# TESTING DATABASE ENGINES VIA QUERY PLANS

JINSHENG BA

NATIONAL UNIVERSITY OF SINGAPORE

2024

# TESTING DATABASE ENGINES VIA QUERY PLANS

by

**JINSHENG BA**
*(B.S., Shandong University)*

**A THESIS SUBMITTED FOR THE DEGREE OF**

**DOCTOR OF PHILOSOPHY**

in

**COMPUTER SCIENCE**

in the

**SCHOOL OF COMPUTING**

of the

**NATIONAL UNIVERSITY OF SINGAPORE**

**2024**

Supervisor:
Dr Manuel Rigger

Examiners:
Dr Umang Mathur
Professor Dong Jin Song
A/P Jiang Yu, Tsinghua University

# Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Jinsheng Ba

18 July 2024

# Acknowledgments

Staying on the island of endless summer, I feel like the day of enrollment is just yesterday. Until writing this thesis, I realized the endless summer has an end. Looking back over the past several years, I have experienced the despair of making no progress and the joy of winning awards. All these experiences helped me grow up. I would not have come so far without the tremendous help I received.

First of all, I would like to express my deepest gratitude to my PhD advisor, Dr. Manuel Rigger for continuously guiding and supporting me in the past several years. I learned a lot of critical knowledge about how to research during the second half of my PhD journey. Manuel taught me what valuable research is and how to do rigorous research. His meticulous feedback on every sentence I crafted and every presentation slide I prepared has been immensely beneficial. Moreover, he has consistently demonstrated patience and provided unconditional support for all my endeavors. The impact of his mentorship resonates through every page of this thesis and will continue to influence the papers I produce in the future.

I extend my heartfelt gratitude to Dr. Marcel Böhme for his invaluable guidance and mentorship during the first half of my PhD. As a novice PhD candidate, I encountered challenges navigating the intricacies of research, and Marcel's unwavering support proved indispensable. He has been patient in helping me even with basic inquiries. He set a good example for me in terms of research, teaching, and productivity. Without his assistance, I would not have been able to progress on my PhD.

I am thankful to Professor Jin-Song Dong, Dr. Umang Mathur, and Associate Professor Yu Jiang for agreeing to serve on my thesis committee, despite their busy schedules. I am grateful to receive valuable feedback from them on the improvement of the thesis. Additionally, I would also like to thank them for assisting my graduation process.

I would like to thank other researchers who have been pivotal in shaping my academic journey. I thank Thuan Pham, Gregory J. Duck, and Zahra Mirzamomen for collaborating on research work during my PhD. I thank Zhen Dong, Xiang Gao, and Ilya Sergey for giving me constructive feedback on my research. I thank Jialin Li and Yannic Noller for providing valuable suggestions for my academic path. Their

professionalism, passion, and kindness inspired me a lot.

I would like to thank all my friends and labmates: Wenjing Deng, Mingyuan Gao, Ivan Ho, Nathee Jaywaree, Haoxiang Jiang, Mingliang Jiang, Yuancheng Jiang, Jinhao Dong, Xiang Liu, Qiuyang Mang, Reza Qorib, Suwei Yang, Xiaoliang Yu, Albert Zhang, Anxing Zhang, Chi Zhang, Yakun Zhang, and Suyang Zhong. They created many unforgettable memories and made my stay in Singapore such a wonderful experience.

I would like to thank my family. I thank my parents for raising me to be more than I could be. They always support me unconditionally, so that I can pursue dreams without distractions. I would also like to express my biggest thanks to my wife, Ms. Jiaxuan Li, for her love, her accompany, her support, her encouragement, and for putting up with many troubles that are due to my academic path. I am lucky to have her, as well as our bunny.

I leave the final thanks to myself, for the efforts I have made toward my dream, for never giving up no matter how hard it was, and for stumbling along the way, but still loving life. I have gone through many phases to become who I am now. Lost and found, all encounters are treasures. In the future, I believe there are still more towering mountains waiting for me. I hope that no matter the hardships ahead, I will move forward without changing my original intention.

Finally, thank you for reading. If this thesis is of any help, I would be honored.

(All the names are listed in alphabetical order).

# Contents

# Abstract

Testing Database Engines via Query Plans

by

Jinsheng Ba

Doctor of Philosophy in Computer Science

National University of Singapore

Database Management Systems (DBMSs) are fundamental software systems that store, maintain, and retrieve data. They are used in almost every personal computer, mobile device, and server. Therefore, it is important to find bugs before they incur severe consequences. Automatic testing is an efficient and effective technique to find crash bugs, which terminate DBMSs, but is struggling to detect logic bugs and performance issues. Logic bugs refer to incorrect results, while performance issues refer to unexpected slow performance. Unlike crash bugs, both categories of bugs do not terminate DBMSs and are hard to observe by existing automatic testing methods. Triggering them requires valid test cases, which are also challenging to generate automatically. In this thesis, I advance automatic testing to efficiently find logic bugs and performance issues in DBMSs. My approaches are united by the idea of leveraging query plans, which are internal representations of how a DBMS executes a query, for automatically testing DBMSs. I put forward the following thesis statement: *Query plans allow efficient and effective testing of DBMSs by providing internal execution information.* I propose four research works to utilize query plans for testing. First, to detect performance issues, I propose *Cardinality Estimation Restriction Testing* (*CERT*), which inspects estimated cardinalities in query plans without measuring execution time. Second, to identify logic bugs, I propose *Differential Query Plans* (*DQP*), which inspects the result consistency of multiple query plans of the same query. Third, to generate diverse test cases for exploring target systems thoroughly, I propose *Query Plan Guidance* (*QPG*) for guiding the test case generation process towards diverse query plans. Last, observing that query plans cannot be conveniently used as they are exposed in various DBMS-specific representations, I propose a *Unified query plan representation* (*UPlan*) based

on an empirical study aiming to reduce the effort of building applications based on query plans. Since most DBMSs—including commercial ones—expose query plans to the user, I consider *CERT*, *DQP*, *QPG*, and *UPlan* generally applicable, black-box approaches for finding logic bugs, performance issues, and building applications on query plans. These methods are effective as they found more than 100 unique and previously unknown bugs in several widely used DBMSs. I view these results as a step towards more reliable DBMSs, and expect this statement of utilizing query plans for testing can be widely adopted to tackle more problems, such as test suite evaluation, debugging deployed DBMSs, and optimization checking.

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Problem

Database Management Systems (DBMSs) are fundamental software systems used to store, retrieve, and query data. The global DBMSs market has grown to $163.93 billion in 2023 at a high compound annual growth rate of 15.4% [148]. DBMSs are used in almost every computing device [185, 187, 184], thus any bug has a potentially severe consequence. The problem that this thesis tackles is *how to effectively and efficiently find bugs in DBMSs for improving reliability.* The growth of the complexity of DBMSs has exacerbated this problem. As a concrete example, *MySQL* has been evolving over the last twenty years to a massive 15 million lines of code, and was disclosed 581 bugs in only 2022 according to its issue tracker.[1]

## 1.2 Challenges and State of the Art

Automatic testing is a practical and widely used method for finding bugs. However, it is typically effective in finding crash bugs, which terminate DBMSs, but is struggling to find logic bugs and performance issues. Logic bugs refer to incorrect results, while performance issues refer to unexpected slow performance. Finding them by testing faces two major challenges:

**C.1 Text Oracle**. Test oracle refers to a mechanism that determines whether the system executes correctly for an input. Unlike crash bugs, which terminate

[1] `https://bugs.mysql.com/`

Code 1.1: An example of logic bugs and performance issues.

```
1 CREATE TABLE users(user_id DECIMAL PRIMARY KEY);
2 CREATE TABLE transactions(transitition_id TEXT, amount DECIMAL(10,2) NOT NULL);
3 CREATE INDEX i0 ON transactions(transitition_id(5));
4 INSERT INTO users VALUES(1), (2);
5 INSERT INTO transactions VALUES('1_c12934', 100000), ('1_e3b664', -10);
6
7 SELECT IFNULL(SUM(amount), 0) as balance FROM users JOIN transactions ON
      transactions.transitition_id = users.user_id; -- {0.00}
8 SELECT IFNULL(SUM(amount), 0) AS balance FROM users JOIN transactions ON
      transactions.transitition_id = users.user_id and
      transactions.transaction_id < 1000; -- 5 minutes
```

the system, logic bugs, which refer to incorrect results, and performance issues, which refer to unexpected slow performance, are challenging to identify. Code 1.1 shows an artificial example of both categories of bugs. It first creates two tables `users` and `transactions` to simulate a bank account system, and inserts two users and transaction records into tables. The first query in line 7 checks the balance of the user `1`, and triggers a logic bug in the DBMS. The balance should be `99990.00`, instead of `0.00`. Without manually inspecting the test case, it is challenging to know whether the result is expected. The second query in line 8 returns the execution result in 10 seconds, and triggers a performance issue in the DBMS. Similarly, it is challenging to know whether the execution time is expected.

**C.2 Test generation**. DBMSs typically accept the statements written in structured languages as inputs, so it is difficult to automatically construct test cases that satisfy the semantic and syntax requirements of the languages and cover diverse DBMS components. To construct the test case shown in Code 1.1, we need to guarantee correct syntax and semantics for the test cases. To find more bugs, we also need other diverse test cases to find bugs in different components of target systems.

To tackle both challenges, multiple methods and techniques were proposed to identify logic bugs and performance issues, and generate diverse and valid test cases. However, these methods struggle to efficiently and effectively test complex program logic in DBMSs.

For **C.1**, to identify logic bugs, differential and metamorphic testing are widely

used. Differential testing identifies bugs by comparing the outputs or behaviors of multiple systems for the same test cases. *RAGS* [161] applied differential testing to examine whether executing the same queries on various DBMSs returns the same results. Metamorphic testing constructs a pair of semantic-related queries and checks whether the results of executing them comply with a predefined relation, *NoREC* [149] and *TLP* [150] applied metamorphic testing to examine the consistency of executing pairs of semantic-equivalent queries on the same DBMS. *DQE* [162] applies metamorphic testing to find bugs in data manipulation statements. Apart from differential and metamorphic testing, *TQS* [167] splits a given input table into several sub-tables and validates the results of the queries that join these sub-tables by retrieving the given table. However, these methods have strict requirements for test cases, such as *NoREC* requires test cases to include `WHERE` clause, so these methods mostly randomly generate test cases, and limited program logic can be tested. Additionally, some methods, such as *TQS*, are sophisticated and not easy to understand and implement.

For **C.1**, to find performance issues, differential and metamorphic testing are also widely used. *APOLLO* [89] is a differential testing method that examines whether the same test cases exhibit similar performance under different versions of the same DBMSs. *AMOEBA* [107] is a metamorphic testing method that examines whether semantic-equivalent queries have similar performance on the same DBMS. However, *APOLLO* can only find regression bugs, which are introduced in the newer version of tested DBMSs. *AMOEBA* has a high false alarm rate, because complex program logic may optimize semantic-equivalent queries significantly differently and is not considered for testing.

For **C.2**, to generate diverse test cases, fuzzing is a widely used method. Fuzzing is a method that generates or mutates inputs for finding memory-related bugs, which typically crash target programs. General fuzzers, such as *AFL* [180] and *LibFuzzer* [181], use code coverage to guide the test case generation process. *Squirrel* [219] and *DynSQL* [82] considered dependencies among statements for mutating test cases. *LEGO* [104] considered statement types for mutating test cases. *Sedar* [50] exchanged test cases among multiple DBMS testing suites for high-quality initial seeds. These methods aim to generate diverse test cases but cannot find logic bugs and performance issues.

For **C.2**, to generate valid test cases for validating the test oracles, grammar-based test case generation methods are widely used. *NoREC*, *TLP*, and *SQLSmith* use a grammar-based generator to randomly generate test cases. However, randomly generated test cases are challenging to cover complex program logic. *AFLRight* [106] mutate test cases according to grammar definitions. However, random mutation easily incurs invalid semantics, so complex program logic is also challenging to test.

## 1.3 Thesis Statement

In this thesis, I propose to advance automatic testing for efficiently finding logic bugs and performance issues in DBMSs by the idea of leveraging query plans to facilitate testing:

**Thesis Statement**

Query plans allow efficient and effective testing of database engines by providing internal execution information.

A query plan is a tree of operations that describes how a statement is executed by a DBMS. Query plans are DBMSs' internal representations and provide a compact and high-level summary of how a query is executed. Our core idea is to use query plans for efficiently and effectively testing complex program logic in DBMSs, such as exploring diverse program behaviors and identifying bugs.

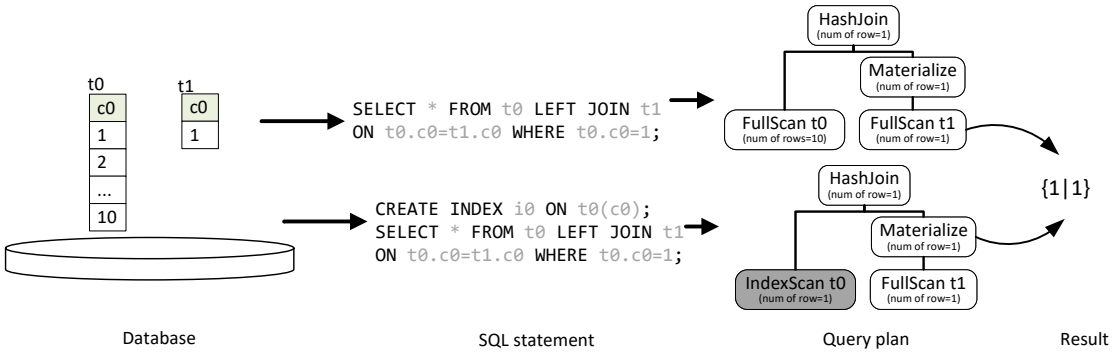


Figure 1.1: Query plan example.

Figure 1.1 shows a query plan example in *PostgreSQL*. Suppose we have two tables in the database, table `t0` has one column `c0` with ten rows, and table `t1` has one

column `c0` with one row. The query `SELECT * FROM t0 LEFT JOIN t1 ON t0.c0=t1.c0 WHERE t0.c0=1` retrieves the data of joining both tables and filters the joined data. Because the query is written in a declarative language, it has to be translated into a query plan before executing, as shown in the right top corner. Each node in the tree of the query plan includes an operation and associated properties. A query plan is executed from root to leaves, and the execution results are returned from leaves to root. In this query plan, `FullScan` in the leaves represents reading all data from tables. `num of row` represents the estimated number of rows that will be returned after this operation. `Materialize` stores data in memory so that its parent operation processes data more efficiently. In the root node, `HashJoin` joins the data from `FullScan t0` and `Materialize`. At last, DBMS executes this query plan and returns the result.

DBMSs commonly expose query plans for tuning performance by executing the statement `EXPLAIN` in a black-box manner. In Figure 1.1, we can examine the query plans to identify potential optimization opportunities. The `num of rows` of `FullScan t0` is ten. We can reduce the number to improve the performance by adding an index. If we execute `CREATE INDEX i0 ON t0(C0)` to create an index and execute the same query again, we can obtain the second query plan as shown in the right-down corner. Using the index, DBMS scans the table `t0` by `IndexScan`, which only reads part of table `t0`, so that we reduce the `num of rows` to one, and the performance is improved.

Query plans provide internal execution information of DBMSs, and we aim to utilize query plans to tackle the challenges `C.1` and `C.2` for efficiently testing complex program logic. For example, in Figure 1.1, observing the leaves are `FullScan` or `IndexScan`, we know whether the program logic of handling indexes is executed. This information provides feedback for generating diverse test cases and facilitates the identification of logic bugs and performance issues.

## 1.4 Research Overview

I propose three methods and present one study in this thesis to realize the thesis statement. Figure 1.2 shows an overview of my research. DBMSs parse and optimize SQL queries into query plans and execute them to return the result. To tackle **C.1**, we propose to use query plans to 1) identify performance issues and 2) identify logic bugs. To tackle **C.2**, we propose to use query plans to 3) generate diverse test cases.

Observing that query plans are represented in DBMS-specific manners, to facilitate the implementation of these testing methods, we studied query plan representations and designed 4) a unified query plan representation.

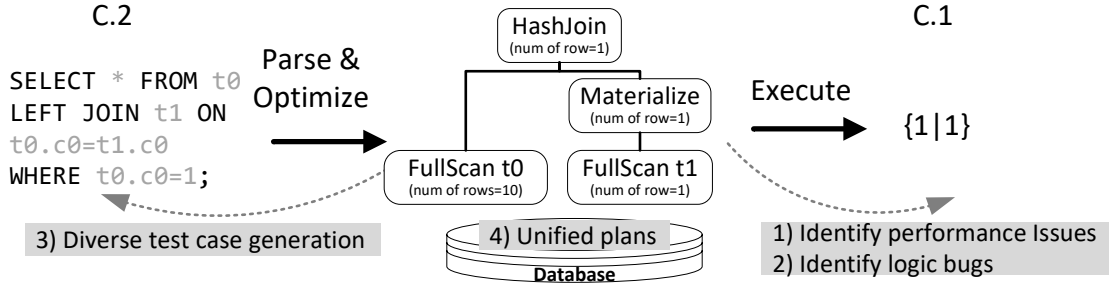


Figure 1.2: Research overview.

The first problem to solve is finding performance issues and belongs to **C.1**. I propose the method called *Cardinality Estimation Restriction Testing* (*CERT*) [86] for identifying performance issues by inspecting estimated cardinalities in query plans, and this work shows that performance issues can be efficiently found by checking query plans without execution. DBMSs are prone to performance issues, where a DBMS produces an inefficient query plan that might lead to the slow execution of a query. Detecting performance issues is a longstanding problem and inherently challenging, because no ground truth information on an expected execution exists. Estimated cardinalities, which represent the estimated number of rows returned by operators in query plans, are shown to be the most important component of query optimization [100]. Therefore, my idea to find performance issues is that finding and fixing issues in cardinality estimation may result in the highest performance gains. I designed a method to check whether a query has a smaller number of estimated rows than a more restrictive query. For example, given a query with **`LEFT JOIN`**, we replace **`LEFT JOIN`** with an **`INNER JOIN`** to construct the restricted query, and the estimated number of rows of the restricted query should be no more than the estimated number of rows of the unrestricted query. Otherwise, there is a potential issue. *CERT* eschews executing queries, which is costly and prone to performance fluctuations. I evaluated *CERT* on three widely used and mature DBMSs: *MySQL*, *TiDB*, and *CockroachDB*. *CERT* found 13 unique, previously unknown issues that have been confirmed or fixed by the developers with no false alarms. *CERT* validates 386×

more queries than *AMOEBA* in the same time period. I believe that these results demonstrate that *CERT* might become a standard technique in DBMS developers' toolbox, due to its efficiency and effectiveness.

The second problem to solve is finding logic bugs and belongs to **C.1**. Recently, researchers have proposed a testing approach called *Transformed Query Synthesis* (*TQS*) specifically designed to find logic bugs in join optimizations, which are a core component in DBMSs. *TQS* is a sophisticated approach that splits a given input table into several sub-tables and validates the results of the queries that join these sub-tables by retrieving the given table. We studied *TQS*'s bug reports, and found that 14 of 15 unique bugs were reported by showing discrepancies in executing the same query with different query plans. Therefore, I propose the method called *Differential Query Plans* (*DQP*) [87] as a simple alternative approach to *TQS*. *DQP* enforces different query plans for the same query and validates that the results are consistent. *DQP* can reproduce 14 of the 15 unique bugs found by *TQS*, and found 26 previously unknown and unique bugs. These results demonstrate that a simple approach with limited novelty can be as effective as a complex, conceptually appealing approach. We hope that the practicality of our approach—I implemented in less than 100 lines of code per system—will lead to its wide adoption.

The third problem to solve is generating diverse test cases and belongs to **C.2**. I propose the concept called *Query Plan Guidance* (*QPG*) [85] for guiding automated testing towards diverse query plans for diverse test cases aiming to expose more bugs, and *QPG* shows a significant advantage over code coverage guidance and random generation methods. I tackled this challenge based on the intuition that by steering testing towards exploring diverse query plans, we also explore more interesting behaviors—some of which are potentially incorrect. Technically, I designed a mutation strategy that gradually applies promising mutations to the database state, causing the DBMS to create different query plans for subsequent queries. I applied *QPG* to three mature, widely-used, and extensively tested DBMSs—*SQLite*, *TiDB*, and *CockroachDB*—and found 53 unique, previously unknown bugs that have been confirmed or fixed by developers. Considering the effectiveness of the mutation strategy, this method exercises 4.85–408.48× more unique query plans than a naive random generation method and 7.46× more than a code coverage guidance method.

The last problem to solve is efficiently utilizing query plans facilitating the above methods for tackling **C.1** and **C.2**. For the above three testing methods, *CERT*, *DQP*, and *QPG*, I observed that query plans are specific to DBMSs, as they reflect their internal execution steps, which differ widely across DBMSs. The plethora of different representations makes it challenging to efficiently build applications on query plans including the testing methods in this thesis. For example, a predicate in the `WHERE` clause of SQL corresponds to a concrete step to filter data in the query plans of *TiDB* [23], but corresponds to a property of another step to scan tables in the query plans of *PostgreSQL* [186]. In this way, *QPG* has to implement DBMS-specific logic to parse the predicates. I conducted an exploratory case study to investigate query plan representations in nine widely-used DBMSs. Our study shows that DBMS-specific query plan representations consist of three conceptual components: operations, properties, and formats. Based on the study, I propose a unified query plan representation and show its utility in three applications, namely visualization, testing, and benchmarking. The results show that existing testing methods can be efficiently adopted to multiple DBMSs, and we found *17* unique and previously unknown bugs in previously untested DBMSs. Additionally, existing visualization tools can support multiple DBMSs based on the unified query plan representation with moderate implementation effort, and comparing unified query plans across DBMSs provides actionable insights to improve their performance. We expect that the unified query plan representation will enable the exploration of additional application scenarios, such as test suite evaluation, debugging deployed DBMSs, and optimization verification.

## 1.5 Contributions

My contributions impact the current state of finding bugs in DBMSs. I summarize the contributions in three levels as follows.

### 1.5.1 Conceptual Contributions

Conceptually, I propose using query plans in automated testing to efficiently and effectively test complex program space. Query plans were initially exposed to users for performance tuning and provided a summary of internal execution information.

I demonstrated that using query plans can efficiently and effectively test complex program space, so more bugs can be found including logic bugs and performance issues. I expect that the research community will take the thesis forward, to further understand and utilize the query plans in testing.

### 1.5.2 Technical Contributions

To underpin the conceptual contribution, I make the following technical contributions:

- A novel technique to identify performance issues by inspecting query plans. Query plans include cardinality estimations, which have been demonstrated to be the most important part of query optimization. I designed a metamorphic relation to find performance issues by inspecting the inconsistency of cardinality estimations. The method found 13 unique and previously unknown bugs.

- A simple method to find logic bugs. A state-of-the-art test oracle *TQS* finds logic bugs by a sophisticated method that derives expected results by splitting and manually joining tables. However, we found that even a simple and straightforward test oracle—different query plans of the same query should return the same result—can be as efficient as *TQS*. We designed a method to realize this simple test oracle and the results show that this test oracle found 26 unique and previously unknown bugs.

- A novel technique to guide the test case generation process with the guidance of query plans. I observed that the query plans of the queries in previously found bugs are compact and simple, so I designed a concrete testing approach that mutates databases aiming to cover more diverse query plans. The results showed that this method found 53 unique and previously unknown bugs in a black-box manner.

- A comprehensive empirical study of query plan representations. I studied the query plan representations of nine popular DBMSs, and proposed a unified query plan representation to reduce the work of building applications on query plans. By integrating the unified query plan representation, the above testing

methods can be directly applied to multiple targets, and found *17* unique and previously unknown bugs.

### 1.5.3 Practical Contributions

I made the source code and experimental data publicly available allowing the research community to better reproduce and advance existing research work.

- *CERT* was integrated into *SQLancer*[2] and is public.
- *QPG* was integrated into *SQLancer*[3] and is public.
- *DQP* was integrated into *SQLancer*[4] and is public.
- *UPlan* has been publicly available.[5]

The integration amplifies the practical impact of my research in this thesis. More than 20 bugs have been found after publishing our papers. *CERT* attracted an open-source contributor to implement this method for more DBMSs.[6] Industry also showed interest in these methods. *TiDB* invited me to give a talk and considered supporting both *QPG* and *CERT* in their internal testing workflow. *SQLite* and *CockroachDB* sent emails to us requesting more technical details of *QPG*.

## 1.6 Research Scope

In this thesis, I restrict the research scope to the testing technique for finding bugs in DBMSs, especially for logic bugs and performance issues. The work in this thesis does not aim to improve other techniques, such as verification and benchmarking. *CERT*, *DQP*, and *QPG* are complementary to existing bug-finding methods, and *UPlan* can facilitate various applications on query plans. In this thesis, I do not consider the other reliability problems related to DBMSs, such as the reliability problem of operating systems that DBMSs are running on.

[2] https://github.com/sqlancer/sqlancer/issues/822
[3] https://github.com/sqlancer/sqlancer/issues/641
[4] https://github.com/sqlancer/sqlancer/issues/918
[5] https://unifiedqueryplan.github.io/
[6] https://github.com/sqlancer/sqlancer/issues/895

## 1.7 Outline

The remainder of this thesis is organized as follows. In Chapter 2, I provide the necessary background knowledge for this thesis including DBMSs and query plans. Next, in Chapter 3, I present the approach *CERT* to solve the problem of test oracle for finding performance issues. Then, in Chapter 4, I present the approach *DQP* as a simple test oracle for finding logic bugs. After that, in Chapter 5, I introduce the concept *QPG* to solve the problem of test case generation. Last, in Chapter 6, I present our empirical study of query plan representations and propose our design of a unified query plan representation. In Chapter 7, I summarize the related work. Lastly, in Chapter 8, I conclude this thesis as well as discuss potential future research directions.

# Chapter 2

# Background

In this chapter, I introduce the necessary background knowledge about DBMSs and related techniques for finding bugs in DBMSs.

## 2.1 Database Management Systems

DBMSs serve as an interface between the database and its users or programs, helping users to store, manipulate, and query data in a convenient and efficient way. Management of data involves both defining data models for the storage of data and providing mechanisms for the manipulation of data. Data models are used to organize data into entities that denote objects in the real world including several attributes to illustrate characteristics of entities. This thesis involves four popular data models: 1) the traditional relational model [42], which organizes data into tables, in which rows are entities and columns are attributes, 2) the graph data model [6], which organizes data into graphs, in which nodes are entities, nodes' properties are attributes, and edges are relations of entities, 3) the time series model [121], which organizes data as a series of data points ordered over time, in which data points are entities and fields in data points are attributes, and 4) the document model [206], which organizes data into semi-structured and unstructured schemas, such as JavaScript Object Notation (JSON) and Extensible Markup Language (XML), in which documents are entities and documents' fields are attributes. To manipulate and query data, most DBMSs adopt declarative languages, which express only the logic of a computation without specifying its specific execution. This thesis involves Structured Query Language (SQL) [27] for the relational and the time series data models, Cypher Query Language (CQL) [49] for the graph data model, and

MongoDB Query Language (MQL) [8] for the document model. *CERT*, *DQP*, and *QPG* mainly consider the relational data model and SQL, which have been adopted by most modern DBMSs, while *UPlan* considers all the above data models and declarative languages for a comprehensive study.

### 2.1.1 Structured Query Language

SQL is the most commonly used language for manipulating the data in DBMSs [27], and has been standardized by ISO/IEC 9075. SQL is widely supported by DBMSs; for example, according to a popular ranking,[1] the 10 most popular DBMSs support it. SQL consists of many types of statements[179], which can be classified into three main sub-languages:

1. Data Query Language (DQL), which provides a **`SELECT`** statement to query data.

2. Data Definition Language (DDL), which is used to create and modify the schemas of data objects, for example, **`CREATE`**, **`DROP`**, and **`ALERT`**.

3. Data Manipulation Language (DML), which is used to modify the contents of data objects, for example, **`INSERT`** and **`UPDATE`**.

While DDL and DML statements can affect the database, queries (*i.e.*, DQL statements) typically cannot. For *CERT*, *DQP*, and *QPG*, our test cases consist of DQL, DDL, and DML statements.

Code 2.1 shows the EBNF representation [133], a metasyntax notation to express context-free grammars, of a SQL query, whose features I considered in *CERT*, *DQP*, and *QPG*. A query starts with the **`SELECT`** keyword. It can optionally be succeeded by a **`DISTINCT`** clause that specifies that only unique records should be returned. A **`JOIN`** clause joins two tables or views; various joins exist that differ on whether and what rows should be joined when the join predicate evaluates to false. A query can contain a single **`WHERE`** clause; only rows for which its predicate evaluates to true are included in the result set. Similar to **`DISTINCT`**, the **`GROUP BY`** clause groups rows that have the same values into a single row. It can be followed by a **`HAVING`** clause that excludes records after grouping them. The **`LIMIT`** clause is used to restrict

[1] `https://db-engines.com/en/ranking` as of March 2023.

Code 2.1: The EBNF representation of a query.

```
1  SELECT [DISTINCT]
2    select_expression (, select_expression)*
3    FROM table_reference (INNER | LEFT | RIGHT | FULL | CROSS JOIN
         table_reference)*
4    (WHERE predicate)+
5    (GROUP BY predicate
6    (HAVING predicate)+)+
7    (LIMIT row_count)+ ;
```

the number of records that are fetched. More advanced features, such as window functions, common table expressions (CTEs), and subqueries can be used. While I did not consider them in this thesis, I believe that the proposed approaches could be extended to support these advanced features.

### 2.1.2 Query Plans

A query plan is a tree of operations that describes how a query is executed by a specific DBMS, and we leverage query plans in this thesis to find bugs in DBMSs. DBMSs literature distinguish between logical and physical query plans [160], the latter of which is typically exposed by DBMSs to users for understanding and optimizing performance-critical queries (*e.g.*, by providing hints to the DBMS). Although not specified by the standard, most mature relational DBMSs, including the 10 most popular relational DBMSs according to the DB-Engines ranking,[2] allow users to obtain a textual representation of a physical query plan by prefixing a query with **`EXPLAIN`**. For a better debugging experience, exposed physical query plans may include additional information, such as the estimated cost or predicate expressions (*e.g.*, used in **`WHERE`** clauses). While the logical query plan closely corresponds to the original declarative query, the physical query plan maps every logical operator to a so-called physical one that can be executed by the DBMS. For example, to translate a read operation on a table, the DBMS might choose one of potentially multiple so-called physical access methods (*e.g.*, a full table scan, or a partial scan with index). Similarly, to join two tables, the DBMS might decide between multiple join algorithms (*e.g.*, hash join or nested loop join) [160]. The query plan is typically short for the physical query plan, so, without special note, the query plan refers to the physical query plan in this thesis.

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

### 2.1.3 Query Optimization

DBMSs include query optimizers that, after parsing the SQL query, determine an efficient query plan. Determining an efficient query plan is challenging, since many factors might influence the plan's performance. One of the most commonly used models is the cost-based [29]—the query plan with the lowest estimated performance cost is chosen. The classical cost-based model includes three major components: cardinality estimation, cost model, and plan space enumeration. To obtain an efficient query plan, the component of plan space enumeration enumerates some subset of valid query plans. The component of cardinality estimation evaluates the estimated number of cardinality for each query plan. Using the estimated cardinality as principle input, the component of the cost model chooses the query plan with the lowest estimated cost.

### 2.1.4 Cardinality Estimation

Cardinality estimation was found to be the most important factor that affects the quality of query optimization [100], and I used the estimated cardinalities for finding performance issues in this thesis. Cardinality estimators typically obtain data statistics of the tables to be queried by sampling [75], through histograms [158], or machine learning algorithms [41, 199, 94]. Then, they enumerate all sub-plan queries, which are queries that process only a subset of tables in a query, and estimate how many rows they fetch. For example, for a query $A \bowtie B \bowtie C$ ($\bowtie$ denotes a join), cardinality estimators could estimate the cardinalities of $A, B, C$ respectively, and then estimate the cardinalities of $A \bowtie B$, $B \bowtie C$, $A \bowtie C$, and $A \bowtie B \bowtie C$. Lastly, the estimated cardinalities of these sub-plan queries help to decide the join order—whether $A \bowtie B$ or $B \bowtie C$ should be executed first.

## 2.2 Bug-finding Techniques

Both industry and academia have been tackling the problem of DBMSs' reliability for many years. In the following, I describe the most important types of techniques and research trends highlighting their advantages and drawbacks. More related works about these techniques are discussed in Section 7.1.

### 2.2.1 Testing

Testing is a cost-efficient and practical technique for experimentally finding bugs. Suppose we have a program $P$ that accepts an input $I$ and returns an output $O$ by executing the input $P(I) => O$, a plethora of testing techniques exist to look for the inputs $\{I_x, ..., I_y\}$ whose outputs $\{O_x, ..., O_y\}$ are unexpected. An unexpected output indicates a bug. We discuss three common testing methods for finding bugs in DBMSs: differential testing, metamorphic testing, and fuzzing.

Differential testing [114] is a method to identify bugs by comparing the outputs or behavior of multiple systems for the same input. For the systems $P$ and $P'$, this method evaluates whether $P(I) == P'(I)$. Slutz *et al.* [161] applied differential testing in a system called *RAGS*, which compares the outputs of executing the same input on different DBMSs to find bugs. Jung *et al.* proposed *APOLLO* [89], which compares the execution time of a query on two versions of a DBMS to find unexpected performance degradation. Both differential testing methods are effective as they successfully found hundreds of bugs without accessing the source code of DBMSs. However, the differential testing method requires that the implementations of different DBMSs or different versions of the same DBMS have the same semantics and performance for the same inputs. Most DBMSs typically have their own language dialects, which restrict the efficiency of the differential testing method.

Metamorphic testing [31, 32] is a method to generate both inputs and validate outputs for finding bugs that are due to incorrect outputs. This method constructs an input $I$ to a system and its output $O$ to derive a new input $I'$ (and output $O'$) and checks whether a so-called *Metamorphic Relation* holds between $O$ and $O'$. *SQLancer*[3] implemented two metamorphic relations *NoREC* [149] and *TLP* [150], both of which construct pairs of semantic-related queries and check whether their outputs comply with the expected metamorphic relations. By constructing another input and using its output as a reference, metamorphic testing can efficiently find incorrect outputs. However, due to the constraints of metamorphic relations, it is challenging to automatically construct diverse inputs to explore various components and complex program space of the target system.

Fuzzing [116] is a method that generates or mutates inputs to target programs for

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

finding memory-related bugs, such as buffer overflow[4] and double free.[5] *AFL* [180] and *LibFuzzer* [181] are the two most influential fuzzers, which randomly mutate a given input $I$ aiming to maximize code coverage. Fuzzing methods found a huge number of memory-related bugs in DBMSs, such as more than 100 bugs in the DBMS *SQLite*.[6] Fuzzing relies on general input generators, so it can be easily applied to multiple systems. However, general input generators by random mutation incur many invalid inputs due to syntax and semantic constraints. Therefore fuzzing is struggling to test complex program space, which requires syntax and semantic correct test cases, and cannot find non-crash bugs, such as incorrect outputs.

### 2.2.2 Verification

Formal verification is another powerful technique to verify whether the system behaves correctly with respect to a formal specification so as to prove the absence of violations of properties, such as bugs. Given a program $P$, we need to abstract its behaviors into $P'$, and a formal verification technique proves whether $P'$ satisfies a given specification $S$. If so, $P'$ works correctly with respect to $S$. Various methods of formal verification have been used for verifying DBMSs. Model checking [36] is a method for a systematically exhaustive exploration of the mathematical finite model, and Diana *et al.* [39] verified part of the SQL specifications by this method. The exhaustion process verifies all possible states to guarantee the correctness of the target system, but suffers from the state explosion problem that the number of possible states exceeds the calculation capability of modern computers. To apply this method to a real-world system, users have to abstract the system into a finite model, which is a non-trivial task, such as how to abstract the system for verifying performance guarantees. Malecha *et al.* [108] implemented a verified relational DBMS, whose specifications were written and verified in Coq,[7] an interactive theorem prover for formal proofs. However, as highlighted by the authors, it is still challenging to comprehensively abstract the states and prove complicated data structures, such as pointers. Formal verification can give a theoretical guarantee for verifying small-scale

[4] https://owasp.org/www-community/vulnerabilities/Buffer_Overflow
[5] https://owasp.org/www-community/vulnerabilities/Doubly_freeing_memory
[6] https://bugs.chromium.org/p/oss-fuzz/issues/list?q=sqlite3&can=1
[7] https://coq.inria.fr/

abstracted models, but is struggling with handling a whole complex program space in DBMS.

### 2.2.3 Test Suites and Benchmarking

The most straightforward and conventional methods to find bugs are through manually constructing test suites and benchmarking. Suppose we have a set of programs $\{P_0, ..., P_m\}$, we execute an input $I$ on each program $P_i$ and evaluate its property against a predefined baseline $B_i$. Most DBMSs maintain their own test suites. These test suites are typically high-quality but small because they are manually crafted. Benchmarking is used for performance regression testing, because performance is one of the most important metrics for DBMSs. *TPC-H* [123], *TPC-DS* [177], *JOB* [100] are widely used benchmarks for DBMSs. However, constructing a benchmark requires much manual effort, and it is inefficient to evaluate DBMSs in a limited number of scenarios, which may not be complex enough.

# Chapter 3

# Cardinality Estimation Restriction Testing

To identify performance issues, in this chapter, we propose *Cardinality Estimation Restriction Testing* (*CERT*), a novel technique that finds performance issues through the lens of cardinality estimation. *CERT* has been published in the 46th International Conference on Software Engineering (ICSE'24) [86].

## 3.1 Introduction

Performance is one of the most important metrics for DBMSs, especially in today's big data era. From a software standpoint, the mainstream direction to optimize DBMSs' performance is to improve query optimization, that is, translating queries specified in the Structured Query Language (SQL) to an efficient query plan that includes concrete steps to execute a query. An efficient query plan is expected to have the best performance, by estimating its attributes, such as cardinality, CPU, memory, and IO [100]. To balance the trade-off between spending little time on optimization, which is performed at run time, and finding an efficient query plan, researchers and practitioners have invested decades of effort into query optimization, covering directions such as search space exploration for join ordering [127, 47, 46], index data structures [61], execution time prediction [4, 197], or parallel execution on multi-core CPUs [55] and GPUs [134].

Finding performance issues in DBMSs—also referred to as optimization opportunities or performance bugs—is challenging. Given a query $Q$ and a database $D$, we want to determine whether executing $Q$ on $D$ results in unexpectedly suboptimal per-

formance. In general, no ground truth is available that specifies whether $Q$ executes within a reasonable time. To exacerbate this issue, DBMSs use various heuristics and cost models during optimizations, or make trade-offs in optimizing specific kinds of queries over others. Second, the execution time of $Q$ might be significant if $D$ is large, making it time-consuming to measure $Q$'s actual performance. Given that the execution time depends on various factors of the execution environment [122] (*e.g.*, the state of caches), it might even be necessary to execute $Q$ multiple times to obtain a reasonably reliable measure of its execution time. Cloud environments are in particular prone to noise [97]; a report on testing SAP HANA [9] has recently stressed that performance testing for cloud offerings of DBMSs—such as SAP HANA Cloud, which runs in Kubernetes pods—is one of the main challenges in testing DBMSs due to inherently noisy environments.

Benchmark suites such as TPC-DS [177] or TPC-H [123] are widely used in practice to monitor DBMSs' performance over versions through predetermined performance baselines, which could be specified [147, 202, 205]. However, deriving an appropriate baseline is challenging and might result in false alarms. Automated testing techniques have been proposed to find performance issues without the need of curating a benchmark suite. *APOLLO* [89] generates databases and queries automatically and validates whether executing the query on different versions of the DBMS results in significantly different execution times. However, *APOLLO* can only find regression bugs. *AMOEBA* [107] finds performance issues by examining discrepancies in the execution time of a pair of semantically equivalent queries. However, semantically equivalent queries do not necessarily exhibit a similar performance as the issues found by *AMOEBA* have a high false positive rate—only 6 of 39 issues were confirmed by the developers, 5 of which were fixed [107]. For the above methods, queries need to be executed on sufficiently large databases to detect significant performance discrepancies.

In this work, we propose *Cardinality Estimation Restriction Testing* (*CERT*), a general technique that finds performance issues by testing the DBMSs' cardinality estimation. *Cardinality estimation* is the process in which *cardinality estimator* computes *estimated cardinalities*, the estimated numbers of rows that will be returned. Since estimated cardinalities are approximate, it is infeasible to check for a specific number. Rather, the core idea of our approach is that making a given query more

restrictive should cause the cardinality estimator to estimate that the more restrictive query should fetch at most as many rows as the original query. More formally, given a query $Q$ and a database $D$, $Card(Q, D)$ denotes the actual cardinality, that is, the exact number of records to be fetched by $Q$ on $D$. If we derive a more restrictive query $Q'$ from $Q$, $Card(Q', D) \leq Card(Q, D)$ always holds. $EstCard(Q, D)$ denotes the estimated cardinality for $Q$, and we expect $EstCard(Q', D) \leq EstCard(Q, D)$ to also hold for any DBMS. We refer to this property as *cardinality restriction monotonicity*. Any violation of this property indicates a potential performance issue.

*CERT* addresses the aforementioned challenges. Cardinality estimation accuracy was shown to be the single most important component for deriving an efficient query plan [100]. Therefore, we believe that pinpointing issues in cardinality estimation would help developers focus on the most relevant issues, addressing which might result in significant performance gains. Additionally, this idea is applicable to finding a broader range of performance issues. For example, we found that other kinds of query optimization issues can be exposed by unexpected estimated cardinalities, as shown in Code 3.2. Furthermore, estimated cardinalities can be readily obtained by DBMSs without executing $Q$; DBMSs typically provide a SQL **`EXPLAIN`** statement that provides this information as part of a query plan, allowing our technique to achieve high throughput. In addition, since our method does not measure run-time performance, it can be used in noisy environments, and minimal test cases that demonstrate the performance issue can be automatically obtained [214]. Finally, *CERT* is a black-box technique that can be applied even without access to the source code.

Code 3.1 shows a running example demonstrating *CERT*. We randomly generate SQL statements as shown in lines 1–4 to create a database state and ensure that each table's data statistics are up to date in lines 5–6. Then, we randomly generate a query with a **`LEFT JOIN`** and derive a more restrictive query by replacing the **`LEFT JOIN`** with an **`INNER JOIN`** as shown in lines 8–9. The second query is more restrictive than the first query as **`INNER JOIN`** should always fetch no more rows than **`LEFT JOIN`**. We examined their estimated cardinalities in query plans, obtained by using an **`EXPLAIN`** statement. Despite having made the query more restrictive, the cardinality estimator estimates that the first query fetches 20 rows, while the second one fetches 60 rows, which is unexpected. The root cause was an incorrect double-counting

Code 3.1: This running example demonstrates a performance issue found by *CERT* in *CockroachDB*.

```
1  CREATE TABLE t0 (c0 INT);
2  CREATE TABLE t1 (c0 INT);
3  INSERT INTO t0 VALUES (1), (2), (3), (4), (5), (6), (7), (8), (9), (10), (11),
       (12), (13);
4  INSERT INTO t1 VALUES (21),(22),(23),(24),(25);
5  ANALYZE t0;
6  ANALYZE t1;
7
8  EXPLAIN SELECT * FROM t0 LEFT JOIN t1 ON t0.c0<1 OR t0.c0>1; -- estimated
       rows: 20 🐞
9  EXPLAIN SELECT * FROM t0 INNER JOIN t1 ON t0.c0<1 OR t0.c0>1; -- estimated
       rows: 60 ✔
10 ------------------------------------------------------------------------------
11 • cross join(left outer)                • cross join
12 | estimated row:20                      | estimated row:60
13 | pred:(c0<1)OR(c0>1)                   |-• filter
14 |-• scan                                |  | estimated row:12
15 |    estimated row:13                   |  | filter:(c0<1)OR(c0>1)
16 |    table: t0@t0_pkey                  |  |-• scan
17 |-• scan                                |       estimated row:13
18      estimated row:5                    |       table: t0@t0_pkey
19      table: t1@t1_pkey                  |-• scan
20                                              estimated row:5
21                                              table: t1@t1_pkey
```

when estimating the selectivity of **OR** expression in the **ON** condition of the **INNER JOIN**. The estimated cardinality of the first query with **LEFT JOIN** should be no less than 60; after the developers fixed this issue, the estimated cardinality is changed to 60. This fix improved the performance of the query `SELECT * FROM t0 LEFT OUTER JOIN t1 ON t0.c0<1 OR t0.c0>1 FULL JOIN t2 ON t0.c0=t2.c0` by 20% as shown in Code 3.6. The improvement was due to a more accurate estimated cardinality, which enabled a better selection of the join order. Note that we avoided executing the query; *CERT* only examines query plans.

We implemented *CERT* in *SQLancer*, a popular DBMS testing tool, and evaluated it on three widely used and mature DBMSs, *MySQL*, *TiDB*, and *CockroachDB*. While *MySQL* is one of the most popular open-source DBMSs, *TiDB* and *CockroachDB* are developed by companies. We reported 14 performance issues to the developers, who confirmed that 13 of them were unique and 12 were unknown. Of these unique issues, 2 issues were fixed, 9 other issues were confirmed, and 2 issues required further investigation. Similar to existing work, *CERT* might report false alarms, since implementations might not strictly adhere to the *cardinality*

*restriction monotonicity*. However, in practice, none of the issues that we reported were considered false alarms. Our evaluation demonstrates the high throughput achieved by eschewing executing queries; our implementation can validate 386× more queries than *AMOEBA* in the same time period. We believe that these results demonstrate that *CERT* might become a standard technique in DBMS developers' toolbox, due to its efficiency and effectiveness, and hope that it will inspire future work on finding performance issues in DBMSs.

Overall, we make the following contributions:

- We present a motivational study to investigate the causes of previous performance issues.
- We propose a novel technique, *CERT*, to test cardinality estimation for finding performance issues in query optimization without measuring execution time. We show a concrete realization of the technique by proposing 12 query-restriction rules.
- We implemented *CERT* in *SQLancer* and evaluated it on multiple aspects. *CERT* found 13 unique issues of cardinality estimation in widely-used DBMSs, and 11 issues were confirmed or fixed. The source code of *CERT* is publicly available, and has been integrated into *SQLancer*.

## 3.2 Performance Issue Study

As a motivating study, to investigate if performance issues are caused by incorrect cardinality estimation in practice, we examined previous performance issues related to query optimization.

**Subjects.** We studied the issues reported for *MySQL*, *TiDB*, and *CockroachDB*. *MySQL* is the most popular relational DBMS according to a survey in 2021.[1] *TiDB* and *CockroachDB* are popular enterprise-class DBMSs, and their open versions on GitHub are highly popular as they have been starred more than 33k and 26k times.

[1] `https://insights.stackoverflow.com/survey/2021#most-popular-technologies-database`

Table 3.1: Previous performance issues.

| DBMS | #ID | Caused by Cardinality Estimation |
|---|---|---|
| *MySQL* | 61631 | ✓ |
| *MySQL* | 56714 | ✓ |
| *MySQL* | 25130 | X |
| *CockroachDB* | 93410 | X |
| *CockroachDB* | 71790 | X |
| *TiDB* | 9067 | ✓ |
| **Sum** | 6 | 3 |

They are widely used and have thus been studied in other DBMS testing works [106, 151, 150].

**Methodology.** We searched for performance issues using the keywords "*slow*" or "*suboptimal*" in the above-stated DBMSs, aiming to obtain issues that relate to either slow execution or suboptimal query plans. For *MySQL*, we chose issues whose status was *closed*, the severity was *(S5) performance*, and the type was *MySQL Server: Optimizer* in the bug tracker.[2] Considering *MySQL* was first released in 1995 and some issues are too old to be reproduced, we investigated the issues in version 5.5 or later. For *TiDB*, we searched its repository[3] by the filter *is:issue is:closed linked:pr label:type/bug slow in:title*. For *CockroachDB*, we searched its repository[4] by the filter *is:issue is:closed linked:pr label:C-bug slow in:title*. Then, we manually analyzed and reproduced each issue to identify whether it was a performance issue related to query optimization and caused by cardinality estimation. Specifically, if the estimated cardinality of the query in a report was changed by the fix, we deemed the performance issue to be caused by incorrect cardinality estimation.

**Analysis.** Table 3.1 shows the studied performance issues. Overall, we identified six performance issues in three DBMSs, and three of them were caused by incorrect cardinality estimation. We attribute the lower number of performance issues to the difficulty in identifying and resolving performance issues in query optimization. For issues #61631 and #56714, they produce inefficient query plans which have higher

[2] `https://bugs.mysql.com/`
[3] `https://github.com/pingcap/tidb/issues`
[4] `https://github.com/cockroachdb/cockroach/issues`

Code 3.2: The performance issue #56714 in *MySQL*.

```
1  CREATE TABLE test (a INT PRIMARY KEY AUTO_INCREMENT, b INT NOT NULL, INDEX
       (b)) engine=INNODB;
2  CREATE TABLE integers(i INT UNSIGNED NOT NULL);
3  INSERT INTO integers(i) VALUES (0), (1), (2), (3), (4), (5), (6), (7), (8),
       (9);
4  INSERT INTO test (b)
5  SELECT units.i MOD 2
6  FROM integers AS units
7      CROSS JOIN integers AS tens
8      CROSS JOIN integers AS hundreds
9      CROSS JOIN integers AS thousands
10     CROSS JOIN integers AS tenthousands
11     CROSS JOIN integers AS hundredthousands;
12
13 EXPLAIN SELECT MAX(a) FROM test WHERE b=0; -- estimated rows: {500360} 🐞, {1}
       ✔
```

estimated cardinalities than the optimal query plans. Although both issues are not directly due to the faults in cardinality estimators, they are still observable through estimated cardinalities, so we deemed both were caused by cardinality estimation. Issue #9067 was caused by the cardinality estimation due to an issue in calculating cardinality for correlated columns. The other three issues, which were not caused by cardinality estimation, were due to inefficient operations. For example, issue #71790 was due to the inefficient implementation of `MERGE JOIN` that does not use the smaller table as the right child, and the estimated cardinality was not changed after fixing the implementation of the operation.

**Case study.** Code 3.2 shows issue #56714 in *MySQL* as an illustrative example of a performance issue caused by cardinality estimation. According to the issue report, this performance issue incurs a slowdown of execution time from 0.01 seconds to 3.02 seconds. Column `b` in table `test` uses an index, but the query in line 13 does not correctly use the index incurring a `FULL TABLE SCAN`, which is slow. Although the root cause for this performance is in index selection, not in the cardinality estimator, the suboptimal index selection affects the estimated cardinality as the `FULL TABLE SCAN` is expected to scan more rows than the `INDEX SCAN`.

Performance issues can arise from inefficient operations, flawed cardinality estimators, and inefficient query plans. The latter two causes can be found by unexpected estimated cardinalities.

## 3.3 Approach

We propose *CERT*, a novel technique for testing cardinality estimation. The core idea is that a given query should not have a lower estimated cardinality than a more restrictive query derived from it. We term this property *cardinality restriction monotonicity* and expect that DBMSs adhere to it in practice. *CERT* is a simple black-box technique, making it widely applicable in practice.

**Method overview.** Figure 3.1 shows an overview of *CERT* based on the running example in Code 3.1. Given a randomly generated query at ①, we derive another more restrictive query at ② and retrieve both queries' query plans. Then, if both query plans are structurally similar at ③, we validate the *cardinality restriction monotonicity* property at ④; we expect the less restrictive query ① to return at least as high estimated cardinality as the more restrictive query ②. Any discrepancy is considered a performance issue. We perform the structural similarity checking in ③ based on the observation that a more restrictive query can result in a significantly different query plan, whose estimated cardinalities are not comparable. Next, we give a detailed explanation of each step.

### 3.3.1 Database and Query Generation

We require a database state and a query for testing. Both database state and query can be manually given or generated. Common generation-based methods include mutation-based methods [219, 106] and rule-based generation methods [182, 150, 151, 149]. How to generate database states and queries is not a contribution of this paper, and our approach can be paired with any database state and query generation method. For example, in Code 3.1, we could randomly generate a database state in lines 1–4 and a query in line 8.

Before executing queries on the generated database state, we execute **ANALYZE**

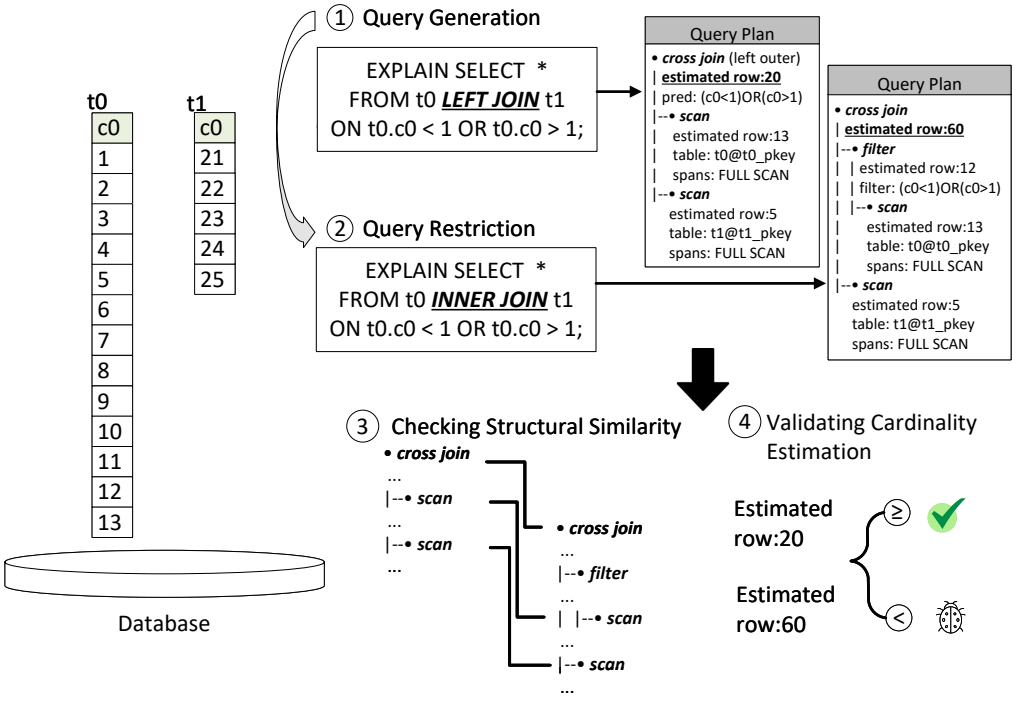


Figure 3.1: Overview of *CERT*.

statements on each table to guarantee that the data statistics are up to date. For Code 3.1, these statements are executed in lines 5–6.

### 3.3.2 Query Restriction

Given a query, we derive a more restrictive query based on two insights. First, for the clauses that we considered, adding a clause to a query makes the query more restrictive except for the `JOIN` clause. Second, given an already existing clause, we can modify the clause or its predicate to obtain a more restrictive query. Specifically, within a query, we randomly choose an SQL clause. Then, we restrict the chosen SQL clause followed by predefined rules to make sure the restricted query fetches no more rows than the original query according to the semantics of queries. For example, we derive the query in ② from the query in ① by replacing the `LEFT JOIN` to `INNER JOIN`. We considered the SQL clauses shown in Code 2.1 and proposed at least one rule for each feature, yielding the 12 rules shown in Table 3.2. Since

Table 3.2: The rules to restrict queries.

| | Clause | Source | Target |
|---|---|---|---|
| 1 | JOIN | LEFT JOIN | INNER JOIN |
| 2 | JOIN | RIGHT JOIN | INNER JOIN |
| 3 | JOIN | FULL JOIN | LEFT JOIN |
| 4 | JOIN | FULL JOIN | RIGHT JOIN |
| 5† | JOIN | CROSS JOIN | FULL JOIN |
| 6 | SELECT | ALL | DISTINCT |
| 7 | GROUP BY | <Empty> | <Predicate> |
| 8 | HAVING | <Empty> | <Predicate> |
| 9 | WHERE | <Empty> | <Predicate> |
| 10 | WHERE | <Predicate> | <Predicate> AND <Predicate> |
| 11 | WHERE | <Predicate> OR <Predicate> | <Predicate> |
| 12 | LIMIT | <Natural number> | <Natural number> - <Natural number> |

† Rule 5 holds when both tables are not empty.

the `JOIN` clause, which specifies two tables or views to be joined, is a major factor influencing the queries' run time [100], 5 of the 12 rules relate to them. Our rules are not exhaustive; we believe that practitioners could propose additional rules depending on their testing focus.

**Rule overview.** In Table 3.2, *CERT* derives a *Target* statement from a *Source* statement by applying a restriction on the shown *Clause*, as demonstrated through *Example*. <Predicate> refers to a boolean expression and <Natural number> to a natural number. These examples are based on a database state with two tables `t0` and `t1`, both of which have only one column `c0`. For each test to be generated, we randomly choose a SQL clause, of which one or more rules are randomly applied to restrict a query. In Figure 3.1, we choose the *JOIN* clause and apply only rule 1, which replaces a `LEFT JOIN` with a `INNER JOIN` in the `JOIN` clause of a query.

**JOIN clause.** Our key insight for testing the `JOIN` clause is the partial inequality relationship in terms of cardinalities between different kinds of joins. For a fixed join predicate, the following inequalities for the different joins' cardinalities hold: `INNER JOIN` $\leq$ `LEFT JOIN/RIGHT JOIN` $\leq$ `FULL JOIN` $\leq$ `CROSS JOIN`. Figure 3.2 illustrates this using a `JOIN` diagram [43] on two tables, each of which has three rows, with the same color denoting the rows that can be matched in the `JOIN` predicate. As determined by the SQL standard, `INNER JOIN` fetches rows that have matching values in both tables; `LEFT JOIN/RIGHT JOIN` fetch all rows from the left/right table and the

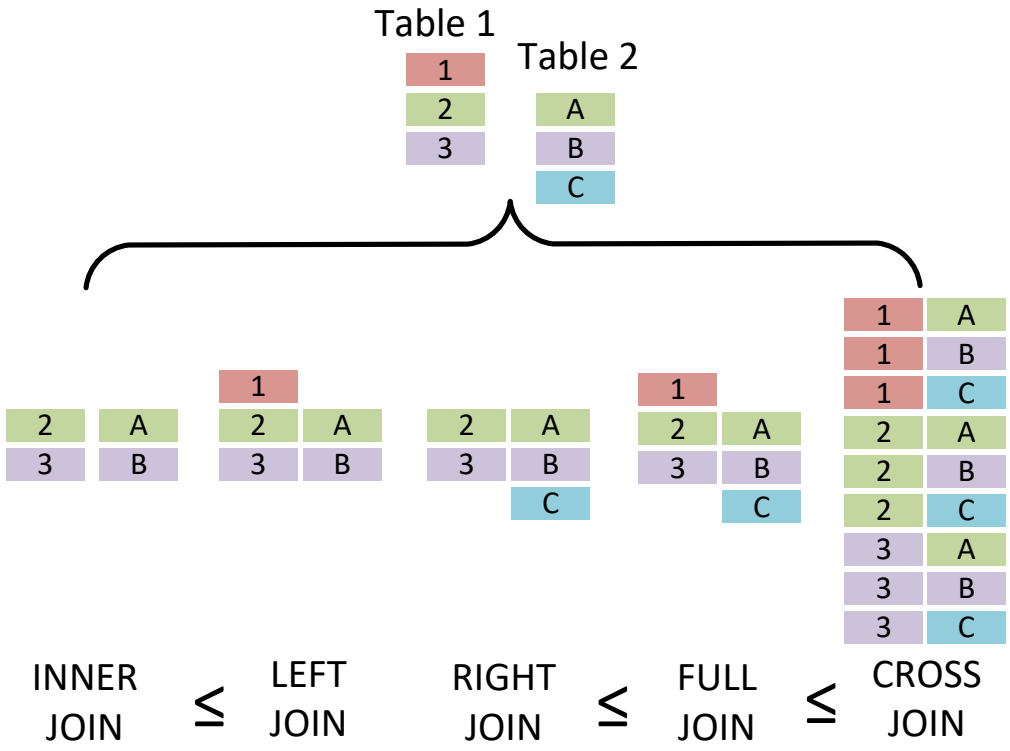


Figure 3.2: The inequality relationships of estimated cardinalities in the **JOIN** clause with an example to join two tables.

matching rows from the respectively other table; **FULL JOIN** fetches all rows from both tables; **CROSS JOIN** fetches all possible combinations of all rows from both tables without an **ON** clause. A corner case for rule 5 concerning the **CROSS JOIN** is that this join may fetch fewer rows than **FULL JOIN** if either of the tables is empty, in which case **CROSS JOIN** fetches zero rows. To avoid potential false alarms, we ensure that each table contains at least one row.

**WHERE clause.** For the **WHERE** clause, our insight is that we can restrict the predicate that is used for filtering rows. If the query contains an empty **WHERE** clause, we restrict the query by adding a random predicate. If the predicate is non-empty but has an **OR** operator, we restrict it by removing either of the **OR**'s operands. Otherwise, we add an **AND** operator with a randomly generated predicate. Restricting predicates would be applicable also to testing **JOIN** clauses; in this work, we aimed to introduce the general idea behind *CERT* and illustrate it on a small set of promising rules. We believe that practitioners who adopt the approach will propose many additional

Code 3.3: An example of query plans that are not structurally similar, which is why we exclude them for testing.

```
1  CREATE TABLE t0 (c0 INT);
2  CREATE TABLE t1 (c0 INT, c1 INT);
3  INSERT INTO t1 VALUES(1,2), (3,4), (5,6), (NULL, NULL);
4  INSERT INTO t0 VALUES(1), (2);
5  ANALYZE t0;
6  ANALYZE t1;
7
8  EXPLAIN SELECT * FROM t0 FULL JOIN t1 ON t1.c1 IN (t1.c1) WHERE CASE WHEN
       t1.rowid > 2 THEN false ELSE t1.c1=1 END; -- estimated rows: 2
9  EXPLAIN SELECT * FROM t0 RIGHT JOIN t1 ON t1.c1 IN (t1.c1) WHERE CASE WHEN
       t1.rowid > 2 THEN false ELSE t1.c1=1 END; -- estimated rows: 3
10 ------------------------------------------------------------------------------
11 • filter                              • cross join(right)
12 | estimated row:2                       | estimated row:3
13 |-• cross join(full)                    |-• scan (t0)
14    | estimated row:6                      | estimated row:2
15    |-• scan (t1)                          |-• filter
16    |    estimated row:4                     | estimated row:1
17    |-• scan (to)                            |-• scan (t1)
18         estimated row:2                         estimated row:4
```

rules.

**Other SQL clauses.** A query can be restricted by a `DISTINCT` clause, which should fetch no more rows than the same query without such a clause, or by replacing its `ALL` clause. Similarly, a query without `GROUP BY` or `HAVING` can be restricted by adding such clauses along with any predicate. A `LIMIT` clause can be added, or a lower limit can be replaced with a higher limit.

### 3.3.3 Checking for Structural Similarity

Even for similar queries, DBMSs may create significantly different query plans. In such cases, the estimated cardinalities might be calculated in different ways, and thus result in false alarms. Code 3.3 shows an example of this problem. The only difference between the two queries in lines 8–9 is that `FULL JOIN` is used in the first query and `RIGHT JOIN` is used in the second query. The estimated cardinalities of them are 2 and 3 respectively. Based on rule 4 alone, this discrepancy would constitute a performance issue; however, consider the query plans in Code 3.3. In lines 11–18, the left part is the query plan of the first query, and the right part is that of the second query. For the first query with `FULL JOIN`, the sequence of

operations is `filter, cross join, scan, scan` in which the operation `filter` is applied *after* the operation `cross join`. For the second query with **`RIGHT JOIN`**, the sequence of operations is `cross join, scan, filter, scan`, in which the operation `filter` is applied *before* the operation `cross join`. The difference is due to a SQL optimization mechanism called *predicate pushdown* [102], which moves a filter to be executed before joining two tables and is applied in the second query. The predicate pushdown does not affect the final result, but can reduce the estimated cardinalities to be joined in the operation **`RIGHT JOIN`**, which is more efficient. However, because the predicate is pushed down, the structure of the query plans changes. Therefore, the estimated cardinalities are calculated in a different manner than that of the first query, and the developers consider the estimated cardinalities of both query plans as incomparable.[5] In Code 3.3, the estimated cardinalities of both operations **`FULL JOIN`** and **`RIGHT JOIN`** are calculated as the sum of the estimated cardinalities in the last step and the operation `filter` is calculated as one-third of the estimated cardinalities in the last step. The estimated cardinality of the first query plan is calculated by $(2 + 4)/3 = 2$, while that of the second query plan is calculated by $4/3 + 2 = 3$.

We identify comparable estimated cardinalities by checking for *structural similarity*. In a query plan $P$, the operation of a node $N$ is denoted as $O_N$. Suppose $N$ has $k$ children, denoted as $C_1, C_2, \ldots, C_k$, the node's flattened operation sequence is an array obtained by concatenating the flattening of each child node: $\text{flatten}(N) = [O_N, \text{flatten}(C_1), \text{flatten}(C_2) \ldots \text{flatten}(C_k)]$. The flattening of the root node is also denoted as $\text{flatten}(P)$. For a pair of query plans $P_a$ and $P_b$, we define both are *structurally similar* only if $ED(\text{flatten}(P_a), \text{flatten}(P_b)) <= 1$, in which $ED$ represents the edit distance [124], a common way of quantifying how dissimilar two strings, of both operation sequences. For example, in the query plans of Figure 3.1, the sequences of operations are `[cross join, scan, scan]` and `[cross join, filter, scan, scan]`. The first sequence can be edited to the second sequence by inserting a `filter` only, and *vice versa*. Both query plans are structurally similar and we validate the *cardinality restriction monotonicity* property. If they are not structurally similar, we continue testing with a new query. The calculation is based on sequences rather than trees, because the computation of the edit distance of trees was shown to be

[5] `https://github.com/cockroachdb/cockroach/issues/89060`

NP-hard [170].

### 3.3.4 Validating Cardinality Estimation

Finally, we validate the *cardinality restriction monotonicity* property on the estimated cardinalities extracted from the query plans. If the estimated cardinality of the original query is lower than that of the more restrictive query, we report the query pair as an issue. In Figure 3.1, the estimated cardinality of the original query is 20, which is lower than that of the other restricted query, which is 60. This indicates an unexpected result, and we report both queries to developers. Recall that we deem the estimated cardinality in the root operation of the query plan as the estimated cardinality of the query and ignore the estimated cardinalities of other operations.

## 3.4 Evaluation

To evaluate the effectiveness and efficiency of *CERT* in finding performance issues through estimated cardinalities, we implemented *CERT* in *SQLancer*,[6] which is an automated testing tool for DBMSs, and, based on our prototype *SQLancer+CERT*, we sought to answer the following questions:

**Q.1 Effectiveness.** Can *CERT* identify previously unknown issues?

**Q.2 Historic Bugs.** Can *CERT* identify historic performance issues?

**Q.3 Efficiency.** How does *CERT* compare to the state-of-the-art approach in terms of accuracy and efficiency?

**Q.4 Sensitivity.** Which rules proposed in Table 3.2 contribute to finding issues? Do DBMSs adhere to the *cardinality restriction monotonicity* property in practice?

**Implementation.** We reused the implementation of a rule-based random generation method of *SQLancer* to generate queries and database states. For each table, we generated 100 `INSERT` statements and ensured that every table contains at least

[6] `https://github.com/sqlancer/sqlancer/releases/tag/v2.0.0`

one row. Then, the generated queries and database states are passed to *CERT* for validating the *cardinality restriction monotonicity* property. The core logic of *CERT* is implemented in only around 200 lines of Java code for each DBMS, suggesting that its low implementation effort might make the approach widely applicable. We used pattern matching to extract operations and estimated cardinalities, and implemented the *Dynamic Programming* (DP) algorithm [183] to calculate the structural similarity.

**Tested DBMSs.** We tested the same DBMSs, *MySQL*, *TiDB*, and *CockroachDB* as we studied in Section 3.2. For Q1, Q3, and Q4, we used the latest available development versions (*MySQL*: 8.0.31, *TiDB*: 6.4.0, *CockroachDB*: 22.2.0). For Q3, in an attempt of a fairer comparison to *AMOEBA*, we chose the historical version of *CockroachDB* 20.2.19, which is the version that *AMOEBA* [107] tested.

**Baselines.** To the best of our knowledge, no existing work can be applied to specifically test cardinality estimation. The most closely related work is *AMOEBA*, which finds performance issues in query optimizers. We did not consider *APOLLO* [89], because it finds only performance regressions. Ensuring a fair comparison with *AMOEBA* is challenging, as the approaches are not directly comparable. *AMOEBA* validates that semantic-equivalent queries exhibit similar performance characteristics, while *CERT* validates the *cardinality restriction monotonicity* property. Furthermore, both tools support a different set of DBMSs; *AMOEBA* supports *CockroachDB* and *PostgreSQL*, while *CERT* supports *CockroachDB*, *TiDB*, and *MySQL*. Thus, we performed the comparison in Q3 using only *CockroachDB*, which is supported by both tools.

**Experimental infrastructure.** We conducted all experiments on a desktop computer with an Intel(R) Core(TM) i7-9700 processor that has 8 physical cores clocked at 3.00GHz. Our test machine uses Ubuntu 20.04 with 8 GB of RAM, and a maximum utilization of 8 cores. We ran all experiments 10 runs for statistical significance.

Table 3.3: The unique issues found by *CERT*.

| DBMS | Bug ID | Version | Rules | Status |
|---|---|---|---|---|
| *MySQL* | 108833 | 8.0.31 | 9 | Verified |
| *MySQL* | 108851 | 8.0.31 | 9 | Verified |
| *MySQL* | 108852 | 8.0.31 | 6 | Verified |
| *TiDB* | 38319 | 51a6684f | 11 | Confirmed |
| *TiDB* | 38747 | 3ef8352a | 7 | Confirmed |
| *TiDB* | 38479 | 3ef8352a | 3 & 5 | Confirmed |
| *TiDB* | 38482 | 3ef8352a | 8 | Confirmed |
| *TiDB* | 38665 | 6c55faf0 | 2 | Confirmed |
| *TiDB* | 38721 | 6c55faf0 | 9 | Confirmed |
| *CockroachDB* | 88455 | 7cde315d | 1 | Fixed |
| *CockroachDB* | 89161 | f188d21d | 11 | Fixed (Known) |
| *CockroachDB* | 89462 | 81586f62 | 8 | Backlogged |
| *CockroachDB* | 90113 | fbfb71b9 | 2 | Backlogged |

## Q.1 Effectiveness

**Method.** We ran *SQLancer+CERT* to find performance issues. Each automatically generated issue report usually includes many SQL statements, making it challenging for developers to analyze the root reason for the issue. To alleviate this problem and better demonstrate the underlying reasons for these issues, we adopted delta debugging [214] to minimize test cases before reporting them to developers. The steps to minimize the test case are 1) incrementally removing some of the SQL statements in the test case and 2) ensuring that the *cardinality restriction monotonicity* property is still violated. After submitting the issue reports with minimized test cases to developers, we submitted follow-up issues only if we believed them to be unique, such as those identified by different rules than previous issues, to avoid duplicate issues. *MySQL* has its own issue-tracking system, and developers add a label *Verified* for the issues that they have confirmed. *TiDB* and *CockroachDB* use GitHub's issue tracker. *TiDB*'s developers assign labels, such as affected versions and modules; we considered the issue as *Confirmed* after such a label was assigned. *CockroachDB*'s developers typically directly replied whether they planned on fixing the issue which we consider *Fixed* or whether the issue was considered a false alarm. In some cases, they added a *Backlogged* label to indicate that they would investigate this issue in

Code 3.4: Rule 6, which identified this issue in *MySQL*, replaces ALL with DISTINCT.

```
1 CREATE TABLE t0(c0 INT, c1 INT UNIQUE) ;
2 INSERT INTO t0 VALUES(-1, NULL),(1, 2),(NULL, NULL),(3, 4);
3 ANALYZE TABLE t0 UPDATE HISTOGRAM ON c0, c1;
4
5 EXPLAIN SELECT ALL t0.c0 FROM t0 WHERE t0.c1; -- estimated rows: 3
6 EXPLAIN SELECT DISTINCT t0.c0 FROM t0 WHERE t0.c1; -- estimated rows: 4
```

the future. For all DBMSs, based on historic reports, we observed that, typically, developers directly reject duplicate issue reports.

**Results.** Table 3.3 shows the unique issues that *CERT* found in three tested DBMSs. The *Bug ID* column shows the bug id in respective bug trackers. The *Version* column shows the versions or git commits of the DBMSs in which we found corresponding issues. The *Rules* column shows which rules identified this issue. In total, we have found 13 unique performance issues, 9 issues have been confirmed by developers in three days, 2 issues have been fixed in one week, 2 issues were backlogged, and 1 issue was duplicated. No false alarm was generated. We speculate that many confirmed bugs remain unfixed, because 1) fixing performance issues requires comprehensive consideration which usually consumes much time, and 2) performance issues might have lower priority than other issues, such as correctness bugs, which cause a query to compute an incorrect result. Among all 13 unique issues, the only known issue that we found in *CockroachDB* had been backlogged for around 10 months since it was first found, and our test case clearly demonstrated the root reason for the issue, which allowed developers to quickly fix it. Apart from the reported issues, *CERT* continuously generates more than ten issue reports per minute. We did not report the additional bug-inducing test cases to the developers to avoid burdening them, because deciding their uniqueness would be challenging. Therefore, we believe that *CERT* could help identify additional performance issues in the future. Overall, all issues were exposed in various SQL clauses and predicates, which may imply no common issues across tested DBMSs. We give two examples of minimized test cases to explain the issues we found as follows.

**An issue identified by rule 6.** Code 3.4 shows a test case exposing a performance issue in *MySQL*. Rule 6, which replaces **`ALL`** with **`DISTINCT`** in Table 3.2, exposed this

Code 3.5: Rule 11, which identified this issue in *CockroachDB*, removes either operand of an OR expression.

```
1 CREATE TABLE t0 (c0 INT);
2 INSERT INTO t0 VALUES (1), (2), (3), (4), (5), (6), (7), (8), (9), (10);
3 ANALYZE t0;
4
5 EXPLAIN SELECT t0.c0 FROM t0 WHERE
      (t0.c0 IS NOT NULL) OR (1 < ALL (t0.c0, t0.c0)); -- estimated rows: 3
6 EXPLAIN SELECT t0.c0 FROM t0 WHERE (t0.c0 IS NOT NULL); -- estimated rows: 10
```

issue. In Code 3.4, the first query in line 5 fetches the rows including duplicate rows, while the second query in line 6 excludes duplicate rows, so the cardinality of the second query should be no more than that of the first query. However, the estimated cardinality of the second query is greater than that of the first query, which is unexpected. Suppose a query $q$ with `ALL` is a subquery of another query $Q$ with `DISTINCT`, this issue affects whether `DISTINCT` should be pushed down to the execution of $q$ for an efficient query plan that aims to retrieve fewest rows from $q$. This issue was confirmed by the *MySQL* developers only three hours after we reported it.

**An issue identified by rule 11.** Code 3.5 shows another test case exposing another performance issue in *CockroachDB* by rule 11, which removes either operand of an OR expression. The predicate `(t0.c0 IS NOT NULL)` in the `WHERE` clause of the second query should fetch no more rows than the predicate `(t0.c0 IS NOT NULL)OR (1 < ALL (t0.c0, t0.c0))` of the first query. However, the estimated cardinality of the second query is greater than that of the first query, which is unexpected. This issue was caused by a buggy logic to handle the `OR` clause. In *CockroachDB*, given predicates `A` and `B`, the estimated cardinality of predicate `A OR B` is calculated by: $P(A\,OR\,B) = P(A) + P(B) - P(A\,AND\,B)$. However, when `A` and `B` depend on the same table or column, the estimated cardinality is unexpected. We found this issue by rule 11 in Table 3.2. Although this issue was known, it had been backlogged for around 10 months since it was first found. When we reported our test case, the developer opened a pull request in their git repository to fix it after three days.

**Performance analysis.** To investigate the extent to which the issues we found affect performance, we evaluated the query performance of the fixed issues 10 and 11

Code 3.6: The performance improvement by fixing our found issues.

```
1  CREATE TABLE t0 (c0 INT);
2  CREATE TABLE t1 (c0 INT);
3  CREATE TABLE t2 (c0 INT);
4  INSERT INTO t0 SELECT * FROM generate_series(1,1000);
5  INSERT INTO t1 SELECT * FROM generate_series(1001,2000);
6  INSERT INTO t2 SELECT * FROM generate_series(1,333100);
7  ANALYZE t0;
8  ANALYZE t1;
9  ANALYZE t2;
10
11 SELECT COUNT(*) FROM t0 LEFT OUTER JOIN t1 ON t0.c0<1 OR t0.c0>1 FULL JOIN t2
       ON t0.c0=t2.c0; -- 399ms → 321ms
12 SELECT COUNT(*) FROM t0 LEFT JOIN t1 ON t0.c0>0 WHERE (t0.c0 IS NOT NULL) OR
       (1 < ALL(t0.c0, t0.c0)); -- 131ms → 109ms
```

on a test case as shown in Code 3.6 that involves joining multiple tables. We could not consider unfixed issues, as it would be unclear how to determine the potential speedup. We executed both queries in lines 12 and 13 before and after the fixes of issues 10 and 11 respectively. After executing either query ten times, we found that the fixes improve the performance by an average of 20% and 17%, respectively. This improvement is due to the more accurate estimated cardinality which allows for more optimal joining orders.

Using *CERT*, we have found 13 unique issues in *MySQL*, *TiDB*, and *CockroachDB*. The fixes improve query performance by 19% on average.

## Q.2 Historic Bugs

**Method.** To evaluate whether *cardinality restriction monotonicity* is sufficiently general to identify previous performance issues that we identified in Table 3.1, we attempted using *CERT* to identify all three performance issues whose fixes changed the estimated cardinalities, namely issues #61631, #56714, and #9067. Specifically, based on the queries in the issue reports, we followed the step ② in Figure 3.1 to randomly construct 10,000 pairs of queries. Then, we checked whether any pair violated the *cardinality restriction monotonicity* before the fix, and adhered to the *cardinality restriction monotonicity* after the fix. If so and both query plans are structurally similar, we concluded that *cardinality restriction monotonicity* could

Code 3.7: Issue #56714 violates the *cardinality restriction monotonicity*.

```
1 ...
2 EXPLAIN SELECT MAX(a) FROM test; -- estimated rows: 1
3 EXPLAIN SELECT MAX(a) FROM test WHERE b=0; -- estimated rows: 500190
```

have identified the performance issue.

**Results.** All three previous performance issues caused by cardinality estimation can be found by *CERT*. For example, considering the performance issue #56714 in Code 3.2, Code 3.7 shows the pair of queries that *CERT* produces to identify the performance issue. The second query has an additional `WHERE` clause compared to the first query, so the estimated cardinality of the second query should be no more than that of the first query. However, due to incorrect usage of the index in column `b`, the second query scans all rows and has a higher estimated cardinality. After the fix, the estimated cardinality of the second query decreases to 1.

The *cardinality restriction monotonicity* can identify historical performance issues caused by cardinality estimation.

## Q.3 Efficiency

**Accuracy.** We evaluated whether *CERT* has higher accuracy in confirmed issues than *AMOEBA*. We evaluated this aspect based on the observation that around five in six reported bugs by *AMOEBA* were false alarms. A high rate of false alarms significantly limits the applicability of an automated testing technique. Recall that it is challenging to make a fair comparison as *CERT* and *AMOEBA* find different kinds of issues affecting performance.

**Results.** Table 3.4 shows the number of all and confirmed/fixed unique performance issues found by *CERT* and *AMOEBA*. The authors of *AMOEBA* reported 25 issues in *CockroachDB*, but only 6 issued (24% accuracy) were confirmed or fixed by developers. In comparison, for *CERT*, 50% of issues in *CockroachDB* and 100% of issues in *MySQL* and *TiDB* were confirmed or fixed. For *CockroachDB*, *CERT* found fewer performance issues than *AMOEBA*, because we did not report

Table 3.4: The number of all (**All**) and confirmed or fixed (**C/F**) unique performance issues.

| | ***CERT*** | | | ***AMOEBA*** | | |
|---|---|---|---|---|---|---|
| **DBMS** | **All** | **C/F** | **%** | **All** | **C/F** | **%** |
| *MySQL* | 3 | 3 | 100% | - | - | - |
| *TiDB* | 6 | 6 | 100% | - | - | - |
| *CockroachDB* | 4 | 2 | 50% | 25 | 6 | 24% |
| **Sum:** | 13 | 11 | 85% | 25 | 6 | 24% |

all found issues to avoid duplicate reports. Overall, these results suggest that *CERT* has high accuracy and is a practical technique for finding relevant performance issues. Despite these promising results, on a conceptual level, similar to *AMOEBA*, we cannot ensure that the performance issues would be considered as such by the developers.

**Throughput.** We evaluated whether *CERT* has a higher testing throughput than *AMOEBA*. State-of-the-art benchmarks and approaches, such as TPC-H [123], *AMOEBA* [107], and *APOLLO* [89] execute queries, which results in relatively low throughput. Therefore, it is expected that *CERT* validates more queries per second. To evaluate this, we determined the average number of test cases per second processed by *SQLancer+CERT* and *AMOEBA* in one hour.

**Results.** In *CockroachDB*, on average across 10 runs and one hour, *CERT* validates 714.54 test cases while *AMOEBA* exercises 1.85 test cases per second. This suggests a 386× performance improvement over *AMOEBA*. Recall that the throughput results are not directly comparable, as different approaches can find different kinds of issues. In addition, *CERT* is immune to performance fluctuation, because the estimated cardinalities are not affected by execution time. Therefore, *CERT* yields the same results in different hardware and network environments.

*SQLancer+CERT* validates 386× more test cases than *AMOEBA* across one hour and 10 runs on average, and is more than twice as accurate as *AMOEBA*.

Table 3.5: The average number of all queries (**Queries**), the queries that violate *cardinality restriction monotonicity* (**Violations**), and the geometric mean of percentage (%) of queries that violate the property across 10 runs and one hour.

| DBMS | Queries (#) | Violations (#) | Violations (%) |
|---|---|---|---|
| *MySQL* | 6,371,222 | 30,841 | 0.28% |
| *TiDB* | 2,895,203 | 8,108 | 0.27% |
| *CockroachDB* | 1,306,807 | 661 | 0.05% |
| | | | **Average:** 0.2% |

## Q.4 Sensitivity

**Sensitivity of rules.** We evaluated which rules presented in Table 3.2 contribute to finding the issues in Table 3.3. Specifically, we recorded which rules were applied to the bug-inducing test cases that we reported. We considered reported bug-inducing test cases, rather than all test cases—recall that *SQLancer+CERT* still reports violations when being run—as we expect the reported issues to be unique based on the developers' verdicts. Table 3.3 shows the rules applied to the corresponding bug-inducing test cases. Overall, 9 out of 12 rules have found at least one issue. Rule 9, which adds a predicate to `WHERE` clause, found the most issues, namely three. We believe that this is because the predicates in `WHERE` clause can vary significantly and thus be diverse and have a higher possibility to expose issues. No issue was found by rules 4, 10, and 12. Rule 4 applies to the `JOIN` clause in which other rules found several issues. Similarly, we believe that rule 10, which restricts a `WHERE` clause by an `AND` operator would find issues after the issues found by other rules applied to `WHERE` clause are fixed. Rule 12 applies to the `LIMIT` clause, which simply returns up to as many rows as specified in its argument. We explain that the simplicity of `LIMIT` explains that we have found no issues in its handling.

**Sensitivity of *cardinality restriction monotonicity*.** We expect that any violation of the *cardinality restriction monotonicity* property indicates a potential issue. To more thoroughly assess our hypothesis, we examined how many queries among all tested queries violate the *cardinality restriction monotonicity* property. If only a small portion of queries violates it, DBMSs are likely to adhere to the *cardinality restriction monotonicity* property, and any violation warrants further

investigation. Otherwise, the property may be not meaningful for developers. Table 3.5 shows the average number of all queries, the queries that violate *cardinality restriction monotonicity* property, and the geometric mean of the percentage of queries that violate the property across 10 runs and one hour. Overall, 0.2% of all generated queries violate the *cardinality restriction monotonicity* property. All three DBMSs, *MySQL*, *TiDB*, and *CockroachDB*, exhibit a similar rate of *cardinality restriction monotonicity* violations. The results demonstrate that DBMSs typically comply with the *cardinality restriction monotonicity* property, as more than 99.5% of generated queries do not violate it.

## 3.5 Discussion

We discuss some key considerations on the design of *CERT*, its characteristics, as well as the evaluation's results.

**Evaluating performance gains.** It is challenging to measure the overall performance gain that fixing the issues reported by *CERT* could achieve in practice. One issue is that measuring the overall performance impact might be misleading, because many other components in query optimizers can affect performance as well. For example, research on the Join Order Benchmark [100] demonstrated that a worse cardinality estimator might lead to better performance due to the issues in other components. In addition, 9 issues were confirmed, but remained unfixed. Since we lack domain knowledge to address the underlying issues, we cannot determine the performance gains that fixing these issues would cause.

**Generality.** The *cardinality restriction monotonicity* property might be applicable also to other kinds of data models than the traditional relational data model. For example, Neo4J, a graph DBMS, also uses a concept similar to query plans—termed execution plans—and cardinality estimation[7] (*EstimatedRows* field in execution plans). Its query optimization also depends on cardinality estimation, which we expect to comply with the *cardinality restriction monotonicity* property. More work

[7] `https://neo4j.com/docs/cypher-manual/current/execution-plans/#execution-plan-introduction`

is required to further explore *cardinality restriction monotonicity* in other DBMSs in the future.

**Threats to Validity.** Our evaluation results face potential threats to validity. One concern is internal validity, that is, the degree to which our results minimize systematic error. *CERT* validates test cases that are randomly generated by *SQLancer*. The randomness process may limit the reproducibility of our results. To mitigate the risk, we repeated all experiments 10 times to gain statistical significance. Another concern is external validity, that is, the degree to which our results can be generalized to and across other DBMSs. We selected various types of DBMSs including different purposes (community-developed: *MySQL* and company-backed: *TiDB* and *CockroachDB*) and languages (C/C++: *MySQL* and Go: *TiDB* and *CockroachDB*). These DBMSs have been widely used in prior research [106, 151, 150]. Given that DBMSs provide similar functionality and features, we are confident that our results generalize to many DBMSs. The last concern is construct validity, that is, the degree to which our evaluation accurately assesses what the results are supposed to. *CERT* found 13 unique performance issues, but only 2 issues had been fixed posing the threat that developers might not fix them in the future or might deem them less important. To address this threat, we communicated with the *TiDB* developers, who informed us that they plan to fix the issues and indicated an interest in using *SQLancer+CERT*.

## 3.6 Conclusion

This chapter presents *CERT*, a novel technique for finding performance issues through the lens of cardinality estimation in DBMSs. The key idea is to, given a query, derive a more restrictive query and validate that the DBMSs' estimated cardinalities that the original query has at least as great estimated cardinality as the more restrictive query; we refer to this property as *cardinality restriction monotonicity*. The evaluation has demonstrated that this technique is effective. Of the 13 unique issues that we found, 2 issues were fixed, 9 issues were confirmed, and 2 issues require further investigation. The fixes improved query performance by 19% on average. Unlike other testing approaches for performance issues, *CERT*

avoids executing queries, achieving a speedup of 386× compared to the state of the art. Finally, it is readily applicable. DBMSs expose estimated cardinalities as part of query plans to users; thus, *CERT* is a black-box technique that is applicable without modifications, even if the DBMSs' source code is inaccessible to the testers. Furthermore, since no queries are executed, *CERT* is resistant to performance fluctuations. Overall, we believe that *CERT* is a useful technique for DBMS developers and testers and hope that the technique will be widely adopted in practice.

# Chapter 4

# Differential Query Plans

To find logic bugs, in this chapter, we propose *Differential Query Plans* (*DQP*), a simple alternative to a state-of-the-art test oracle *TQS*. This work has been published in the 2024 ACM SIGMOD/PODS International Conference on Management of Data [87].

## 4.1 Introduction

A key feature of relational Database Management Systems (DBMSs) is to join the data in multiple tables using a `JOIN`. Various strategies and optimizations have been proposed specifically to optimize the execution of joins [127, 47, 46]. Given the complexity of such optimizations, query optimizers might apply a semantically incorrect optimization, which could result in logic bugs. In Code 4.1, the second query at line 8 triggers a logic bug in the DBMS, as it should return a non-zero result, instead of zero.

Recently, automated testing approaches for DBMSs have gained broad adoption to find logic bugs [106, 151, 149, 150], as they can often find many bugs that have been overlooked by manually written tests, which are costly to write. Importantly, such approaches provide so-called *test oracles*, mechanisms to check whether the computed result by the DBMS is correct. Test oracles are typically either combined with semi-random database and query generators [85, 182], or existing benchmarks such as TPC-H [123] or TPC-DS [177]. *TQS* [167] is an automated testing approach for detecting logic bugs in query optimizations. Notably, it is the state-of-the-art approach for testing join optimizations. To tackle the test-oracle problem, it simulates joins to derive a query's ground-truth results, and the simulation is performed by

Code 4.1: A bug found that may incur money loss by Differential Query Plans (*DQP*) in *MySQL*.

```
1  CREATE TABLE user(user_id DECIMAL PRIMARY KEY);
2  CREATE TABLE transaction(transitition_id TEXT, amount DECIMAL(10,2) NOT NULL);
3  CREATE INDEX i0 ON transaction(transitition_id(5));
4  INSERT INTO user VALUES(1), (2);
5  INSERT INTO transaction VALUES('1_c12934', 100000), ('1_e3b664', -10);
6
7  SELECT IFNULL(SUM(amount), 0) AS balance FROM user JOIN transaction ON
       transaction.transitition_id = user.user_id;  -- 99990.00 ✔
8  SELECT /*+ JOIN_ORDER(transaction, user)*/ IFNULL(SUM(amount), 0) as balance
       FROM user JOIN transaction ON transaction.transitition_id = user.user_id;
       -- 0.00 🐞
9  ------------------------------------------------------------------------------
10 nested_loop                                  nested_loop
11 +- table                                     +- table
12 |  table_name: user                          |  table_name: transaction
13 |  access_type: index                       |  access_type: all
14 +- table                                     +- table
15 |  table_name: transaction                   |  table_name: user
16 |  access_type: all                          |  access_type: eq_ref
```

table splitting. Specifically, it validates the correctness of join optimizations by splitting a given table into several sub-tables, and deriving ground-truth results of a query that joins these sub-tables by retrieving the given table. To generate more diverse test cases for finding more bugs, *TQS* randomly injects noise, such as `NULL` and `0`, to these sub-tables and models database schemas as a graph data model to evaluate the similarity of **`JOIN`**s. 115 bugs were found by this approach as claimed in the *TQS* paper. However, *TQS* suffers from two major challenges. First, this method is complex to understand and implement. *TQS* requires splitting and maintaining the data schemas with reference to a given table and modeling data schemas into graphs to decide whether two graphs are isomorphic for evaluating the similarity of queries. Second, the testing scope is small. *TQS* can apply only to equijoins. Although, as claimed in the *TQS* paper, this method could conceptually be extended to non-equijoins, this method cannot test other SQL features, which are directly executed to obtain results in *TQS*.

To understand the bug-finding effectiveness of *TQS*, we studied the bug reports of *TQS* in the public issue trackers. We identified 15 unique bugs, of which 14 were reported in the manner that the different results of executing the same query with different query hints. This manner is similar to Code 4.1, meaning that deriving the ground-truth results is not necessary for finding these bugs.

Based on our observation, in this chapter, we propose a simple and easy-to-understand approach to achieve the same level of bug-finding effectiveness as *TQS*. We propose checking the result consistency of executing different query plans of the same query, and refer to this method as *Differential Query Plan* (*DQP*). More formally, given a database $D$ and a query $Q$, the DBMS executes $Q$ on $D$ by the query plan $P$ to obtain the result $Q(P, D)$. For another possible query plan $P'$ of $Q$, if $Q(P, D) \neq Q(P', D)$, it indicates a bug.

To realize this technique as a black-box approach that eschews modifications to the DBMSs, we propose using query hints and setting system variables that are already provided by the DBMSs to affect the generated query plans. We believe that this technique is obvious and simple, but addresses both challenges of *TQS* and has a similar bug-finding effectiveness as *TQS*. Moreover, *DQP* can test more query optimizations rather than only join optimizations, as query hints and system variables can affect the optimizations of other SQL features. Importantly, the approach is simple and straightforward to implement, as *DQP* does not need to maintain data structures, such as graphs and sub-tables, for deriving the ground-truth results.

Code 4.1, which we briefly introduced above, shows a motivating example of a bug found by *DQP* in *MySQL*. Suppose we are in a bank scenario in which *MySQL* stores the user information in the table `user` and transaction records in the table `transaction`. Lines 1–3 create both tables with an index, and lines 4–5 insert data into both tables. The data in column `transaction_id` is composed of a user ID and a randomly generated transaction ID, such as `1_c12934` represents the user `1` making a transaction whose ID is `c12934`. Line 7 checks the balance of user `1` and obtains the expected result, `9990.00`. If the query results in an inefficient query plan, a database administrator might decide to enforce another query plan by adding a query hint as shown in line 8. The hint `/*+ JOIN_ORDER(transaction, user)*/` instructs the DBMS to process table `transaction` before `user` when performing the join. However, this query returns a wrong result `0.00`. Both query plans are shown in lines 10–16. This bug may incur severe consequences, as all money of user `1` is lost.

We implemented *DQP* in less than 100 lines of Java code for each DBMS based on *SQLancer* [150], a widely used DBMS testing framework. *SQLancer* provides generators for databases and queries that we reused. We evaluated *DQP* on the three DBMSs as in the *TQS* paper, *MySQL*, *MariaDB*, and *TiDB*. The results show

that *DQP* can reproduce 14 out of the 15 unique bugs found by *TQS*, and all 10 bugs related to join optimizations. In the same systems extensively tested by *TQS*, *DQP* found 26 previously unknown unique bugs, and 21 of them are logic bugs. Within these logic bugs, 15 are related to join optimizations suggesting that these bugs were overlooked by *TQS*, and 6 are related to other query optimizations that can not be found by *TQS*. Compared with *TQS*, *DQP* is simple yet general, and efficient.

Overall, we make the following contributions:

- We studied the bug-finding efficiency of a state-of-the-art work *TQS* for finding logic bugs in join optimizations;
- We demonstrated that the simple and easy-to-understand testing approach *Differential Query Plans* (DQP) testing shows the same level of bug-finding effectiveness as the more complex approach *TQS*;
- We implemented and evaluated the approach, which has found 26 unique, previously unknown bugs in widely-used DBMSs.

## 4.2 *TQS* Study

*TQS* successfully found bugs in *MySQL*, *MariaDB*, *TiDB*, and PolarDB. However, it is a complex method that requires implementing multiple graphs and tables as internal components for deriving the ground-truth results. In this section, we study *TQS* to understand whether these bugs found by *TQS* can be found by a simpler method by answering the following questions:

**RQ.1 Join-related Bugs.** How many bugs reported by *TQS* are related to join optimizations? *TQS* aims to find bugs in join optimizations, so we study how many found bugs are related to them.

**RQ.2 Bug-reporting Manners.** How were the bugs reported by *TQS*? Convincing developers that their DBMS, and not *TQS*, is computing an incorrect result might be challenging. As detailed in our answer to this question, we found that the bugs were explained not based on their ground-truth results, which motivates our simpler testing approach.

### 4.2.1 *TQS* Summary

*Transformed Query Synthesis* (*TQS*) [167] was proposed as an approach to detect logic bugs of join optimizations in DBMSs. *TQS* includes two major components: *Data-guided Schema and query Generation* (*DSG*) and *Knowledge-guided Query space Exploration* (*KQE*). *TQS* requires a wide table as an input table. The input table can be manually given, and in the *TQS* paper, the authors used the TPC-H[1] and KDD Competition[2] databases. First, *DSG* splits the wide table into multiple sub-tables through database normalization, which is an established technique that minimizes data redundancy and dependency by organizing data into separate tables. Then, *DSG* randomly constructs a query to join these sub-tables. The ground-truth results are derived by retrieving the wide table. The derivation process is not easy to implement as maintaining the relations between the given wide table and the split tables is necessary and complex. To make the generated queries more diverse, *KQE* evaluates whether a randomly generated query is similar to a previous query, and will adjust the random generation process to reduce the possibility of generating similar queries. The similarity is evaluated by modeling database schemas into an embedding-based graph, in which each query is a sub-graph, and *KQE* checks whether two sub-graphs are isomorphism. The authors claimed that *TQS* found 115 bugs within 24 hours including 7, 5, 5, and 3 types of bugs in *MySQL*, *MariaDB*, *TiDB*, and PolarDB.

### 4.2.2 Study Scope

We chose the public bug list from *TQS*[3] as our studied target. The public bug list is the only source that we could obtain to study *TQS* except for its paper. We did not involve the source code of *TQS*, because the source code is unavailable and the authors answered in emails that they were currently not able to provide it to us. We explain our attempts to obtain the source code in Section 4.6.

[1] `https://www.tpc.org/tpch/`
[2] `https://archive.ics.uci.edu/dataset/129/kdd+cup+1998+data`
[3] `https://github.com/xiutangzju/tqs/blob/d5f8f5/index.md`

Table 4.1: The bugs reported by *TQS*.

| DBMS | Bug | Type ID | Unique | Join | Query Plan |
|---|---|---|---|---|---|
| *MySQL* | 106713 | 3 | ✓ | | ✓ |
| *MySQL* | 106715 | 4 | ✓ | ✓ | ✓ |
| *MySQL* | 106716 | 7 | ✓ | ✓ | ✓ |
| *MySQL* | 106717 | 5 | ✓ | | ✓ |
| *MySQL* | 106718 | 2 | ✓ | | ✓ |
| *MySQL* | 106611 | 6 | | | ✓ |
| *MySQL* | 106710 | 1 | ✓ | | ✓ |
| *MySQL* | 99273 | | ✓ | | |
| *MySQL* | 109211 | | ✓ | ✓ | ✓ |
| *MySQL* | 109212 | | ✓ | ✓ | ✓ |
| *MariaDB* | 28214 | 8 | ✓ | ✓ | ✓ |
| *MariaDB* | 28215 | 9 | ✓ | ✓ | ✓ |
| *MariaDB* | 28216 | 10 | ✓ | ✓ | ✓ |
| *MariaDB* | 28217 | 11 | ✓ | ✓ | ✓ |
| *MariaDB* | 29695 | 12 | ✓ | ✓ | ✓ |
| *TiDB* | 33039 | 13 | | ✓ | ✓ |
| *TiDB* | 33041 | 14 | | ✓ | ✓ |
| *TiDB* | 33042 | 15 | ✓ | ✓ | ✓ |
| *TiDB* | 33045 | 16 | | ✓ | ✓ |
| *TiDB* | 33046 | 17 | | ✓ | ✓ |

### 4.2.3 Data Preprocessing

**Target DBMSs.** We studied the bug reports of *MySQL*, *MariaDB*, and *TiDB*. *TQS* was originally evaluated on four DBMSs: *MySQL*, *MariaDB*, *TiDB*, and PolarDB, but we observed that the public bug list does not include the bug reports of PolarDB. As a result, we studied the bug reports of the first three DBMSs, in which the authors claimed that *TQS* found 92 bugs of 17 bug types. While the paper claims that all found bugs had been reported, the actual number of bug reports in the public bug list is 21, namely 11 bug reports in *MySQL*, 5 in *MariaDB*, and 5 in *TiDB*.

To avoid that we missed any bug reports, as the list of found bugs provided by the *TQS* authors could be incomplete, we further searched the submission history of the first author in the corresponding issue trackers. We did not search for other authors as we could not find other authors' accounts in these issue trackers. Specifically,

we searched the issue trackers of *MySQL*,[4] *MariaDB*,[5] and *TiDB*.[6] We failed to identify other bug reports. We show all bug reports in Table 4.1, in which we excluded bug #106473, because the developers of *MySQL* rejected the bug report.[7] We also observed that bugs #106611, #106710, and #99273 were reported by a non-author, which is mentioned in the acknowledgment of the *TQS* paper. Based on our observation and investigation, we infer that the 92 bugs in the paper refer to bug-inducing test cases, a large portion of which is duplicate, instead of unique, valid bugs.

**Bugs in the *TQS* paper.** To validate our assumption that 92 bugs in the *TQS* paper refer to bug-inducing test cases, we did a matching analysis to examine the correlation between the public bug list and the 17 bug types from Table 4 in the *TQS* paper. Specifically, for a bug in the public bug list, we searched for a matching bug type with the same bug status, bug severity, and similar description described in the *TQS* paper. We observed that the titles of bug reports are similar to the descriptions of bug types, but not exactly the same, so we adopted the algorithm gestalt pattern matching [144] to calculate whether two strings are similar, and the similarity is indicated as a floating number ranging from 0 to 1 indicating the degree of similarity. We deemed the highest score as the closest match. After matching, we further manually examined and corrected the matching according to the semantic information of bug reports. Specifically, we matched bug #28217 and bug type 11 as they have similar descriptions—incorrect results by limiting join buffers but different severity levels, although they have different bug severity. We also matched bug #33042 and bug type 15 as they have the same keyword "empty resultset".

In Table 4.1, the column *Type ID* shows our matching results. 17 bug reports can be matched to the 17 bug types in the *TQS* paper, and 3 bug reports have no matching bug types. This result shows that each bug report in the public bug list refers to a bug type in the *TQS* paper, rather than indicating a bug. Three bug reports have no matched bug types, and a possible explanation is that they were

[4] `https://bugs.mysql.com/search.php?cmd=display&status=All&severity=all&reporter=16399198`

[5] `https://jira.mariadb.org/browse/MDEV-29695?jql=reporter="XiuTang"`

[6] `https://github.com/pingcap/tidb/issues?q=is:issue+author:xiutangzju`

[7] `https://bugs.mysql.com/bug.php?id=106473`

submitted after submitting the *TQS* paper.

**Unique bugs.** Given the 20 bug reports, we investigated the uniqueness of these bugs according to the developers' responses. Typically, developers clearly respond to a bug report if it duplicates a previously reported bug. We carefully reviewed all developers' responses in the bug reports to check whether a bug is a duplicate.

In Table 4.1, the column *Unique* shows the unique bugs. 15 of the 20 bugs are unique. For *MySQL*, bug #106611 is a duplicate of the previously found bug #105773, and the developers confirmed this duplication within 24 hours after submitting the bug report. For *TiDB*, bugs #33049, #33041, #33045, #33046 are duplicates of bug #33042 reported by *TQS* as well, and these duplicates were confirmed by developers within 3 days after submitting the bug reports. We further study *TQS* based on the 15 unique bugs.

## RQ.1 Join-related Bugs

We evaluated how many bugs are related to join optimizations. *TQS* aims to detect logic bugs in join optimizations, *DSG* derives ground-truth results of a join, and *KQE* drives the test case generation towards exercising diverse join optimizations. Therefore, it is important to determine how many bugs are related to join optimizations. We examined the test cases in the bug reports and checked whether a test case includes at least one **`JOIN`** clause. If so, we deemed the bug report to be related to join optimizations. Although join optimizations might also apply to other clauses, such as subqueries [45], *DSG* cannot derive the ground-truth results for these clauses,[8] so we considered only queries with **`JOIN`** clauses as join-related queries for this study.

In Table 4.1, the column *Join* shows the bug reports whose test case includes at least one **`JOIN`** clause. In total, 10 of 15 unique bugs (67%), are related to join optimizations. For the five bugs that are not related to join optimizations, #106713, #106717, #106718, #106710, #99273, a common feature is that the bug-inducing test cases of them include at least one **`SUBQUERY`** clause. The core components of *TQS* show no obvious contribution to the finding of these non-join-related bugs. It is

[8] In Section 3.3 of *TQS* paper, the authors claimed: "*DSG randomly generates other expressions based on the join clauses.*"

Code 4.2: *MySQL* bug #106713 found by *TQS*.

```
1 CREATE TABLE IF NOT EXISTS t0(c0 DECIMAL ZEROFILL  COLUMN_FORMAT DEFAULT );
2 INSERT HIGH_PRIORITY INTO t0(c0) VALUES(NULL), (2000-09-06), (NULL);
3 INSERT INTO t0(c0) VALUES(NULL);
4 INSERT DELAYED INTO t0(c0) VALUES(2016-02-18);
5
6 SELECT t0.c0 FROM t0 WHERE t0.c0 IN (SELECT t0.c0 FROM t0 WHERE (t0.c0 NOT IN
      (SELECT t0.c0 FROM t0 WHERE t0.c0 )) = (t0.c0) ); --
      {0000001985},{0000001996}
7 SELECT t0.c0 FROM t0 WHERE t0.c0 IN (SELECT /*+ no_semijoin()*/ t0.c0 FROM t0
      WHERE (t0.c0 NOT IN (SELECT t0.c0 FROM t0 WHERE t0.c0 )) = (t0.c0) ); --
      empty set
```

Code 4.3: *MySQL* bug #99273 found by *TQS*.

```
1 CREATE TABLE t1 (a INT, b INT);
2 INSERT INTO t1 VALUES (1,1),(2,1),(3,2),(4,2),(5,3),(6,3);
3 SET SQL_MODE = 'ONLY_FULL_GROUP_BY';
4 SELECT a FROM t1 as t1 GROUP BY a HAVING (SELECT t1.a FROM t1 AS t2 GROUP BY b
      LIMIT 1); -- {1},{2},{3},{4},{5},{6}
5 INSERT INTO t1 values (null, 4);
6 SELECT a FROM t1 as t1 GROUP BY a HAVING (SELECT t1.a FROM t1 AS t2 GROUP BY b
      LIMIT 1); -- empty set
```

unclear how *TQS* constructs the ground-truth results for these cases, because *TQS* directly executes non-join SQL clauses to obtain the results.[9]

## RQ.2 Bug-reporting Manners

We examined the bug descriptions and test cases of these bug reports to study how these bugs were reported and explained. We observed that all 10 join-related bugs and 14 of 15 unique bugs were reported in the same manner, by demonstrating that different query plans of the same query compute inconsistent results. Code 4.2 shows an example of bug #106713 in *MySQL*. The author argued the buggy behavior by showing that a query with the query hint `/*+ no_semijoin()*/` returns a different result than the same query, but without the query hint. A query hint suggests the DBMS to generate or avoid a specific query plan, and `no_semijoin()` disables the semijoin during query optimization. The only exception is bug #99273, as shown in Code 4.3. This bug is included in the public bug list, but can not be matched to any bug type in the *TQS* paper. This bug was explained by the unexpected behavior that a query returns fewer rows after inserting a row with `NULL`. The root reason

[9] In Section 3.4 of *TQS* paper, the authors claimed: "*DSG also executes the generated filters and projections defined in the AST*".

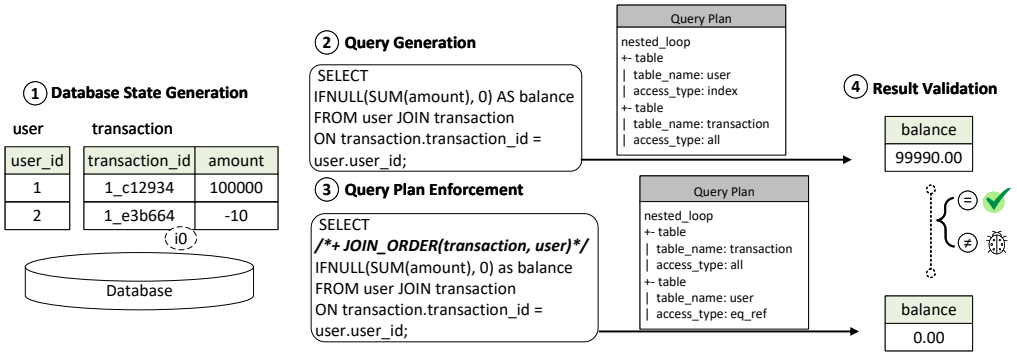


Figure 4.1: Overview of *DQP*.

is an incorrect optimization for **SUBQUERY**, but is not related to **JOIN**. It is unclear how *TQS* derives the ground-truth result for a **SUBQUERY**, as *TQS* can only derive the results of **JOIN**. We also noticed that this bug was found in 2020, while all other bugs were found in 2022. Based on our observations, we assume that most of these bugs can also be found by checking inconsistencies in executing the same query with different query plans, which is a much simpler testing method than *TQS*.

14 of the 15 unique TQS bugs and all 10 **JOIN**-related bugs were reported by showing discrepancies across the executions of different query plans of the same query.

## 4.3 Approach

We propose a simple approach, which we term *Differential Query Plans* (*DQP*) testing, to find bugs in join optimizations. Our core idea is to find bugs by comparing the results of the same query while enforcing different query plans for it. Compared to *TQS*, *DQP* does not require implementing graph and table structures for deriving ground-truth results. Moreover, *DQP* supports finding bugs in a variety of query optimizations instead of only in equijoin optimizations. Our key contribution is not the novelty of the approach, but the insight that a simple and easy-to-understand technique performs at the same level as a more sophisticated approach.

**Approach overview.** Figure 4.1 shows an overview of *DQP* illustrated based on Code 4.1. First, *DQP* generates a database state $D$ in step ①. Then, in step ②, *DQP* generates a query $Q$, and enforces different query plans $P$ and $P'$ to execute it. In step ③, *DQP* obtains the results of the executions. A discrepancy in the results, that is, $Q(P, D) \neq Q(P', D)$, indicates a potential bug.

**Database State Generation (①)** For a fully automated approach, we assume $D$ to be randomly generated. Common generation methods include mutation-based methods [219, 106] and rule-based methods [182, 150, 151, 149]. To create $D$ in Figure 4.1, *DQP* executes lines 1–5 in Code 4.1. Generating a database state is not a contribution of this paper, and *DQP* can be paired with any database state generation method. In fact, $D$ could also be manually specified.

**Query Generation (②)** Based on $D$, *DQP* randomly generates a query $Q$ in step ② whose results we subsequently automatically validate to find bugs. Similar to database state generation, many query generation approaches have been proposed [11, 24, 89, 117, 139, 157, 165], and *DQP* can, in principle, be paired with any of these query generation methods.

**Query Plan Enforcement (③)** *DQP* executes $Q$, for which the DBMS derives a query plan $P$. Then, *DQP* attempts to force the DBMS to derive an alternative query plan $P'$ for the same query. Query hints and system variables are two ways that affect query plans by using SQL statements without the need to modify the source code of DBMSs. In Section 4.4 we describe more details about both ways. In Figure 4.1, *DQP* enforces a $P'$ that has a different join order than $P$ by the query hint `/*+ JOIN_ORDER(transaction, user)*/`.

**Result Validation (④)** In step ③, *DQP* executes $Q(P, D)$ and $Q(P', D)$ to obtain results, and we check their consistency. Here, $Q(P, D) = 99990.00$, while $Q(P', D) = 0.00$, so a bug is found.

## 4.4 Implementation

We implemented *DQP* in *SQLancer*,[10] a DBMS testing framework that randomly generates database states and queries complying with the SQL grammar, and subsequently refer to our prototype as *SQLancer+DQP*. We discuss the technical details for implementing *SQLancer+DQP* in this section.

### 4.4.1 Database and Query Generation

We adopted the grammar-based method provided by *SQLancer* to randomly generate syntactically correct SQL statements. *SQLancer* encodes the grammar of DBMSs into a tree structure, and *DQP* randomly walks the tree to generate an SQL statement. To generate $D$, *DQP* generates non-query statements, such as **`CREATE TABLE`** and **`CREATE INDEX`**. Similarly, to generate $Q$, *DQP* randomly walks the tree to generate query statements, that is **`SELECT`**. Grammar-based generation methods are also used in *TQS* and SQLSmith [182].

**Generating `JOIN`.** While *SQLancer* already generates **`JOIN`**s for many DBMSs, it lacks the support of **`JOIN`**s for *MySQL* and *MariaDB*. We updated *SQLancer* to support generating the **`JOIN`** clause for *MySQL* and *MariaDB* with reference to the code of **`JOIN`** in *TiDB*'s implementation[11] in *SQLancer*.

### 4.4.2 Query Plan Enforcement

Query hints and system variables are two ways that affect query plans by SQL statements without requiring modifications to the source code of the DBMS under test.

**Query hints.** A query hint is a comment-like clause in a query and can affect the behaviors of the query optimizer. Query hints are widely supported by popular DBMSs, such as *MySQL*,[12] *MariaDB*,[13] and *TiDB*.[14] For the query hints that require

[10] `https://github.com/sqlancer/sqlancer`

[11] `https://github.com/sqlancer/sqlancer/blob/cddff6/src/sqlancer/tidb/ast/TiDBJoin.java`

[12] `https://dev.mysql.com/doc/refman/8.0/en/optimizer-hints.html`

[13] `https://mariadb.com/kb/en/optimizer-hints/`

[14] `https://docs.pingcap.com/tidb/stable/optimizer-hints`

table or column names as parameters, we randomly generate such names according to the query. In Figure 4.1, the query hint `/*+ JOIN_ORDER(transaction, user)*/` enforces the query optimizer to join both tables in a specific order, which is the difference between $P'$ and $P$.

**System variables.** Another way to affect query plans is by setting system variables that affect the query optimizer. The variable `optimizer_switch` for *MySQL*[15] and *MariaDB*[16] is a system variable that affects query optimization and thus the generated query plans. Concretely, *DQP* executes a **`SET`** statement with the query to configure the system variable to enforce a different query plan. For example, in *MariaDB*, *DQP* may execute this **`SET`** statement and the query: **`SET`** `STATEMENT optimizer_switch='index_merge=on'FOR` **`SELECT`** `t0.c0` **`FROM`** `t0`. The prefix **`SET`** configures the system variable taking effect for the following **`SELECT`** statement, and `index_merge` controls whether to enable the index merge optimization.

**Efficiency consideration.** For testing efficiency, we enforce multiple query plans $\{P', P'', ...\}$ by enumerating all possible query hints and values of the system variable in an iteration. Doing so is feasible as we observed that both query hints and potential values associated with the system variable are finite and small in number. We examined DBMS documents and extracted 32 query hints and 26 options for the system variable `optimizer_switch` in *MySQL*, 37 options for the system variable `optimizer_switch` in *MariaDB*, and 22 query hints in *TiDB* for enforcing different query plans. In Figure 4.1, for simplicity, we only show the executions of $P$ and $P'$.

### 4.4.3 Result Validation

We initially observed false alarms during step ④. As clarified by the DBMS developers, these false alarms were due to ambiguous queries, which refer to queries whose results are not guaranteed to be consistent or predictable. To exclude these false alarms, we identify ambiguous queries by checking whether a different row order in tables affects the result. After implementing this technique, we observed no false alarms. After an iteration, *DQP* returns back to step ① or ② to start a new

[15] `https://dev.mysql.com/doc/refman/8.0/en/switchable-optimizations.html`
[16] `https://mariadb.com/kb/en/optimizer-switch/`

iteration. Since generating $D$ is relatively slow, *DQP* returns to step ② by default. Only after a fixed number of iterations, does *DQP* return to step ①. The number is configurable, and we configured the number to be 10,000, which was empirically determined to work well in prior work [85].

**Ambiguous queries.** Ambiguous queries may incur false alarms, as also observed in other DBMS testing approaches [149, 150]. Investigating and analyzing all categories of ambiguous queries is challenging and exceeds the scope of this paper. We discuss the ambiguous queries that we encountered in practice. One kind of ambiguous queries[17] is including non-aggregated columns in a `SELECT` clause. A column that is not included in `GROUP BY`, is a non-aggregated column. If we include non-aggregate columns in `SELECT`, some DBMSs return the non-aggregated column of a random row from each group. The other DBMSs, such as PostgreSQL, reject such ambiguous queries. Code 4.4 shows a concrete example that we encountered when testing *TiDB*. For the test case in the top half, both queries retrieve the column `t0.c0`, which is not included in `GROUP BY`. The function `CAST` converts both 0.9 and 0.8 to 1, so both rows in `t0` will be in the same group, but both queries return different results as they return a random row of the group. These ambiguous queries cause false alarms in the validation step ④.

**Algorithm 1:** Ambiguous query identification

**Input:** query: $Q$, two query plans of $Q$: $P$ $P'$, database: $D$

```
ambiguous = false
Minimize(Q, D, P, P')
for D' in Permutation(D) do
    if Validate(P, P', Q, D') ≠ Validate(P, P', Q, D) then
        ambiguous = true
        break
    end if
end for
```

**Output:** *ambiguous*

**Ambiguous query identification algorithm.** Algorithm 1 shows our algorithm to identify ambiguous queries by checking whether a different row order in tables

[17] `https://docs.pingcap.com/tidb/v6.5/dev-guide-unstable-result-set`

Code 4.4: An unstable behavior identified by recondition.

```
1  CREATE TABLE t0(c0 FLOAT);
2  INSERT INTO t0 VALUES (0.9), (0.8);
3  CREATE INDEX i0 ON t0(c0);
4  SET @@sql_mode='';
5
6  SELECT t0.c0 FROM t0 GROUP BY CAST(t0.c0 AS DECIMAL); -- {0.8}
7  SELECT /*+ IGNORE_INDEX(t0, i0)*/t0.c0 FROM t0 GROUP BY CAST(t0.c0 AS
       DECIMAL); -- {0.9}
8
9  --------------------------------------------------------------------------------
10 CREATE TABLE t0(c0 FLOAT);
11 INSERT INTO t0 VALUES (0.8), (0.9);
12 CREATE INDEX i0 ON t0(c0);
13 SET @@sql_mode='';
14
15 SELECT t0.c0 FROM t0 GROUP BY CAST(t0.c0 AS DECIMAL); -- {0.8}
16 SELECT /*+ IGNORE_INDEX(t0, i0)*/t0.c0 FROM t0 GROUP BY CAST(t0.c0 AS
       DECIMAL); -- {0.8}
```

affects the validation result. First, to reduce the computational complexity, we minimize $Q$ and $D$. A bug-inducing test case that is identified by the step ③ typically includes hundreds of SQL statements to initialize database states and query results. To efficiently execute them multiple times for identifying ambiguous queries, we minimize each test case both using C-Reduce [146] and manually. Figure 4.1 includes the minimized test case that includes only two tables and four rows. Then, we permutate the rows in all tables. For each permutation $D'$, if it affects the validation result, $Validate(P, P', Q, D') \neq Validate(P, P', Q, D)$, the query is an ambiguous query. In Code 4.4, permutating the rows in `t0` incurs a different result for the second query, and the discrepancy disappears. *DQP* identifies and ignores this test case as the discrepancy likely indicates an ambiguous query.

**Algorithm scalability.** We believe that in practice, Algorithm 1 is feasible, as databases used to reproduce most bugs in existing works are small after minimization; for example, the average number of SQL statements in minimized bug-inducing test cases is 3.69 across 499 historical bugs found by *SQLancer*.[18] That most test cases can be reproduced with only small bug-inducing test cases has been observed in various testing works, such as for file systems [120], Java programs [5], and answer-set programs [128], and is known as the so-called *small-scope hypothesis*. For Figure 4.1,

[18] `https://github.com/sqlancer/bugs/blob/96cbb8/bugs.json`

each table includes two rows, so the number of permutations is $2! * 2! = 4$. Except for the original permutation, the loop at line 3 executes at most 3 times. Through the test case minimization, the execution number of the loop reduces exponentially.

## 4.5 Evaluation

To evaluate the effectiveness and efficiency of *DQP*, we sought to answer the following questions:

**Q.1 Bug Reproduction.** Can *DQP* find the bugs found by *TQS*?

**Q.2 New Bugs.** Can *DQP* find previously unknown bugs?

**Q.3 Bug-finding Efficiency.** How efficiently can *DQP* find bugs?

**Q.4 Bug-finding Effectiveness.** How effective is *DQP* compared to other test oracles for finding logic bugs?

**Q.5 Coverage.** To what extent does *DQP* cover query optimizers?

**Tested DBMSs.** We tested the same DBMSs, *MySQL*, *MariaDB*, and *TiDB* as we studied in Section 4.2. *MySQL* is one of the most popular relational DBMSs. *MariaDB* is another popular DBMS that was forked from *MySQL*. *TiDB* is a popular enterprise-class DBMS, and its open version on GitHub has been starred more than 35k times. Importantly, these DBMSs were also tested by *TQS*. Because the bug reports of PolarDB were not published by the *TQS* authors, we did not test PolarDB. For Q2 and Q4, we used the latest available development versions (*MySQL*: 8.1.0, *MariaDB*: 11.1.2, *TiDB*: 7.4.0). For a fair comparison in Q3, we used the same versions as *TQS* used (*MySQL*: 8.0.28, *MariaDB*: 10.8.2, *TiDB*: 5.4.0). All DBMSs were running in default configurations.

**Experimental infrastructure.** We conducted all experiments on an AMD EPYC 7763 processor that has 64 physical and 128 logical cores clocked at 2.45GHz. Our test machine uses Ubuntu 22.04.2 with 512 GB of RAM, and a maximum utilization of 40 cores.

## Q.1 Bug Reproduction

We evaluated whether *DQP*, as a simple testing approach, can find the logic bugs found by *TQS*. As found in Section 4.2, 14 of 15 unique bugs and all 10 join-related bugs were reported in the same manner, which is similar to the approach of *DQP*, so we assumed *DQP* can find these bugs as well. We used the test cases in the public bug reports from *TQS* (see Table 4.1) as the initial database state and the original query, and *DQP* enforces a different query plan for the query followed by step ③ in Figure 4.1. If we observe any discrepancy between the results returned by the original and derived queries, we deem that the bug can be found by *DQP*.

**Results.** *DQP* can identify 14 of 15 unique bugs that were reported by *TQS*. In Code 4.2, which shows a previous-discussed bug-inducing test case found by *TQS*, the bug-inducing test case includes two queries, the only difference of which is the query hint, and the description of the bug reason is "`no_semijoin` produce wrong results". Therefore, *DQP* can derive the second query by adding the query hint `no_semijoin`, and easily find this bug.

We also found that all 10 join-related bugs can be found by *DQP*, as they were all reported in a manner like Code 4.2. Although *TQS* finds a bug by comparing the execution result against a ground-truth result, the *TQS* authors explained the bugs to developers by providing a reference query that has an inconsistent result as the buggy query. The result demonstrates that *DQP* can find the majority of bugs that were found by *TQS*.

14 of 15 unique bugs, and all 10 join-related bugs found by *TQS* can be detected by *DQP*.

## Q.2 New Bugs

Apart from reproducing existing bugs found by *TQS*, we evaluated whether *SQLancer+DQP* can find previously unknown bugs. We would expect so due to the broader testing scope. *DQP* can be applied also to non-equijoin and queries without **`JOIN`**s, given that these queries' query plans can be influenced by query hints or system variables. We ran *SQLancer+DQP* twice for 24 hours on three DBMSs

Table 4.2: Previously unknown and unique bugs found by *DQP*.

| DBMS | Bug | Status | Severity | Logic | Join |
|---|---|---|---|---|---|
| *MySQL* | 112243 | Confirmed | Non-critical | ✓ | ✓ |
| *MySQL* | 112242 | Confirmed | Serious | ✓ | |
| *MySQL* | 112264 | Confirmed | Serious | ✓ | ✓ |
| *MySQL* | 112269 | Confirmed | Serious | ✓ | ✓ |
| *MySQL* | 112296 | Confirmed | Non-critical | ✓ | ✓ |
| *MariaDB* | 32076 | Confirmed | Major | ✓ | |
| *MariaDB* | 32105 | Confirmed | Major | ✓ | ✓ |
| *MariaDB* | 32106 | Confirmed | Major | ✓ | ✓ |
| *MariaDB* | 32107 | Confirmed | Major | ✓ | ✓ |
| *MariaDB* | 32108 | Confirmed | Major | ✓ | ✓ |
| *MariaDB* | 32143 | Confirmed | Major | ✓ | ✓ |
| *MariaDB* | 32186 | Confirmed | Major | ✓ | ✓ |
| *TiDB* | 46535 | Confirmed | Major | ✓ | ✓ |
| *TiDB* | 46538 | Confirmed | Moderate | | |
| *TiDB* | 46556 | Confirmed | Major | | |
| *TiDB* | 46580 | Fixed | Critical | ✓ | ✓ |
| *TiDB* | 46598 | Confirmed | Major | ✓ | |
| *TiDB* | 46599 | Confirmed | Major | ✓ | |
| *TiDB* | 46601 | Fixed | Critical | ✓ | |
| *TiDB* | 47019 | Confirmed | Major | ✓ | |
| *TiDB* | 47020 | Confirmed | Major | ✓ | ✓ |
| *TiDB* | 47286 | Confirmed | Major | ✓ | ✓ |
| *TiDB* | 47345 | Confirmed | Critical | ✓ | ✓ |
| *TiDB* | 47346 | Confirmed | Major | | |
| *TiDB* | 47347 | Confirmed | Major | | |
| *TiDB* | 47348 | Confirmed | Moderate | | |
| **Sum:** | 26 | | | 21 | 15 |

aiming to find bugs. To reduce the possibility of finding duplicate bugs, between two runs, we disabled the query hints and system variables that contribute to the bugs found in the first run. Developers typically confirm our bug reports within several days; however, due to the development process, these bugs usually require several weeks or months to be fixed for the next release version. A duplicated bug reported is identified by developers as they typically explicitly respond if a duplicate issue is reported. To avoid reporting duplicate bug reports, we reported only the bugs that were likely unique, rather than all bug-inducing test cases. Specifically, for each query hint and available option of the system variables, we report at most one bug.

**Bug overview.** Table 4.2 shows the unique, previously unknown 26 bugs found by *SQLancer+DQP*. The column *Logic* represents whether the bug is a logic bug, and the column *Join* represents whether the bug is related to join optimizations. We submitted 32 bug reports to developers, in which 26 bugs were confirmed as unique and previously unknown bugs, 1 bugs were duplicates, 1 was waiting for further analysis, and 4 bugs were false alarms due to ambiguous queries. These false alarms inspired us to design the Algorithm 1, and we observed no false alarms after implementing the algorithm. Note that #47019 and #47020 in *TiDB* are potential duplicates as they cannot be observed after fixing the bug #46601. We are awaiting the developers' response to confirm whether they are duplicates.[19] Therefore, we deemed them unique as developers did not claim they were duplicates.

**Logic bugs.** Out of the unique and previously unknown 26 bugs, 21 were logic bugs in query optimizations, as they were found due to inconsistent results returned by different query plans of the same query. The non-logic bugs were due to internal errors and crashes that can be exposed without comparing the results of executing different query plans.

**Join-related bugs.** 15 of 21 logic bugs were about join optimizations as their minimized test case requires at least one `JOIN`. While *DQP* can find bugs across various query optimizations, the majority of the found bugs relate to join optimizations, and the bugs in join optimizations ideally should be found by *TQS*. To identify bugs related to join optimizations, we followed the same classification method as in Section 4.2, which considers join optimization bugs as logic bugs that include at least one `JOIN` clause. The results show that join optimizations are more buggy than other query optimizations, and *TQS* overlooked our found bugs in join optimizations. Our simple approach, *DQP*, shows a surprising effectiveness in finding these join-optimization bugs.

**Bug severity.** One important question is whether the bugs *DQP* found are important. For *TiDB*, whose bug severity is specified by developers, 12 of 14 found bugs have the bug severity of *Major* or *Critical*, which represents that the bugs

[19] `https://github.com/pingcap/tidb/issues/47019#issuecomment-1734913792`

Code 4.5: A bug found by *DQP* by setting system variables in *MySQL*.

```
1 CREATE TABLE t0(c0 INT);
2 INSERT INTO t0(c0) VALUES(1);
3 CREATE INDEX i0 USING HASH ON t0(c0) INVISIBLE;
4
5 SELECT t0.c0 FROM t0 WHERE COALESCE(0.6) IN (t0.c0); -- {}
6 SET SESSION optimizer_switch = 'use_invisible_indexes=on';
7 SELECT t0.c0 FROM t0 WHERE COALESCE(0.6) IN (t0.c0); -- {1}
```

seriously affected the target system and typically have high priorities to be fixed. The one bug found by *TQS* also has the bug severity of *Critical*. For *MySQL* and *MariaDB*, we found that the bug severities are specified by users and typically not updated by the developers. Thus, we believe that they are inaccurate. However, since they were reported in the *TQS* paper, we also provide them for comparison. 10 of the 12 found bugs in *MySQL* and *MariaDB* have bug severity of *Serious* and *Major*, while all 12 bugs in *TQS* paper have the bug severity of *Serious*, *Major*, or *Critical*. To further demonstrate the importance of our found bugs, we present two selected examples.

**Example 1: a bug found by setting system variables.** Code 4.5 shows an example of a bug we found in *MySQL* by controlling the system variable `optimizer_switch`. The configuration `use_invisible_indexes` controls whether the query optimizer considers invisible indexes, which are excluded from query optimizations by default. In this example, the index `i0` is set to `INVISIBLE`, so the first query retrieves the data without using the index. When setting the variable to `use_invisible_indexes=on`, the second query uses index `i0` to retrieve the data. This bug is due to the incorrect index optimization. Without *DQP*, it is hard to know if a query using that index returns an incorrect result. We specified the severity *Serious* when submitting the bug report. Of the 21 logic bugs *DQP* found in query optimizations, 10 bugs are found by setting system variables.

**Example 2: a bug found by setting query hints.** Code 4.6 shows another example bug we found in *TiDB* by setting the query hint `MERGE_JOIN`. The query hint `MERGE_JOIN` instructs the query optimizer to use the sort-merge join algorithm when executing the **`JOIN`** operator. With reference to the developers' reply, the root cause for the bug is due to the operation `Projection`, which corresponds to a projection

Code 4.6: A bug found by *DQP* by setting query hints in *TiDB*.

```
1  CREATE TABLE t0(c0 INT);
2  CREATE TABLE t1(c0 BOOL, c1 BOOL);
3  INSERT INTO t1 VALUES (false, true);
4  INSERT INTO t1 VALUES (true, true);
5  CREATE VIEW v0(c0, c1, c2) AS SELECT t1.c0, LOG10(t0.c0), t1.c0 FROM t0, t1;
6  INSERT INTO t0(c0) VALUES (3);
7
8  SELECT COUNT(v0.c2) FROM v0,  t0 CROSS JOIN t1 ORDER BY -v0.c1; -- empty set
9  SELECT /*+ MERGE_JOIN(t1, t0, v0)*/COUNT(v0.c2) FROM v0,  t0 CROSS JOIN t1
       ORDER BY -v0.c1; -- {4}
```

operation in relational algebra. When using `MERGE_JOIN`, the operation `Projection` wrongly returns an empty output, so the first query returns an unexpected empty result. *DQP* found this bug by comparing the results of the same query with and without the query hint `MERGE_JOIN`. `Projection` is prevalent as it is usually executed for a **`SELECT`**, so the developers assigned the severity *Critical* to this bug and fixed it in one week. Of the 21 logic bugs *DQP* found in query optimizations, 11 bugs are found by setting query hints.

*DQP* enabled us to find and report 26 unique, previously unknown bugs, which were missed by *TQS*.

## Q.3 Bug-finding Efficiency

We evaluated how many bugs *DQP* can find in 24 hours. We ran *DQP* with the default configurations of *SQLancer* for 24 hours, and measured the number of bug-inducing test cases. We excluded bugs that cause crashes or internal errors, because they are not directly found by *DQP*, but by an implicit test oracle. Although *TQS* was evaluated in a similar experiment in Section 5.2 of *TQS* paper, it is challenging to make a fair comparison, due to the aforementioned unavailability of its source code, and because some experimental configurations are not clear. First, both *TQS* and *DQP* adopt a grammar-based test case generation method, but the implementation differences are unclear, such as the possible expressions for **`WHERE`**. While other DBMS testing works [219, 85] also omit detailed descriptions, they provide the source code, from which this information can be extracted. Second, *TQS* supports multiple threads, but we have not found the number of threads used for their efficiency

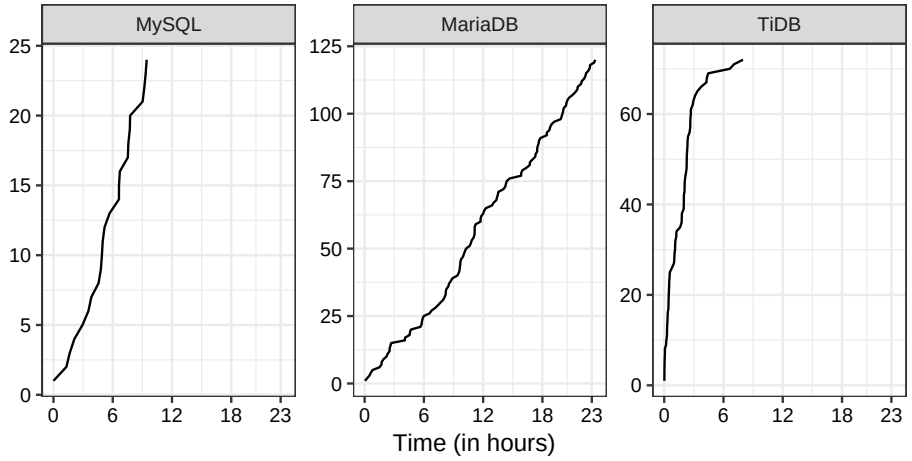


Figure 4.2: Number of bug-inducing test cases found by *DQP* in 24 hours and 10 runs.

evaluation in Figure 8 of Section 5.2 in the *TQS* paper. For *SQLancer+DQP*, we ran 10 threads, which is a common practice for evaluating testing tools [95] and is also used by other DBMS testing work [85]. Third, it is unclear whether the *TQS* authors counted only logic bugs or all kinds of bugs. Last, *TQS* and *DQP* were evaluated on different machines, which have a significant impact on efficiency, so their efficiency results are not directly comparable.

**Results.** Figure 4.2 shows the number of bug-inducing test cases found by *DQP* in *MySQL*, *MariaDB*, and *TiDB* for 24 hours. In total, *DQP* found 24, 120, and 72 bug-inducing test cases in three DBMSs respectively. Due to several crash bugs found by *SQLancer+DQP*, *MySQL* and *TiDB* exited at around 9 hours. Compared with the results as shown in Section 5.2 of *TQS* paper, *DQP* shows a clear advancement over *TQS*.

Compared with the results in Section 5.2 of the *TQS* paper, *DQP* demonstrates significant progress in bug detection efficiency compared to *TQS*. Recall that it is challenging to conduct a fair comparison with *TQS*. Nevertheless, the substantial number of bug-inducing test cases found by *SQLancer+DQP* demonstrates its efficiency even without sophisticated techniques to improve test-case generation.

*SQLancer+DQP* found 216 bug-inducing test cases in 24 hours in *MySQL*, *MariaDB*, and *TiDB*.

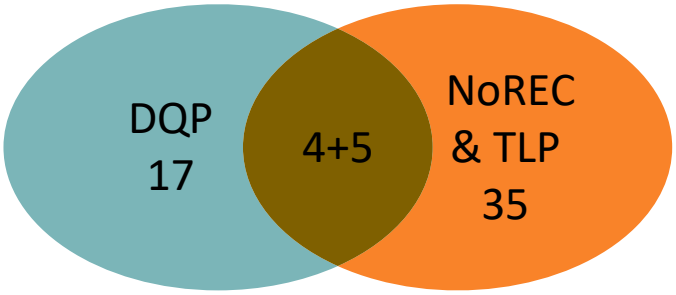


Figure 4.3: The number of bugs detected by oracles.

Code 4.7: Equivalent test cases of Code 4.5 for *NoREC* and *TLP*.

```
1 CREATE TABLE t0(c0 INT);
2 INSERT INTO t0(c0) VALUES(1);
3 CREATE INDEX i0 USING HASH ON t0(c0) INVISIBLE;
4 -------------------------------------NoREC-------------------------------------
5 SELECT COUNT(*) FROM t0 WHERE COALESCE(0.6) IN (t0.c0); -- {0}
6 SELECT SUM(count) FROM (SELECT (COALESCE(0.6) IN (t0.c0)) IS TRUE AS count
      FROM t0) as t; -- {0}
7 -------------------------------------TLP---------------------------------------
8 SELECT t0.c0 FROM t0; -- {1}
9 SELECT t0.c0 FROM t0 WHERE COALESCE(0.6) IN (t0.c0) UNION SELECT t0.c0 FROM t0
      WHERE NOT (COALESCE(0.6) IN (t0.c0)) UNION SELECT t0.c0 FROM t0 WHERE
      (COALESCE(0.6) IN (t0.c0)) IS NULL; -- {1}
```

## Q.4 Bug-finding Effectiveness

We compared *DQP* with two state-of-the-art oracles for finding logic bugs: Non-optimizing Reference Engine Construction (*NoREC*) [149], and Ternary Logic Partitioning (*TLP*) [150]. *NoREC* checks for inconsistent results of a predicate used in a query that the DBMS might optimize and one that is used in a query that is difficult to optimize. *TLP* expects a query and derives multiple more complex queries, each of which computes a partition of the result to check whether the combined partitions and the original query's results are equivalent. Both oracles are implemented in *SQLancer*. We did not consider other test oracles for finding logic bugs, such as Pivoted Query Synthesis (*PQS*) [151], which is not supported for the three evaluated DBMSs in *SQLancer*.

**Methodology.** We used the same methodology as prior works [84, 150, 149] to conduct a manual and best-effort analysis to identify the overlap and unique bugs found by *DQP*, *NoREC*, and *TLP*. It is difficult to distinguish whether two bug-inducing test cases found by different methods incur the same underlying bug [110].

We only collected the minimized test cases that are reported to developers assuming that each test case represented a unique bug. While we cannot completely rule out misclassifications that might be due to overlooking that a bug could be found by another query, we believe that the majority of cases were clear. In total, we collected all 41 logic bugs, of which 40 are reproducible, found by *NoREC* and *TLP* for *MySQL*, *MariaDB*, and *TiDB* from the public bug list,[20] and 21 logic bugs found by *DQP* in Table 4.2. Then, we used a bug-inducing test case for *DQP* to derive another test case by applying *NoREC* and *TLP* to the same database and the corresponding query, and vice versa.

**Results.** Figure 4.3 shows the number of bugs found by *DQP*, *NoREC*, and *TLP*. 17 of 21 logic bugs found *DQP* cannot be found by *NoREC* or *TLP*. Out of the 17 bugs, 10 bugs are because both the original query and the derived query result in correct or incorrect query plans. The other 7 bugs cannot be rewritten to equivalent test cases for *NoREC* or *TLP* due to the lack of necessary clauses for both oracles, such as `WHERE`. We also found that *DQP* cannot reproduce all 4 bugs found by *NoREC*, and 31 of 36 bugs found by *TLP*. The result shows that the bugs found by *DQP* rarely overlap with the bugs found by *NoREC* and *TLP*, suggesting that *DQP* is complementary to *NoREC* and *TLP*.

**Example.** Code 4.7 shows an example of rewriting the bug-inducing test case of Code 4.5 to equivalent test cases for *NoREC* and *TLP*—note that this is a mechanical transformation. For *NoREC*, the first query is the same as the original query in line 5 of Code 4.5, and the second query is derived from the first query by moving the predicate in `WHERE`. For *TLP*, the first query is generated by omitting the `WHERE` in the original query, and the second query is a union of three queries with different predicates in `WHERE`. Both oracles find a bug if both queries return inconsistent results. Without setting the option `use_invisible_indexes=on`, the buggy index `i0` is not considered in query optimization, so all queries checked by *NoREC* and *TLP* fail to use the buggy index to expose the bug.

*NoREC* and *TLP* cannot find 17 of 21 logic bugs found by *DQP*.

[20] `https://github.com/sqlancer/bugs/blob/96cbb856/bugs.json`

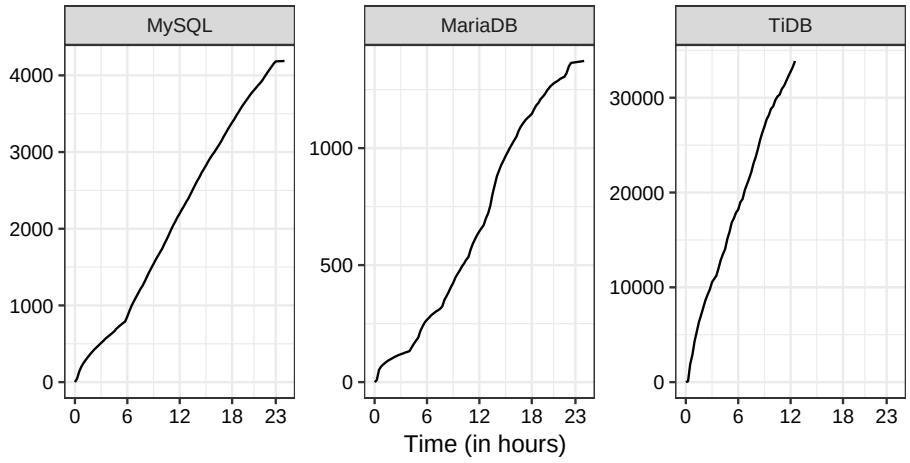


Figure 4.4: Average number of unique query plans covered by *DQP* in 24 hours and 10 runs.

## Q.5 Coverage

We evaluated how well *DQP* exercises query optimizers, which is the key component that we aimed to test. We considered various metrics to capture the notion of coverage. First, since *DQP* enforces different query plans of the same queries, we examined how comprehensively query plans are covered by *plan coverage*, which refers to the ratio of exercised unique query plans to the estimated number of all observable unique query plans. Then, we used query hints and system variables to enforce query plans, so we evaluated to what extent they affect query optimizers by *hint and variable coverage* and *join coverage*. Last, we also evaluated *code coverage*, a common metric to evaluate how much code is tested.

**Plan coverage.** We measured the ratio of unique query plans *DQP* covers for all observable unique query plans. A query plan represents an optimized query, and a higher number of unique query plans implies that more query optimization strategies are applied. A challenge with respect to measuring the number of unique query plans is that query plans include unstable auxiliary information, which usually differs for almost every query plan. We consider a query plan *structurally unique*, if the query plan is still unique after removing such information. To exclude this information, we omitted schema names (*e.g.*, column and table names), estimated cost (*e.g.*, cardinalities), and random identifiers (*e.g.*, line identifiers) in query plans. This method follows the practice reported in a prior work [85]. Another challenge is

the unknown upper bound of the number of query plans, as it is unclear how many possible combinations of operations for a query plan exist; note that the number is infinite in principle because an additional `JOIN` clause will typically result in a more complex query plan. As a best effort, we estimated the upper bound by combining all unique query plans covered by *DQP*, *NoREC*, and *TLP* across 24 hours and 10 runs, assuming the number of combined unique query plans as the upper bound. Suppose $D_i, N_i, T_i$ represent the set of unique query plans covered by three oracles respectively for a DBMS in run $i$, then the estimated number of the upper bound is $|\bigcup_{i=1}^{10}(D_i \cup N_i \cup T_i)|$.

**Results.** Figure 4.4 shows the average number of unique query plans covered by *SQLancer+DQP* across 10 runs in 24 hours. In summary, for *MySQL*, *MariaDB*, and *TiDB*, the estimated upper bounds of plan coverage are 27156, 7553, and 253947, and *SQLancer+DQP* covers 15.42% (4187.5), 18.17% (1372.4), and 13.34% (33876.6) on average for each run. Due to several crash bugs found by *SQLancer+DQP* in *TiDB*, all 10 runs exited in around 12 hours. Although in a shorter time period, *TiDB* covered the most unique query plans. A possible reason is that *TiDB* includes richer elements in query plans than others. For example, *TiDB* is a distributed DBMS, and indicates the execution node of each operator in query plans, while other DBMSs do not have similar information. Although covering less than average 20% plan coverage for three DBMSs, *DQP* has much higher plan coverage than *NoREC* and *TLP*, both of which achieved less than 1% average plan coverage across 10 runs. This is expected, because *NoREC* and *TLP* do not optimize for query plan coverage. We note that the overall low coverage can be explained by diverse query plans being explored across runs—less than 50% overlapped for *SQLancer+DQP* across 10 runs, and thus the sum of average coverage numbers for each run of *DQP*, *NoREC*, and *TLP* is not close to 100%. The reason may be randomly generated databases and queries, which typically differ across runs. We cannot compare the plan coverage by *DQP* and *TQS*, because *TQS*'s source code is unavailable, and no query plan coverage numbers were reported in its paper. For all three DBMSs, *SQLancer+DQP* covers thousands of unique query plans, which shows that *SQLancer+DQP* is significant in testing query optimization.

Table 4.3: The number of query hints or system variables that affect the three categories of query optimizations.

| DBMS | Join | Index | Table |
|---|---|---|---|
| *MySQL* | 14 | 26 | 18 |
| *MariaDB* | 18 | 5 | 14 |
| *TiDB* | 10 | 4 | 8 |
| **Sum:** | 42 | 35 | 40 |

**Hint and variable coverage.** We identified three categories of query optimizations that can be affected by query hints or system variables. Table 4.3 shows the number of query hints or systems variables of each category. Although query plans and query optimizations are DBMS-specific and not directly comparable, we found that the three DBMSs provide hints or variables to affect common categories of optimizations: *Join*, the algorithms and orders of joining two tables; *Index*, the algorithms and applicable range for indexes; and *Table*, the strategies to write and read tables, such as table caching for repeated queries and full table scan for small tables. As a concrete example, to affect join optimizations, query hint `HASH_JOIN` for *MySQL* and *TiDB* is used to enforce using the hash algorithm, and variable `hash_join_cardinality` for *MariaDB* is used to decide whether using historical cardinality statistics for optimizing hash joins. Although *MariaDB* is derived from *MySQL*, both have a different number of query hints and system variables. For example, for the category *Index*, *MySQL* has two query hints and four system variables tailored for affecting the algorithm of index merge, which is an optimization for using indexes to merge results from multiple scans, while *MariaDB* does not have a similar hint or variable to affect it. *TiDB* has additional optimizations of the category *Table* for switching storage engines, which can be affected by query hints. For example, the query hint `READ_FROM_TIFLASH` is used to enforce reading tables from TiFlash, a storage engine of *TiDB*. This optimization is specific to *TiDB*, while *MySQL* and *MariaDB* can only specify storage engines when creating tables. All three DBMSs provide support for query hints and system variables, influencing the same three categories of query optimizations. However, they impact specific query optimizations to each DBMS.

**Join coverage.** Since *DQP* aims to find bugs in join optimization, we also evaluated how many join operators in query plans *SQLancer+DQP* covered by setting query hints and system variables across 10 runs in 24 hours. We examined whether our query plans cover the join operators illustrated in the documents of *MySQL*,[21] *MariaDB*,[22] and *TiDB*.[23] In total, both *MySQL* and *MariaDB* have 12 join operators, and *TiDB* has 3 join operators. *SQLancer+DQP* covered 7 of 12 join operators for *MySQL* and *MariaDB*, and all 3 join operators for *TiDB*. Both *MySQL* and *MariaDB* do not cover four join operators: `fulltext`, which is used for full-text indexes; `index_merge`, which is used for union or intersection expressions; `unique_subquery`, and `index_subquery`, both of which are used for subqueries. *MySQL* and *MariaDB* provide a specific query hint `INDEX_MERGE` to enable `index_merge`, but the join operator is not covered by our implementation of *DQP*, since it requires specific expressions in queries. We have not found any hint or variable that directly enforces the other three join operators. Additionally, *MySQL* does not cover the join operator `system`, which is used for system tables, and *MariaDB* does not cover `ref_or_null`, which is used for index lookup with null values during joining. The reason for the two not-covered operators may be the randomness of test case generators, as either operator is covered by the other DBMS.

**Code coverage.** While we were primarily interested in the number of covered unique query plans, code coverage is a common metric for evaluating how much a system might be tested. We used gcov,[24] a coverage tool for C/C++ language, to collect line coverage of *MySQL* and *MariaDB*, and used cover,[25] a coverage tool for the Go language, to collect the statement coverage of *TiDB*. Line coverage is the default metric for gcov, and statement coverage is the default metric for cover. We ran *SQLancer+DQP* for 24 hours and 10 runs simultaneously. Due to resource limitations, each target DBMS ran one instance, and we measured their sum line and statement coverage across 10 runs. For the same reason that *TQS*'s source code is unavailable, we cannot compare the code coverage between *DQP* and *TQS*.

[21] `https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_system`
[22] `https://mariadb.com/kb/en/explain/#type-column`
[23] `https://docs.pingcap.com/tidb/stable/explain-joins`
[24] `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`
[25] `https://go.dev/testing/coverage/`

Since *DQP* finds bugs in JOIN optimization, we measured only the code coverage of query optimization. Specifically, we measured the code in the folder *sql* for *MySQL* and *MariaDB*, and in the folder *planner* for *TiDB*. The results show that *SQLancer+DQP* covered 22.2% and 27.7% line coverage for *MySQL* and *MariaDB*, and 36.1% statement coverage for *TiDB*. The coverage appears to be low, as less than 50% coverage for all DBMSs. However, this is expected because we cannot enumerate all possible query plans.

*DQP* covers thousands of unique query plans and more than half join operators for *MySQL*, MariDB, and *TiDB* in 24 hours.

## 4.6 Discussion

We discuss some key characteristics of *DQP*, as well as the evaluation of *TQS*.

**Bug diversity.** Our found bugs can affect a variety of different queries. The bugs are typically due to incorrect optimizations, such as the incorrect index optimization as shown in Code 4.5, and the incorrect join optimization as shown in Code 4.6. These buggy optimizations can affect other queries as well, not only the bug-inducing test cases that make use of specific query hints or values of system variables. For example, considering Code 4.5, if the index `i0` is not created with `INVISIBLE` by `CREATE INDEX i0 USING HASH ON t0(c0)`, the first query `SELECT t0.c0 FROM t0 WHERE COALESCE(0.6)IN (t0.c0)` returns the incorrect result 1 without setting up any query hint or system variables.

**Path to adoption.** We believe that a simple testing approach has the potential to be widely adopted. From a conceptual perspective, *DQP* is a general black-box method that compares the results of different query plans, which is an easy-to-understand method. It is not necessary to instrument code for tracing internal execution information or understand how the result is computed. From an implementation perspective, *DQP* is easy to implement as we implemented *DQP* in less than 100 lines of Java code per DBMS. From an integration perspective, *DQP* can be paired with existing available databases and query generators, or test suites.

From an applicability perspective, *DQP* can test a significant number of DBMSs, as 8 out of 10 most popular relational DBMSs[26] support controlling query optimization by users. The remaining two DBMSs are Microsoft Access and Snowflake, for which we have not found any document that explicitly explains how to manually control query optimization. Considering these features of *DQP*, we argue that *DQP* can be widely adopted.

**Method contribution and novelty.** The core contribution of this paper is that we demonstrated that the simple and easy-to-understand testing *DQP* approach shows the same level of bug-finding effectiveness as the more complex *TQS* approach. The authors of *TQS* mentioned the comparison of queries with query hints in Section 5.3 of the *TQS* paper by disabling the derivation of ground-truth results. Some systems in practice, such as DuckDB, already use similar techniques as well in their own testing framework. DuckDB does this by running both an unoptimized and optimized version of a query, and checking consistency of the results. It controls the optimizations by a specific statement `PRAGMA enable_verification`, which is customized for DuckDB.[27] The core contribution of this work is not the novelty of *DQP*. Our core contribution is the insight that such a simple approach achieves the same level of bug-finding efficiency as the sophisticated method *TQS*. In general, in a testing context, we believe that simple, practical approaches provide significant benefits over complex, but conceptually appealing ones.

**The importance of *TQS*.** *TQS* is the first approach for testing logic bugs in join optimizations, and thus demonstrated the severity of the problem. Importantly, *TQS* provides a new paradigm for finding logic bugs in DBMSs. In this work, we showed that a simple method achieves the same level of bug-finding efficiency as *TQS*. Nevertheless, *TQS* can find bugs that can not be found by *DQP*. Bug #99273 in Table 4.1 was found without showing discrepancies across executing different query plans of the same query. Although it is unclear how *TQS* derives the ground-truth result for this query that does not have a **`JOIN`**, *DQP* can not find this bug.

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

[27] `https://duckdb.org/dev/sqllogictest/intro#query-verification`

**Inconsistent bug number of *TQS*.** We identified 15 unique bugs in the public bug list from *TQS*, while the authors of *TQS* claimed to have found 92 bugs. Based on our study, we suspect that the authors used confusing terminology in the paper, by refering to *bug-inducing test cases* as "bugs" and *unique bugs* as "bug kinds", which we clarified in this work. Additionally, we acknowledge that bug deduplication is an open problem [33], and we also observed duplicate issues being counted as bugs in other work.

**Unavailable *TQS* source code.** Two reasons exist that prevent us from comparing with *TQS* in our study and evaluation. First, the authors of *TQS* have not released its source code. We sent two emails to all authors of *TQS* requesting the source code, but the authors replied that the source code was not ready for release: "*I'm currently working on a follow-up project that builds upon the research I presented. As a result, I'm in the process of refining and enhancing the codebase for both projects. Once this work is complete, I plan to make the source code available as open source or share it with colleagues who express an interest.*" We also noticed that the authors published another tool demonstration paper [168] that includes the implementation of *TQS*. Unfortunately, after carefully checking and debugging its source code,[28] we found that the repository lacks the core approach implementation of *TQS*, as also observed by another interested party.[29] Second, it is challenging to re-implement *TQS* as it consists of complex steps, with important details not being described in the paper. For example, the authors claimed "*We directly use these data-driven schema normalization methods to generate our testing database schema*", but it is not clear what concrete method they used to split the wide table and what sub-tables are generated. *TQS* adopts Abstract Syntax Tree (AST)-based random query generation, which is implemented "*similarly to RAGS and SQLSmith*", but it is insufficient to know what queries it can generate, such as what expressions are generated and the maximum depth of the AST.

**Threats to Validity.** Our evaluation results face potential threats to validity. A major concern is that *TQS* source code and *TQS*'s bug reports on PolarDB are not

[28] `https://github.com/xiutangzju/dlbd/tree/b85b1f`
[29] `https://github.com/xiutangzju/dlbd/issues/1`

available. To alleviate this risk, we communicated with the authors of *TQS*, which told us that *TQS*'s source code has not been ready for release yet. Because it is challenging to re-implement *TQS*, we extracted the public bug list and conducted a rigorous study described in Section 4.2. From the practical perspective, we compared the bugs found by *TQS* and *DQP* to evaluate their bug-finding capabilities. From the theoretic perspective, we discussed their conceptual differences in this section. Another concern is the correctness of our implementation. To mitigate this risk, *DQP* was built on a popular DBMS testing framework *SQLancer*, and we make the source code publicly available. The last concern is the reliability of the results we presented. To make sure our found bugs are real bugs, we reported each found bug to developers and annotated the bug status according to developers' replies. We also make all bug reports public.

## 4.7 Conclusion

In this chapter, we have studied the state-of-the-art testing approach *TQS*, and have proposed a simple, yet effective alternative approach *DQP*. The core idea of *DQP* is comparing the consistency across the executions of different query plans of the same query, which we derive by adding query hints or setting system variables. Compared to *TQS*, *DQP* only needs to compare results instead of constructing multiple graphs and tables for deriving ground-truth results, and supports finding bugs in more query optimizations instead of only in equijoin optimizations. Our evaluation has demonstrated that *DQP* can find 14 of the 15 unique bugs and all 10 join-related bugs found by *TQS*. Additionally, *DQP* has found 26 previously unknown and unique bugs in *MySQL*, *MariaDB*, and *TiDB*, which were tested by *TQS*. Our core contribution is the insight that a simple and easy-to-understand approach is similarly effective as the more sophisticated method *TQS*. *DQP* requires little implementation effort, is compatible with any test case generation methods, is similarly efficient as *TQS*, and is a black-box testing method. We encourage DBMS developers to use *DQP* in practice, as a cost-efficient way to find critical bugs in DBMSs.

# Chapter 5

# Query Plan Guidance

To realize a fully automated testing approach, test oracles, such as *CERT* and *DQP*, are paired with a test case generation technique; a test case refers to a database state and a query on which the test oracle can be applied. To make test case generation efficient, in this chapter, we propose the concept of *Query Plan Guidance* (*QPG*) for guiding automated testing towards "interesting" test cases. *QPG* has been published in the 45th International Conference on Software Engineering (ICSE'23) [85] and was awarded the ACM SIGSOFT Distinguished Paper Award.

## 5.1 Introduction

Logic bugs, which refer to incorrect results returned by DBMSs, are a particularly challenging category of bugs to find as they silently compute an incorrect result—unlike, for example, crash bugs [205, 219], which cause the process to be terminated. Consider Code 5.1, where the `SELECT` statement triggers a logic bug that causes the returned result to unexpectedly contain a record, while it should be empty. It is difficult to find logic bugs since the bugs silently compute an incorrect result—unlike crash bugs, which terminate the process—and it is also challenging to obtain the right answer to validate the results. Finding such bugs requires a so-called test oracle, which validates the DBMS' result. Recently, effective test oracles [150, 149, 151] have been proposed that brought validating the results of such queries within reach.

Besides a test oracle, automatically finding logic bugs requires a test case generation method. For finding logic bugs in DBMSs, a test case refers to a database state

and a query on which the test oracle can be applied. Recall that test case generation techniques face two main challenges. First, diverse test cases should be generated that stress various parts of the DBMS to increase the chance of finding bugs in them. Second, the test cases should be valid both syntactically and semantically while also corresponding to the structure imposed by the test oracle.

In this work, we propose *Query Plan Guidance* (*QPG*), a concept that utilizes query plans to guide the test-case generation process towards diverse test cases. A query plan is a tree of operations that describes how an SQL statement is executed by a DBMS. It is readily provided by DBMSs—users can typically obtain a textual representation using an `EXPLAIN` SQL statement—and is typically inspected by DBMS users for tuning the performance of queries. Our insight is that a query plan provides a compact and high-level summary of how a query is executed, therefore, covering more unique query plans increases the likelihood of finding logic bugs. Consider Code 5.1, which illustrates two scenarios of executing test cases with SQLite. In the first scenario, the `CREATE INDEX` statement highlighted in red is omitted, causing the `SELECT` statement to return an empty result. This result is expected, since column `c` in table `t2` has no data, and the join condition `c=3` is false. In the second scenario, the `CREATE INDEX` statement is executed, which causes SQLite to unexpectedly fetch the row {|1|2|}. An index is an auxiliary data structure used by queries [60], which should not have any semantic effect. While in both scenarios, the same query is executed, the query plans shown below the test cases differ due to the two different database states. The left query plan for the correct execution indicates that the records from table `t2` are read sequentially (`SCAN t2`). In contrast, the right query plan indicates that the DBMS used the index to read the data (`SCAN t2 USING COVERING INDEX i0`), which was incorrect. Besides indexes, various other factors can influence query plans (*e.g.*, data characteristics).

To investigate the potential of using query plan coverage as guidance, we studied the query plan distribution of the SQL statements in the present 499 bugs found by *SQLancer*. Our results show that, for the test cases of previously found bugs, the query plan distribution of the SQL statements is sparse, so the query plan is diverse and is possible to be used as a signal for guiding. We also found that the average query plan length of the queries in previously found bugs is 2, which indicates a potential to find more bugs in unexplored complex query plans.

Code 5.1: A bug found by *QPG* in SQLite due to incorrect use of an index in combination with a JOIN. Given the same SELECT, the left query plan is produced if no index is present, while the right one uses the index.

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

To generate valid queries that correspond to the oracles' constraints, we propose mutating the database state rather than the queries. Specifically, we re-use the existing random grammar-based generation approach of *SQLancer* [150] to generate the queries. However, we record all seen query plans for a given database state and mutate this state when no new query plans are observed, indicating that the current database state's potential for enabling unobserved query plans has been saturated. We modeled the decision-making process for selecting the most promising mutation—an SQL statement that modifies the database state—as a multi-armed bandit problem and assigned a high priority to the SQL statement that results in the most new query plans across all executions. The multi-armed bandit problem is a model in which a fixed limited set of resources has to be allocated between competing choices in a way that maximizes the expected gain [15].

We implemented *QPG* in *SQLancer* and evaluated it on SQLite, TiDB, and CockroachDB. We found 53 unique, previously unknown bugs, all of which have been acknowledged by the developers. Of these, 46 have already been fixed. Three bugs in SQLite had been hidden for more than six years before we found them, despite the extensive existing testing efforts by the authors of *SQLancer* and *SQLRight*, demonstrating the practical need for a more efficient test case generation approach.

Table 5.1: Subjects for the query plan study.

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

To trigger many of the bugs, complex query plans are required, indicated by the average length of query plans being 2.47× longer than that of the previously found bugs. In terms of efficiency, our *QPG*-based implementation covers 4.85–408.48× more unique query plans than *SQLancer* and *SQLRight* in 24 hours. In summary, our results demonstrate that *QPG* is efficient in exploring more unique and complex query plans, which contribute to finding logic bugs. The developers from SQLite and TiDB showed interest in our method *QPG*, and TiDB invited me to give a talk about more details.

Overall, we make the following contributions:

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

## 5.2 Query Plan Study

To investigate the potential of using query plans as guidance, we studied the uniqueness and complexity of query plans of the queries in previously found bugs.

Table 5.2: Query plans of the queries in previously-found bugs. Length indicates the average number of operations in a query plan.

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

We hypothesized that we would see a wide variety of query plans, suggesting that a bug-finding technique optimized for exploring more unique query plans might be effective.

**Subjects.** We chose the public bug reports from *SQLancer* as our subjects. *SQLancer* provides a public list[1] including all found bugs and corresponding test cases for 499 bug reports across 9 DBMSs. We excluded 4 bugs found in the DBMS *TDEngine*, as this DBMS does not expose query plans. The query plan of a given query can vary over versions; thus, to obtain accurate query plans, we chose the most relevant release versions when the corresponding bugs were found. The details of the chosen DBMSs are shown in Table 5.1.

**Obtaining query plans.** For all 495 bug-inducing test cases, we instrumented all queries (*i.e.*, `SELECT` statements) by using `EXPLAIN` statements as listed in Table 5.1. By executing the test cases using the command-line interface of each DBMS, we could obtain corresponding query plans. Depending on the DBMS, query plans might include various additional auxiliary information. We identified three such types. One type is the estimated cost (*e.g.*, in *PostgreSQL*), which differs for almost every query. The second type is expressions in `WHERE` clauses, which are included in the query plans of some DBMSs (*e.g.*, *CockroachDB*). The third type is random

[1] `https://github.com/sqlancer/bugs`

identifiers, which are used to distinguish operations in a query plan (*e.g.*, *MariaDB* and *MySQL*). To exclude such auxiliary information, we accordingly adjusted the parameters of the `EXPLAIN` statements, as shown in Table 5.1. Lastly, we removed the names of tables, views, and indexes of the obtained query plans to distinguish query plans based on their structure only. This was based on the intuition that two query plans with the same execution logic, but different table names, would be processed similarly by the DBMSs (*e.g.*, `SCAN t1`, and `SCAN t2`).

**Uniqueness analysis.** Table 5.2 shows the query plan distribution. In total, we obtained 316 query plans, of which 57.28% were unique. The number of query plans is lower than that of test cases because 1) not all test cases have queries and 2) some queries that previously exposed bugs were rejected by subsequent versions of the DBMSs. The minimal percentage of unique query plans is 30.51% in DuckDB. The maximum one is 100.00% in *MariaDB*, due to a low number of test cases. Overall, for the queries in previously found bugs, the variety of different query plans indicates that covering a wider variety of query plans might increase the likelihood of discovering bugs.

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

**Complexity analysis.** We examined the complexity of the query plans of the queries in previously found bugs. A query plan with many operations is due to a complex database state or query. For instance, in *SQLite*, a query plan that retrieves data from two tables requires at least three operations: `SCAN table t0`, `SCAN table t1`, and `MERGE results`, which is more complex than `SCAN table t0` alone. As shown in the *Length* column of Table 5.2, the average number of operations per query plan is 2.59, which illustrates that the majority of bug-related query plans are compact. We further found that the most frequent query plan across eight DBMSs is `SCAN table t0`, which represents a sequential scan on a single table, without using an index. For example, in SQLite, 26 of 118 query plans consist of a single table scan. This demonstrates that the query plans for the previously found bugs are simple. While this could indicate that, compact and simple query plans are sufficient to trigger

these previously found bugs—as suggested by the small-scope hypothesis [5]—it could also be that existing approaches have focused their testing on simple queries and database states. We speculate that covering more complex query plans might increase the likelihood of discovering bugs.

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

## 5.3 Approach

To efficiently detect logic bugs in DBMSs, we propose to mutate databases with *Query Plan Guidance (QPG)* towards more unique and increasingly complex database states. Our insight is that the internal execution logic of the DBMS for a given query is reflected by its query plan and, therefore, covering more unique query plans increases the likelihood of finding logic bugs. Compared with naive random generation, our method gradually mutates database states enabling subsequent queries to cover more unique and complex query plans. We chose to mutate database states rather than queries, since test oracles have various constraints on queries, which are difficult to meet using mutational approaches [106]. Compared with other coverage-based grey-box testing tools for DBMSs, such as *Squirrel* [219] and *SQLRight* [106], we consider our method as black-box testing, as *QPG* requires no access to the source code of the DBMS and uses information readily provided by mature DBMSs. Thus, the technique can also be applied to commercial closed-source DBMSs.

**System overview.** Figure 5.1 shows an overview of our *QPG* realization based on Code 5.1. Given an initial database state at ①, *QPG* generates a random SQL query at ② and executes it on the database to validate the query's result using the test oracle. If the oracle indicates a bug, *QPG* outputs a bug report and restarts the testing process. Otherwise, it records the query plan and appends it to the query plan pool at ③. Typically, the execution continues at ② with the same database state. However, if no new unique query plan has been observed after a fixed number of iterations, *QPG* mutates the database state at ④ by applying a mutation operator

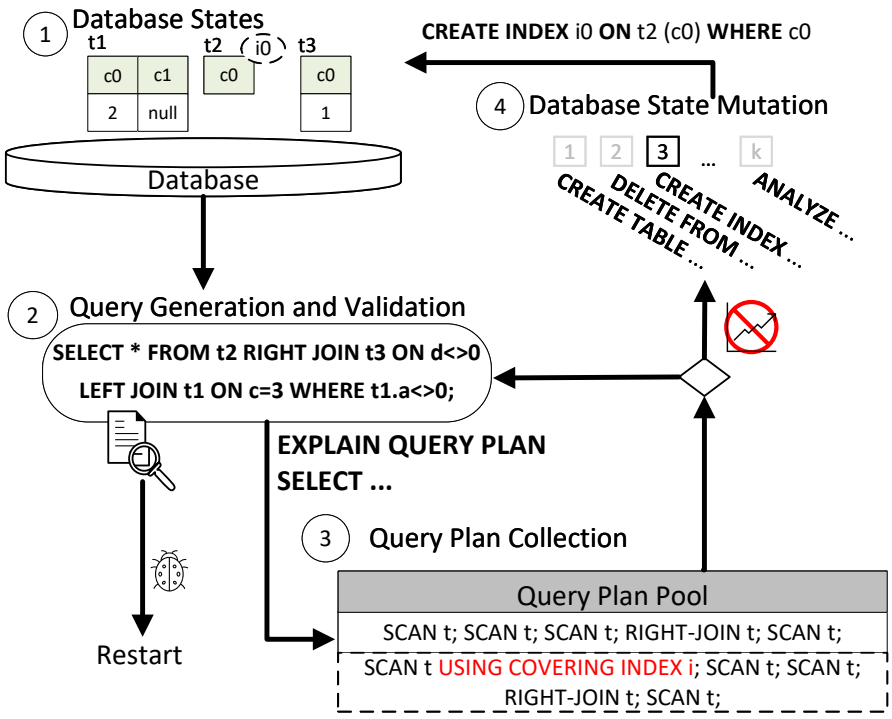


Figure 5.1: Overview of *QPG*. The dashed lines refer to the data affected by ④ in the next iteration.

to the current database state to create a new one, assuming that this new state will subsequently lead to new unique query plans being explored.

### 5.3.1 Database States

The initial database state can be either randomly generated or manually given. In our implementation, we generate it by randomly executing DDL and DML statements. To avoid empty database states, we execute `CREATE TABLE` statements first. For example, to create the initial database state in Figure 5.1, we execute lines 1–5 in Code 5.1. We do not directly manipulate database files, since they are highly structured [81], and any unexpected byte may incur an error that would impede the testing process.

### 5.3.2 Query Generation and Validation

**Query generation.** We generate queries whose results we subsequently automatically validate to find bugs. The generated queries must comply with two main constraints. First, queries must be semantically valid with respect to the database state. For example, they must reference only existing tables and views. Second, they must adhere to the constraints imposed by the test oracles. For example, the *NoREC* test oracle requires a `WHERE` clause, but forbids other clauses (*e.g.*, `HAVING` or `GROUP BY`). To address this, we adopt *SQLancer*'s rule-based random generation approach that generates queries based on the SQL dialects' grammar adhering to the imposed constraints. Many query generation approaches have been proposed [11, 24, 89, 117, 139, 157, 165], and our method can, in principle, be paired with any of these query generation methods.

**Validation.** We use the state-of-the-art logic-bug oracles NoREC [149] and TLP [150] to validate the queries' results. Both are metamorphic testing approaches [32] and, given a query, derive another query whose result set is used to validate the original query's result. In Figure 5.1, given the three tables and the test oracle, we generate the query `SELECT * FROM t2 RIGHT JOIN t3 ON d<>0 LEFT JOIN t1 ON c=3 WHERE t1.a<>0`. Since the test oracle indicates that the empty result returned is correct, execution continues at ③. If the test oracle indicates a bug, we output the bug report and restart the testing process.

### 5.3.3 Query Plan Collection

We collect query plans by instrumenting queries using the `EXPLAIN` statement, which is the same approach as presented in Section 5.2. In Figure 5.1, the statement to obtain the query plan is `EXPLAIN QUERY PLAN SELECT * FROM t2 RIGHT JOIN t3 ON d<>0 LEFT JOIN t1 ON c=3 WHERE t1.a<>0`. We obtain the query plan (shown in the left part of lines 12–17 in Code 5.1), and remove table and index names.

We insert query plans into the query plan pool in which we store unique query plans. The pool is implemented as a hash table in which the keys are query plans, and the values are the corresponding query strings. Given a query plan, we check whether the query plan exists in the pool, and insert it if not. In Figure 5.1, the pool

is initially empty, so we insert the query plan (the first line at ③). If no new query plan is inserted into the pool for a fixed number of queries, we invoke ④ aiming to cause the DBMS to explore more unique query plans. Otherwise, we continue to test the DBMS using the same database state at ②. The number is configurable, and a higher number indicates that we test the DBMS using more queries on a single database state, while a lower one means that we test the DBMS using more database states. The number is set to 1,000 by default, which we determined to work well empirically.

### 5.3.4 Database State Mutation

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

As mutation operators, we consider both the same DDL and DML statements used for generating the initial database state, such as **`CREATE TABLE`**, **`CREATE INDEX`**, and **`ANALYZE`**. A key challenge is to apply promising mutations that likely result in queries triggering new query plans. We model this task as the Multi-Armed Bandit (MAB) problem [57, 15], which is a popular and efficient method that has been used in various fuzzing works [213, 174, 196, 145]. In MAB, a fixed limited set of resources has to be allocated between competing choices to maximize the expected gain. In our scenario, given a limited computational resource, we choose the SQL statements (choices) to mutate database states to maximize the number of covered unique query plans (gain).

To maximize the expected gain, an automated agent attempts to acquire new knowledge (called "exploration") and optimizes its decisions based on existing knowledge (called "exploitation"). In our problem scenario, given the knowledge that the gains of only some mutation operators have been observed, we consider selecting the next mutation operator from either explored or unexplored mutation operators. Making the decision based on explored mutation operators (exploitation) tends to increase the gain, but may miss potentially higher gain from unexplored mutation operators. Many algorithms have been proposed to strike a balance between exploration and exploitation. We adopt the classic episode greedy algorithm [96],

which chooses the operator with the highest known gain with a certain probability and a random one otherwise.

Our algorithm works as follows. At $t$ times when database state mutation ④ is invoked, we choose one mutation operator followed by Equation 5.1. $k$ is the number of candidate mutation operators. $\hat{\mu}_{i(t)}$ is the known gain of the mutation operator $i$ at time $t$. $\epsilon$ is a fixed probability ranging from 0 to 1; its default value is 0.7, which we determined to work well empirically. With $(1-\epsilon)$ probability, we choose the operator that has the maximum known gain and randomly choose one otherwise.

$$j(t) = \begin{cases} \arg\max_{i=1\ldots k} \left(\hat{\mu}_{i(t)}\right) & (1-\epsilon) \\ random(k) & (\epsilon) \end{cases} \tag{5.1}$$

**Encoding known gain $\hat{\mu}_i$.** $\hat{\mu}_i$ is measured as weighted average gain—different from the standard algorithm, which uses an unweighted average—across all iterations where $i$ was chosen. A DBMS is a stateful system. The database state depends not only on the last applied mutation operator, but also on the previous database state. Applying the same mutation operator on changing database states creates different database states, so the gain of a mutation operator across iterations is not independent and identically distributed. For the same mutation operator, the gain in the last iteration is closer to the real gain in the last database state. To approximate the known gain, we use a weight average number in which the latter gain has a higher weight than the former gain. Equation 5.2 is our equation for updating $\hat{\mu}_i$ in each iteration. $Q$ is the gain for the last time $i$ was chosen. $w$ is the weight of $Q$, which is a constant ranging from 0 to 1; its default value is 0.25, which we determined to work well empirically. Independent from the number of iterations, the prior gains only take up $(1-w)$ weight for $\hat{\mu}_i$. For example, given $w = 0.1$, $\hat{\mu}_{i(999)} = 0.1$, $Q = 2$ for the $1,000_{th}$ iteration, the $\hat{\mu}_{i(1000)} = 0.1 + (2-0.1) * 0.1 = 0.29$, which is much higher than the unweighted average number $0.1 + (2-0.1)/1000 = 0.1019$ and closer to the $Q$. For efficiency, all parallel testing processes share the same $\hat{\mu}_i$.

$$\hat{\mu}_{i(t+1)} = \hat{\mu}_{i(t)} + (Q - \hat{\mu}_{i(t)}) * w \tag{5.2}$$

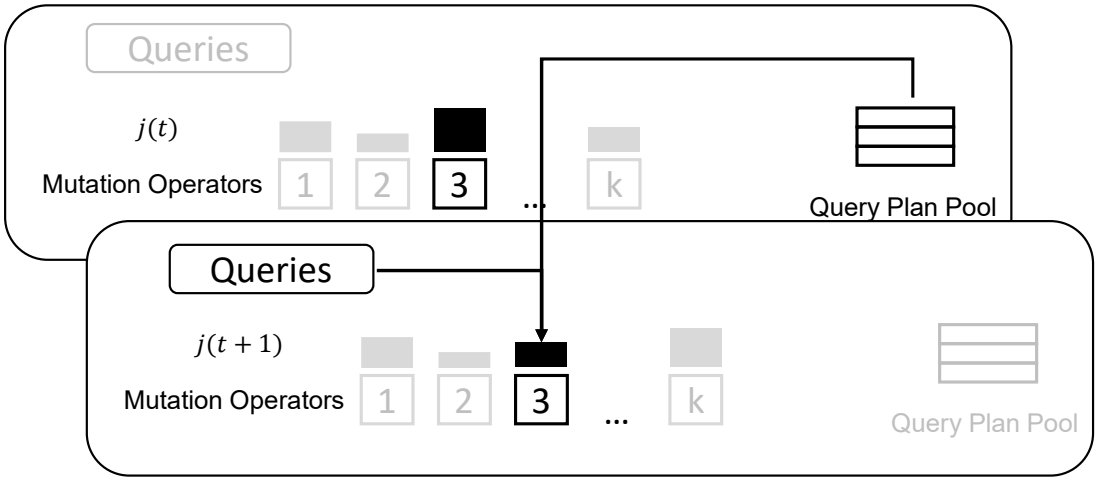


Figure 5.2: The workflow of measuring the known gain at ④.

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

Figure 5.2 shows the workflow of measuring the known gain $\hat{\mu}_i$ at ④. If the mutation operator 3 is chosen in iteration $t$ due to its highest $j(t)$, we update $\hat{\mu}_3$ in the next iteration $t + 1$ with the queries that are generated after iteration $t$ and the queries of the query plan pool in iteration $t$. Following that, we calculate $j(t + 1)$ and choose the mutation operator $k$.

In Figure 5.1, we apply `CREATE INDEX i0 ON t2 (c)WHERE c=3`, which creates an

index `i0` at ①. Suppose we generate the same query at ②, then we observe the new query plan shown on the right in lines 12–17 in Code 5.1 and insert it into the query plan pool. As a result, the bug is exposed at ②.

Lastly, we clear the database state after a fixed number of tested queries aiming to maximize the number of covered unique query plans. In general, by gradually mutating the same database state, we explore more unique and increasingly complex database states. However, the current database state may limit the possible state space to mutate into, which is why we clear the database state and restart the testing process after a fixed number of tested queries. The number is configurable and is set to a reasonable default value of 1,000,000, which we found to work well in our experiments (see Table 5.4).

### 5.3.5 Implementation

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

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

## 5.4 Evaluation

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

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

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

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

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

**Tested DBMSs.** We tested *SQLite*, *TiDB*, and *CockroachDB*. *SQLite* is the most popular embedded DBMS—embedded DBMSs are built together with and run in the same process as the application—and is used in every IOS and Android smartphone [185]. *TiDB* and *CockroachDB* are popular enterprise-class DBMSs, and their open versions on Github are highly popular as they have been starred more than 34.8k and 27.8k times. They are widely used and have thus also been used in other DBMS testing works [106, 151, 150, 219]. We did not consider other popular DBMSs due to various reasons. For example, for *MySQL* and closed-source DBMSs, bug fixes can be validated only after new releases; until then, it is difficult to identify new bugs, as already-known bugs might be repeatedly triggered. Furthermore, for some DBMSs, such as *MySQL*, many previously reported bugs remain unfixed, impeding the testing process, which was also noted in prior work [150]. As a black-box method, *QPG* supports any DBMS, regardless of what programming languages it is written in; *SQLite* is written in C, while *TiDB* and *CockroachDB* are written in Go. For Q1, Q2, and Q4, we used the latest available development versions (*SQLite*: 3.39.0, *TiDB*: 6.3.0, *CockroachDB*: 23.1). For Q3, to make a fair comparison, we chose the historical versions of DBMSs that all tools have tested and can find bugs in (*SQLite*: 3.36.0, *TiDB*: 4.0.15, and *CockroachDB*: 21.2.2).

Table 5.3: The number of new bugs found by *SQLancer+QPG*.

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

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

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

## Q.1 New Bugs

We ran *SQLancer+QPG* for approximately two months—during which we also implemented the approach—aiming to find bugs. To better demonstrate the underlying issue for each bug found, we minimized the test case both using C-Reduce[146] and manually. After reporting the bugs to the developers, we suspended the testing process until the bug was fixed to avoid duplicate reports whenever possible; when bugs were not fixed within a timespan of weeks, we reported multiple bugs that we suspected to be unique. The bugs in *SQLite* were usually fixed within 24 hours, while the bugs in *TiDB* and *CockroachDB* were usually fixed within several weeks. As a result, we focused on testing *SQLite*. We used NoREC [149] and TLP [150], which are the state-of-the-art oracles supported by both *SQLancer* and *SQLRight*.

Code 5.2: A bug in the RIGHT JOIN feature of *SQLite*.

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

**Bugs overview.** Table 5.3 shows the number of unique, previously unknown bugs found by *SQLancer+QPG*. We found 53 bugs in total, all of which have been confirmed. Of these, 46 have already been fixed. Although *SQLancer* had been extensively applied to these DBMSs, we were still able to find these bugs with the help of *QPG*. Of the 53 bugs, 28 were logic bugs found by the test oracles *TLP* and *NoREC*, and 25 bugs were associated with crashes or internal errors. This demonstrates that the complex database states generated by *QPG* are beneficial not only to finding logic bugs, but also to other kinds of bugs. Although *CockroachDB* used the *TLP* oracle in their Continuous Integration (CI) process,[3] we still found 16 previously unknown bugs using *QPG*. For the new features in *SQLite*, *QPG* found 13 bugs in **`RIGHT JOIN`**, 2 bugs in **`FULL JOIN`**, and no bug in **`STRICT`**. We give two examples of found bugs as follows.

**Example 1: a bug in the `RIGHT JOIN` feature.** Code 5.2 shows a test case exposing a logic bug that we found in *SQLite*. The **`SELECT`** statement incorrectly returns an empty result, because of an incorrect optimization of `ISNULL` when used with a **`RIGHT JOIN`**. The query plan of the **`SELECT`** statement is six operations long: scanning all tables once in four operations, and joining table `t2` with another scan on `t2` in two operations. The query plan is relatively long, because joining tables typically involves multiple operations. 13 bugs in *SQLite* were in the **`RIGHT JOIN`** feature, in which *QPG* generates more complex database states to find bugs.

**Example 2: a bug in JSON feature.** Code 5.3 is another logic bug that had existed in *SQLite* since July 23, 2016. The **`SELECT`** statement incorrectly returns an empty result because of an incorrect optimization of the `json_quote` function in the

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

Code 5.3: A bug in `json_quote` function of *SQLite*.

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

Table 5.4: Query Plans of the queries in newly found bugs.

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

context of a `VIEW`, which is necessary to find the bug. The bug cannot be found if the second line is replaced by `CREATE TABLE v1(b)AS SELECT json(1)`. In *SQLite*, we found three bugs that had been hidden for more than six years, and *SQLancer+QPG* is the first tool to find them despite extensive efforts by the authors of *SQLancer* and *SQLRight*.

**The uniqueness and complexity of query plans.** To better understand how and whether *QPG* enables exploring a variety of query plans, we analyzed the query plans of the queries in Table 5.3. In total, we obtained 69 query plans, of which 63.77% are unique. This further demonstrates the diversity of query plans. On average, the length of query plans of queries was 6.35. In comparison with Table 5.2, where the average number of operations in a query plan was 2.59, more complex query plans are required to expose these newly found bugs, and *QPG* was successful in causing them to be generated.

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

## Q.2 Covering Unique Query Plans

We evaluated whether *SQLancer+QPG* can cover more unique query plans than *SQLancer* and *SQLRight* in 24 hours. Our study in Section 5.2 shows that query plans in previously found bugs are diverse, so covering more unique query plans

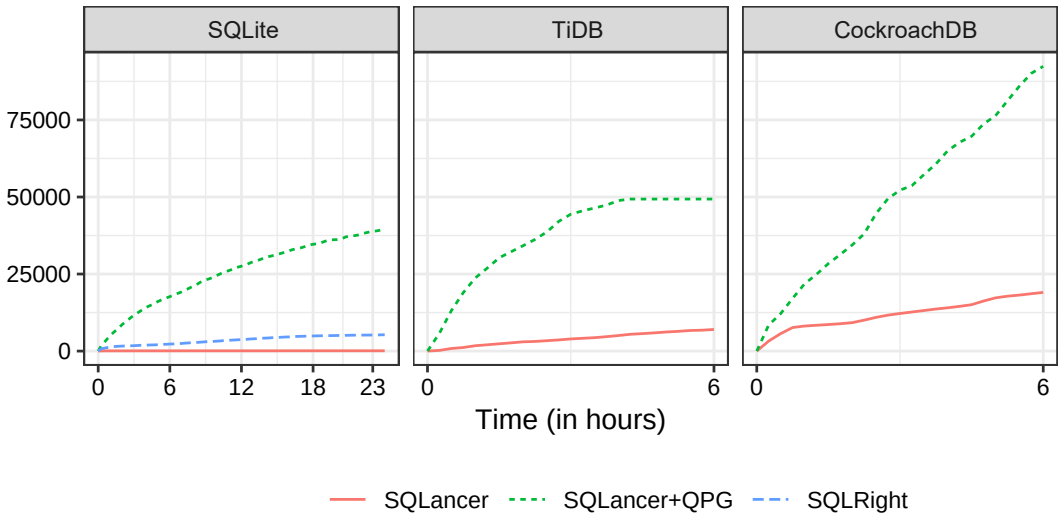


Figure 5.3: The average number of unique query plans across 10 runs in 24 hours.

Table 5.5: The average and median number of query plan lengths across 10 runs in 24 hours.

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

likely increases the probability of finding bugs. We designed *SQLancer+QPG* to explore more unique and complex query plans than *SQLancer*. We used the *TLP* oracle, which is the only test oracle that is supported by all DBMSs we considered.

**Measurements.** Figure 5.3 shows the average number of unique query plans covered by all tools across 10 runs in 24 hours. We recorded the query plans every 15 minutes and removed the names of tables, views, and indexes as described in Section 5.2. For *TiDB* and *CockroachDB*, we could run *SQLancer+QPG* at most for 6 hours, because *SQLancer+QPG* found several crash bugs that remained unfixed during our evaluation. We could run *SQLRight* only on *SQLite*, as *SQLRight* does not support *TiDB* and *CockroachDB*. Table 5.5 shows the average and median lengths of query plans of the queries executed across 10 runs in 24 hours.

Table 5.6: The line and branch coverage across 10 runs in 24 hours.

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

**Results.** On both metrics, the number of unique query plans and their complexity, *SQLancer+QPG* clearly outperforms *SQLancer* and *SQLRight*. *SQLancer+QPG* exercises 4.85–408.48× more unique query plans than *SQLancer* and 7.46× more than *SQLRight*. *CockroachDB* provides fine-grained query plans, which is why *SQLancer+QPG* most clearly outperformed *SQLancer* on this DBMS. The growth rate of *SQLancer+QPG* in *TiDB* stagnates at around 5 hours due to a crash bug that terminated the *TiDB* server process. Table 5.5 shows that the average length of query plans in *SQLancer+QPG* is 1.59–3.79× longer than for *SQLancer*, and 2.16× longer than for *SQLRight*. Only 6 hours are shown for *TiDB* and *CockroachDB* because of crashes. To mitigate randomness, we measured the Vargha-Delaney[171] ($\hat{A}_{12}$) and Wilcoxon rank-sum test[109] ($U$) of *SQLancer+QPG* against *SQLancer*. $\hat{A}_{12}$ measures the *effect size* and gives the probability that random testing of *SQLancer+QPG* is better than random testing of *SQLancer* (*i.e.*, $\hat{A}_{12} > 0.5$ means *SQLancer+QPG* is better). The Wilcoxon rank sum test $U$ is a non-parametric *statistical hypothesis test* to assess whether the result differs across both tools. We reject the null hypothesis if $U < 0.05$, that is, *SQLancer+QPG* outperforms *SQLancer* with statistical significance. For both metrics, $\hat{A}_{12} = 1$ and $U < 0.05$ for *SQLancer+QPG* against *SQLancer* on all DBMSs. The results show that our algorithm continuously generates significantly more unique and complex database states for testing.

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

**Code coverage.** While we were primarily interested in covering more unique query plans, code coverage is a common metric of interest that also gives some insights on how much of a system might be tested. Thus, we evaluated the line and

Table 5.7: The number of all and unique bugs found across 10 runs.

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

branch coverage of all three tools. Since *TiDB* and *CockroachDB* are written in Go, which is not supported by *SQLRight*, we measured code coverage only for *SQLite*. Table 5.6 shows the average percentage of line and branch coverage across 10 runs in 24 hours. Although *SQLancer+QPG* does not aim to maximize code coverage, *SQLancer+QPG* still outperforms *SQLancer* on both line coverage and branch coverage because of more unique query plans covered. *SQLRight* clearly achieves the highest coverage. The reasons for this are that 1) *SQLRight* was designed to increase code coverage, 2) *SQLancer* and *SQLancer+QPG* only generate SQL statements for the core logic of DBMS, while *SQLRight* produces all kinds of SQL statements by parsing the grammar files from DBMSs, and 3) *SQLRight* provides high-quality seeds that already cover 34.1% line coverage and 26.4% branch coverage, outperforming the other tools even without mutations. Since *SQLite* achieves 100% branch coverage in their internal testing,[4] we believe that higher code coverage has a limited contribution to finding logic bugs.

## Q.3 Bug Finding Efficiency

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

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

Table 5.7 shows the sum of all bugs and only assumed-unique bugs found by each tool in 24 hours and 10 runs. Since crash bugs terminate the whole process, all experiments concluded in less than 24 hours until the first crash was observed (*SQLite*: 9 hours, *TiDB*: 1 hour, and *CockroachDB*: 16 hours). We did not restart the testing process as this would disadvantage *SQLancer+QPG* by making it lose the database states. Overall, *SQLancer+QPG* found 2× more bugs and 1.4× more unique bugs than *SQLancer*; 65× more bugs and 17× more unique bugs than *SQLRight*. As duplicate bugs significantly slow down the testing process and hinder finding other bugs, the number of unique bugs is much smaller than the number of all bugs. In *TiDB*, we found several easy-to-reach bugs in `JOIN`s, which do not require complex database states, so the number of all bugs is much higher than for the others. The results further show that bugs can be more efficiently found by exploring more unique query plans.

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

## Q.4 Sensitivity Analysis

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

**Contributions of algorithm components** . Our major contributions are *query plan collection* ③ and *database state mutation* ④ shown in Figure 5.1. To assess their contributions, we derived a new configuration $SQLancer + QPG_r$ that enables only the query plan collection ③, and randomly applies mutations in ④. Figure 5.4 shows the average number of covered unique query plans across 10 runs in 24 hours with the *TLP* oracle. *SQLancer+QPG* outperforms $SQLancer + QPG_r$, demonstrating the contribution of ④. $SQLancer + QPG_r$ outperforms *SQLancer*, demonstrating the contribution of ③. *SQLancer+QPG* has a higher growth rate than $SQLancer + QPG_r$, because ④ gradually learns which mutation operators are promising. Due to the crash bugs, we ran *TiDB* and *CockroachDB* for only 6 hours.

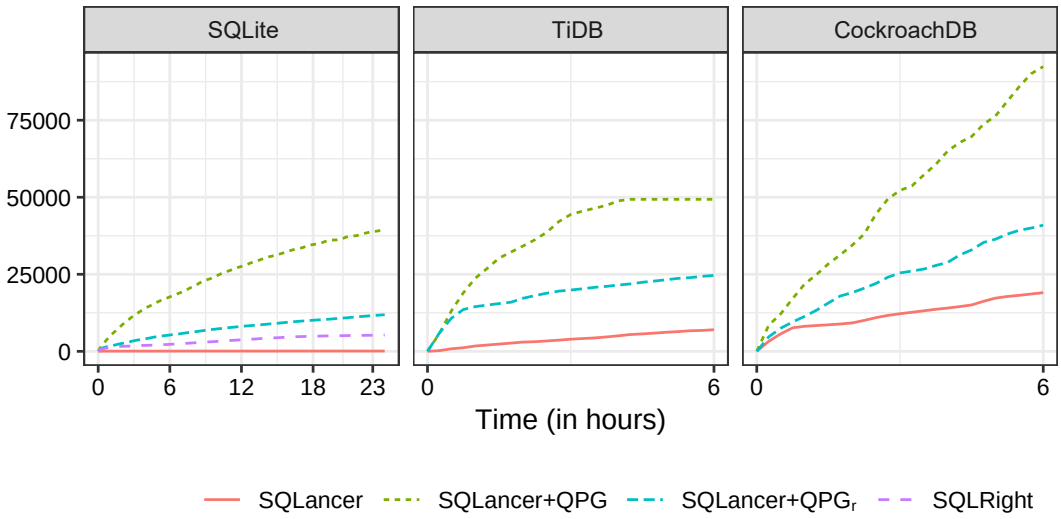


Figure 5.4: The average number of covered unique query plans to evaluate the contributions of algorithm components across 10 runs in 24 hours.

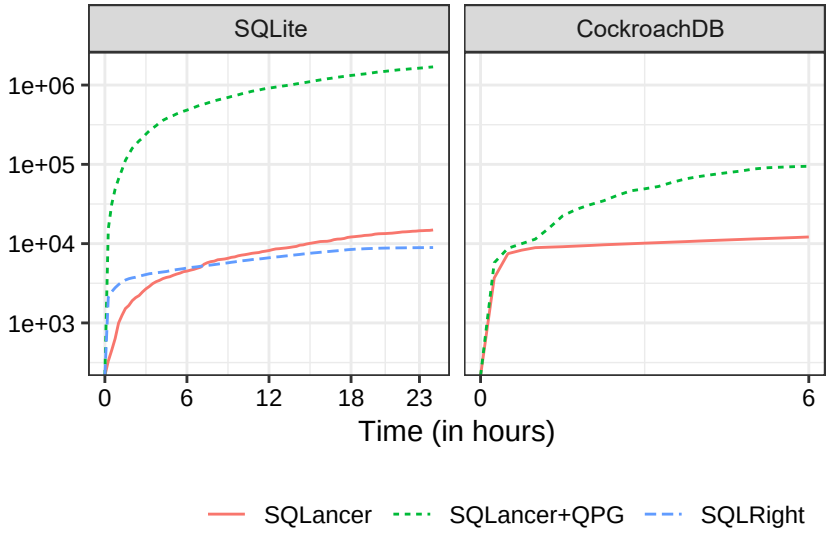


Figure 5.5: The average number of covered unique query plans by the NoREC oracle across 10 runs in 24 hours. The y-axis uses a log scale.

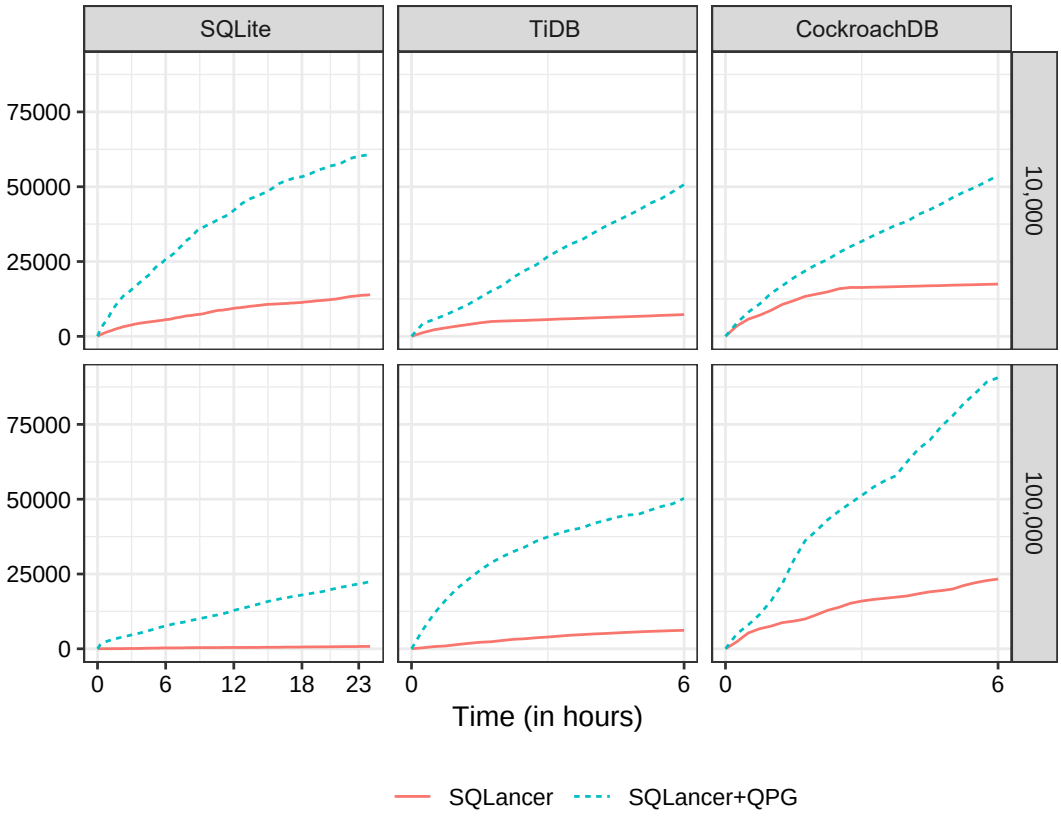


Figure 5.6: The average number of covered unique query plans by varying the maximum number of queries per database state across 10 runs in 24 hours.

**Sensitivity of oracles.** We also evaluated *SQLancer+QPG* with *NoREC*, which is the second state-of-the-art oracle. Figure 5.5 shows the average number of covered unique query plans across 10 runs in 24 hours for the *NoREC* oracle. *SQLancer* lacks a *NoREC* oracle for *TiDB*, so we exclude it here. All tools have a higher number of covered unique query plans with the *NoREC* than with the *TLP* oracle, because of different constraints on queries from *NoREC* and *TLP*. Similar to *TLP*, *SQLancer+QPG* gains a significant advantage over *SQLancer* and *SQLRight* with the *NoREC* oracle.

**Sensitivity of maximum queries per database** . Both *SQLancer+QPG* and *SQLancer* have a configuration to control the number of tested queries before clearing database states and starting a fresh testing instance. The default value for both is 1,000,000. Often, starting a fresh testing instance at ① may result in a higher

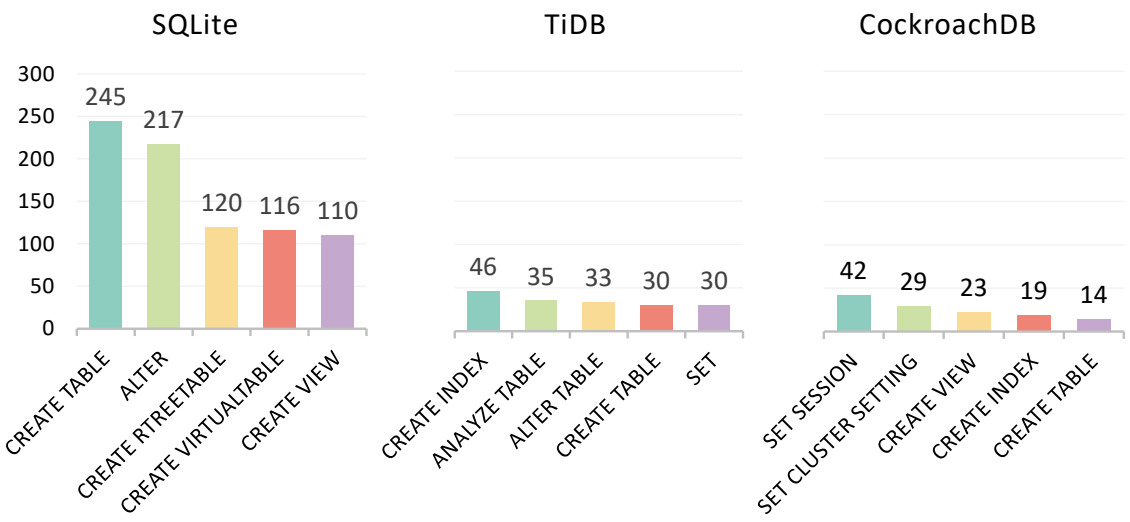


Figure 5.7: How often a mutation was executed for the five most frequently executed mutations for *SQLite*, *TiDB*, and *CockroachDB* across 10 runs.

number of covered unique query plans. To evaluate whether *SQLancer+QPG* still performs well when more frequently resetting database states, we adjusted the number to 10,000 and 100,000, and evaluated the number of their covered unique query plans. Figure 5.6 shows the average number of covered unique query plans under the various maximum number of queries per database state. *SQLancer+QPG* gains a significant advantage over *SQLancer* in all experiments. We clearly see that the rate of newly discovered query plans of *SQLancer* stagnates over time, while *SQLancer+QPG*'s rate continues to increase. Configuring the number is a trade-off since *SQLancer+QPG* creates more complex query plans with a higher number of maximum queries per database state and more unique query plans with a lower number. A user can adjust the configuration option depending on the testing goals.

**Sensitivity of mutations.** To evaluate the contribution of each mutation, we examined how often each mutation (*i.e.*, SQL statement) was executed across 10 runs in 24 hours. Figure 5.7 shows the five most frequently executed mutations for each DDBMS. The most frequently-executed mutation for *SQLite* is `CREATE TABLE`. Other frequently executed mutations either create other kinds of tables that are unique to *SQLite* or change the schema of existing tables using `ALTER`. This is expected, as more kinds of tables subsequently cause *SQLite* to explore more query plans. Despite frequently creating additional tables, we did not observe excessive execution

times, as we limited the maximum number of tables and indexes. For *TiDB* and *CockroachDB*, the number of mutations is much lower than that for *SQLite*, as we could run them for only up to 6 hours. *QPG* favors the mutation **`CREATE INDEX`** for *TiDB*, because indexes allow it to use more efficient physical operators when reading data. For *CockroachDB*, *QPG* favors the mutation **`SET SESSION`**, because it changes the system options, which can have an impact on the query plan. *QPG* favors creating tables as various types of tables are supported in *SQLite*. Overall, all DBMSs have common frequent mutations, such as **`CREATE TABLE`**, yet have distinct frequent mutations, such as **`SET`**, depending on the various characteristics of DBMS.

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

## 5.5 Conclusion

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

## 5.6 Data Availability

Our implementation and experimental data are publicly available at `https://zenodo.org/record/7553013`.

# Chapter 6

# Unified Query Plan Representation

We presented three testing methods based on query plans. However, in practice, query plans are represented in a database-specific manner, making it challenging to efficiently build potentially widely applicable applications. In this chapter, we present an exploratory case study to investigate query plan representations in nine widely-used database systems and propose a *Unified query plan representation* (*UPlan*). This work has been submitted to the 47th International Conference on Software Engineering (ICSE 2025) [88].

## 6.1 Introduction

DBMSs expose query plans to software developers and programs in various formats (*e.g.*, in textual format), in which case we refer to them as *serialized query plans*. Query plans and thus serialized query plans are specific to DBMSs, as they reflect their internal execution steps, which might differ widely across DBMSs. To exacerbate the issue, unlike query languages, for which widely-used, standardized [67], and formalized [65] languages exist, such as the Structured Query Language (SQL) [191], the formats in which serialized query plans are exposed are non-standardized and DBMS-specific. For example, a predicate in the `WHERE` clause of SQL corresponds to a concrete step to filter data in the query plans of *TiDB* [23], but corresponds to a property of another step to scan tables in the query plans of *PostgreSQL* [186]. Overall, we refer to the different ways in which serialized query plans are represented as *query plan representations*.

The plethora of different representations makes it challenging to build general applications on serialized query plans. One common application that uses them is

visualization tools for query plans. Visualization tools visualize serialized query plans, aiming to help users understand and analyze query plans, which is a non-trivial task as highlighted by *PostgreSQL* developers as follows. Due to the DBMS-specific query plan representations, a visualization tool is typically specific to a DBMS, such as pgMustard[1] and PEV2[2] for *PostgreSQL*, and Workbench[3] for *MySQL*, rather than supporting multiple DBMSs. Importantly, automated testing approaches for DBMSs also utilize query plans. As we presented in previous chapters, Query Plan Guidance (*QPG*) [85] uses them as a feedback metric, while Cardinality Estimation Restriction Testing (*CERT*) [86] uses them as part of its test oracle to find performance issues. However, due to DBMS-specific query plan representations, *QPG* and *CERT* were implemented in a DBMS-specific way, which is time-consuming and error-prone.

> "Plan-reading is an art that requires some experience to master, [...]"[a]
>
> [a] https://www.postgresql.org/docs/14/using-explain.html

In this chapter, we systematically study query plan representations. We present an exploratory case study [154], which is a method to investigate a phenomenon in depth, including both qualitative and quantitative research methods. We collected documents, source code, and third-party applications of the query plans in nine popular DBMSs across five different data models, and summarized the commonalities and differences of query plan representations. Our study shows that query plan representations are based on three conceptual components: operations, properties, and formats. Based on the study, we designed a unified query plan representation, which allows the representation of all conceptual components we studied from DBMS-specific query plans.

To demonstrate the utility of the unified query plan representation, we implemented a prototype called *UPlan* to maintain the unified representation, and show three applications based on it. First, we modified a visualization tool to support our unified query plan representation. The results show that existing visualization tool implementations for a specific DBMS could support at least five DBMSs through *UPlan* with only moderate implementation effort. Second, we re-implemented *QPG*

[1] https://www.pgmustard.com/
[2] https://explain.dalibo.com/about
[3] https://dev.mysql.com/doc/workbench/en/wb-performance-explain.html

and *CERT* in a DBMS-agnostic way based on *UPlan*, enabling the large-scale adoption of both testing methods. We found *17* previously unknown and unique bugs in *MySQL*, *PostgreSQL*, and *TiDB*, and 1 bug in the original *QPG* implementation when parsing *TiDB*'s query plans. Last, the unified query plan representation enables the comparison of query plans in different DBMSs, which provides actionable insights for query optimization. We hope that our unified representation reduces the effort to build applications on serialized query plans. We include all study results and the prototype in our supplementary materials. In addition, we provide a comprehensive website, which provides supportive additional information including illustrative examples, explanations of studied query plans, and example applications.

Overall, we make the following contributions:

- A study of query plan representations and results that are publicly available, allowing both practitioners and researchers to study the results.
- A proposal of a unified query plan representation.
- A reusable library *UPlan* for maintaining the unified query plan representation.
- Three examples of how such a representation facilitates the applications on serialized query plans.

## 6.2 Query Plan Case Study

We adopted an exploratory case study as the method to investigate query plan representations. The case study, as an empirical method, is used for investigating a contemporary phenomenon in depth and within its real-world context [153] (the "case"). An exploratory case study is a specific case study that generates new questions, propositions, or hypotheses during the study. In this paper, we chose this method, because we wanted to gain an in-depth understanding of how query plans are represented within mature DBMSs. We followed common guidelines of case study research [154] to design and conduct this study.

Table 6.1: The studied nine popular DBMSs ranging from various data models, development modes, and release dates.

| DBMS | Version | Data Model | Release | Rank |
|---|---|---|---|---|
| *InfluxDB* [7] | 2.7.0 | Time-series | 2013 | 28 |
| *MongoDB* [130] | 6.0.5 | Document | 2009 | 5 |
| *MySQL* [44] | 8.0.32 | Relational | 1995 | 2 |
| *Neo4j* [143] | 5.6.0 | Graph | 2007 | 22 |
| *PostgreSQL* [186] | 14.7 | Relational | 1989 | 4 |
| *SQL Server* [178][1] | 16.0.4015.1 | Relational | 1989 | 3 |
| *SQLite* [136][2] | 3.41.2 | Relational | 1990 | 10 |
| *SparkSQL* [115][3] | 3.3.2 | Relational | 2014 | 37 |
| *TiDB* [23] | 6.5.1 | Relational | 2016 | 107 |

[1] Commercial DBMS.
[2] Embedded DBMS.
[3] Analytics engine.

### 6.2.1 Case Study Design

**Objectives and research questions.** The goal of this study was to investigate the phenomenon of non-standardized query plan representations within real-world DBMSs. We aim to achieve this goal by answering the following research questions (RQs):

RQ.1 How are serialized query plans represented?

RQ.2 Do query plan representations share a common conceptual basis?

**Case selection.** The case study of this paper is characterized as single-case [154]: the query plan representation is the case, while different DBMSs are the units of the analysis. Table 6.1 shows the DBMSs that we selected for the study. To conduct a representative and comprehensive study, we made a diverse selection of DBMSs. First, we chose DBMSs of various data models: relational, document, graph, and time-series data models, based on the assumption that the DBMSs of different models might have different query plan representations. The relational data model is the most widely-used one [37], and other non-relational models, which are also called NoSQL models, are widely used for maintaining unstructured or semi-structured

data [163]. Then, we chose both classic and new-generation DBMSs [135] ranging from release years from the 1980s to the 2010s. The architectures of the chosen DBMSs include standalone DBMSs and an embedded DBMS, *SQLite*, which runs in the same process as the application. We included both a commercial DBMS, *SQL Server*, and open-source DBMSs. Apart from conventional DBMSs, we included an analytics engine for large-scale data processing, *SparkSQL*, which also optimize queries. To choose widely-used DBMSs for study, we chose them referring to the DBMS ranking website[4] as shown in the column *Rank*. We chose the latest release version of each DBMS.

**Data collection.** We collected data from multiple sources: documents, source code, officially integrated development environments (IDEs), and third-party applications based on query plans. The use of multiple data sources allowed us to perform data source triangulation [154], that is, we could confirm the study results from the above four different types of data sources. We included detailed lists of the data sources in the supplementary materials for reference.

**Data analysis.** For each DBMS, we first examined its official documents describing its query plan representation. If the source code was available, we inspected the relevant source code to gain a better understanding of query plan representations. We also ran official test cases in the official IDEs to observe query plan representations in real-world test cases. For the DBMSs that had open-sourced query plan applications, we further examined how query plan representations are explained and utilized from a third-party perspective. To answer RQ.1, we analyzed the query plan representations from the above data sources qualitatively. To answer RQ.2, we qualitatively identified conceptual components in the query plan representations, and quantitatively compared these components. To satisfy observation triangulation [154], one author conducted the study, and another author validated the finding against the raw data.

[4] `https://db-engines.com/en/ranking` as of July 2023.

Code 6.1: Query plan examples of *PostgreSQL* and *SQLite* in text format.

```
1  CREATE TABLE t0 (c0 INT);
2  CREATE TABLE t1 (c0 INT);
3  CREATE TABLE t2 (c0 INT PRIMARY KEY);
4  INSERT INTO t0 SELECT * FROM generate_series(1,1000000);
5  INSERT INTO t2 SELECT * FROM generate_series(1,100);
6
7  ---------------------------------PostgreSQL----------------------------------
8  EXPLAIN (SUMMARY TRUE) SELECT t1.c0 FROM t0 INNER JOIN t1 ON t0.c0 = t1.c0
       WHERE t0.c0 < 100 GROUP BY t1.c0 UNION SELECT c0 FROM t2 WHERE c0 < 10;
9
10 HashAggregate (cost=62998.82..63009.32 rows=1050 width=4)
11  Group Key: t1.c0
12  ->Append (cost=27150.40..62996.20 rows=1050 width=4)
13    ->Group (cost=27150.40..62949.08 rows=200 width=4)
14      Group Key: t1.c0
15      ->Gather Set (cost=27150.40..62948.08 rows=400 width=4)
16        Workers Planned: 2
17        ->Group (cost=26150.38..61901.89 rows=200 width=4)
18          Group Key: t1.c0
19          ->Set Join (cost=26150.38..56906.48 rows=1998164 width=4)
20            Set Cond: (t0.c0 = t1.c0)
21            ->Sort (cost=25970.60..26362.39 rows=156719 width=4)
22              Sort Key: t0.c0
23              ->Parallel Seq
                    Scan on t0 (cost=0.00..10301.95 rows=156719 width=4)
24                Filter: (c0 < 100)
25            ->Sort (cost=179.78..186.16 rows=2550 width=4)
26              Sort Key: t1.c0
27              ->Seq Scan on t1 (cost=0.00..35.50 rows=2550 width=4)
28    ->Bitmap Heap Scan on t2 (cost=10.74..31.37 rows=850 width=4)
29      Recheck Cond: (c0 < 10)
30      ->Bitmap Index Scan on t2_pkey (cost=0.00..10.53 rows=850 width=0)
31        Index Cond: (c0 < 10)
32 Planning Time: 0.124 ms
33
34 ---------------------------------SQLite----------------------------------
35 EXPLAIN QUERY PLAN SELECT t1.c0 FROM t0 INNER JOIN ...;
36
37 `--COMPOUND QUERY
38    |--LEFT-MOST SUBQUERY
39    |  |--SCAN t0
40    |  |--SEARCH t1 USING AUTOMATIC COVERING INDEX (c0=?)
41    |  `--USE TEMP B-TREE FOR GROUP BY
42    `--UNION USING TEMP B-TREE
43       `--SEARCH t2 USING COVERING INDEX sqlite_autoindex_t2_1 (c0<?)
```

### 6.2.2 Findings Overview

We found that the studied DBMSs share three conceptual components: *operations*, *properties*, and *formats*. *Operations* are concrete steps executed by DBMSs to retrieve, process, or output data in response to a query, such as `Full Table Scan`, which refers to the step to scan an entire table. While, query plans are considered DAGs, we found that, in practice, all *operations* in serialized query plans are organized in a tree structure, and each *operation* takes the output of its children as input and produces a new output that contains an intermediate result. The root *operation* produces the result of the whole query. Each *operation* is associated with zero or multiple *properties*, which involve *operation*-related information, such as `row`, which refers to the estimated number of rows returned. Apart from *operation*-associated *properties*, plans also have *properties* associated with them, such as `planning time`, which refers to the time to generate the query plan. DBMSs typically allow serializing query plans in various *formats*, such as text, table, JSON, and XML. We detail each conceptual component in the next subsections.

Code 6.1 shows two examples of query plan representations of *PostgreSQL* and *SQLite* in a textual format. Lines 1–5 show the SQL statements that create and populate the tables. The function `generate_series` generates data to populate tables `t0` and `t2`. For *PostgreSQL*, executing the statement in line 8 outputs the serialized query plan as shown in lines 10–32. `(SUMMARY TRUE)` in line 8 specifies the output of plan-associated properties in line 32. For *SQLite*, executing the statement in line 35 outputs the serialized query plan as shown in lines 37–43. The bold texts denote operations, and the non-bold texts denote properties. For example, the operation **HashAggregate** in line 10 has the properties `cost, rows, width, and Group Key`. The tree structure is denoted through hierarchical indents, in which the operation with longer indents is a child of the operation with short indents. Although both DBMSs are based on the relational data model, and support a textual format, their query plan representations are significantly different.

Table 6.2: The number of operations in query plan representations.

| DBMS | Producer | Bag | Join | Folder | Projector | Executor | Consumer | Total |
|---|---|---|---|---|---|---|---|---|
| *InfluxDB* | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| *MongoDB* | 14 | 9 | 0 | 5 | 3 | 10 | 3 | 44 |
| *MySQL* | 15 | 3 | 2 | 1 | 0 | 2 | 0 | 23 |
| *Neo4j* | 18 | 11 | 43 | 6 | 3 | 17 | 13 | 111 |
| *PostgreSQL* | 18 | 8 | 3 | 3 | 0 | 9 | 1 | 42 |
| *SQL Server* | 15 | 3 | 3 | 3 | 0 | 16 | 19 | 59 |
| *SQLite* | 3 | 6 | 3 | 0 | 0 | 5 | 0 | 17 |
| *SparkSQL* | 7 | 1 | 2 | 6 | 0 | 43 | 18 | 77 |
| *TiDB* | 19 | 6 | 7 | 5 | 1 | 13 | 5 | 56 |
| **Avg:** | 12 | 5 | 7 | 3 | 1 | 13 | 7 | 48 |

Table 6.3: The number of properties in query plan representations.

| DBMS | Cardinality | Cost | Configuration | Status | Total |
|---|---|---|---|---|---|
| *InfluxDB* | 5 | 0 | 0 | 1 | 6 |
| *MongoDB* | 16 | 5 | 18 | 12 | 51 |
| *MySQL* | 3 | 6 | 3 | 10 | 22 |
| *Neo4j* | 3 | 3 | 12 | 7 | 25 |
| *PostgreSQL* | 8 | 17 | 42 | 40 | 107 |
| *SQL Server* | 4 | 4 | 7 | 3 | 18 |
| *SQLite* | 0 | 0 | 3 | 0 | 3 |
| *SparkSQL* | 11 | 11 | 0 | 0 | 22 |
| *TiDB* | 2 | 5 | 4 | 1 | 12 |
| **Avg:** | 6 | 6 | 10 | 8 | 30 |

### 6.2.3 Operations

**Identification.** We identified operations from the source code. *MongoDB*, *MySQL*, *PostgreSQL*, and *TiDB* specify operations in enumeration variables or lists. *Neo4j* and *SparkSQL* define each operation as a class or structure. *SQLite* defines operations as strings that are passed to the query plan generation process. We found that only *SQL Server* and *Neo4j* provided detailed documents for operations, while the other DBMSs' documents had incomplete lists of operations, and relied on illustrative examples.

**Classification.** We classified operations into seven categories, as shown in the left part of Table 6.2. On average, every DBMS defines 48 operations in query plans. *Neo4j* has the most operations, while *SQLite* has the fewest operations. We noticed that *Neo4j* requires more operations on nodes and relationships of the graph data

model, such as the operation to set node properties,[5] and *SQLite* is a lightweight DBMS with a limited number of operations, such as lacking the operations for creating tables. Overall, NoSQL DBMSs have more operations than relational DBMSs. An exception is *InfluxDB*, which does not define operations. *InfluxDB*'s operations are disregarded in query plans due to the limited set of operations supported by the single-entity time-series data.

**Terminology.** We define the operation categories through the following terms. Recall that data models organize data in entities and attributes, so we represent an entity $E = \{a_1, a_2, ..., a_n\}$, in which $a$ is an attribute, and represent a multiset[6] of entities $D = \{E_1, E_2, ..., E_n\}$. $O_c(input) = output$ represents that an operation $O$ of the category $c$ receives input from children operations and produces output to the parent operation. Considering that many operations might apply to varying numbers of children, we chose to model the input from all children as a single input. We give a detailed explanation of each operation category as follows.

**Producer.** The *Producer* category consists of the operations that retrieve data from storage or return constants instead of from children's operations. We denote them as $O_{Producer}() = D_o$. The operations in the *Producer* category are data sources of queries, so they are typically leaf nodes of query plans. For example, in Code 6.1, the operation **SEARCH** in line 43 represents a full table scan, and **Bitmap Heap Scan** in line 28 represents a data scan from bitmaps in heap memory. Six of nine DBMSs define more than ten operations in the *Producer* category, because reading data is usually expensive, and thus reads are customized for different scenarios aiming to improve efficiency. For example, indexes [71, 99] can be used to efficiently read data.

**Bag.** The *Bag* category consists of the operations that change the permutation and combination of entities, such as sort and union, with no changes to attributes. We denote them as $O_{Bag}(\{E_i|E_i \in D_i\}) = \{E_o|E_o \in D_o, D_o \subseteq D_i\}$. In Code 6.1, the operation **Append** of *PostgreSQL* merge entities from different children operations to a single set by the operations in lines 13 and 28, and is associated with query

[5] `https://neo4j.com/docs/cypher-manual/current/execution-plans/operators#query-plan-set-node-properties-from-map`

[6] A collection of elements in which elements may occur more than once.

Table 6.4: An example query plan of the *Neo4j* operations of the Join category.

Planner COST
Runtime version 5.10

| **Operator** | **Rows** | **...** |
|---|---|---|
| +ProduceResults | 8 | ... |
| +UndirectedRelationshipIndexContainsScan | 8 | ... |

Total database accesses: 5, total allocated memory: 184

clause **`UNION`** in line 8. To execute a similar functionality in *SQLite*, the operations **COMPOUND QUERY** and **UNION** together combine data objects by the operations in lines 37 and 42, and both operations are also in the *Bag* category.

**Join.** The *Join* category consists of the operations that generate new entities by recombining attributes. We denote them as $O_{Join}(\{E_i|E_i \in D_i\}) = \{E_o|\forall a_o \in E_o, a_o \in D_o, D_o \subseteq D_i\}$. In Code 6.1, the operation **Set Join** in line 19 combines two ordered data from the operations in lines 21 and 25 based on the common fields `t0.c0` and `t1.c0`. *MongoDB* has no *Join* operations, because it includes only a single document entity for querying and lacks support for combining data from multiple documents. *Neo4j* has 34 operations, while other DBMSs have less than 10 operations, because we classified the operations on the edges of the graph data model as belonging to the *Join* category. In the graph data model, edges establish relationships between nodes, and a broader range of operations can be performed on the edges. For example, in *Neo4j*, executing the simple query **`MATCH`** `()-[r]->()` **`WHERE`** `r.title ENDS WITH 'developer'RETURN r` retrieves the relationships whose properties satisfy `r.title ENDS WITH 'developer'`. The corresponding query plan is shown in Table 6.4. Each line in the table represents an operation and associated properties, and the content outside the table is plan-associated properties. The query plan scans the relationships, which indicates both nodes (*i.e., entities*) each relationship connects, so the operation **UndirectedRelationshipIndexContainsScan** belongs to the Join category.

**Folder.** The *Folder* category consists of the operations that derive new entities from a set of entities. Formally, we denote them as $O_{Folder}(\{E_i|E_i \in D_i\}) = \{E_o|E_o = \psi(D_o), D_o \subseteq D_i\}$, in which $\psi()$ denotes a function, such as $MAX()$. In Code 6.1,

the operations **HashAggregate** and **Group** are in the *Folder* category and represent data aggregation and grouping, respectively. *SQLite* does not define operations in the *Folder* category, but shows similar information in properties together with the operations in the *Producer* category. The operations in the *Folder* denote DBMSs' data transformation capability, so most DBMSs support operations in *Folder* category.

**Projector.** The *Projector* category consists of the operations that remove attributes from all entities. Formally, we denote them as $O_{Projector}(\{E_i|E_i \in D_i\}) = \{E_o|E_o \subseteq E_i\}$. No operation in Code 6.1 belongs to this category, and 6 of 9 DBMSs have no operations in this category. We observed that these operations correspond to **`SELECT`** clauses, and they are not explicitly denoted in query plans.

**Executor.** The *Executor* category consists of the operations that make no change to entities and attributes. We denote them as $O_{Executor}(D_i) = \{D_o|D_o = D_i\}$. In Code 6.1, the operation **Gather Set** in line 15 merges the data from the operation **Group** running in other computing nodes for a distributed architecture. DBMSs define these operations to cater to various designs and goals. For example, *PostgreSQL* defines the operation **MEMORIZE** to cache the output from node children into memory to speed up processing. We classified these operations into the *Executor* category. The average number of operations in the *Executor* category is 13, and *SparkSQL* has significantly more operations, 43, in the *Executor* category than others, because it defines multiple operations to interact with other components, such as the Python library *pandas*.

**Consumer.** The *Consumer* category consists of operations that have no output. We denote them as $O_{Consumer}(D_i) = \{\}$. Apart from queries, which, in the SQL, are **`SELECT`** statements, DBMSs also support other statements, such as **`CREATE`** and **`UPDATE`** in SQL. DBMSs also expose query plans for these statements, and name them as execution plans for wider usage.[7] The operations of the *Consumer* category usually modify stored data or system variables. For example, *SparkSQL* uses the operation **SetCatalogAndNamespace** to control a particular system variable, and we assigned

[7] `https://neo4j.com/docs/cypher-manual/5/execution-plans/`

these operations to the *Consumer* category.

### 6.2.4 Properties

**Identification.** Each property is associated with either an operation or a query plan, and the available properties are statically encoded as strings near the generation processes of associated operations or query plans in the source code. *InfluxDB*'s query plan representation includes only a list of plan-associated properties, while other DBMSs include both plan-associated and operation-associated properties. In Code 6.1, *PostgreSQL*'s operations have various general properties enclosed in brackets, along with operation-specific properties in the subsequent lines. At the bottom of the serialized query plan, the property `Planning Time` is plan-associated and represents the time to produce the query plan. In the documentation, similar to operations, only *SparkSQL* provides a comprehensive list of properties, while other DBMSs only show examples of properties. We explain that it is difficult to maintain the documentation of properties, which are diverse and evolving over versions. To maintain the information of properties, some third-party tools, like pgMustard, maintain a curated list of properties with accompanying explanations, but are usually commercial.

**Classification.** We identified four categories of properties, as shown in the right part of Table 6.3. On average, every DBMS defines 30 properties. *PostgreSQL* has the most properties, since it includes many fine-grained properties. For example, it defines three properties to show the status of parallel computations: `worker number`, `worker launched`, and `worker planned`, while other DBMSs provide at most one property for parallel computation. Some properties in a DBMS may be operations in another DBMS, we classified them according to the definition in most DBMSs. For example, in Code 6.1, the property `Filter` in line 24 represents the predicates to exclude data, while *TiDB* denotes a filter in an operation **Select**. Because 5 of 9 DBMSs denote filters in properties, we classified the operation in *TiDB* also into properties. We give a detailed explanation of each property category as follows.

**Cardinality.** The *Cardinality* category consists of the numeric properties that denote the estimated data size returned by operations. The properties in this category can be associated with operations of any category or the serialized query plan as a whole. In Code 6.1, the properties `rows` and `width` belong to the *Cardinality* category and represent the estimated number of returned rows and width. These estimates are derived from statistical information [80] that DBMSs collect, such as the total number of rows and maximum values. Query plans with lower estimated cardinalities are more likely to be selected for execution during cost-based query optimization. Some DBMSs, such as *MySQL*, provide more fine-grained information about the number of rows that are read and returned. As a lightweight DBMS, *SQLite* uses simple heuristics to estimate cardinalities, and omits properties in the *Cardinality* category.

**Cost.** The *Cost* category consists of the numeric properties that denote the estimated resource consumption. The properties in this category can be associated with operations of any category or the serialized query plan as a whole. In Code 6.1, the property `cost` is in the *Cost* category, and the two numbers of `cost` denote the cost scores of starting and finishing the associated operation by estimating the total consumption of disk and CPU. As in the *Cardinality* category, *SQLite* lacks properties in the *Cost* category.

**Configuration.** The *Configuration* category consists of the properties that configure the operations' parameters, and their values are configuration options which are usually strings or boolean values. The properties in the *Configuration* category can be associated with operations in any category or the serialized query plan, and are typically specific to operations. In Code 6.1, *PostgreSQL*'s properties `Group Key`, `Set Cond`, `Sort Key`, `Recheck Cond`, `Index Cond`, `Filter` are in the *Configuration* category and are specific to the associated operations to show the keys used to group, the condition to join, the key to sort, the condition to check, the index condition, and the predicate to exclude data, respectively. *SQLite*'s property **`USING`** `COVERING` **`INDEX`** denotes the index condition.

Table 6.5: The officially supported formats of query plans.

| | Natural | | | Structured | | |
|---|---|---|---|---|---|---|
| **DBMS** | **Graph** | **Text** | **Table** | **JSON** | **XML** | **YAML** |
| *InfluxDB* | | ✓ | | | | |
| *MongoDB* | ✓ | | | ✓ | | |
| *MySQL* | ✓ | | ✓ | ✓ | | |
| *Neo4j* | ✓ | | ✓ | ✓ | | |
| *PostgreSQL* | | ✓ | ✓ | ✓ | ✓ | ✓ |
| *SQL Server* | ✓ | ✓ | ✓ | | ✓ | |
| *SQLite* | | ✓ | | | | |
| *SparkSQL* | ✓ | ✓ | | | | |
| *TiDB* | ✓ | | ✓ | ✓ | | |

**Status.** The *Status* category consists of the properties of run-time status, and their values are runtime metrics which are usually strings or numbers. These properties can be associated with operations in any category or the serialized query plan, and typically differ depending on the operations they are attached to. In Code 6.1, the property `Workers Planned` is in the *Status* category and shows the number of available computing nodes to execute the associated operation. As another example, *TiDB* defines the property `taskType` to show the name of the computing nodes that the operation is assigned to execute. The properties in the *Status* category show running status, and are determined by the execution environment, while the properties in the Parameters are usually decided by queries. The properties in both *Status* and *Configuration* categories are customized, and thus are typically different across DBMSs, while the properties in other categories share similar semantics or functionalities across DBMSs.

### 6.2.5 Formats

DBMSs serialize query plans to various formats for different purposes. The formats are typically controlled by a specific configuration in queries, such as for *PostgreSQL*, the statement `EXPLAIN (FORMAT JSON)SELECT...` serializes the query plan representation to JSON format. We classified all formats into two categories: *natural* formats which are optimized for readability, and *structured* formats which are optimized for machine reading. We also consider the graph formats that are

supported in official IDEs.

Table 6.5 shows the different formats of query plan representations. Overall, DBMSs support more formats in the *natural* category rather than the *structured* category, suggesting that DBMSs prioritize readability over machine processing. Due to the lack of a standard, none of the formats is supported by all DBMSs. For the same DBMS, the formats in the *natural* category usually include less information than the formats in the *structured* category. For example, in Code 6.1, the property `Parent Relationship` represents how the associated operation passes data to another operation. This property is ignored in the text format of the *natural* category, but is shown in the JSON format of the *structured* category. We provide more details of each format as follows.

**Natural category.** The *natural* category includes graph, text, and table formats. Query plans are usually serialized as graphs for DBMSs' IDEs, such as Workbench[8] for *MySQL*, Compass[9] for *MongoDB*. Text formats represent query plans as plain text, such as shown in Code 6.1. Table formats encode each operation and associated properties in a line, and use line numbers to represent the tree structure of query plans. Graph formats are intuitive to understand, so graph formats are supported by most DBMSs.

**Structured category.** The *structured* category includes the JSON, XML, and YAML formats. These formats are standardized and widely used for exchanging data [79, 22]. JSON is more widely supported by DBMSs than other *structured* formats, and *PostgreSQL* supports all *structured* formats. *Structured* formats are not supported by some DBMSs, such as *InfluxDB*, *SQLite*, and *SparkSQL*. *SQLite* can output a *structured* format of bytecode, which includes low-level instructions, not the operations and properties, so we do not consider it as a *structured* format of query plan representations.

**Visualization.** Apart from the graph formats in official IDEs, third-party visualization tools show query plans based on *structured* formats to enhance the readability of query plans. Table 6.6 shows the visualization tools we found for the studied

[8] `https://dev.mysql.com/doc/workbench/en/wb-performance-explain.html`
[9] `https://www.mongodb.com/docs/compass/current/query-plan/`

Table 6.6: Third-party visualization tools for query plans.

| **Tool** | **DBMSs** | **License** |
|---|---|---|
| Postgres Explain Visualizer 2 [194] | *PostgreSQL* | Open-source |
| pgmustard [193] | *PostgreSQL* | Commercial |
| pganalyze [192] | *PostgreSQL* | Commercial |
| ApexSQL [188] | *SQL Server* | Commercial |
| Plan Explorer [129] | *SQL Server* | Commercial |
| Azure Data Studio [189] | *SQL Server* | Commercial |
| Dbvisualizer [190] | *MySQL*, *PostgreSQL*, *SQL Server* | Commercial |

DBMSs. Six of the seven tools are commercial, suggesting the value of understanding query plan representations for developers. Building these tools requires non-trivial effort, because a tool is specific to a DBMS.

## 6.3 Unified Query Plan Representation

Our study in Section 6.2 shows that query plan representations share the same conceptual basis, which is why we propose a unified query plan representation that is:

1) complete, to include all information of a query plan,

2) general, to support various DBMSs we studied,

3) extensible, to support the DBMSs we did not study.

### 6.3.1 Design

To define and illustrate the unified query plan representation, we adopted Extend Backus Naur Form (EBNF) [133], which is a metasyntax notation to express context-free grammars. Code 6.2 shows the unified query plan representation in EBNF. Following our study in Section 6.2, the unified representation includes the three identified conceptual components of different categories. We define **plan** as a **tree** that can have plan-associated **properties**. Within the **tree**, a **node** is defined as an **operation** and zero or multiple operation-associated **properties**. **operations** and **properties** are key-value pairs including corresponding **categories**, **identifiers**, and

Code 6.2: The unified query plan representation in EBNF.

```
1  plan ::= ( tree )? properties
2  tree ::= node ( '--children-->' '{' tree (',' tree)* '}' )?
3  node ::= operation properties
4  operation ::= 'Operation' ':' operation_category '->' operation_identifier
5  properties ::= ( property ( ',' property )* )?
6  property ::= property_category '->' property_identifier ':' value
7  operation_category ::= 'Producer' | 'Bag' | 'Join' | 'Folder' | 'Executor' |
       'Projector' | 'Consumper'
8  property_category ::= 'Cardinality' | 'Cost' | 'Configuration' | 'Status'
9  operation_identifier ::= keyword
10 property_identifier ::= keyword
11 keyword ::= letter ( letter | digit | '_' )*
12 value ::= string | number | boolean | 'null'
13 string ::= '"' ( letter | digit )* '"'
14 number ::= '-'? digit+
15 boolean ::= 'true' | 'false'
16 letter ::= [a-zA-Z]
17 digit ::= [0-9]
```

**values**. In Figure 6.1, we also visualized the EBNF into a railroad diagram for an intuitive explanation. To make it concise, we only show the symbols **tree** and **node**, which express the high-level structure.

We use a unified naming convention to denote operations and properties in the unified query plan representation. A unified naming convention increases the readability and consistency of the representation while avoiding name collisions. Section 6.2 shows that various operations and properties share similar semantics, so we mapped DBMS-specific names of operations and properties to unified names. For example, we mapped the operation name **Seq Scan** in *PostgreSQL*, **Table Scan** in *SQL Server*, and **TableFullScan** in *TiDB* to **Full Table Scan**.

To support the unified naming convention, we also allow an operation or property in a query plan to be mapped to a property or operation in the unified representation. For example, *TiDB* defines a specific operation to filter data, and we mapped it to a property for consistency across other DBMSs.

### 6.3.2 Analysis

We qualitatively analyze whether this design achieves three goals.

**Completeness.** The unified query plan representation includes the three conceptual components: operations, properties, and formats, which we identified in

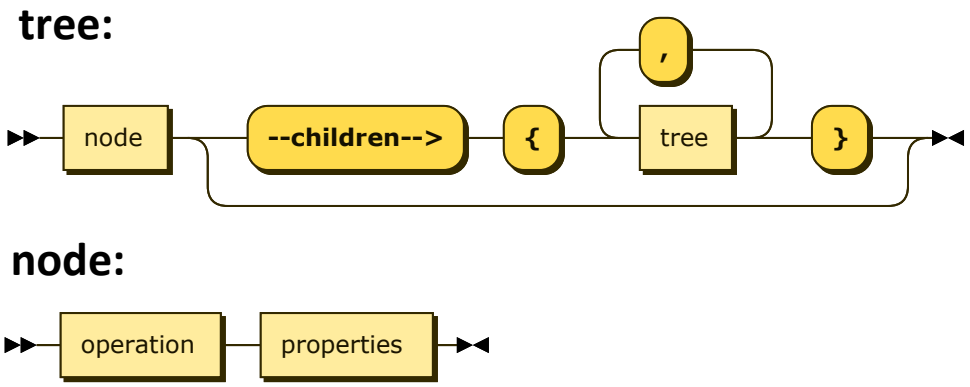


Figure 6.1: Railroad diagrams of the symbols tree and node.

Section 6.2. Operations and properties are included in the tree of the unified representation, and the unified representation can be serialized into other standard formats, such as JSON and XML, which are used in query plan representations.

**Generality.** The unified query plan representation supports the query plan representations of the nine DBMSs that we studied. *InfluxDB*'s query plan includes a list of properties without operations, which can be represented by plan-associated properties in the unified representation. For the other DBMSs, we identified operations and properties, mapped them into unified names, and organized them into our unified representation.

**Extensibility.** The definitions of operations, properties, and categories in the unified query plan representation can be extended or shrunk while keeping forward and backward compatibility. Forward compatibility refers to allowing a system accepts input intended for a later version of itself, and backward compatibility refers to allowing a system accepts input intended for an older version of itself [173]. Both evaluate whether the applications based on the unified query plan representation still work if we update the representation to support more or different DBMSs. To keep forward compatibility, we can add more categories by expanding **operation_category** to include more category names, and add more operations and properties whose names comply with the definition of the keyword at line 11 at Code 6.2. An existing application still can parse the revised representation by ignoring the newly added categories, operations, and properties, or handle them in a generic way (*e.g.*, a

visualization tool could represent unknown operations using a generic visual shape). For backward compatibility, similarly, an existing application still can parse the old version of the representation, whose categories, operations, and properties are included in the new version of the representation.

## 6.4 Applications

We implemented the prototype of a reusable library, *UPlan*, to maintain the unified query plan representation, and sought to demonstrate its utility through three applications:

**A.1 Testing.** The DBMS testing methods *QPG* and *CERT* were implemented in a DBMS-specific way due to DBMS-specific query plan representations, and we show how the unified query plan representation allows both methods to be implemented in a DBMS-agnostic way.

**A.2 Visualization.** Visualization tools visually display serialized query plans to ease understanding, but are typically specific to a particular DBMS. We show a general visualization tool based on the unified query plan representation.

**A.3 Benchmarking.** Benchmarking is an important method to evaluate the performance of DBMSs. We show a case analysis of a comparison of query plan representations using *UPlan*. We hope that allowing developers to easily compare different DBMSs' query plans enables them to improve their DBMSs' query optimization capabilities.

**DBMSs.** For A.2 and A.3, we used MongoDB, MySQL, Neo4j, PostgreSQL, and TiDB, because they support the JSON format of query plans because JSON is the most widely supported structural format. Within these DBMSs, we used MySQL, PostgreSQL, and TiDB for A.1 as they are supported by *SQLancer*. We used the same versions of DBMSs that we studied in the Table 6.1.

**Data set.** For A.1, we used *SQLancer* to generate test cases. For A.2 and A.3, we collected the serialized query plans of the queries from the TPC-H benchmark

suite [123] on the five DBMSs. TPC-H benchmark suite comprises 8 tables and 22 queries for relational DBMSs: MySQL, PostgreSQL, and TiDB in our experiments. MongoDB and Neo4j require manual effort to adapt TPC-H as it was not designed for non-SQL languages and non-relational data models. MongoDB adopts a document data model, which lacks support for join operations, so we embedded all entities in one document and rewrote queries 1, 3, and 4 in the MongoDB Query Language (MQL), following a tutorial.[10] For Neo4j, which adopts a graph data model, we mapped nodes into rows of the relational data model and edges into foreign keys of the relational data model following another tutorial,[11] and rewrote queries 1–14, 16–19 using the Cypher Query Language (CQL) following an example.[12]

**Implementation.** *UPlan* is a reusable library that consists of around 300 lines of Python code to implement a reusable library, which allows adding or updating operations and properties. The prototype supports serializing query plan representations into JSON as well as text formats, and provides an interface for supporting more formats. We also implemented five customized converters to parse the query plan representations from existing JSON formats to the unified query plan representation, and each parser has around 200 lines of code.

## A.1 Testing

We show an application on *UPlan* to implement the testing methods *QPG* [85] and *CERT* [86] in a DBMS-agnostic way. *QPG* is a test case generation approach that is guided by query plans; specifically, it mutates a database if no new query plans have been observed for a specific number of randomly generated queries, aiming to subsequently exercise new query plans, and thus exploring "interesting" behaviors. In terms of implementation, evaluating whether a query plan is structurally different from another requires ignoring unstable information, such as random identifiers and the estimated cost in query plans. *CERT* is a test oracle for finding performance issues by comparing estimated cardinalities, which have to be extracted from query

[10] `https://alronz.github.io/Factors-Influencing-NoSQL-Adoption/site/MongoDB/Examples/DenormalisedModel/`

[11] `https://alronz.github.io/Factors-Influencing-NoSQL-Adoption/site/Neo4j/Examples/TPC-HQueries/`

[12] `https://github.com/aiquis/tpch-neo4j/blob/5e4e5c/tpch_queries.cql`

plans. Both methods were implemented in *SQLancer*, which is a popular and widely used tool for automatically testing DBMSs. The original implementations used DBMS-specific and error-prone methods, such as string matching and substitution that had to be implemented for every DBMS that was supported. Both approaches support relational DBMSs, but were not applied to the popular open-source DBMSs *MySQL* and *PostgreSQL*, as additional DBMS-specific parsers would have been required. Based on *UPlan*, we implemented general parsers for both methods to support all *UPlan*-compatible DBMSs.

To evaluate *UPlan* on *QPG* and *CERT*, we applied the general parser to MySQL, PostgreSQL, and TiDB. We ran our revised versions of *QPG* and *CERT* for 24 hours and found *17* bugs, as shown in Table 6.7. All reported bugs are unique and previously unknown. Additionally, *CERT* found hundreds of potential bug-inducing test cases in 24 hours, but it is challenging to distinguish their uniqueness, which requires developers' expertise. To avoid burdening developers, we will report them after these bugs have been fixed. Code 6.3 shows a bug in MySQL found by the test case generated by *QPG* with *UPlan*. Note that we identified this bug by the test oracle Ternary Logic Partitioning (TLP) [150]; however, for presentation, we simplified the bug-inducing test case by demonstrating that the same query returns different results in lines 4 and 6 depending on whether the index exists. The cause of the bug was an incorrect table look-up due to the index inserted by the SQL statement in line 5. Using *UPlan*, we were able to apply *QPG* to MySQL easily, which was previously incompatible and untested by *QPG*. This enabled us to find this bug in MySQL.

Table 6.7: Previously unknown and unique bugs found with *UPlan*.

| **DBMS** | ***QPG*** | ***CERT*** | **All** |
|---|---|---|---|
| MySQL | 6 | 1 | 7 |
| PostgreSQL | 0 | 1 | 1 |
| TiDB | 7 | 2 | 9 |
| | | **Sum:** | 17 |

We also found that *UPlan* reduces the risk of implementation bugs for DBMS-specific query plan parsers. We identified an implementation bug in *SQLancer*.[13]

[13] `https://github.com/sqlancer/sqlancer/pull/900`

Code 6.3: Bug #113302 found by *QPG* with *UPlan*.

```
1 CREATE TABLE t0(c0 INT, c1 INT);
2 INSERT INTO t0(c1, c0) VALUES(0, 1);
3
4 SELECT * FROM t0 WHERE t0.c1 IN (GREATEST(0.1, 0.2)); -- empty result
5 CREATE INDEX i0 ON t0(c1);
6 SELECT * FROM t0 WHERE t0.c1 IN (GREATEST(0.1, 0.2)); -- {1|0}
```

Specifically, the query plan parser for *TiDB* failed to exclude random identifiers due to an incorrect parameter for **`EXPLAIN`**. With the single implementation for a parser and the unified query plan representation, we have a lower risk of introducing these implementation bugs.

*UPlan* enables large-scale adoption for testing methods *QPG* and *CERT* in a DBMS-agnostic implementation way.

## A.2 Visualization

We implemented a visualization tool for serialized query plans by modifying PEV2 [194], a customized query plan visualization tool for *PostgreSQL*, to use the unified query plan representation. We modified its parser to support identifying the unified query plan representation, and updated its definitions of visualized elements, such as operation names.

Using *UPlan*, our customized PEV2 supports query plan visualization of the five DBMSs in the same implementation by modifying around 800 lines of code. Suppose implementing a new visualization tool for another DBMS also requires at least an additional 800 lines of code, then it would require $800 * (5 - 1) = 3,200$ lines of code to support the five DBMSs based on PEV2 without the unified query plan representation. *UPlan* reduces the effort of building a new visualization tool for serialized query plans.

Figure 6.2 shows six examples of visualized serialized query plans from the two tools in five DBMSs. The left green dashed box shows the visualized serialized query plan in PEV2, which supports only *PostgreSQL*, while the right orange dashed box includes the visualized serialized query plans in our customized PEV2, which uses the unified query plan representation, and thus supports multiple DBMSs. These examples are based on query 1 of the TPC-H benchmark suite. An operation and

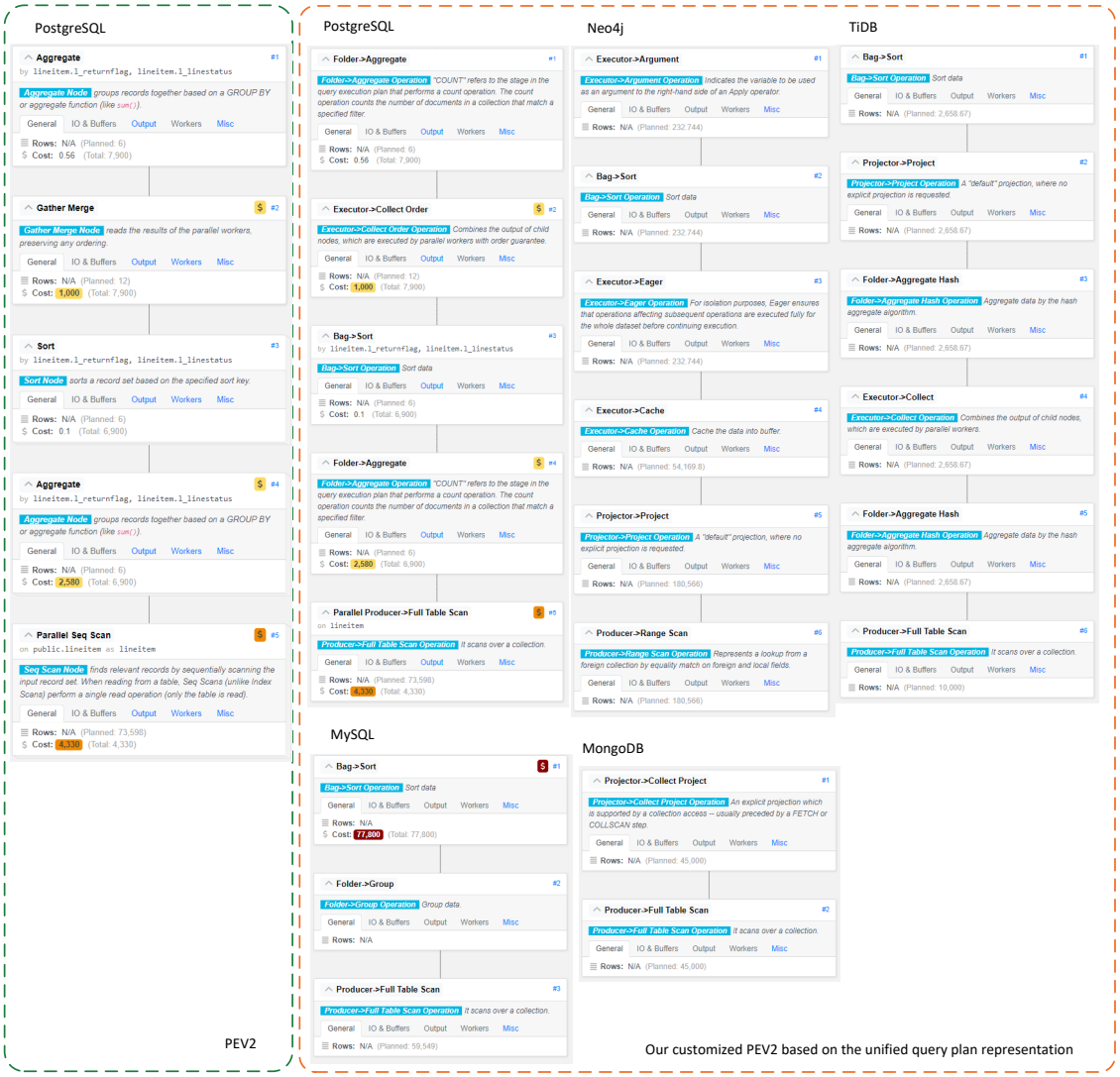


Figure 6.2: Visualized serialized query plan of query 1 from TPC-H benchmark by two tools in five DBMSs.

its associated properties are visualized in a node. For example, in the first node of *MySQL* query plan, **Bag->Sort** represents the operation **Sort**, which belongs to the category **Bag**. The following is the description and properties. PEV2 shows the original names of operations and properties in *PostgreSQL*'s query plans, while our customized PEV2 shows the unified names of operations and properties.

Existing DBMS-specific visualization tools could support more DBMSs if they supported our unified query plan representation, and only moderate implementation effort is required.

Table 6.8: The average number of operations in query plans of the queries from the TPC-H benchmark suite.

| DBMS | Producer | Bag | Join | Folder | Projector | Executor | Total |
|---|---|---|---|---|---|---|---|
| *MongoDB* | 1.00 | 0.00 | 0.00 | 0.00 | 1.00 | 0.00 | 2.00 |
| *MySQL* | 4.55 | 0.82 | 2.77 | 0.86 | 0.00 | 0.27 | 9.27 |
| *Neo4j* | 0.39 | 0.78 | 2.89 | 0.06 | 0.72 | 3.06 | 7.89 |
| *PostgreSQL* | 3.95 | 1.32 | 2.64 | 1.73 | 0.00 | 2.45 | 12.09 |
| *TiDB* | 4.18 | 0.82 | 2.73 | 1.41 | 1.77 | 3.73 | 14.64 |

## A.3 Benchmarking

In this application, we show how to find potential optimization opportunities by comparing the serialized query plans across DBMSs using the unified query plan representation. Query optimization is a critical process for DBMSs' performance. To evaluate how effective a query optimization is, existing methods depend on measuring DBMSs' execution time on standard datasets, such as TPC-H [123] and the Join Order Benchmark (JOB) [100]. The execution time shows the overall performance difference between various DBMSs, but cannot provide possible reasons for performance gaps across DBMSs. *UPlan* enables comparing various serialized query plans across DBMSs. Specifically, we collected metrics on the number of operations in DBMSs' query plan representations and show an example of analyzing the difference.

Table 6.8 shows the average number of operations in each category for the query plans of queries from the TPC-H benchmark suite. We omitted the *Consumer* category as we did not encounter any such operations. The relational DBMSs, *MySQL*, *PostgreSQL*, and *TiDB*, have more operations than the non-relational DBMSs, *MongoDB* and *Neo4j*. It is because relational DBMSs have more operations in the *Producer* category. To further explain the difference in the *Producer* category, we looked into query plans and found that each table in relational DBMSs requires at least one operation to read data, while non-relational DBMSs usually read all data in one or two operations. Apart from *MongoDB*, the other four DBMSs have a similar number of *Join* operations. The results also show that the query plans of the queries in the TPC-H benchmark suite usually do not cover all categories of operations due to the limited set of queries. The operations of the *Producer* category are typically expensive for performance, and hence, developers typically

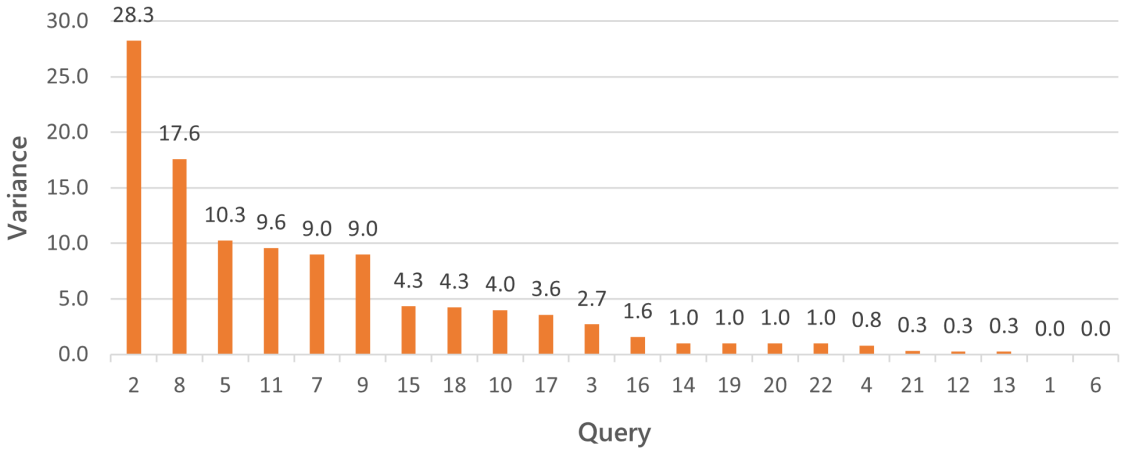


Figure 6.3: Variance of the number of Producer operations for each query in TPC-H benchmark suite across five DBMSs.

aim to analyze and reduce the occurrence of the operations in this category. We explain an example to analyze them as follows.

Figure 6.3 shows the variance of the number of operations in the *Producer* category for each query plan. Among 22 queries, the variances of six queries are more than 5, indicating a significant difference. Queries 2, 8, 5, 7, and 9 have a significant variance due to different data models. For example, for query 2, the DBMSs of relational data models, *MySQL*, *TiDB*, and *PostgreSQL*, have 10, 12, and 9 operations, while the DBMS based on the graph data model, *Neo4j*, has only 1 operation in the *Producer* category. Query 11 has a significant variance due to another potential optimization issue, and we explain it as follows.

Code 6.4 shows query 11[14] from the TPC-H benchmark suite and the corresponding serialized query plans of *PostgreSQL* and *TiDB* using the unified query plan representations in text format. The query references the three tables `PARTSUPP`, `SUPPLIER`, `NATION` twice in the **`FROM`** and **`HAVING`** clauses respectively. *PostgreSQL* uses six table scans, one for each table reference in the original query, while *TiDB* could optimize the query to use only three scans. The table `partsupp` is scanned twice in lines 19 and 21, because *TiDB* reduces the data size of table scanning by retrieving a secondary index before the scan operation. The first scan retrieves indexes only to obtain the row id, which is used for the second scan. Suppose both DBMSs execute the operations in order and the same operation has the same performance

[14] `https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v3.0.1.pdf`

Code 6.4: The unified serialized query plan representations in the text format for query 11 in TPC-H. Underlines represent table names. Predicates in the query and properties in the query plan representations are ignored for brevity.

```
SELECT ... FROM PARTSUPP, SUPPLIER, NATION WHERE ...
HAVING ... > (SELECT ... FROM PARTSUPP, SUPPLIER, NATION WHERE ...) ...;
--------------------------------------------------------------------------------
PostgreSQL:                          TiDB:
Bag->Sort                            Projector->Project
 Folder->Aggregate                    Bag->Sort
  Join->Hash Join                      Folder->Aggregate Hash
   Producer->Full Table                 Projector->Project
   name object: partsupp                 Join->Index Hash
   Executor->Hash Row                     Join->Index Hash
    Join->Hash                             Executor->Collect
     Producer->Full Table                   Producer->Full Table
     name object: supplier                    name object: nation
     Executor->Hash Row                     Executor->Collect Order
      Producer->Full Table                   Producer->Index-only Scan
      name object: nation                      name object: supplier
 Folder->Aggregate                        Executor->Collect Order
  Join->Hash Join                          Producer->Index-only Scan
   Producer->Full Table                    name object: partsupp
   name object: partsupp                    Producer->Id Scan
   Executor->Hash Row                       name object: partsupp
    Join->Hash Join
     Producer->Full Table
     name object: supplier
     Executor->Hash Row
      Producer->Full Table
      name object: nation
```

overhead in both DBMSs, then the query plan with three table scans is more efficient than the query plan with six table scans. *UPlan* enables this analysis and provides actionable insight for DBMS developers to improve query performance by reducing three repeated table scans. We reported this issue to the *PostgreSQL* developers, who confirmed that this case indicates an unsupported optimization by *PostgreSQL*. One developer considered how this could be supported—"*[...] maybe there is some easy way to hook this into the same code used by GROUPING SETS [...]*".[15]

Comparing the unified query plan representation provides actionable insights.

[15] https://www.postgresql.org/message-id/flat/CAMkU%3D1yL%2Bg0VHM_OEgkhMiq6AbOOMN3zVXYK6pfs2HgQEsFb0g%40mail.gmail.com#087581bdd2c9fc164931e0140b414f26

## 6.5 Discussion

**Paths to adoption.** Developers can make use of the unified query plan representation by customized converters to convert original serialized query plans into the unified query plan representation. For all three applications we showed, we implemented five customized converters, each of which has around 200 lines of code, within one week only. If the converters can be implemented by DBMS developers or experts, who are well-versed in the query plans, it is plausible that a higher-quality converter could be developed in a shorter time. We hope that DBMSs will directly expose query plans in the unified representation in the long term, thus avoiding a conversion.

**Additional use cases.** We envision several additional use cases that are enabled by our unified query plan representations. Toward a comprehensive evaluation of query optimization, additional metrics could be explored, such as similarity on tree structures [207], to compare different DBMSs' query plans using our unified query plan representation. To find issues in query optimization, we can apply differential testing [114] or other methods to compare the unified query plan representations among DBMSs. Query optimization approaches based on machine learning have been proposed that take query plans as input and output suggestions for indexes [40], views [212], and join orders [112, 211], so our unified query plan representation would allow exchanging training data in different DBMSs to improve the performance of models.

**Threats to validity.** Our study faces several threats to validity, which denotes the trustworthiness [126, 154] of the results, and to what extent they are unbiased. A major concern is the degree to which the data and analysis depend on the specific researchers. We followed the best practice of triangulation [154], which refers to taking multiple perspectives toward the same object, to increase reliability. For data triangulation, we collected data from multiple sources: documentation, source code, and third-party applications. For observer triangulation, one author conducted the study, and another author validated the findings against the raw data. For methodological triangulation, we used a qualitative method to analyze

query plan representations, and a quantitative method to examine and classify the three conceptual components of query plan representations. Furthermore, we have made the process of data collection and analysis publicly available, along with comprehensive study results presented in the supplementary materials of this paper. Another concern is the degree to which our results can be generalized to and across the query plan representations of other DBMSs. We selected representative DBMSs of various data models: relational, document, graph, and time series. The last concern is the degree to which the study really assesses the research questions we aim at. We collected the data ourselves, and our analysis of the semantics may be inconsistent with the intentions of the developers that implemented the query plans.

## 6.6 Conclusion

We have presented an exploratory case study to investigate how query plan representations are in nine widely-used DBMSs. Our study has shown that query plan representations share conceptual components among different DBMSs: operations, properties, and formats. Based on the study, we designed the unified query plan representation to reduce the effort to build applications based on query plans. We implemented a reusable library *UPlan* to maintain the unified representation, and evaluated it on five DBMSs. The results show that existing DBMS-specific visualization tools could support at least five DBMSs by using *UPlan* with only moderate implementation effort, and existing testing methods can be efficiently adopted. Additionally, *UPlan* also enables comparing query plans in different DBMSs, which provides actionable insights. This paper provides a comprehensive study of query plan representations, and can be used as a reference for other research on serialized query plans. We believe the unified query plan representation provides more opportunities to research serialized query plans in the future.

# Chapter 7

# Related Work

In this chapter, I first discuss the related work on finding logic bugs and performance issues in DBMSs, which is the focus of this thesis. Then I discuss the research about query plans. Last, from a broader perspective, I also discuss other reliability problems related to DBMSs.

## 7.1 Techniques for Finding Bugs

Various techniques for finding bugs in DBMSs have been explained in section 2.2, and we discuss the related work on these techniques in this section.

### 7.1.1 Metamorphic Testing

The most related research is about metamorphic testing. Metamorphic testing mostly finds logic bugs and performance issues, and has been applied successfully in various domains, such as web applications [28] and compilers [98]. For DBMSs, *NoREC* [149] and *TLP* [150] oracles were proposed as a metamorphic testing method for detecting logic bugs in `SELECT` statements. These three oracles were implemented in *SQLancer* and have found hundreds of bugs. *DQE* [162] adopted metamorphic testing to find logic bugs in `UPDATE` and `INSERT` statements. Jiang *et al.* [84] adopted metamorphic testing to find logic bugs in graph database systems. Liu *et al.* proposed *AMOEBA* [107], which compares the execution time of a semantically equivalent pair of queries to identify an unexpected slowdown. In this thesis, *QPG* can be paired with these metamorphic testing methods, and *CERT* is a metamorphic testing approach for finding performance issues with the *cardinality restriction monotonicity*

property.

### 7.1.2 Differential Testing

Much work is about applying differential testing for DBMSs. Slutz *et al.* [161] compared the output of executing the same test case on different DBMSs to find logic bugs. Ghit *et al.* [54] compared the output of executing the same test case on different versions of Spark[1] to find regression bugs. Yagoub *et al.* [202] proposed to use differential testing to find performance regression bugs before deploying any change into a production environment. Jung *et al.* proposed *APOLLO* [89], which compares the execution times of a query on two versions of a database system to find performance regression bugs. *Mozi* [105] examined the equivalence of executing the same query under different configurations of DBMSs. *DQP* is a differential testing method described in this thesis to compare executing different query plans of the same query. *DQP* has commonalities with *Mozi* as some configurations can affect query plans, but the core contribution of *DQP* is the insight that a simple method can outperform a complicated method.

### 7.1.3 Fuzzing

Fuzzing has gained increased attention, because of the success of the coverage-guided grey-box fuzzers such as *AFL* [180] and *LibFuzzer* [181], which mutate inputs aiming to maximize code coverage for automatically finding memory-related bugs. *AFL*, *LibFuzzer*, and other fuzzers enable the large-scale continuous fuzzing service *OSS-Fuzz* [159], which automatically tests open-source programs, including *SQLite*[2] and other DBMSs, and generates bug reports. A challenge of mutation is that it easily generates invalid inputs. To generate grammar-valid inputs, *SQLSmith* [182] used predefined grammars to generate valid inputs and has found over 100 bugs in widely-used DBMSs. To generate semantically valid inputs, Fu *et al.* [51] proposed to incorporate target DBMS metadata information to guide the input generation. *Squirrel* [219] proposed to analyze data dependencies among statements to generate both grammar-correct and semantic-correct inputs. Some researchers proposed complex oracles to find more kinds of bugs in fuzzing. *SQLRight* [106] combined

[1] `https://spark.apache.org/`

[2] `https://github.com/google/oss-fuzz/tree/41ee0518/projects/sqlite3`

grey-box fuzzing and metamorphic testing to find logic bugs using code coverage as guidance for test case generation. In this thesis, *QPG* is compatible with these fuzzing methods, and *CERT* focuses on finding logic bugs and performance issues.

### 7.1.4 Formal Verification

Different from the above testing techniques, which can show the presence of bugs, formal verification is a technique to show the absence of bugs by verifying a system with respect to a formal specification. Negri *et al.* [125] proposed a formal semantic for SQL language. Benzaken *et al.* [13] proposed a Coq formalization of SQL to enable formal verification on relational databases. Guagliardo *et al.* [66] further improved the formalization by considering bag semantics and nulls. Then, Guagliardo *et al.* [68] proposed a Codd semantic for NULL to distinguish different Nulls. Malecha *et al.* [108] implemented a verified relational DBMS, whose specifications were written and verified in Coq. Diana *et al.* [39] verified part of the SQL specifications by model checking. Tan *et al.* [166] proposed to verify the serializability of executions in key-value stores by Satisfiability Modulo Theories (SMT) solvers, which is a method to determine whether a mathematical formula is satisfiable, and the formula is an abstraction from the target system. Tang *et al.* [167] proposed to use a method based on constraint solving to validate joining algorithms against ground-truth results. Apart from verifying results, Chu *et al.* [35] used constraint-solving methods to decide equivalent queries. In contrast to formal verification, this thesis focuses on the testing technique.

### 7.1.5 Performance Benchmarking

Benchmarking is a common practice to identify performance regressions, and to continuously improve the DBMSs' performance on a set of benchmarks. *TPC-H* [123] and *TPC-DS* [177] are the most popular benchmarks and are considered the industry standard. Boncz *et al.* studied and identified 28 "chokepoints" (*i.e.*, optimization challenges) of the *TPC-H* benchmark [20]. Poess *et al.* modified and analyzed the *TPC-DS* benchmark [138] to measure SQL-based big data systems. Karimov *et al.* proposed a benchmark for stream-data-processing systems [90]. Boncz *et al.* proposed an improved *TPC-H* benchmark,*JCC-H* [19], which introduces *Join-*

*Crossing-Correlations* (*JCC*) to evaluate the scenarios where data in one table can affect the behaviors of operations involving data in other tables. Leis *et al.* proposed the *Join Order Benchmark (JOB)* [100], which uses more complex join orders. Raasveldt *et al.* described common pitfalls when benchmarking DBMSs and demonstrated how they can affect a DBMS's performance [142]. In this thesis, *CERT* is complementary to benchmarking; while benchmarking focuses on workloads deemed relevant for users, *CERT* can find performance issues through the lens of cardinality estimation even on previously unseen workloads.

### 7.1.6 Query Generation

Targeted and random generations are two major directions in query generation for testing methods. As for targeted query generation, Bati *et al.* [11] proposed to incorporate execution feedback, such as code coverage, for guiding query generation to reach a specific code location. Khalek *et al.* [1] used a solver-backed approach to generate syntactically and semantically correct queries. Generating queries that satisfy cardinality constraints has been proven to be computationally hard, which is why heuristic algorithms were proposed [25, 118]. As for random query generation, *SQLSmith* [182], mentioned in Subsection 7.1.3, uses a predefined grammar to randomly generate semantic valid queries. *APOLLO* [89], mentioned in Subsection 7.1.2, also uses predefined grammar to generate queries for finding regression performance issues. *Squirrel* [219] and *SQLRight* [106], both of which are mentioned in Subsection 7.1.3, use a mutation-based method to randomly generate new queries, but such approaches are prone to generating queries that are semantically invalid. In this thesis, *QPG* uses a grammar-based random generation method for generating valid queries.

### 7.1.7 Database Generation

Similarly, targeted and random generations are two major directions in database generation. As for targeted database generation, Binnig *et al.* [17] proposed to use symbolic execution to specify constraints and generate databases that satisfy the constraints. Binnig *et al.* [16] proposed to reverse a given query and database schema to generate databases. Kersten *et al.* [91] proposed to simulate real-world

databases by profiling databases without accessing the data in databases. As for random database state generation, Gray *et al.* [62] proposed to quickly generate billions-record databases using parallel algorithms. Coverage-based methods [219, 106], mentioned in Subsection 7.1.3, generate new databases by mutating given SQL statements that are used to create the database. Compared with these methods, *QPG* uses query plans as guidance to generate more diverse databases for efficiently finding logic bugs.

## 7.2 Query Plans

Related work about query plans has been proposed for understanding the performance of queries, while I introduce it to the domain of automatic testing for DBMSs.

### 7.2.1 Query Plans in Testing

There is no direct work to use query plans for automatically finding bugs, but researchers used it to improve the quality of query optimizations. Gu *et al.* [64] proposed measuring the accuracy of query optimizations by forcing the generation of multiple alternative query plans for each test case, timing the execution of all alternatives, and ranking the plans by their effective costs with the goal of comparing this ranking with the ranking of the estimated cost. Pasupuleti *et al.* [132] proposed to mitigate bugs in query optimizations by automatically switching query plans. In this thesis, *QPG* uses query plans to guide test case generation, while *CERT* uses query plans to find performance issues.

### 7.2.2 Manipulating Query Plans

Various techniques have been proposed to manipulate query plans. AEM [132] uses query hints to switch query plans for bypassing bugs in run-time. PgCuckoo [77] provides a plugin for PostgreSQL, so that PostgreSQL can execute arbitrary query plans. However, a significant challenge, as claimed in the PgCuckoo paper, is that manually manipulating query plans has a high invalid rate as the operations in a query plan typically have dependencies on each other. TAQP [64] uses query hints to switch query plans and measures execution time to check whether the query plan

chosen by query optimizers is the optimal one. Compared with these methods, *DQP* adopts a black-box manner to manipulate query plans by query hints and system variables for finding logic bugs.

### 7.2.3 Query Optimization

Researchers and practitioners have invested decades of effort into improving query optimization. An important optimization is the join order. Neumann *et al.* [127] proposed to simplify the query's join graph for a complex join to optimize the join order. DeHaan *et al.* [38] designed an algorithm to combine enumeration and search for optimizing the join order. Regarding index optimization, Bayer *et al.* [12] proposed B-trees, which are widely used for index optimizations. Optimization models are critical to estimating performance overhead for a query plan. Akdere *et al.* [4] proposed a machine learning model to estimate performance overhead by extracting plans and operators. Wu *et al.* [197] improved the performance estimation by calibrating the estimated performance overhead of a system resource unit, such as the I/O cost of accessing a page, by profiling tool. Predicates can be efficiently executed in different locations of a query plan. Yan *et al.* [203] proposed to use symbolic execution to verify which candidate predicates can be correctly pushed down to another operator for optimizing predicate execution. Apart from obtaining optimal query plans, Giceva *et al.* [55] proposed to efficiently execute query plans on multi-core CPUs, and Paul *et al.* [134] proposed to execute query plans on GPUs. Apart from improving query optimizations, some work evaluates query optimizations in practice. Gu *et al.* [64] measured the accuracy of query optimizations by comparing the estimated performance overhead and actual execution time of a query plan. Leis *et al.* [100] investigated the impact of the components of a query optimizer on performance and found cardinality estimation to be the most important sub-component that affects query optimization. In this thesis, I focus on improving testing techniques via query plans, instead of improving query optimizations.

### 7.2.4 Cardinality Estimation

Various approaches have been proposed to improve the accuracy of cardinality estimation, which is one of the most important parts of query optimization. Han

*et al.* [73] comprehensively evaluated various algorithms for cardinality estimation, which describes some of the subsequent important approaches. *PostgreSQL* [186] and *MultiHist* [140] applied one-dimensional and multi-dimensional histograms to estimate cardinality. Similarly, *UniSample* [101, 217] and *WJSample* [103] used sampling-based methods to estimate cardinality. Apart from these traditional approaches, machine learning-based methods have gained attention recently. *MSCN* [94, 210], *LW-XGB* [41], and *UAE-Q* [199] used deep neural networks, classic lightweight regression models and deep auto-regression models to learn to map each query to its estimated cardinality directly. In addition, *NeuroCard* [209], *BayesCard* [198], *DeepDB* [76], and *FSPN* [200, 220] utilized deep auto-regression models and three probabilistic graphical models BN, SPN, and FSPN to predicate the data distribution for cardinality estimation. In this thesis, *CERT* is a black-box technique that inspects cardinality estimation to find performance issues.

### 7.2.5 Applications Based on Serialized Query Plans

Several applications based on serialized query plans exist. QE3D [155] visualizes distributed serialized query plans for an intuitive understanding and analysis. Machine learning algorithms have utilized serialized query plans for query optimization. Yuan *et al.* [212] used machine learning to select optimal views. Yu *et al.* [211] used reinforcement learning to determine the join order. Marcus *et al.* [111] and Ryan *et al.* [113] applied machine learning algorithms to generate query plans. Zhao *et al.* [216] used machine learning to convert serialized query plan representations to vector representations to facilitate other machine learning algorithms. In *UPlan*, we propose a unified query plan representation, which reduces the effort to build the applications based on serialized query plans.

## 7.3 Other Reliability Problems in DBMSs

DBMSs are complex systems, which suffer from more types of reliability problems, other than logic bugs and performance issues. In this section, I discuss the related research about other reliability problems.

### 7.3.1 Environmental Reliability

DBMSs do not run in isolation and rely on the underlying Operating Systems (OS) to provide storage, networking, computation, and other services. Environments are complicated, so some work investigated whether environments can affect DBMSs' robustness. Zheng *et al.* [218] studied the impact of power fault on DBMS reliability, and found that all studied eight DBMSs exhibit erroneous behavior under power fault. Pillai *et al.* [137] studied the impact of the faults of file systems on DBMS reliability and found 60 vulnerabilities. To improve environmental robustness, both formal verification and random testing have been widely researched. Chen *et al.* [30] used formal methods to prove the correctness of the FSCQ file system. Bornholt *et al.* [21] deployed a lightweight formal method to validate a key-value storage node in the Amazon cloud storage engine. Mohan *et al.* [120] proposed to enumerate workload in a bounded space to test whether file systems correctly recover to a recent state after a power-loss crash. Kim *et al.* [92] used the fuzzing technique to find non-crash bugs in file systems. In this thesis, both *QPG* and *CERT* work in a constant environment assuming no fault in the environment.

### 7.3.2 Configuration Reliability

DBMSs, as well as other complex systems, introduce an increasing number of configuration options to provide flexibility, but this mechanism also affects DBMSs' robustness leading to memory-related bugs, logic bugs, and performance issues. Toman *et al.* [169] adopted dynamic analysis to find incorrect usage of configuration options. Sun *et al.* [164] proposed to reuse official test cases and deploy them in real-world production to find logic bugs, and Cheng *et al.* [34] further improved this method by prioritizing test cases before deploying. Wang *et al.* [176] designed a method to find logic bugs in configurations by comparing consistency between loading configurations at the start-up stage and updating them on-the-fly. The bugs in configurations have shown a significant impact on performance. Han *et al.* [72] studied the correlation between performance issues and configurations, and found that more than half of previously found performance issues are due to misconfigurations or are triggered by specific configurations. He *et al.* [74] adjusted configurations and checked whether the performance is expected to find configuration-related

performance issues. Some works focus on how to better analyze and fix the bugs related to configurations. Wen *et al.* [195] applied machine learning algorithms to predicate whether a bug report is related to configurations. Xu *et al.* [201] adopted a static taint tracking method to analyze source code and automatically generate the code for checking configuration correctness. In this thesis, both *QPG* and *CERT* work in the default configuration options of DBMSs assuming no fault in the configurations.

### 7.3.3 Transactional Reliability

Another research topic focuses on the transaction, which is a feature of DBMSs that executes a set of operations as a single unit [14], and must guarantee the properties: atomicity, consistency, isolation, and durability (ACID). Transaction is a complex procedure, so many methods have been proposed to find logic bugs that violate the ACID properties. Cerone *et al.* [26] proposed a framework to specify consistency models for transactions uniformly, which facilitates the test oracle constructions in manual and automatic checking methods. Biswas *et al.* [18] investigated the complexity of checking transactional correctness, and found consistency models are polynomial-time checkable while prefix consistency is NP-complete. Kingsbury *et al.* [93] recorded and analyzed execution traces to find the violations of the ACID properties by comparing them against predefined consistency models. Tan *et al.* [166] improved the throughput of these checking tools by hardware accelerating and new encoding of transactional patterns. Apart from randomly generated transactions, Huang *et al.* [78] adopted Satisfiability Modulo Theories (SMT) solvers to check whether a violation is feasible. To find bugs in complex transactions, Jiang *et al.* [83] instrumented statements to capture statement dependencies for generating complex transactions. In this thesis, *QPG* and *CERT* do not specifically generate and test transactional correctness but are compatible with the statements of controlling the transactional process in user interfaces.

### 7.3.4 DBMS Application Reliability

Many applications are built on top of DBMSs, and their reliability is also a related research topic. Many researchers focus on generating diverse database states

to test the applications more thoroughly. Pan *et al.* [131] proposed a method to generate database states by using dynamic symbolic execution to simulate the interaction between the application and the associated DBMS. Agrawal *et al.* [3] proposed to generate specific database states by using SMT solvers. Yan *et al.* [204] proposed to mutate existing database states with the guidance of branch coverage of the application. Apart from test case generation, Gligoric *et al.* [58] adopted the model checking method to find concurrency bugs in web applications that use DBMSs. In this thesis, I focus on the reliability of the DBMSs, not the applications built on top of them.

### 7.3.5 Bug Minimization and Deduplication

Bug minimization, which minimizes the test case size, and bug deduplication, which drops repeated test cases, are two research directions to reduce the effort of debugging. Automatically bug-finding methods, such as *SQLancer* [150, 149, 151] and *SQLSmith* [182], typically produce a lot of long and duplicate bug-inducing test cases, which significantly burden the workload of debugging work by developers. Most related work focuses on general test cases, instead of specifically DBMS test cases. Zeller *et al.* [215] proposed delta debugging, which is a technique to gradually remove some portions of test cases while bugs are still observable. Wang *et al.* [175] improved the efficiency of dealt debugging by removing the most likely redundant portions according to a probabilistic estimation model. Apart from research works, some industry tools have been widely used. C-reducer [3] is a tool that implements the delta debugging algorithm and is embedded in the C/C++ compiler testing work CSmith [208]. SQLReduce [4] is a tool for minimizing SQL test cases specifically by trimming syntax trees and is developed by *PostgreSQL*. As for bug deduplication, the stack trace is one of the widely-used metrics to distinguish duplicate memory-related bugs [156]. Vasiliev *et al.* [172] proposed to use machine-learning algorithms to calculate the similarity of stack traces for bug deduplication. Rodrigues *et al.* [152] improved the efficiency of stack trace comparison by matching identical frames in two stack traces independently. Another method for distinguishing duplicate bugs is commit bisection, which deduplicates bugs according to the firstly introduced commit

[3] `https://github.com/csmith-project/creduce`
[4] `https://github.com/credativ/sqlreduce`

and has already been used in *SQLite* [5]. Abreu *et al.* [2] experimentally evaluated the automatic commit bisection, and found it can help developers deduplicate and fix bugs 2.23 times faster. In this thesis, I focus on bug-finding techniques and directly use existing techniques to minimize and duplicate bugs.

## 7.4 Research Methodologies

Various reserach methodologies are adopted in this thesis, and we discuss them in this section.

### 7.4.1 Simple Over Complex

Several existing works in other domains adopt a similar methodology as *DQP* to propose a simple technique that can outperform an existing sophisticated one. Kali [141] uses a simple method that only deletes functionality and outperforms previous sophisticated techniques for automatically generating software patches. Fu *et al.* [52] demonstrated that a simple tuned Support Vector Machine (SVM) can outperform a sophisticated Convolutional Neural Network (CNN) algorithm. This paper was also inspired by Kali.

### 7.4.2 Standardization

Multiple works were proposed to standardize DBMSs, similar to *UPlan*. Feng *et al.* [48] proposed a unified architecture to reduce the implementation effort for in-RDBMS analytics. Mitschang [119] proposed a unified view of design data and knowledge representation when supporting database systems to non-standard applications, such as Computer-Aided Design (CAD). Ginsburg *et al.* [56] proposed a unified approach to query sequenced data. Gueidi *et al.* [69] proposed a unified modeling method for Non-relational DBMSs (NoSQL) to facilitate the applications on NoSQL. Compared to these works, *UPlan* propose a unified framework for the query plan representations in DBMSs.

[5] `https://news.ycombinator.com/item?id=37255022`

# Chapter 8

# Conclusion

In this chapter, I summarize the thesis and discuss several possible future research directions.

## 8.1 Summary

In this thesis, I have proposed leveraging query plans to advance state-of-the-art testing methods for efficiently finding logic bugs and performance issues in DBMSs. First, I proposed a novel approach *CERT* to identify performance issues through the lens of cardinality estimation in DBMSs. The key idea is to, given a query, derive a more restrictive query and validate that the DBMSs' estimated cardinalities that the original query has at least as great estimated cardinality as the more restrictive query; I refer to this property as *cardinality restriction monotonicity*. *CERT* has found 13 unique previously-unknown bugs. Second, I proposed the method *DQP*, a simple alternative test oracle to a state-of-the-art and complex test oracle *TQS*. The key idea is that different query plans of the same query should return the same result. Otherwise, a logic bug is found. Although *DQP* is simple and straightforward, it found 26 unique and previously unknown bugs that were overlooked by *TQS*, without complex calculations. Third, I proposed the concept of *QPG* for generating diverse test cases for efficiently finding logic bugs. The core idea is that covering more unique query plans might increase the likelihood of finding logic bugs. *QPG* enabled us to find 53 unique, previously unknown bugs in widely-used and extensively-tested DBMSs—*SQLite*, *TiDB*, and *CockroachDB*. Last, observing query plan representations are specific to DBMSs making it challenging to widely apply the above testing methods, I conducted an exploratory case study of query

plan representations and proposed a unified query plan representation *UPlan*. By integrating *UPlan*, the above testing methods can be directly applied to a large number of targets. These methods are black-box methods that are applicable without modifying the source code of target systems and enable finding difficult-to-trigger bugs.

## 8.2 Future work

Current research exploration of leveraging query plans in the testing domain is still in the early stage. I believe query plans have more potential to be discovered for improving existing testing methods, and this area of research needs more effort. Specifically, I have the following potential future research directions:

**Evaluating the quality of test suites.** A high-quality test suite assures that the target program is tested thoroughly. Evaluating the quality of a test suite is challenging due to the unknown space of the target program's behaviors. A common criterion is code coverage [59], where higher code coverage implies more behaviors tested. However, for DBMSs, 100% code coverage cannot guarantee all behaviors are covered.[1] I found that some bug-inducing test cases have covered code repetition, but uncovered query plans than previous test cases [85]. Thus, I envision that query plan coverage is a more accurate metric to evaluate the quality of test suites. Across multiple test suites, a higher-quality test suite has higher query plan coverage. Within the same test suite, a high-quality test case has more distinct parts of the query plan from others.

**Detecting bugs in various types of DBMSs.** Apart from the DBMSs of the relational model I tested, The DBMSs of more types of data models, such as graph, document, and key-value, are also widely used[2] and are affected by bugs. I believe my research can be extended to test them. As a specific example, MongoDB, a document database that is used for flexible data structures, has a similar concept of query plans,[3] which also expose internal execution information in a black-box manner.

[1] `https://www.sqlite.org/testing.html#mcdc`
[2] `https://db-engines.com/en/ranking_categories`
[3] `https://www.mongodb.com/docs/manual/core/query-plans`

Therefore, test oracles based on query plans might detect logic bugs in MongoDB. Another type of DBMS, the distributed DBMS, leaves more room for bugs, because it is deployed in clusters in which nodes interact with each other to process SQL statements. Detecting logic bugs in distributed DBMSs is challenging [70], because the number of potential execution sequences is enormous, and no ground-truth answers can be used to validate results automatically. An existing method Jepsen[4] checks linearizability, but cannot identify incorrect results. Query plans expose the information of various possible and valid execution sequences for the same query across multiple nodes.[5] Therefore, I believe we can detect logic bugs in distributed DBMSs by measuring the inconsistencies across the execution results of multiple equivalent query plans for the same query.

**Simulating DBMS workloads in production environments.** To continuously improve DBMSs, developers require real-world execution traces for analysis, such as which operation degrades performance and how bugs are triggered. Considering privacy, few customers of DBMSs are willing to share their database schema, data samples, and queries with developers [91], especially in production environments. Query plans include concrete execution steps of queries on databases, and the operations in query plans are affected by databases. For example, when accessing a table, the operation `IndexLookupReader` is only used when the indexes in queried tables do not completely cover calculated columns, otherwise, the operation `IndexReader` is used.[6] Therefore, my intuition is that we can infer databases from query plans. Considering the same example above, if the operation `IndexLookupReader` is used in the query plan, we infer that part of the calculated columns have indexes. Given a query plan, we want to infer several possible queries, database schema, and data samples that execute the same query plan, then developers can execute them to simulate customers' environments for analysis.

**Verification of query optimizers.** I believe that query plans can contribute to the verification of query optimizers. As explained in section 2.2, existing verification methods face the challenges of state abstraction and scalability problems. Query

[4] `https://jepsen.io`
[5] `https://www.cockroachlabs.com/docs/stable/explain.html#distsql-option`
[6] `https://docs.pingcap.com/tidb/dev/choose-index#access-tables`

plans provide a compact abstraction of internal execution logic and include a countable number of states. We can construct specifications according to the semantics of query plans, and evaluate them by model checking or run-time verification [10] methods. For example, the hash join operation in *CockroachDB*[7] is expected to have a left child operation with a smaller estimated cardinality than that of the right child operation, and we can encode this property into specifications. This method could reduce the effort and improve the accuracy of the state abstraction. This method is also practical, because query plans can be obtained in a black-box manner.

**Debugging DBMSs.** To debug bugs in DBMSs, existing debuggers, such as *Habitat* [63], are usually applied at the language (*i.e.* SQL) level. Developers still need to manually examine source code, with one study suggesting around 95% [53]. Query plans include internal execution information, so I believe we can use query plans to automatically debug the execution steps of DBMSs. Given a query, we may add breakpoints to the operations on its query plan, so the query plan is executed step by step and we examine the behaviors, such as the estimated cardinality, of each step. Unlike general debugging tools, such as gdb[8] for C/C++ programs, we debug DBMSs at a semantic level, instead of a source code level aiming to facilitate understanding the behaviors of DBMSs. Note that existing DBMSs do not support such a mechanism to add breakpoints into query plans, and we may need additional effort to implement such a feature for query plans.

**Vision: Internal representations in testing.** Existing testing methods typically consider inputs, outputs, and domain-agnostic information, such as code coverage, for testing, while this thesis put forward that for the DBMSs, internal representations are useful for efficiently and effectively testing complex program space. Since many important kinds of systems, such as compilers,[9] use internal intermediate representations, I envision that, in general, internal representations can be a useful testing element, in addition to inputs and outputs. We could extend our methods to more general methods for testing other systems, such as identifying performance

[7] `https://www.cockroachlabs.com/docs/stable/joins.html#hash-joins`
[8] `https://www.gnu.org/savannah-checkouts/gnu/gdb/index.html`
[9] `https://llvm.org/docs/LangRef.html`

issues in compilers by inspecting internal representations. At the same time, the requirements of testing can motivate the design of internal representations. For example, if a data read operation should always consume less memory than a data write operation, we can request developers to include the memory consumption information in internal representations. Overall, I believe our work represents a conceptual advance in testing. We expect that the research community will take the work in this thesis forward, to further understand, utilize, and improve internal representations in testing.

# Bibliography

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

[2] R. Abreu, F. Ivančic, F. Nikšic, H. Ravanbakhsh, and R. Viswanathan, "Reducing time-to-fix for fuzzer bugs", in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2021, pp. 1126–1130.

[3] P. Agrawal, B. Chandra, K. V. Emani, N. Garg, and S. Sudarshan, "Test data generation for database applications", in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, IEEE, 2018, pp. 1621–1624.

[4] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik, "Learning-based query performance modeling and prediction", in *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, A. Kementsietsidis and M. A. V. Salles, Eds., IEEE Computer Society, 2012, pp. 390–401. [Online]. Available: https://doi.org/10.1109/ICDE.2012.64.

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

[6] R. Angles and C. Gutierrez, "Survey of graph database models", *ACM Comput. Surv.*, vol. 40, no. 1, 1:1–1:39, 2008. [Online]. Available: https://doi.org/10.1145/1322432.1322433.

[7] T. J. Anih, C. A. Bede, and C. F. Umeokpala, "Detection of anomalies in a time series data using influxdb and python", *CoRR*, vol. abs/2012.08439, 2020. arXiv: 2012.08439. [Online]. Available: https://arxiv.org/abs/2012.08439.

[8] R. Arora and R. R. Aggarwal, "Modeling and querying data in mongodb", *International Journal of Scientific and Engineering Research*, vol. 4, no. 7, pp. 141–144, 2013.

[9] T. Bach, A. Andrzejak, C. Seo, C. Bierstedt, C. Lemke, D. Ritter, D. Hwang, E. Sheshi, F. Schabernack, F. Renkes, G. Gaumnitz, J. Martens, L. Hömke, M. Felderer, M. Rudolf, N. Jambigi, N. May, R. Joy, R. Scheja, S. Schwedes, S. Seibel, S. Seifert, S. Haas, S. Kraft, T. Kroll, T. Scheuer, and W. Lehner, "Testing very large database management systems: The case of SAP HANA", *Datenbank-Spektrum*, vol. 22, no. 3, pp. 195–215, 2022. [Online]. Available: https://doi.org/10.1007/s13222-022-00426-x.

[10] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, "Introduction to runtime verification", *Lectures on Runtime Verification: Introductory and Advanced Topics*, pp. 1–33, 2018.

[11] H. Bati, L. Giakoumakis, S. Herbert, and A. Surna, "A genetic approach for random testing of database systems", in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07, Vienna, Austria: VLDB Endowment, 2007, pp. 1243–1251, ISBN: 978-1-59593-649-3.

[12] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices", in *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, 1970, pp. 107–141.

[13] V. Benzaken, E. Contejean, and S. Dumbrava, "A coq formalization of the relational data model", in *Programming Languages and Systems: 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings 23*, Springer, 2014, pp. 189–208.

[14] P. A. Bernstein and E. Newcomer, "Principles of transaction processing", Morgan Kaufmann, 2009.

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

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

[17] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu, "Qagen: Generating query-aware test databases", in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '07, Beijing, China: Association for Computing Machinery, 2007, pp. 341–352, ISBN: 9781595936868.

[18] R. Biswas and C. Enea, "On the complexity of checking transactional consistency", *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–28, 2019.

[19] P. A. Boncz, A. G. Anadiotis, and S. Kläbe, "JCC-H: adding join crossing correlations with skew to TPC-H", in *Performance Evaluation and Benchmarking for the Analytics Era - 9th TPC Technology Conference, TPCTC 2017, Munich, Germany, August 28, 2017, Revised Selected Papers*, R. Nambiar and M. Poess, Eds., ser. Lecture Notes in Computer Science, vol. 10661, Springer, 2017, pp. 103–119. [Online]. Available: https://doi.org/10.1007/978-3-319-72401-0%5C_8.

[20] P. A. Boncz, T. Neumann, and O. Erling, "TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark", in *Performance Characterization and Benchmarking - 5th TPC Technology Conference, TPCTC 2013, Trento, Italy, August 26, 2013, Revised Selected Papers*, R. Nambiar and M. Poess, Eds., ser. Lecture Notes in Computer Science, vol. 8391, Springer, 2013, pp. 61–76. [Online]. Available: https://doi.org/10.1007/978-3-319-04936-6%5C_5.

[21] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, *et al.*, "Using lightweight formal methods to validate a key-value storage node in amazon s3", in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 836–850.

[22] J. Bosak, "Xml, java, and the future of the web", *World Wide Web J.*, vol. 2, no. 4, pp. 219–227, 1997. [Online]. Available: `http://metalab.unc.edu/pub/sun-info/standards/xml/why/xmlapps.htm`.

[23] F. Bousquet, R. Lifran, M. Tidball, S. Thoyer, and M. Antona, "Editorial introduction", *J. Artif. Soc. Soc. Simul.*, vol. 4, no. 2, 2001. [Online]. Available: `http://jasss.soc.surrey.ac.uk/4/2/0.html`.

[24] N. Bruno, S. Chaudhuri, and D. Thomas, "Generating queries with cardinality constraints for dbms testing", *IEEE Trans. on Knowl. and Data Eng.*, vol. 18, no. 12, pp. 1721–1725, Dec. 2006, ISSN: 1041-4347.

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

[26] A. Cerone, G. Bernardi, and A. Gotsman, "A framework for transactional consistency models with atomic visibility", in *26th International Conference on Concurrency Theory (CONCUR 2015)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

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

[28] W. K. Chan, S. C. Cheung, and K. R. Leung, "A metamorphic testing approach for online testing of service-oriented software applications", *International Journal of Web Services Research (IJWSR)*, vol. 4, no. 2, pp. 61–81, 2007.

[29] S. Chaudhuri, "An overview of query optimization in relational systems", in *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, A. O. Mendelzon and J. Paredaens, Eds., ACM Press, 1998, pp. 34–43. [Online]. Available: `https://doi.org/10.1145/275487.275492`.

[30] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, "Using crash hoare logic for certifying the fscq file system", in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 18–37.

[31] T. Y. Chen, S. C. Cheung, and S. Yiu, "Metamorphic testing: A new approach for generating next test cases", *CoRR*, vol. abs/2002.12543, 2020. arXiv: 2002.12543. [Online]. Available: https://arxiv.org/abs/2002.12543.

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

[33] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers", in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 197–208.

[34] R. Cheng, L. Zhang, D. Marinov, and T. Xu, "Test-case prioritization for configuration testing", in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 452–465.

[35] S. Chu, C. Wang, K. Weitz, and A. Cheung, "Cosette: An automated prover for sql." In *CIDR*, 2017.

[36] E. M. Clarke, "Model checking", in *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17*, Springer, 1997, pp. 54–56.

[37] E. F. Codd, "A relational model of data for large shared data banks", *Commun. ACM*, vol. 13, no. 6, pp. 377–387, 1970. [Online]. Available: https://doi.org/10.1145/362384.362685.

[38] D. DeHaan and F. W. Tompa, "Optimal top-down join enumeration", in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007, pp. 785–796.

[39] R. Diana, H. Marques-Neto, L. Zarate, and M. Song, "A symbolic model checking appproach to verifying transact-sql", in *2012 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, IEEE, 2012, pp. 1735–1741.

[40] B. Ding, S. Das, R. Marcus, W. Wu, S. Chaudhuri, and V. R. Narasayya, "AI meets AI: leveraging query executions to improve index recommendations", in *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds., ACM, 2019, pp. 1241–1258. [Online]. Available: https://doi.org/10.1145/3299869.3324957.

[41] A. Dutt, C. Wang, A. Nazi, S. Kandula, V. R. Narasayya, and S. Chaudhuri, "Selectivity estimation for range predicates using lightweight models", *Proc. VLDB Endow.*, vol. 12, no. 9, pp. 1044–1057, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p1044-dutt.pdf.

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

[43] L. Eder, "Say no to venn diagrams when explaining joins", https://blog.jooq.org/say-no-to-venn-diagrams-when-explaining-joins/, Accessed: 2022-11-15, 2022.

[44] M. Egea, C. Dania, and M. Clavel, "Mysql4ocl: A stored procedure-based mysql code generator for OCL", *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, vol. 36, 2010. [Online]. Available: https://doi.org/10.14279/tuj.eceasst.36.445.

[45] M. Elhemali, C. A. Galindo-Legaria, T. Grabs, and M. M. Joshi, "Execution strategies for sql subqueries", in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007, pp. 993–1004.

[46] P. Fender and G. Moerkotte, "Counter strike: Generic top-down join enumeration for hypergraphs", *Proc. VLDB Endow.*, vol. 6, no. 14, pp. 1822–1833, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol6/p1822-fender.pdf.

[47] P. Fender, G. Moerkotte, T. Neumann, and V. Leis, "Effective and robust pruning for top-down join enumeration algorithms", in *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, A. Kementsietsidis and M. A. V.

Salles, Eds., IEEE Computer Society, 2012, pp. 414–425. [Online]. Available: https://doi.org/10.1109/ICDE.2012.27.

[48] X. Feng, A. Kumar, B. Recht, and C. Re, "Towards a unified architecture for in-rdbms analytics", in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, Eds., ACM, 2012, pp. 325–336. [Online]. Available: https://doi.org/10.1145/2213836.2213874.

[49] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, "Cypher: An evolving query language for property graphs", in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, G. Das, C. M. Jermaine, and P. A. Bernstein, Eds., ACM, 2018, pp. 1433–1445. [Online]. Available: https://doi.org/10.1145/3183713.3190657.

[50] J. Fu, J. Liang, Z. Wu, and Y. Jiang, "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*, 2024, pp. 1–12.

[51] J. Fu, J. Liang, Z. Wu, M. Wang, and Y. Jiang, "Griffin : Grammar-free DBMS fuzzing", in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, ACM, 2022, 49:1–49:12. [Online]. Available: https://doi.org/10.1145/3551349.3560431.

[52] W. Fu and T. Menzies, "Easy over hard: A case study on deep learning", in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 49–60.

[53] S. Gathani, P. Lim, and L. Battle, "Debugging database queries: A survey of tools, techniques, and users", in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020, pp. 1–16.

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

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

[56] S. Ginsburg and X. Wang, "Pattern matching by rs-operations: Towards a unified approach to querying sequenced data", in *Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, 1992, pp. 293–300.

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

[58] M. Gligoric and R. Majumdar, "Model checking database applications", in *Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 19*, Springer, 2013, pp. 549–564.

[59] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers", in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 72–82.

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

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

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

[63] T. Grust, F. Kliebhan, J. Rittinger, and T. Schreiber, "True language-level sql debugging", in *Proceedings of the 14th International Conference on Extending Database Technology*, 2011, pp. 562–565.

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

[65] P. Guagliardo and L. Libkin, "A formal semantics of SQL queries, its validation, and applications", *Proc. VLDB Endow.*, vol. 11, no. 1, pp. 27–39, 2017. [Online]. Available: `http://www.vldb.org/pvldb/vol11/p27-guagliardo.pdf`.

[66] P. Guagliardo and L. Libkin, "A formal semantics of sql queries, its validation, and applications", *Proceedings of the VLDB Endowment*, vol. 11, no. 1, pp. 27–39, 2017.

[67] P. Guagliardo and L. Libkin, "How standard is the SQL standard?" In *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018*, D. Olteanu and B. Poblete, Eds., ser. CEUR Workshop Proceedings, vol. 2100, CEUR-WS.org, 2018. [Online]. Available: `https://ceur-ws.org/Vol-2100/paper16.pdf`.

[68] P. Guagliardo and L. Libkin, "On the codd semantics of sql nulls", *Information Systems*, vol. 86, pp. 46–60, 2019.

[69] A. Gueidi, H. Gharsellaoui, and S. B. Ahmed, "Towards unified modeling for nosql solution based on mapping approach", in *Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 25th International Conference KES-2021, Virtual Event / Szczecin, Poland, 8-10 September 2021*, J. Watrobski, W. Salabun, C. Toro, C. Zanni-Merk, R. J. Howlett, and L. C. Jain, Eds., ser. Procedia Computer Science, vol. 192, Elsevier, 2021, pp. 3637–3646. [Online]. Available: `https://doi.org/10.1016/j.procs.2021.09.137`.

[70] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, *et al.*, "What bugs live in the cloud? a study of 3000+ issues in cloud systems", in *Proceedings of the ACM symposium on cloud computing*, 2014, pp. 1–14.

[71] A. Guttman, "R-trees: A dynamic index structure for spatial searching", in *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, B. Yormark, Ed., ACM Press, 1984, pp. 47–57. [Online]. Available: https://doi.org/10.1145/602259.602266.

[72] X. Han and T. Yu, "An empirical study on performance bugs for highly configurable software systems", in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2016, pp. 1–10.

[73] Y. Han, Z. Wu, P. Wu, R. Zhu, J. Yang, L. W. Tan, K. Zeng, G. Cong, Y. Qin, A. Pfadler, Z. Qian, J. Zhou, J. Li, and B. Cui, "Cardinality estimation in DBMS: A comprehensive benchmark evaluation", *Proc. VLDB Endow.*, vol. 15, no. 4, pp. 752–765, 2021. [Online]. Available: https://www.vldb.org/pvldb/vol15/p752-zhu.pdf.

[74] H. He, Z. Jia, S. Li, E. Xu, T. Yu, Y. Yu, J. Wang, and X. Liao, "Cp-detector: Using configuration-related performance properties to expose performance bugs", in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 623–634.

[75] M. Heimel, M. Kiefer, and V. Markl, "Self-tuning, gpu-accelerated kernel density models for multidimensional selectivity estimation", in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, T. K. Sellis, S. B. Davidson, and Z. G. Ives, Eds., ACM, 2015, pp. 1477–1492. [Online]. Available: https://doi.org/10.1145/2723372.2749438.

[76] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig, "Deepdb: Learn from data, not from queries!" *Proc. VLDB Endow.*, vol. 13, no. 7, pp. 992–1005, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p992-hilprecht.pdf.

[77] D. Hirn and T. Grust, "Pgcuckoo: Laying plan eggs in postgresql's nest", in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1929–1932.

[78] K. Huang, S. Liu, Z. Chen, H. Wei, D. Basin, H. Li, and A. Pan, "Efficient black-box checking of snapshot isolation in databases", *arXiv preprint arXiv:2301.07313*, 2023.

[79] E. International, "Ecma-404—the json data interchange format", 2013.

[80] Y. E. Ioannidis, "The history of histograms (abridged)", in *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer, Eds., Morgan Kaufmann, 2003, pp. 19–30. [Online]. Available: `http://www.vldb.org/conf/2003/papers/S02P01.pdf`.

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

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

[83] Z.-M. Jiang, S. Liu, M. Rigger, and Z. Su, "Detecting transactional bugs in database engines via Graph-Based oracle construction", in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, Boston, MA: USENIX Association, Jul. 2023, pp. 397–417, ISBN: 978-1-939133-34-2. [Online]. Available: `https://www.usenix.org/conference/osdi23/presentation/jiang`.

[84] Y. Jiang, J. Liu, J. Ba, R. H. C. Yap, Z. Liang, and M. Rigger, "Detecting logic bugs in graph database management systems via injective and surjective graph query transformation", in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, IEEE Computer Society, 2023, pp. 531–542.

[85] **Jinsheng Ba** and M. Rigger, "Testing database engines via query plan guidance", in *The 45th International Conference on Software Engineering (ICSE'23)*, **ACM SIGSOFT Distinguished Paper Award**, May 2023.

[86] **Jinsheng Ba** and M. Rigger, "Finding performance issues in database engines via cardinality estimation testing", in *The 46th International Conference on Software Engineering (ICSE'24)*, Apr. 2024.

[87] **Jinsheng Ba** and M. Rigger, "Keep it simple: Testing databases via differential query plans", *Proc. ACM Manag. Data (SIGMOD'24)*, Jun. 2024.

[88] **Jinsheng Ba** and M. Rigger, "Towards a unified query plan representation for database applications", in *Arxiv*.

[89] J. Jung, H. Hu, J. Arulraj, T. Kim, and W. Kang, "APOLLO: automatic detection and diagnosis of performance regressions in database systems", *Proc. VLDB Endow.*, vol. 13, no. 1, pp. 57–70, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol13/p57-jung.pdf.

[90] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream data processing systems", in *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, IEEE Computer Society, 2018, pp. 1507–1518. [Online]. Available: https://doi.org/10.1109/ICDE.2018.00169.

[91] M. L. Kersten, Y. Zhang, N. Nes, and P. Koutsourakis, "Bridging the chasm between science and reality." In *CIDR*, 2021.

[92] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, "Finding semantic bugs in file systems with an extensible fuzzing framework", in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 147–161.

[93] K. Kingsbury and P. Alvaro, "Elle: Inferring isolation anomalies from experimental observations", *arXiv preprint arXiv:2003.10554*, 2020.

[94] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper, "Learned cardinalities: Estimating correlated joins with deep learning", in *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, www.cidrdb.org, 2019. [Online]. Available: http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf.

[95] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing", in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 2123–2138.

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

[97] C. Laaber, J. Scheuner, and P. Leitner, "Software microbenchmarking in the cloud. how bad is it really?" *Empir. Softw. Eng.*, vol. 24, no. 4, pp. 2469–2508, 2019. [Online]. Available: https://doi.org/10.1007/s10664-019-09681-1.

[98] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs", *ACM Sigplan Notices*, vol. 49, no. 6, pp. 216–226, 2014.

[99] T. J. Lehman and M. J. Carey, "A study of index structures for main memory database management systems", in *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, Eds., Morgan Kaufmann, 1986, pp. 294–303. [Online]. Available: http://www.vldb.org/conf/1986/P294.PDF.

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

[101] V. Leis, B. Radke, A. Gubichev, A. Kemper, and T. Neumann, "Cardinality estimation done right: Index-based join sampling", in *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, www.cidrdb.org, 2017. [Online]. Available: http://cidrdb.org/cidr2017/papers/p9-leis-cidr17.pdf.

[102] A. Y. Levy, I. S. Mumick, and Y. Sagiv, "Query optimization by predicate move-around", in *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, J. B. Bocca, M. Jarke, and C. Zaniolo, Eds., Morgan Kaufmann, 1994, pp. 96–107. [Online]. Available: http://www.vldb.org/conf/1994/P096.PDF.

[103] F. Li, B. Wu, K. Yi, and Z. Zhao, "Wander join: Online aggregation via random walks", in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, F. Özcan, G. Koutrika, and S. Madden, Eds., ACM, 2016, pp. 615–629. [Online]. Available: https://doi.org/10.1145/2882903.2915235.

[104] J. Liang, Y. Chen, Z. Wu, J. Fu, M. Wang, Y. Jiang, X. Huang, T. Chen, J. Wang, and J. Li, "Sequence-oriented dbms fuzzing", in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, IEEE, 2023, pp. 668–681.

[105] J. Liang, Z. Wu, J. Fu, M. Wang, C. Sun, and Y. Jiang, "Mozi: Discovering dbms bugs via configuration-based equivalent transformation", in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[106] Y. Liang, S. Liu, and H. Hu, "Detecting logical bugs of DBMS with coverage-based guidance", in *31st USENIX Security Symposium (USENIX Security 22)*, USENIX Association, Aug. 2022, ISBN: 978-1-939133-31-1.

[107] X. Liu, Q. Zhou, J. Arulraj, and A. Orso, "Automatic detection of performance bugs in database systems using equivalent queries", in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, ACM, 2022, pp. 225–236. [Online]. Available: https://doi.org/10.1145/3510003.3510093.

[108] G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky, "Toward a verified relational database management system", in *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2010, pp. 237–248.

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

[110] M. Marcozzi, Q. Tang, A. F. Donaldson, and C. Cadar, "Compiler fuzzing: How much does it matter?" *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[111] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Making learned query optimization practical", *SIGMOD Rec.*, vol. 51, no. 1, pp. 6–13, 2022. [Online]. Available: https://doi.org/10.1145/3542700.3542703.

[112] R. Marcus and O. Papaemmanouil, "Deep reinforcement learning for join order enumeration", in *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2018, Houston, TX, USA, June 10, 2018*, R. Bordawekar and O. Shmueli, Eds., ACM, 2018, 3:1–3:4. [Online]. Available: https://doi.org/10.1145/3211954.3211957.

[113] R. C. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, "Neo: A learned query optimizer", *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1705–1718, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p1705-marcus.pdf.

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

[115] Q. Meng, X. Ma, W. Lu, and Z. Yao, "A spatial SQL based on sparksql", in *Geo-Spatial Knowledge and Intelligence - 4th International Conference on Geo-Informatics in Resource Management and Sustainable Ecosystem, GRMSE 2016, Hong Kong, China, November 18-20, 2016, Revised Selected Papers, Part I*, H. Yuan, J. Geng, and F. Bian, Eds., ser. Communications in Computer and Information Science, vol. 698, Springer, 2016, pp. 437–443. [Online]. Available: https://doi.org/10.1007/978-981-10-3966-9%5C_50.

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

[117] C. Mishra, N. Koudas, and C. Zuzarte, "Generating targeted queries for database testing", in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08, Vancouver, Canada: ACM, 2008, pp. 499–510, ISBN: 978-1-60558-102-6. [Online]. Available: `http://doi.acm.org/10.1145/1376616.1376668`.

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

[119] B. Mitschang, "Towards a unified view of design data and knowledge representation", in *Expert Database Systems, Proceedings from the Second International Conference, Vienna, Virginia, USA, April 25-27, 1988*, L. Kerschberg, Ed., Benjamin/Cummings, 1988, pp. 133–159.

[120] J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram, "Finding crash-consistency bugs with bounded black-box crash testing", in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 33–50.

[121] A. Mueen, E. J. Keogh, Q. Zhu, S. Cash, and M. B. Westover, "Exact discovery of time series motifs", in *Proceedings of the SIAM International Conference on Data Mining, SDM 2009, April 30 - May 2, 2009, Sparks, Nevada, USA*, SIAM, 2009, pp. 473–484. [Online]. Available: `https://doi.org/10.1137/1.9781611972795.41`.

[122] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, M. L. Soffa and M. J. Irwin, Eds., ACM, 2009, pp. 265–276. [Online]. Available: `https://doi.org/10.1145/1508244.1508275`.

[123] R. O. Nambiar, M. Poess, A. Masland, H. R. Taheri, M. Emmerton, F. Carman, and M. Majdalany, "TPC benchmark roadmap 2012", in *Selected Topics in Performance Evaluation and Benchmarking - 4th TPC Technology Conference, TPCTC 2012, Istanbul, Turkey, August 27, 2012, Revised Selected*

*Papers*, R. O. Nambiar and M. Poess, Eds., ser. Lecture Notes in Computer Science, vol. 7755, Springer, 2012, pp. 1–20. [Online]. Available: https://doi.org/10.1007/978-3-642-36727-4%5C_1.

[124] G. Navarro, "A guided tour to approximate string matching", *ACM computing surveys (CSUR)*, vol. 33, no. 1, pp. 31–88, 2001.

[125] M. Negri, G. Pelagatti, and L. Sbattella, "Formal semantics of sql queries", *ACM Transactions on Database Systems (TODS)*, vol. 16, no. 3, pp. 513–534, 1991.

[126] D. Neuman, "Evaluating evolution: Naturalistic inquiry and the perseus project", *Comput. Humanit.*, vol. 25, no. 4, pp. 239–246, 1991. [Online]. Available: https://doi.org/10.1007/BF00116078.

[127] T. Neumann, "Query simplification: Graceful degradation for join-order optimization", in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, U. Çetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, Eds., ACM, 2009, pp. 403–414. [Online]. Available: https://doi.org/10.1145/1559845.1559889.

[128] J. Oetsch, M. Prischink, J. Pührer, M. Schwengerer, and H. Tompits, "On the small-scope hypothesis for testing answer-set programs", in *Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning*, 2012.

[129] A. Orthey, B. Fresz, and M. Toussaint, "Motion planning explorer: Visualizing local minima using a local-minima tree", *IEEE Robotics Autom. Lett.*, vol. 5, no. 2, pp. 346–353, 2020. [Online]. Available: https://doi.org/10.1109/LRA.2019.2958524.

[130] H. Ouyang, H. Wei, H. Li, A. Pan, and Y. Huang, "Checking causal consistency of mongodb", *J. Comput. Sci. Technol.*, vol. 37, no. 1, pp. 128–146, 2022. [Online]. Available: https://doi.org/10.1007/s11390-021-1662-8.

[131] K. Pan, X. Wu, and T. Xie, "Guided test generation for database applications via synthesized database interactions", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 2, pp. 1–27, 2014.

[132] K. K. Pasupuleti, J. Li, H. Su, and M. Ziauddin, "Automatic sql error mitigation in oracle", *Proceedings of the VLDB Endowment*, vol. 16, no. 12, pp. 3835–3847, 2023.

[133] R. E. Pattis, "Teaching EBNF first in CS 1", in *Proceedings of the 25th SIGCSE Technical Symposium on Computer Science Education, SIGCSE 1994, Phoenix, Arizona, USA, March 10-12, 1994*, R. Beck and D. Goelman, Eds., ACM, 1994, pp. 300–303. [Online]. Available: https://doi.org/10.1145/191029.191155.

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

[135] A. Pavlo and M. Aslett, "What's really new with newsql?" *SIGMOD Rec.*, vol. 45, no. 2, pp. 45–55, 2016. [Online]. Available: https://doi.org/10.1145/3003665.3003674.

[136] D. Pawlaszczyk, "Sqlite", in *Mobile Forensics - The File Format Handbook - Common File Formats and File Systems Used in Mobile Devices*, C. Hummert and D. Pawlaszczyk, Eds., Springer, 2022, pp. 129–155. [Online]. Available: https://doi.org/10.1007/978-3-030-98467-0%5C_5.

[137] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "All file systems are not created equal: On the complexity of crafting {crash-consistent} applications", in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 433–448.

[138] M. Poess, T. Rabl, and H. Jacobsen, "Analysis of TPC-DS: the first standard benchmark for sql-based big data systems", in *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*, ACM, 2017, pp. 573–585. [Online]. Available: https://doi.org/10.1145/3127479.3128603.

[139] M. Poess and J. M. Stephens Jr., "Generating thousand benchmark queries in seconds", in *Proceedings of the Thirtieth International Conference on Very*

*Large Data Bases - Volume 30*, ser. VLDB '04, Toronto, Canada: VLDB Endowment, 2004, pp. 1045–1053, ISBN: 0-12-088469-0.

[140] V. Poosala and Y. E. Ioannidis, "Selectivity estimation without the attribute value independence assumption", in *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, Eds., Morgan Kaufmann, 1997, pp. 486–495. [Online]. Available: http://www.vldb.org/conf/1997/P486.PDF.

[141] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems", in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 24–36.

[142] M. Raasveldt, P. Holanda, T. Gubner, and H. Mühleisen, "Fair benchmarking considered difficult: Common pitfalls in database performance testing", in *Proceedings of the 7th International Workshop on Testing Database Systems, DBTest@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, A. Böhm and T. Rabl, Eds., ACM, 2018, 2:1–2:6. [Online]. Available: https://doi.org/10.1145/3209950.3209955.

[143] K. Rabuzin, M. Cerjan, and S. Krizanic, "Supporting data types in neo4j", in *New Trends in Database and Information Systems - ADBIS 2022 Short Papers, Doctoral Consortium and Workshops: DOING, K-GALS, MADEISD, MegaData, SWODCH, Turin, Italy, September 5-8, 2022, Proceedings*, S. Chiusano, T. Cerquitelli, R. Wrembel, K. Nørvåg, B. Catania, G. Vargas-Solar, and E. Zumpano, Eds., ser. Communications in Computer and Information Science, vol. 1652, Springer, 2022, pp. 459–466. [Online]. Available: https://doi.org/10.1007/978-3-031-15743-1%5C_42.

[144] J. W. Ratcliff, D. Metzener, *et al.*, "Pattern matching: The gestalt approach", *Dr. Dobb's Journal*, vol. 13, no. 7, p. 46, 1988.

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

[146] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs", in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, Beijing, China: Association for Computing Machinery, 2012, pp. 335–346, ISBN: 9781450312059. [Online]. Available: https://doi.org/10.1145/2254064.2254104.

[147] K. Rehmann, C. Seo, D. Hwang, B. T. Truong, A. Boehm, and D. H. Lee, "Performance monitoring in SAP hana's continuous integration process", *SIGMETRICS Perform. Evaluation Rev.*, vol. 43, no. 4, pp. 43–52, 2016. [Online]. Available: https://doi.org/10.1145/2897356.2897362.

[148] Research and Markets, "Database Software Global Market Report 2023", 2023. [Online]. Available: https://www.researchandmarkets.com/reports/5735140/database-software-global-market-report#product--related-products.

[149] M. Rigger and Z. Su, "Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction", in *Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, Sacramento, California, United States, 2020.

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

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

[152] I. M. Rodrigues, D. Aloise, and E. R. Fernandes, "Fast: A linear time stack trace alignment heuristic for crash report deduplication", in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 549–560.

[153] P. van Rosmalen, E. A. Boyle, J. van der Baaren, A. I. Kärki, and A. del Blanco Aguado, "A case study on the design and development of minigames for

research methods and statistics", *EAI Endorsed Trans. Serious Games*, vol. 1, no. 3, e5, 2014. [Online]. Available: https://doi.org/10.4108/sg.1.3.e5.

[154] P. Runeson, M. Höst, A. Rainer, and B. Regnell, "Case Study Research in Software Engineering - Guidelines and Examples", Wiley, 2012, ISBN: 978-1-118-10435-4. [Online]. Available: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-1118104358.html.

[155] D. Scheibli, C. Dinse, and A. Boehm, "QE3D: interactive visualization and exploration of complex, distributed query plans", in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, T. K. Sellis, S. B. Davidson, and Z. G. Ives, Eds., ACM, 2015, pp. 877–881. [Online]. Available: https://doi.org/10.1145/2723372.2735364.

[156] A. Schroter, A. Schröter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?" In *2010 7th IEEE working conference on mining software repositories (MSR 2010)*, IEEE, 2010, pp. 118–121.

[157] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system", in *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '79, Boston, Massachusetts: Association for Computing Machinery, 1979, pp. 23–34, ISBN: 089791001X. [Online]. Available: https://doi.org/10.1145/582095.582099.

[158] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system", in *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, P. A. Bernstein, Ed., ACM, 1979, pp. 23–34. [Online]. Available: https://doi.org/10.1145/582095.582099.

[159] K. Serebryany, "Oss-fuzz-google's continuous fuzzing service for open source software",, 2017.

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

[161] D. R. Slutz, "Massive stochastic testing of sql", Tech. Rep. MSR-TR-98-21, Aug. 1998, p. 9. [Online]. Available: https://www.microsoft.com/en-us/research/publication/massive-stochastic-testing-of-sql/.

[162] J. Song, W. Dou, Z. Cui, Q. Dai, W. Wang, J. Wei, H. Zhong, and T. Huang, "Testing database systems via differential query execution", in *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*, 2023.

[163] C. Strauch, U.-L. S. Sites, and W. Kriha, "Nosql databases", *Lecture Notes, Stuttgart Media University*, vol. 20, no. 24, p. 79, 2011.

[164] X. Sun, R. Cheng, J. Chen, E. Ang, O. Legunsen, and T. Xu, "Testing configuration changes in context to prevent production failures", in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 735–751.

[165] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis, "Cockroachdb: The resilient geo-distributed sql database", in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20, Portland, OR, USA: International Foundation for Autonomous Agents and Multiagent Systems, 2020, ISBN: 9781450367356.

[166] C. Tan, C. Zhao, S. Mu, and M. Walfish, "Cobra: Making transactional key-value stores verifiably serializable", in *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, USENIX Association, 2020, pp. 63–80. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/tan.

[167] X. Tang, S. Wu, D. Zhang, F. Li, and G. Chen, "Detecting logic bugs of join optimizations in dbms", *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–26, 2023.

[168] X. Tang, S. Wu, D. Zhang, Z. Wang, G. Yuan, and G. Chen, "A demonstration of dlbd: Database logic bug detection system", *Proceedings of the VLDB Endowment*, vol. 16, no. 12, pp. 3914–3917, 2023.

[169] J. Toman and D. Grossman, "Staccato: A bug finder for dynamic configuration updates", in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[170] H. Touzet, "Tree edit distance with gaps", *Inf. Process. Lett.*, vol. 85, no. 3, pp. 123–129, 2003. [Online]. Available: https://doi.org/10.1016/S0020-0190(02)00369-1.

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

[172] R. Vasiliev, D. Koznov, G. Chernishev, A. Khvorov, D. Luciv, and N. Povarov, "Tracesim: A method for calculating stack trace similarity", in *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation*, 2020, pp. 25–30.

[173] W. Wallace, "Review of "designing with web standards (second edition) by jeffrey zeldman", peachpit press, 2006, ISBN 0321385551", *ACM Queue*, vol. 5, no. 4, p. 56, 2007. [Online]. Available: https://doi.org/10.1145/1255421.1255432.

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

[175] G. Wang, R. Shen, J. Chen, Y. Xiong, and L. Zhang, "Probabilistic delta debugging", in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 881–892.

[176] T. Wang, Z. Jia, S. Li, S. Zheng, Y. Yu, E. Xu, S. Peng, and X. Liao, "Understanding and detecting on-the-fly configuration bugs", in *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, 2023.

[177] Website, "Tpc-ds benchmark", https://www.tpc.org/tpcds/, Accessed: 2022-11-15, 1988.

[178] Website. “Microsoft sql server”, (1989), [Online]. Available: https://www.microsoft.com/en-us/sql-server/.

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

[180] Website, “American fuzzy lop (afl) fuzzer”, http://lcamtuf.coredump.cx/afl/technical_details.txt, Accessed: 2022-06-08, 2013.

[181] Website, “Libfuzzer – a library for coverage-guided fuzz testing.” https://llvm.org/docs/LibFuzzer.html, Accessed: 2022-06-08, 2013.

[182] Website, “Sqlsmith”, https://github.com/anse1/sqlsmith, Accessed: 2022-06-08, 2015.

[183] Website, “Dynamic programming and edit distance”, https://www.cs.jhu.edu/~langmea/resources/lecture_notes/dp_and_edit_dist.pdf, Accessed: 2022-11-15, 2020.

[184] Website, “Cockroachdb customers”, https://www.cockroachlabs.com/customers, Accessed: 2022-06-08, 2022.

[185] Website, “Most widely deployed and used database engine”, https://www.sqlite.org/mostdeployed.html, Accessed: 2022-06-08, 2022.

[186] Website, “Postgresql”, https://www.postgresql.org/docs/current/row-estimation-examples.html, Accessed: 2022-11-15, 2022.

[187] Website, “Tidb customers”, https://en.pingcap.com/customers, Accessed: 2022-06-08, 2022.

[188] Website. “Apexsql”, (2023), [Online]. Available: https://www.apexsql.com/products/sql-tools-bundle-fundamentals/.

[189] Website. “Azure data studio”, (2023), [Online]. Available: https://learn.microsoft.com/en-us/sql/azure-data-studio/query-plan-viewer?view=sql-server-ver16.

[190] Website. “Dbvisualizer”, (2023), [Online]. Available: https://www.dbvis.com/.

[191] Website. “Iso/iec 9075-1:2023 information technology — database languages sql”, (2023), [Online]. Available: https://www.iso.org/standard/76583.html.

[192] Website. “Pganalyze”, (2023), [Online]. Available: https://pganalyze.com/.

[193] Website. “Pgmustard”, (2023), [Online]. Available: https://www.pgmustard.com/.

[194] Website. “Postgres explain visualizer”, (2023), [Online]. Available: https://explain.dalibo.com/.

[195] W. Wen, T. Yu, and J. H. Hayes, “Colua: Automatically predicting configuration bug reports and extracting configuration options”, in *2016 IEEE 27Th international symposium on software reliability engineering (ISSRE)*, IEEE, 2016, pp. 150–161.

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

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

[198] Z. Wu, A. Shaikhha, R. Zhu, K. Zeng, Y. Han, and J. Zhou, “Bayescard: Revitilizing bayesian frameworks for cardinality estimation”, *arXiv preprint arXiv:2012.14743*, 2020.

[199] Z. Wu, P. Yu, P. Yang, R. Zhu, Y. Han, Y. Li, D. Lian, K. Zeng, and J. Zhou, “A unified transferable model for ml-enhanced DBMS”, in *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*, www.cidrdb.org, 2022. [Online]. Available: https://www.cidrdb.org/cidr2022/papers/p6-wu.pdf.

[200] Z. Wu, R. Zhu, A. Pfadler, Y. Han, J. Li, Z. Qian, K. Zeng, and J. Zhou, “FSPN: A new class of probabilistic graphical model”, *CoRR*, vol. abs/2011.09020, 2020. arXiv: 2011.09020. [Online]. Available: https://arxiv.org/abs/2011.09020.

[201] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, "Early detection of configuration errors to reduce failure damage", in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 619–634.

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

[203] C. Yan, Y. Lin, and Y. He, "Predicate pushdown for data science pipelines", *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 1–28, 2023.

[204] C. Yan, S. Nath, and S. Lu, "Generating test databases for database-backed applications", in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, IEEE, 2023, pp. 2048–2059.

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

[206] J. Yang and X. Chen, "A semi-structured document model for text mining", *J. Comput. Sci. Technol.*, vol. 17, no. 5, pp. 603–610, 2002. [Online]. Available: `https://doi.org/10.1007/BF02948828`.

[207] R. Yang, P. Kalnis, and A. K. H. Tung, "Similarity evaluation on tree-structured data", in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, F. Özcan, Ed., ACM, 2005, pp. 754–765. [Online]. Available: `https://doi.org/10.1145/1066157.1066243`.

[208] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers", in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[209] Z. Yang, A. Kamsetty, S. Luan, E. Liang, Y. Duan, X. Chen, and I. Stoica, "Neurocard: One cardinality estimator for all tables", *Proc. VLDB Endow.*, vol. 14, no. 1, pp. 61–73, 2020. [Online]. Available: `http://www.vldb.org/pvldb/vol14/p61-yang.pdf`.

[210] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica, "Deep unsupervised cardinality estimation", *Proc. VLDB Endow.*, vol. 13, no. 3, pp. 279–292, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol13/p279-yang.pdf.

[211] X. Yu, G. Li, C. Chai, and N. Tang, "Reinforcement learning with tree-lstm for join order selection", in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, IEEE, 2020, pp. 1297–1308. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00116.

[212] H. Yuan, G. Li, L. Feng, J. Sun, and Y. Han, "Automatic view generation with deep learning and reinforcement learning", in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, IEEE, 2020, pp. 1501–1512. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00133.

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

[214] A. Zeller, "Why programs fail - a guide to systematic debugging", Elsevier, 2006, ISBN: 978-1-55860-866-5.

[215] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input", *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[216] Y. Zhao, G. Cong, J. Shi, and C. Miao, "Queryformer: A tree transformer model for query plan representation", *Proc. VLDB Endow.*, vol. 15, no. 8, pp. 1658–1670, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p1658-zhao.pdf.

[217] Z. Zhao, R. Christensen, F. Li, X. Hu, and K. Yi, "Random sampling over joins revisited", in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, G. Das, C. M. Jermaine, and P. A. Bernstein, Eds., ACM, 2018,

pp. 1525–1539. [Online]. Available: https://doi.org/10.1145/3183713.3183739.

[218] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh, "Torturing databases for fun and profit", in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, J. Flinn and H. Levy, Eds., USENIX Association, 2014, pp. 449–464. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zheng%5C_mai.

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

[220] R. Zhu, Z. Wu, Y. Han, K. Zeng, A. Pfadler, Z. Qian, J. Zhou, and B. Cui, "FLAT: fast, lightweight and accurate method for cardinality estimation", *Proc. VLDB Endow.*, vol. 14, no. 9, pp. 1489–1502, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol14/p1489-zhu.pdf.

# Publications during PhD Study

[1] **Jinsheng Ba** and M. Rigger, "Testing database engines via query plan guidance", in *The 45th International Conference on Software Engineering (ICSE'23)*, **ACM SIGSOFT Distinguished Paper Award**, May 2023.

[2] **Jinsheng Ba** and M. Rigger, "Finding performance issues in database engines via cardinality estimation testing", in *The 46th International Conference on Software Engineering (ICSE'24)*, Apr. 2024.

[3] **Jinsheng Ba** and M. Rigger, "Keep it simple: Testing databases via differential query plans", *Proc. ACM Manag. Data (SIGMOD'24)*, Jun. 2024.

[4] **Jinsheng Ba** and M. Rigger, "Towards a unified query plan representation for database applications", in *Arxiv*.

[5] **Jinsheng Ba**, G. J. Duck, and A. Roychoudhury, "Efficient greybox fuzzing to detect memory errors", in *The 37th IEEE/ACM International Conference on Automated Software Engineering (ASE'22)*, **ACM SIGSOFT Distinguished Paper Award**, Oct. 2022.

[6] **Jinsheng Ba**, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, "Stateful greybox fuzzing", in *31st USENIX Security Symposium (SEC'22)*, Boston, MA: USENIX Association, Aug. 2022, pp. 3255–3272, ISBN: 978-1-939133-31-1. [Online]. Available: `https://www.usenix.org/conference/usenixsecurity22/presentation/ba`.

This thesis includes [1, 2, 3, 4], which contribute to the thesis statement.