Detecting Logical Bugs of DBMS with Coverage-based Guidance

Yu Liang[†] Song Liu[†*] Hong Hu[†]

[†]*Pennsylvania State University* [*]*Qi-AnXin Tech. Research Institute*

Abstract

Database management systems (DBMSs) are critical components of modern data-intensive applications. Developers have adopted many testing techniques to detect DBMS bugs such as crashes and assertion failures. However, most previous efforts cannot detect logical bugs that make the DBMS return incorrect results. Recent work proposed several oracles to identify incorrect results, but they rely on rule-based expression generation to synthesize queries without any guidance.

In this paper, we propose to combine coverage-based guidance, validity-oriented mutations and oracles to detect logical bugs in DBMS systems. Specifically, we first design a set of general APIs to decouple the logic of fuzzers and oracles, so that developers can easily port fuzzing tools to test DBMSs and write new oracles for existing fuzzers. Then, we provide validity-oriented mutations to generate high-quality query statements in order to find more logical bugs. Our prototype, `SQLRight`, outperforms existing tools that only rely on oracles or code coverage. In total, `SQLRight` detects 18 logical bugs from two well-tested DBMSs, `SQLite` and `MySQL`. All bugs have been confirmed and 14 of them have been fixed.

1 Introduction

Database management systems (DBMSs) are used extensively in data-intensive programs, helping billions of devices host trillions of databases [23, 51, 54–57]. Any bug in DBMSs will affect a large number of users. Although many efforts have been spent on testing DBMS systems [10, 58, 70, 71], most of them focus on finding crashes and assertion failures that finally make the execution terminate accidentally. They cannot detect *logical* bugs that make the DBMS return unexpected results, like leaking extra rows. Since logical bugs usually do not crash the program, we need an oracle to determine whether each execution produces the correct result or not. However, building an oracle is time-consuming and error-prone due to the various language dialects and features [49].

Recent progress on DBMS oracles shed light on logical bug detection [43–45]. For example, Rigger *et al*. constructed several general oracles by transforming SQL queries into semantically equivalent forms. One DBMS may process these queries with different code paths, but the final results should be the same. Any inconsistency indicates a potential logical bug. `SQLancer`, the tool that implements these oracles, has successfully found many logical bugs [30, 42]. However, `SQLancer` relies on a rule-based generator to synthesize original queries, which may limit its capability to explore program states. Specifically, it creates expressions for `WHERE` and `JOIN` clauses based on the grammar of the specific DBMS and the schema of the underlying database. Considering the huge query space, it could invest a great deal of time and effort on similar queries, which cannot inspect diverse program code.

Coverage-guided program testing, or *fuzzing*, has been adopted to test a wide range of programs and successfully found thousands of memory-related bugs [18, 27, 32, 48, 70]. The core idea of fuzzing is to utilize the code coverage to guide the input generation. A fuzzing tool, or *fuzzer*, picks up an input from a queue and randomly updates it to produce new test cases. Then, it runs the program with the new test case and at the same time collects the code (like basic blocks or branches) being executed. If the execution triggers new code, the fuzzer will add the new test case to the queue and use it for future mutation; otherwise, the fuzzer will drop the new test case and move on to mutate another input in the queue. Previous works applied coverage-guided fuzzing to test DBMSs and demonstrated the benefits of detecting memory-related bugs [10, 60, 71]. However, no research ever tried to apply coverage-based guidance on logical bug detection.

In this paper, we aim to understand the benefits of coverage-based guidance on logical bug detection, specifically for DBMS systems. Our study reveals several challenges of adopting current mutation-based fuzzers to find logical bugs. The main problem comes from generating valid SQL queries. Since we need to compare the results of DBMS executions, the generated query should pass both syntax and semantic checks and successfully produce meaningful outputs. Existing fuzzers can synthesize decent queries that trigger assertion failures and crashes, but many of them are not completely

valid and thus cannot be used for finding logical bugs.

To address this problem, we propose *validity-oriented mutation*, which defines a set of policies to improve the validity of generated queries. First, we design an automatic method to convert the SQL parser of each DBMS for fuzzing purposes. Currently, most fuzzers use one parser to handle different DBMS systems [10, 71]. Since many DBMSs use their own SQL dialects to support unique features [49], the unified parser may produce a lot of incompatible queries. Even worse, the incorrect queries could trigger error-handling code and are prioritized by coverage-based guidance. Therefore, we provide one query parser for each DBMS to reduce the invalid queries. Our second effort is to build a context-based instantiation algorithm that enforces accurate dependencies between SQL elements. For example, `DROP TABLE X` removes table `X` from the current database. Our method will follow the semantics and remove the corresponding table from the context so that it will not be used for the following statements. We also consider the oracle requirement to produce more useful queries. Specifically, we allow the oracle to label necessary elements as immutable. Our cooperative mutation engine will avoid changing these elements while randomly updating others. At last, we eliminate non-deterministic behaviors from the queries to avoid unnecessary false alarms. With these solutions, we effectively improve the query validity rate.

Another problem that hinders fuzzing to detect logical bugs is the lack of unified interfaces between fuzzers and DBMS oracles. Most fuzzers simply rely on the operating system or various sanitizers to detect bugs [28, 46, 47], while detecting logical bugs requires generating proper SQL statements and checking the execution results. To fill this gap, we design a set of expressive APIs to simplify the implementation of DBMS oracles. Our APIs decouple the fuzzing logic and the oracle logic. Developers can focus on one area, either fuzzing or oracle, to easily apply existing methods to detect logical bugs.

We implement a system, `SQLRight`, that combines the coverage-based guidance, validity-oriented mutations and oracles to detect logical bugs for DBMS systems. `SQLRight` first mutates existing queries cooperatively. It inserts a set of oracle-required statements, and applies our validity-oriented mutations to improve the validity rate. Then, it sends the query to the oracle to create functionally equivalent query counterparts. `SQLRight` feeds all generated queries to the DBMS, and collects the execution results and the coverage information. After that, `SQLRight` invokes the oracle to compare the results of different queries to identify logical bugs. At last, it inserts the coverage-improving queries into the queue for future mutations. We implemented four oracles, including two proposed in previous works [43, 44] and two proposed in this paper.

We evaluate `SQLRight` on three popular DBMS systems: `SQLite` [51], `PostgreSQL` [39] and `MySQL` [34]. `SQLRight` successfully detects 18 logic bugs within 60 days. We have reported all of our findings to their developers: all bugs get confirmed, and 14 have been fixed. To understand the contributions of code coverage and query validity, we conduct unit tests by disabling each of them one by one. Our evaluation shows that both the coverage and the validity help find more logical bugs, but the latter makes more contributions than the former. We also compare our system with the state-of-the-art tools, including `SQLancer` (using oracles to detect logical bugs) and `Squirrel` (using code coverage to detect crashes and assertion failures). We port our oracle APIs to `Squirrel` to help it detect logical bugs. After testing for 72 hours, `SQLRight` reports 12 unique logical bugs, `Squirrel` detects one bug, and `SQLancer` does not find any bug. The result shows that combining the coverage-based guidance and oracles can help trigger more logical bugs in DBMS systems.

```
01 CREATE TABLE person (pid INT);
02 INSERT INTO person VALUES (1), (10), (10);
03 CREATE UNIQUE INDEX idx ON person (pid) WHERE pid=1;
04 SELECT DISTINCT pid FROM person WHERE pid=10;
05 -- output: 10\n10
06 -- expect: 10
```

Listing 1: A logical bug of `SQLite` due to the improper optimization on `DISTINCT` when the index is a unique partial index.

In summary, this paper makes the following contributions:

- We study the efficacy of coverage-guided fuzzing in detecting logical bugs for DBMS systems and confirm two useful factors: query validity and code coverage.
- We implement `SQLRight`, a coverage-guided fuzzer to detect logical bugs for DBMSs. `SQLRight` provides general APIs to simplify the oracle development and embeds validity-oriented mutations to improve the query quality.
- We evaluate `SQLRight` on the real-world DBMS systems and find 18 logical bugs. Our unit tests demonstrate the contribution of different components of our tool.

We have released the source code of `SQLRight` at https://github.com/psu-security-universe/sqlright to help enhance the security and robustness of DBMS systems.

2 Background & Challenges

2.1 An Example Logical Bug

Listing 1 shows a logical bug of `SQLite`, which is related to the unique partial index. This bug was detected by our tool `SQLRight`, and has been fixed by `SQLite` developers. The first statement creates a table `person`, which has only one column `pid` with type `INT`. The second statement inserts three rows into the table, one `1` and two `10`s. The third statement creates a unique partial index `idx`, which only maintains records for rows whose `pid` value is `1`. Therefore, only the first row will be connected into `idx`. The last `SELECT` statement asks for rows whose `pid` value is `10`, where the keyword `DISTINCT` requires removing redundant results. Based on the table content, the result should be one row `10`. However, `SQLite` produces two rows `10`⏎`10`, violating the `DISTINCT` requirement.

The problem is due to an incorrect optimization of `SELECT`. When all columns in `WHERE` are connected to some unique

```
01 CREATE TABLE person (pid INT);
02 INSERT INTO person VALUES (1), (10), (10);
03 SELECT DISTINCT pid FROM person WHERE pid=10;
```

Listing 2: A functional-equivalent query of Listing 1 query by deleting the `CREATE UNIQUE INDEX` statement.

indexes, `SQLite` treats the `DISTINCT` keyword as unnecessary and simply ignores it during the query processing. However, it forgets the case of partial index, where only partial rows are connected. To fix this bug, `SQLite` checks the index type and only applies the optimization to the case of complete indexes.

Security Impact. This logical bug treats duplicated data rows as unique. Depending on the concrete use cases, it can lead to various security consequences. For example, if the query intentionally uses `DISTINCT` to hide the number of matched rows for privacy purposes, the bug will leak the number information. If the query is used to distribute random passwords to users, it may send the same password to multiple different users. Since `SQLite` is widely used in public, line in 3.5 billion smartphones [51], we believe this deduplication bug has severe functional and security consequences.

2.2 Oracles for Logical Bug Detection

Logical bugs are more challenging to detect than memory-related issues. Once a memory bug is triggered, the execution will likely crash, and we can capture that easily. Most logical bugs do not crash the program, but just produce incorrect results. We need an oracle to provide the expected result. For example, in Listing 1, the oracle should indicate the expected result as `10`. However, constructing a complete, error-free oracle is challenging. Senior analysts can manually analyze results to identify bugs, but cannot handle large-scale test cases. Differential analysis provides an automatic, scalable solution [16,29,49,52]. For example, `RAGS` sends one query to different DBMSs and compares their results to find bugs [49]. However, due to the diverse dialects and extensions, popular DBMSs only share a small portion of features, and cross-DBMS validation cannot detect DBMS-specific bugs [49].

Recent works construct functionally equivalent queries and check whether the DBMS produces the same result for all queries [43–45]. For example, for a given query, oracle `NoREC` shifts all conditions from `WHERE` clauses to `SELECT` expressions, which effectively disables most optimizations applied to the original query [43]. The number of `TURE` rows for the modified query should be the same as the number of rows for the original one. Oracle `TLP` splits a condition `x` in a `WHERE` clause into three subqueries: `x IS TRUE`, `x IS FALSE`, and `x IS NULL`. It combines the results from three subqueries and checks the equivalence with the original one [44]. As oracles are built on high-level semantics instead of low-level implementations, we can use them to test multiple DBMS systems [30].

We can define a simple oracle, dubbed `Index`, to detect the bug in Listing 1: *removing indexes from the database should not affect the query result*. We can implement oracle `Index` to delete existing indexes (or inserts new ones) and check whether the DBMS produces the same result as before. One functionally equivalent form of Listing 1 is given in Listing 2 by deleting the `CREATE UNIQUE INDEX` statement. We can use oracle `Index` to detect this bug without any domain knowledge. More importantly, we can extend the test to a large scale.

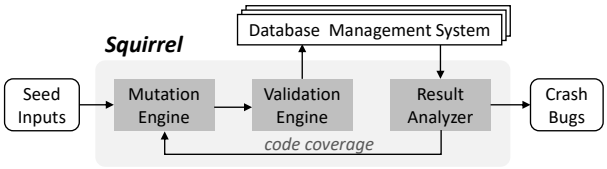


Figure 1: Overview of `Squirrel` [71]. It utilizes the coverage-based guidance to test DBMS systems for crashes and assertion failures.

2.3 Coverage-guided Testing

Coverage-guided testing, or *fuzzing* [32], has been widely used to test a large set of programs and successfully found thousands of bugs [18, 27, 48, 70]. Modern fuzzing tools, called *fuzzers*, utilize code coverage to guide input selection and mutation. Specifically, given one program input, a fuzzer first randomly updates its content to generate a set of new inputs. It feeds new inputs to the program and monitors the execution. If the program crashes or reports assertion failures, the fuzzer treats the input as the proof-of-concept (PoC) of the underlying bug. For input that triggers no crashes, the fuzzer will check whether the execution reaches new code paths, like basic blocks or branches. If so, it will add the input into a queue. Otherwise, it will drop the input and pick up the next input from the queue for another round of fuzzing. Coverage-guided testing has been used to test various programs, including but not limited to operating systems [11, 22, 37, 63, 65], compilers [10, 19, 38], web browsers [17, 48, 66], document readers [12, 64], and even smart contracts [20, 35, 62].

Recent works also ported coverage-guided fuzzing to test DBMS systems [21, 26, 60, 71]. Figure 1 shows an overview of `Squirrel` [71], a recent work that aims to detect crashes and assertion failures from DBMS systems. `Squirrel` takes a set of inputs, *i.e.*, SQL queries, as the seed of fuzzing. It first translates the query into an intermediate representation (IR), which contains many structural information. Then, `Squirrel` takes three mutation methods to modify the query IR and create new ones, including node insertion, deletion and replacement. For each newly generated IR, `Squirrel` builds the data dependency graph between different operands (e.g., tables and columns) and fills the operands with randomly generated strings. After that, it translates the new IR back to queries and feeds it to DBMS. At last, `Squirrel` reports crashes and prioritizes code-revealing queries for the next round of fuzzing. `Squirrel` successfully found a set of crashes and assertion failures from commonly used DBMS systems such as `SQLite`.

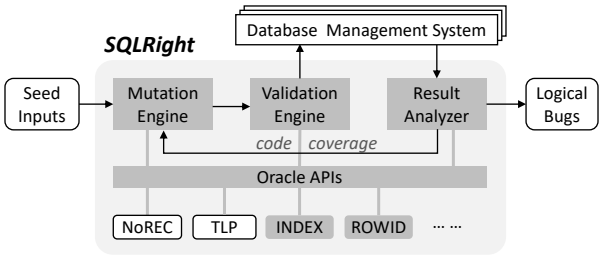


Figure 2: Overview of `SQLRight`. It utilizes the coverage-based guidance to test DBMS systems for logical bugs.

2.4 Challenges of Fuzzing Logical Bugs

To the best of our knowledge, no effort has ever tried to apply coverage-guided fuzzing to test DBMSs for logical bugs. There are mainly two challenges that hinder researchers from combining fuzzing and DBMS oracles. First, current fuzzers still cannot generate high-quality SQL queries. For example, in the most recent work, `Squirrel` embedded two new techniques, specifically syntax-preserving mutation and semantics-guided instantiation, aiming to improve the validity of the automatically generated queries. However, even with these advanced techniques, `Squirrel` can merely achieve around 30% validity for `SQLite`, and gets even worse validity rates for other DBMS systems. We can tolerate such low validity when testing DBMSs for crashes and assertion errors, as invalid queries may still trigger some bugs. However, a tool for finding logical bugs cannot make use of any invalid query, since the DBMS will not produce any meaningful results and the oracle cannot work. Second, current fuzzers mainly rely on operating systems and sanitizers to detect bugs (like assertion failures) [13, 28, 46, 47]. They cannot collaborate with DBMS oracles to detect logical bugs. We need to redesign the fuzzer architecture to support diverse DBMS oracles.

3 Design of `SQLRight`

We propose two practical solutions to address the aforementioned challenges and adopt the coverage-based guidance on testing DBMS for logical bugs. First, we provide the validity-oriented generation, which contains a set of strategies to produce valid, deterministic SQL queries (§3.1). The generated queries not only achieve high validity in syntax and semantics, but also exclude random behaviors which may cause false positives. Second, we design a set of general, comprehensive APIs to support developing new DBMS oracles (§3.2). These APIs decouple oracles and fuzzers, making it easier for users to test DBMS systems. They also assist DBMS developers to adopt coverage-based guidance for finding logical bugs.

System overview. Figure 2 shows an overview of our tool, `SQLRight`, the first testing platform that combines coverage-based guidance, validity-oriented mutations, and oracles to find logical bugs for DBMS systems. It accepts the target program (*i.e.*, the DBMS) and a set of sample queries (*i.e.*, SQL statements) as inputs and will produce reports of logical bugs. First, `SQLRight` adds all sample queries into a queue. For each round of fuzzing, it picks up one query from the queue and applies mutation to generate new queries. Then, it updates query operands, like table names and column names. After that, `SQLRight` sends the new query to the DBMS and inspects the execution result to identify unexpected behaviors. If the new query triggers new code, `SQLRight` adds the query into the queue for future testing. Unlike traditional memory-bug fuzzers (like `Squirrel` in Figure 1), `SQLRight` cooperates with DBMS oracles to produce high-quality queries to identify logical bugs. Specifically, for query mutation, it invokes oracle APIs to update oracle-specific SQL statements to prepare for result checking. For query validation, it again calls oracle APIs to translate the query into semantically equivalent variants. After the execution, it relies on the oracle to decide whether the queries trigger logical bugs.

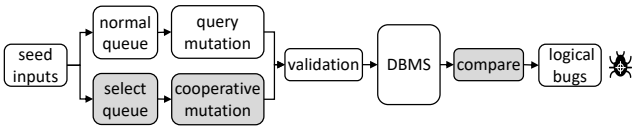


Figure 3: Cooperative mutation. To support the oracle to compare results, `SQLRight` adopts a dedicated select-queue and a cooperative mutation step to generate valid `SELECT` statements.

3.1 Validity-oriented Query Generation

`SQLRight` demands high-quality SQL queries to stress various aspects of different DBMSs. Any queries that cause syntax or semantic errors would not be useful for finding logical bugs. Unfortunately, generating semantically correct queries has proven to be NP-hard [29]. Recent fuzzers utilize type-based mutation and semantic-guided instantiation in order to produce valid queries [71]. However, their validity rate, about 30%, is still insufficient to effectively test DBMS systems. Even worse, most of the generated queries are not ready for detecting logical bugs. Therefore, we proposed several practical techniques to improve query validity.

3.1.1 Cooperative Mutation

`SQLRight` takes two separate mutation strategies in parallel to generate different components of a query set. As shown in Figure 3, it maintains two queues: the *select queue* contains only `SELECT` statements and is used to generate proper `SELECT` queries that produce outputs; the *normal queue* hosts other statements and is used to prepare the database for testing, like table creation and value insertion. During the fuzzing initialization, `SQLRight` scans all seed inputs, saves all `SELECT` statements to the select queue, and keeps other statements in the normal queue. For each fuzzing round, `SQLRight` collects a set of statements from the normal queue and relies on the normal mutation engine to generate new queries. Then, it

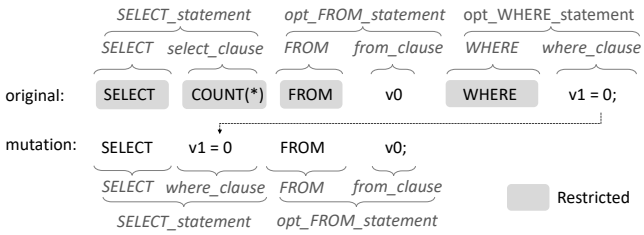


Figure 4: Cooperative mutation on `SELECT`. To support the `NoREC` oracle, we fix `SELECT_statement`, `FROM` and `WHERE` of the statement and leave others for the mutation engine to change.

invokes the cooperative mutation of the oracle to create and append several `SELECT` statements. After the combination, we leverage the instantiation to build a concrete query set.

During the cooperative mutation, the oracle preserves the query elements that are useful for the correctness test. For example, oracle `NoREC` requires the `SELECT` statement to have both `FROM` clause and `WHERE` clause. We provide interfaces for the oracle to notify the mutation engine not to delete these necessary nodes or add new nodes if the original statement does not have one. Figure 4 illustrates how to control mutations on the statement `SELECT COUNT(*) FROM v0 WHERE v1=0`. Given the IR of the query, we add attributes to three nodes, specifically, `SELECT_statment`, `FROM`, and `WHERE`, to mark them as immutable. As a result, the crucial components for oracle `NoREC` are preserved. The mutation engine still has full flexibility to update the `from_clause` node and the `where_clause` node to generate new `SELECT` statements. The IR-based mutation engine supports various patterns and different conditions in the `from_clause` and `where_clause`, and thus provides rich capability comparable to the unconstrained mutator [71].

3.1.2 Dedicated Parsing

We customize our mutation engine for each DBMS to improve syntax correctness. Most popular DBMS systems have their own customized SQL dialects, which only share limited features. Therefore, one unified SQL grammar for common functionalities only covers a small portion of DBMS code and cannot find bugs in DBMS-unique features [49]. In contrast, a catch-all grammar that aims to support all features could lead to many invalid queries. Listing 3 shows an example that a string with the single quote symbol has different meanings in different DBMSs. `SQLite` can run this query successfully, as it treats a single-quoted string as a constant by default or as an identifier if a string is not allowed in its current location. However, `PostgreSQL` reports an error for this query as it never accepts single-quoted strings as identifiers. To address the limitation, we design a tool to port the parser of each DBMS into `SQLRight` automatically. Our observation is that most popular DBMS systems use `GNU Bison` [14] to compile their parser front-ends. Therefore, we follow the grammar rules defined by `Bison` and translate the DBMS parser front-ends to `SQLRight`'s IR. Thanks to the well-documented `Bison` format,

```
01 CREATE TABLE v0 (v1 TEXT);
02 INSERT INTO v0 VALUES ('text');
03 SELECT v1 FROM 'v0';
```

Listing 3: Queries allowed in `SQLite` but rejected by `PostgreSQL`. `SQLite` takes flexible rules to identify strings and can treat the `'v0'` as an identifier, while `PostgreSQL` strictly takes `'v0'` as a string.

our tool can easily port different parsers for `SQLRight`. With the original DBMS parser front-ends, `SQLRight` can support the full dialect syntaxes for each DBMS and guarantees high syntax correctness to improve query validity.

3.1.3 Context-based IR Instantiation

The previous work `Squirrel` [71] identifies data dependency among all SQL statements to help instantiate query operands, like table names and column names. However, the dependency graph constructed by `Squirrel` tightly couples multiple SQL statements into one static graph. When dealing with complicated queries, it cannot update the data dependency to reflect the dynamic SQL context. Specifically, it keeps the static data dependency unchanged across the whole query sequence and cannot adjust itself to fit in the ever-changing relationship between different SQL statements. To address this limitation, we design a context-based IR instantiation algorithm. It dynamically updates the data dependency based on the SQL context and fills in accurate concrete values to the query skeletons. Instead of grouping multiple SQL statements into one dependency graph, `SQLRight` solves one SQL statement at a time. It only saves the necessary dependency information into the library when solving multiple SQL statements.

Listing 4 shows how `SQLRight` utilizes the context to instantiate SQL IR. To generate new queries, `SQLRight` ignores all existing operands and will assign them different names. The first statement creates one table with three columns. Since this is a creation operation, `SQLRight` allocates `v0` as the table name, `c1`, `c2`, and `c3` as the column names, and saves these names into the context. When we need tables and columns, `SQLRight` can quickly retrieve them from the current context. The second statement inserts one row into a table. `SQLRight` searches in the current context and finds only one table available, *i.e.*, `v0` with three columns. Therefore, it uses `v0` here and prepares one row with three values. The next statement changes one column name. After searching in the current context, `SQLRight` finds one table name `v0` and three associated column names `c1`, `c2` and `c3`. Based on the semantics, it uses `v0` as the table name, and randomly picks `c3` as the old column name. It also allocates another column name `c4` as the new name. After this statement, `SQLRight` updates the context to drop `c3` from `v0` and add `c4` instead. When it comes to the `SELECT` statement, it can find table name `v0` and three associated columns, excluding the old `c3`. In contrast, the original method of `Squirrel` will fail to capture the dynamic change by `ALTER` and may use `c3` as a column name of `v0` for `SELECT`.

```
-- Context-based IR Instantiation
CREATE TABLE v0 (c1 INT, c2 INT, c3 INT);
-- save v0 & mapping c1->v0, c2->v0, c3->v0.
INSERT INTO v0 VALUES (0, 0, 0);
-- Get v0 & 3 columns from v0.
ALTER TABLE v0 RENAME c3 TO c4;
-- drop c3->v0 & add c4->v0.
SELECT * FROM v0 WHERE c1 = c2;
----------------------------------------
-- Squirrel dependency graph
SELECT * FROM v0 WHERE c1 = c3;
-- Error: no such column: v0.c3
```

Listing 4: Example that demonstrates the benefits of SQLRight's Context-based IR Instantiation.

3.1.4 Non-determinism Mitigation

Several DBMS functionalities contain non-deterministic behaviors, which make two executions produce different results even when no logical bug is triggered. Such queries will confuse DBMS oracles and lead to false alarms. Previous fuzzers that focus on memory-related bugs do not care about the query results and just ignore this problem. Therefore, we cannot directly use their generated queries to find logical bugs. To avoid false alarms in this category, we identify and remove statements or keywords with non-deterministic behaviors.

Currently, SQLRight considers non-deterministic behaviors in three categories. One category contains functions returning random results by design. For example, random() in SQLite produces a pseudo-random integer number. Another category contains queries whose results depend on ever-changing environment variables, like date and time (e.g., julianday() in SQLite and current_timestamp in PostgreSQL). Listing 5 shows an example that makes the oracle report a false alarm. The example creates two tables, v0 with ROWID and v2 without ROWID. Based on SQLite document, the same query on these two tables should return the same result. However, due to the random behavior of random(), two executions produce different results, and the oracle will report a potential bug (false positive). Replacing random() with a constant value or deleting the INSERT statement can fix the problem.

The third source of non-determinism comes from undefined behaviors. Specifically, the result is not specified in the DBMS standard, and it completely depends on each dynamic execution to (randomly) decide how to generate the result. For example, the LIMIT clause cuts off the result of SELECT to at most N rows. It is designed to prevent a large number of outputs. However, the LIMIT clause leads to inconsistent results between semantically equivalent queries since it depends on the DBMS to decide which N rows to return. We remove LIMIT clauses from all generated SQL queries.

3.2 General Interfaces for DBMS Oracles

Oracles are crucial to detect logical bugs, but no oracle can detect all bugs from all DBMS systems. Considering the various SQL dialects and extensions [49], we need multiple diverse oracles to cover different logical bugs. It is essential

```
CREATE TABLE v0 (v1 PRIMARY KEY)
INSERT INTO v0 VALUES(random());
SELECT * FROM v0;
---------------------------------
CREATE TABLE v2 (v3 PRIMARY KEY) WITHOUT ROWID
INSERT INTO v2 VALUES(random());
SELECT * FROM v2;
```

Listing 5: False positive case due to non-deterministic behaviors. Function random() returns a random integer value, and thus running these two queries will produce different results.

to make our fuzzing platform support many different oracles. In this work, we propose a set of general APIs that allow developers to adopt existing oracles and develop new ones.

SQLRight provides four general APIs for the fuzzer to communicate with DBMS oracles, specifically, preprocess(), append_output(), transform() and compare(). We explain the purpose of each API using Table 1, which shows the currently supported oracles and the functionalities of their APIs. The Basic column defines the default behaviors of all oracles.

Preprocessing. preprocess() takes the query set as input and performs necessary operations to get it ready for future steps. One common task here is to check the applicability of the oracle on the given query set. If necessary, the oracle can modify the query set to produce compatible queries. For example, oracle Rowid tries to find CREATE TABLE statements from the query. If no tables are created, it will notify the fuzzer to skip the current query and move on to the next one. Oracle Index identifies and deletes CREATE UNIQUE INDEX statements from the query set. By design, a column with a unique index will not allow insertions of duplicated values, so having it or not will affect the final results and may lead to false alarms.

Appending Output Statements. Since we need to inspect the results of DBMS executions to identify bugs, SQLRight provides the append_output() API for each oracle to insert proper output-generation statements. The SELECT statement queries the underlying database and returns the results, so the default behavior of this API is to append several SELECT statements to the end of the given query set. During the query validation, these statements will be filled with proper table names and column names so that they can produce meaningful results. Oracles can have their special output statements or apply other policies based on their functionalities. For example, because oracle NoREC shifts conditions from WHERE to SELECT, it only appends SELECT statements that have a FROM clause and a WHERE clause (both clauses are optional for SELECT).

Transformation. The core task of an oracle is to transform one query into different equivalent variants. transform() accepts the query set as input and returns one or more equivalent query sets. For example, oracle NoREC returns only one variant where the WHERE conditions in the original query are moved to the SELECT expressions. Oracle Index may insert or delete various CREATE INDEX statements into the given query set and thus can result in multiple variants. SQLRight executes all variants returned by transform() and compares their results.

API	Basic	NoREC	TLP	Index	Rowid
`preprocess()`	check applicability	basic	basic	remove `UNIQUE INDEX`	basic
`attach_output()`	append random `SELECT`	`SELECT COUNT(*) FROM x WHERE x;`	basic	basic	basic
`transform()`	-	expr: `WHERE` → `SELECT`	∅ → `T` ∪ `F` ∪ `NULL`	insert `CREATE INDEX`	add `WITHOUT ROWID` to table
`compare()`	#rows = #rows	#rows = #(TRUE rows)	basic	basic	basic

Table 1: Oracles and their APIs. Our system currently supports four oracles: `NoREC`, `TLP`, `Index` and `Rowid`. `Basic` is the base interface that implements default behaviors. ∪ combines the results of different queries; - means no functionality; `basic` means the default behavior in `Basic`.

Result Comparison. `SQLRight` allows the oracle to define their own comparison methods to identify unexpected results. A simple comparison algorithm could require all results to be exactly the same. However, this method could be overly strict and may introduce false alarms. For example, in the results of the `Index` oracle, the order of rows could be different due to extra indexes. Therefore, we provide a loose comparison function that merely checks the number of rows in the results. However, if aggregate functions (like `MIN` and `SUM`) are used in `SELECT`, the comparison should expect the same output. We take special actions to handle oracle `NoREC`, as the new query form always generates one row for each record (`TRUE` or `FALSE` based on the condition). In this case, the `compare()` API should compare the number of rows in the original result and the number of `TRUE` rows in the transformed one.

We present the detailed steps that `SQLRight` takes to find the bug of Listing 1 using oracle `Index` in Appendix B.

4 Implementation

We implement `SQLRight` as a prototype of the first coverage-guided DBMS fuzzer for finding logical bugs. Our implementation is based on `Squirrel` [71] and `SQLancer` [30]. Specifically, we adopt the query-mutation module of `Squirrel` and implement our general oracle APIs and validity-oriented mutations. Currently, `SQLRight` supports four oracles, including `NoREC` and `TLP` ported from `SQLancer` [43, 44], and `Index` and `Rowid` we propose in this paper. We port the parsers of `SQLite`, `MySQL`, and `PostgreSQL` to `SQLRight` to test these DBMSs effectively. Next, we present several implementation details that could help the future development of other DBMS fuzzers.

Memory-efficient Mutation. We redesign the way of storing SQL queries to reduce memory use. `Squirrel` uses a library to map each node type, like `select-clause`, to a set of IR nodes. At the time of insertion or replacement, it visits the library to retrieve a set of type-matched candidates. Although keeping IR nodes in memory accelerates the mutation, each IR node occupies non-trivial memory, and the library will take hundreds of Gigabytes after testing for a few days. For example, `Squirrel` allocates more than 16KB to store the simple queries in Listing 1. To reduce the memory use, we store the query string instead of the IR in the memory and save the string pointers in the library. Since the string is much smaller (*e.g.*, 117B for Listing 1), one can run the fuzzer on low-resource platforms and easily launch a large number of instances. The tradeoff is that for every time we mutate the query, `SQLRight` needs to parse the string into IR again, which will slow down the mutation. The evaluation in §5.2 shows that `SQLRight` generates fewer queries than `Squirrel` within the same time period. However, thanks to the validity-oriented design, `SQLRight` achieves a higher speed in generating *valid* queries, which helps it outperform `Squirrel` on coverage.

Bug Bisecting. We adopt the bug bisecting method to identify duplicated bug reports [21]. For each bug report, we use the binary search algorithm to locate the commit that initially introduced the bug from all code commits. With the optimistic assumption that one commit introduces one bug, we use the first buggy commit to label the underly bug. If two queries share the same first buggy commit, we only report one of them and treat another as duplicated. We implement our bug bisector using Python, which takes the bug-triggering query as input and automatically locates the first buggy commit. When it reaches a new commit, the bisector will compile the new version and test it with the bug-triggering query. To make bisecting faster, we save every DBMS executable into a cache system and search in the cache to find built binary quickly. If all necessary versions are cached, our bisector can complete the task within one second. Note that `fossil` [41], the version control system for `SQLite`, provides the `bisect` command to assist bug bisecting. However, it only calculates the middle commit and updates the code accordingly. Our bisector will compile the code and verify the test result automatically.

Query Minimizer. Bug-triggering queries may contain many complicated statements, which are difficult for developers to diagnose. Therefore, we develop a utility that can automatically minimize the bug-triggering query. Our minimizer makes use of the IR delete operation of `SQLRight`, which removes one node from the query IR representation. After removing one node, if the remaining query is still valid and can trigger the bug, we will delete more nodes from the current statements; otherwise, we will add the node back and move on to delete the next node. The minimization algorithm keeps the deletion process until that removing any node will render the remaining part invalid or miss the bug. In that case, we will report the current query as the minimum version.

5 Evaluation

We evaluate our tool `SQLRight` on real-world popular DBMS systems to answer the following questions:

Q1. Can `SQLRight` detect real-world logical bugs? (§5.1)

ID	DBMS	Description	Oracle	D	Status	Fix
Logical Bugs						
1	SQLite	UNIQUE PARTIAL INDEX, DISTINCT	INDEX	5	fixed	c2f940b
2	SQLite	JOIN and LIKELY/UNLIKELY	NoREC	3	fixed	2363a14
3	SQLite	IN-early-out optimization	TLP	2	fixed	eb40248
4	SQLite	EXISTS (SELECT...) to IN	NoREC	2	fixed	16252d7
5	SQLite	Misuse aggregate in ORDER BY	NoREC	2	fixed	0d11d77
6	SQLite	Aggregate in ORDER BY	NoREC	1	confirmed	-
7	SQLite	WITHOUT ROWID with DESC	NoREC	7	fixed	f65c929
8	SQLite	unordered WITHOUT ROWID table	ROWID	2	fixed	c21bc5a
9	SQLite	mixed table name and CTE	NoREC	2	fixed	0f0959c
10	SQLite	inconsistent constant propagaion	NoREC	1	fixed	9be208a
11	SQLite	IS NOT NULL optimization	NoREC	2	fixed	8cc2393
12	SQLite	large value loss of precision	NoREC	4	fixed	f9c6426
13	SQLite	constraint check in ALTER TABLE	NoREC	2	fixed	e379499
14	SQLite	equivalence transfer optimization	NoREC	2	fixed	8b24c17
15	MySQL	ANY ALL optimization error	NoREC	5	confirmed	-
16	MySQL	UNIQUE INDEX with NULL	NoREC	1	confirmed	-
17	MySQL	GTID_SUBSET	NoREC	4	fixed	8.0.30
18	MySQL	large value in GROUP BY	TLP	3	confirmed	-
Crashes						
1	SQLite	distinct aggregation	-	4	fixed	0e47898
2	SQLite	ROWID from views	-	2	fixed	0f0959c
3	SQLite	nested CTE	-	5	fixed	94225d6
4	SQLite	foreign_key_check	-	2	fixed	68db1ff
5	SQLite	UPDATE from VIRTUAL TABLE	-	3	fixed	2547cfe
Assertion Failure						
1	SQLite	pLeft==pRight	-	3	fixed	240f749
2	SQLite	iColnCol	-	4	fixed	b986600
3	SQLite	memIsValid(&aMem[pOp->p3])	-	2	fixed	c9f0b9c
4	SQLite	target>0&&target<=pParse->nMem	-	2	fixed	7072404

Table 2: Detected Bugs. SQLRight detected 27 bugs, including 18 logical bugs, 5 crashes and 4 assertion failures. We have reported these bugs to their developers and got 27 confirmed and 23 fixed. **D** (depth) means the number of mutations to trigger the bug.

Q2. Can SQLRight find more bugs than existing tools? (§5.2)
Q3. How does code coverage guide the testing? (§5.3)
Q4. How does query validity help detect bugs? (§5.4)

Experimental Setup. To answer **Q1**, we use SQLRight to test three popular DBMS systems, SQLite, MySQL and PostgreSQL, which are commonly used in previous works to evaluate bug-finding tools for DBMSs [21,43–45,71]. For **Q2**, we compare SQLRight with SQLancer and Squirrel, the state-of-the-art bug-finding tools for DBMSs. Since Squirrel cannot detect logical bugs, we port our oracles to Squirrel, denoted as Squirrel$_{+oracle}$. To answer **Q3**, we compare coverage feedback with three methods to handle new queries: dropping all, saving all and randomly saving some. To answer **Q4**, we disable each query-validity component of SQLRight to find the impact on code coverage, query validity and bug detection.

We conduct SQLite experiments on a Ubuntu 20.04 system with two 28-cores Intel(R) Xeon(R) Gold 6258R CPUs and 791 GB memory. We perform the experiments of PostgreSQL and MySQL on three computers with Ubuntu 20.04 system, 8-cores Intel(R) Core(TM) i7-10700 CPU, and 64 GB memory. Since SQLancer requires the particular SQLite version 3.34.0, we adopt this version for evaluation. For other DBMSs, we choose the latest version, specifically PostgreSQL 14.0 and MySQL 8.0.27. We compile SQLite and PostgreSQL using AFL [70] but enlarge the coverage map size to 256K to mitigate the collision issue [15]. Since MySQL is multi-threaded, to avoid data race on updating the coverage map, we instrument it with block coverage and disable hit counts. We collect the seed corpus from the official unit tests of each DBMS, specifically, SQLite TCL Test Scripts, PostgreSQL official test infrastructure, and MySQL unit test samples. We use the same seed corpus for SQLRight and Squirrel. SQLancer is a generation-based tool and does not require any seed input.

5.1 DBMS Logical Bugs

Due to the limited availability of resources, we conduct bug-finding experiments for each DBMS with different durations. Specifically, we spend 60 days testing SQLite, 14 days testing PostgreSQL and 7 days testing MySQL. In total, SQLRight detects 27 bugs, including 18 logical bugs, 5 crashes, and 4 assertion failures. Except for 3 logical bugs from MySQL, all other bugs are from SQLite; we do not find any bugs from PostgreSQL. Despite the different testing durations, we can find the similar pattern of bug numbers from previous works [43–45, 71], where most bugs are from SQLite, and very few are from PostgreSQL. We have reported all findings to their developers. All bugs have been confirmed, and 23 of them have been fixed. SQLRight detects more logical bugs than crashes and assertion failures. Since we design a set of solutions to improve the query validity, SQLRight generates fewer abnormal statements that render the DBMS crashing.

Table 2 provides more bug details. Column "Description" shows that logical bugs come from diverse optimizations, and many bugs are due to an interplay of multiple optimizations. For example, the first bug is caused by an improper DISTINCT optimization when the table has a unique partial index: DISTINCT and UNIQUE PARTIAL INDEX work well alone, but the bug appears when a query satisfies both conditions. Column "Oracle" presents the oracle that detects each bug. Most bugs are detected by NoREC, and other oracles detect one or two bugs. In fact, several bugs can be detected by multiple oracles through different queries, but we only record the first detection and treat others as duplications.

We present the details of two logical bugs to help demonstrate the necessity of feedback and validity for bug finding. Appendix A presents the details of three more logical bugs.

Necessity of Coverage-based Feedback. Listing 6 demonstrates a SQLite logical bug due to incorrect index lookup. The first statement creates a WITHOUT ROWID table with two columns. v1 is the PRIMARY KEY of the table and is in DESC order. The second statement inserts one row to v0, and the third statement creates index v3 on column v2. The SELECT statement searches for rows that satisfy conditions v2=10 and v1<11. Row (10,10) satisfies the condition and should be returned. However, due to an incorrect index search that mixes up the DESC PRIMARY KEY and index v3, SQLite does not return anything. At the end of Listing 6 we show the related seed input that is mutated to trigger the bug. As we can see, the original input lacks most of the necessary components to trigger the bug, like WITHOUT ROWID, DESC, INDEX, and SELECT.

```
CREATE TABLE v0 (v1 INT PRIMARY KEY DESC,
   v2 INT) WITHOUT ROWID;
INSERT INTO v0 VALUES (10, 10);
CREATE INDEX v3 ON v0 (v2);
SELECT * FROM v0 WHERE v2 = 10 AND v1 < 11;
-- output:  (empty)
-- expect:  10|10
-- original seeds -------
CREATE TABLE v0 (v1 INT PRIMARY KEY, v2 INT);
INSERT INTO v0 VALUES (10, 10);
```

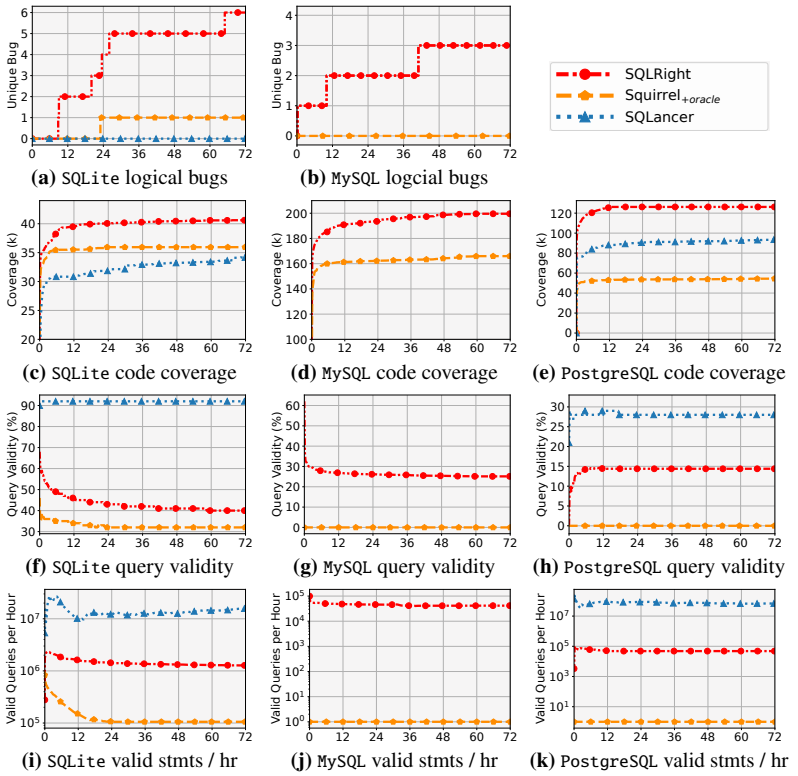
Listing 6: A logical bug of SQLite that mutes outputs because of incorrect index lookup.

```
CREATE TABLE v0 (v1, v2 PRIMARY KEY);
CREATE INDEX v3 ON v0 (v2, v2);
INSERT INTO v0 (v1, v2) VALUES (10, 'x');
SELECT * FROM v0 AS a13, v0 AS a14
   WHERE a13.v1 = a13.v2 AND a13.v1 = 'x';
-- output: 10|x|10|x
-- expect: (empty)
SELECT * FROM v0 AS a13, v0 AS a14
   WHERE v1 = v2 AND v1 = 'x';
-- output: Error: ambiguous column name: v1
```

Listing 7: A logical bug of SQLite related to column aliases. SELECT statements should return nothing

Oracle	Feedback	Mutation Depth				Max Depth
		0	1 ∼ 3	4 ∼ 7	7+	
NoREC	SQLRight	24738	9761	3125	2443	25
	$\text{SQLRight}_{\text{drop}}$	24738	10478	-	-	1
	$\text{SQLRight}_{\text{rand}}$	24738	7753	2561	-	7
	$\text{SQLRight}_{\text{save}}$	24738	10030	356	-	4
TLP	SQLRight	24738	9662	3594	1580	21
	$\text{SQLRight}_{\text{drop}}$	24738	10294	-	-	1
	$\text{SQLRight}_{\text{rand}}$	24738	7217	881	-	7
	$\text{SQLRight}_{\text{save}}$	24738	8916	-	-	3

Table 3: Code coverage triggered by queries with different depths. SQLRight maintains higher depth.



(a) SQLite logical bugs (b) MySQL logcial bugs (c) SQLite code coverage (d) MySQL code coverage (e) PostgreSQL code coverage (f) SQLite query validity (g) MySQL query validity (h) PostgreSQL query validity (i) SQLite valid stmts / hr (j) MySQL valid stmts / hr (k) PostgreSQL valid stmts / hr

Figure 5: Comparison between different tools (NoREC). a-k show the number of unique bugs, code coverage, query validity and validity over time. SQLancer does not implement NoREC for MySQL, so we skip the evaluation. We run each fuzzing instance for 72 hours for and repeat five times.

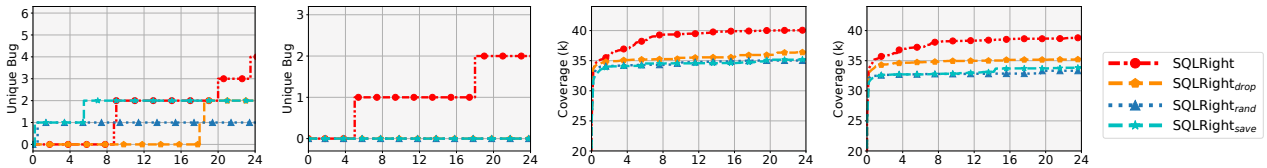
From the seed input, SQLRight must accumulate seven consecutive mutations to generate this bug-triggering query. Each mutation triggers new code coverage and is saved to the queue. Without the guidance of code coverage, SQLRight can hardly maintain such a deep mutation chain, nor detect this bug.

Necessity of SQL Validity. With validity-improvement techniques, SQLRight can detect bugs that $\text{Squirrel}_{+\text{oracle}}$ and SQLancer will miss. Listing 7 shows one logical bug that can only be detected with the per-DBMS parser and the context-based instantiation. The first statement creates table v0 with two columns: v1 and v2. The second line creates an index attached to two v2. The third statement inserts one row into the table. Two SELECT statements search in v0 to find rows where v1=v2 and v1='x'. These conditions are not satisfied, as v1 and v2 have different values. However, if we refer table v0 using aliases, SQLite will incorrectly optimize the query to ignore the first condition and return the row. SQLancer cannot detect this bug since it does not use aliases when generating SELECT statements. Although $\text{Squirrel}_{+\text{oracle}}$ allows alias during parsing, it does not use it for instantiation. If the table has aliases, $\text{Squirrel}_{+\text{oracle}}$ will resolve the condition to v1=v2, resulting in an error of "ambiguous column name". SQLRight adopts context-based instantiation, which avoids the collision problem and successfully triggers this bug.

5.2 Comparison with Existing Tools

SQLancer contains three oracles, NoREC, TLP, and PQS. Both NoREC and TLP just modify query statements to detect bugs, and SQLRight can easily support them using our general APIs. However, PQS relies on database content to construct statements, and we consider supporting it in the future. Therefore, we use NoREC and TLP to compare SQLRight, SQLancer, and $\text{Squirrel}_{+\text{oracle}}$. We launch five instances in each setting and run them for 72 hours. Since SQLancer does not support NoREC for MySQL, we skip this setting. Figure 5 (NoREC) and Figure 8 (TLP) in Appendix show the evaluation results.

Unique Logical Bugs. Based on Figure 5a b and Figure 8a b, SQLRight reports the most 12 bugs across all settings, including 6 SQLite bugs and 3 MySQL bugs using NoREC, and 2 SQLite bugs and 1 MySQL bug using TLP. $\text{Squirrel}_{+\text{oracle}}$ merely found 1 bug, which is from SQLite using NoREC; SQLancer did not find any logical bug. The empty outcome from SQLancer could be due to its extensive use in testing these DBMS systems [43–45]. The generation-based method has reached its limit on finding more bugs in its supported mutations. This demonstrates the necessity of bringing more diverse mechanisms to detect logical bugs. SQLRight combines code coverage, oracles, and query validity, and can detect significantly more logical bugs than SQLite and MySQL.



(a) bugs of SQLite (NoREC) **(b)** bugs of SQLite (TLP) **(c)** coverage of SQLite (NoREC) **(d)** coverage of SQLite (TLP)

Figure 6: Contribution of code-coverage to DBMS fuzzing. We launch five instances for each tool and run each instance for 24 hours. a and b show the number of unique SQLite bugs found by NoREC and TLP; c and d show the average number of SQLite edges using NoREC and TLP.

All tools fail to find any bug in PostgreSQL, indicating its good code quality observed by previous works [43–45, 71].

Code Coverage. According to Figure 5c d e and Figure 8c d e, SQLRight triggers noticeably higher code coverage than others on three DBMS systems. Considering that SQLancer is a generation-based tester and has a high validity rate, the extra coverage of SQLRight over SQLancer mainly comes from the coverage-based guidance and full-featured per-DBMS parsers. Although Squirrel$_{+oracle}$ also utilizes code coverage to guide testing, the accurate parser and context-based instantiation help SQLRight outperform this state-of-the-art coverage-based DBMS tester. For example, given the same seed corpus, SQLRight can correctly parse more queries and achieves higher code coverage even at the beginning. With the higher coverage, SQLRight is able to find more bugs.

Query Validity. From Figure 5f g h and Figure 8f g h we can find that SQLancer achieves the highest query validity, where over 80% of all queries are valid for SQLite, 99% for MySQL, and over 28% for PostgreSQL. Our tool SQLRight achieves around 30%, 25%, 10% validity for three tested DBMSs. This result is reasonable since SQLancer follows well-defined rules to generate SQL statements, leading to more valid queries. Nevertheless, SQLRight achieves significantly higher validity than the Squirrel$_{+oracle}$, which can hardly produce any useful queries for oracles. This result confirms the necessity to improve the validity of detecting logical bugs. Another observation is that SQLancer can keep a consistent high validity rate for 72 hours. Mutation-based fuzzers, regardless of their configurations, get a lower validity rate as time goes on. The reason is that during fuzzing, some invalid queries trigger new code coverage (*e.g.*, error-handling code) and are added to the queue. Mutating invalid queries will likely produce more invalid ones, which turns the overall validity rate down. We also measure the speed of generating valid queries and get the same pattern, shown in Figure 5i j k and Figure 8i j k. SQLancer can generate more valid queries than others, but its efficiency varies over time.

Overall, SQLRight can find more logical bugs than SQLancer and Squirrel$_{+oracle}$. It also outperforms existing tools in triggering more program code. Although SQLancer produces high-quality queries, its lack of query diversity makes it less effective in finding new logical bugs.

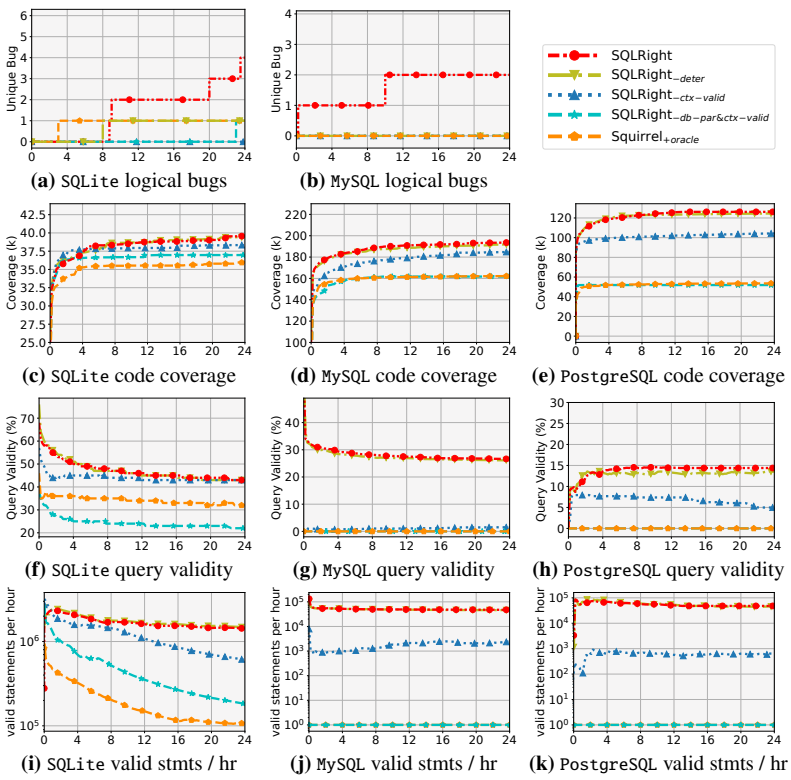
5.3 Contribution of Coverage Feedback

We compare SQLRight with three different feedback methods: SQLRight$_{drop}$ drops all generated queries and only mutates seed inputs; SQLRight$_{save}$ saves all generated queries (no matter whether they trigger new coverage or not) into the queue and takes turns to mutate each query; SQLRight$_{rand}$ randomly saves 10% of all generated queries to the queue. We perform the unit tests using SQLite with NoREC and TLP. For each setting, we ran five instances in parallel for 24 hours. Figure 6 shows the average results of code coverage and logical bugs. We also measure the contribution of queries in different depths. Specifically, if a query is generated from seed inputs through N mutations, we define its *depth* as N. Table 3 shows the breakdown of code coverage regarding query depth.

Unique Logical Bugs. Figure 6a and b show the number of detected bugs. For both NoREC and TLP, SQLRight reports the most bugs, including 4 bugs using NoREC and 2 bugs using TLP. Using NoREC, SQLRight$_{drop}$ and SQLRight$_{save}$ report 2 bugs, and SQLRight$_{rand}$ only detects 1. Without code coverage, SQLRight did not find any logical bug using TLP.

Code Coverage. According to Figure 6c and d, SQLRight achieves the highest code coverage: 40.1K branches with NoREC and 39.2K branches with TLP. SQLRight$_{drop}$ detects 35.2K branches using NoREC and 35.0K branches using TLP, which are 13.7% and 11.9% fewer than SQLRight. SQLRight$_{rand}$ and SQLRight$_{save}$ trigger even fewer branches than SQLRight$_{drop}$. The result shows the benefit of coverage-based guidance in generating more diverse queries.

Mutation Depth. Table 3 shows that SQLRight maintains deeper mutation chains, where SQLRight can accumulate 21 mutations (column "Max Depth") while others at most accumulate 7 mutations. High-depth queries help SQLRight trigger more code coverage. In NoREC, besides the seed-triggered code (depth 0), 14.9% of new coverage is triggered by queries with 4-7 depth, and 17.0% is triggered by queries with 8 or more mutations; in TLP, the contributions of high-depth queries are 18.4% and 11.4%, respectively. SQLRight$_{drop}$ only mutates seed inputs to produce 1-depth queries. Although it gets more coverage on depth 1, SQLRight finally wins thanks to high-depth queries. SQLRight$_{rand}$ and SQLRight$_{save}$ can accumulate some mutations. However, without the coverage-based guidance, the accumulation is slow and falls behind SQLRight. Accumulated mutations also help detect more bugs. Column



(a) SQLite logical bugs **(b)** MySQL logical bugs **(c)** SQLite code coverage **(d)** MySQL code coverage **(e)** PostgreSQL code coverage **(f)** SQLite query validity **(g)** MySQL query validity **(h)** PostgreSQL query validity **(i)** SQLite valid stmts / hr **(j)** MySQL valid stmts / hr **(k)** PostgreSQL valid stmts / hr

Figure 7: Unit Tests for SQLRight validity components (NoREC). a and b show the number of unique logical bugs. c-k show code coverage, query validity and valid statements over time for each instance. We run each instance for 24 hours, repeat 5 times and report the average results.

```
CREATE TABLE v0 (v1 TEXT);
INSERT INTO v0 VALUES ('0');
CREATE VIEW v2 (v3) AS SELECT v1 FROM v0
            UNION ALL SELECT v1=10 FROM v0;
SELECT       *  FROM v2 NATURAL JOIN v2;
SELECT COUNT(*) FROM v2 NATURAL JOIN v2;
-- output of 1st SELECT: 0\n0\n0\n0
-- output of 2nd SELECT: 2
```

Listing 8: A false positive case related to view affinity (SQLite). The affinity of v3 is indeterminate.

```
CREATE TABLE v0 (c1, c2);
INSERT INTO v0 VALUES (NULL, 10);
INSERT INTO v0 VALUES (NULL, NULL);
CREATE VIEW v3 (c4, c5) AS SELECT
  MIN (c1), c2 FROM v0 WHERE c1 IS NULL;
SELECT COUNT(*) FROM v3 WHERE c5;
SELECT COUNT(c5) FROM v3;
-- output of 1st SELECT: 0
-- output of 2nd SELECT: 1
```

Listing 9: A false positive due to unspecified subquery ordering (SQLite).

```
SET SESSION sql_mode = sys.list_drop(
    @@session.sql_mode,'ONLY_FULL_GROUP_BY');
CREATE TABLE v0(c1 INT);
INSERT INTO v0 VALUES (1), (1), (3);
SELECT c1 FROM v0 GROUP BY NULL;
-- output: 1
SELECT c1 FROM v0 WHERE c1=1 GROUP BY NULL
UNION SELECT c1 FROM v0 WHERE NOT c1=1
                         GROUP BY NULL
UNION SELECT c1 FROM v0 WHERE c1=1 IS NULL
                         GROUP BY NULL;
-- output: 1\n3
```

Listing 10: A false positive case related to GROUP BY NULL (MySQL). If a SELECT statement contains GROUP BY NULL, MySQL would always return one matching row.

"D" of Table 2 shows the depth of bug-triggering queries. Among 18 logical bugs, only 3 can be triggered with 1 mutation, while others require up to 7 mutations. Crashes and assertion failures also require multiple mutations, demonstrating the consistent strength of coverage-based feedback.

> Coverage-based guidance helps generate more diverse queries and accumulate useful mutations, which helps discover more bugs than the no-feedback baselines.

5.4 Contribution of Validity

We evaluate validity-improvement techniques introduced in §3.1, including cooperative mutation, DBMS-specific parser, context-based instantiation, and non-determinism mitigation. SQLRight$_{\text{-deter}}$ disables the non-determinism mitigation techniques. SQLRight$_{\text{-ctx-valid}}$ disables the context-based instantiation and uses the dependency graph-based validation from Squirrel. Based on SQLRight$_{\text{-ctx-valid}}$, SQLRight$_{\text{-db-par\&ctx-valid}}$ further reuses the SQL parser of Squirrel. Squirrel$_{\text{+oracle}}$ does not adopt any validity-improvement techniques. We conduct unit tests on three DBMSs with two oracles. We evaluate each setup with five instances and run them for 24 hours. Figure 7 (NoREC) and Figure 9 (TLP) in Appendix show the results.

Unique Logical Bugs. SQLRight triggers the most bugs, shown in Figure 7a b and Figure 9a b, including 4 bugs in SQLite using NoREC, 2 bugs in SQLite using TLP, 2 bugs in MySQL using NoREC, and 1 bug in MySQL using TLP. SQLRight$_{\text{-db-par\&ctx-valid}}$ and Squirrel$_{\text{+oracle}}$ merely find 1 bug using NoREC. Other settings find no bugs.

Code Coverage. According to Figure 7c d e and Figure 9c d e, SQLRight dominates the code coverage number across all unit tests. The results show that the validity techniques help generate more diverse queries and explore more code. Among all validity-improvement techniques, the per-DBMS parser contributes the most to code coverage. Because the parser understands different SQL dialects, it can explore custom features specific to each DBMS. These features are not reachable through Squirrel, leading to the significant coverage gap between SQLRight$_{\text{-ctx-valid}}$ and SQLRight$_{\text{-db-par\&ctx-valid}}$.

Query Validity. The per-DBMS parser again plays the most important role in query validity. When testing PostgreSQL and MySQL, parsers from Squirrel cannot generate any useful queries. First, Squirrel parsers do not support custom functions in SELECT, including function COUNT, which is necessary to generate NoREC-compatible queries. Second, these parsers do not support UNION ALL, which is the key component to implement compatible queries for TLP. Among three DBMSs, SQLRight has the lowest validity for PostgreSQL. Since

DBMS	Oracle	FPs		Main Reason
SQLite	NoREC	SQLRight:	19	VIEW affinity, subquery ordering
		SQLRight$_{\text{-deter}}$:	42	randomblob and julianday() func
SQLite	TLP	SQLRight:	8	subquery ordering
		SQLRight$_{\text{-deter}}$:	53	randomblob func
MySQL	NoREC	SQLRight:	0	-
		SQLRight$_{\text{-deter}}$:	18	rand func, auto insert TRIGGERs
MySQL	TLP	SQLRight:	10	GROUP BY NULL
		SQLRight$_{\text{-deter}}$:	64	rand func, auto insert TRIGGERs

Table 4: False Positives by Non-deterministic Behaviors.

PostgreSQL enforces the most rigorous syntax and semantic rules, it is hard to generate valid SQL queries [43–45, 71].

False Positives due to Non-determinism. Table 4 shows the false positive numbers after bisecting, and Figure 7 and Figure 9 present the code coverage and validity of SQLRight$_{\text{-deter}}$. SQLRight$_{\text{-deter}}$ shares similar code coverage and validity with SQLRight, but it produces 95 false positives in SQLite and 82 false positives in MySQL. Most false positives are due to random functions, like randomblob, julianday, and so on. By disabling all non-deterministic behaviors, SQLRight still introduces a small number of false positives. They are mainly due to the special semantics in each DBMS, which we will provide more details in §6. We do not observe false positives from PostgreSQL, even with SQLRight$_{\text{-deter}}$. The reason is that the seed corpus contains no random behaviors.

The validity-oriented optimizations in SQLRight can help generate higher validity queries, reduce false positives, and ultimately help discover more bugs.

6 Discussion

In this section, We first discuss the common reasons for false positives (FP) and invalid queries of SQLRight. Then, we present the importance of seed inputs for finding logical bugs.

FP1: View Affinity in SQLite. We find several false positives related to the data affinity issue in View of SQLite. Listing 8 presents one example that triggers the false alarm. We expect that the number of rows returned by line 5 is the same as the result of line 6 (likely, 4). However, the first SELECT returns four rows while the second returns 2. SQLite developers explain that this inconsistency is due to the undefined affinity. Specifically, SQLite assigns each column an affinity (similar to data type) and uses different comparison algorithms for each affinity. In this example, the affinity of v2.v3 is indeterminate due to different affinities from two sources: SELECT v1 FROM v0 returns a column with affinity TEXT, while SELECT v1=10 FROM v0 returns a column with affinity NONE. Therefore, in the last two SELECTs, SQLite is free to choose any affinity to compare v3. In line 5, SQLite uses affinity TEXT and returns four rows, while in line 6, SQLite chooses affinity BLOB and only finds two matched rows. We find that the affinity issue is commonly reported by third-party tools as "bugs", including SQLancer [30]. Although SQLite documents have clearly explained this problem [50], we suspect that DBMS administrators may still miss it and create ambiguous queries.

FP2: Ordering in SQLite Subqueries. Listing 9 shows one false positive due to the undefined ordering in subqueries. This query first creates table v0 with columns c1 and c2 and inserts rows (NULL,10) and (NULL,NULL). Then, it creates view v3 to access particular rows in v0. If a row in v0 satisfies condition c1 IS NULL, the minimum c1 will be assigned to c4, and c2 will be assigned to c5. Both SELECT statements count c5, but they return different results. This problem is due to the undefined ordering in SQLite subqueries. The VIEW creation statement uses an aggregate function MIN in the subquery. To keep the row number consistent, SQLite will return one row of c2 and assign it to c5. However, which row of c2 to return is not defined, *i.e.*, it can be either 10 or NULL since both rows in v0 satisfy the condition c1 IS NULL. If NULL is returned, the counting result will be 0; if 10 is used, the result will be 1.

FP3: GROUP BY NULL in MySQL. Listing 10 shows a common false positive in MySQL. This false positive is reproducible when we turn off ONLY_FULL_GROUP_BY from sql_mode. The sample query first creates table v0 and inserts rows (1),(1),(3). It then uses oracle TLP to check the rows of v0. The returned results from two SELECTs are different, triggering TLP to report a potential bug. The difference is due to GROUP BY NULL, which forces SELECT to return one row regardless of really matched rows. The first SELECT has one SELECT clause and thus returns one row, while the second SELECT has three SELECT clauses and returns two rows. This issue can also be triggered if an SQL expression in the GROUP BY clause returns NULL. Therefore, it is easy to trigger this false positive unless we manually exclude expressions from GROUP BY. Fortunately, due to the small number of false positives, we can easily identify and ignore such false alarms.

Examples for invalid queries. Although SQLRight uses several techniques to improve validity, it cannot guarantee 100% correctness. Listing 11 shows a set of invalid queries SQLRight generates for PostgreSQL. The queries first create a table with the custom type circle, insert one circle value and then try to print out the table after reordering its rows. The reordering operation will fail since PostgreSQL does not have a compare method for circle. To fix this program, SQLRight must understand PostgreSQL geometric types and their related functions. It should generate queries to compare the radius, diameter, area, or other geometric characteristics from circle. The last SELECT shows one of correct queries. Due to the large number of custom data types and their related functions, we consider supporting them in the future.

Importance of Seed Corpus. During the evaluation, we notice that seed inputs are important for SQLRight to find logical bugs. SQLRight relies on random mutations to generate new queries, while the seed inputs provide all elements for mutation. If the seed inputs cover a broad range of functionalities, SQLRight can generate more diverse testing queries, and thus can test more aspects of DBMSs. To guarantee the quality of seed inputs, we gather queries from various unit tests of

```
CREATE TABLE v0 (c1 circle);
-- circle formatting '(x, y), r'
INSERT INTO v0 VALUES ('(1, 2), 3');
SELECT v0.c1 FROM v0 ORDER BY c1;
-- outputs: ERROR: Could not identify an ordering
--          operator for type circle at character 33
SELECT v0.c1 FROM v0 ORDER BY area(c1);
-- outputs: c1 = "<(1,2),3>"
```

Listing 11: Invalid SQL queries for PostgreSQL, generated by SQLRight. PostgreSQL defines 'circle' as a build-in data type. However, 'circle' type doesn't support compare operators by default.

each tested DBMS. For example, we collected over 160 seed inputs for SQLite from the TCL Tests script, covering various SQLite functions such as WHERE, ROWID and IN. TCL Test is the built-in test tool for SQLite. Its code is available in the SQLite GitHub repository, and it contains 1,272 test files and more than 46,000 unique test cases. These TCL Test scripts not only contain previously discovered bugs but also cover many SQLite-specific functionalities such as ROWID. We plan to introduce more high-quality and diverse seed inputs to SQLRight in our future work.

7 Related Work

In this section, we discuss the recent testing approaches that are related to SQLRight, focusing on DBMS testing.

Detecting Logic Bugs in DBMSs. Differential testing [29] is commonly used to detect logical bugs from DBMS systems [43–45, 49, 61, 68]. One direction is to run one query with different DBMS systems [16, 49, 52] and check the result consistency. Another direction is to run one query with different settings of one DBMS, like versions or optimization levels [67, 68]. SQLancer proposes the third direction, which constructs functionally equivalent queries to test one DBMS [43–45]. This method requires a deep understanding of SQL and DBMS [30]. SQLRight adopts the third method to generate equivalent queries. It also uses random mutation to diversify the query and relies on coverage-based guidance to help explore the program states. Our evaluation demonstrates the benefits of code coverage for finding logical bugs.

Detecting Performance Bugs in DBMS. Performance is an important metric of DBMS systems, and many test efforts aim to pinpoint the performance bottleneck [25]. BmPad [40] estimates execution times for a set of queries and checks whether real execution times are expected. AMOEBA [26] constructs function-equivalent queries and checks whether a DBMS can handle them in similar times. Apollo detects performance regression bugs, where the new version runs slower than the old one [21]. A common challenge is to determine whether a performance downgrade is a bug, or just a design choice. As each DBMS supports many features, developers tend to optimize the system for commonly used queries. Such optimization could slow down rarely used statements. Therefore, it requires extra efforts from DBMS developers to inspect and confirm performance bugs. SQLRight focuses on logical bugs, which usually have a strict policy on result consistency. Most reported bugs have been fixed immediately after our reports.

Detecting Crash Bugs in DBMS. Most testing efforts have been spent on detecting program crashes through two methods, either generation-based method or mutation-based. The generation-based method takes well-defined rules to synthesize valid DBMS queries [33, 58, 59]. Since generating a completely valid DBMS query is NP-complete [29], most generators focus on syntax validity and take practical ways to improve semantic correctness. SQLsmith [58] is highly customized and effective for PostgreSQL [71]. It generates constrained queries based on the understanding of the underlying database schema. Several works reduce the query generation problem to the satisfying constraints [1, 31] and use SAT solvers to provide valid queries [2]. Differently, Chandra *et al*. [7] propose to generate a proper database to boost the DBMS testing. SQLRight is different from these mutators in two ways. First, it does not require any predefined database, but creates one for each query. Second, it utilizes code coverage to highlight promising queries for testing.

Several fuzzers have shown their capability to test DBMS systems [3, 4, 8, 9, 24, 27, 36, 53, 69, 70]. Grimoire utilizes grammar-like combinations to synthesize highly structured inputs [6]. Bati *et al*. [5] proposed an engine that inserts or removes grammar components from existing SQL statements. However, due to the strict grammar and semantic requirements, most tools cannot explore deep logic of DBMS, like query planning and execution. The most recent work Squirrel translates SQL strings to an IR-based representation, and relies on the IR type to increase the validity of query generation [71]. SQLRight adopts the type-guided mutation of Squirrel, and takes one step further to cooperate with DBMS oracles to produce high-quality, oracle-compatible SQL statements. More importantly, SQLRight focuses on detecting logical bugs other than crashes and assertion failures.

8 Conclusion

We design the first tool, SQLRight, that combines code coverage, validity-oriented mutations and oracles to find logical bugs for DBMS systems. SQLRight contains a set of general APIs so that users can easily use existing fuzzers to test DBMSs and develop new oracles. We also improve the query validity to boost the fuzzing efficacy. Evaluations confirm that coverage-based guidance and query validity help SQLRight find more bugs than existing tools. Overall, SQLRight detects 18 logical bugs from SQLite, PostgreSQL and MySQL. Developers have confirmed all reported bugs and fixed 14 of them.

Acknowledgment

We thank the anonymous reviewers and development teams of SQLite and MySQL for their helpful feedback. This re-

search was supported, in part, by a subcontract of the DARPA AIMEE under Agreement No. HR00112090034, and an IST seed grant from Pennsylvania State University.

References

[1] Shadi Abdul Khalek and Sarfraz Khurshid. Automated SQL Query Generation for Systematic Testing of Database Engines. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2010.

[2] Alloy. Alloy - Documentation of Alloy SAT solver. https://alloytools.org/documentation.html. (visited in June 2021).

[3] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2019.

[4] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2019.

[5] Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. A Genetic Approach for Random Testing of Database Systems. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*, Vienna, Austria, 2007.

[6] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing Structure while Fuzzing. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, Santa Clara, CA, 2019.

[7] Bikash Chandra, Bhupesh Chawda, Biplab Kar, K. V. Maheshwara Reddy, Shetal Shah, and S. Sudarshan. Data Generation for Testing and Grading SQL Queries. *The VLDB Journal*, 24(6), Aug 2015.

[8] Peng Chen and Hao Chen. Angora: Efficient Fuzzing By Principled Search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, 2018.

[9] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Long Lu, et al. Savior: Towards bug-driven hybrid testing. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, Virtual, 2020.

[10] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. One Engine to Fuzz 'em All: Generic Language Processor Testing with Semantic Validation. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*, Virtual, May 2021.

[11] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. NTFUZZ: Enabling Type-Aware Kernel Fuzzing on Windows with Static Binary Analysis. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*, Virtual, May 2021.

[12] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases. In *Proceedings of the 28th Annual Network and Distributed System Security Symposium (NDSS)*, Virtual, 2020.

[13] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. Fuzzing Binaries for Memory Safety Errors with QASan. In *Proceedings of 2020 IEEE Secure Development (SecDev)*, Virtual, 2020.

[14] Free Software Foundation. Gnu bison, 2014. https://www.gnu.org/software/bison/.

[15] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path Sensitive Fuzzing. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, 2018.

[16] Bogdan Ghit, Nicolas Poggi, Josh Rosen, Reynold Xin, and Peter Boncz. SparkFuzz: Searching Correctness Regressions in Modern Query Engines. In *Proceedings of the Workshop on Testing Database Systems (DBTest)*, Portland, Oregon, 2020.

[17] Google. ClusterFuzz. https://google.github.io/clusterfuzz. (visited in June, 2021).

[18] Google. Honggfuzz. https://google.github.io/honggfuzz/. (visited in June 2021).

[19] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Proceedings of the 26th Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2019.

[20] Bo Jiang, Ye Liu, and W. K. Chan. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018.

[21] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, 2020.

[22] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. HFL: Hybrid Fuzzing on the Linux Kernel. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2020.

[23] Doug Laney. 3-D Data Management: Controlling Data Volume, Velocity and Variety. Technical report, February 2001.

[24] Caroline Lemieux and Koushik Sen. Fairfuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018.

[25] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proceedings of the VLDB Endowment*, 12(12):2118–2130, 2019.

[26] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. Automated Performance Bug Detection in Database Systems. *arXiv preprint arXiv:2105.10016*, 2021.

[27] LLVM. LibFuzzer - A Library For Coverage-guided Fuzz Testing. http://llvm.org/docs/LibFuzzer.html. (visited in June 2021).

[28] LLVM. Undefined Behavior Sanitizer (UBSan). https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. (visted in June 2021).

[29] Eric Lo, Carsten Binnig, Donald Kossmann, M. Tamer Özsu, and Wing-Kai Hon. A Framework for Testing DBMS Features. *The VLDB Journal*, 19(2):203–230, April 2010.

[30] Manuel Rigger. SQLancer. https://github.com/sqlancer/sqlancer. (visited in June 2021).

[31] Michaël Marcozzi, Wim Vanhoof, and Jean-Luc Hainaut. Test Input Generation for Database Programs Using Relational Constraints. In *Proceedings of the Fifth International Workshop on Testing Database Systems (DBTest)*, Scottsdale, Arizona, 2012.

[32] Barton P. Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[33] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. Generating Targeted Queries for Database Testing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, New York, NY, USA, 2008.

[34] MySQL. MySQL Customers. https://www.mysql.com/customers/. (visited in Jan 2022).

[35] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. sfuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, 2020.

[36] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2019.

[37] Shankara Pailoor, Andrew Aday, and Suman Jana. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, Baltimore, MD, 2018.

[38] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, 2020.

[39] PostgreSQL. Aboug PostgreSQL. https://www.postgresql.org/about/. (visited in Jan 2022).

[40] Kim-Thomas Rehmann, Changyun Seo, Dongwon Hwang, Binh Truong, Alexander Böhm, and Donghun Lee. Performance Monitoring in SAP HANA's Continuous Integration Process. *ACM SIGMETRICS Performance Evaluation Review*, 43:43–52, 02 2016.

[41] Richard Hipp. What is Fossil? https://www2.fossil-scm.org/home/doc/trunk/www/index.wiki. (visited on Jan 2021).

[42] Manuel Rigger. Bugs Found in Database Management Systems. https://www.manuelrigger.at/dbms-bugs/. (visited in June 2021).

[43] Manuel Rigger and Zhendong Su. 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 (ESEC/FSE)*, 2020.

[44] Manuel Rigger and Zhendong Su. Finding Bugs in Database Systems via Query Partitioning. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

[45] Manuel Rigger and Zhendong Su. Testing Database Engines via Pivoted Query Synthesis. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, 2020.

[46] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, 2012.

[47] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the workshop on binary instrumentation and applications*, pages 62–71, 2009.

[48] Kostya Serebryany. Sanitize, Fuzz, and Harden Your C++ Code. San Francisco, CA, 2016. USENIX Association.

[49] Donald R Slutz. Massive Stochastic Testing of SQL. In *VLDB*, volume 98, pages 618–622. Citeseer, 1998.

[50] SQLite. Datatypes In SQLite. https://www.sqlite.org/datatype3.html. (visited in Jan 2021).

[51] SQLite. Most Widely Deployed and Used Database Engine. https://www.sqlite.org/mostdeployed.html. (visited in June 2021).

[52] SQLite Team. sqllogictest Documentation. https://www.sqlite.org/sqllogictest/doc/trunk/about.wiki. (visited in June 2021).

[53] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2016.

[54] Michael Stonebraker, Sam Madden, and Pradeep Dubey. Intel "Big Data" Science and Technology Center Vision and Execution Plan. *ACM SIGMOD Record*, 42(1):44–49, 2013.

[55] Well-known Users of SQLite. https://www.sqlite.org/famous.html. (visited in June 2021).

[56] MySQL Customers. https://www.mysql.com/customers/. (visited in June 2021).

[57] PostgreSQL Clients. https://wiki.postgresql.org/wiki/PostgreSQL_Clients. (visited in June 2021).

[58] SQLSmith. https://github.com/anse1/sqlsmith. (visited in June 2021).

[59] J. Wang, P. Zhang, L. Zhang, H. Zhu, and X. Ye. A Model-based Fuzzing Approach for DBMS. In *Proceedings of the 8th International Conference on Communications and Networking in China (CHINACOM)*, Aug 2013.

[60] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. Industry Practice of Coverage-Guided Enterprise-Level DBMS Fuzzing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021.

[61] Gary Wassermann and Zhendong Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, New York, NY, 2007.

[62] Valentin Wüstholz and Maria Christakis. Harvey: A Greybox Fuzzer for Smart Contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, 2020.

[63] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data Race Fuzzing for Kernel File Systems. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, 2020.

[64] Peng Xu, Yanhao Wang, Hong Hu, and Purui Su. Cooper: Testing the Binding Code of Scripting Languages with Cooperative Mutation. In *Proceedings of the 29th Annual Network and Distributed System Security Symposium (NDSS 2022)*, San Diego, CA, apr 2022.

[65] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing File Systems via Two-Dimensional Input Space Exploration. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[66] Wen Xu, Soyeon Park, and Taesoo Kim. FREEDOM: Engineering a State-of-the-Art DOM Fuzzer. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Orlando, FL, November 2020.

[67] Khaled Yagoub, Peter Belknap, Benoit Dageville, Karl Dias, Shantanu Joshi, and Hailing Yu. Oracle's SQL Performance Analyzer. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 31(1), 2008.

[68] Jiaqi Yan, Qiuye Jin, Shrainik Jain, Stratis D. Viglas, and Allison Lee. Snowtrail: Testing with Production Queries on a Cloud Database. In *Proceedings of the Workshop on Testing Database Systems (DBTest)*, New York, NY, USA, 2018.

[69] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, Baltimore, MD, August 2018.

[70] Michal Zalewski. American Fuzzy Lop (2.52b). http://lcamtuf.coredump.cx/afl. (visited in June 2021).

[71] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Orlando, USA, November 2020.

A More Case Studies on Detected Bugs

```
CREATE TABLE v0 (v1 PRIMARY KEY, v2 , v3);
INSERT INTO v0 VALUES ('111', '222', '333');
CREATE TABLE v4 (v5 PRIMARY KEY);
INSERT INTO v4 VALUES ('0');
SELECT * FROM v4 JOIN v0 ON likely(v3=v1) AND v3='111';
-- output: 0|111|222|333
-- expect: (empty)
```

Listing 12: A logical bug of SQLite that leaks data due to incorrect constant propagation. The SELECT statement should not return any rows, but the buggy SQLite returns one row.

Data Leakage Bug. SQL statements in Listing 12 trigger logical bug 2 in Table 2, which returns extra rows. The first two statements create table v0 and inserts one row '111','222','333', while the following two statements create table v4 with one row '0'. The SELECT statement joins rows if they satisfy the conditions v3=v1 and v3='111'. Function likely() provides a hint about the comparison result, and by design will not change the results. Since v0 has no rows satisfying the conditions, SQLite should return nothing. However, due to the incorrect constant propagation, the conditions are incorrectly optimized to v1='111', and SQLite returns row 0|111|222|333. From the security perspective, attackers may exploit this bug to steal information, if they can trigger the DBMS to conduct queries similar to Listing 12. In a worse scenario, if a system relies on such SELECT for authentication, *e.g.*, searching for matched user name and password, attackers may get system access without valid credentials. Our bisector locates the buggy commit introduced in July 2018, which means the vulnerability has been there for several years. SQLRight can detect this bug using NoREC and TLP within six hours. Considering the simplicity of these queries, real-world uses of SQLite may have triggered this bug.

```
CREATE TABLE v0 (v1 INTEGER PRIMARY KEY) WITHOUT ROWID;
INSERT INTO v0 VALUES (10) ;
ALTER TABLE v0 ADD v2 INT;
SELECT * FROM v0 WHERE v1=20 OR (v1=10 AND v2=10);
-- output: 10|NULL
-- expect: (empty)
```

Listing 13: A logical bug of SQLite that leaks data due to incorrect primary key. We expect the SELECT statement returns no rows, but the buggy SQLite provides one extra row.

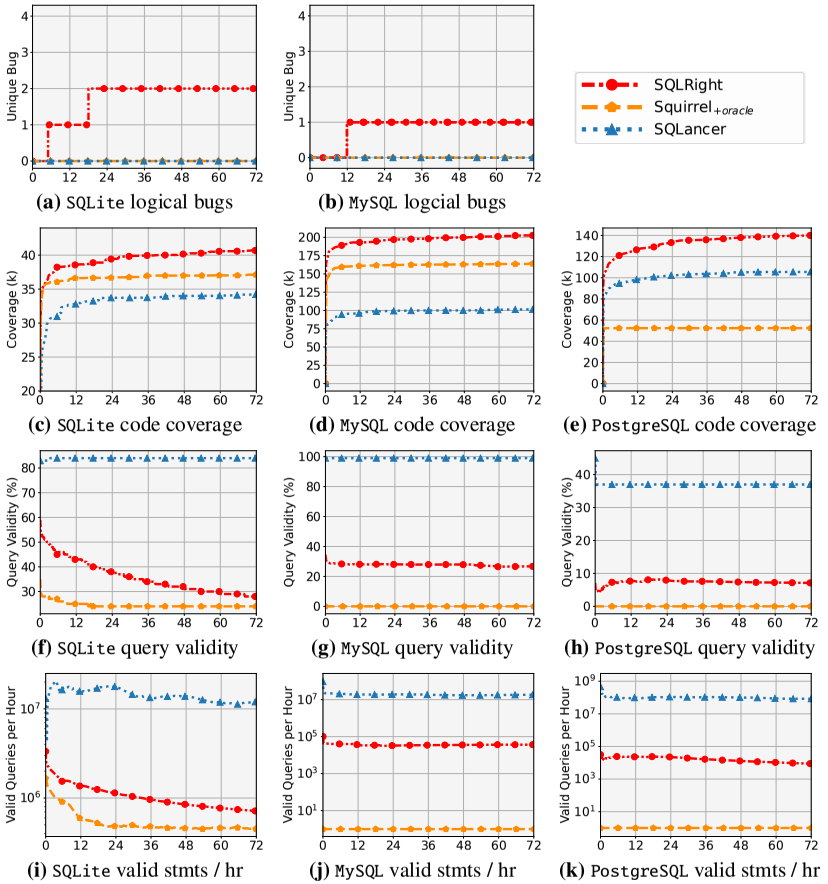
Data Leakage due to Primary Key. Listing 13 shows queries that trigger logical bug 8 in Table 2, which returns extra rows. The first statement creates table v0 with one column v1, where v1 has the INTEGER type and is the primary key. The second statement inserts one row 10 into this table. The third statement adds another column v2 of type INT into v0. By default, all rows in v0 will have a NULL value in column v2. The last SELECT statement searches in table v0 for rows where v1=20 or both v1 and v2 have the value of 10. Since table v0 only has one row 10|NULL, we expect the DBMS returns no rows. However, it returns 10|NULL, even though the content does not satisfy the conditions in WHERE.

SQLRight can detect this bug by oracle TLP and oracle Rowid. Regardless of the oracle, to trigger the bug the query should create a WITHOUT ROWID table. In an ordinary table (a table created without WITHOUT ROWID), SQLite creates a hidden column rowid and uses it as the real primary key. The PRIMARY KEY column is merely implemented as a UNIQUE INDEX that redirects the search back to rowid. However, in a WITHOUT ROWID table the extra column rowid is disabled. SQLite implements the PRIMARY KEY columns as the unique id of each row, and directly uses them to narrow down the search. Without the extra rowid column, SQLite can improve the speed of fetching data and reduce the storage size. In the first statement of Listing 13, v0 is a WITHOUT ROWID table. SQLite treated column v1 as the real primary key and stored it in a sorted manner. In the third statement, SQLite adds column v2 and stores it without sorting. The unsorted v2 should not be used to narrow the search. However, the query planner of SQLite does not take this exception into account and uses v2 to traverse the table. Therefore, SQLite returns the extra row to the query. We reported the bug to SQLite developers, and they fix it by not considering the unsorted column when constructing the query plan.

```
CREATE TABLE v0 (v2 INT, v1 INT);
INSERT INTO v0 VALUES (10, 10);
CREATE INDEX v4 ON v0 (v1) WHERE v2>NULL;
SELECT * FROM v0 WHERE v2 IS NOT NULL;
-- output: (empty)
-- expect: 10|10
```

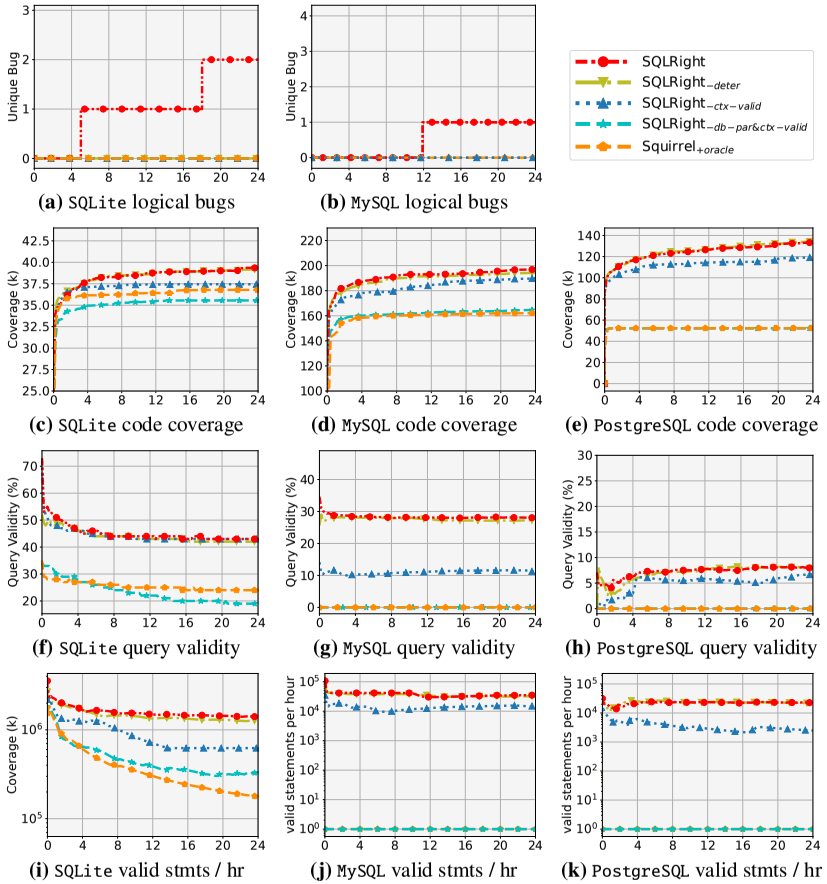
Listing 14: A logical bug of SQLite due to futile index. The SELECT statement should return one row, but SQLite returns nothing as it incorrectly uses the index v4.

Missing Data due to Futile Index. Listing 14 shows the query set that triggers logical bug 11 in Table 2, which drops expected rows from the result. The first two statements of this query create table v0 and insert one row 10|10. The third statement creates a partial index v4 on column v1 when v2>NULL. Since the comparison with NULL always returns NULL, no rows are added into index v4. The last statement searches in table v0 and aims to find rows where v2 is not NULL. Since the table only has one row 10|10, we expect the query returns one row. However, the buggy version of SQLite does not return any rows. The reason is that SQLite confuses >NULL with IS NOT NULL, and thus uses the empty index v4 to handle the SELECT statement. SQLRight can detect this bug using oracle NoREC and oracle Index. After our report, the developers have fixed this bug by resolving the confusion.



(a) SQLite logical bugs (b) MySQL logcial bugs (c) SQLite code coverage (d) MySQL code coverage (e) PostgreSQL code coverage (f) SQLite query validity (g) MySQL query validity (h) PostgreSQL query validity (i) SQLite valid stmts / hr (j) MySQL valid stmts / hr (k) PostgreSQL valid stmts / hr

Figure 8: Comparison between different tools (TLP). a-k show the number of unique logical bugs, code coverage, query validity and valid statements over time for each fuzzing instance. We run each instance for 72 hours, repeat for five times and report the average.



(a) SQLite logical bugs (b) MySQL logical bugs (c) SQLite code coverage (d) MySQL code coverage (e) PostgreSQL code coverage (f) SQLite query validity (g) MySQL query validity (h) PostgreSQL query validity (i) SQLite valid stmts / hr (j) MySQL valid stmts / hr (k) PostgreSQL valid stmts / hr

Figure 9: Unit Tests for SQLRight components (TLP). a-k show the number of unique logical bugs, code coverage, query validity and valid statements over time for each fuzzing instance. We run each instance for 24 hours, repeat for five times and report the average.

B Generating the Motivating Example

```
01 CREATE TABLE person (
02   pid      INTEGER PRIMARY KEY,
03   tid      INTEGER REFERENCES team,
04   leader   BOOLEAN,
05 );
06 CREATE UNIQUE INDEX leader ON person(tid) WHERE leader;
07 SELECT pid FROM person WHERE leader AND tid=1;
```

Listing 15: Original queries that lead to Listing 1. We collected this seed input from SQLite website about unique partial indexes.

To prepare fuzzing, we collect the seed input from the SQLite website about unique partial indexes, shown in Listing 15. Together with other seed inputs, we launch SQLRight to test the SQLite. During the initialization, SQLRight scans all seed inputs. When reaching Listing 15, it extracts the SELECT statement at line 7 and saves it to the select queue; other statements are stored in the normal queue. After that, SQLRight starts to mutate existing queries to get new ones.

When lines 1-6 are selected for mutation, SQLRight first invokes the preprocess() function of oracle Index. preprocess() determines that this query set is compatible, and it removes line 6 to avoid false alarms. Then, the mutation engine randomly modifies the remaining components, *i.e.*, CREATE TABLE, to generate new statements. By some chance, it deletes column tid and column leader from table person, and inserts an INSERT statement that adds three values into the table. Meanwhile, SQLRight invokes function append_output() of oracle Index to insert SELECT statements. append_output() picks up existing SELECTs from the select queue, and mutates them in a controlled manner: it will keep the SELECT keyword but can modify other components. After generating a set of valid SELECTs, SQLRight appends them to the previously generated query set, *i.e.*, after CREATE TABLE and INSERT. At this moment, we complete the query mutation.

Next, SQLRight applies the validation to update the operands in each query, to make the whole query set semantically correct. For example, it updates the table names in FROM clauses of SELECTs so that they use the table created by CREATE TABLE. After that, SQLRight sends the query set to function transform() to produce functionally equivalent forms. transform() picks some CREATE INDEX statements from the normal queue, and mutates them for new operations, like adding the UNIQUE before INDEX. It inserts the generated CREATE UNIQUE INDEX statements into the given query set randomly, and updates the operands accordingly (similar to validation). transform() returns the new query sets back to SQLRight, and SQLRight feeds the original and new query sets to SQLite for execution. After the execution, SQLRight invokes function compare() to decide whether the results are consistent or not. In this example, compare() notices that one result has one row while another has two. Therefore, it notifies SQLRight about the inconsistency, and SQLRight will raise an alarm of the potential logical bug.

Regardless of the consistency, SQLRight checks whether the execution of SQLite triggers new code coverage or not. If so, it extracts SELECTs from the combined query set and saves them into the select queue, and stores other statements into the normal queue. Now, SQLRight completes one fuzzing round, and it will move on to mutate other query sets.