Automated Discovery of Test Oracles for Database Management Systems Using LLMs

QIUYANG MANG*, UC Berkeley, USA
RUNYUAN HE, UC Berkeley, USA
SUYANG ZHONG, National University of Singapore, Singapore
XIAOXUAN LIU, UC Berkeley, USA
HUANCHEN ZHANG, Tsinghua University, China
ALVIN CHEUNG, UC Berkeley, USA

Since 2020, automated testing for Database Management Systems (DBMSs) has flourished, uncovering hundreds of bugs in widely-used systems. A cornerstone of these techniques is *test oracle*, which typically implements a mechanism to generate equivalent query pairs, and subsequently runs the pair and identifies bugs by checking the consistency of their results. While running these oracles can be automated, designing the mechanism to generate equivalent queries remains a fundamentally manual endeavor. This paper explores the use of large language models (LLMs) to automate the discovery of equivalent queries in the design of test oracles, addressing a long-standing bottleneck towards fully automated DBMS testing.

Although LLMs demonstrate impressive creativity, they are prone to hallucinations that can produce numerous false positive bug reports. Furthermore, their high monetary cost and latency mean that LLM invocations should be limited to ensure that bug detection is efficient and economical. To this end, we introduce Argus, a novel framework built upon the core concept of the *Constrained Abstract Query*—a SQL skeleton containing placeholders and their associated instantiation conditions, *e.g.,* the placeholder must be filled by a Boolean column. Argus uses LLMs to generate pairs of these skeletons, with their equivalence formally proven using a SQL equivalence solver to ensure soundness. After that, the placeholders in the verified skeletons are instantiated with concrete, reusable SQL snippets that are also synthesized by LLMs to produce complex test cases. We have implemented Argus and evaluated it on five extensively tested DBMSs, discovering 41 previously unknown bugs, 36 of which are logic bugs, with 36 confirmed and 27 already fixed by the developers. The artifacts for Argus are available at https://github.com/joyemang33/Argus

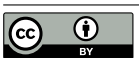
CCS Concepts: • **Information systems → Database management system engines**; • **Software and its engineering → Software testing and debugging**; • **Computing methodologies →** *Machine learning*.

Additional Key Words and Phrases: Database Management Systems, Software Testing, Test Oracles, Large Language Models, SQL, Logic Bugs

ACM Reference Format:
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 (SIGMOD), Article 140 (June 2026), 28 pages. https://doi.org/10.1145/3802017

*Corresponding authors: Qiuyang Mang (qmang@berkeley.edu) and Huanchen Zhang (huanchen@tsinghua.edu.cn).

Authors' Contact Information: Qiuyang Mang, UC Berkeley, USA, qmang@berkeley.edu; Runyuan He, UC Berkeley, USA; Suyang Zhong, National University of Singapore, Singapore; Xiaoxuan Liu, UC Berkeley, USA; Huanchen Zhang, Tsinghua University, China, huanchen@tsinghua.edu.cn; Alvin Cheung, UC Berkeley, USA.



ACM 2836-6573/2026/6-ART140
https://doi.org/10.1145/3802017

```
CREATE TABLE t1(c INT);
INSERT INTO t1 VALUES (1);
SELECT c / 3 FROM t1 WHERE false; -- {} ✔
SELECT c / 3 FROM t1 EXCEPT SELECT c / 3 FROM t1;
-- {0.3333} 🐞
```

Listing 1. Motivating bug: incorrect EXCEPT result in TiDB, which can only be detected by a specific test oracle.

1 Introduction

Database Management Systems (DBMSs) are a foundational component of modern software, yet their complexity makes them prone to bugs that can compromise application behavior and data integrity. Logic bugs are particularly insidious; they cause a DBMS to return incorrect results without raising errors, therefore silently corrupting downstream applications [47]. In response, the research community has developed automated testing techniques [24, 47–49, 74] that have discovered hundreds of bugs in real-world systems.

The most critical component of these techniques is the *test oracle* [6] that can determine the correctness of a query's output without access to the ground truth. Given a query, oracles for DBMS testing implement a transformation mechanism to generate a semantically equivalent variant. Then, by executing both queries and checking for result consistency, these oracles can detect logic bugs. For example, Ternary Logic Partitioning (TLP) [48] is a highly effective oracle that partitions a query Q based on a predicate P and then checks equivalence between Q and the union of its three-way partition Q `WHERE` P, Q `WHERE NOT` P, and Q `WHERE` P `IS NULL`. If a DBMS returns different results for the original query and its partitioned version, that indicates a bug.

While human-designed oracles have found many bugs (reported in over 20 top-conference papers), the manual creation of oracles is a bottleneck, trapping researchers in a cycle of designing increasingly specialized oracles to find bugs missed by previous ones. For example, a TiDB [22] bug introduced in 2019 Listing 1 went undetected for years despite extensive testing[1], because it required a specific oracle to check that any query Q `EXCEPT` Q should yield an empty result. This example highlights a fundamental challenge: manually designed oracles are not only difficult to conceive but also tend to overlook bugs. Automating the discovery of test oracles is therefore essential to enable scalable DBMS testing.

Large Language Models (LLMs), with their success in code generation [20, 41], offer a promising avenue for automating this process. A naive approach would be to prompt an LLM to generate a semantically equivalent variant for a given seed SQL query. However, this strategy suffers from two limitations:

(1) **Scalability:** LLM invocations are costly and have high latency. Compared to traditional SQL generators [16, 53], this is infeasible for modern DBMS testing, which often requires executing thousands of queries per minute to find bugs efficiently, such as in SQLancer [3].

(2) **Soundness:** LLMs are prone to hallucination and may generate query pairs that are not truly equivalent. Such unsound oracles produce false positives, thereby undermining the reliability of the testing process. Note that verifying equivalence empirically by running the query pairs on multiple database instances is unreliable, as this cannot guarantee the complete removal of semantically inequivalent pairs [19], and may also filter out true bugs that might exist on all databases under test [56].

[1] Commit 7de6200, introduced in 2019.

```
CREATE TABLE t1(c0 VARCHAR, ...);
CREATE TABLE t2(...);
SELECT * FROM t1, □1 ▷ Table(...);                    -- Q1
SELECT * FROM t1, □1 ▷ Table(...)
WHERE (□2 ▷ Expr(t1:BOOLEAN) IS TRUE) UNION ALL
SELECT * FROM t1, □1 ▷ Table(...)
WHERE (□2 ▷ Expr(t1:BOOLEAN) IS FALSE) UNION ALL
SELECT * FROM t1, □1 ▷ Table(...)
WHERE (□2 ▷ Expr(t1:BOOLEAN) IS NULL);                -- Q2
□1 ▷ Table(...) ↦ t1 ASOF JOIN t2
□2 ▷ Expr(t1:BOOLEAN) ↦ json_valid(t1.c0)
```

Listing 2. An example of representing and instantiating TLP [48] oracle in CAQ.

Key insights. To address these challenges, we propose Argus, a novel, fully automated framework for finding logic bugs in DBMSs. Argus uses a two-stage process that separates oracle discovery from test case generation. First, it leverages an LLM in an offline phase to discover reusable test oracles. Then, these oracles are formally verified for correctness before being instantiated into thousands of concrete test cases. Since generating abstract test oracles is much cheaper and faster than directly using LLMs to craft concrete test cases, this approach tackles both scalability and soundness head-on. Intuitively, these abstract oracles are parameterized query templates that can be instantiated many times with different SQL snippets.

Test oracle representation. Argus introduces a novel and expressive representation for test oracles that can be easily understood by LLMs called the *equivalent Constrained Abstract Query (CAQ) pair*. A CAQ is an abstract SQL query with *placeholders* that can represent tables, columns, predicates, and other SQL components. These placeholders are also associated with the corresponding constraints, which are essential to implement test oracles. For example, a CAQ with two placeholders might specify that placeholders must be syntactically identical, or that a predicate must be a Boolean expression when instantiated. An *equivalent CAQ pair* consists of two CAQs that are semantically identical for every possible instantiation of placeholders, *i.e.,* for all possible fillings of the placeholders with concrete SQL snippets that satisfy their constraints. CAQs allow Argus to represent a wide range of test oracles, including several existing ones. For instance, TLP [48] can be represented as an equivalent CAQ pair, *i.e.,* Q_1 and Q_2, as shown in Listing 2. In this representation, Q_1 serves as the original query template, while Q_2 is its three-way partitioned version. Here, to ensure the syntax validity of the instantiated queries from the two CAQs, $\square_1$ is restricted to represent any table, and $\square_2$ must be a Boolean expression constructed using the columns of table t_1. These two CAQs are semantically equivalent for all valid instantiations of the placeholders that satisfy their constraints, where $\square_1$ and $\square_2$ should also be instantiated consistently in both queries.

Test oracle generation, verification, and instantiation. To generate candidate oracles, Argus prompts an LLM with an initial CAQ and instructs it to produce a semantically equivalent one. To address the *soundness challenge*, each generated CAQ pair is then formally verified. We employ SQLSolver [13], a state-of-the-art SQL equivalence prover, to confirm their equivalence. If the prover successfully validates the equivalence, the CAQ pair is accepted as a valid test oracle; otherwise, it is discarded. Since all existing provers are designed for concrete queries, they cannot directly handle CAQs. We bridge this gap by *instantiating the placeholders with virtual tables and columns*, thus transforming them into concrete queries. This verification step ensures that only provably correct oracles are used for testing.

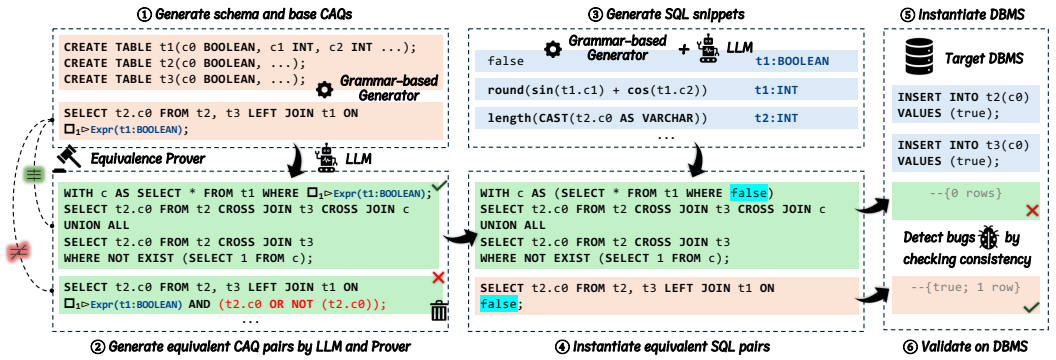


Fig. 1. Overall pipeline of Argus.

To tackle the *scalability challenge*, each verified oracle is instantiated into thousands of concrete test cases. Specifically, we populate the placeholders in each CAQ pair from a large corpus of pre-generated SQL snippets. This approach allows us to generate test cases for novel and complex features provided by the DBMS under test, even though they are not supported by our prover. For instance, in Listing 2, the placeholders $\square_1$ and $\square_2$ can be instantiated with various table joins and Boolean expressions, respectively, such as `t1 ASOF JOIN t2` and `json_valid(t1.c0)`. These snippets can be generated offline by a hybrid approach combining an LLM to cover diverse database features and a high-throughput generator, such as SQLancer [49], even though the prover currently cannot reason about them.

Evaluation. We have implemented Argus and evaluated it on five widely used and comprehensively tested DBMSs: Dolt [14], DuckDB [46], MySQL [40], PostgreSQL [38], and TiDB [22]. Argus discovered 41 unique and previously unknown bugs. Of these, 36 have been confirmed and 27 have been fixed by the developers. The bugs discovered include 36 critical logic bugs that lead to incorrect query results, while the remaining 5 cause performance problems or crashes. In our empirical comparisons with state-of-the-art open source testing tools in DuckDB [46], Argus demonstrated an improvement of up to 1.19× in code coverage, and 6.43 × in metamorphic coverage [2], a recently developed coverage indicator to assess the ability to find logic bugs. In addition, we also compared Argus with a union of existing test oracles, rewriting them into equivalent CAQ pairs equipped with the same query generator. The results show that Argus's new oracles can detect 3.33× more unique logic bugs than the sum of prior oracles from these works [5, 24, 47, 48] within 6 hours of testing on Dolt [14]. Our ablation studies further validate the effectiveness of the SQL equivalence prover in ensuring soundness and the contribution of CAQ to enhancing scalability. This paper makes the following contributions.

- We introduce the concept of the *equivalent CAQ pair* — a novel and expressive representation for test oracles in DBMSs — that can also capture features not supported by existing SQL equivalence provers. Based on that, we propose a new methodology that leverages LLMs to automatically discover test oracles for DBMSs by using them to generate CAQs. To our knowledge, we are the first tool that utilizes LLMs to generate DBMS test oracles.
- We address the *soundness challenge* by formally verifying the correctness of CAQ pairs, and address the *scalability challenge* by pre-generating a large corpus of reusable SQL snippets, which can then be used to efficiently instantiate equivalent CAQ pairs into numerous concrete tests.

- We perform an *extensive evaluation* on five widely tested DBMSs, uncovering 41 previously unknown bugs, which outperforms the state-of-the-art techniques in several coverage metrics and the number of unique logic bugs found.

2 Background

Test oracle. A test oracle is a mechanism for verifying whether a system's output is correct for a given input [6]. In the context of DBMS testing, test oracles typically operate by transforming a given SQL query into a semantically equivalent variant, as in TLP [48], NoREC [47], and EET [24]. By doing so, the oracle can compare the execution results of the original and transformed queries to detect potential logic bugs. In this work, CAQ provides a unified framework for LLMs to generate test oracles automatically. We show that most of these prior oracles can be formalized as instances of equivalent CAQ pairs, such as that shown in Listing 2. We include more examples in Appendix C.

SQL equivalence verification. The goal of SQL equivalence verification is to determine if two SQL queries are semantically equivalent, meaning they produce the same result when executed on all possible input database instances. SQL equivalence is generally undecidable [62], but partial deciders exist and are widely used in various scenarios such as query rewriting [35, 64] and text-to-SQL [70]. One line of work has focused on SQL equivalence provers that are sound (*i.e.,* they only confirm true equivalences) but not necessarily complete (*i.e.,* they may fail to identify all equivalent pairs) [10, 11, 13, 63]. On the other hand, there are also some works on SQL equivalence disprovers [19, 79], which can conservatively produce counterexamples for nonequivalent query pairs.

In this paper, we use LLMs to generate test oracles and use SQL provers to validate their equivalence. However, existing provers are limited, as they only support concrete SQL pairs and can reason about limited SQL features. To bridge this gap, we extend the provers to handle CAQs by introducing virtual tables and columns. By doing so, our framework can support many more features by instantiating these placeholders in the verified test oracles.

3 Argus Overview

We now use the detection of a previously unknown logic bug in DuckDB [46] as a case study to provide an overview of the Argus framework.

Fig. 1 presents the overall pipeline of Argus, which consists of two stages: *Test Oracle Discovery* (① and ②) and *Test Cases Instantiation* (③ – ⑥). The first stage aims to automatically discover a large number of high-quality test oracles in the form of equivalent CAQ pairs, while the second stage focuses on deriving concrete test cases from these CAQ pairs to detect bugs in DBMSs.

In ①, we generate a set of CAQs with their associated schemas by a grammar-based generator without using LLMs, and this will be detailed in Section 5.1. Specifically, we use SQLancer++ [82]'s query generator to produce seed queries, but we do not use their predefined test oracles (such as TLP [48] and NoREC [47]) for bug detection. For example, we generate the following CAQ:

```
SELECT t2.c0 FROM t2, t3 LEFT JOIN t1
ON □1 ▹ Expr(t1:BOOLEAN);
```

After that, in ② these CAQs will serve as seed queries for the subsequent LLM-based test oracle discovery. First, we employ an iterative prompting strategy to guide LLMs to generate various equivalent CAQs for each seed query, which will be elaborated in Section 5.2. For instance, given the above CAQ as input, the LLM may generate the following CAQs:

```
C1: WITH c AS (SELECT * FROM t1 WHERE □1 ▹ Expr(t1:BOOLEAN))
SELECT t2.c0 FROM t2 CROSS JOIN t3 CROSS JOIN c
```

```
UNION ALL
SELECT t2.c0 FROM t2 CROSS JOIN t3
WHERE NOT EXIST (SELECT 1 FROM c);

C2: SELECT t2.c0 FROM t2, t3 LEFT JOIN t1
ON □1 ▹ Expr(t1:BOOLEAN) AND (t2.c0 OR NOT (t2.c0));
```

Recall that LLM-generated CAQs are not guaranteed to be semantically equivalent due to the hallucinations. The first candidate, *i.e.,* C_1, is indeed a valid transformation. Its equivalence holds because it correctly expands the `LEFT JOIN` clause into a `UNION ALL`, and then uses a Common Table Expression (CTE) to reposition the `WHERE` clause. However, the second candidate, *i.e.,* C_2, is invalid since as it fails to consider the corner case of `NULL` values in SQL's three-valued logic.

To filter out such inequivalent candidates, we employ a SQL equivalence prover called SQL-Solver [13] for this purpose. Specifically, we will discard any candidate that cannot be proven equivalent to the input query. The details about how to communicate with CAQ pairs and concrete SQL provers will be presented in Section 5.3.

In the next stage, our task is to instantiate each verified CAQ pair into multiple concrete SQL query pairs by filling the placeholders in the CAQs. Before that, we generate a corpus of SQL snippets for placeholder filling, ensuring efficiency and cost-effectiveness during testing through reuse. This corpus also needs to be scalable, complex, and diverse enough to cover various database features.

In ③, we use a hybrid approach that integrates LLMs with a grammar-based generator, producing complex, feature-rich snippets (by LLMs) while maintaining scalability (by grammar-based generator). Note that LLMs may not generate the snippets that return the specified datatypes, or even generate invalid SQL snippets that will cause false positives during testing, such as non-deterministic features. To address this, we employ runtime validation to filter snippets. This will be discussed in Section 6.1.

With the generated corpus, in ④ we instantiate each CAQ pair into multiple concrete SQL query pairs by filling the placeholders with snippets from the corpus. The details about how to reuse the corpus for instantiating multiple CAQ pairs associated with different schemas will be presented in Section 6.2. For example, for the above equivalent CAQ pair, we must fill the placeholder `□1 ▹ Expr(t1:BOOLEAN)` with a snippet that produces a Boolean value from the table `t1`. For instance, the generated snippet `false` can be used to instantiate the CAQ pair into the following concrete SQL query pair:

```
Q1: SELECT t2.c0 FROM t2, t3 LEFT JOIN t1 ON false;
Q2: WITH c AS (SELECT * FROM t1 WHERE false) ...
```

Once we have the concrete equivalent SQL query pairs derived by LLM-generated test oracles, *e.g.,* $Q_1 \equiv Q_2$ in the above example, in steps ⑤ and ⑥ we can execute them on the target DBMS to check for bugs. For example, when we insert two rows `t2(c0:true)` and `t3(c0:true)` into an empty database, respectively, both queries should return one row. However, in DuckDB v1.4.0, Q_1 returns one row while Q_2 returns zero rows, indicating a logic bug in DuckDB. The root cause is that DuckDB incorrectly assumes that an empty materialized CTE will always cause the outer query to return no rows, which does not hold when there is a `UNION ALL` operator. The details of database instantiation and bug reporting will be discussed in Sec. 6.3.

4 Constrained Abstract Query

In this section, we formally introduce *Constrained Abstract Query (CAQ)*, the core concept of Argus.

$$
\begin{aligned}
\text{CAQ pair} &::= \text{Schema},\ \text{CAQ},\ \text{CAQ} \\
\text{Schema} &::= \texttt{SQLTableDef} \mid (\texttt{SQLTableDef},\ \text{Schema}) \\
\text{CAQ} &::= (\text{Query},\ \text{PlaceholderMap}) \\
\text{Query} &::= \texttt{SQLSelectStmt}[\square_1, \square_2, \dots] \\
\text{PlaceholderMap} &::= \{\square_1 \triangleright \text{Constraint}, \square_2 \triangleright \text{Constraint}, \dots\} \\
\text{Constraint} &::= \texttt{Expr(TableName : SQLDatatype)} \mid \\
&\qquad \texttt{Table(SQLTableDef)}
\end{aligned}
$$

Fig. 2. BNF Grammar for CAQ Pairs, where we omit the full SQL grammar for simplicity.

A CAQ (Constrained Abstract Query) is a parameterized SQL query template that contains placeholders to be instantiated with concrete SQL fragments, together with constraints specifying valid instantiations for each placeholder. Fig. 2 formalizes the structure of CAQ pairs using a BNF grammar. A CAQ pair consists of two CAQs and a shared schema that defines all referenced tables. Each CAQ is represented as a tuple (Query, PlaceholderMap), where Query is a `SQLSelectStmt` containing placeholders, and PlaceholderMap is a set that defines a constraint for each placeholder. Each placeholder $\square_i$ is constrained to be either an Expr or a Table. An Expr placeholder denotes an expression whose scope is bound to a specific source table and whose return type is specified (e.g., a `BOOLEAN` column reference, an expression that returns an `INT` or `TEXT`). A Table placeholder, on the other hand, denotes a table or a subquery, and its constraint specifies the expected schema (column names and datatypes) of the subquery result. Table placeholders may appear wherever an SQL table expression is valid, such as in a `FROM` clause or in nested subqueries, while Expr placeholders may appear wherever an expression can be evaluated within the scope of their associated table, such as in the `SELECT` clause or filter conditions. The two types of placeholders are designed to capture complementary SQL features: Expr placeholders model expressions involving functions and operators, whereas Table placeholders model compositional constructs involving subqueries, `JOIN`, `EXISTS`, and `IN`.

We now define how we instantiate a CAQ and how to check equivalence between a CAQ pair.

Definition 4.1 (CAQ Instantiation)**.** Given a CAQ q and its associated PlaceholderMap $\{\square_1 \triangleright c_1, \square_2 \triangleright c_2, \dots, \square_k \triangleright c_k\}$, an *instantiation* of q, denoted q^*, is obtained by replacing each placeholder $\square_i$ in q with a concrete SQL snippet s_i that satisfies the corresponding constraint c_i and a set of general constraints C. The substitution can be written as:

$$q^* = q[\square_1 \mapsto s_1, \square_2 \mapsto s_2, \dots, \square_k \mapsto s_k]$$

Specifically, each c_i refers to the Constraints shown in Fig. 2. In addition, to prevent false positive bug reports when using the instantiated query q^* for testing, we must also satisfy a set of general constraints C, such as avoiding non-deterministic features, *e.g.,* `RANDOM` and `CURRENT_TIMESTAMP`, and ensuring the snippet is valid SQL. Note that these are also widely adopted constraints in prior work on manually designed test oracles [3, 24, 48, 74]. The details of C will be discussed in Sec. 6.2.

We say that two CAQs are equivalent if all their possible instantiations are semantically equivalent.

Definition 4.2 (Equivalent CAQ Pair)**.** Given two CAQs q_1 and q_2 defined over the same Schema s (which specifies the tables referenced in both queries) with the same PlaceholderMap $\{\square_1 \triangleright c_1, \square_2 \triangleright c_2, \dots, \square_k \triangleright c_k\}$, we say that q_1 and q_2 form an *Equivalent CAQ Pair*, denoted (s, q_1, q_2), if for every possible instantiated concrete queries (Definition 4.1), q_1^* and q_2^*, we have $q_1^* \equiv q_2^*$, *i.e.,* they will return the same result set when executed on any database instance conforming to schema s.

Algorithm 1: Discovering New Test Oracles

Input: A DBMS $\mathcal{D}$, an LLM $\mathcal{M}$, a prover $\mathcal{P}$, a grammar-based generator $\mathcal{G}$, the number of schemas N, the number of maximum trials for each seed query MaxDepth.
Output: A set of test oracles $\mathcal{O}$.

1 **Function** GenerateEqual($\mathcal{M}, \mathcal{D}, \mathcal{P}, q$, StopThreshold $\leftarrow 3$)**:**
2 Equal $\leftarrow \{\}$, Fail $\leftarrow \{\}$, FailCount $\leftarrow 0$, Count $\leftarrow 0$;
3 **while** FailCount < StopThreshold **do**
4 $q' \leftarrow \mathcal{M}(q, \text{Sample(Equal)}, \text{Sample(Fail)})$;
5 **if** $\mathcal{P} \vdash (q \equiv q')$ and $\mathcal{D} \models q'$ // Sec. 5.3
6 **then**
7 Equal $\leftarrow$ Equal $\cup \{q'\}$;
8 FailCount $\leftarrow 0$;
9 StopThreshold $\leftarrow$ StopThreshold + 1;
10 **else**
11 Fail $\leftarrow$ Fail $\cup \{q'\}$;
12 FailCount $\leftarrow$ FailCount + 1;
13 Count $\leftarrow$ Count + 1;
14 **if** Count $\geq$ MaxDepth **then**
15 **break**;
16 **return** Equal;
17 $\mathcal{O} \leftarrow \emptyset$;
18 **for** $i \leftarrow 1$ **to** N **do**
19 $s \leftarrow \text{GenerateSchema}(\mathcal{G}, \mathcal{D})$; // Sec. 5.1
20 $q \leftarrow \text{GenerateCAQ}(\mathcal{G}, \mathcal{D}, s)$; // Sec. 5.1
21 $\{q_1, q_2, \ldots, q_k\} \leftarrow \text{GenerateEqual}(\mathcal{M}, \mathcal{D}, \mathcal{P}, q)$;
22 $\mathcal{O} \leftarrow \mathcal{O} \cup \{(s, q, q_1), (s, q, q_2), \ldots, (s, q, q_k)\}$;
23 **return** $\mathcal{O}$;

According to the above definition, once we verify that two CAQs form an equivalent pair, we can derive multiple concrete equivalent SQL query pairs by instantiating the CAQs with various snippets that satisfy the constraints of the placeholders. This allows us to decompose the test oracle discovery problem to CAQ pair generation and equivalence checking problems.

5 Test Oracle Discovery

We now present our test oracle discovery algorithm, outlined in Algorithm 1 and consists of three phases: database seeding, CAQ pairs generation, and equivalence checking. It leverages four key components as input: a target DBMS $\mathcal{D}$ (*e.g.,* DuckDB [46]), an LLM $\mathcal{M}$ (*e.g.,* GPT o4-mini [42]), a SQL prover $\mathcal{P}$ (*e.g.,* SQLSolver [13]), and a grammar-based generator $\mathcal{G}$ (*e.g.,* SQLancer [3]).

5.1 Database Seeding

To discover diverse test oracles, we generate CAQ pairs with multiple database schemas. As equivalent CAQ pairs are associated with their database schemas, we first iterate N times to generate diverse schemas and their corresponding CAQs (line 18). In each iteration, we start by database seeding, which involves generating a random database schema (line 19), and then producing a CAQ that conforms to the generated schema, *i.e.,* the seed query q (line 20). Though existing grammar-based generators, *e.g.,* SQLancer can be directly used for schema and concrete

```
1 CREATE TABLE t1(c0 INT, placeholder1 BOOLEAN);
2 CREATE TABLE t2(c0 INT);
3 CREATE TABLE vtable1(c0 INT, c1 INT);
4 SELECT t1.placeholder1 FROM t1, vtable1;
```

Listing 3. An example of using concrete SQL query to represent CAQ when interfacing with grammar-based generators and provers.

query generation, they cannot directly produce queries with placeholders. The key challenge lies in producing those entries in the seed CAQ while ensuring the syntactic correctness after instantiating them. To do so, we ask the generator to produce additional columns and tables in the schema, and then use those *virtual columns* and *virtual tables* to represent the two types of placeholders, respectively, *i.e.,* Expression and Table shown in Fig. 2. As shown in Listing 3, we create a virtual column `placeholder1` to represent an expression, *i.e.,* $\square_1 \triangleright$ `Expr(t1:BOOLEAN)`, and a virtual table `vtable1` for a table placeholder *i.e.,* $\square_1 \triangleright$ `Table(t0:INT, t1:INT)`. The generator, unaware of this abstraction, directly uses the concrete `placeholder1` and `vtable1`, effectively producing a valid CAQ. This approach makes our CAQ representation compatible with SQL provers expecting concrete syntax and easy for LLMs to understand.

5.2 CAQ Pairs Generation

After obtaining a seed CAQ q and its schema s, we proceed to generate a set of equivalent CAQs $\{q_1, q_2, \ldots, q_k\}$. This is done in GenerateEqual (line 1 – 16), which iteratively prompts an LLM $\mathcal{M}$ to produce novel variants of q. The key motivation behind using LLM here is to leverage its ability to understand the query semantics and guide the equivalent CAQ generation, while traditional grammar-based generators like SQLsmith [53] are typically not semantics-aware and thus struggle to generate equivalent queries. To generate variants efficiently, our key strategy is to leverage in-context learning by providing the LLM with not only the seed query q, but also carefully selected samples from two dynamically updated sets: Equal, *i.e.,* queries previously verified as equivalent to q, and Fail, *i.e.,* those that failed verification. The Equal set promotes both correctness and diversity, while the Fail set provides examples of failures to avoid.

A crucial aspect of our approach is how we guide the LLM towards generating diverse CAQs to trigger different optimization paths and detect potential bugs. Instead of pursuing syntactic differences, our goal is to produce queries with varied *query plans*, as this is a key principle to creating effective test oracles in previous works [3, 5, 74]. Qualitatively, we prompt the LLM to generate novel queries that are different from the provided examples (line 4). In addition, we also quantitatively measure the similarity between query plans using the tree edit distance [75]. This metric is then used to guide the selection of examples for the next iteration's prompt. Specifically, we use k-means to cluster the CAQs in the Equal set based on the tree edit distance value and then sample queries from each cluster. This strategy ensures the LLM is consistently shown a diverse range of successful query structures, pushing it to explore novel execution plans. We defer the details of the clustering and sampling algorithms to Appendix A. In summary, we can formalize the overall objective of the LLM's generation task as follows:

Problem 5.1 (Equivalent CAQ Generation)**.** Given a DBMS $\mathcal{D}$, a schema S, a seed CAQ q, and a set Equal of previously verified CAQs equivalent to q, generate a new CAQ q' such that (1) $q' \equiv q$ and (2) the difference between q''s query plan and the plans of queries in Equal is maximized.

After the LLM generates a candidate query q', we employ a SQL equivalence decider $\mathcal{P}$ to formally verify its equivalence to the seed query q, *i.e.,* $\mathcal{P} \vdash q \equiv q'$ (line 5). Additionally, we execute q' on the DBMS $\mathcal{D}$ to ensure it is syntactically correct and runs without errors, *i.e.,* $\mathcal{D} \models q'$. If both

checks pass, q' is added to the Equal set (line 7 — 9); otherwise, it goes into the Fail set (line 11 — 12). This iterative process continues until a stopping criterion is met, such as reaching a maximum number of failed attempts (line 3) or hitting a predefined depth limit for iteration (line 14).

5.3 Equivalence Checking

The CAQs generated by LLM are not guaranteed to be semantically equivalent. Hence, we use a SQL prover $\mathcal{P}$ to formally verify the equivalence between the seed CAQ q and a generated candidate CAQ q'. Since CAQs contain placeholders, they cannot be directly verified by SQL provers. Therefore, we substitute the placeholders with virtual columns and tables (as shown in Listing 3). We then pass the schema with these virtual entities to the prover and determine equivalence between the two concrete queries. This works because the virtual columns and tables behave like any concrete SQL entities of the same type during equivalence checking. If the prover can formally prove that $q \equiv q'$, we say the two CAQs and their schema s form a valid test oracle, denoted as the triplet (S, q, q'), which can be used to test the target DBMS $\mathcal{D}$ in the following process. We leverage the prover as a sound verification tool and deliberately retain only those CAQ pairs that are formally verified. This design choice prioritizes reliability, ensuring that our resulting test oracles are free from false positives and can be used confidently in the subsequent testing phase.

Another limitation of these provers is that they typically support only a subset of SQL features. To mitigate this, we first prove the equivalence between two CAQs, which typically contain fewer SQL features than concrete queries. After that, we can instantiate those PlaceholderEntry in the CAQs with more complex and feature-rich expressions and tables to create concrete queries for testing. This highlights the effectiveness of Argus's two-stage design, improving efficiency while also enriching the set of supported SQL features in the final test cases.

6 Test Cases Instantiation

We now describe our approach to instantiate test cases from the test oracles discovered in Sec. 5. As illustrated in Alg. 2, our approach consists of: (1) synthesizing a corpus of SQL snippets, (2) instantiating test queries by populating placeholders in the test oracles with those generated in the first step, and (3) creating database instances to execute the instantiated test queries to detect bugs.

6.1 Corpus Synthesis

The test case instantiation process begins with the synthesis of a SQL snippet corpus. This corpus provides a pool of expressions and tables used to populate the placeholders within our test oracles (line 15). We pre-generate this corpus rather than querying the LLM for each placeholder individually to amortize the cost of LLM invocations and improve efficiency, as many snippets can be reused across multiple test oracles. To generate an effective and diverse corpus, we employ a hybrid strategy that leverages both an LLM $\mathcal{M}$ and a grammar-based generator $\mathcal{G}$. We use this hybrid approach since: (1) LLMs excel at generating complex query structures and diverse database features, while (2) grammar-based generators can systematically cover a wide range of corner values and edge cases. As the grammar-based method follows established techniques, we focus this section on our novel LLM-based approach.

Our LLM-based method starts with a default schema, which contains a single table incorporating all data types supported by the target DBMS:

```
CREATE TABLE t(c1 INT, c2 BOOLEAN, c3 TEXT ...);
```

Using this schema, we prompt $\mathcal{M}$ to generate SQL expressions resulting in diverse datatypes from the default table. For example, we generate expressions such as `(c1 + c2)` : `INT`, `(c2 AND c3 IS NOT NULL)` : `BOOLEAN`, and `CONCAT(c3, "test")` : `TEXT`. The diversity comes from combining

Algorithm 2: Test Cases Instantiation Given Oracles

Input: A DBMS $\mathcal{D}$, an LLM $\mathcal{M}$, a grammar-based generator $\mathcal{G}$, a set of test oracles O generated in Sec. 5, and the number of test cases to be instantiated per test oracle K.
Output: A set of bug reports $\mathcal{B}$.

Function InstantiateTestCase($o = (s, q, q')$, $\mathcal{L}$) **:**
  $q^* \leftarrow q$, $q'^* \leftarrow q'$;
  foreach Placeholder e in q or q' **do**
    if e : Expression **then**
      $t \leftarrow$ TableName(e), $d \leftarrow$ Datatype(e);
      $e^* \leftarrow$ SampleExpression($\mathcal{L}$, type = d);
      foreach col $\in$ ColumnNames(e^*) **do**
        $d^* \leftarrow$ Datatype(col);
        $e^* \leftarrow e^*$[col $\mapsto$ SampleColumn(t, type = d^*)];
    else
      $s^* \leftarrow$ Schema(e);
      $e^* \leftarrow$ SampleTable($\mathcal{L}$, source = s, target = s^*);
    $q^* \leftarrow q^*[e \mapsto e^*]$, $q'^* \leftarrow q'^*[e \mapsto e^*]$;
  return (s, q^*, q'^*);
$\mathcal{L} \leftarrow$ GenerateCorpus($\mathcal{D}, \mathcal{M}, \mathcal{G}$) ; // Sec. 6.1
$\mathcal{B} \leftarrow \emptyset$;
foreach $o = (s, q, q') \in O$ **do**
  for $i \leftarrow 1$ **to** K **do**
    $(s, q^*, q'^*) \leftarrow$ InstantiateTestCase($o, \mathcal{D}$);
    CreateDatabaseInstance($s, \mathcal{D}$) ; // Sec. 6.3
    if Execute($q^*, \mathcal{D}$) $\neq$ Execute($q'^*, \mathcal{D}$) **then**
      $\mathcal{B} \leftarrow \mathcal{B} \cup \{(s, \mathcal{D}.\text{data}, q^*, q'^*)\}$; // Sec. 6.3
return $\mathcal{B}$;

multiple columns with various SQL operators and functions, not just selecting individual columns. These expression snippets can be reused across different test oracles by substituting the table and column names properly, which we will discuss in Sec. 6.2. Furthermore, to guide $\mathcal{M}$ with the target DBMS $\mathcal{D}$'s features, we employ a simple yet effective *documentation-augmented generation* method: We first collect the official documentation of the target DBMS, *e.g.*, function reference pages and operator descriptions, and partition it into smaller pieces, where each piece describes a specific feature, such as an aggregate function or a string operator. Then we sample a small set of feature documentation and prompt $\mathcal{M}$ to generate SQL snippets with the sampled features. This approach ensures that the generated SQL snippets are diverse enough to cover a wide range of features.

Note that, for table snippets, we can only reuse them in test oracles that share the same schema. This is because table snippets must match the exact schema of the source tables they reference—including the specific table names, column names, and their datatypes. For instance, a table snippet that performs `JOIN t1, t2` can only be applied to test oracles whose schema contains both tables `t1` and `t2` with compatible structures. Nevertheless, we can still leverage $\mathcal{M}$ to generate table snippets for each schema without much overhead, as the number of unique schemas is usually small in practice.

```
SELECT {GeneratedExpr} FROM t; -- datatype
SELECT * FROM {GeneratedTable}; -- schema
```

Listing 4. An example of runtime validating LLM-generated snippets in DuckDB [46].

```
Expression 1 =  (t.c1 + t.c1 > 100) : BOOLEAN
Expression 2 = bit_count(t.c2) : INT
⇒ bit_count(t.c1 + t.c1 > 100) : INT
```

Listing 5. An example of cross-combining expressions.

After the LLM-generated snippets, we need to validate whether they produce the expected datatype (*i.e.,* for expressions) or schema (*i.e.,* for tables) that we desire. To this end, we use runtime validation by executing the generated snippets on the target DBMS $\mathcal{D}$ and checking their actual output types or schemas. For example, as shown in Listing 4, we can directly run those `SELECT` statements and then check the output column types or table schema. If the execution fails, we discard the snippet; otherwise, we record its actual output type or schema for later use in Sec. 6.2.

Finally, to further generate more complex expressions, we introduce a technique we term *cross-combination*. The key idea is to substitute a column reference in one expression with another expression, provided that their datatypes are compatible. For instance, as shown in Listing 5, consider a Boolean expression `t.c1 + t.c1 > 100` and an integer expression `bit_count(t.c2)`, where `t.c2` is a Boolean column. Because the first expression evaluates to a Boolean, it can replace `t.c2` in the second one. This substitution yields a new, valid composite expression: `bit_count(t.c1 + t.c1 > 100)`. This recursive process allows us to build intricate expressions from simpler, generated snippets, which enriches our corpus and combines the strengths of LLM and grammar-based methods.

6.2 Query Instantiation

After synthesizing a corpus of SQL snippets, we instantiate the test queries by populating the placeholders in the test oracles with concrete expressions and tables from the generated corpus. We detail this process in Alg. 2: given a test oracle $o = (s, q, q')$, we iterate through each placeholder e in both queries q and q' (line 3). Depending on whether the placeholder represents an expression or a table, we handle them differently. For an expression placeholder, we first determine its expected datatype d and the table t that the placeholder is defined over (from the CAQ's placeholder map). Then, we randomly sample an expression e^* from the corpus $\mathcal{L}$ that matches the expected datatype d (line 6). Since the columns in the sampled expression e^* refer to the default table, we need to replace them with columns from the target table t that have compatible datatypes. We achieve this by iterating through each column in e^* (line 7), determining its datatype, and substituting it with a randomly sampled column from table t that matches the datatype (line 9).

For a table placeholder, we first extract its expected schema s^*. Then, we randomly sample a table e^* from the corpus $\mathcal{L}$ that matches the expected schema s^* and is derived from the source schema s (line 12). Specifically, there are three ways to replace a table placeholder: *i.e.,* direct replacement with a table name, using a Common Table Expression (CTE), or `CREATE VIEW`. We randomly choose one of the three methods. Finally, we replace the placeholder e in both queries q and q' with the instantiated expression or table e^* (line 14). By following this procedure, we ensure that the instantiated queries q^* and q'^* are syntactically valid and semantically equivalent, ready for execution on the target DBMS.

Recall that, in Sec. 5.3, we prove the equivalence of the abstract query pairs (q, q') under the semantics of virtual columns and tables. However, this equivalence is only guaranteed when those virtual columns and tables exist; this may not hold when the queries are instantiated with

concrete expressions and tables that introduce semantic misalignments. Therefore, to ensure that the instantiated queries q^* and q'^* remain equivalent, we enforce the following general constraints C (defined in Definition 4.2) on the sampled query snippets.

(1) **Determinism** (Table and Expression): The snippet should not contain any non-deterministic functions, such as `RANDOM()`.
(2) **Null-preserving** (Expression): The expression should preserve a null return value when evaluated on rows only containing null values. For example, `c1 + c2` is null-preserving, while `IFNULL(c1, 0)` is not.
(3) **Empty Results-preserving** (Expression): The expression should return an empty result set when evaluated on an empty table. For example, `sum(c1)` is not empty-preserving.

The first constraint is also widely used in previous works [47–49] and can be checked by applying a regular expression to identify non-deterministic features. The second and third constraints resolve the semantic misalignment that arises when virtual columns and tables are instantiated with concrete expressions and tables. These constraints are necessary because, in the presence of outer joins, virtual columns may no longer faithfully capture the semantics of concrete expressions after instantiation. *e.g.,* outer joins can introduce `NULL` values into virtual columns, causing instantiated expressions to be evaluated differently from the version fed to the SQL equivalence prover.

After satisfying these constraints, we can ensure that the instantiated queries q^* and q'^* remain equivalent, proven by contradiction: Intuitively, if the two queries after instantiation were not semantically equivalent, then there would exist a concrete database instance on which they produce different results. Using this instance, we can populate the virtual tables and columns in the prover's schema with the corresponding values. After that, we obtain a database instance under the schema used in the prover where the two queries return different results, which contradicts the original semantic equivalence. A detailed analysis of the counterexamples, together with the full proof for the cases satisfying them, is provided in Appendix B.

Furthermore, we validate these constraints at runtime: we execute each snippet on $\mathcal{D}$ and check the returned types or schemas and behavior on `NULL` or empty inputs.

6.3 Database Instantiation and Bug Reporting

After instantiating the test queries, we create the database instances to execute them (line 20 and 21). We basically follow the widely-used random data generation method [47–49] to populate the tables in the instantiated schema s with random tuples, and then create random indices on those tables to diversify the execution plans.

Finally, we execute the instantiated queries q^* and q'^* on the target DBMS $\mathcal{D}$ and compare their results (line 20). If the results differ, we report a logic bug along with the instantiated test case, database schema, and table data. In addition, if a query causes a crash of $\mathcal{D}$, we also report a crash bug. Note that, though our test oracles can also detect performance issues similar to [25, 36], we do not detect them in a large scale due to those issues are typically regarded as expected behaviors by developers [4].

7 Evaluation

In this section, we evaluate Argus on five aspects:

(1) How many new bugs can Argus find in real-world DBMSs, and what LLM-generated test oracles discover them? (Sec. 7.1)
(2) How does Argus compare to other baseline tools in terms of code coverage? (Sec. 7.2)
(3) How does the number of automatically discovered oracles affect the number of unique bugs found, compared to prior work where oracles are manually designed? (Sec. 7.3)

DBMS	Tested by	GitHub stars	Released	LOC
Dolt	[82]	19.1k	2018	380k
DuckDB	[15, 16, 48, 74, 82]	32.7k	2019	1,496k
MySQL	[24, 31, 49, 57, 58, 60, 61, 74, 80, 82]	11.7k	1995	5,532k
PostgreSQL	[16, 24, 30, 47, 49, 58, 80, 82]	18.5k	1995	938k
TiDB	[3, 24, 57, 58, 60, 61, 74, 82]	39.0k	2016	1,398k

Table 1. DBMSs under test by Argus.

DBMS	Reported	Bug status				Bug type	
		Fixed	Conf.	Dup.	Pend.	Logic	Other
Dolt	19	18	1	0	0	18	1
DuckDB	8	6	0	1	1	4	4
MySQL	8	0	5	1	2	8	0
PostgreSQL	1	1	0	0	0	1	0
TiDB	5	2	3	0	0	5	0
Total	41	27	9	2	3	36	5

Table 2. Argus found bugs statistics.

(4) How effective is Argus's SQL equivalence prover in filtering out false positives? (Sec. 7.4)
(5) What are the time and monetary costs required for Argus to generate a given number of test cases? (Sec. 7.5)

Testbed. We conducted all experiments using a machine with 64 cores and 128 GB memory running on Ubuntu 24.04. We use o4-mini [42] for all LLM calls with an Azure OpenAI API service in our experiments. We evaluate Argus on five widely-used DBMSs: Dolt [14], DuckDB [46], MySQL [40], PostgreSQL [38], and TiDB [22]. Note that, we only choose DBMSs that are actively maintained and have been comprehensively tested by at least one prior work, making new bug findings more valuable. The details of the target DBMS are shown in Table 1.

7.1 New Bugs and Oracles

Bug report policy. We continuously ran Argus on the five DBMSs over three months. We used the latest development versions and reported bugs only when they could be reproduced on their latest versions. To avoid rediscovering known bugs, we first carefully reviewed the open issues of the target DBMS and our previous reports. Whenever a new patch was released for one of our reports, we switched to testing the latest release of the target DBMS.

Overall result. Table 2 summarizes the statistics of the 41 bugs that Argus discovered during our testing campaign. In total, Argus detected 41 previously unknown bugs, of which 36 are logic bugs that cause incorrect query results. We also detected 5 other types of bugs, including crashes and performance issues. While finding such bugs is not the main focus of this paper, this still highlights the effectiveness of Argus to generate complex, concrete queries for testing. Among the reports, 36 bugs have been confirmed by the developers; 27 have already been fixed, while 9 are still in progress. The remaining 2 bugs are duplicates: after developers reproduced and analyzed them, they found that these bugs shared the same root cause as some previously unfixed bugs. The 36 logic bugs we found underscore the effectiveness of our LLM-generated test oracles. Compared with recent studies that use manually designed test oracles for bug finding (*e.g.,* reporting 21 [5], 24 [74], and 35 [24] logic bugs), Argus finds more logic bugs in comparison, even though the DBMSs

```
CREATE TABLE t(c INT);
INSERT INTO t VALUES (1);
SELECT sub.c FROM (
    SELECT □1 ▷ Expr(t:INT) ↦
    json_array_length(json_array(3, 2, t.c))
    AS c FROM t
) AS sub
RIGHT JOIN t ON FALSE; -- {2} 🐞
SELECT sub.c FROM (
    SELECT NULL AS c FROM t
) AS sub
RIGHT JOIN t ON FALSE; -- {NULL} ✔
```

Listing 6. Incorrect `json` functions handling in PostgreSQL when executing `RIGHT JOIN`.

```
CREATE TABLE t(c0 INT);
INSERT INTO t VALUES (1);
SELECT * FROM t LEFT JOIN (
    SELECT MOD(5, 2) AS c0 FROM t
) AS t2 ON FALSE
WHERE t2.c0 IS NOT NULL; -- {1} 🐞 {} ✔
```

Listing 7. A logic bug in MySQL found by the similar test oracle in Listing 6.

under test have already been extensively tested by prior work using manually crafted oracles that are designed for them.

Next, we show several representative bugs discovered by Argus in our new test oracles.

Example (1). Listing 6 presents a logic bug in PostgreSQL that was detected by Argus. The test oracle leverages the null-handling semantics of `RIGHT JOIN`, which implies that when a join condition is `false`, all columns from the left table must be `NULL`. Consequently, the first query should always yield a `NULL` row, which is identical to the second query's output, regardless of the value of the placeholder $\square_1$. However, due to a bug in its handling of `json` functions, PostgreSQL erroneously returns 2. This discovery highlights both the effectiveness of our novel, automatically generated test oracles and the LLM's ability to synthesize complex queries involving advanced SQL features, such as `json` functions. The finding is particularly noteworthy given that PostgreSQL is one of the world's most robust DBMSs. While several recent database testing studies have targeted PostgreSQL [58, 71, 81, 82], none reported finding new bugs, which attests to the effectiveness of our approach. After we reported the issue, developers confirmed and fixed it within 24 hours.

We also find that prior test oracles, such as TLP [48], struggle to detect this bug. For example, when we append a predicate to the first query, such as P = `sub.c` > 2, and then apply the three variants `WHERE` P, `WHERE NOT` P, and `WHERE` P `IS NOT NULL`, the results of the three partitioned queries remain identical. As a result, TLP does not report a bug in this case.

Moreover, a similar oracle uncovered a logic bug in MySQL (Listing 7), further demonstrating the versatility of the oracles generated by Argus. Specifically, the query in Listing 7 uses a `LEFT JOIN` combined with a `WHERE` clause to filter out rows where the right table's column is `NULL`, which should always yield an empty result set. However, due to a bug in MySQL's handling of the `MOD` function in this context, it incorrectly returns a row with value 1.

Example (2). Listing 8 shows a logic bug in Dolt, which was detected using an oracle generated by Argus that leverages the semantics of the `EXISTS` predicate and primary key constraints. Specifically,

```
CREATE TABLE t(c0 INT, c1 INT, PRIMARY KEY (c0, c1));
INSERT INTO t VALUES (1, 1);
INSERT INTO t VALUES (2, 2);
INSERT INTO t VALUES (2, 3);
SELECT * FROM □1 ▷ Table(c0:INT, c1:INT) ↦ t
WHERE EXISTS (
  SELECT 1 FROM t AS x WHERE x.c0 = t.c0 );
-- {(1,1), (2,2), (2,3), (1,1), (2,2), (2,3)} 🐞
SELECT * FROM □1 ▷ Table(c0:INT, c1:INT) ↦ t;
-- {(1,1), (2,2), (2,3)} ✔
```

Listing 8. **EXISTS** incorrectly duplicates rows in Dolt.

```
CREATE TABLE t0(c0 BOOLEAN);
CREATE TABLE t1(c0 INT);
INSERT INTO t0 VALUES (TRUE);
INSERT INTO t1 VALUES (0);
SELECT v.c0, t1.c0 FROM t0
CROSS JOIN LATERAL (
    □1 ▷ Table(c0:INT) ↦ (SELECT 1 AS c0) AS v
) JOIN t1 ON v.c0 > t1.c0; -- {} 🐞
SELECT v.c0, t1.c0 FROM t0
CROSS JOIN (
    □1 ▷ Table(c0:INT) ↦ (SELECT 1 AS c0) AS v
) JOIN t1 ON v.c0 > t1.c0; -- {(TRUE, 0)} ✔
```

Listing 9. Incorrect handling of **LATERAL** joins in Dolt.

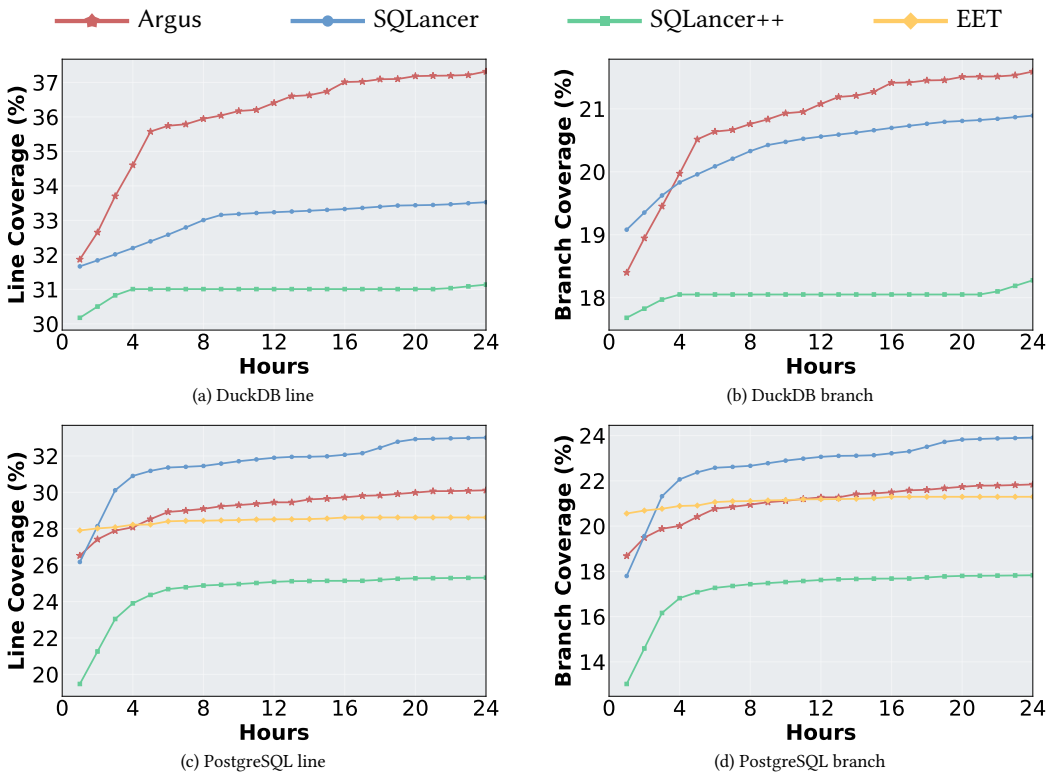
the EXISTS in the first query predicate should always return TRUE if the t.c0 is not nullable, which is guaranteed by the primary key constraint on (c0, c1). Thus, the first query should return all rows from table t, identical to the second query. However, due to a bug in Dolt's handling of the EXISTS predicate, it erroneously duplicates all rows in the output.

Example (3). Listing 9 shows another bug in Dolt with the handling of LATERAL joins in Dolt, which was detected using an oracle that leverages the semantics of CROSS JOIN and LATERAL joins. A LATERAL join allows the right-side subquery to reference columns from the left-side input; Dolt implements this feature as part of its SQL support. Specifically, if a CROSS JOIN LATERAL is used to join a left table with a constant right table, *i.e.,* SELECT1. , *e.g.,* a Table placeholder $\square_1$, the result should be identical to a regular CROSS JOIN with the same right table, *i.e.,* the first query is semantically equivalent to the second query. However, due to a bug in Dolt's handling of LATERAL joins, when instantiating the right table to (SELECT 1 AS c0) AS v, the first query incorrectly returns an empty result set instead of the expected row.

Example (4). The Listing 10 presents a DuckDB crash caused by a complex expression synthesized by Argus. The expression, generated by an LLM, contains a CTE named seq that produces a sequence of integers. The crash occurs when this CTE is used with an aggregation function, *i.e.,* SUM, inside an EXISTS predicate. Notably, synthesizing such a non-trivial expression is challenging for traditional grammar-based generators, which require both complex query structures and various SQL features. This case demonstrates that by leveraging the generative capabilities of LLMs for SQL snippets, Argus can effectively uncover not only logic bugs but also critical crashes.

```
CREATE TABLE t0(c0 INT);
CREATE TABLE t1(c0 BOOLEAN);
SELECT * FROM t0
WHERE EXISTS (
  SELECT 1 FROM t1
  WHERE □1 ▹ Expr(t0:INT) ↦
  (WITH seq(i) AS (VALUES (1)) SELECT sum(i) * t0.c0 FROM seq) IS NOT NULL
); -- (crash)
```

Listing 10. A crash bug in DuckDB triggered by a LLM-synthesized expression with **seq**.



(a) DuckDB line (b) DuckDB branch

(c) PostgreSQL line (d) PostgreSQL branch

Fig. 3. Code coverage achieved by Argus, SQLancer, and SQLancer++ on DuckDB and PostgreSQL over 24-hour runs.

7.2 Code Coverage

We compared Argus with three DBMS logic bugs finding tools, SQLancer [3, 47–49, 74], SQLancer++ [82] and EET [24] in multiple coverage metrics. We compared on DuckDB [46] and PostgreSQL [38]. As a state-of-the-art, open-source DBMS testing framework, SQLancer supports most of the latest test oracles [4, 57, 74]. SQLancer++ is a scalable variant that can be easily extended to multiple DBMSs with only lightweight modifications. EET [24] is a recent tool based on SQLsmith [53]'s generator and their carefully designed test oracles to produce complex queries.

In terms of test oracles, *code coverage* is an indicator for the number of features and execution plans that are tested by them. Although code coverage does not strongly correlate with the ability

Approach	Lines	Functions	Branches
SQLancer	3.256%	1.230%	1.313%
Argus	**17.820%**	**7.910%**	**7.315%**
	5.473×	**6.431×**	**5.571×**

Table 3. Average metamorphic coverage on DuckDB of 10 test suites for Argus and SQLancer.

to find logic bugs [82], it is still a fair metric to evaluate the diversity of test cases generated by Argus and the effectiveness of its corpus synthesis. Meanwhile, *metamorphic coverage* [2] is a more relevant metric for evaluating the effectiveness of test oracles and is highly related to logic bug finding abilities according to the historical bug study in SQLite and DuckDB [2]. Unlike traditional code coverage, which simply counts all executed code, metamorphic coverage measures the portions of the program exercised differently by the two executions in a pair of equivalent queries. Intuitively, because only these differential execution paths can lead to inconsistent results between semantically equivalent queries, metamorphic coverage provides a more precise measure of the code actually validated by the test oracle. Empirical studies on DBMSs like SQLite and DuckDB show that metamorphic coverage correlates strongly with real logic bugs and overlaps substantially with historical bug-fix locations.

Code coverage. Fig. 3 shows the line and branch code coverage for DuckDB and PostgreSQL over a 24-hour testing period starting from a clean build and empty database. In DuckDB, Argus achieves 19.9% and 18.1% higher line and branch coverage than SQLancer++, respectively, and 11.3% and 3.4% higher than SQLancer. Note that, we did not compare with EET on DuckDB, as it does not support this DBMS. In PostgreSQL, Argus slightly underperforms SQLancer overall but outperforms SQLancer++ (by 19.0% in line coverage and 22.5% in branch coverage) and EET (by 5.2% and 2.6%, respectively). This suboptimal performance is expected, as SQLancer has been extensively optimized specifically for PostgreSQL by the open-source community over many years. Specifically, SQLancer supports 22 types of DDL statements (e.g., `CREATE SEQUENCE`) and various DML statements beyond `SELECT` queries that PostgreSQL supports, whereas Argus focuses on detecting logic bugs across different DBMS implementations.

For a finer-grained comparison, we additionally evaluated Argus and SQLancer on PostgreSQL with respect to optimizer code coverage, since the optimizer is a core component that is closely tied to `SELECT` queries and widely studied in prior work [47, 60], along with the diversity of features exercised by `SELECT` queries that are covered by the test queries. The results show that Argus achieved 66.46% line coverage and 55.85% branch coverage of PostgreSQL's optimizer, compared to 62.33% and 52.17% by SQLancer. To evaluate feature diversity, we randomly sampled ten `SELECT` queries from each tool and counted their unique features using two third-party parsers: *pglast* [17] (counting number of unique PostgreSQL types) and *sqlparse* [87] (counting number of unique AST nodes). We found that Argus covered 23 features in pglast and 151 features in sqlparse, while SQLancer only covered 15 and 76, respectively. This demonstrates Argus's stronger ability to generate feature-rich queries. The sampled queries are provided in Appendix D.

Note that we did not compare with mutation-based testing tools, such as SQLRight [31] and SQuirrel [80], as they highly rely on the quality of seed queries and use the official unit tests provided by the DBMS as the initial input. This introduces bias in the comparison, as the official unit tests are usually well-designed and purposefully cover many code paths for the DBMSs that they are designed for. In fact, SQLRight achieved 51.3% line coverage on PostgreSQL in the first few minutes after feeding PostgreSQL's official unit test cases as seed queries, but quickly saturated after that without further improvement during the remainder of the testing period.

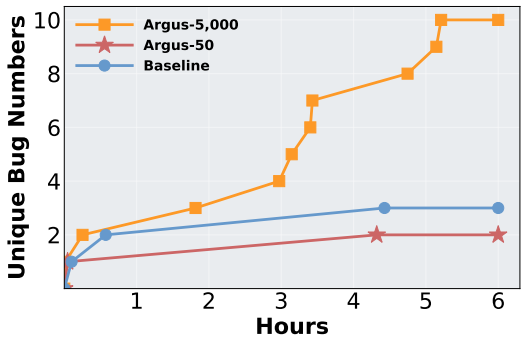


Fig. 4. Number of unique logic bugs found by different sets of oracles within a 6-hour testing on Dolt.

Overall, our results demonstrate that Argus can generate feature-rich, diverse, and complex test cases at scale, covering more optimizer paths than state-of-the-art techniques.

Metamorphic coverage. For metamorphic coverage, we compared Argus with SQLancer on DuckDB across a fixed number of test cases, following the same setting in [2]. We did not evaluate metamorphic coverage on PostgreSQL, which has not been supported by [2]'s implementation. Following the experimental setup in [2], we generated 10 test suites for SQLancer, each with 100 test cases. Specifically, SQLancer produce 50% of the test cases from TLP [48] and 50% from NoREC [47]. For Argus, we generated 10 CAQ pairs as test suites, and instantiated each pair with 100 different snippets, respectively.

We reported the average metamorphic coverage of the 10 test suites in Table 3. As shown, Argus achieves 5.473×, 6.431×, and 5.571× more line, function, and branch coverage than SQLancer, respectively. We believe this is because Argus's new test oracles can significantly alter the query structures between equivalent queries, thanks to LLM's creativity, which covers a broader range of different code paths than prior works. This significant improvement demonstrates the effectiveness of Argus's new test oracles in comparing different execution paths, and the potential to detect more logic bugs that state-of-the-art testers missed.

7.3 Effect of Test Oracles

We now evaluate Argus's new oracles by comparing the number of unique logic bugs they detected in Dolt v1.0.0. We selected this historical version to ensure a fair comparison, as `git-bisect` allows us to precisely link each bug to its corresponding fix commit, providing an accurate count of unique bugs per method.

We compared a composite baseline of test oracles against two sets of LLM-generated test oracles by Argus, with sizes of 50 and 5,000, *i.e.,* Argus-50 and Argus-5,000, respectively. The baseline combines 11 test oracles from four previous works: TLP [48], NoREC [47], EET [24], and DQP [5]. To minimize confounding factors, we standardized the experimental conditions. Specifically, the baseline oracles were represented in the same CAQ pair format and instantiated using the same snippet corpus as Argus. Furthermore, since the CAQ format requires an associated schema, we also randomly generated a new schema for the baseline oracles before each instantiation. The details about how to convert the baseline oracles to CAQ pairs are provided in Appendix C.

Fig. 4 shows the number of unique logic bugs found by different sets of oracles within a 6-hour testing. We select a 6-hour time window for a fair comparison, following common practice in fuzz testing [26] and prior studies on DBMS logic bug detection [81], which typically fix the testing duration within a range of 1 – 24 hours. As shown, Argus-5,000 found 10 unique bugs, significantly outperforming the baseline, which found only 3. In contrast, Argus-50 found just 2 bugs. This

```
SELECT * FROM v WHERE TRUE;
SELECT * FROM v WHERE v.c >= v.c OR v.c < v.c
```

Listing 11. A false positive example from LLM-as-a-judge.

underperformance was expected, as the baseline oracles were carefully designed by human experts. These results demonstrate the importance of the number of oracles in detecting unique logic bugs and underscore the limitations of manual oracle design, thus highlighting the necessity of Argus's automated test oracle discovery.

7.4 Effect of SQL Equivalence Prover

We evaluate our SQL equivalence prover's false positive and false-negative rates against an LLM-as-a-judge baseline. For this baseline, we use GPT-5 to determine whether a candidate CAQ pair is equivalent directly.

False positive bug reports. To quantify the impact of the prover on false positives, we performed an ablation that disables the prover while using an LLM-as-a-judge approach [78], and ran Argus on DuckDB until we collected 20 bug reports. We then manually adjudicated each report as a true positive or a false positive. Surprisingly, we found that **all** of them were false positives. For instance, LLM-as-a-judge incorrectly decides the two queries shown in Listing 11 as equivalent, which is actually inequivalent when $v.c$ is `NULL`.

To further investigate the issue of extremely high false positive rates, we generated 20 candidate CAQ pairs that the LLM judged as equivalent, and manually verified them. We found that 1 of these 20 pairs was an incorrect test oracle, *i.e.,* it consists of inequivalent CAQ pairs. While the accuracy of LLM-as-a-judge is high, bugs in mature DBMSs are exceedingly rare, as it often takes thousands of queries to discover a single bug. Consequently, even a low false positive rate can generate a volume of false reports that overwhelmingly drown out true positives. In addition, such false positives are particularly unacceptable when Argus is integrated into CI/CD pipelines or production environments, as developers have limited time to investigate each report. Therefore, the SQL equivalence prover is necessary to ensure the soundness of Argus's test oracles and make it practical for real-world use.

In terms of the performance with the SQL equivalence prover, we sample 20 CAQ pairs that are accepted by the prover as equivalent, as well as 20 pairs that are instantiated from them. After our manual validation, we find that all 20 CAQ pairs and concrete instances derived from them are indeed equivalent. This demonstrates the prover's effectiveness in identifying and validating equivalent queries and reducing false positives.

False negatives in test-oracle generation. To estimate the prover's false-negative rate, *i.e.,* its inability to prove correct test oracles, we ran two configurations of Argus on DuckDB: one using the prover to check CAQ pairs, and one using LLM-as-a-judge. In each configuration, we continued testing until the tool had rejected 10 candidate CAQ pairs as "inequivalent," and then we manually adjudicated each pair. We found that 8 of the 10 pairs were false negatives for the prover, while 5 of the 10 pairs were false negatives for the LLM-as-a-judge. The results also show that the prover's false negative rate is comparable to that of state-of-the-art LLMs. Note that, false negatives from the prover are expected in general, as it is intentionally conservative to ensure soundness.

7.5 Cost and Efficiency Analysis

We next evaluate CAQ's impact on cost and efficiency by comparing Argus against the naive baseline introduced in Sec. 1. This baseline uses the same SQL equivalence prover but directly prompts an LLM to generate entire pairs of equivalent queries from the seed queries rather than

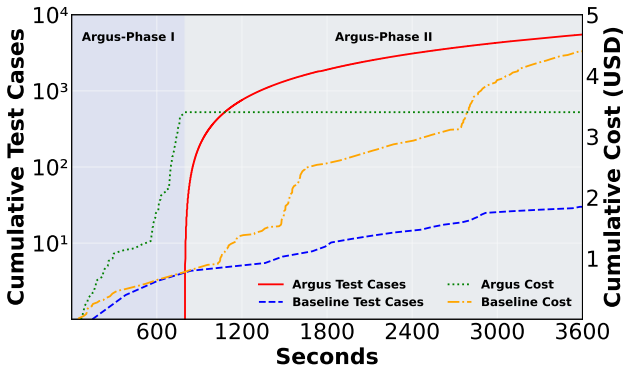


Fig. 5. Cost and efficiency comparison between Argus and the naive baseline on Dolt over a 1-hour run.

Method	Avg. Similarity	# of Valid Queries
Clustering	**0.617**	**86**
Beam-search	0.766	48
Naive	0.688	81

Table 4. Comparison of Argus's equivalent CAQ generation methods with baselines. Lower similarity indicates higher diversity.

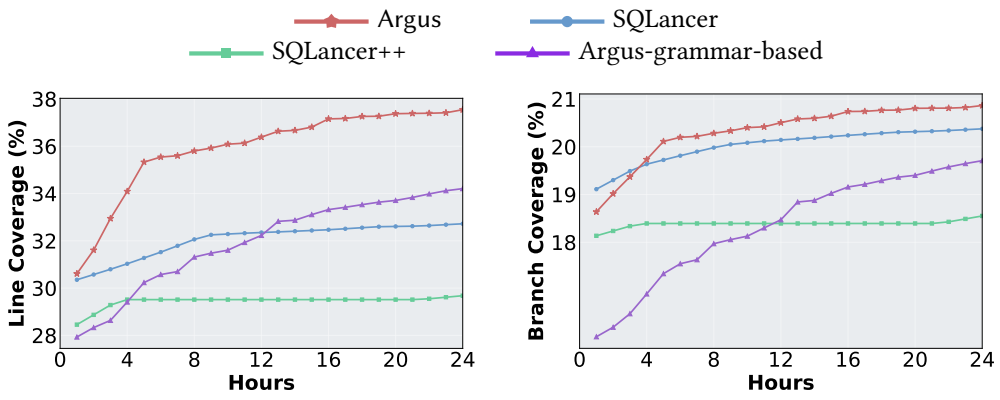
CAQ pairs. After testing on Dolt 1.0.0 for one hour, Argus detected a logic bug while the baseline found none, and we measured the corresponding test cases generated and LLM costs.

As shown in Fig. 5, Argus achieves a significantly higher test case throughput than the naive baseline. Unlike the baseline, which slowly generates complete equivalent pairs, Argus spends only a few minutes creating abstract CAQ pairs (Phase I) and then rapidly instantiates them into numerous concrete test queries (Phase II). This process is also highly economical: generating the CAQ pairs costs about $3 in LLM calls, and instantiating test cases costs roughly $1 per 1,000 instantiated test cases, and leveraging a reusable corpus of 100,000 snippets built for only $12. This shows that Argus is a cost-effective solution for detecting logic bugs with a high throughput.

7.6 Component-wise Analysis

We analyze the contributions of two Argus's key components: the diversity-oriented equivalent query generation (Sec. 5.2) and the LLM-powered SQL snippets generation (Sec. 6.1).

Diversity-oriented equivalent CAQ generation. To evaluate the impact of diversity-oriented equivalent query generation, we compare Argus 's CAQ generation method against two baselines: (i) a naive approach that independently prompts an LLM to generate equivalent queries, and (ii) a beam-search–based method that produces multiple candidates at each step and selects a diverse subset, according to a diversity metric, to condition the next iteration. For all methods, we give the same seed CAQs, fix the LLM budget to 1,000 generated CAQ candidates and evaluate: (1) the number of valid queries (*i.e.,* those that pass the SQL equivalence checker); and (2) the similarity of the generated queries, measured by the average tree-edit distance between DuckDB query plans across all pairs of equivalent queries. Specifically, given two query plan tree T_1 and T_2, let $\mathrm{dist}(T_1, T_2)$ be the tree-edit distance between them. The similarity score is computed as: $\frac{2}{N(N-1)} \sum_{i \neq j} \left(1 - \frac{\mathrm{dist}(T_i, T_j)}{|T_i| + |T_j|}\right)$, where N is the number of valid query pairs. As shown in Table 4, Argus 's CAQ generation method outperforms both baselines by producing more valid CAQs while maintaining the lowest similarity.



(a) DuckDB line (b) DuckDB branch

Fig. 6. Ablation study of code coverage achieved by Argus and a variant using only SQLancer's grammar-based generator on DuckDB over 24-hour runs.

```
1 CREATE TABLE t1(c0 BOOLEAN, c1 BOOLEAN);
2 SELECT false FROM t1;
3 SELECT (t1.c1 < t1.c0) < (t1.c1 = t1.c1) FROM t1;
```

Listing 12. Incorrect equivalence proof in SQLSolver [13].

LLM-powered SQL snippet generation. We evaluate the effectiveness of LLM-powered SQL snippet generation by comparing Argus's method against a variant that only uses SQLancer's grammar-based generator to build the snippet corpus during the CAQ instantiation phase. We use the code coverage metric in Sec. 7.2 to evaluate both methods after a 24-hour testing on DuckDB. As shown in Fig. 6, Argus consistently outperforms the grammar-based variant in both line and branch coverage. Without LLM-generated snippets, the grammar-based variant can still outperform baselines in terms of line coverage but struggles with branch coverage.

8 Discussion

Additional finding. In addition to improving DBMS robustness, our testing also revealed several bugs in existing SQL equivalence provers. These bugs could lead to inherent false positives that, if unaddressed, would hinder large-scale testing. For example, an early version of SQLSolver [13] incorrectly proved the two queries in Listing 12 to be equivalent: the first query always returns `false` when `t1` is non-empty, whereas the second query can return `true` or even `NULL` depending on the values of `c0` and `c1`. Such issues are common among research prototypes that have been validated only on small benchmarks — for instance, the widely used Calcite test cases [7] include only about 200 equivalent query pairs. We reported 10 bugs in SQLSolver and QED [63] during early development of Argus, all of which were quickly fixed by the developers. After these fixes, our testing proceeded without being affected by those prover bugs. These findings demonstrate that our approach not only helps uncover DBMS bugs but also contributes to improving the reliability of SQL equivalence provers. Importantly, the soundness of our approach is independent of the correctness of specific prover implementations—the latter only affects practical false positives before fixing.

Customized testing. Argus is highly configurable and supports a broad range of testing scenarios. By adjusting the LLM generation prompt, users can focus on specific SQL features without modifying

the underlying code, enabling flexible and comprehensive testing. For example, to generate table snippets involving `OUTER JOIN`s, the prompt can explicitly request: "Ensure that the generated snippet includes at least one `OUTER JOIN` (*e.g.,* `LEFT JOIN`, `RIGHT JOIN`, `FULL OUTER JOIN`, ...)".

Query classes supported by SQL equivalence provers. SQL equivalence provers is a key component of Argus, but typically only support a limited set of SQL features due to the inherent complexity of SQL semantics. To the best of our knowledge, existing SQL equivalence provers support the core syntax implemented in Calcite [7], including outer joins, nested queries, and simple aggregations like `SUM` and `COUNT`. However, they generally do not understand the semantics of most DBMS-specific or user-defined functions, treating them as uninterpreted functions, which may result in omitting some potential test oracles. For example, if a DBMS defines a function $\texttt{add}(x, y) := x + y$, the prover cannot determine whether `SELECT` $\texttt{add}(x, y)$ `FROM` t is equivalent to `SELECT` $x + y$ `FROM` t. Another limitation stems from the fact that SQL equivalence provers heavily rely on SMT solvers, whose complexities can grow rapidly with the size of the CAQ pairs analyzed. This may limit the query sizes of the test oracles that Argus can generate. Nevertheless, these restrictions from SQL equivalence provers can be mitigated in Argus's test cases instantiation phase, which can generate complex SQL queries by filling in CAQ placeholders with diverse SQL snippets. Those snippets can include SQL features that currently lack support in existing provers, as well as increase the overall complexity of the generated test cases.

Limitations. Our approach has two limitations that present opportunities for future work.

(1) Argus currently focuses on relational DBMS and `SELECT` queries like prior works [3, 5, 24, 47–49, 74], but the underlying principles can be extended to other types of databases and query languages, such as graph DBMS [23, 34, 37, 86] and spatial DBMS testing [12].
(2) Argus focuses on automatically discovering test oracles, but does not provide a mechanism to prioritize or rank the generated oracles. We believe that, after Argus, DBMS testing research can shift from manually crafting test oracles to developing techniques for prioritizing and selecting the most effective oracles from a large pool of LLM-generated candidates.

9 Related Work

LLM in systems research. LLMs have been applied to various system research tasks, including code generation [20, 27, 39, 44, 66], automated tuning [18, 76], data processing [32, 33, 43, 54, 73], program analysis [28], networking [77], and algorithm discovery [41, 50, 72].

(1) In LLM-aided testing, prior work has primarily focused on generating effective inputs. For example, Fuzz4all [65] is a universal fuzzing framework that leverages LLMs to generate test inputs for various applications, while other research applies LLMs to fuzzing in specific domains such as compilers [67], kernels [68], and smart contracts [55]. Similarly, ShQvel [81] uses LLMs to generate feature-rich SQL queries for testing DBMS with manually crafted test oracles, while SQLStorm [52] employs LLMs to synthesize a large-scale benchmark for evaluating performance of DBMSs. While prior approaches generate complex test inputs to find crash bugs, our work focuses on generating test oracles to detect logic bugs, along with effective methods for test oracle initializations, which are orthogonal and complementary to existing efforts.
(2) In LLM-for-DBMS, prior work mainly focus on text-to-SQL [21], query optimization [29, 59], dialect translation [84], and hyperparameter tuning [18, 85]. Among these works, LLM-R2 [29] provides a framework to guide LLMs' use of existing rules, *e.g.,* Calcite, to optimize SQL queries. Though this method can also be applied to our task, it is impractical because of the high cost of rewriting concrete queries for testing and the difficulty of finding bugs by DBMS's query rewriting rules, which are typically simple and well-tested. To the best of our knowledge, our

work is the first to leverage LLMs to generate test oracles for DBMSs, and we believe that our rule generation framework can also be applied to other DBMS tasks in the future, such as query optimization with performance improvement.

DBMS testing and verification. There is a rich body of work on DBMS testing, verification, and their applications. Prior work in DBMS testing has largely followed two distinct research paths: *test query generation* for fuzzing and *test oracle design* for detecting logic bugs. Test query generation focuses on automatically generating complex SQL queries to find crashes, with techniques being either grammar-based, exemplified by tools like SQLsmith [53] and SQLancer [49], or mutation-based, as seen in Griffin [16]. Grammar-based techniques typically use predefined grammars to construct valid SQL queries, while mutation-based approaches modify existing queries to create new test cases. Beyond the generation of test queries, recent research has also explored constructing meaningful database states [1, 8, 9, 51, 69]. In parallel, the second path concentrates on test oracles for detecting "silent" bugs such as logic and performance issues, an area that has traditionally relied on significant manual effort. For instance, oracles like TLP [48], NoREC [47], and EET [24] were designed for logic bugs, while others such as CERT [4] and Apollo [25] target performance issues. This manual crafting of oracles remains a primary bottleneck in fully automated DBMS testing. Argus is the first work that tackles this critical challenge by automatically discovering test oracles.

Recent work has focused on improving the effectiveness of SQL equivalence provers [10, 11, 13, 63, 83] and disprovers [19, 79]. These tools have been widely applied in various DBMS scenarios, including query rewriting [64], text-to-SQL [70], and user-database interaction [45]. Among these, Wetune [64] is the most relevant to our work, as it employs an SQL equivalence prover to verify query rewriting rules generated via a brute-force search. However, its enumeration-based approach suffers from two key limitations. First, it can only discover simple rules due to the high time complexity of enumeration. Second, because it lacks a placeholder mechanism like CAQ, the resulting rules support a very limited set of SQL features. For example, the average AST node count of SQL queries generated by Wetune's rules is 99.6, whereas in Argus test cases it reaches 777.8. This indicates that Argus targets much richer and complex SQL structures than enumeration-based tools. These restrictions render the approach impractical for testing modern DBMSs. In contrast, our LLM-based approach overcomes these shortcomings by generating complex rules and instantiating them with a diverse range of SQL snippets.

10 Conclusion

The manual construction of test oracles is a major bottleneck in finding logic bugs in modern DBMSs. We present Argus, the first automated framework that uses LLMs to generate sound and effective test oracles. Argus achieves scalability through its novel CAQ abstraction and guarantees soundness with an SQL equivalence prover. On five extensively-tested DBMSs, Argus discovered 41 previously unknown bugs, including 36 logic bugs. Compared to prior work, Argus's LLM-generated test oracles demonstrate improved metamorphic coverage and a stronger logic-bug finding abilities. Argus's approach not only provides a solution to the long-standing bottleneck of fully automated DBMS testing but also opens new opportunities for testing other complex software with LLMs.

11 Ackowledgements

We thank Jinsheng Ba and Manuel Rigger for their insightful feedback, Haoran Ding, Sicheng Pan, and Shuxian Wang for their support on SQL equivalence solvers, and the DBMS developers for confirming and fixing the reported bugs. This work is supported by NSF grants IIS-1955488, IIS-2027575, DOE awards DE-SC0016260, AC02-05CH11231, and DARPA Agreement No. HR00112590131.

References

[1] Arvind Arasu, Raghav Kaushik, and Jian Li. 2011. Data generation using declarative constraints. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 685–696.

[2] Jinsheng Ba, Yuancheng Jiang, and Manuel Rigger. 2025. Metamorphic Coverage. *arXiv preprint arXiv:2508.16307* (2025).

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

[4] 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*. 1–13.

[5] Jinsheng Ba and Manuel Rigger. 2024. Keep it simple: Testing databases via differential query plans. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.

[6] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2014), 507–525.

[7] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. 2018. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*. 221–230.

[8] Carsten Binnig, Donald Kossmann, Eric Lo, and M Tamer Özsu. 2007. QAGen: generating query-aware test databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 341–352.

[9] Chunyu Chen, Zhengjie Miao, Yong Zhang, and Jiannan Wang. 2025. ParSEval: Plan-aware Test Database Generation for SQL Equivalence Evaluation. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4750–4762.

[10] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL.. In *CIDR*. 1–7.

[11] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: Proving query rewrites with univalent SQL semantics. *Acm sigplan notices* 52, 6 (2017), 510–524.

[12] Wenjing Deng, Qiuyang Mang, Chengyu Zhang, and Manuel Rigger. 2024. Finding logic bugs in spatial database engines via affine equivalent inputs. *Proceedings of the ACM on Management of Data* 2, 6 (2024), 1–26.

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

[14] Dolt. 2025. Dolt Homepage. [EB/OL]. https://www.dolthub.com/

[15] 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

[16] 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

[17] Lele Gaifax and Contributors. 2025. pglast: A PostgreSQL AST and statements prettifier for Python. https://github.com/lelit/pglast. Accessed: 2025-10-15.

[18] Victor Giannakouris and Immanuel Trummer. 2025. λ-tune: Harnessing large language models for automated database system tuning. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–26.

[19] Yang He, Pinhan Zhao, Xinyu Wang, and Yuepeng Wang. 2024. VeriEQL: Bounded equivalence verification for complex SQL queries with integrity constraints. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 1071–1099.

[20] Charles Hong, Sahil Bhatia, Alvin Cheung, and Yakun Sophia Shao. 2025. Autocomp: LLM-Driven Code Optimization for Tensor Accelerators. *arXiv preprint arXiv:2505.18574* (2025).

[21] Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2024. Next-generation database interfaces: A survey of llm-based text-to-sql. *arXiv preprint arXiv:2406.08426* (2024).

[22] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[23] Yuancheng Jiang, Jiahao Liu, Jinsheng Ba, Roland HC Yap, Zhenkai Liang, and Manuel Rigger. 2024. Detecting logic bugs in graph database management systems via injective and surjective graph query transformation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[24] 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)*. 821–835.

[25] 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.

[26] 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

[27] Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E Gonzalez, and Ion Stoica. 2025. S*: Test time scaling for code generation. *arXiv preprint arXiv:2502.14382* (2025).

[28] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 474–499.

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

[30] Jie Liang, Yaoguang Chen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Yu Jiang, Xiangdong Huang, Ting Chen, Jiashui Wang, and Jiajia Li. 2023. Sequence-Oriented DBMS Fuzzing. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 668–681. https://doi.org/10.1109/ICDE55515.2023.00057

[31] 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

[32] Yiming Lin, Madelon Hulsebos, Ruiying Ma, Shreya Shankar, Sepanta Zeigham, Aditya G Parameswaran, and Eugene Wu. 2024. Towards accurate and efficient document analytics with large language models. *arXiv preprint arXiv:2405.04674* (2024).

[33] Yiming Lin, Madelon Hulsebos, Ruiying Ma, Shreya Shankar, Sepanta Zeighami, Aditya G Parameswaran, and Eugene Wu. 2025. Querying templatized document collections with large language models. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE, 2422–2435.

[34] Shuang Liu, Junhao Lan, Xiaoning Du, Jiyuan Li, Wei Lu, Jiajun Jiang, and Xiaoyong Du. 2024. Testing graph database systems with graph-state persistence oracle. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 666–677.

[35] Xiaoxuan Liu, Shuxian Wang, Mengzhu Sun, Sicheng Pan, Ge Li, Siddharth Jha, Cong Yan, Junwen Yang, Shan Lu, and Alvin Cheung. 2022. Leveraging application data constraints to optimize database-backed web applications. *arXiv preprint arXiv:2205.02954* (2022).

[36] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. 2022. Automatic detection of performance bugs in database systems using equivalent queries. In *Proceedings of the 44th International Conference on Software Engineering*. 225–236.

[37] Qiuyang Mang, Jinsheng Ba, Pinjia He, and Manuel Rigger. 2025. Finding Logic Bugs in Graph-processing Systems via Graph-cutting. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–27.

[38] Bruce Momjian. 2001. *PostgreSQL: introduction and concepts*. Vol. 192. Addison-Wesley New York.

[39] Niels Mündler, Jingxuan He, Hao Wang, Koushik Sen, Dawn Song, and Martin Vechev. 2025. Type-Constrained Code Generation with Language Models. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 601–626.

[40] MySQL. 2025. MySQL Homepage. [EB/OL]. https://www.mysql.com

[41] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131* (2025).

[42] OpenAI. 2025. Introducing OpenAI o3 and o4-mini. https://openai.com/index/introducing-o3-and-o4-mini/.

[43] Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. 2024. Semantic Operators: A Declarative Model for Rich, AI-based Data Processing. *arXiv preprint arXiv:2407.11418* (2024).

[44] Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. [n. d.]. The Berkeley Function Calling Leaderboard (BFCL): From Tool Use to Agentic Evaluation of Large Language Models. In *Forty-second International Conference on Machine Learning*.

[45] Joanna Purich, Anthony Wise, and Leilani Battle. 2025. An adaptive benchmark for modeling user exploration of large datasets. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–24.

[46] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 international conference on management of data*. 1981–1984.

[47] 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*. 1140–1152.

[48] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[49] 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.

[50] Dmitry Rybin, Yushun Zhang, and Zhi-Quan Luo. 2025. XX^T Can Be Faster. *arXiv preprint arXiv:2505.09814* (2025).

[51] Anupam Sanghi and Jayant R Haritsa. 2023. Synthetic data generation for enterprise dbms. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 3585–3588.

[52] Tobias Schmidt, Viktor Leis, Peter Boncz, and Thomas Neumann. 2025. SQLStorm: Taking Database Benchmarking into the LLM Era. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4144–4157.

[53] Andreas Seltenreich. 2022. Sqlsmith. https://github.com/anse1/sqlsmith.

[54] Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G Parameswaran, and Eugene Wu. 2024. Docetl: Agentic query rewriting and evaluation for complex document processing. *arXiv preprint arXiv:2410.12189* (2024).

[55] Chaofan Shou, Jing Liu, Doudou Lu, and Koushik Sen. 2024. Llm4fuzz: Guided fuzzing of smart contracts with large language models. *arXiv preprint arXiv:2401.11108* (2024).

[56] Donald R Slutz. 1998. Massive stochastic testing of SQL. In *VLDB*, Vol. 98. 618–622.

[57] 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 (April 2024), 1884–1897. https://doi.org/10.14778/3659437.3659445

[58] 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. *Proceedings of the VLDB Endowment (VLDB)* (2025).

[59] Jie Tan, Kangfei Zhao, Rui Li, Jeffrey Xu Yu, Chengzhi Piao, Hong Cheng, Helen Meng, Deli Zhao, and Yu Rong. 2025. Can Large Language Models Be Query Optimizer for Relational Databases? *arXiv preprint arXiv:2502.05562* (2025).

[60] Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. 2023. Detecting logic bugs of join optimizations in dbms. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.

[61] Xiu Tang, Shijie Yang, Sai Wu, Dongxiang Zhang, Wenchao Zhou, Feifei Li, and Gang Chen. 2025. Unveiling Logic Bugs in SPJG Query Optimizations within DBMS. *ACM Transactions on Database Systems* (2025).

[62] Boris A Trakhtenbrot. 1950. Impossibility of an algorithm for the decision problem for finite classes. In *Doklady Akademiia Nauk SSSR*, Vol. 70. 569.

[63] Shuxian Wang, Sicheng Pan, and Alvin Cheung. 2024. QED: A powerful query equivalence decider for SQL. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3602–3614.

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

[65] 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*. 1–13.

[66] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 819–831.

[67] 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. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 709–735.

[68] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2025. Kernelgpt: Enhanced kernel fuzzing via large language models. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 560–573.

[69] Jingyi Yang, Peizhi Wu, Gao Cong, Tieying Zhang, and Xiao He. 2022. SAM: Database generation from query workloads with supervised autoregressive models. In *Proceedings of the 2022 International Conference on Management of Data*. 1542–1555.

[70] 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. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–28.

[71] Shiyang Ye, Chao Ni, Jue Wang, Qianqian Pang, Xinrui Li, and Xiaodan Xu. 2025. Sembug: Detecting Logic Bugs in Dbms Through Generating Semantic-Aware Non-Optimizing Query. In *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*. IEEE, 124–135.

[72] Cunxi Yu, Rongjian Liang, Chia-Tung Ho, and Haoxing Ren. 2025. Autonomous Code Evolution Meets NP-Completeness. *arXiv preprint arXiv:2509.07367* (2025).

[73] Sepanta Zeighami, Shreya Shankar, and Aditya Parameswaran. 2025. Cut Costs, Not Accuracy: LLM-Powered Data Processing with Guarantees. *arXiv preprint arXiv:2509.02896* (2025).

[74] Chi Zhang and Manuel Rigger. 2025. Constant Optimization Driven Database System Testing. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–24.

[75] Kaizhong Zhang and Dennis Shasha. 1989. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing* 18, 6 (1989), 1245–1262.
[76] Michael R Zhang, Nishkrit Desai, Juhan Bae, Jonathan Lorraine, and Jimmy Ba. 2023. Using large language models for hyperparameter optimization. *arXiv preprint arXiv:2312.04528* (2023).
[77] Qizheng Zhang, Ali Imran, Enkeleda Bardhi, Tushar Swamy, Nathan Zhang, Muhammad Shahbaz, and Kunle Olukotun. 2024. Caravan: Practical online learning of In-Network ML models with labeling agents. In *Proceedings of the 3rd Workshop on Practical Adoption Challenges of ML for Systems*. 17–20.
[78] Fuheng Zhao, Lawrence Lim, Ishtiyaque Ahmad, Divyakant Agrawal, and Amr El Abbadi. 2023. Llm-sql-solver: Can llms determine SQL equivalence? *arXiv preprint arXiv:2312.10321* (2023).
[79] Pinhan Zhao, Yuepeng Wang, and Xinyu Wang. 2025. Polygon: Symbolic Reasoning for SQL using Conflict-Driven Under-Approximation Search. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 1315–1340.
[80] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 955–970. https://doi.org/10.1145/3372297.3417260
[81] Suyang Zhong and Manuel Rigger. 2025. Testing Database Systems with Large Language Model Synthesized Fragments. *arXiv preprint arXiv:2505.02012* (2025).
[82] 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
[83] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Jinpeng Wu. 2020. SPES: A two-stage query equivalence verifier. *arXiv preprint arXiv:2004.00481* (2020).
[84] Wei Zhou, Yuyang Gao, Xuanhe Zhou, and Guoliang Li. 2025. CrackSQL: A Hybrid SQL Dialect Translation System Powered by Large Language Models. *arXiv preprint arXiv:2504.00882* (2025).
[85] Xuanhe Zhou, Zhaoyan Sun, and Guoliang Li. 2024. Db-gpt: Large language model meets database. *Data Science and Engineering* 9, 1 (2024), 102–111.
[86] Zeyang Zhuang, Penghui Li, Pingchuan Ma, Wei Meng, and Shuai Wang. 2023. Testing graph database systems via graph-aware metamorphic relations. *Proceedings of the VLDB Endowment* 17, 4 (2023), 836–848.
[87] Ándré Albrecht and Contributors. 2025. sqlparse: A non-validating SQL parser module for Python. https://github.com/andialbrecht/sqlparse. Accessed: 2025-10-15.

Received October 2025; revised January 2026; accepted February 2026