *ShQveL* supports all kinds of SQL statements. Second, in CatSQL, the filled sketches are final system outputs. In contrast, *ShQveL* uses the filled sketches to improve its generator, and the generator generates outputs. Third, CatSQL translates natural language queries to SQL queries, while *ShQveL* aims to learn features of different dialects for DBMS testing.

Validation of LLM-generated SQL. Existing works leverage different approaches to validate LLM-generated SQL. LITHE [7] validates semantic equivalence between LLM-suggested rewrites and original queries. QO-Verify [29] uses query optimizers to verify LLM-based rewrites. SQLDriller [48] generates counterexample database instances to check that a natural-language query and its candidate SQL translation agree on execution. Most closely related to *ShQveL* is the concurrent work Argus [28], which prompts an LLM to synthesize equivalent query pairs as test oracles and validates the equivalence using a SQL equivalence prover. Argus addresses the long-standing challenge of designing high-quality test oracles for automated DBMS testing. In contrast, *ShQveL* uses the LLM to synthesize test inputs fed into a traditional generator and paired with existing oracles such as TLP and NoREC, targeting the problem of improving test input diversity across SQL dialects, and supports a broader range of SQL features (e.g., DML statements) than Argus. The two directions are complementary: oracles generated by Argus could be combined with *ShQveL*’s generator augmentation to further enhance testing effectiveness.

8 Conclusion

Existing SQL generators for logic bug detection require significant human effort for adaptation to various DBMSs. In this paper, we have presented an LLM-aided approach, *ShQveL*, for automatically augmenting these existing generators. To the best of our knowledge, this is the first approach that utilizes LLMs within an automated testing tool for DBMSs. Besides, we introduce the notion of SQL sketches to incorporate SQL features automatically into query generators. Conceptually, we believe that techniques inspired by our approach could be applied to efficiently utilize LLMs in the context of migration of queries to other dialects, text-to-SQL, or autocompletion in SQL editors.

ShQveL has discovered 55 previously unknown bugs in five DBMSs, 50 of which have been fixed. While manually written dialect-specific generators still outperform *ShQveL* in some cases—3% more branch coverage on SQLite—the difference is small. Furthermore, our approach can be applied to any SQL-based DBMS without any implementation effort.

Acknowledgments

We would like to sincerely thank all the anonymous reviewers for their valuable feedback and insights that helped us improve the quality of this paper. We want to thank all the DBMS developers for responding to our bug reports and for analyzing and fixing the issues we identified. This research is supported by the Google South & Southeast Asia Research Award, the National Research Foundation, Singapore and the Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Fuzz Testing), and a Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant. Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of Google, the National Research Foundation, Singapore, the Cyber Security Agency of Singapore, or the Singapore Ministry of Education.

References

- [1] Jinsheng Ba and Manuel Rigger. 2023. Testing Database Engines via Query Plan Guidance. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) (ICSE '23). IEEE Press, 2060–2071. <https://doi.org/10.1109/ICSE48619.2023.00174>
- [2] Jinsheng Ba and Manuel Rigger. 2024. CERT: Finding Performance Issues in Database Systems Through the Lens of Cardinality Estimation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 133, 13 pages. <https://doi.org/10.1145/3597503.3639076>
- [3] Jinsheng Ba and Manuel Rigger. 2024. Keep It Simple: Testing Databases via Differential Query Plans. *Proc. ACM Manag. Data* 2, 3, Article 188 (May 2024), 26 pages. <https://doi.org/10.1145/3654991>
- [4] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. 2020. Boosting fuzzer efficiency: an information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) (ESEC/FSE 2020). Association for Computing Machinery, New York, NY, USA, 678–689. <https://doi.org/10.1145/3368089.3409748>
- [5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
- [6] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) (ISSTA 2023). Association for Computing Machinery, New York, NY, USA, 423–435. <https://doi.org/10.1145/3597926.3598067>
- [7] Sriram Dharwada, Himanshu Devrani, Jayant R Haritsa, and Harish Doraiswamy. 2026. LITHE: A Query Rewrite Advisor using LLMs.. In *EDBT*. 233–246.
- [8] Han Fu, Chang Liu, Bin Wu, Feifei Li, Jian Tan, and Jianling Sun. 2023. CatSQL: Towards Real World Natural Language to SQL Applications. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1534–1547. <https://doi.org/10.14778/3583140.3583165>
- [9] Jingzhou Fu, Jie Liang, Zhiyong Wu, and Yu Jiang. 2024. Sedar: Obtaining High-Quality Seeds for DBMS Fuzzing via Cross-DBMS SQL Transfer. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 146, 12 pages. <https://doi.org/10.1145/3597503.3639210>
- [10] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2023. Griffin: Grammar-Free DBMS Fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) (ASE '22). Association for Computing Machinery, New York, NY, USA, Article 49, 12 pages. <https://doi.org/10.1145/3551349.3560431>
- [11] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. *Proc. VLDB Endow.* 17, 5 (Jan. 2024), 1132–1145. <https://doi.org/10.14778/3641204.3641221>

- [12] Andrew Gelman, John B Carlin, Hal S Stern, and Donald B Rubin. 1995. *Bayesian data analysis*. Chapman and Hall/CRC.
- [13] GitHub. 2023. GitHub Copilot. <https://copilot.github.com> Accessed: 2025-02-22.
- [14] Zongyin Hao, Quanfeng Huang, Chengpeng Wang, Jianfeng Wang, Yushan Zhang, Rongxin Wu, and Charles Zhang. 2023. Pinolo: Detecting Logical Bugs in Database Management Systems with Approximate Query Synthesis. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 345–358.
- [15] 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. <https://doi.org/10.1145/3703155>
- [16] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. 2023. Detecting Transactional Bugs in Database Engines via Graph-Based Oracle Construction. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 397–417. <https://www.usenix.org/conference/osdi23/presentation/jiang>
- [17] Zu-Ming Jiang and Zhendong Su. 2024. Detecting Logic Bugs in Database Engines via Equivalent Expression Transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 821–835. <https://www.usenix.org/conference/osdi24/presentation/jiang>
- [18] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. <https://openreview.net/forum?id=VTF8yNQm66>
- [19] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. APOLLO: Automatic detection and diagnosis of performance regressions in database systems. *Proceedings of the VLDB Endowment* 13, 1 (2019), 57–70.
- [20] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (Toronto, Canada) (CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. <https://doi.org/10.1145/3243734.3243804>
- [21] Jinyang Li, Binyuan Hui, Ge Qu, Jiayi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C.C. 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 *Proceedings of the 37th International Conference on Neural Information Processing Systems (New Orleans, LA, USA) (NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 1835, 28 pages.
- [22] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).
- [23] 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. <https://www.vldb.org/pvldb/vol18/p53-yuan.pdf>
- [24] Yu Liang, Song Liu, and Hong Hu. 2022. Detecting Logical Bugs of DBMS with Coverage-based Guidance. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 4309–4326. <https://www.usenix.org/conference/usenixsecurity22/presentation/liang>
- [25] Li Lin, Qinglin Zhu, Hongqiao Chen, Zhuangda Wang, Rongxin Wu, and Xiaoheng Xie. 2025. QTRAN: Extending Metamorphic-Oracle Based Logical Bug Detection Techniques for Multiple-DBMS Dialect Support. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA033 (June 2025), 22 pages. <https://doi.org/10.1145/3728908>
- [26] Jie Liu and Barzan Mozafari. 2026. GenRewrite: Query Rewriting via Large Language Models. *Proc. ACM Manag. Data* 4, 1, Article 70 (April 2026), 26 pages. <https://doi.org/10.1145/3786684>
- [27] Shuang Liu, Chenglin Tian, Jun Sun, Ruifeng Wang, Wei Lu, Yongxin Zhao, Yinxing Xue, Junjie Wang, and Xiaoyong Du. 2024. Semantic Conformance Testing of Relational DBMS. *Proc. VLDB Endow.* 18, 3 (Nov. 2024), 850–862. <https://doi.org/10.14778/3712221.3712247>
- [28] Qiuyang Mang, Runyuan He, Suyang Zhong, Xiaoxuan Liu, Huanchen Zhang, and Alvin Cheung. 2026. Automated Discovery of Test Oracles for Database Management Systems Using LLMs. *Proc. ACM Manag. Data* 4, 3, Article 140 (May 2026), 28 pages. <https://doi.org/10.1145/3802017>
- [29] Vivek R. Narasayya and Surajit Chaudhuri. 2026. Leveraging Query Optimizers to Verify the Soundness of LLM-based Query Rewrites for Real-World Workloads, and More. In *16th Conference on Innovative Data Systems Research, CIDR 2026, Chaminade, CA, USA, January 18-21, 2026*. www.cidrdb.org. <https://vldb.org/cidrdb/2026/leveraging-query-optimizers-to-verify-the-soundness-of-llm-based-query-rewrites-for-real-world-workloads.html>
- [30] Xianfei Ou, Cong Li, Yanyan Jiang, and Chang Xu. 2025. The Mutators Reloaded: Fuzzing Compilers with Large Language Model Generated Mutation Operators. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (Hilton La Jolla Torrey Pines, La Jolla, CA, USA) (ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 298–312. <https://doi.org/10.1145/3622781>

3674171

- [31] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (Virtual Event, USA) (ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1140–1152. <https://doi.org/10.1145/3368089.3409710>
- [32] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 211 (nov 2020), 30 pages. <https://doi.org/10.1145/3428279>
- [33] Manuel Rigger and Zhendong Su. 2020. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 667–682.
- [34] Andreas Seltenreich. 2022. Sqlsmith. <https://github.com/anse1/sqlsmith>
- [35] Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 618–622.
- [36] Armando Solar-Lezama. 2008. *Program synthesis by sketching*. University of California, Berkeley.
- [37] Jiansen Song, Wensheng Dou, Ziyu Cui, Qianwang Dai, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2023. Testing Database Systems via Differential Query Execution. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2072–2084. <https://doi.org/10.1109/ICSE48619.2023.00175>
- [38] Jiansen Song, Wensheng Dou, Yu Gao, Ziyu Cui, Yingying Zheng, Dong Wang, Wei Wang, Jun Wei, and Tao Huang. 2024. Detecting Metadata-Related Logic Bugs in Database Systems via Raw Database Construction. *Proc. VLDB Endow.* 17, 8 (may 2024), 1884–1897. <https://doi.org/10.14778/3659437.3659445>
- [39] Jiansen Song, Wensheng Dou, Yingying Zheng, Yu Gao, Ziyu Cui, Wei Wang, and Jun Wei. 2025. Detecting Schema-Related Logic Bugs in Relational DBMSs via Equivalent Database Construction. *Proc. VLDB Endow.* 18, 7 (March 2025), 2281–2294. <https://doi.org/10.14778/3734839.3734861>
- [40] Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. 2023. Detecting Logic Bugs of Join Optimizations in DBMS. *Proc. ACM Manag. Data* 1, 1, Article 55 (may 2023), 26 pages. <https://doi.org/10.1145/3588909>
- [41] Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A. Saurous, and Yoon Kim. 2023. Grammar Prompting for Domain-Specific Language Generation with Large Language Models. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 65030–65055. https://proceedings.neurips.cc/paper_files/paper/2023/file/cd40d0d65bfbb894ccc9ea822b47fa8-Paper-Conference.pdf
- [42] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019. Synthesizing database programs for schema refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (Phoenix, AZ, USA) (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 286–300. <https://doi.org/10.1145/3314221.3314588>
- [43] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (Lisbon, Portugal) (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 126, 13 pages. <https://doi.org/10.1145/3597503.3639121>
- [44] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024. WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 296 (Oct. 2024), 27 pages. <https://doi.org/10.1145/3689736>
- [45] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2025. KernelGPT: Enhanced Kernel Fuzzing via Large Language Models (ASPLOS '25). Association for Computing Machinery, New York, NY, USA, 560–573. <https://doi.org/10.1145/3676641.3716022>
- [46] John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652.
- [47] Yupeng Yang, Yongheng Chen, Rui Zhong, Jizhou Chen, and Wenke Lee. 2024. Towards Generic Database Management System Fuzzing. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 901–918. <https://www.usenix.org/conference/usenixsecurity24/presentation/yang-yupeng>
- [48] Yicun Yang, Zhaoguo Wang, Yu Xia, Zhuoran Wei, Haoran Ding, Ruzica Piskac, Haibo Chen, and Jinyang Li. 2025. Automated Validating and Fixing of Text-to-SQL Translation with Execution Consistency. *Proc. ACM Manag. Data* 3, 3, Article 134 (June 2025), 28 pages. <https://doi.org/10.1145/3725271>
- [49] Chao Zhang, Yuren Mao, Yijiang Fan, Yu Mi, Yunjun Gao, Lu Chen, Dongfang Lou, and Jinshu Lin. 2024. FinSQL: Model-Agnostic LLMs-based Text-to-SQL Framework for Financial Analysis. In *Companion of the 2024 International Conference on Management of Data (Santiago AA, Chile) (SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 93–105. <https://doi.org/10.1145/3626246.3653375>

- [50] Chi Zhang and Manuel Rigger. 2025. Constant Optimization Driven Database System Testing. *Proc. ACM Manag. Data* 3, 1, Article 24 (Feb. 2025), 24 pages. <https://doi.org/10.1145/3709674>
- [51] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lema Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, et al. 2023. Siren’s song in the AI ocean: a survey on hallucination in large language models. *arXiv preprint arXiv:2309.01219* (2023).
- [52] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) (*ISSTA 2024*). Association for Computing Machinery, New York, NY, USA, 1592–1604. <https://doi.org/10.1145/3650212.3680384>
- [53] Suyang Zhong and Manuel Rigger. 2024. Understanding and Reusing Test Suites Across Database Systems. *Proc. ACM Manag. Data* 2, 6, Article 253 (Dec. 2024), 26 pages. <https://doi.org/10.1145/3698829>
- [54] Suyang Zhong and Manuel Rigger. 2026. Scaling Automated Database System Testing. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (USA) (*ASPLOS ’26*). Association for Computing Machinery, New York, NY, USA, 1677–1692. <https://doi.org/10.1145/3779212.3790215>
- [55] Wei Zhou, Yuyang Gao, Xuanhe Zhou, and Guoliang Li. 2025. Cracking SQL Barriers: An LLM-based Dialect Translation System. *Proc. ACM Manag. Data* 3, 3, Article 141 (June 2025), 26 pages. <https://doi.org/10.1145/3725278>
- [56] Xuanhe Zhou, Zhaoyan Sun, and Guoliang Li. 2024. Db-gpt: Large language model meets database. *Data Science and Engineering* 9, 1 (2024), 102–111.

Received 17 January 2026; revised 19 May 2026; accepted 12 June 2026