Testing Database Engines via Query Plan Guidance

Jinsheng Ba
National University of Singapore

Manuel Rigger
National University of Singapore

***Abstract*—Database systems are widely used to store and query data. Test oracles have been proposed to find logic bugs in such systems, that is, bugs that cause the database system to compute an incorrect result. To realize a fully automated testing approach, such test oracles are paired with a test case generation technique; a test case refers to a database state and a query on which the test oracle can be applied. In this work, we propose the concept of *Query Plan Guidance (QPG)* for guiding automated testing towards "interesting" test cases. SQL and other query languages are declarative. Thus, to execute a query, the database system translates every operator in the source language to one of the potentially many so-called physical operators that can be executed; the tree of physical operators is referred to as the query plan. Our intuition is that by steering testing towards exploring a variety of unique query plans, we also explore more interesting behaviors—some of which are potentially incorrect. To this end, we propose a mutation technique that gradually applies promising mutations to the database state, causing the DBMS to create potentially unseen query plans for subsequent queries. We applied our method to three mature, widely-used, and extensively-tested database systems—SQLite, TiDB, and CockroachDB—and found 53 unique, previously unknown bugs. Our method exercises 4.85–408.48× more unique query plans than a naive random generation method and 7.46× more than a code coverage guidance method. Since most database systems—including commercial ones—expose query plans to the user, we consider QPG a generally applicable, black-box approach and believe that the core idea could also be applied in other contexts (e.g., to measure the quality of a test suite).**

***Index Terms*—automated testing, test case generation**

I. Introduction

Database Management Systems (DBMSs) are fundamental software systems used to store, retrieve, and run queries on data. They are used in almost every computing device [1]–[3], thus any bug has a potentially severe consequence. Logic bugs, which refer to incorrect results returned by DBMSs, are a particularly challenging category of bugs to find as they silently compute an incorrect result—unlike, for example, crash bugs [4], [5], which cause the process to be terminated. Consider Listing 1, where the `SELECT` statement triggers a logic bug that causes the returned result to unexpectedly contain a record, while it should be empty. Finding such bugs requires a so-called test oracle, which validates the DBMS' result. Recently, effective test oracles [6]–[8] have been proposed that brought validating the results of such queries within reach.

Besides a test oracle, automatically finding logic bugs requires a test case generation method. For finding logic bugs in DBMSs, a test case refers to a database state and a query on which the test oracle can be applied. Test case generation techniques face two main challenges. First, "interesting" test cases should be generated that stress various parts of the DBMS to increase the chance of finding bugs in them. No clear definition or metric on what an interesting test case constitutes exists, as it is unknown in advance by which logic bugs a DBMS is affected. Second, the test cases should be valid both syntactically and semantically while also corresponding to the structure imposed by the test oracle; for example, the NoREC oracle requires a query with a `WHERE` clause, but no more complex clauses (*e.g.*, `HAVING` clauses) [7] while also forbidding various functions and keywords from being used (*e.g.*, aggregate functions).

Both generation-based and mutation-based approaches have been proposed to be paired with the above test oracles [6]–[8]. SQLancer uses a generation-based approach in which test cases are generated adhering to the grammar of the respective SQL dialects as well as the constraints imposed by the test oracles. Overall, this approach makes it likely to generate valid test cases; we observed that about 90% of the queries generated by SQLancer for SQLite are valid. However, the test case generation approach receives no guidance that could steer it towards producing interesting test cases. Recently, SQLRight [9] was proposed to address this shortcoming. SQLRight mutates test cases aiming to maximize the DBMS' covered code, thus building on the success of grey-box fuzzing [10], [11]. While SQLRight improved on SQLancer's test case generation in various metrics, code coverage alone was shown to be an imperfect proxy metric for DBMSs [12] and stateful systems in general [13], as it cannot precisely model the state of databases. Despite using mutation operators that aim to maximize the validity of queries, SQLRight achieves a lower rate of valid queries of 40% [9]. Other test case generation approaches have been proposed that aim at finding crash bugs and thus disregard the test oracle's constraints, which is why we do not further consider them. These include mutation-based approaches such as Squirrel [5] or DynSQL [14], and generation-based ones such as SQLsmith [15] or RAGS [16].

In this paper, we propose *Query Plan Guidance* (*QPG*), a technique that utilizes query plans to guide the test-case generation process towards interesting test cases. A query plan is a tree of operations that describes how an SQL statement is executed by a DBMS. It is readily provided by DBMSs—users can typically obtain a textual representation using an `EXPLAIN` SQL statement—and is typically inspected by DBMS users for tuning the performance of queries. Our insight is that a query plan provides a compact and high-level summary of how a query is executed, therefore, covering more unique query plans increases the likelihood of finding logic

Listing 1. A bug found by *QPG* in SQLite due to an incorrect use of an index in combination with a JOIN. Given the same SELECT, the left query plan is produced if no index is present, while the right one uses the index.

```
1  CREATE TABLE t1(a INT, b INT);
2  INSERT INTO t1(a) VALUES(2);
3  CREATE TABLE t2(c INT);
4  CREATE TABLE t3(d INT);
5  INSERT INTO t3 VALUES(1);
6  CREATE INDEX i0 ON t2(c) WHERE c=3;
7
8  SELECT * FROM t2 RIGHT JOIN t3 ON d<>0 LEFT JOIN
       t1 ON c=3 WHERE t1.a<>0; --  {} ✔ {|1|2|} 🐞
9  ----------------------------------------------
10 QUERY PLAN
11 WITHOUT INDEX i0:            WITH INDEX i0:
12 |--SCAN t2                   |--SCAN t2 USING
13                                   COVERING INDEX i0
14 |--SCAN t3                   |--SCAN t3
15 |--SCAN t1                   |--SCAN t1
16 `--RIGHT-JOIN t3             `--RIGHT-JOIN t3
17    `--SCAN t3                   `--SCAN t3
```

bugs. Consider Listing 1, which illustrates two scenarios of executing test cases with SQLite. In the first scenario, the `CREATE INDEX` statement highlighted in red is omitted, causing the `SELECT` statement to return an empty result. This result is expected, since column `c` in table `t2` has no data and the join condition `c=3` is false. In the second scenario, the `CREATE INDEX` statement is executed, which causes SQLite to unexpectedly fetch the row $\{|1|2|\}$. An index is an auxiliary data structure used by queries [17], which should not have any semantic effect. While in both scenarios, the same query is executed, the query plans shown below the test cases differ due to the two different database states. The left query plan for the correct execution indicates that the records from table `t2` are read sequentially (`SCAN t2`). In contrast, the right query plan indicates that the DBMS used the index to read the data (`SCAN t2 USING COVERING INDEX i0`), which was incorrect. Besides indexes, various other factors can influence query plans (*e.g.*, data characteristics).

To generate valid queries that correspond to the oracles' constraints, we propose mutating the database state rather than the queries. Specifically, we re-use the existing random grammar-based generation approach of SQLancer [6] to generate the queries. However, we record all seen query plans for a given database state and mutate this state when no new query plans are observed, indicating that the current database state's potential for enabling unobserved query plans has been saturated. We modeled the decision-making process for selecting the most promising mutation—an SQL statement that modifies the database state—as a multi-armed bandit problem and assigned a high priority to the SQL statement that results in the most new query plans across all executions. The multi-armed bandit problem is a model in which a fixed limited set of resources have to be allocated between competing choices in a way that maximizes the expected gain [18].

We implemented *QPG* in *SQLancer* and evaluated it on SQLite, TiDB, and CockroachDB. We found 53 unique, previously unknown bugs, all of which have been acknowledged by the developers. Of these, 35 have already been fixed. Three bugs in SQLite had been hidden for more than six years before we found them, despite the extensive existing testing efforts by the authors of SQLancer and SQLRight, demonstrating the practical need for a more efficient test case generation approach. To trigger many of the bugs, complex query plans are required, indicated by the average length of query plans being 2.47× longer than that of the previously found bugs. In terms of efficiency, our *QPG*-based implementation covers 4.85–408.48× more unique query plans than *SQLancer* and *SQLRight* in 24 hours.

Overall, we make the following contributions:

- We studied the query plans of the queries in previously-found bugs to gauge the idea's potential;
- We propose *Query Plan Guidance* as a general idea for utilizing query plans for testing;
- We propose a concrete testing approach that mutates database state rather than queries to be compatible with existing test oracles;
- We implemented and evaluated the approach, which has found 53 unique, previously unknown bugs in widely-used DBMSs.

II. Background

Database management systems. Database Management Systems (DBMSs) serve as an interface between applications and back-end data, helping users to store, manipulate, and query data based on an abstract data model. The relational data model [19] is the most common model that has been adopted by most modern DBMSs. In this paper, we focus on testing such relational DBMSs.

Structured Query Language. The most commonly used language for interacting with relational DBMSs is the Structured Query Language (SQL) [20], which has been standardized by ISO/IEC 9075. SQL consists of many types of statements [21], which can be classified into three main sub-languages:

1) Data Query Language (DQL), which provides a `SELECT` statement to query data.
2) Data Definition Language (DDL), which is used to create and modify the schemas of data objects, for example, `CREATE`, `DROP`, and `ALERT`.
3) Data Manipulation Language (DML), which is used to modify the contents of data objects, for example, `INSERT` and `UPDATE`.

While DDL and DML statements can affect the database state, queries (*i.e.*, DQL statements) typically cannot. Our test cases consist of DQL, DDL, and DML statements.

Query plans. A query plan is a tree of operations that describes how an SQL statement is executed by a specific DBMS. Although not specified by the standard, most mature relational DBMSs, including the 10 most popular relational DBMSs according to the DB-Engines ranking,[1] allow users to query a textual representation of a query plan by prefixing a query with `EXPLAIN`. DBMSs cannot always determine

[1] https://db-engines.com/en/ranking/relational+dbms

the most efficient query plan [22], [23], requiring users to understand and optimize performance-critical queries (*e.g.*, by providing hints to the DBMS) based on their query plans. For a better debugging experience, exposed query plans may include additional information, such as the estimated cost or predicate expressions (*e.g.*, used in `WHERE` clauses). Database literature distinguishes between logical and physical query plans [24], the latter which is typically exposed by the DBMSs. While the logical query plan closely corresponds to the original declarative query, the physical query plan maps every logical operator to a so-called physical one that can be executed by the DBMS. For example, to translate a read operation on a table, the DBMS might choose one of potentially multiple so-called physical access methods (*e.g.*, a full table scan, or a partial scan with index). Similarly, to join two tables, the DBMS might decide between multiple join algorithms (*e.g.*, hash join or nested loop join) [24]. Various factors influence what query plan a DBMS derives for a given query, such as characteristics of the data stored in the database [25], the existence of auxiliary data structures (*e.g.*, indexes) [26], the tables as well as views present in the database, and configuration options. In this work, we use query plans in a black-box way, that is, without regarding the semantics of operators to guide testing.

Logic bugs. Logic bugs are bugs that cause a system to compute incorrect results. Recently, Rigger et al. proposed several oracles [6]–[8] that have found hundreds of unique bugs in widely-used DBMSs. In this work, we used the two latest test oracles, which represent the state of the art. Ternary Logic Partitioning (TLP) expects a query and derives multiple more complex queries, each of which computes a partition of the result to then check whether their results are equivalent. For example, from `SELECT * FROM t0` and a random predicate `t0.c0>0`, TLP derives `SELECT * FROM t0 WHERE (t0.c0>0)`, `SELECT * FROM t0 WHERE NOT (t0.c0>0)`, and `SELECT * FROM t0 WHERE (t0.c0>0)ISNULL`, whose combined records must be equivalent to the first query. Non-optimizing Reference Engine Construction (NoREC) [7] checks for inconsistent results values of a predicate used in a query that the DBMS might optimize and one that is used in a query that is difficult to optimize. For example, for a predicate `t0.c0>0`, NoREC compares the number of rows returned by a query `SELECT * FROM t0 WHERE (t0.c0>0)` with how often `TRUE` is contained in the result returned for `SELECT (t0.c0>0)FROM t0`. Both oracles have constraints on the query formats. For example, NoREC requires a `WHERE` clause, but forbids aggregate functions and other more complex clauses. In principle, our method can be paired with any oracle.

III. Query Plan Study

To investigate the potential of using query plans as guidance, we studied the uniqueness and complexity of query plans of the queries in previously-found bugs. We hypothesized that we would see a wide variety of query plans, suggesting that a bug-finding technique optimized for exploring more unique query plans might be effective.

TABLE I
SUBJECTS FOR THE QUERY PLAN STUDY.

DBMS	Version	LoC	EXPLAIN Statement
CockroachDB	19.2.12	1.1M	EXPLAIN (OPT)...
DuckDB	0.19	59K	EXPLAIN...
H2	2.0.202	0.3M	EXPLAIN...
MariaDB	10.4.25	3.6M	EXPLAIN FORMAT='JSON'...
MySQL	5.7.33	3.8M	EXPLAIN FORMAT='JSON'...
PostgreSQL	11.16	1.4M	EXPLAIN (COSTS FALSE)...
SQLite	3.30.0	0.3M	EXPLAIN QUERY PLAN...
TiDB	3.0.12	0.8M	EXPLAIN...

TABLE II
QUERY PLANS OF THE QUERIES IN PREVIOUSLY-FOUND BUGS. LENGTH INDICATES THE AVERAGE NUMBER OF OPERATIONS IN A QUERY PLAN.

		Query Plans		
DBMS	Bugs	Sum	Unique	Length
CockroachDB	68	37	32	3.43
DuckDB	75	59	18	2.00
H2	19	10	7	3.70
MariaDB	7	5	5	1.00
MySQL	40	35	22	1.03
PostgreSQL	31	9	3	2.33
SQLite	193	118	62	2.14
TiDB	62	43	32	5.07
			Avg:	2.59

Subjects. We chose the public bug reports from *SQLancer* as our subjects. *SQLancer* provides a public list[2] including all found bugs and corresponding test cases for 499 bug reports across 9 DBMSs. We excluded 4 bugs found in the DBMS TDEngine, as this DBMS does not expose query plans. The query plan of a given query can vary over versions; thus, to obtain accurate query plans, we chose the most relevant release versions when the corresponding bugs were found. The details of the chosen DBMSs are shown in Table I.

Obtaining query plans. For all 495 bug-inducing test cases, we instrumented all queries (*i.e.*, `SELECT` statements) by using *EXPLAIN* statements as listed in Table I. Depending on the DBMS, query plans might include various additional auxiliary information. We identified three such types. One type is the estimated cost (*e.g.*, in PostgreSQL), which differs for almost every query. The second type is expressions in `WHERE` clauses, which are included in the query plan by some DBMSs (*e.g.*, CockroachDB). The third type is random identifiers, which are used to distinguish operations in a query plan (*e.g.*, MariaDB and MySQL). To exclude such auxiliary information, we accordingly adjusted the parameters of the `EXPLAIN` statements, as shown in Table I. Lastly, we removed the names of tables, views, and indexes of the obtained query plans to distinguish query plans based on their structure only. This was based on the intuition that two query plans with the same execution

[2]https://github.com/sqlancer/bugs

logic, but different table names, would be processed similarly by the DBMSs (*e.g.*, `SCAN t1`, and `SCAN t2`).

Uniqueness analysis. Table II shows the query plan distribution. In total, we obtained 316 query plans, of which 57.28% were unique. The number of query plans is lower than that of test cases because 1) not all test cases have queries and 2) some queries that previously exposed bugs were rejected by subsequent versions of the DBMSs. The minimal percentage of unique query plans is 30.51% in DuckDB. The maximum one is 100.00% in MariaDB, due to a low number of test cases. Overall, for the queries in previously-found bugs, the variety of different query plans indicates that covering a wider variety of query plans might increase the likelihood of discovering bugs.

> Query plans of the queries in previously-found bugs vary significantly, as 57.28% of the query plans are unique.

Complexity analysis. We examined the complexity of the query plans of the queries in previously-found bugs. A query plan with many operations is due to a complex database state or query. For instance, in SQLite, a query plan that retrieves data from two tables requires at least three operations: `SCAN table t0`, `SCAN table t1`, and `MERGE results`, which is more complex than `SCAN table t0` alone. As shown in the *Length* column of Table II, the average number of operations per query plan is 2.59, which illustrates that the majority of bug-related query plans are compact. We further found that the most frequent query plan across eight DBMSs is `SCAN table t0`, which represents a sequential scan on a single table, without using an index. For example, in SQLite, 26 of 118 query plans consist of a single table scan. This demonstrates that the query plans for the previously-found bugs are simple. While this could indicate that, compact and simple query plans are sufficient to trigger these previously found bugs—as suggested by the small-scope hypothesis [27]—it could also be that existing approaches have focused their testing on simple queries and database states. We speculate that covering more complex query plans might increase the likelihood of discovering bugs.

> Query plans of the queries in previously-found bugs are compact and simple, as the average number of operations in a query plan is only 2.59.

IV. Approach

To efficiently detect logic bugs in DBMSs, we propose to mutate databases with *Query Plan Guidance (QPG)* towards more unique and increasingly complex database states. Our insight is that the internal execution logic of the DBMS for a given query is reflected by its query plan and, therefore, covering more unique query plans increases the likelihood of finding logic bugs. Compared with naive random generation, our method gradually mutates database states enabling subsequent queries to cover more unique and complex query plans. We chose to mutate database states rather than queries, since test oracles have various constraints on queries, which are difficult to meet using mutational approaches [9]. Compared with other coverage-based grey-box testing tools for DBMSs, such as Squirrel [5] and SQLRight [9], we consider our method as black-box testing, as *QPG* requires no access to the source code of the DBMS and uses information readily provided by mature DBMSs. Thus, the technique can also be applied to commercial closed-source DBMSs.

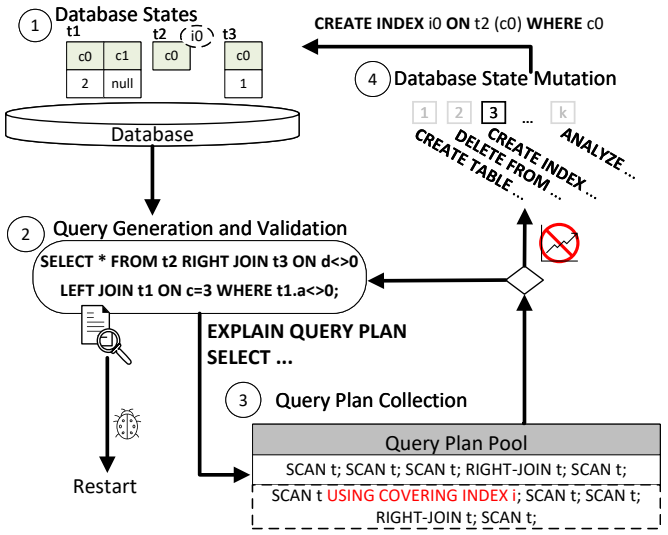

Fig. 1. Overview of QPG. The dashed lines refer to the data affected by ④ in the next iteration.

System overview. Figure 1 shows an overview of our *QPG* realization based on Listing 1. Given an initial database state at ①, *QPG* generates a random SQL query at ② and executes it on the database to validate the query's result using the test oracle. If the oracle indicates a bug, *QPG* outputs a bug report and restarts the testing process. Otherwise, it records the query plan and appends it to the query plan pool at ③. Typically, the execution continues at ② with the same database state. However, if no new unique query plan has been observed after a fixed number of iterations, *QPG* mutates the database state at ④ by applying a mutation operator to the current database state to create a new one, assuming that this new state will subsequently lead to new unique query plans being explored.

A. Database States. (①)

The initial database state can be either randomly generated or manually given. In our implementation, we generate it by randomly executing DDL and DML statements. To avoid empty database states, we execute `CREATE TABLE` statements first. For example, to create the initial database state in Figure 1, we execute lines 1–5 in Listing 1. We do not directly manipulate database files, since they are highly structured [28], and any unexpected byte may incur an error that would impede the testing process.

B. Query Generation and Validation. (②)

Query generation. We generate queries whose results we subsequently automatically validate to find bugs. The generated queries must comply with two main constraints. First,

queries must be semantically valid with respect to the database state. For example, they must reference only existing tables and views. Second, they must adhere to the constraints imposed by the test oracles. For example, the NoREC test oracle requires a `WHERE` clause, but forbids other clauses (*e.g.*, `HAVING` or `GROUP BY`). To address this, we adopt SQLancer's rule-based random generation approach that generates queries based on the SQL dialects' grammar adhering to the imposed constraints. Many query generation approaches have been proposed [12], [29]–[34], and our method can, in principle, be paired with any of these query generation methods.

Validation. We use the state-of-the-art logic-bug oracles NoREC [7] and TLP [6] to validate the queries' results. Both are metamorphic testing approaches [35] and, given a query, derive another query whose result set is used to validate the original query's result. In Figure 1, given the three tables and the test oracle, we generate the query `SELECT * FROM t2 RIGHT JOIN t3 ON d<>0 LEFT JOIN t1 ON c=3 WHERE t1.a<>0`. Since the test oracle indicates that the empty result returned is correct, execution continues at ③. If the test oracle indicates a bug, we output the bug report and restart the testing process.

C. Query Plan Collection. (③)

We collect query plans by instrumenting queries using *EXPLAIN* statements, which is the same approach as presented in Section III. In Figure 1, the statement to obtain the query plan is `EXPLAIN QUERY PLAN SELECT * FROM t2 RIGHT JOIN t3 ON d<>0 LEFT JOIN t1 ON c=3 WHERE t1.a<>0`. We obtain the query plan (shown in the left part of lines 12–17 in Listing 1), and remove table and index names.

We insert query plans into the query plan pool in which we store unique query plans. The pool is implemented as a hash table in which the keys are query plans, and the values are the corresponding query strings. Given a query plan, we check whether the query plan exists in the pool, and insert it if not. In Figure 1, the pool is initially empty, so we insert the query plan (the first line at ③). If no new query plan is inserted into the pool for a fixed number of queries, we invoke ④ aiming to cause the DBMS to explore more unique query plans. Otherwise, we continue to test the DBMS using the same database state at ②. A higher number indicates that we test the DBMS using more queries on a single database state, while a lower one means that we test the DBMS using more database states. The number is set to 1,000 by default, which we determined to work well empirically.

D. Database State Mutation. (④)

If no new query plan has been observed for a fixed number of queries, we invoke the database state mutation ④, which manipulates the database state, aiming to cause the DBMS to explore different query plans for the subsequent queries.

As mutation operators, we consider both the same DDL and DML statements used for generating the initial database state, such as `CREATE TABLE`, `CREATE INDEX`, and `ANALYZE`. A key challenge is to apply promising mutations that likely result in queries triggering new query plans. We model this task as the Multi-Armed Bandit (MAB) problem [18], [36], which is a popular and efficient method that has been used in various fuzzing works [37]–[40]. In MAB, a fixed limited set of resources has to be allocated between competing choices to maximize the expected gain. In our scenario, given a limited computational resource, we choose the SQL statements (choices) to mutate database states to maximize the number of covered unique query plans (gain).

To maximize the expected gain, an automated agent attempts to acquire new knowledge (called "exploration") and optimizes its decisions based on existing knowledge (called "exploitation"). In our problem scenario, given the knowledge that the gains of only some mutation operators have been observed, we consider selecting the next mutation operator from either explored or unexplored mutation operators. Making the decision based on explored mutation operators (exploitation) tends to increase the gain, but may miss potentially higher gain from unexplored mutation operators. Many algorithms have been proposed to strike a balance between exploration and exploitation. We adopt the classic episode greedy algorithm [41], which chooses the operator with the highest known gain with a certain probability and a random one otherwise.

Our algorithm works as follows. At t times when database state mutation ④ is invoked, we choose one mutation operator followed by Equation 1. k is the number of candidate mutation operators. $\hat{\mu}_{i(t)}$ is the known gain of the mutation operator i at time t. ϵ is a fixed probability ranging from 0 to 1; its default value is 0.7, which we determined to work well empirically. With $(1-\epsilon)$ probability, we choose the operator that has the maximum known gain and randomly choose one otherwise.

$$j(t) = \begin{cases} \arg\max_{i=1\ldots k} \left(\hat{\mu}_{i(t)}\right) & (1-\epsilon) \\ random(k) & (\epsilon) \end{cases} \tag{1}$$

Encoding known gain $\hat{\mu}_i$. $\hat{\mu}_i$ is measured as weighted average gain—different from the standard algorithm, which uses an unweighted average—across all iterations where i was chosen. A DBMS is a stateful system. The database state depends not only on the last applied mutation operator, but also on the previous database state. Applying the same mutation operator on changing database states creates different database states, so the gain of a mutation operator across iterations is not independent and identically distributed. For the same mutation operator, the gain in the last iteration is closer to the real gain on the last database state. To approximate the known gain, we use a weight average number in which the latter gain has a higher weight than the former gain. Equation 2 is our equation for updating $\hat{\mu}_i$ in each iteration. Q is the gain for the last time i was chosen. w is the weight of Q, which is a constant ranging from 0 to 1; its default value is 0.25, which we determined to work well empirically. Independent from the number of iterations, the prior gains only take up $(1-w)$ weight for $\hat{\mu}_i$. For example, given $w = 0.1$, $\hat{\mu}_{i(999)} = 0.1$, $Q = 2$ for the $1,000_{th}$ iteration, the $\hat{\mu}_{i(1000)} = 0.1 + (2-0.1) * 0.1 = 0.29$, which is much higher than the unweighted average number

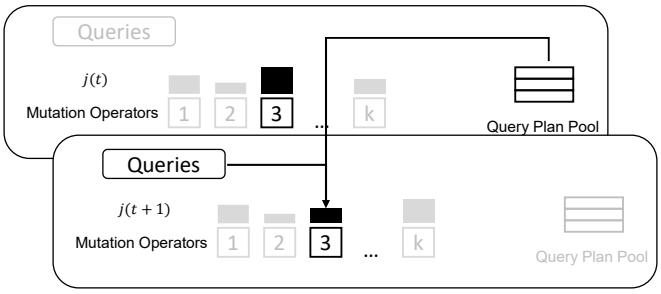

Fig. 2. The workflow of measuring the known gain at ④.

$0.1 + (2 - 0.1)/1000 = 0.1019$ and closer to the Q. For efficiency, all parallel testing processes share the same $\hat{\mu}_i$.

$$\hat{\mu}_{i(t+1)} = \hat{\mu}_{i(t)} + (Q - \hat{\mu}_{i(t)}) * w \tag{2}$$

Encoding instant gain Q. Q is measured by the proportion of queries that explore new query plans when they are executed on the latest database state. The queries include those in the query plan pool, and a set of newly generated queries based on the latest database state. The query plan pool includes all unique query plans and corresponding queries, which we re-execute to evaluate how many new query plans are explored for the same queries. To ensure that the queries in the query plan pool are always valid, we drop the invalid ones that are due to the changes of the database state. We observed that, in practice, this limits the pool to a reasonable size ($< 8,000$ entries). However, for some mutation operators, such as `CREATE TABLE`, none of these queries is related to the newly-created table, so no new query plan is observed. It would be unjust to judge its gain as zero, so we generate a set of new queries and examine how many new query plans are explored. For example, after applying the mutation operator i, $2/50$ queries in the query plan pool and $10/20$ queries in the set of newly generated queries explore unseen query plans, meaning that we compute the instant gain as $Q = 2/50 + 10/20 = 0.54$.

Figure 2 shows the workflow of measuring the known gain $\hat{\mu}_i$ at ④. If the mutation operator 3 is chosen in iteration t due to its highest $j(t)$, we update $\hat{\mu}_3$ in the next iteration $t+1$ with the queries that are generated after iteration t and the queries of the query plan pool in iteration t. Following that, we calculate $j(t+1)$ and choose the mutation operator k.

In Figure 1, we apply `CREATE INDEX i0 ON t2 (c)WHERE c=3`, which creates an index `i0` at ①. Suppose we generate the same query at ②, then we observe the new query plan shown on the right in lines 12–17 in Listing 1 and insert it to the query plan pool. As a result, the bug is exposed at ②.

Lastly, we clear the database state after a fixed number of tested queries aiming to maximize the number of covered unique query plans. In general, by gradually mutating the same database state, we explore more unique and increasingly complex database states. However, the current database state may limit the possible state space to mutate into, which is why we clear the database state and restart the testing process after a fixed number of tested queries. The number is configurable and is set to a reasonable default value of 1,000,000, which we found to work well in our experiments (see Section V).

E. Implementation

We implemented the described *QPG* approach in *SQLancer*[3] and subsequently refer to our prototype as *SQLancer+QPG*. In addition, we updated SQLancer to support the latest version of SQLite which has three new features, namely `RIGHT JOIN`, `FULL OUTER JOIN`, and `STRICT`. We implemented our method in around 1,000 lines of Java code and adapted each DBMS-specific component in additional 100 lines of Java code, such as defining the specific statements for collecting query plans. We designed our approach to be compatible with existing testing tools; thus, for the *Database States* ① and *Query Generation and Validation* ② steps, we reuse the implementation of *SQLancer*. We implemented the algorithm described in *Database State Mutation* ④ as a standalone module that is reused across DBMSs. We used DDL and DML statements supported by *SQLancer* as mutation operators (23 mutations for SQLite, 13 mutations for TiDB, and 17 mutations for CockroachDB) which may contribute to covering more unique query plans, and the detailed list can be found in our artifact. To avoid a large number of tables and indexes causing a low testing throughput, we restricted their maximum number to an arbitrary, but reasonable limit—a maximum of 10 tables and 20 indexes.

V. Evaluation

To evaluate the effectiveness and efficiency of *QPG* in finding bugs in DBMSs, we seek to answer the following questions based on our prototype *SQLancer+QPG*:

Q.1 New Bugs. Can *QPG* help with finding new bugs? Are complex query plans required to find these bugs?

Q.2 Covering unique query plans. Can *QPG* cover more unique query plans than naive random generation and code-coverage guidance methods?

Q.3 Bug Finding Efficiency. Can *QPG* find bugs more efficiently than naive random generation and code-coverage guidance methods?

Q.4 Sensitivity Analysis. What is the contribution of each component of *QPG*? How does *QPG* perform under different configurations?

Tested DBMSs. We tested SQLite, TiDB, and CockroachDB. SQLite is the most popular embedded DBMS—embedded DBMSs are built together with and run in the same process as the application—and is used in every IOS and Android smartphone [1]. TiDB and CockroachDB are popular enterprise-class DBMSs, and their open versions on Github are highly popular as they have been starred more than 31.9k and 25.2k times. They are widely used and have thus also been used in other DBMS testing works [5], [6], [8], [9]. We did not consider other popular DBMSs due to various reasons. For example, for MySQL and closed-source DBMSs, bug fixes can

[3]https://github.com/sqlancer/sqlancer

TABLE III
THE NUMBER OF NEW BUGS FOUND BY *SQLancer+QPG*.

DBMS	Crash	Error	Logic	All
SQLite	0	5	23	28
TiDB	2	4	3	9
CockroachDB	3	11	2	16
Sum:	5	20	28	53

be validated only after new releases; until then, it is difficult to identify new bugs, as already-known bugs might be repeatedly triggered. Furthermore, for some DBMSs, such as MySQL, many previously-reported bugs remain unfixed, impeding the testing process, which was also noted in prior work [6]. As a black-box method, *QPG* supports any DBMS, regardless of what programming languages it is written in; SQLite is written in C, while TiDB and CockroachDB are written in Go. For Q1, Q2, and Q4, we used the latest available development versions (SQLite: 3.39.0, TiDB: 6.3.0, CockroachDB: 23.1). For Q3, to make a fair comparison, we chose the historical versions of DBMSs that all tools have tested and can find bugs in (SQLite: 3.36.0, TiDB: 4.0.15, and CockroachDB: 21.2.2).

Baselines. We compared *SQLancer+QPG* with *SQLancer* and *SQLRight*. While both of them have been designed to find logic bugs, their test case generation techniques differ. *SQLancer* implements a naive random generation method. It is the baseline on which *SQLancer+QPG* is built. It has been starred more than 1,000 times on GitHub and is widely used by companies. *SQLRight* is the state-of-the-art tool and uses code-coverage guidance. By comparing with them, we gain insights into the benefits of *QPG* against naive random generation and code-coverage-guided methods for finding logic bugs.

Experimental infrastructure. We conducted all experiments on an Intel(R) Xeon(R) Gold 6230 processor that has 40 physical and 80 logical cores clocked at 2.10GHz. Our test machine uses Ubuntu 20.04 with 768 GB of RAM, and a maximum utilization of 40 cores. We repeated all experiments 10 times for statistically significant results.

Q.1 New Bugs

We ran *SQLancer+QPG* during approximately two months—during which we also implemented the approach—aiming to find bugs. To better demonstrate the underlying issue for each bug found, we minimized the test case both using C-Reduce [42] and manually. After reporting the bugs to the developers, we suspended the testing process until the bug was fixed to avoid duplicate reports whenever possible; when bugs were not fixed within a timespan of weeks, we reported multiple bugs that we suspected to be unique. The bugs in SQLite were usually fixed within 24 hours, while the bugs in TiDB and CockroachDB were usually fixed within several weeks. As a result, we focused on testing SQLite. We used NoREC [7] and TLP [6], which are the state-of-the-art oracles supported by both *SQLancer* and *SQLRight*.

Listing 2. A bug in the RIGHT JOIN feature of SQLite.

```
1  CREATE TABLE t1(a CHAR);
2  CREATE TABLE t2(b CHAR);
3  INSERT INTO t2 VALUES('x');
4  CREATE TABLE t3(c CHAR NOT NULL);
5  INSERT INTO t3 VALUES('y');
6  CREATE TABLE t4(d CHAR);
7
8  SELECT * FROM t4 LEFT OUTER JOIN t3 ON TRUE
       INNER JOIN t1 ON t3.c='' RIGHT OUTER
       JOIN t2 ON t3.c='' WHERE t3.c ISNULL;  --
       {} 🐞, {|||x} ✔
```

Listing 3. A bug in `json_quote` function of SQLite.

```
1  CREATE TABLE t1 (a CHAR);
2  CREATE VIEW v1(b) AS SELECT json(TRUE);
3  INSERT INTO t1 VALUES ('x');
4
5  SELECT * FROM v1, t1 WHERE NOT json_quote(b); --
       {} 🐞, {1|x} ✔
```

Bugs overview. Table III shows the number of unique, previously unknown bugs found by *SQLancer+QPG*. We found 53 bugs in total, all of which have been confirmed. Of these, 35 have already been fixed. Although *SQLancer* had been extensively applied to these DBMSs, we were still able to find these bugs with the help of *QPG*. Of the 53 bugs, 28 were logic bugs found by the test oracles TLP and NoREC, and 25 bugs were associated with crashes or internal errors. This demonstrates that the complex database states generated by *QPG* are beneficial not only to finding logic bugs, but also to other kinds of bugs. Although CockroachDB used the TLP oracle in their Continuous Integration (CI) process,[4] we still found 16 previously unknown bugs using *QPG*. For the new features in SQLite, *QPG* found 13 bugs in **`RIGHT JOIN`**, 2 bugs in **`FULL JOIN`**, and no bug in **`STRICT`**. We give two examples of found bugs as follows.

Example 1: a bug in the ***`RIGHT JOIN`*** *feature.* Listing 2 shows a test case exposing a logic bug that we found in SQLite. The **`SELECT`** statement incorrectly returns an empty result, because of an incorrect optimization of `ISNULL` when used with a **`RIGHT JOIN`**. The query plan of the **`SELECT`** statement is six operations long: scanning all tables once in four operations, and joining table `t2` with another scan on `t2` in two operations. The query plan is relatively long, because joining tables typically involves multiple operations. 13 bugs in SQLite were in the **`RIGHT JOIN`** feature, in which *QPG* generates more complex database states to find bugs.

Example 2: a bug in JSON feature. Listing 3 is another logic bug that had existed in SQLite since July 23, 2016. The **`SELECT`** statement incorrectly returns an empty result because of an incorrect optimization of the `json_quote` function in the context of a **`VIEW`**, which is necessary to find the bug. The bug cannot be found if the second line is replaced by **`CREATE TABLE`** `v1(b)` **`AS SELECT`** `json(1)`. In SQLite, we found three

[4]https://github.com/cockroachdb/cockroach/commit/777382e6

TABLE IV
QUERY PLANS OF THE QUERIES IN NEWLY FOUND BUGS.

DBMS	All	Unique	Length
SQLite	51	29	5.55
TiDB	12	9	5.67
CockroachDB	6	6	7.83
		Avg:	6.35

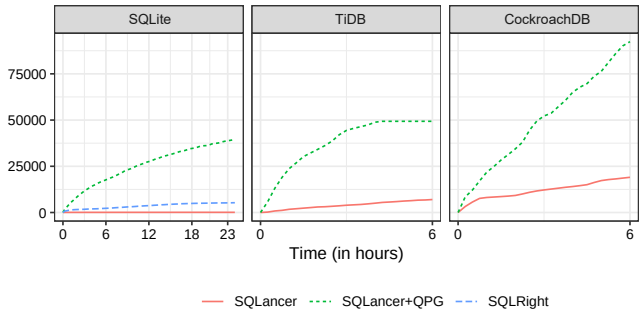

Fig. 3. The average number of unique query plans across 10 runs in 24 hours. We could run TiDB and CockroachDB only for 6 hours due to crashes.

bugs that had been hidden for more than six years, and *SQLancer+QPG* is the first tool to find them despite extensive efforts by the authors of *SQLancer* and *SQLRight*.

The uniqueness and complexity of query plans. To better understand how and whether *QPG* enables exploring a variety of query plans, we analyzed the query plans of the queries in Table III. In total, we obtained 69 query plans, of which 63.77% are unique. This further demonstrates the diversity of query plans. On average, the length of query plans of queries was 6.35. In comparison with Table II, where the average number of operations in a query plan was 2.59, more complex query plans are required to expose these newly found bugs, and *QPG* was successful in causing them to be generated.

With the help of *QPG*, we found 53 unique, previously unknown bugs where the average length of query plans of queries is 6.35.

Q.2 Covering Unique Query Plans

We evaluated whether *SQLancer+QPG* can cover more unique query plans than *SQLancer* and *SQLRight* in 24 hours. Our study in Section III shows that query plans in previously-found bugs are diverse, so covering more unique query plans likely increases the probability of finding bugs. We designed *SQLancer+QPG* to explore more unique and complex query plans than *SQLancer*. We used the TLP oracle, which is the only test oracle that is supported by all DBMSs we considered.

Measurements. Figure 3 shows the average number of unique query plans covered by all tools across 10 runs in 24 hours. We recorded the query plans every 15 minutes and removed the names of tables, views, and indexes as described in Section III. For TiDB and CockroachDB, we could run *SQLancer+QPG* at most for 6 hours, because *SQLancer+QPG* found several crash bugs that remained unfixed during our evaluation. We could run *SQLRight* only on SQLite, as *SQLRight* does not support TiDB and CockroachDB. Table V shows the average and median lengths of query plans of the queries executed across 10 runs in 24 hours.

TABLE V
THE AVERAGE AND MEDIAN NUMBER OF QUERY PLAN LENGTHS ACROSS 10 RUNS IN 24 HOURS. ONLY 6 HOURS ARE SHOWN FOR TIDB AND COCKROACHDB BECAUSE OF CRASHES.

	SQLancer		*SQLRight*		*SQLancer+QPG*	
DBMS	**Avg**	**Median**	**Avg**	**Median**	**Avg**	**Median**
SQLite	2.95	2.00	2.17	1.00	4.69	4.00
TiDB	3.97	2.00	-	-	15.04	8.20
CockroachDB	4.55	4.00	-	-	8.87	6.90
Avg:	3.82	2.67	2.17	1.00	9.53	6.37

TABLE VI
THE LINE AND BRANCH COVERAGE ACROSS 10 RUNS IN 24 HOURS.

	SQLancer		*SQLRight*		*SQLancer+QPG*	
DBMS	**Line**	**Branch**	**Line**	**Branch**	**Line**	**Branch**
SQLite	30.3%	22.7%	48.1%	38.9%	32.6%	24.4%

Results. On both metrics, the number of unique query plans and their complexity, *SQLancer+QPG* clearly outperforms *SQLancer* and *SQLRight*. *SQLancer+QPG* exercises 4.85–408.48× more unique query plans than *SQLancer* and 7.46× more than *SQLRight*. CockroachDB provides fine-grained query plans, which is why *SQLancer+QPG* most clearly outperformed *SQLancer* on this DBMS. The growth rate of *SQLancer+QPG* in TiDB stagnates at around 5 hours due to a crash bug that terminated the TiDB server process. Table V shows that the average length of query plans in *SQLancer+QPG* is 1.59–3.79× longer than for *SQLancer*, and 2.16× longer than for *SQLRight*. To mitigate randomness, we measured the Vargha-Delaney [43] ($\hat{A}_{12}$) and Wilcoxon rank-sum test [44] (U) of *SQLancer+QPG* against *SQLancer*. $\hat{A}_{12}$ measures the *effect size* and gives the probability that random testing of *SQLancer+QPG* is better than random testing of *SQLancer* (*i.e.*, $\hat{A}_{12} > 0.5$ means *SQLancer+QPG* is better). The Wilcoxon rank sum test U is a non-parametric *statistical hypothesis test* to assess whether the result differs across both tools. We reject the null hypothesis if $U < 0.05$, that is, *SQLancer+QPG* outperforms *SQLancer* with statistical significance. For both metrics, $\hat{A}_{12} = 1$ and $U < 0.05$ for *SQLancer+QPG* against *SQLancer* on all DBMSs. The results show that our algorithm continuously generates significantly more unique and complex database states for testing.

QPG exercises 4.85–408.48× more unique query plans than a naive random generation method and 7.46× more than a code-coverage guidance method.

Code coverage. While we were primarily interested in

TABLE VII
THE NUMBER OF ALL AND UNIQUE BUGS FOUND ACROSS 10 RUNS.

	SQLancer		*SQLRight*		*SQLancer+QPG*	
DBMS	**All**	**Unique**	**All**	**Unique**	**All**	**Unique**
SQLite	2	1	2	1	4	2
TiDB	56	10	-	-	118	12
CockroachDB	4	2	-	-	8	3
Sum:	62	13	2	1	130	17

covering more unique query plans, code coverage is a common metric of interest that also gives some insights on how much of a system might be tested. Thus, we evaluated the line and branch coverage of all three tools. Since TiDB and CockroachDB are written in Go, which is not supported by *SQLRight*, we measured code coverage only for SQLite. Table VI shows the average percentage of line and branch coverage across 10 runs in 24 hours. Although *SQLancer+QPG* does not aim to maximize code coverage, *SQLancer+QPG* still outperforms *SQLancer* on both line coverage and branch coverage because of more unique query plans covered. *SQLRight* clearly achieves the highest coverage. The reasons for this are that 1) *SQLRight* was designed to increase code coverage, 2) *SQLancer* and *SQLancer+QPG* only generate SQL statements for the core logic of DBMS, while *SQLRight* produces all kinds of SQL statements by parsing the grammar files from DBMSs, and 3) *SQLRight* provides high-quality seeds that already cover 34.1% line coverage and 26.4% branch coverage, outperforming the other tools even without mutations. Since SQLite achieves 100% branch coverage in their internal testing,[5] we believe that higher code coverage has a limited contribution for finding logic bugs.

Q.3 Bug Finding Efficiency

We evaluated whether *SQLancer+QPG* finds bugs faster than *SQLancer* and *SQLRight*. To this end, we ran *SQLancer+QPG*, *SQLancer*, and *SQLRight* for 24 hours with the TLP oracle. We used a best-effort method to distinguish unique bugs by checking whether 1) stack traces are the same (crash bugs); 2) error messages are the same (error bugs); 3) SQL clause structures are the same (logic bugs), such as two bugs' queries that only have `RIGHT JOIN` and `GROUP BY` clauses are deemed to be duplicate bugs.

Table VII shows the sum of all bugs and only assumed-unique bugs found by each tool in 24 hours and 10 runs. Since crash bugs terminate the whole process, all experiments concluded in less than 24 hours until the first crash was observed (SQLite: 9 hours, TiDB: 1 hour, and CockroachDB: 16 hours). We did not restart the testing process as this would disadvantage *SQLancer+QPG* by making it lose the database states. Overall, *SQLancer+QPG* found 2× more bugs and 1.4× more unique bugs than *SQLancer*; 65× more bugs and 17× more unique bugs than *SQLRight*. As duplicate bugs significantly slow down the testing process and hinder finding other bugs, the number of unique bugs is much smaller than the number of all bugs. In TiDB, we found several easy-to-reach bugs in `JOIN`s, which do not require complex database states, so the number of all bugs is much higher than for the others. The results further show that bugs can be more efficiently found by exploring more unique query plans.

[5]https://www.sqlite.org/testing.html#mcdc

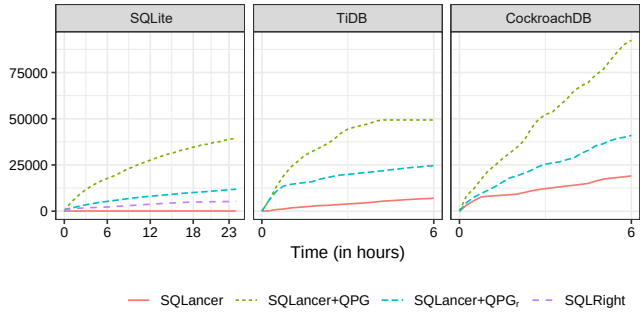

Fig. 4. The average number of covered unique query plans to evaluate **the contributions of algorithm components** across 10 runs in 24 hours.

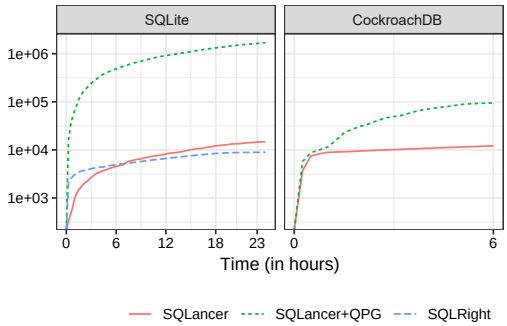

Fig. 5. The average number of covered unique query plans by the **NoREC oracle** across 10 runs in 24 hours. The y axis uses a log scale.

> *QPG* finds previous bugs 1.4× faster than a naive random generation method and 17× faster than a code-coverage guidance method.

Q.4 Sensitivity Analysis

To evaluate the contribution of *SQLancer+QPG*'s components, we performed a sensitivity analysis.

Contributions of algorithm components. Our major contributions are *query plan collection* ③ and *database state mutation* ④ shown in Figure 1. To assess their contributions, we derived a new configuration $SQLancer + QPG_r$ that enables only the query plan collection ③, and randomly applies mutations in ④. Figure 4 shows the average number of covered unique query plans across 10 runs in 24 hours with the TLP oracle. *SQLancer+QPG* outperforms $SQLancer + QPG_r$, demonstrating the contribution of ④. $SQLancer + QPG_r$ outperforms *SQLancer*, demonstrating the contribution of ③. *SQLancer+QPG* has a higher growth rate than $SQLancer + QPG_r$, because ④ gradually learns which mutation operators are promising. Due to the crash bugs, we ran TiDB and CockroachDB for only 6 hours.

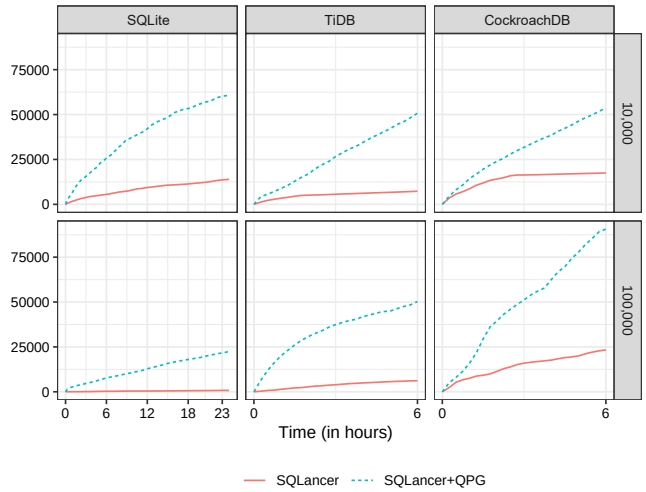

Fig. 6. The average number of covered unique query plans by **varying the maximum number of queries per database state** across 10 runs in 24 hours.

Sensitivity of oracles. We also evaluated *SQLancer+QPG* with NoREC, which is the second state-of-the-art oracle. Figure 5 shows the average number of covered unique query plans across 10 runs in 24 hours for NoREC oracle. *SQLancer* lacks a NoREC oracle for TiDB, so we exclude it here. All tools have a higher number of covered unique query plans with the NoREC than with the TLP oracle, because of different constraints on queries from NoREC and TLP. Similar to TLP, *SQLancer+QPG* gains a significant advantage over *SQLancer* and *SQLRight* with the NoREC oracle.

Sensitivity of maximum queries per database. Both *SQLancer+QPG* and *SQLancer* have a configuration to control the number of tested queries before clearing database states and starting a fresh testing instance. The default value for both is 1,000,000. Often, starting a fresh testing instance at ① may result in a higher number of covered unique query plans. To evaluate whether *SQLancer+QPG* still performs well when more frequently resetting database states, we adjusted the number to 10,000 and 100,000, and evaluated the number of their covered unique query plans. Figure 6 shows the average number of covered unique query plans under the various maximum number of queries per database state. *SQLancer+QPG* gains a significant advantage over *SQLancer* in all experiments. We clearly see that the rate of newly discovered query plans of *SQLancer* stagnates over time, while *SQLancer+QPG*'s rate continues to increase. Configuring the number is a trade-off since *SQLancer+QPG* creates more complex query plans with a higher number of maximum queries per database state and more unique query plans with a lower number. A user can adjust the configuration option depending on the testing goals.

Sensitivity of mutations. To evaluate the contribution of each mutation, we examined how often each mutation (*i.e.*, SQL statement) was executed across 10 runs in 24 hours. Figure 7 shows the five most frequently executed mutations for each DDBMS. The most frequently-executed mutation for SQLite is `CREATE TABLE`. Other frequently executed mutations

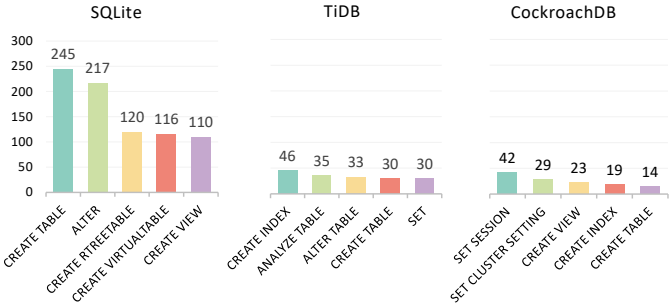

Fig. 7. How often a mutation was executed for the five most frequently executed mutations for SQLite, TiDB, and CockroachDB across 10 runs.

either create other kinds of tables that are unique to SQLite or change the schema of existing tables using `ALTER`. This is expected, as more kinds of tables subsequently cause SQLite to explore more query plans. Despite frequently creating additional tables, we did not observe excessive execution times, as we limited the maximum number of tables and indexes. For TiDB and CockroachDB, the number of mutations is much lower than that for SQLite, as we could run them for only up to 6 hours. *QPG* favors the mutation `CREATE INDEX` for TiDB, because indexes allow it to use more efficient physical operators when reading data. For CockroachDB, *QPG* favors the mutation `SET SESSION`, because it changes the system options, which can have an impact on the query plan. *QPG* favors creating tables as various types of tables are supported in SQLite. Overall, all DBMSs have common frequent mutations, such as `CREATE TABLE`, yet have distinct frequent mutations, such as `SET`, depending on the various characteristics of DBMS.

For all three analyses, $\hat{A}_{12} = 1$ and $U < 0.05$ for *SQLancer+QPG* against *SQLancer* on all DBMSs, which indicates the results are statistically significant.

VI. Related Work

Fuzzing. Fuzzing is an automatic software testing technique that generates or mutates inputs to target programs for finding crash bugs [45]. In recent years, it has gained increased attention, because of the success of the coverage-guided grey-box fuzzers such as AFL [10], [11], which instrument target programs to record code coverage which is subsequently used to mutate inputs to maximize code coverage. A plethora of works [46] have been proposed to improve fuzzing in various aspects. While QPG relates to grey-box fuzzing, we focus on finding logic bugs and DBMSs specifically, and guide test case generation by query plans rather than code coverage.

Finding logic bugs. Various techniques have been proposed to find logic bugs in DBMSs. Differential testing [47] is a general technique that compares the outputs of multiple systems for the same input to detect potential discrepancies indicating bugs; various approaches use it as a test oracle for finding logic bugs by using different DBMSs [16], [48], [49] or different versions of a DBMS [4], [50]. While such approaches have successfully found bugs, they are prone to

false alarms due to differences in SQL dialects or expected differences between versions. Subsequently, three test oracles were proposed and implemented in *SQLancer* [6]–[8]. While we evaluated our technique with the state-of-the-art oracles NoREC and TLP, our method is compatible with any oracles.

Query generation. Targeted and random generations are two major directions in query generation. As for targeted query generation, Bati et al. [29] proposed to incorporate execution feedback, such as code coverage, for guiding query generation to reach a specific code location. Khalek et al. [51] used a solver-backed approach to generate syntactically and semantically correct queries. Generating queries that satisfy cardinality constraints has been proven to be computationally hard, which is why heuristic algorithms were proposed [52], [53]. As for random query generation, SQLsmith [15] uses a predefined grammar to randomly generate semantic valid queries and has found over 100 bugs in widely-used DBMSs. APOLLO [12] also uses a predefined grammar to generate queries for finding regression performance issues. Similarly, we use a grammar-based random generation method for generating valid queries. Squirrel [5] and *SQLRight* [9] use a mutation-based method to generate new queries, but such approaches are prone to generating queries that are semantically invalid.

Database state generation. Similarly, targeted and random generations are two major directions in database state generation. As for targeted database state generation, QAGen [54] uses symbolic execution to specify constraints and generate queries that satisfy the constraints. SPQR [55] generates the database state for a given query and expected results. As for random database state generation, Gray et al. [56] proposed to quickly generate billions-record databases using parallel algorithms. Coverage-based methods [5], [9] generate new database states by mutating given SQL statements that are used to create the database state. Compared with these methods, we used query plans as guidance to generate more diverse database states for efficiently finding logic bugs.

Query plan in testing. Database researchers have invested decades of effort to improve the performance of DBMSs, often by improving the performance of generated query plans or the operators used in them [17], [57]–[60]; providing a comprehensive summary of these exceeds the scope of this paper. In terms of testing, Gu et al. [23] proposed measuring the accuracy of query optimizers by forcing the generation of multiple alternative query plans for each test case, timing the execution of all alternatives, and ranking the plans by their effective costs with the goal of comparing this ranking with the ranking of the estimated cost. Leis et al. [22] measured both the effects of the cost model and cardinality estimators used to derive an efficient query plan. Rather than improving, studying, or testing the accuracy of query plans, we use query plans to guide test case generation.

VII. Conclusion

In this paper, we have proposed the concept of *Query Plan Guidance* (*QPG*) to efficiently detect logic bugs in DBMSs. Its core insight is that the DBMS' internal execution logic for a given query is reflected by its query plan and, therefore, covering more unique query plans might increase the likelihood of finding logic bugs. Our study shows that the query plans of the queries in previously-found bugs vary significantly, but are simple. Thus, we designed an algorithm to gradually mutate database states toward more unique and complex query plans. *QPG* enabled us to find 53 unique, previously unknown bugs in widely-used and extensively-tested database systems—SQLite, TiDB, and CockroachDB. The experiments show that *QPG* results in 4.85–408.48× more unique query plans than a random-generation method and 7.46× more than a code coverage-guidance method. *QPG* also improves logic-bug finding efficiency by 2×. Overall, this paper has demonstrated that *QPG* is a general-applicable, black-box approach that increases bug-finding efficiency and enables finding difficult-to-trigger bugs. While we demonstrated *QPG* in the context of automated testing, we believe that the core idea could be applied also in other contexts (*e.g.*, to measure the quality of a test suite).

VIII. Data Availability

Our implementation and experimental data are publicly available at https://zenodo.org/record/XXXX

References

[1] Website, "Most widely deployed and used database engine," https://www.sqlite.org/mostdeployed.html, 2022, accessed: 2022-06-08.

[2] ——, "Tidb customers," https://en.pingcap.com/customers, 2022, accessed: 2022-06-08.

[3] ——, "Cockroachdb customers," https://www.cockroachlabs.com/customers, 2022, accessed: 2022-06-08.

[4] J. Yan, Q. Jin, S. Jain, S. D. Viglas, and A. Lee, "Snowtrail: Testing with production queries on a cloud database," in *Proceedings of the Workshop on Testing Database Systems*, 2018, pp. 1–6.

[5] R. Zhong, Y. Chen, H. Hu, H. Zhang, W. Lee, and D. Wu, "Squirrel: Testing database management systems with language validity and coverage feedback," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 955–970.

[6] M. Rigger and Z. Su, "Finding bugs in database systems via query partitioning," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, 2020.

[7] ——, "Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction," in *Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, 2020.

[8] ——, "Testing database engines via pivoted query synthesis," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. Banff, Alberta: USENIX Association, Nov. 2020.

[9] Y. Liang, S. Liu, and H. Hu, "Detecting logical bugs of DBMS with coverage-based guidance," in *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Aug. 2022.

[10] Website, "American fuzzy lop (afl) fuzzer," http://lcamtuf.coredump.cx/afl/technical_details.txt, 2013, accessed: 2022-06-08.

[11] ——, "libfuzzer – a library for coverage-guided fuzz testing." https://llvm.org/docs/LibFuzzer.html, 2013, accessed: 2022-06-08.

[12] J. Jung, H. Hu, J. Arulraj, T. Kim, and W. Kang, "Apollo: Automatic detection and diagnosis of performance regressions in database systems," *Proc. VLDB Endow.*, vol. 13, no. 1, p. 57–70, Sep. 2019. [Online]. Available: https://doi.org/10.14778/3357377.3357382

[13] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, "Stateful greybox fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Aug. 2022.

[14] Z.-M. Jiang, J.-J. Bai, and Z. Su, "Dynsql: Stateful fuzzing for database management systems with complex and valid sql query generation," Aug. 2023.

[15] Website, "Sqlsmith," https://github.com/anse1/sqlsmith, 2015, accessed: 2022-06-08.

[16] D. R. Slutz, "Massive stochastic testing of sql," Tech. Rep. MSR-TR-98-21, August 1998. [Online]. Available: https://www.microsoft.com/en-us/research/publication/massive-stochastic-testing-of-sql/

[17] G. Graefe, "Query evaluation techniques for large databases," *ACM Computing Surveys (CSUR)*, vol. 25, no. 2, pp. 73–169, 1993.

[18] D. A. Berry and B. Fristedt, "Bandit problems: sequential allocation of experiments (monographs on statistics and applied probability)," *London: Chapman and Hall*, vol. 5, no. 71-87, pp. 7–7, 1985.

[19] E. E, F. Codd, "Derivability, redundancy and consistency of relations stored in large data banks," *ACM SIGMOD Record*, vol. 38, no. 1, pp. 17–36, 2009.

[20] D. D. Chamberlin and R. F. Boyce, "Sequel: A structured english query language," in *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, 1974, pp. 249–264.

[21] Website, "Iso/iec 9075:1992, database language sql- july 30, 1992," https://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt, 1992, accessed: 2022-06-08.

[22] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proceedings of the VLDB Endowment*, vol. 9, no. 3, pp. 204–215, 2015.

[23] Z. Gu, M. A. Soliman, and F. M. Waas, "Testing the accuracy of query optimizers," in *Proceedings of the Fifth International Workshop on Testing Database Systems*, 2012, pp. 1–6.

[24] A. Silberschatz, H. F. Korth, S. Sudarshan *et al.*, *Database system concepts*. McGraw-Hill New York, 2002, vol. 1, ch. 15.

[25] V. Poosala, *Histogram-based estimation techniques in database systems*. The University of Wisconsin-Madison, 1997.

[26] G. Graefe *et al.*, "Modern b-tree techniques," *Foundations and Trends® in Databases*, vol. 3, no. 4, pp. 203–402, 2011.

[27] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov, "Evaluating the "small scope hypothesis"," in *In Popl*, vol. 2, 2003.

[28] S. Jeon, J. Bang, K. Byun, and S. Lee, "A recovery method of deleted record for sqlite database," *Personal and Ubiquitous Computing*, vol. 16, no. 6, pp. 707–715, 2012.

[29] H. Bati, L. Giakoumakis, S. Herbert, and A. Surna, "A genetic approach for random testing of database systems," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07. VLDB Endowment, 2007, pp. 1243–1251.

[30] N. Bruno, S. Chaudhuri, and D. Thomas, "Generating queries with cardinality constraints for dbms testing," *IEEE Trans. on Knowl. and Data Eng.*, vol. 18, no. 12, pp. 1721–1725, Dec. 2006.

[31] C. Mishra, N. Koudas, and C. Zuzarte, "Generating targeted queries for database testing," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 499–510. [Online]. Available: http://doi.acm.org/10.1145/1376616.1376668

[32] M. Poess and J. M. Stephens, Jr., "Generating thousand benchmark queries in seconds," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, ser. VLDB '04. VLDB Endowment, 2004, pp. 1045–1053.

[33] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '79. New York, NY, USA: Association for Computing Machinery, 1979, p. 23–34. [Online]. Available: https://doi.org/10.1145/582095.582099

[34] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis, "Cockroachdb: The resilient geo-distributed sql database," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2020.

[35] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. Tse, and Z. Q. Zhou, "Metamorphic testing: A review of challenges and opportunities," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–27, 2018.

[36] J. Gittins, K. Glazebrook, and R. Weber, *Multi-armed bandit allocation indices*. John Wiley & Sons, 2011.

[37] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, "{EcoFuzz}: Adaptive {Energy-Saving} greybox fuzzing as a variant of the adversarial {Multi-Armed} bandit," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2307–2324.

[38] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy, and N. Abu-Ghazaleh, "{SyzVegas}: Beating kernel fuzzing odds with reinforcement learning," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2741–2758.

[39] M. Wu, L. Jiang, J. Xiang, Y. Huang, H. Cui, L. Zhang, and Y. Zhang, "One fuzzing strategy to rule them all," in *Proceedings of the International Conference on Software Engineering*, 2022.

[40] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 861–875.

[41] V. Kuleshov and D. Precup, "Algorithms for multi-armed bandit problems," *arXiv preprint arXiv:1402.6028*, 2014.

[42] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 335–346. [Online]. Available: https://doi.org/10.1145/2254064.2254104

[43] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[44] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[45] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[46] Website, "Recent papers related to fuzzing," https://wcventure.github.io/FuzzingPaper/, 2018, accessed: 2022-06-08.

[47] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[48] B. Ghit, N. Poggi, J. Rosen, R. Xin, and P. Boncz, "Sparkfuzz: searching correctness regressions in modern query engines," in *Proceedings of the workshop on Testing Database Systems*, 2020, pp. 1–6.

[49] Website, "Sqllogictest document," https://www.sqlite.org/sqllogictest/doc/trunk/about.wiki, 2022, accessed: 2022-06-08.

[50] K. Yagoub, P. Belknap, B. Dageville, K. Dias, S. Joshi, and H. Yu, "Oracle's sql performance analyzer." *IEEE Data Eng. Bull.*, vol. 31, no. 1, pp. 51–58, 2008.

[51] S. Abdul Khalek and S. Khurshid, "Automated sql query generation for systematic testing of database engines," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 329–332.

[52] N. Bruno, S. Chaudhuri, and D. Thomas, "Generating queries with cardinality constraints for dbms testing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, no. 12, pp. 1721–1725, 2006.

[53] C. Mishra, N. Koudas, and C. Zuzarte, "Generating targeted queries for database testing," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 499–510.

[54] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu, "Qagen: Generating query-aware test databases," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 341–352.

[55] C. Binnig, D. Kossmann, and E. Lo, "Reverse query processing," in *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 2006, pp. 506–515.

[56] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly generating billion-record synthetic databases," in *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, 1994, pp. 243–252.

[57] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs, and J. F. Naughton, "Predicting query execution time: Are optimizer cost models really unusable?" in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 2013, pp. 1081–1092.

[58] J. McHugh and J. Widom, "Query optimization for xml," Stanford InfoLab, Tech. Rep., 1999.

[59] J. Paul, J. He, and B. He, "Gpl: A gpu-based pipelined query processing engine," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1935–1950.

[60] J. Giceva, G. Alonso, T. Roscoe, and T. Harris, "Deployment of query plans on multicores," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, pp. 233–244, 2014.