Understanding and Reusing Test Suites Across Database Systems

SUYANG ZHONG, National University of Singapore, Singapore
MANUEL RIGGER, National University of Singapore, Singapore

Database Management System (DBMS) developers have implemented extensive test suites to test their DBMSs. For example, the SQLite test suites contain over 92 million lines of code. Despite these extensive efforts, test suites are not systematically reused across DBMSs, leading to wasted effort. Integration is challenging, as test suites use various test case formats and rely on unstandardized test runner features. We present a unified test suite, SQuaLity, in which we integrated test cases from three widely-used DBMSs, SQLite, PostgreSQL, and DuckDB. In addition, we present an empirical study to determine the potential of reusing these systems' test suites. Our results indicate that reusing test suites is challenging: First, test formats and test runner commands vary widely; for example, SQLite has 4 test runner commands, while MySQL has 112 commands with additional features, to, for example, execute file operations or interact with a shell. Second, while some test suites contain mostly standard-compliant statements (e.g., 99% in SQLite), other test suites mostly test non-standardized functionality (e.g., 31% of statements in the PostgreSQL test suite are non-standardized). Third, test reuse is complicated by various explicit and implicit dependencies, such as the need to set variables and configurations, certain test cases requiring extensions not present by default, and query results depending on specific clients. Despite the above findings, we have identified 3 crashes, 3 hangs, and multiple compatibility issues across four different DBMSs by executing test suites across DBMSs, indicating the benefits of reuse. Overall, this work represents the first step towards test-case reuse in the context of DBMSs, and we hope that it will inspire follow-up work on this important topic.

CCS Concepts: • **Information systems** → *Data management systems*; • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: DBMS testing, test case reuse

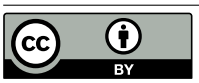
ACM Reference Format:
Suyang Zhong and Manuel Rigger. 2024. Understanding and Reusing Test Suites Across Database Systems. *Proc. ACM Manag. Data* 2, 6 (SIGMOD), Article 253 (December 2024), 26 pages. https://doi.org/10.1145/3698829

1 Introduction

Database Management Systems (DBMSs) are large, complex software systems that are widely used by applications to store and retrieve data. DBMSs consist of various components, such as query processors [36], log managers [28], storage engines [10], and optimization engines [30, 44]. Thus, DBMSs are usually large and complex. MySQL, one of the most popular open-source DBMSs, consists of about 3.5M lines of code (LOC). Even SQLite, a relatively lightweight relational DBMS, contains about 241K LOC. Consequently, DBMSs can be affected by bugs.

Unsurprisingly, DBMSs are typically well-tested. Researchers have developed various techniques to automate testing DBMSs and successfully found logic bugs [23, 33–35, 38, 40], performance issues [5, 20, 24], crash bugs [37, 50], and transaction bugs [21]. Developers also test DBMSs

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



ACM 2836-6573/2024/12-ART253
https://doi.org/10.1145/3698829

using test suites. These test suites typically consist of test cases as well as a test runner that can be used to execute the test cases and validate their results. Most importantly, so-called *end-to-end test suites* consist of SQL statements and test the DBMSs from a user perspective. For example, `sqllogictest` [1] is SQLite's test suite, which includes millions of SQL statements. A common reason for the prevalence of such SQL test suites is to avoid a lock-in regarding their own implementation details [46].

Despite developers' significant effort in developing these test suites, test cases are not systematically reused between DBMSs, wasting significant developer effort. A recent interdisciplinary seminar on DBMS reliability stresses the importance of this issue and identified *"a common testcase specification format and a test corpus that can be shared between DBMS engineering teams"* as one of the primary future challenges [9]. We believe that a software engineering angle on this problem is promising, as test reuse has been a significant concern for the software engineering community.

Reusing test cases poses three challenges. First, the test case formats designed by different DBMSs' developers are different, which prevents their direct reuse. Some DBMSs' test suites use SQL scripts to execute the engine and validate the results, while others rely on additional annotations or commands in the test files, for example, `for-loops` to execute one statement multiple times. Second, while SQL has been standardized by ANSI/ISO [2], in practice, DBMSs implement various SQL dialects that differ in syntax and semantics. Most DBMSs provide unique features. For test cases exercising such features, simply running them on other DBMSs would fail, because they would not implement these features. For example, PostgreSQL provides functions starting with "pg_", which are typically system functions used for administrative tasks or system monitoring. Test cases containing these functions would likely result in errors when being executed on other DBMSs. Third, results are usually inconsistent among different DBMSs, or even the same DBMS in different configurations, for the same SQL query. The reason could be floating-point precision differences, unstable query plans, or content discrepancies.

To tackle the above challenges, this paper describes a large-scale, systematic study that we performed on four DBMSs—SQLite, MySQL, PostgreSQL, and DuckDB—to investigate the reusability of DBMSs' test suites. To this end, we pose the questions below:

- *RQ1: What features do end-to-end DBMS testing frameworks provide?* As a first step, we determined whether the test formats and runners are sufficiently similar to enable reusing their tests.
- *RQ2: What do test cases typically look like?* Next, we sought to understand the characteristics of different test suites' test cases with respect to, for example, what SQL statements they included. A high percentage of SQL statements defined by the standard suggests potentially higher reusability.
- *RQ3: What are the challenges of executing end-to-end tests?* We aimed to understand the effort required in developing test runners to execute the test suites. For example, we sought to understand whether executing the tests would be as simple as sending the SQL statements to the DBMS under test and validating its results, or whether any additional challenges would need to be accounted for.
- *RQ4: Can test cases written for one DBMS find bugs in other DBMSs?* As our main goal was to investigate whether test suites can be reused, we sought to understand whether, assuming a common test format and runner, we would be able to successfully execute test cases written for one DBMS on other DBMSs. Furthermore, we aimed to investigate whether failing test cases could indicate overlooked bugs.

To answer the above questions, we studied DBMSs' test suites and derived a test format and test runners that allowed us to execute the test suites of SQLite, PostgreSQL, and DuckDB on any of

these DBMSs; besides the insights of the empirical study, we present SQuaLity, a unified test suite, which, to the best of our knowledge, represents the first step toward systematic test case reuse for DBMSs.

Our results demonstrate that test reuse can be useful, as we found new bugs, despite the challenges posed by the various test case formats and DBMS-specific features used in the test cases. For RQ1, we found that test suites of DBMS used different formats of their test cases, and contained 4 to 112 unique non-SQL commands interpreted by the test runners, for environmental settings or execution control. For RQ2, we found that `SELECT`, `INSERT` and `CREATE TABLE` are the most common SQL statements in the test suites, which indicates a high potential for reusability, as these statements are standardized. However, we also found that test suites contained dialect-specific statements, posing difficulties for reuse. For RQ3, we found that test cases required specific environments, extensions, and client dependencies to be tested appropriately. For RQ4, we found 3 crashes as well as 3 hangs and reported the issues to the developers, overall 3 of which have been fixed. We also found suspicious discrepancies across SQL dialects, some of which we reported to the developers, which subsequently led them to discuss the intended semantics of the statements.

In summary, we make the following contributions:

- an empirical study on the characteristic and reusability of different DBMSs' test suites (see Section 3–5);
- SQuaLity, a test suite that unifies the tests of SQLite, PostgreSQL, and DuckDB.

2 Methodology

In this paper, we sought to adapt the existing DBMSs' test suites to a common platform, to study both the opportunities and challenges of reusing the test suites, aiming to answer the above-posed research questions. We (1) extracted the test cases of each DBMS and converted them to a common format, (2) implemented a unified test runner to execute the test cases across DBMSs, and (3) analyzed the test cases we extracted in step (1) and the results of the test cases in step (2). We detail these steps in the following paragraphs.

Terminology. We specify the terms we use in the remainder of the paper. A *test case* consists of an SQL statement and a specification of its expected behavior. A *test file* contains several SQL *test cases* and a *test suite* consists of multiple test files. Note that test cases in a test file might have implicit dependencies between each other. For example, one test case might check that a row was inserted successfully into a table, while another test case might validate a query's expected result, which depends on the successful execution of the `INSERT` statement. The *test runner* parses the *test files*, executes the *test cases*, and validates them. In the subsequent paragraphs, the term *test suite* is used to denote a combination of both the test runner and the collection of test cases, unless explicitly stated otherwise. *Failed test cases* refer to test cases that, when executed by the test runner, produce behavior that deviates from the expected outcome. Thus, the term refers to both statements that compute the expected result, or expectedly fail execution.

Selecting DBMSs. We chose four popular open-source DBMSs, whose test suites we subsequently investigated (see Table 1). We selected these DBMSs because they are highly popular based on ranking such as the DB-Engines ranking [39], which ranks DBMSs according to their popularity, or their number of GitHub stars. These DBMSs have also been the target of various automated testing works [4, 14, 18, 23, 33–35, 37, 50]. MySQL and PostgreSQL are traditional client-server DBMSs. SQLite and DuckDB [32] are embedded DBMSs, where the database system runs in the same process as the application it uses, making it particularly suitable for, for example, embedded

Table 1. DBMS rankings and their test suites information

DBMS Names	DB-Engines Rankings	GitHub Stars	DBMS Version	Test Suite Version	Test Files
SQLite	9	4.5k	3.41.1	`a22803`	622
MySQL	2	9.5k	8.0.33	`ea7087`	1418
PostgreSQL	4	13.2k	15.2	`bc9993`	212
DuckDB	103	11.9k	0.8.1	`6536a7`	2537

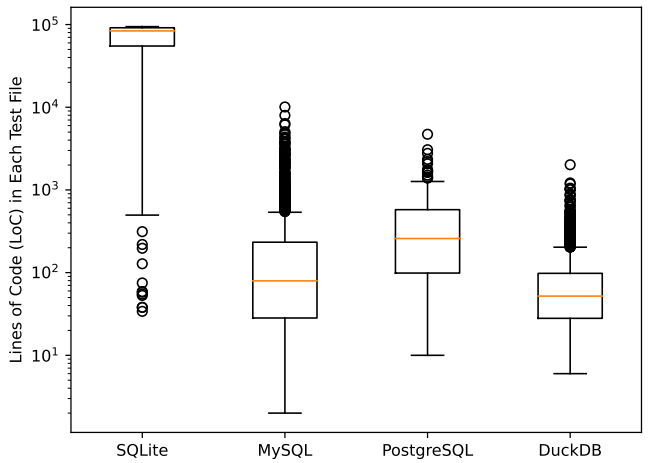


Fig. 1. Number of test case lines per file of each DBMS (logarithmic scale).

devices. DuckDB is an Online Analytical Processing (OLAP) system, while the other three DBMSs are Online Transactional Processing (OLTP) systems.

Selecting test suites. From the four selected DBMSs, we identified four test suites, SQL Logic Test (SLT) for SQLite, MySQL Test Framework [3], the PostgreSQL regression test, and the DuckDB test suite, as shown in Table 1. Many DBMSs use several test suites for different purposes, implemented in different programming languages. All four DBMSs mentioned above have comprehensive SQL test suites, with detailed documentation and a large number of test cases (see Figure 1). In our study, we considered only test suites written in SQL, as we sought to reuse the test suites, and each DBMS uses SQL as a primary interface. Differences between DBMS implementations would make it difficult to directly reuse test cases consisting of API calls. For example, test cases from DuckDB written using the C API, necessitate recompilation and relinking with every modification in the common header. This process can be time-consuming, particularly when the test suite is extensive [31]. Besides, the internal APIs that other DBMSs provide would significantly differ. We omitted these non-SQL test suites, as they could not be easily reused. We disregarded commercial DBMSs like Oracle or SQL Server, as the test suites and runners are not publicly available.

SQuaLity. We built a unified test suite, SQuaLity,[1] to answer RQ1–RQ4. Its core logic is implemented in about 3,000 LOC to support analyzing test cases and executing them across different DBMSs. SQuaLity can parse test files from each DBMS into individual SQL statements and extract the test runner commands. The unified format is currently designed as an internal intermediate representation. Thus, we do not consider the format as a core contribution of this work and for brevity, omitted describing it. To execute test cases and obtain the test case results, SQuaLity uses the Python DBMS connectors, allowing it to compare the DBMSs' results in a consistent manner. If we had used the *Command Line Interface* (CLI) clients, we would have had to parse and convert the CLI results returned in text format, which differs for each DBMS. SQuaLity executes and validates the test cases in a statement-by-statement manner. We omitted support for uncommon test suite features to limit the implementation effort to a reasonable degree—one of the authors was focusing on the project for a period of 12 months.

Methodology RQ1. We systematically analyzed and identified the common features of each test suite, with a focus on test case formats and test runner functionalities. We conducted three analyses, in each of which we could refine the results and insights of the previous analyses. First, we collected the test cases and documentation of each test suite. In this step, we analyzed how the test runner interprets elements of test cases, including how it parses the test files, executes the test runner commands, and validates the test results. Second, we analyzed the commonalities and differences between these test suites, which are discussed in Section 3. Based on the findings, we implemented SQuaLity. Third, we parsed and executed the test cases using SQuaLity, which converts various test case formats into an internal unified format and identifies test runner commands. This, in turn, allowed us to refine and validate our findings from step one.

Analyzing the test cases. We sought to investigate the test cases at the granular level of individual SQL statements, which helped us answer the subsequent RQs. However, extracting individual SQL statements from the test cases is challenging, as the SQL statements are embedded in comments, test runner commands, and the specified expected results. A naive approach, for example, using regular expressions, would lead to false positives and negatives due to the complexity of SQL syntax. Besides, distinguishing different SQL statement types is difficult, considering that we analyzed test cases from different DBMSs with different dialects. As we sought to understand whether the test cases are suitable for execution across DBMSs, we adopted a best-effort approach. We obtained the test cases from each DBMS test suite by implementing a corresponding parser that obtained each individual runner command and SQL statement within each test file. Then, we relied on `sqlparse`,[2] a best-effort SQL-dialect agnostic parser, to identify the type of each individual SQL statement.

Methodology RQ2. We investigated the test contents by analyzing the individual SQL statements for SQLite, PostgreSQL, and DuckDB. We omitted the MySQL test suite because its complex test case format and numerous runner commands make it highly specific to MySQL, limiting its potential reuse. We wanted to determine how SQL is used in the three test suites, because this could answer whether the test cases are suitable for reusing. To this end, we first analyzed whether the test cases use SQL statements whose syntax is defined in the SQL standard [2]—we subsequently refer to such statements as *standard compliant*. We categorized all the SQL statements according to their statement type (*e.g.*, the statement type of `SELECT * FROM t0` is `SELECT`) and observed the distribution of different SQL commands usage as well as whether they are standard-compliant. While the analysis was on a statement level, statements could still contain dialect-specific functions and keywords; for example, while in `SELECT to_json ( date ' 2014 -05 -28 ');`, the `SELECT`

[1]Available at https://doi.org/10.5281/zenodo.13896444.
[2]https://github.com/andialbrecht/sqlparse

statement is standardized, it references a PostgreSQL-specific JSON function. We addressed this limitation in RQ4 by investigating whether the DBMSs could successfully produce the expected results for the statements.

Understanding the testing environment. We observed that the unified test suites need to carefully consider the state of the DBMS and environment to successfully execute the test cases. A naive implementation of a test runner would pass the SQL statements to the DBMSs, but we found that various *dependencies*—certain pre-execution set-up and requisites of the DBMS and the environment—need to be considered to produce the intended results. Each DBMS has unique configurations, and implementing a new test runner could lead to missing configurations (or misconfigurations), producing unexpected results that are difficult to detect [45]. The documentation of the PostgreSQL test suite mentions the above issue.[3] Besides, considering that DBMSs allow interactions through various clients—CLIs and database connectors often provided for various programming languages—the risk exists that clients might present results in different ways. Test cases that are strongly dependent on such dependencies could hardly be reused across DBMSs.

Methodology RQ3. We sought to systematically study the dependencies of each test suite as they complicate the implementation of test runners. Conceptually, we *transplanted* [8] the test suites into SQuaLity's test corpus, that is, by extracting and parsing test cases from the test suite of each DBMS into a format that SQuaLity supports. We executed and validated the transplanted test suites on the *donor*—the DBMS for which the test suites were originally designed. We collected the test results, including *passed test cases*, for which the actual behavior matched the expected behavior specified by the corresponding test cases, and conversely, *failed test cases*. Then, we randomly sampled—following the methodology of other studies [7]—100 failed test cases per DBMS to investigate the dependencies of each test suite. We manually re-executed the test cases using the client of the donor test suite to determine if they were dependent on specific clients. Additionally, we examined the documentation of each donor to investigate potential environment and extension-related issues.

Methodology RQ4. We sought to understand whether test cases written for one DBMS (*donor*) could expose issues in another DBMS (*host*). We executed test suites from different donors across DBMSs using our unified test runner and compared the actual results with the expected results. We analyzed the failed test cases, which could be caused by SQL dialect differences, settings discrepancies, or potential bugs. We exhaustively investigated the failures in SLT and applied the same sampling methods used in RQ3 to the other test suites. For SLT, since it contains mostly standard-compliant SQL commands and error types are relatively simple, we could investigate all the failed test cases. Conversely, other test suites exhibit various failures, and a thorough investigation would cost much effort. To address this, for the results of each test pair (*e.g.*, executing SQLite on the DuckDB test suite), we randomly sampled a test case and designed a rule that could identify other cases sharing similar patterns with it, subsequently filtering the remaining cases. We iteratively applied this procedure until either all cases could be classified according to the established rules, or the number of rules had reached an arbitrary, but reasonable, predefined limit of 15. When the 15 rules could not classify all the unexpected results, we sampled the execution results.

RQ4 Failure investigation. As part of RQ4, we systematically analyzed the root causes for the failures, similar to a recent study on transaction incompatibilities [11]. We analyzed a failed test case in the following steps. First, we isolated the SQL statements from the test file, reduced them [48], and re-executed them. Second, we manually investigated the documentation of each DBMS for an explanation of the observed discrepancy. We classified the discrepancy as a compatibility issue,

[3] https://www.postgresql.org/docs/15/regress-evaluation.html

Listing 1. SQLite tests typically consist of a single file, where each SQL statement is annotated by its type and expected effect/result

```
statement ok
CREATE TABLE t1(a INTEGER, b INTEGER, c INTEGER)

statement ok
INSERT INTO t1(c,b,a) VALUES (3,4,2), (5,1,3), (1,6,4)

query I rowsort
SELECT a, b FROM t1 WHERE c > a;
----
2
4
3
1
```

where the behavior was documented by the developers. Third, if the documentation failed to explain the discrepancies, we reported them to the developers.

3 Test Suite Features (RQ1)

Overview. We investigated four test suites—SLT, the MySQL Test Framework, the PostgreSQL regression tests, and the DuckDB test suite—from two perspectives: test case formats and test runner commands. Test cases differ in how they are structured in test files and how expected results are represented. Additionally, different runners use various non-SQL commands, rather than only sending SQL statements to the DBMS and validating their results.

Test file formats. The test file format varies based on whether the SQL statements and result specifications are explicitly separated in each file. DuckDB specified their test cases in the SLT format, with some minor differences from the original SLT format as detailed below. Each test case of SLT is an independent file consisting of several records, usually SQL statements with a header that specifies the expected behavior, as shown in Listing 1. In this example, two statements create a table `t1` and insert three rows, which are expected to execute successfully. Then, a `SELECT` query fetches the data from the table and uses a `rowsort` comparator to validate the results, which is specified after the "`----`". We discuss the query result format in the subsequent paragraph. For each record, its SQL statement is executed individually to validate if the DBMS computes the correct result [1]. Conversely, the test cases of MySQL and PostgreSQL are not explicitly split in each test file, that is, the test runner executes the entire test file all at once. Then, it only compares whether the output of the entire test file matches the expected result. Listing 2 shows an example of a MySQL test. Each MySQL test case is a pair of two files, a test file, and a result file. A test file is a series of SQL statements and test runner commands. A result file is a copy of the test file, with the expected results after each SQL statement.

Result formats. The query results specification for each test case varies across different test suites. Results in SLT test cases are in value-wise ordering, in which each value of the query must appear on a separate line, as shown in Listing 1. DuckDB uses both value-wise ordering, as well as row-wise ordering, the latter of which resembles the structure of a table. Each line in the expected result represents a table row. Listing 3 shows an example in which the result is in row-wise order. MySQL uses test results in a row-wise format, with an additional header to denote the column names. PostgreSQL results, returned by the CLI, are strings representations of the table, containing

Listing 2. MySQL tests typically consist of two files: a test file that contains SQL and test runner commands, and a result file that contains the expected output from the runner executing the test file.

```
# t/example.test
CREATE TABLE t1(a INTEGER, b INTEGER, c INTEGER);
INSERT INTO t1(c,b,a) VALUES (3,4,2), (5,1,3), (1,6,4);
SELECT a, b FROM t1 WHERE c > a;
# r/example.result
CREATE TABLE t1(a INTEGER, b INTEGER, c INTEGER);
INSERT INTO t1(c,b,a) VALUES (3,4,2), (5,1,3), (1,6,4);
SELECT a, b FROM t1 WHERE c > a;
a b
2 4
3 1
```

Listing 3. DuckDB test results are usually in row-wise format: each line is a row of the result table

```
statement ok
CREATE TABLE t1(a INTEGER, b INTEGER, c INTEGER)

statement ok
INSERT INTO t1(c,b,a) VALUES (3,4,2), (5,1,3), (1,6,4)

query I
SELECT a, b FROM t1 WHERE c > a;
----
2 4
3 1
```

the table header, values, and row count. All formats convey information about table values. Thus, lossless information transfer between the formats is possible.

Non-SQL commands overview. We listed the Non-SQL commands in Table 2 and detail them in the next paragraphs. "✓" indicates that the corresponding DBMS's test suite supports the feature, while the number indicates the total number of supported commands. Test suites can implement the same functionalities using different names; we grouped them together in the table. The MySQL test suite contains the most runner commands, making it difficult to reuse its test cases. Conversely, SLT supports only few runner commands. DuckDB's test suite format is based on SLT, but provides additional functionality. PostgreSQL test cases are SQL scripts including psql commands. Psql[4] is the CLI for PostgreSQL and is used to execute the PostgreSQL test cases.

Environmental settings. Commonly, test runners support commands to set up the testing environment. *Include* is used to read test code from other files, which enables developers to extract contents shared by multiple test files. *Set Variable* is used to set a specific variable that may be used in the following test procedure. *Load* loads data or custom functions from a specific source location.

Execution flow control. Test runners support commands to manage the execution of SQL statements, allowing for more control than just sequential execution. *Loop* is used to execute the specific commands several times. *Skiptest* is used to conditionally skip the execution of tests. For example, in SLT, developers used these commands (see Listing 4) to implement test cases where the SQL

[4]https://www.postgresql.org/docs/current/app-psql.html

Listing 4. In MySQL, the "/" operator consistently performs floating-point division, even if both operands are integers. The "DIV" operator should be used if an integer division is needed.

```
onlyif mysql # DIV for integer division:
query I rowsort label-11
SELECT ALL 62 DIV ( + - 2 )
----
-31

skipif mysql # not compatible
query I rowsort label-11
SELECT ALL 62 / ( + - 2 )
----
-31
```

Table 2. Non-SQL commands of each DBMS test runner

Feature	SQLite	MySQL	PostgreSQL	DuckDB
Include	-	✓	✓	-
Set Variable	✓	✓	✓	✓
Load	-	✓	✓	✓
Loop	-	✓	-	✓
Skiptest	✓	-	✓	✓
Multi-Connections	-	✓	✓	✓
CLI Commands	-	-	114	-
Runner Commands	4	112	-	16

syntax varies from one DBMS to another. As another example, in DuckDB, `require sqlsmith` is used to check whether the current DuckDB has loaded the `sqlsmith` extension, which is a popular fuzz testing tool; if not, subsequent statements are skipped. Some test runners support *Multi-Connections*, enabling several connections to one database at a time.

Psql CLI commands. As mentioned above, the PostgreSQL regression test suite uses the CLI to execute the tests, which provides meta-commands or shell-like features. In total, the suite makes use of 59 out of 114 unique CLI commands. Certain CLI commands have corresponding SQL commands in other DBMSs, for example, `\c testdb` is equivalent to `USE testdb`, which changes the current database to `testdb`. Some commands handle client-level functions, for example, `\setenv` is used to set an environmental variable. We did not seek to interpret and implement these commands; they are processed by the client of the DBMS, not the test runner.

MySQLTest commands. The MySQL Test Framework supports 112 unique runner commands. Besides the features listed in Table 2, the test runner of MySQL supports various commands for local file operations (*e.g.*, `writefile filename`), server monitoring (*e.g.*, `shutdown_server [timeout]`), and shell interaction (*e.g.*, `exec command`). This enhances the test runner's ability to test the MySQL server, but the large amount of runner commands poses challenges to reusing the test cases.

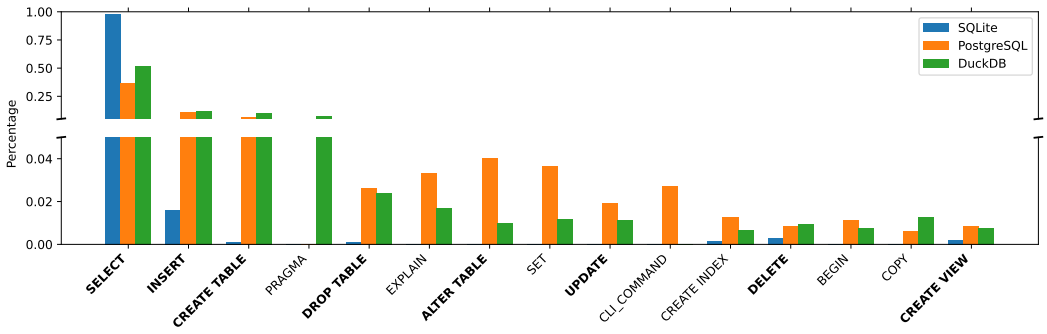


Fig. 2. Distribution of SQL statement types in each DBMS test suite, with standard-compliant SQL statements denoted in bold.

Table 3. The percentage of standard-compliant SQL statements among the test cases

DBMS Test Suite	Overall Standard SQL	Exclusive Standard Files
SQLite	99.76%	63.92%
PostgreSQL	68.89%	10.37%
DuckDB	76.14%	16.24%

The SQLite, PostgreSQL, and DuckDB test suites use easily parsable test formats, whereas MySQL's cases embedded runner commands, which complicates reusability. Besides, the MySQL test runner's support for 112 runner commands (*e.g.*, compared to 4 in SQLite), and PostgreSQL's use of CLI commands both pose challenges to reuse.

4 Test Case Patterns (RQ2)

We analyzed the SQL test cases, primarily by quantifying the frequency of each type of SQL statement present within the test suites. Figure 2 shows the percentage of SQL statements in the three selected DBMSs' test suites. We ordered the SQL statements based on their mean occurrence frequency across three different DBMSs and selected the 15 highest-ranking ones for presentation.

Overview. As shown in Table 3, SLT is the most standard-compliant test suite. Almost all test cases consist of standard-compliant SQL statements. However, only 63.9% of the SLT test files contain only standard-compliant test cases, because 35.9% of the test files contain `CREATE INDEX` statements, used to create an index on columns to improve the speed of data retrieval [15], which are not specified in the ANSI/ISO standard, despite being widely supported by DBMSs. If we counted `CREATE INDEX` as a standard-compliant statement, a higher standard compliance, that is, 99.8% of the test files contain standard-compliant test cases, could be achieved. The proportion of standard SQL statements in PostgreSQL is 68.9%, which is the lowest.

Common SQL statements. As shown in Figure 2, all test suites contain common and standard-compliant SQL statements, such as `SELECT`, `CREATE`, and `INSERT`, demonstrating the potential of reusing the test cases. `SELECT` allows querying data from the database. It is the most used SQL statement among all test suites. `CREATE` is one of the Data Definition Language (DDL) statements in

Listing 5. By altering the explain_output setting to OPTIMIZED_ONLY, the display changes from the default physical plan to present the optimized logical plan.

```
1 CREATE TABLE integers(i integer, j integer, k integer);
2 INSERT INTO integers VALUES (5, 5, 5), (10, 10, 10);
3 PRAGMA explain_output = OPTIMIZED_ONLY;
4 EXPLAIN SELECT k FROM integers where j=5;
```

SQL, and it is primarily used to create tables. DROP and ALTER are also DDL statements that remove or modify the tables the previous CREATE statements create. Many other statements (*e.g.*, SELECT and INSERT) operate on the table or a database object that was created using the CREATE statement. INSERT is used to insert data into a table. It is frequently used following a CREATE TABLE statement. UPDATE and DELETE are used to modify and remove the data in the table, respectively. While SQLite has the largest test suite SLT, with 622 test files, and each test file has 11907 SQL statements on average, it contains only such fundamental SQL statements as above. The prevalence of standard SQL statements in SLT indicates a higher likelihood of being suitable for reuse.

Unstandardized SQL statements. The test suites of PostgreSQL and DuckDB, despite having a smaller number of test cases, contain a broader range of statements, including SQL statements related to transactions, query plans, and DBMS-specific features.

Statements such as SET in PostgreSQL (3.62%), and PRAGMA in DuckDB (6.99%) configure the DBMS. Neither of the statements is specified by the SQL standard. SET is used for changing session-level settings. PRAGMA is specific to SQLite and DuckDB. For example, DuckDB uses a PRAGMA statement to change how query plans are displayed, as shown in Listing 5. These statements inhibit the reuse because they are not standard-compliant and may not execute successfully across different DBMSs. DBMSs like SQLite silently ignore unknown parameters specified by PRAGMA. Subsequently, unsupported settings can lead to unexpected results in those test cases relying on them.

In addition to SELECT statements, test suites also apply other non-standard queries to validate specific properties of the system. EXPLAIN statements are used in PostgreSQL and DuckDB test suites to check the query plan correctness. DuckDB uses these statements to validate whether optimizations are expectedly performed [31]. Listing 5 shows SQL statements from DuckDB test cases. The EXPLAIN statement retrieves the optimized logical query plan, which describes how the DBMS retrieves data from the table `integers` and filters based on the specified predicate. Test cases using the EXPLAIN statement are unlikely to be reusable, because EXPLAIN is not a standard SQL statement, and the result formats of query plans differ between DBMSs [6].

BEGIN and ROLLBACK are statements that relate to transactions. Although START TRANSACTION is the standard SQL statement to start a transaction, we observed that these test suites more frequently use BEGIN. SQLite even lacks support for the standard syntax. Typically, a transaction is either committed using COMMIT (0.24%) or aborted using ROLLBACK (0.42%). Considering that ROLLBACK and COMMIT are standard SQL, less effort is needed to reuse these test cases.

Infrequently used SQL statements. An average of 10.6% statements are infrequently used in the test suite and thus not shown in Figure 2, such as WITH statements (0.48%) to define a *Common Table Expression* (CTE) and EXECUTE statements (0.39%) to run a stored procedure or prepared statement. We also found non-existent types of SQL statements. First, developers specified intentionally incorrect statements to test the parser of DBMS, and our analyzer did not attribute these statements to the correct type (*e.g.*, SELEC in DuckDB test cases). Second, as we used a best-effort parser to analyze the SQL statement, certain corner cases (*e.g.*, `(((((select * from int8_tbl)))))`) would

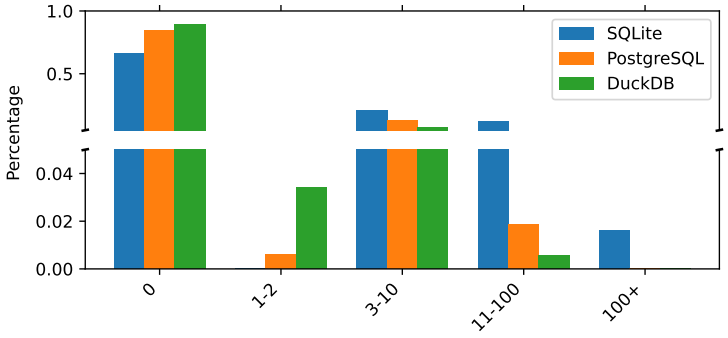


Fig. 3. Distribution of tokens in WHERE predicates of SELECT statements. 0 shows SELECT statements without WHERE clause.

Listing 6. Examples of SELECT statements in test suites

```
1 SELECT interval '1-2'; -- 0 token
2 SELECT a, b FROM t1 WHERE c > a; -- 3 tokens
3 SELECT unit.total_profit FROM unit, unit2; -- Implicit join
4 SELECT a, test.b, c FROM test INNER JOIN test2 ON test.b = 2 ORDER BY c; -- Inner
      join
```

be classified as ((((SELECT). However, these instances are rare, accounting for less than 0.1% of the total test cases, and can therefore be considered negligible.

SELECT *query complexity.* In terms of SELECT statements, we found that most are rather simple. This is demonstrated in Figure 3, which shows the number of tokens in WHERE predicates. Most queries, namely 79.9%, lack WHERE clauses, such as the example query shown in line 1 of Listing 6. Based on our understanding of inspecting the test cases, queries without WHERE clauses were often used to test the specific functions and operators by evaluating them on constants, without referencing any tables, or to fetch and validate the data from the table after specific operations. On average, 13.5% of queries contain 3 to 10 tokens in the predicates, such as the query shown in line 2 of Listing 6. 1.6% of SLT query predicates consist of more than 100 tokens, indicating the use of complex expressions. We also analyze the complexity of JOIN usage. Only 7.2% of queries consist of either implicit (*e.g.*, line 3 in Listing 6) or explicit joins (*e.g.*, INNER JOIN, LEFT JOIN and RIGHT JOIN). Specifically, 5.1% of queries contain implicit joins and 1.1% of queries contain INNER JOIN in the three test suites.

The results suggest that DBMSs typically test DBMS-specific functionality, such as illustrated by more than 80% of test files in PostgreSQL and DuckDB containing non-standard SQL statements. SQLite seems to be an exception, as 99.76% of statements are standard-compliant.

5 Test Case Dependencies (RQ3)

Overview. Table 4 shows the results of validation on the donor (*e.g.*, running SLT on SQLite) using SQuaLity. SQuaLity executed close to 6 million test cases from more than 7.4 million test cases

Table 4. Running donor test suites against donor

DBMS Names	Total Test Cases	Executed Cases	Failed Cases
SQLite	7,406,130	5,939,879	2
PostgreSQL	36,677	35,534	4,075
DuckDB	33,113	20,619	1,035

Table 5. Classification of a sample of 100 failing test cases for each DBMS and its test suite

	Reason	SQLite	DuckDB	PostgreSQL
Environment	File Paths	0	22	14
	Setting	0	0	7
	Set Up	0	0	67
Extension	Extension	0	0	10
Client	Format	0	58	0
	Numeric	0	17	0
	Exception	0	2	0
Misc	Runner	2	1	2

that we collected. The remaining tests included directives that caused them to be skipped. Failed cases refer to test cases where SQL statements behave unexpectedly, as introduced in Section 2. These were due to dependency issues, which indicate challenges for reuse. The detailed reasons are shown in Table 5, and we will discuss them in the subsequent paragraphs.

Pre-filtered test cases. As shown in Table 4, SQuaLity skipped 19.80%, 3.12%, and 39.47% of the test cases of SQLite, PostgreSQL, and the DuckDB test suite, respectively. SLT supports the *skiptest* runner command, which can be used when the test case is DBMS-specific. When testing against SQLite itself, some test cases were filtered intentionally, because they were designed to be executed on other DBMSs, and thus the test runner did not execute all the test cases. Similarly, in DuckDB, the runner command `require` halts all the following test cases if one extension has not been loaded. PostgreSQL test cases are not explicitly separated, as mentioned in Section 3. Thus, we omitted test cases that were difficult to parse.

Environmental settings. As shown in Table 5, in 22 out of the 100 sampled failed test cases in DuckDB, and in 88 for PostgreSQL, the test cases are dependent on specific environment settings. The difference between the environment of developers and ours leads to discrepancies between the actual and expected results. First, 22 of the DuckDB test cases and 14 of the PostgreSQL ones failed due to incorrect file path names that resulted in failures to load data, which affected the subsequent test cases. The data source was typically specified by hardcoded paths or paths set by environmental variables, and incorrect path names could result in no loaded data. Second, 7 PostgreSQL cases failed due to incorrect configurations. DBMSs support locale settings to manage internationalization, which led to the difference between actual results and expected results. Several other configurations including different system configurations, testing schema names, and default output format could also lead to discrepancies. Third, 67 failed PostgreSQL cases were due to the unsuccessful setup for the test database. For example, PostgreSQL used a scheduler to execute

Listing 7. This `CREATE FUNCTION` statement is used to create a function in current database by loading C library from "`regresslib`".

```
CREATE FUNCTION test_opclass_options_func(internal)
    RETURNS void
    AS :'regresslib', 'test_opclass_options_func'
    LANGUAGE C;
```

specific test cases first to set up certain environment settings. Conversely, test files in SQLite and DuckDB have no dependencies on each other and can be executed independently. Test cases associated with environmental settings pose challenges for reusability, as different DBMSs use diverse default configurations and environment setting methods.

Database engine extensions. 10 of the failing PostgreSQL test cases require extensions to be loaded, such as the shared library `regress.so`, which exposes functions such as shown in Listing 7, which are used in the subsequent test cases. Lack of or failure to load these functions could result in the failure of the current statement and subsequent test cases. Besides, as previously noted, DuckDB test cases use the runner command `require` to explicitly state their required extensions, and the remaining test cases are not executed if the required extension is not loaded. Thus, we encountered no failed test cases in our DuckDB samples due to this issue, since 26.2% of the cases have been pre-filtered. Extension-related test cases inhibit reusing, as, typically, extensions are DBMS-specific.

Clients. 77 of the failing DuckDB test cases rely on specific clients. Our selected DBMSs support various client APIs, and the respective test runners are implemented using different client APIs, C API for SQLite, C++ for DuckDB, and psql for PostgreSQL. Certain result objects are differently presented depending on the client, which led to discrepancies. Listing 8 shows an example of result discrepancies under different clients. The discrepancies mainly arise from variations in output formats, such as the representation of nested data types like lists and structs, binary objects, and numeric data, which affected 58 DuckDB test cases. Numeric data issues arise from the original test runner's lenient comparison, as our test runner demands an exact match with the expected results, given that it could provide consistency and catch subtle issues. This affected 17 DuckDB test cases. As exemplified in Listing 10, the expected result listed in the test case is 4999, while the true expected answer should be 4999.5. The discrepancies in result formats caused by different clients add to the difficulty of reusing the test cases.

Finally, we observed issues in clients, as shown in Listing 11. A `Not Implemented Error` arose in the DuckDB Python interface, while the same statements were executed as expected in the CLI, despite both being executed on version `0.8.1`. We found one case whose result differs between the CLI and Python clients (see Listing 9). We have reported this issue to the developers,[5] after which they implemented tests for various client connectors. These cases demonstrate that specific clients are required for some test cases.

Implications. Omitted test cases from the donor systems were due to dependency issues related to environment, extension, and client, as shown in Table 5. Dependencies on these components are often related to DBMS-specific features and configurations that are not defined by the SQL standard and vary between DBMSs (*e.g.*, one cannot use MySQL CLI on PostgreSQL). As a result, reusing these test cases across different DBMSs is challenging and provides limited benefits. We believe that, when designing a unified test suite for reuse, minimizing dependencies is crucial. Each test case should be independent of others and free from DBMS-specific features.

[5]https://github.com/duckdb/duckdb/issues/5413

Listing 8. Different results (after converting to string)

```
SELECT ARRAY[1,2,3,'4'];
-- Result in DuckDB
[1, 2, 3, 4]
-- Result in PostgreSQL
{1,2,3,4}
-- Result in DuckDB Python API
['1', '2', '3', '4']
```

Listing 9. Different results (after flattening values in a line) when executing the same query using the CLI and Python interfaces of DuckDB

```
SELECT 1 UNION ALL SELECT * FROM range(2, 100) UNION ALL SELECT 999 LIMIT 5;
-- CLI: 1 2 3 4 5
-- Python interface: 1 999 2 3 4
```

Listing 10. In DuckDB, floating-point results are considered matching if the difference is less than 1%

```
statement ok
create table quantile as select range r, random() from range(10000) union all
    values (NULL, 0.1), (NULL, 0.5), (NULL, 0.9) order by 2;

query I
SELECT median(r) FROM quantile -- Actual: 4999.5
----
4999
```

Listing 11. Executing these statements using the DuckDB Python interface results in an error

```
CREATE TABLE tbl1 (union_struct UNION(str VARCHAR, obj STRUCT(k VARCHAR, v INT)));
INSERT INTO tbl1 VALUES  ({'k': 'key1', 'v': 1});
SELECT * FROM tbl1; -- Expected: {'k': key1, 'v': 1}
```

We identified three major kinds of test case dependencies concerning the environment, extensions, and clients. The SQLite test cases require few dependencies. 88% of the failed PostgreSQL test cases were environment-related, and 77% of the DuckDB ones relied on specific clients.

6 Test Suite Compatibility (RQ4)

We executed the test cases from SLT, the PostgreSQL regression test suite, and the DuckDB test suite on SQLite, PostgreSQL, DuckDB, and MySQL. First, we present the overall execution results of these test suites. Second, we show the crashes and hangs found during execution, which are excluded from the above results. Third, we investigate the failed test cases found during execution. All the test cases have been reduced [47] for presentation.

Overall execution. Figure 4 shows a heatmap that shows the execution results of running the test suites across different DBMSs, excluding test cases that crash or hang the DBMS engine. SLT was the most compatible test suite, as all three other DBMSs passed over 98% of its test cases. The PostgreSQL regression suite was the most incompatible one. It obtained an average success rate of 28.1%. The DuckDB test suite obtained an average success rate of 45.2%. These results

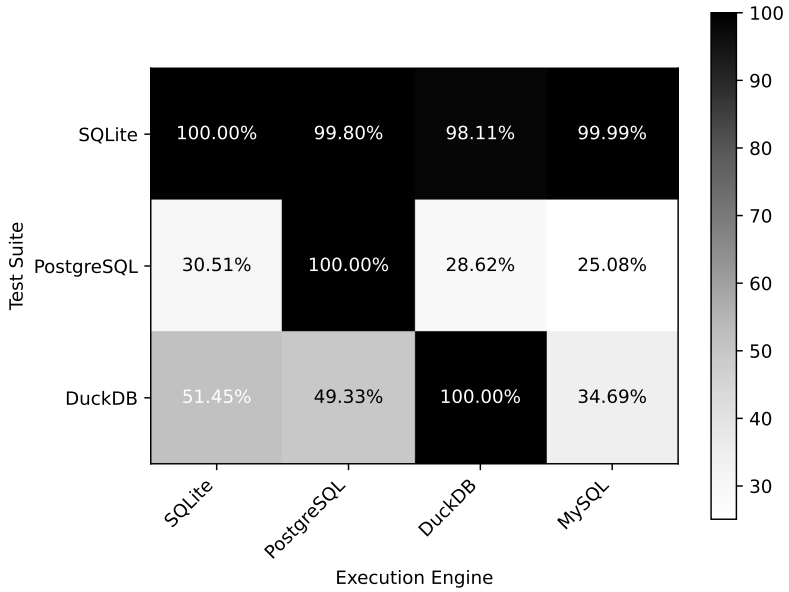


Fig. 4. The percentage of SQL test cases that execute successfully. At each intersection of *execution engine* and *test suite*, the color represents the success rate with respect to the total number of test cases. Darker shades indicate a higher success rate. For example, 51.45% of the DuckDB test cases are executed as expected on SQLite.

Listing 12. A crash in DuckDB

```
ALTER SCHEMA a RENAME TO b;
```

Listing 13. A crash related to transaction handling in DuckDB

```
CREATE TABLE a (b int);
BEGIN;
INSERT INTO a VALUES (1);
UPDATE a SET b = b + 10;
COMMIT;
UPDATE a SET b = b + 10;
```

Listing 14. A crash in a recursive CTE in MySQL

```
WITH RECURSIVE t(x) AS (
  SELECT 1
    UNION ALL
  (SELECT x+1 FROM t WHERE x < 4
    UNION
  SELECT x*2 FROM t WHERE x >= 4 AND x < 8)
) SELECT * FROM t ORDER BY x;
```

Listing 15. A recursive CTE led to an infinite loop

```
WITH RECURSIVE x(n) AS (
  SELECT 1 UNION ALL
  SELECT n+1 FROM x WHERE n IN (SELECT * FROM x)
) SELECT * FROM x;
```

Listing 16. An eponymous table-value function in PostgreSQL triggered an overflow in SQLite

```
SELECT count(*) FROM
generate_series(9223372036854775807,9223372036854775807);
```

were expected based on our analysis in Section 4, as the PostgreSQL regression suite contains the lowest percentage of standard-compliant SQL test cases and makes use of CLI commands, while SLT contains the most standard-compliant cases. MySQL achieved a higher success rate on SLT compared to the other two test suites, because of the `skiptest` runner commands in SLT (see Section 3), which restricted the runner to execute only general test cases and those implemented in the MySQL SQL dialect. We investigated the reason for the crashes and hangs, and examined the compatibility issues, which we explain in the next paragraphs.

Crashes. We found three crashes—unexpected terminations of the DBMS. Listing 14 shows a recursive CTE that crashed the MySQL server when executing `FollowTailIterator::Read()`, which has been addressed. Given the critical nature of the bug, it was assigned a CVE (2024-20962). We discovered this crash using the DuckDB test suite. Listing 12 shows a test case leading to a crash in DuckDB `0.7.0`. In the previous version (`0.6.1`), DuckDB correctly threw an `Not implemented Error`. Listing 13 shows a crash that was caused by `UPDATE` after the `COMMIT` of the transaction. We discovered these two crashes by executing the PostgreSQL regression test suite on DuckDB's latest release. While DuckDB had already accumulated a large test suite, the fact that we encountered these crashes highlights the importance of test case reuse.

Hangs. We observed three hangs—the DBMS entered an infinite loop when executing a statement or exhibited an overly-long execution time. We found one of them when executing a query from SLT on MySQL, which joined more than 40 tables. This caused MySQL to take more than one minute to compile and determine an efficient join order in the default setting, `optimizer_search_depth = 62`. By replacing this value with 0, the query could be executed within one millisecond. Although this behavior was expected according to the MySQL documentation,[6] the other three DBMSs could return results in a reasonable time with default settings. This suggests that users might need to change settings when migrating databases from other systems. We found another hang when executing one PostgreSQL test (`with.sql`) on DuckDB, as shown in Listing 15. PostgreSQL and MySQL prevented this query from being executed by issuing an error, while it caused an infinite loop in DuckDB. This hang reflects differences in the design philosophies behind the DBMSs. Developers mentioned it is a deliberate design for DuckDB, and adding constraints is strongly against DuckDB's friendly SQL [29] because it would restrict users. In addition, in a short, ad-hoc fuzzing campaign, in which we used the test suites as seed inputs, we discovered one hang (see Listing 16), which was confirmed as a bug[7] and subsequently fixed. The hang was triggered when executing the `generate_series()` function in SQLite. This function is a table-value function from PostgreSQL that SQLite implemented as an extension. This bug was introduced more than 3 years

[6]https://dev.mysql.com/doc/refman/8.0/en/controlling-query-plan-evaluation.html
[7]https://sqlite.org/forum/forumpost/754e2d

Table 6. Executing SQLite, DuckDB, PostgreSQL, and MySQL on the SLT, DuckDB test suite, and PostgreSQL regression test suite. We comprehensively analyzed the SLT test results, and randomly sampled and analyzed 100 test cases from DuckDB test suite and PostgreSQL regression test suite.

Failed Reasons	SQL Logic Test			DuckDB Test Suite			PostgreSQL Test Suite		
Granularity	DuckDB	PostgreSQL	MySQL	SQLite	PostgreSQL	MySQL	SQLite	DuckDB	MySQL
Statements	1,317	4,905	915	37%	25%	41%	54%	53%	58%
Functions	0	0	0	34%	26%	18%	11%	10%	8%
Types	43	12	0	13%	36%	22%	15%	23%	16%
Operators	7,075	6,069	0	14%	0%	10%	20%	3%	16%
Configurations	0	0	0	0%	3%	4%	0%	8%	2%
Semantic	104,033	609	0	2%	9%	2%	0%	3%	0%
Misc	0	0	0	0%	1%	3%	0%	0%	0%
Timeout	0	0	1	0	0	0	1	1	0
Crash	0	0	0	0	0	1	0	2	0

ago, which stresses the significance of test suite reuse, as it had been overlooked by other automated testing tools [35, 50].

Failed cases. Failed test cases are due to (1) unexpected execution statuses and (2) unexpected query results. When executing SLT on the three other hosts, 16.6% of the failed test cases were due to unexpected execution statuses, and the percentage is 94.9% and 96.0% for the DuckDB and PostgreSQL test suites. The reason for the higher execution success rate of SLT is that the SLT test cases contain mostly standard-compliant SQL statements, while the other two contain non-standard statements as well as DBMS-specific functions and data types. For example, the `pg_typeof()` function to obtain the data type of its argument is implemented by PostgreSQL and DuckDB, while no similar built-in function is supported in MySQL. Conversely, failed test cases in SLT were mostly due to unexpected query results, while the other two contain only a few. Certain functions or operators share the same name between DBMSs while the semantics are not the same, which mostly contributed to these mismatched query results. For example, when executing query `SELECT COALESCE(1, 1.0);`, SQLite returns the integer value 1, PostgreSQL returns the floating-point value 1, and both MySQL and DuckDB return the floating-point value `1.0`. However, all four DBMSs return the integer value 1 for `SELECT COALESCE(1, 1);`.

Incompatibility issues. We analyzed the root causes of the failed cases and categorized them into several types: unsupported (1) SQL statements, (2) functions, (3) types, (4) operators, as well as issues caused by (5) system settings, and (6) inconsistent semantics. We group together other reasons as a category miscellaneous, abbreviated as *misc*. Note that one failed test case might be due to multiple issues. We selected the first error type that we identified—often indicated by an error message from the DBMS. Table 6 shows the reasons for failed test cases. Each cell in the table represents the count or percentage of 100 samples of the failed cases. For example, when executing the DuckDB test suite on SQLite, 37% of the samples were due to unsupported SQL statements.

The unsupported *Statements* category refers to SQL statements from donor test cases that fail to execute on the host. These kinds of failures can occur for various reasons: either the host system lacks support for the statement, or there are differences in syntax and restrictions. We found 9, 30, and 26 of these kinds of issues when executing the SQLite, DuckDB, and PostgreSQL test suites, respectively. Fewer test cases in SLT failed for this reason compared to those in the DuckDB and PostgreSQL suites, which aligns with the findings from RQ2. First, non-standard SQL statements (*e.g.*, `PRAGMA` and `SET`), which we mentioned in Section 4, led to most of the unexpected failures.

Second, DBMS-specific clauses in statements also led to failures, for example, one unique feature `ASOF_JOIN` in DuckDB caused a failure. Third, DBMSs may apply different constraints on specific operations; for example, different restrictions concerning the `WITH` statement, which we mentioned in the paragraphs above. These statement-level issues can lead to unexpected failures during execution and are the primary cause of most of the failed cases we encountered.

The *Functions* category refers to predefined functions not supported on the host. No SLT test case contains an unsupported function, while 9 and 7 issues in DuckDB and PostgreSQL test suites, respectively, are related to unsupported functions, such as functions for querying system information, text processing, or nested data. For example, in DuckDB, `SELECT range(3);` returns a three elements list `[0,1,2]`, while the other DBMSs do not support `range()`.

The *Types* category refers to the donor-specific data types that are not supported in the host (*e.g.*, nested data and big integer). 1, 5, and 10 issues are related to types in the SQLite, DuckDB, and PostgreSQL test suites, respectively. One issue that caused MySQL to fail to execute the DuckDB and PostgreSQL test cases is its requirement to specify the maximum length for the `VARCHAR` type. Note that, SQLite had fewer issues in the Types category, as it has a dynamic type system,[8] which, for example, allows users to store values of any data type in a column, even if the declared column does not match the value's type. This is the reason why SQLite achieves a higher success rate on the PostgreSQL and DuckDB test suites than others.

The *Operators* category refers to unsupported donor-specific operators that either lead to unexpected syntax errors or encounter incompatible operand pairs. Examples include the type cast operator (`::`) mentioned above, and the unsupported `+` operator between string and integer in PostgreSQL, which is supported in SQLite. We identified 2, 3, and 6 issues in SQLite, DuckDB, and PostgreSQL test suite, respectively.

The *Configurations* category refers to unsupported configuration variables in the corresponding host. For example, one DuckDB test file initially uses `SET default_null_order='nulls_first';` to change the setting for the order of null values. This statement failed in PostgreSQL because it is an unknown system setting. As a result, when executing subsequent test cases in the same test file, PostgreSQL is unable to return values in the expected order.

The *Semantic* category refers to functions, operators, or statements that share the same name but exhibit semantic inconsistencies. Specifically, they return different values when executed under different DBMSs, leading to discrepancies in query results. We observed 5, 7 and 3 related issues in SQLite, DuckDB, and PostgreSQL test suite. Although SLT has a large number of failing tests due to semantic reasons, most of the cases have the same root cause. For example, all 104K failing cases for DuckDB are due to the inconsistent behavior for the `/` operator, which is a decimal division in DuckDB, but an integer division in SQLite. Some DBMSs would be built compatibly with other DBMSs (*e.g.*, DuckDB aims to partly match the semantics of PostgreSQL). SQuaLity could identify cases that exhibit inconsistencies, offering developers guidance, as these inconsistencies would not cause exceptions, but silently produce unexpected results, which are difficult to notice. Listing 17 and Listing 18 show differences in anonymous records sorting and eponymous functions (*e.g.* PostgreSQL functions starting with "pg_") respectively between DuckDB and PostgreSQL. We reported the issue in Listing 17 to the developers, and after a discussion, they concluded to deliberately deviate from other DBMSs like PostgreSQL.

Summary statistics. We present summary statistics of test cases of each test suite in Table 7. We manually classified the test cases based on the failures due to feature, syntax, and semantic differences. SLT is the most standard-compliant test suite, and thus only 12.9% of the failures were due to unique features or deviations in syntax. Conversely, the PostgreSQL and DuckDB test suites

[8] https://sqlite.org/flextypegood.html

Table 7. Summary of test cases from each test suite that bring difficulties for reuse

	SQLite	DuckDB	PostgreSQL
Dialect-specific features	0.1%	70.2%	72.7%
Syntax differences	12.8%	23.9%	26.4%
Semantic differences	87.1%	5.9%	0.9%

Listing 17. The result value of DuckDB is `true`, whereas other DBMSs return `null`.

```
SELECT (null, 0) > (0, 0);
-- DuckDB: true
-- PostgreSQL: null
```

Listing 18. DuckDB always return `true` even if passing invalid arguments to this function.

```
select has_column_privilege(1,1,1);
-- DuckDB: true
-- PostgreSQL: ERROR
```

consist of test cases for unique features, which caused most failures when executing test cases across different DBMSs.

Implications. We summarize the below implications about the test suite compatibility. First, it is difficult to reuse dialect-specific features, such as unique data types. Second, we believe that the syntax differences could be partially addressed by using SQL translators—potentially by using large language models—by converting SQL statements from one dialect to another to prevent errors. Third, manual effort is necessary to account for semantic differences—such as functions, operators, or statements that share the same name but exhibit semantic inconsistencies. These differences may arise from developers' different design choices or potential bugs in the system. Developer inspection of these cases is beneficial for identifying bugs and compatibility issues, as well as improving documentation. This effort is feasible, because semantic issues comprise only a small portion of all failures.

We reused three test suites across four DBMSs, which we used to find 3 crashes and 3 hangs. Sampling 100 failing tests, we identified various compatibility issues: 18 in SQLite, 59 in DuckDB, and 53 in PostgreSQL, which can be attributed to SQLite's test cases exercising mostly standard functionality, while the latter two comprised more DBMS-specific test cases.

7 Related Work

DBMS Testing. Methods have been proposed for automatically identifying bugs in DBMSs in recent years. SQLancer creates databases and queries, and then autonomously validates the results provided by the DBMS [4, 33–35]. Transformed Query Synthesis (TQS) [40] detected bugs of join optimization in DBMSs and TxCheck [19] detected transactional bugs by constructing graph-based oracles. APOLLO [20] identifies performance bugs by executing queries on both an older and a newer version of the same DBMS. Our test suite has the potential to augment the efficiency of these methods. In particular, it can be employed to serve as a basis for additional mutations. For example, Sedar [13] successfully detected crashes by reusing test cases from other DBMSs. However, in contrast to our work, Sedar only re-used the SQL statements, and not their results specification.

Studies on SQL features. Several empirical studies have been conducted that closely relate to our study. Cui et al. [11] studied transaction implementations and found transactional compatibility issues across DBMSs using a differential-testing methodology, while our study has a more general scope in identifying compatibility issues. Gupta et al. [17] investigated the use of procedural extensions in SQL—specifically, stored procedures, user-defined functions (UDFs), and triggers—in the context of Microsoft Azure SQL Database Service. They subsequently developed a benchmark suite and evaluated these procedural extensions, that is, the time/resources diverse procedural SQL components spent in a given workload. Toussaint et al. [42] conducted a survey on the usage of `NULL` values among DBMS users. They examined the drawbacks associated with `NULL` values in SQL and how they are handled. Vogelsgesang et al. [43] analyzed the characteristics of queries generated by a modern *business intelligence* (BI) tool, and found that they differ from benchmarks used in the industry. Both the above work and our study investigated how SQL is used in practice. However, our study primarily focuses on the testing perspective. We have examined the SQL test suites, aiming to reuse the test cases in order to discover new bugs within the DBMSs and enhance their reliability.

Programming language standards. Our study gave quantitative and qualitative evidence of how SQL dialects differ in practice. Various studies have been conducted to explore the standards of SQL and different programming languages, or formalize them. Guagliardo et al. [16] addressed the lack of formal semantics for real-world SQL queries and provided formal semantics for a basic class of SQL queries. Memarian et al. [27] provided an in-depth investigation into the semantics of pointers and memory in C. Furthermore, they extended their study concerning pointer provenance [26]. They examined the discrepancies between the actual C standard and what expert C programmers believe the standard ensures, while we focus on discrepancies between different DBMSs.

Test reusability. Various studies have been conducted on software reuse. Tiwari et al. [41] argued the importance of software reuse and discussed approaches to test reuse. They mentioned test reuse could reduce the test effort. Recently, test reuse has been proposed for UI testing. Zhao et al. [49] introduced a framework that could automatically evaluate UI test reuse. Mariani et al. [25] conducted the first empirical study on GUI events' semantic matching and reported significant findings to enhance test reuse approaches. Regression test suites have long been a resource for mutation-based testing. Le et al. [22] found bugs in LLVM by using test cases from GCC. Zhong et al. [50] used the test suite of each DBMS as the seed corpus for their mutation-based fuzzing approach, which found many bugs in DBMSs.

8 Threats to validity

We used a common methodology [12] to identify and mitigate potential threats to our work.

Internal validity. The main threat to the internal validity of this work is potential inaccuracies in the manual classification of failed test cases in RQ3 and RQ4, which could introduce bias, leading to misclassification of the reasons for failed cases. We alleviate this threat by carefully investigating the documentation and discussing any unclear reasons among the authors. Where we still failed to determine the reasons after discussion and studying the user manuals and implementation of the DBMSs, we reported the issue to DBMS developers. Thus, we expect that any misclassifications are unlikely and would have little impact on the overall results.

Construct validity. The main threat to construct validity is that we designed our own test suite, whose design decisions might affect the analyses. For instance, we chose to execute all test cases using Python clients, and in Section 5, we found that test cases can compute different results depending on which client is used. Implementing test case translators and using the original test

Table 8. Coverage of executing each original test suite and SQuaLity on SQLite, DuckDB, and PostgreSQL

	SQLite		DuckDB		PostgreSQL	
	Line	Branch	Line	Branch	Line	Branch
Original tests	26.9%	19.8%	72.8%	46.4%	62.1%	47.2%
SQuaLity	43.4%	34.5%	74.0%	47.2%	63.0%	48.2%

runner (*e.g.*, parsing DuckDB test cases to run on SQLite via the SLT runner) would not scale for a larger number of test suites, as every test runner would need to support every test case format. Despite this potential threat, we believe that the high-level insights of the study are valid even when considering such different potential experimental setups.

External validity. The external validity of our study might be constrained by the limited scope of the open-source DBMSs that we examined. We investigated four widely used open-source DBMS test suites and reused three of them. However, commercially developed systems, for example, Oracle Database, might have more extensive test suites since companies might have more resources. To mitigate this issue, we have also investigated the test suite of CockroachDB—a DBMS mainly developed commercially (by Cockroach Labs)—and observed similar trends there. Additionally, the evolution of DBMSs might lead to changes in the test suites we studied, potentially limiting the applicability of our results over time.

9 Implications

Based on our study, we have identified important actionable insights, which we subsequently summarize.

Overcoming scalability challenges. While being the first systematic effort in test reuse of DBMSs, SQuaLity has not fully addressed the issues of unifying SQL test suites. DBMS test suites contain a variety of different test runner commands that need to be supported in our design. We believe that supporting various test case formats and runner commands, as attempted in our work, is non-scalable due to the amount of manual implementation effort that is involved. One pragmatic, alternative approach to integrating various existing test suites based on different runner commands could involve removing all test runner commands from the test cases, executing them on the *donor* system, and assuming the outputs as the ground truth. However, this approach would necessitate manual inspection of the results to validate whether they are correct.

Test suites for new DBMSs. For newly developed DBMSs, we recommend adopting the SQLite test format and test suite. The reason for this is three-fold: (1) the test format is relatively simple (see Section 3), facilitating the implementation of the test runner required to execute the test suite on the DBMS; (2) SQLite's tests are mostly standard-compliant, with only few testing SQLite-specific functionality, allowing their reuse (see Section 4); and (3) compared to other systems, they have relatively few dependencies (see Section 5). We found that, besides SQLite and DuckDB, also other DBMSs, such as MonetDB[9] and Databend[10] use the SLT format in their test suites. As an alternative, for DBMSs designed to be (partly) compatible with a potential *donor* DBMS, we advise utilizing its test suites, as they can find both bugs and compatibility issues (see Section 6).

Benefits of reusing test suites. Our study provides the below key insights for DBMS developers. First, SLT, which contains mainly standard-compliant SQL statements, can be adapted to different

[9]https://github.com/MonetDB/MonetDB/tree/master/sql/test
[10]https://github.com/datafuselabs/databend

DBMSs with a higher success rate, and could thus help detect logic errors for basic features during development.[11] Second, other test suites often contain complex and DBMS-specific features that may restrict or even prohibit test case reuse; despite this, we observed that some features are shared across different systems and the test cases are valuable for finding potential bugs. Third, reusing the composed test suite SQuaLity can help increase test coverage compared to using only the original test suite, which we determined in an additional coverage experiment (see Table 8). We explain this increase by the larger range of SQL statements and SQL features from different test suites that can improve coverage for features not covered in the existing test suite as well as by testing error handling of both parsing and semantic analysis when processing unsupported or partly-supported features. SLT achieves a branch coverage of only 20% on SQLite. This is due to SLT containing mostly standard-compliant SQL commands. SQLite has three harnesses. For example, the 100% branch coverage for SQLite is achieved by another test suite of SQLite, TH3,[12] which not only contains SQL test cases, but also test cases written in C. Although SQLite and SLT are open-sourced, the TH3 test suite is proprietary.

Supporting a new DBMS. Applying SQuaLity on a new DBMS requires supplementing the general test runner implementation with DBMS-specific logic, as also explained in our artifact. Specifically, interfaces need to be implemented, which establish connections to the DBMS, set up an initial database and subsequently remove it, and execute statements and queries. Implementing them typically requires a low effort, as they are implemented in 33 LOC on average for the DBMSs in our experiments. Executing SQuaLity on a new DBMS is likely to cause test failures, caused by different root causes. First, crashes are never expected. Hangs, which are also abnormal, require developers to inspect them—they can be missed optimization or bugs. Second, when the DBMS under test is designed for compatibility with existing dialects (*e.g.*, PostgreSQL), developers are advised to inspect any inconsistencies provoked by the test cases from the DBMSs they aim to be compatible with. The discrepancies could indicate a potential bug since the semantics should match. For failures in test cases with dialect incompatibilities, the possibility of finding bugs is lower, and such cases could be removed. Third, developers can identify patterns in unexpected error messages to identify potential bugs. For example, messages starting with "`INTERNAL Error`" are never expected in DuckDB, and suggest bugs. Matching the patterns (*e.g.*, using regular expressions) of the error messages in the execution log can help detect issues after running SQuaLity.

Filling testing gaps. Our study provides actionable insights on what features are undertested, warranting additional test cases to be written. For example, the SQL statements distribution in RQ2 suggests that only few test cases exercise `WITH` statements (0.48%), which are used to specify CTEs. In particular, we found a bug in MySQL (see Listing 14), which was identified and assigned CVE-2024-20962, due to an issue caused by a recursive CTE. This suggests that this feature, and other statements for which only few test cases exist, might be insufficiently tested.

Understanding SQL dialects. While SQL is commonly seen as a single language, various SQL dialects exist as our experiments indicate. For example, previous research [16] on formalizing SQL semantics has focused on general semantics, rather than accounting for different SQL dialects. We identified various incompatibility issues when executing test suites across different DBMSs—18 in SQLite, 59 in DuckDB, and 53 in PostgreSQL (see RQ4)—due to different reasons (*e.g.*, incompatible statements, operators, and types). These results might inform variability points for formal semantics.

[11] https://github.com/dolthub/dolt/issues/7079
[12] https://sqlite.org/th3.html

Enhancing automated testing. Many DBMS fuzzing approaches require a seed corpus, based on which they derive follow-up inputs through mutation, and our test suite—containing more than 7 million SQL test cases—could be used as such a corpus. Recent work has demonstrated the benefits of this. For example, Sedar [13] has detected bugs in well-known DBMSs by reusing SQL test cases.

10 Conclusion

In this paper, we have contributed a unified test suite consisting of more than 7 million SQL statements from three test suites of widely used DBMSs as well as an empirical study concerning different test suite features, test case patterns, and testing dependencies. Our findings for test suite reuse are encouraging. We identified crashes and hangs by executing test cases written for one DBMS on other DBMSs, indicating the benefits of test-case reuse. However, we also identified various challenges that complicate test reuse. This includes the feature variety in test suite formats, test runner commands (up to 112 commands, RQ1), SQL dialects (up to 31% of the test cases in PostgreSQL are not standard-compliant, RQ2), and environmental dependencies (such as file path and locale configurations, RQ3). We hope that this first study toward test case reuse for DBMSs will inspire follow-up work and inform practitioners of the opportunities and challenges of reusing test cases.

Acknowledgements

This research is supported by the Ministry of Education, Singapore, under the Academic Research Fund Tier 1 (FY2023) as well as the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Fuzz Testing). Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, and Cyber Security Agency of Singapore.

References

[1] 2022. Sqllogictest. https://www.sqlite.org/sqllogictest/doc/trunk/about.wiki
[2] 2023. ISO/IEC 9075 "Information technology - Database languages - SQL".
[3] 2023. MySQL Test Commands. https://dev.mysql.com/doc/dev/mysql-server/latest/PAGE_MYSQL_TEST_COMMANDS.html Accessed: 2023-8-8.
[4] Jinsheng Ba and Manuel Rigger. 2023. Testing Database Engines via Query Plan Guidance. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 2060–2071. https://doi.org/10.1109/ICSE48619.2023.00174
[5] Jinsheng Ba and Manuel Rigger. 2024. CERT: Finding Performance Issues in Database Systems Through the Lens of Cardinality Estimation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (, Lisbon, Portugal,) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 133, 13 pages. https://doi.org/10.1145/3597503.3639076
[6] Jinsheng Ba and Manuel Rigger. 2024. An Exploratory Case Study of Query Plan Representations. *arXiv preprint arXiv:2408.07857* (2024).
[7] Sebastian Baltes and Paul Ralph. 2022. Sampling in software engineering research: A critical review and guidelines. *Empirical Software Engineering* 27, 4 (2022), 94.
[8] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated Software Transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 257–269. https://doi.org/10.1145/2771783.2771796
[9] Alexander Böhm, Maria Christakis, Eric Lo, and Manuel Rigger. 2022. Ensuring the Reliability and Robustness of Database Management Systems (Dagstuhl Seminar 21442). In *Dagstuhl Reports*, Vol. 11. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.
[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2, Article 4 (jun 2008), 26 pages. https://doi.org/10.1145/1365815.1365816

[11] Ziyu Cui, Wensheng Dou, Qianwang Dai, Jiansen Song, Wei Wang, Jun Wei, and Dan Ye. 2023. Differentially Testing Database Transactions for Fun and Profit. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 35, 12 pages. https://doi.org/10.1145/3551349.3556924

[12] Davide Falessi, Natalia Juristo, Claes Wohlin, Burak Turhan, Jürgen Münch, Andreas Jedlitschka, and Markku Oivo. 2018. Empirical software engineering experts on the use of students and professionals in experiments. *Empirical Software Engineering* 23 (2018), 452–489.

[13] Jingzhou Fu, Jie Liang, Zhiyong Wu, and Yu Jiang. 2024. Sedar: Obtaining High-Quality Seeds for DBMS Fuzzing via Cross-DBMS SQL Transfer. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 146, 12 pages. https://doi.org/10.1145/3597503.3639210

[14] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2023. Griffin: Grammar-Free DBMS Fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 49, 12 pages. https://doi.org/10.1145/3551349.3560431

[15] Goetz Graefe et al. 2011. Modern B-tree techniques. *Foundations and Trends® in Databases* 3, 4 (2011), 203–402.

[16] Paolo Guagliardo and Leonid Libkin. 2017. A Formal Semantics of SQL Queries, Its Validation, and Applications. *Proc. VLDB Endow.* 11, 1 (sep 2017), 27–39. https://doi.org/10.14778/3151113.3151116

[17] Surabhi Gupta and Karthik Ramachandra. 2021. Procedural Extensions of SQL: Understanding their usage in the wild. *Proceedings of the VLDB Endowment* 14, 8 (2021), 1378–1391.

[18] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 4949–4965. https://www.usenix.org/conference/usenixsecurity23/presentation/jiang-zu-ming

[19] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. 2023. Detecting Transactional Bugs in Database Engines via Graph-Based Oracle Construction. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 397–417. https://www.usenix.org/conference/osdi23/presentation/jiang

[20] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. APOLLO: Automatic detection and diagnosis of performance regressions in database systems. *Proceedings of the VLDB Endowment* 13, 1 (2019), 57–70.

[21] Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proc. VLDB Endow.* 14, 3 (nov 2020), 268–280. https://doi.org/10.14778/3430915.3430918

[22] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 216–226. https://doi.org/10.1145/2594291.2594334

[23] Yu Liang, Song Liu, and Hong Hu. 2022. Detecting Logical Bugs of DBMS with Coverage-based Guidance. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 4309–4326. https://www.usenix.org/conference/usenixsecurity22/presentation/liang

[24] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. 2022. Automatic Detection of Performance Bugs in Database Systems Using Equivalent Queries. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 225–236. https://doi.org/10.1145/3510003.3510093

[25] Leonardo Mariani, Ali Mohebbi, Mauro Pezzè, and Valerio Terragni. 2021. Semantic Matching of GUI Events for Test Reuse: Are We There Yet?. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 177–190. https://doi.org/10.1145/3460319.3464827

[26] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. 2019. Exploring C Semantics and Pointer Provenance. *Proc. ACM Program. Lang.* 3, POPL, Article 67 (jan 2019), 32 pages. https://doi.org/10.1145/3290380

[27] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the Depths of C: Elaborating the de Facto Standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/2908080.2908081

[28] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.* 17, 1 (mar 1992), 94–162. https://doi.org/10.1145/128765.128770

[29] Alex Monahan. 2022. Friendlier SQL with DuckDB. https://duckdb.org/2022/05/04/friendlier-sql.html

[30] Thomas Neumann and Bernhard Radke. 2018. Adaptive Optimization of Very Large Join Queries. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 677–692. https://doi.org/10.1145/3183713.3183733

[31] Mark Raasveldt. 2022. DuckDB Testing - Present and Future. Keynote speech at DBTest '22: Proceedings of the 2022 workshop on 9th International Workshop of Testing Database Systems.

[32] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.

[33] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1140–1152. https://doi.org/10.1145/3368089.3409710

[34] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 211 (nov 2020), 30 pages. https://doi.org/10.1145/3428279

[35] Manuel Rigger and Zhendong Su. 2020. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 667–682.

[36] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '79)*. Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/582095.582099

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

[38] Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 618–622.

[39] solid IT. 2023. DB-Engines Ranking. https://db-engines.com/en/ranking

[40] Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. 2023. Detecting Logic Bugs of Join Optimizations in DBMS. *Proc. ACM Manag. Data* 1, 1, Article 55 (may 2023), 26 pages. https://doi.org/10.1145/3588909

[41] Rajeev Tiwari and Noopur Goel. 2013. Reuse: Reducing Test Effort. *SIGSOFT Softw. Eng. Notes* 38, 2 (mar 2013), 1–11. https://doi.org/10.1145/2439976.2439982

[42] Etienne Toussaint, Paolo Guagliardo, Leonid Libkin, and Juan Sequeda. 2022. Troubles with Nulls, Views from the Users. *Proc. VLDB Endow.* 15, 11 (jul 2022), 2613–2625. https://doi.org/10.14778/3551793.3551818

[43] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *Proceedings of the Workshop on Testing Database Systems* (Houston, TX, USA) *(DBTest'18)*. Association for Computing Machinery, New York, NY, USA, Article 1, 6 pages. https://doi.org/10.1145/3209950.3209952

[44] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a Learning Optimizer for Shared Clouds. *Proc. VLDB Endow.* 12, 3 (nov 2018), 210–222. https://doi.org/10.14778/3291264.3291267

[45] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do Not Blame Users for Misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 244–259. https://doi.org/10.1145/2517349.2522727

[46] YouTube. 2020. DuckDB – the SQLite for analytics (Mark Raasveldt, CWI). https://www.youtube.com/watch?v=PFUZlNQIndo. Accessed: 2024-07-19.

[47] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why? *SIGSOFT Softw. Eng. Notes* 24, 6 (oct 1999), 253–267. https://doi.org/10.1145/318774.318946

[48] A. Zeller and R. Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200. https://doi.org/10.1109/32.988498

[49] Yixue Zhao, Justin Chen, Adriana Sejfia, Marcelo Schmitt Laser, Jie Zhang, Federica Sarro, Mark Harman, and Nenad Medvidovic. 2020. FrUITeR: A Framework for Evaluating UI Test Reuse. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1190–1201. https://doi.org/10.1145/3368089.3409708

[50] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 955–970. https://doi.org/10.1145/3372297.3417260

Received April 2024; revised July 2024; accepted August 2024