Finding Performance Issues in Database Engines via Cardinality Estimation Testing

Jinsheng Ba
National University of Singapore
Singapore
bajinsheng@u.nus.edu

Manuel Rigger
National University of Singapore
Singapore
rigger@nus.edu.sg

ABSTRACT

Database Management Systems (DBMSs) process a given query by creating an execution plan, which is subsequently executed, to compute the query's result. Deriving an efficient query plan is challenging, and both academia and industry have invested decades into researching query optimization. Despite this, DBMSs are prone to performance issues, where a DBMS produces an inefficient query plan that might lead to the slow execution of a query. Finding such issues is a longstanding problem and inherently difficult, because no ground truth information on an expected execution time exists. In this work, we propose *Cardinality Estimation Restriction Testing* (CERT), a novel technique that detects performance issues through the lens of cardinality estimation. Given a query on a database, CERT derives a more restrictive query (*e.g.*, by replacing a `LEFT JOIN` with an `INNER JOIN`), whose estimated number of rows should not exceed the number of estimated rows for the original query. CERT tests cardinality estimators specifically, because they were shown to be the most important component for query optimization; thus, we expect that finding and fixing such issues might result in the highest performance gains. In addition, we found that some other kinds of query optimization issues are exposed by the unexpected cardinality estimation, which can also be detected by CERT. CERT is a black-box technique that does not require access to the source code; DBMSs expose query plans via the `EXPLAIN` statement. CERT eschews executing queries, which is costly and prone to performance fluctuations. We evaluated CERT on three widely used and mature DBMSs, MySQL, TiDB, and CockroachDB. CERT found 13 unique issues, of which 2 issues were fixed and 9 confirmed by the developers. We expect that this new angle on finding performance bugs will help DBMS developers in improving DMBSs' performance.

KEYWORDS

Database, Performance Issue, Cardinality Estimation

1 INTRODUCTION

Database Management Systems (DBMSs) are fundamental software systems that allow users to retrieve, update, and manage data [28, 47, 58]. Most DBMSs support the *Structured Query Langage* (SQL), which allows users to specify queries in a declarative way. Subsequently, the DBMSs translate the query into a concrete execution plan. To balance the trade-off between spending little time on optimization, which is performed at run time, and finding an efficient execution plan, researchers and practitioners have invested decades of effort into query optimization, covering directions such as search space exploration for join ordering [12, 13, 36], index data structures [17], execution time prediction [1, 60], or parallel execution on multi-core CPUs [16] and GPUs [37].

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 suboptimal performance. 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 [35] (*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 [27]; a report on testing SAP HANA [2] 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 [52] or TPC-H [53] are widely used in practice to monitor DBMSs' performance over versions. However, they can be used only to detect regression bugs on a specific set of benchmarks. While predetermined performance baselines or thresholds could be specified [41, 64, 65], 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 [24] generates databases and queries automatically and identifies performance regression issues by validating whether executing the query on different versions of the DBMS results in significantly different execution times. Since APOLLO can find only regression issues, AMOEBA [34] was proposed, which can find performance issues also on previously unseen databases and queries. The core idea of AMOEBA is to construct a pair of semantically equivalent queries, expecting that the query optimizer can derive an equivalent execution plan; a significant discrepancy in the execution time would indicate a performance issue. Although AMOEBA's core idea is appealing and resulted in many issues being reported, many of them were considered false alarms by the developers—only 6 of 39 issues that they reported were confirmed by the developers, 5 of which were fixed [34]. For both APOLLO and AMOEBA, 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 detects performance issues by testing the DBMSs' cardinality estimation. Cardinality estimators estimate how many rows will be processed by each operation in a query. Since they provide only approximate results, it is infeasible to check for a specific number of estimated rows. 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 execution plan [29]. Therefore, we believe that pinpointing issues in cardinality estimation would help developers to focus on the most relevant issues, addressing which might result in significant performance gains. Additionally, this idea is applicable to detect a broader range of performance issues. For example, we found that query optimization issues can be exposed by unexpected cardinality estimations, as shown in Listing 3. The idea is general and can be applied to any given Q and D. Furthermore, cardinality estimations can be readily obtained by DBMSs without executing Q; DBMSs typically provide a SQL `EXPLAIN` statement that provides this information as part of the 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 [69]. Finally, CERT is a black-box technique that can be applied even without access to the source code.

Listing 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 tables' 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`. To validate that this expectation is reflected in the cardinality estimator's result, we first obtain both queries' number of estimated rows in the corresponding query plans, obtained by using an `EXPLAIN` statement. The cardinality estimator estimates that the first query fetches 20 rows, while the second one fetches 60 rows. It is unexpected that the second query is expected to fetch more estimated rows than the first query, which indicates an issue that we have found in CockroachDB. The root cause was an incorrect double-counting when estimating the selectivity of `OR` expression in the `ON` condition of the `INNER JOIN`. The number of estimated rows of the first query with `LEFT JOIN` should be no less than 60; after the developers fixed this issue, the cardinality estimator changed its estimate to 60. This fix improves 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 25% as shown in Listing 7. The improvement was due to a more accurate cardinality estimate, which enabled a better selection of the join order. Note that we avoided executing the query; CERT only examines query plans.

Listing 1: This running example demonstrates a performance issue found by CERT in CockroachDB. It is due to an incorrect double-counting when estimating the selectivity of OR expressions in join ON conditions.

```
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)    |
14 |                        |-• filter
15 |-• scan                 |  | estimated row:12
16 |     estimated row:13   |  | filter:(c0<1)OR(c0>1)
17 |     table: t0@t0_pkey  |  |
18 |     spans: FULL SCAN   |  |-• scan
19 |                        |       estimated row:13
20 |-• scan                 |       table: t0@t0_pkey
21       estimated row:5    |       spans: FULL SCAN
22       table: t1@t1_pkey  |
23       spans: FULL SCAN   |-• scan
24                                estimated row:5
25                                table: t1@t1_pkey
26                                spans: FULL SCAN
```

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.

Listing 2: The typical structure of a query, that is, a SELECT statement. Optional elements are denoted by `[ ]`, while repetitions are denoted by a `*` suffix.

```
SELECT
  [ALL | DISTINCT]
  select_expression [, select_expression...]
  FROM table_reference [INNER | LEFT | RIGHT | FULL |
      CROSS JOIN table_reference...]*
  [WHERE where_condition]
  [GROUP BY column_expression
  [HAVING where_condition]]
  [LIMIT row_count];
```

- 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.

2 BACKGROUND

Structured Query Language. Structured Query Language (SQL) [6] is a declarative programming language that expresses only the logic of a computation without specifying its specific execution. SQL is widely supported by DBMSs; for example, according to a popular ranking,[1] the 10 most popular DBMSs support it. Listing 2 shows the typical structure of a query, whose features we considered in this work. 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 the number of records that are fetched. More advanced features, such as window functions, common table expressions (CTEs), and subqueries can be used. While we did not consider them in this work, we believe that our proposed approach could be extended to support them.

Query optimization. DBMSs include query optimizers that, after parsing the SQL query, determine an efficient *query plan*, which specifies how a SQL query is executed. Determining an efficient query plan is challenging, since many factors might influence the plan's performance. The most commonly used models are cost-based [7]—the query plan with the lowest projected performance cost is chosen. Cardinality estimation was found to be the most important factor that affects the quality of query optimization [29]. Cardinality estimators typically obtain data statistics of the tables to be queried by sampling [20], through histograms [46], or machine learning algorithms [10, 26, 62]. 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 number of rows of A, B, C respectively, and then estimate the number of rows of $A \bowtie B$, $B \bowtie C$, $A \bowtie C$, and $A \bowtie B \bowtie C$. Lastly, the number of estimated rows of these sub-plan queries helps to decide the join order—whether $A \bowtie B$ or $B \bowtie C$ should be executed first.

Query plan. A query plan is a tree of operations that describes how a specific DBMS executes a SQL statement. A query plan can be obtained by executing a query with the prefix `EXPLAIN`. Query plans typically include the number of estimated rows, called *cardinalities*, for operations that affect the number of subsequent rows. Listing 1 shows the two query plans for the two queries of the running example in lines 11–26. The first query uses a `LEFT JOIN`, and its query plan includes three operations: `cross join`, `scan`, and `scan`. The second query uses an `INNER JOIN`, and its query plan includes four operations: `cross join`, `filter`, `scan`, and `scan`. The structural difference between both query plans is the location of operation `filter`, as the predicate `(c0<1)OR(c0>1)` in the `ON` clause of both queries is part of the `cross join` of the first query plan, but is a separate operation `filter` in the second query plan. For the first query plan, the estimated cardinality 20 of the root node `cross join (left outer)` is determined based on the predicate as well as the two estimates for the number of records in tables `t0` and `t1`, which are 13 and 5. For the second query plan, the number of estimated rows for `cross join` is 60; here, the estimate is partly based on the `filter` operation, which is estimated to return 12 rows. In this work, when we refer to the expected number of rows of a query, we refer to the query's root node.

3 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.[2] 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. They are widely used and have thus been studied in other DBMS testing works [33, 43, 44].

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.[3] 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[4] by the filter *is:issue is:closed linked:pr label:type/bug slow in:title.* For CockroachDB, we searched its repository[5] 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 is a performance issue related to query optimization and affects

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

[2]https://insights.stackoverflow.com/survey/2021#most-popular-technologies-database
[3]https://bugs.mysql.com/
[4]https://github.com/pingcap/tidb/issues
[5]https://github.com/cockroachdb/cockroach/issues

Table 1: Previous performance issues and whether they affect cardinality estimation.

DBMS	#ID	Cardinality Estimation
MySQL	61631	Affect
MySQL	56714	Affect
MySQL	25130	Not
CockroachDB	93410	Not
CockroachDB	71790	Not
TiDB	9067	Affect
Sum	6	3

Listing 3: 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} ✔
```

cardinality estimation. Specifically, if the cardinality estimation of the query in a report was changed by the fix, we deemed the performance issue to be related to incorrect cardinality estimation.

Analysis. Table 1 shows the performance issues we found and whether they affect cardinality estimation. Overall, we identified six performance issues in three DBMSs, and three of them affect cardinality estimation. We attribute the lower number of performance issues to the difficulty in identifying and resolving performance issues in query optimization. The issues #61631 and #56714 affect cardinality estimation, because they produce suboptimal query plans which have larger estimated cardinalities than the optimal query plans. Although both issues are not due to the faults in cardinality estimation, they are still observable through cardinality estimation. Issue #9067 affects cardinality estimation due to an issue in computing cardinality for correlated columns. The other three issues, which do not affect 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 cardinality was not changed after fixing the implementation of the operation.

Case study. Listing 3 shows issue #56714 in MySQL as an illustrative example of a performance issue affected 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 estimation, the suboptimal index selection affects the number of 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 estimation, and suboptimal query plans. The latter two causes can affect the number of estimated cardinality.

4 APPROACH

We propose CERT, a novel technique for testing cardinality estimation. The core idea is that a given query should not fetch fewer rows 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 1 shows an overview of CERT based on the running example in Listing 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 many estimated rows 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 rows are not comparable. Next, we give a detailed explanation of each step.

4.1 Database and Query Generation

We require a database state and a query for testing. Both database state and query can be manually given or automatically generated. Common generation-based methods include mutation-based methods [33, 72] and rule-based generation methods [42–44, 55]. How to generate database states and queries is not our contribution in this work, and our approach can be paired with any database state and query generation method. For example, in Listing 1, we 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` statements on each table to guarantee that the data statistics are up to date. For Listing 1, these statements are executed in lines 5–6.

4.2 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. We considered the SQL features shown in Listing 2 and propose at least one rule for each feature, yielding the 12 rules shown in Table 2. Since the `JOIN` clause, which specifies two tables or views to be joined, is a major factor influencing the queries' run time [29], 5 of the 12 rules relate to them. Our rules are not exhaustive; we believe that practitioners

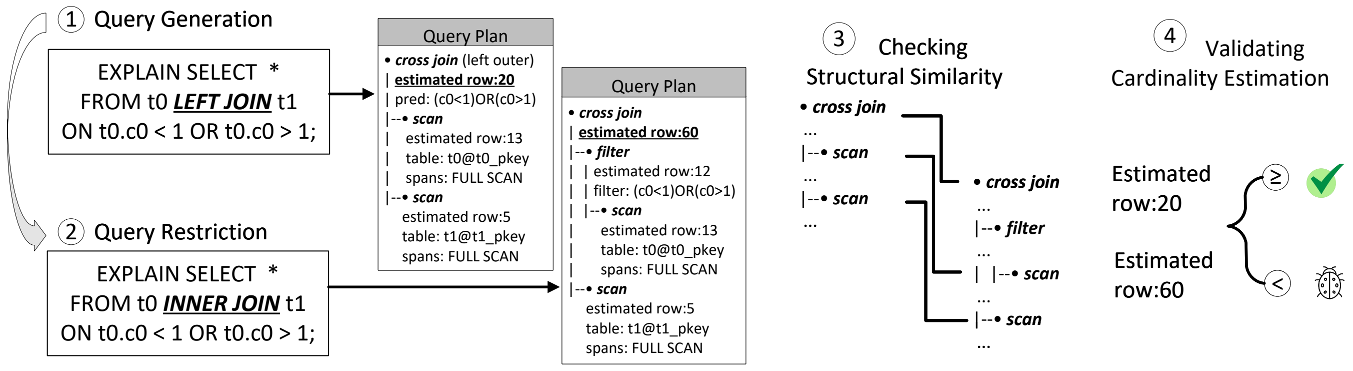

Figure 1: Overview of CERT.

could propose many additional rules depending on their testing focus.

Rule overview. In Table 2, the *Clause* column denotes the SQL clause on which the corresponding rule is applied. The *Source* and *Target* columns denote the source pattern and the restricted target pattern. The <Empty> placeholder refers to an empty clause. The <Predicate> placeholder refers to any boolean expressions in SQL that evaluates to `TRUE`, `FALSE`, or `NULL`. The <Positive Num> placeholder refers to any positive constant number in SQL. Lastly, to facilitate understanding these rules, we provide an example for each rule in the *Example* column. 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 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` ≤ `LEFT JOIN`/`RIGHT JOIN` ≤ `FULL JOIN` ≤ `CROSS JOIN`. Figure 2 illustrates this using a `JOIN` diagram [11] 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 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

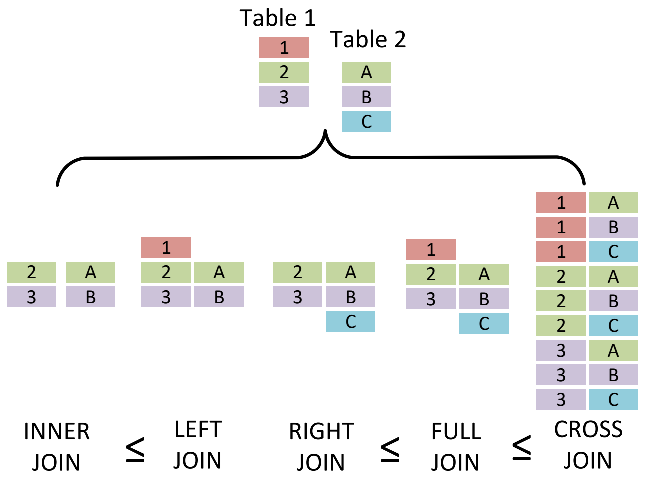

Figure 2: The inequality relationships of cardinality estimation in the `JOIN` clause with an example to join two tables.

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 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.

4.3 Checking for Structural Similarity

Even for similar queries, DBMSs may create significantly different query plans. In such cases, the cardinality estimates might be incomparable and thus result in false alarms. We address this challenge by checking whether query plans are *structurally similar*; if not, we eschew inspecting their estimated number of rows.

Table 2: The rules to restrict queries.

	Clause	Source	Target	Example
1	JOIN	LEFT JOIN	INNER JOIN	SELECT * FROM t0 ~~LEFT~~ INNER JOIN t1 ON ...;
2	JOIN	RIGHT JOIN	INNER JOIN	SELECT * FROM t0 ~~RIGHT~~ INNER JOIN t1 ON ...;
3	JOIN	FULL JOIN	LEFT JOIN	SELECT * FROM t0 ~~FULL~~ LEFT JOIN t1 ON ...;
4	JOIN	FULL JOIN	RIGHT JOIN	SELECT * FROM t0 ~~FULL~~ RIGHT JOIN t1 ON ...;
5†	JOIN	CROSS JOIN	FULL JOIN	SELECT * FROM t0 ~~CROSS~~ FULL JOIN t1;
6	SELECT	ALL	DISTINCT	SELECT ~~ALL~~ DISTINCT * FROM t0;
7	GROUP BY	<Empty>	<Predicate>	SELECT * FROM t0 GROUP BY c0 ;
8	HAVING	<Empty>	<Predicate>	SELECT * FROM t0 GROUP BY c0 HAVING c0>0 ;
9	WHERE	<Empty>	<Predicate>	SELECT * FROM t0 WHERE c0>0 ;
10	WHERE	<Predicate>	<Predicate> AND <Predicate>	SELECT * FROM t0 WHERE c0>0 AND c0!=8 ;
11	WHERE	<Predicate> OR <Predicate>	<Predicate>	SELECT * FROM t0 WHERE c0>0 ~~OR c0!=8~~ ;
12	LIMIT	<Positive Num>	<Positive Num> - <Positive Num>	SELECT * FROM t0 LIMIT ~~10~~ 5 ;

† Rule 5 holds when both tables are not empty.

Listing 4: An example of query plans that are not structurally similar, which is why we exclude them for testing. The query plans are simplified for better illustration.

```
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 (t0)               |-• scan (t1)
18         estimated row:2             estimated row:4
```

Listing 4 shows a pair of query plans. 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 rows 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 Listing 4. 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* [31], 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 number of rows 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 numbers of estimated rows are calculated in a different manner than that of the first query and we consider the number of rows estimated by the two query plans as incomparable. In Listing 4, the estimated rows of both operations `FULL JOIN` and `RIGHT JOIN` are calculated as the sum of the estimated rows in the last step and the operation `filter` is calculated as one-third of the estimated rows in the last step. The number of estimated rows 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 a comparable number of estimated rows by checking for *structural similarity*. For a pair of query plans, if the sequences of operations, obtained by flattening the query plans, can be edited to each other by at most one step, we define both query plans are *structurally similar*. This notion is inspired by the Levenshtein distance.[6] For example, in the query plans of Figure 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 Levenshtein distance of trees was shown to be NP-hard [51].

4.4 Validating Cardinality Estimation

Finally, we validate the *cardinality restriction monotonicity* property on the estimated number of rows extracted from the query plans. If the number of estimated rows of the original query is smaller

[6] The Levenshtein distance is a string metric for measuring the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one word into the other.

than that of the more restrictive query, we report the query pair as an issue. In Figure 1, the number of estimated rows of the original query is 20, which is smaller 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 rows in the root operation of the query plan as the estimated rows of the query and ignore the estimated rows in other operations.

4.5 Implementation

We implemented CERT in SQLancer,[7] which is an automated testing tool for DBMSs that checks discrepancies in queries' results. The extended version of SQLancer is referred to as SQLancer+CERT subsequently. We reused the implementation of a rule-based random generation method of queries and database states of SQLancer. For each table, we generate 100 `INSERT` statements and ensure that every table contains at least 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.

Extracting cardinality estimations. DBMSs provide query plans using various formats. For text-based query plan formats, such as for the CockroachDB's query plans shown in Listing 1, we used pattern matching to extract operations and their corresponding numbers of estimated rows. Some DBMSs, such as MySQL and TiDB, return query plans in a structured way as a table. For them, we directly extract operations and numbers of estimated rows from the corresponding columns. Given the extracted operation sequences, we implemented the *Dynamic Programming* (DP) algorithm [56] to calculate the structural similarity.

5 EVALUATION

To evaluate the effectiveness and efficiency of CERT in finding issues in cardinality estimation, we sought to answer the following questions based on our prototype SQLancer+CERT:

Q.1 Effectiveness. Can CERT identify previously unknown issues?
Q.2 Historic Bugs. Can CERT identify historic performance issues?
Q.3 Accuracy. Can CERT produce fewer false alarms than existing testing methods for finding performance issues?
Q.4 Efficiency. Can CERT process more test cases in a fixed resource budget than existing testing methods for performance issues?
Q.5 Sensitivity. Which rules proposed in Table 2 contribute to finding issues? Do DBMSs adhere to the *cardinality restriction monotonicity* property in practice?

Tested DBMSs. We tested the same DBMSs, MySQL, TiDB, and CockroachDB as we studied in Section 3. For Q1, Q3, and Q5, we used the latest available development versions (MySQL: 8.0.31, TiDB: 6.4.0, CockroachDB: 22.2.0). For Q4, in an attempt of a fairer comparison to AMOEBA, we chose the historical version of CockroachDB 20.2.19, which is the version that was tested by AMOEBA [34].

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

Baselines. To the best of our knowledge, no existing work can be applied to specifically find issues in cardinality estimators. The most closely related work is AMOEBA, which detects performance issues in query optimizers. We did not consider APOLLO [24]; unlike CERT and AMOEBA, APOLLO detects 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 we validate the *cardinality restriction monotonicity* property to find issues in cardinality estimators. Furthermore, both tools support a different set of DBMSs; for example, AMOEBA lacks support for MySQL and TiDB. Thus, we performed the comparison in Q4 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 run all experiments 10 runs for statistical significance.

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

Results. Table 3 shows the unique issues that we found in the cardinality estimators of three tested DBMSs. The *Version* column shows the versions or git commits of the DBMSs in which we found corresponding issues. The *Rules* column shows which rules detected this issue, and the *Modifications to the query* column highlights the key SQL clauses and our restriction operations that exposed the issues. In total, we have reported 13 unique issues in cardinality estimators. 9 issues (1–9) have been confirmed by developers in three days, 2 issues (10–11) have been fixed in one week, and 2 issues (12–13) were backlogged. No false alarm was generated. We speculate that many confirmed bugs remain unfixed, because 1)

Table 3: The issues in cardinality estimation found by CERT. The red strike-through text is removed, while the content in green color is added.

	DBMS	Version	Rules	Modifications to the query	Status
1	MySQL	8.0.31	9	... WHERE t0.c0 > t0.c1 ...	Verified
2	MySQL	8.0.31	9	... WHERE t1.c1 BETWEEN (SELECT 1 WHERE FALSE) AND (t1.c0) ...	Verified
3	MySQL	8.0.31	6	... DISTINCT ...	Verified
4	TiDB	51a6684f	11	...WHERE (TRUE) ~~OR(TO_BASE64(t0.c0))~~ ...	Confirmed
5	TiDB	3ef8352a	7	... GROUP BY t0.c0 ...	Confirmed
6	TiDB	3ef8352a	3,5	... ~~CROSS~~ LEFT JOIN...	Confirmed
7	TiDB	3ef8352a	8	... HAVING (t1.c0)REGEXP(NULL) ...	Confirmed
8	TiDB	6c55faf0	2	... ~~RIGHT~~ INNER JOIN...	Confirmed
9	TiDB	6c55faf0	9	... WHERE v0.c2 ...	Confirmed
10	CockroachDB	7cde315d	1	... ~~LEFT~~ INNER JOIN...	Fixed
11	CockroachDB	f188d21d	11	...WHERE (t0.c0 IS NOT NULL) ~~OR (1 < ALL (t0.c0 & t0.c0))~~ ...	Fixed (Known)
12	CockroachDB	81586f62	8	... HAVING (t1.c0 ::CHAR) = 'a' ...	Backlogged
13	CockroachDB	fbfb71b9	2	... ~~RIGHT~~ INNER JOIN...	Backlogged

```
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
```

Listing 5: A performance issue in MySQL was identified by rule 6, which replaces ALL with DISTINCT.

Listing 6: A performance issue in CockroachDB was identified by rule 11, which 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
```

fixing issues in cardinality estimation requires comprehensive consideration which usually consumes much time, and 2) issues in cardinality estimation 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 (11) 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 three 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. Listing 5 shows a test case exposing a performance issue in MySQL's cardinality estimator. Rule 6, which replaces `ALL` with `DISTINCT` in Table 2, exposed this issue. In Listing 5, the first query in line 5 fetches the rows including duplicate rows, while the second query in line 6 excludes duplicate rows, so the fetched rows of the second query should be no more than that of the first query. However, the number of estimated rows 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 already three hours after we reported it.

An issue identified by rule 11. Listing 6 shows another test case exposing an issue CockroachDB's cardinality estimator 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 number of estimated rows 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 number of estimated rows of predicate `A OR B` is calculated by:

Listing 7: 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
8   ANALYZE t0;
9   ANALYZE t1;
10  ANALYZE t2;
11
12  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 →
        301ms
13  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
```

$P(A\ OR\ B) = P(A) + P(B) - P(A\ AND\ B)$. However, when A and B depend on the same table or column, the number of estimated rows was unexpected. We found this issue by rule 11 in Table 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 whether the issues we found affect performance, we evaluated the query performance of the fixed issues 10 and 11 on a test case as shown in Listing 7 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 25% and 17%, respectively. This improvement is due to the correct estimated cardinality which allows for more optimal joining orders.

Using CERT, we have found 13 unique issues in cardinality estimators of MySQL, TiDB, and CockroachDB. Fixing these issues improves query performance by 21% 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 1, we attempted using CERT to detect all three performance issues whose fix changed the cardinality estimate, namely issues #61631, #56714, and #9067. To this end, based on the queries in the issue reports, we used CERT to construct 1000 pairs of queries, and 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 structural equivalent, we concluded that *cardinality restriction monotonicity* could have identified the performance issue.

Results. All three previous performance issues that affect cardinality estimation can be found by CERT. For example, considering the performance issue #56714 in Listing 3, Listing 8 shows the pair

Listing 8: The issue #56714 violates *cardinality restriction monotonicity.*

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

Table 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%

of queries that CERT produces to find 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 cardinality estimation of the second query decreases to 1.

The *cardinality restriction monotonicity* can identify historical performance issues that affect cardinality estimation.

Q.3 Accuracy

Method. 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 detect different kinds of issues affecting performance.

Results. Table 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 all found issues to avoid duplicate reports. Since AMOEBA supports only CockroachDB from the DBMSs that we tested, to further investigate the accuracy of AMOEBA, we examined its accuracy on PostgreSQL, which is the other DBMS that it supports. None of their 14 issues were confirmed by developers. 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.

Table 5: The number of issues found by each rule.

Rule	MySQL	TiDB	CockroachDB	Sum
1			⋆	1
2		⋆	⋆	2
3		⋆		1
4				0
5		⋆		1
6	⋆			1
7		⋆		1
8		⋆	⋆	2
9	⋆⋆	⋆		3
10				0
11		⋆	⋆	2
12				0

85% of issues found by CERT were confirmed or fixed, while only 24% of reported bugs by AMOEBA were confirmed or fixed by the developers. This result demonstrates the practicality of CERT.

Q.4 Efficiency

Method. We evaluated whether CERT has a higher testing throughput than AMOEBA. State-of-the-art benchmarks and approaches, such as TPC-H [53], AMOEBA [34], and APOLLO [24] execute queries, which results in relatively low throughput. Therefore, it is expected that CERT can validate 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. Note, however, that the throughput results are not directly comparable, as different approaches can find different kinds of issues. In addition, testing cardinality estimation is not prone to performance fluctuation, because the cardinality estimation results 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.

Q.5 Sensitivity

Sensitivity of rules. We evaluated which rules presented in Table 2 contribute to finding the issues in Table 3. Specifically, we recorded which rules were applied in 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 5 shows the number of issues found by each rule. Each star denotes a unique issue. 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 records as specified in its argument. We explain that the simplicity of `LIMIT` explains that we have found no issues in its handling.

Table 6: 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%

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 6 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.

6 DISCUSSION

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

Non-empty tables. When applying CERT, each table must be non-empty. It is based on two considerations. Firstly, it is a precondition of our rules in Table 2. As explained in Section 4.2, rule 5, which restricts `CROSS JOIN` to `FULL JOIN` in Table 2, holds only when either table has at least one row. It is because when either table is empty, operation `CROSS JOIN` fetches zero rows, but operation `FULL JOIN` fetches non-zero rows. We avoid this situation by ensuring that either table contains at least one row. Secondly, we believe that non-empty tables help in finding realistic issues. In a production environment, databases usually do not have empty tables. Because an empty table may result in an inefficient query plan and an inaccurate number of estimated rows[8], cardinality estimators usually assume that each table is non-empty to ensure an overall reasonable cardinality estimation. Developers might consider the performance issues caused by empty tables unrealistic and could be unwilling to fix them.

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 [29] demonstrated that worse cardinality estimation 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 to other DBMSs, apart from relational DBMSs, which we tested in this work. Many DBMSs manage data by declarative languages, in which cardinality estimators are widely used. Neo4J, a graph DBMS, also uses a concept similar to query plans—termed execution plans—and cardinality estimation[9] (*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 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 [33, 43, 44]. We directly obtained the source code from the official git repositories and conducted our experiments on the programs that are directly compiled from raw source code without any modification to avoid bias in results. Our found issues were also confirmed by developers. 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.

[8]See https://postgrespro.com/list/thread-id/2502461

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

7 RELATED WORK

We briefly summarize the works that are most closely related to this work.

Automatically testing for performance issues. The most related strand of research is on finding performance issues in DBMSs automatically. Jung et al. proposed APOLLO [24], which compares the execution times of a query on two versions of a database system to find performance regression bugs. Liu et al. proposed AMOEBA [34], which compares the execution time of a semantically-equivalent pair of queries to identify an unexpected slowdown. In contrast, CERT specifically tests cardinality estimators, which are most critical for query optimization [29]. Thus, we believe that issues found by CERT might be most relevant for DBMSs' performance. Unlike these works, CERT avoids executing queries by inspecting only query plans, allowing for higher throughput and making the approach robust against unanticipated performance fluctuations.

Performance studies of DBMSs. Researchers have conducted studies on the performance of DBMSs. Gu et al. [18] measured the accuracy of query optimizers by forcing the generation of multiple alternative query plans for each query, measuring the actual execution time of all alternatives, ranking the plans by their actual costs obtained by executing the query plans to then compare this ranking with the ranking of the estimated cost. Leis et al. [29] investigated the impact of the components of a query optimizer on performance and found cardinality estimation to be the most important subcomponent that affects query optimization. Campbell et al. [14] shared their experience on migrating SQL Server to a new cardinality estimator. Jin et al. [23] studied real-world performance bugs in five projects, including MySQL, to determine performance bugs' root causes, fixes, and other characteristics. While these works studied DBMSs' performance, CERT finds performance issues automatically.

Performance benchmarking. Benchmarking DBMSs is a common practice to identify performance regressions, and to continuously improve the DBMSs' performance on a set of benchmarks. *TPC-H* [53] and *TPC-DS* [52] 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 [4]. Poess et al. modified and analyzed the TPC-DS benchmark [38] to measure SQL-based big data systems. Karimov et al. proposed a benchmark for stream-data-processing systems [25]. Boncz et al. proposed an improved TPC-H benchmark,*JCC-H* [3], 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)* [29], 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 [40]. CERT is complementary to benchmarking; while benchmarking focuses on workloads deemed relevant for users, CERT can find issues in cardinality estimators even on previously unseen workloads.

DBMS testing. Besides testing DBMSs' performance, automated testing approaches have been proposed to find other types of bugs. Many fuzzing works on finding security-relevant bugs, such as memory errors, were proposed. SQLSmith [55], Griffin [15], DynSQL [22], and ADUSA [34] used grammar-based methods to generate test cases for detecting memory errors. Squirrel [72], inspired by grey-box fuzzers such as AFL [54], used code coverage as a guidance to find memory errors. Apart from security-relevant bugs, logic bugs, which refer to incorrect results returned by DBMSs, are difficult to find as they require a *test oracle*, a mechanism that decides whether the test case's output is expected. The start-of-the-art oracles include PQS [44], NoREC [42], and TLP [43], which have been integrated into SQLancer and have found hundreds of logic bugs in DBMSs. Transactional properties have also been tested. Tan et al. [48] used formal methods to verify the serializability of executions in key-value stores. Zheng et al. [71] found violations of ACID (atomicity, consistency, isolation, and durability) properties of key-value stores. Similar to testing approaches that find logic bugs and test properties, CERT's core component is a new test oracle, which, however, finds performance issues that violate the *cardinality restriction monotonicity* property.

Cardinality estimators. Various approaches have been proposed to improve the accuracy of cardinality estimators. Han et al. [19] comprehensively evaluated various algorithms for cardinality estimation, which describes some of the subsequent important approaches. PostgreSQL [57] and MultiHist [39] applied one-dimensional and multi-dimensional histograms to estimate cardinality. Similarly, UniSample [30, 70] and WJSample [32] used sampling-based methods to estimate cardinality. Apart from these traditional approaches, machine learning-based methods have gained attention recently. MSCN [26, 68], LW-XGB [10], and UAE-Q [62] 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 [67], BayesCard [61], DeepDB [21], and FSPN [63, 73] utilized deep auto-regression models and three probabilistic graphical models BN, SPN, and FSPN to predicate the data distribution for cardinality estimation. CERT is a black-box technique that could, in principle, be applied to any cardinality estimator. However, we believe that fixing issues in learning-based estimators might be challenging.

Metamorphic testing. At a conceptual level, CERT can be classified as a metamorphic testing approach [8, 9], which is a method to generate both test cases and validate results. Metamorphic testing uses an input I to a system and its output O to derive a new input I' (and output O'), for which a test oracle can be provided that checks whether a so-called *Metamorphic Relation* holds between O and O'. For CERT, I corresponds to the original query, O is the number of estimated rows, I' is the restricted query, and O' is its number of estimated rows. The metamorphic relation that we validate is the *cardinality restriction monotonicity* property. Metamorphic testing has been applied successfully in various domains [9, 45]. TLP [43] and NoREC [42] are other metamorphic testing approaches that were proposed for testing DBMSs, as discussed above.

Identifying performance issues in other domains. Automated testing approaches have been proposed to find performance issues also in other domains, such as compilers, where a performance issue refers to inefficient code produced for an input program. Deadspy [5] identifies dead stores by dynamically detecting successive writes to the same memory locations without a read between them. Runtime value numbering [59] identifies computation redundancies that compilers failed to eliminate. CIDetector [49] empirically compares dead and redundant operation elimination between various compilers. Theodoridis et al. proposed a differential testing technique based on the insights that dead code elimination can be used to test for a wide range of potentially-missed optimizations [50]. Besides automated testing, developer-in-the-loop techniques have been proposed. For example, in the context of data-centric systems, Yang et al. proposed Panorama, an approach that allows web developers to understand and optimize their database-backed web applications [66].

8 CONCLUSION

We have presented CERT, a novel technique for finding discrepancies in cardinality estimators, aiming to identify performance issues in DBMSs. Our key idea is to, given a query, derive a more restrictive query and validate that the DBMSs' cardinality estimators project that the original query fetches at least as many rows as the more restrictive query; we refer to this property as *cardinality restriction monotonicity*. Our evaluation has demonstrated that this technique is effective. Of the 13 unique issues that we reported, 2 issues were fixed, 9 issues were confirmed, and 2 issues require further investigation. The fixes improved query performance by 21% on average. We also found this technique to be efficient. Unlike other testing approaches, it avoids executing queries, achieving a speedup of 386× compared to the state of the art. Finally, it is readily applicable. DBMSs expose cardinality estimates 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.

REFERENCES

[1] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B Zdonik. 2012. Learning-based query performance modeling and prediction. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 390–401.

[2] Thomas Bach, Artur Andrzejak, Changyun Seo, Christian Bierstedt, Christian Lemke, Daniel Ritter, Dong Won Hwang, Erda Sheshi, Felix Schabernack, Frank Renkes, et al. 2022. Testing Very Large Database Management Systems: The Case of SAP HANA. *Datenbank-Spektrum* (2022), 1–21.

[3] Peter Boncz, Angelos-Christos Anatiotis, and Steffen Kläbe. 2018. JCC-H: Adding Join Crossing Correlations with Skew to TPC-H. In *Performance Evaluation and Benchmarking for the Analytics Era*, Raghunath Nambiar and Meikel Poess (Eds.). Springer International Publishing, Cham, 103–119.

[4] Peter Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 61–76.

[5] Milind Chabbi and John Mellor-Crummey. 2012. DeadSpy: A Tool to Pinpoint Program Inefficiencies. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (San Jose, California) *(CGO '12)*. Association for Computing Machinery, New York, NY, USA, 124–134. https://doi.org/10.1145/2259016.2259033

[6] Donald D Chamberlin and Raymond F Boyce. 1974. SEQUEL: A structured English query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*. 249–264.

[7] Surajit Chaudhuri. 1998. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 34–43.

[8] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 2002. Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543* (2002).

[9] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. 2018. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–27.

[10] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity estimation for range predicates using lightweight models. *Proceedings of the VLDB Endowment* 12, 9 (2019), 1044–1057.

[11] Lukas Eder. 2022. Say NO to Venn Diagrams When Explaining JOINs. https://blog.jooq.org/say-no-to-venn-diagrams-when-explaining-joins/. Accessed: 2022-11-15.

[12] Pit Fender and Guido Moerkotte. 2013. Counter strike: Generic top-down join enumeration for hypergraphs. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1822–1833.

[13] Pit Fender, Guido Moerkotte, Thomas Neumann, and Viktor Leis. 2012. Effective and robust pruning for top-down join enumeration algorithms. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 414–425.

[14] Campbell Fraser, Leo Giakoumakis, Vikas Hamine, and Katherine F Moore-Smith. 2012. Testing cardinality estimation models in SQL Server. In *Proceedings of the Fifth International Workshop on Testing Database Systems*. 1–7.

[15] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2022. Griffin: Grammar-Free DBMS Fuzzing. In *Conference on Automated Software Engineering (ASE'22)*.

[16] Jana Giceva, Gustavo Alonso, Timothy Roscoe, and Tim Harris. 2014. Deployment of query plans on multicores. *Proceedings of the VLDB Endowment* 8, 3 (2014), 233–244.

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

[18] Zhongxian Gu, Mohamed A Soliman, and Florian M Waas. 2012. Testing the accuracy of query optimizers. In *Proceedings of the Fifth International Workshop on Testing Database Systems*. 1–6.

[19] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2022. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *Proc. VLDB Endow.* 15, 4 (apr 2022), 752–765. https://doi.org/10.14778/3503585.3503586

[20] Max Heimel, Martin Kiefer, and Volker Markl. 2015. Self-tuning, GPU-accelerated kernel density models for multidimensional selectivity estimation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1477–1492.

[21] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *Proceedings of the VLDB Endowment* 13, 7, 992–1005.

[22] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation. In *32st USENIX Security Symposium (USENIX Security 23)*.

[23] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-World Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/2254064.2254075

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

[25] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking Distributed Stream Data Processing Systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 1507–1518. https://doi.org/10.1109/ICDE.2018.00169

[26] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677* (2018).

[27] Christoph Laaber, Joel Scheuner, and Philipp Leitner. 2019. Software microbenchmarking in the cloud. How bad is it really? *Empirical Software Engineering* 24, 4 (2019), 2469–2508.

[28] Doug Laney et al. 2001. 3D data management: Controlling data volume, velocity and variety. *META group research note* 6, 70 (2001), 1.

[29] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[30] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling.. In *Cidr*.

[31] Alon Y Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. 1994. Query optimization by predicate move-around. In *VLDB*. 96–107.

[32] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data*. 615–629.

[33] Yu Liang, Song Liu, and Hong Hu. 2022. Detecting Logical Bugs of DBMS with Coverage-based Guidance. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 4309–4326.

[34] Xinyu Liu, Qi Zhou, Joy Arulrai, and Alessandro Orso. 2022. Automatic Detection of Performance Bugs in Database Systems using Equivalent Queries. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 225–236. https://doi.org/10.1145/3510003.3510093

[35] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing Wrong Data without Doing Anything Obviously Wrong!. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. Association for Computing Machinery, New York, NY, USA, 265–276. https://doi.org/10.1145/1508244.1508275

[36] Thomas Neumann. 2009. Query simplification: graceful degradation for join-order optimization. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 403–414.

[37] Johns Paul, Jiong He, and Bingsheng He. 2016. GPL: A GPU-based pipelined query processing engine. In *Proceedings of the 2016 International Conference on Management of Data*. 1935–1950.

[38] Meikel Poess, Tilmann Rabl, and Hans-Arno Jacobsen. 2017. Analysis of TPC-DS: The First Standard Benchmark for SQL-Based Big Data Systems. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 573–585. https://doi.org/10.1145/3127479.3128603

[39] Viswanath Poosala and Yannis E Ioannidis. 1997. Selectivity estimation without the attribute value independence assumption. In *VLDB*, Vol. 97. 486–495.

[40] Mark Raasveldt, Pedro Holanda, Tim Gubner, and Hannes Mühleisen. 2018. Fair Benchmarking Considered Difficult: Common Pitfalls In Database Performance Testing. In *Proceedings of the Workshop on Testing Database Systems* (Houston, TX, USA) *(DBTest'18)*. Association for Computing Machinery, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/3209950.3209955

[41] Kim-Thomas Rehmann, Changyun Seo, Dongwon Hwang, Binh Than Truong, Alexander Boehm, and Dong Hun Lee. 2016. Performance monitoring in sap hana's continuous integration process. *ACM SIGMETRICS Performance Evaluation Review* 43, 4 (2016), 43–52.

[42] Manuel Rigger and Zhendong Su. 2020. 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* (Sacramento, California, United States) *(ESEC/FSE 2020)*. https://doi.org/10.1145/3368089.3409710

[43] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 211 (2020). https://doi.org/10.1145/3428279

[44] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Banff, Alberta.

[45] Sergio Segura and Zhi Quan Zhou. 2018. Metamorphic testing 20 years later: A hands-on introduction. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 538–539.

[46] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. 23–34.

[47] Michael Stonebraker, Sam Madden, and Pradeep Dubey. 2013. Intel" big data" science and technology center vision and execution plan. *ACM SIGMOD Record* 42, 1 (2013), 44–49.

[48] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. Cobra: Making Transactional {Key-Value} Stores Verifiably Serializable. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 63–80.

[49] Jialiang Tan, Shuyin Jiao, Milind Chabbi, and Xu Liu. 2020. What every scientific programmer should know about compiler optimizations?. In *Proceedings of the 34th ACM International Conference on Supercomputing*. 1–12.

[50] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding missed optimizations through the lens of dead code elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 697–709.

[51] Hélène Touzet. 2003. Tree edit distance with gaps. *Inform. Process. Lett.* 85, 3 (2003), 123–129. https://doi.org/10.1016/S0020-0190(02)00369-1

[52] Website. 1988. TPC-DS Benchmark. https://www.tpc.org/tpcds/. Accessed: 2022-11-15.

[53] Website. 1988. TPC-H Benchmark. https://www.tpc.org/tpch/. Accessed: 2022-11-15.

[54] Website. 2013. American Fuzzy Lop (AFL) Fuzzer. http://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed: 2022-11-15.

[55] Website. 2015. SQLsmith. https://github.com/anse1/sqlsmith. Accessed: 2022-11-15.

[56] Website. 2020. Dynamic programming and edit distance. https://www.cs.jhu.edu/~langmea/resources/lecture_notes/dp_and_edit_dist.pdf. Accessed: 2022-11-15.

[57] Website. 2022. PostgreSQL. https://www.postgresql.org/docs/current/row-estimation-examples.html. Accessed: 2022-11-15.

[58] Website. 2022. What is Database. https://www.oracle.com/database/what-is-database. Accessed: 2022-11-15.

[59] Shasha Wen, Xu Liu, and Milind Chabbi. 2015. Runtime Value Numbering: A Profiling Technique to Pinpoint Redundant Computations. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 254–265. https://doi.org/10.1109/PACT.2015.29

[60] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigümüs, and Jeffrey F Naughton. 2013. Predicting query execution time: Are optimizer cost models really unusable?. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 1081–1092.

[61] Ziniu Wu, Amir Shaikhha, Rong Zhu, Kai Zeng, Yuxing Han, and Jingren Zhou. 2020. BayesCard: Revitilizing Bayesian Frameworks for Cardinality Estimation. *arXiv preprint arXiv:2012.14743* (2020).

[62] Ziniu Wu, Pei Yu, Peilun Yang, Rong Zhu, Yuxing Han, Yaliang Li, Defu Lian, Kai Zeng, and Jingren Zhou. 2021. A unified transferable model for ml-enhanced dbms. *arXiv preprint arXiv:2105.02418* (2021).

[63] Ziniu Wu, Rong Zhu, Andreas Pfadler, Yuxing Han, Jiangneng Li, Zhengping Qian, Kai Zeng, and Jingren Zhou. 2020. FSPN: A New Class of Probabilistic Graphical Model. *arXiv preprint arXiv:2011.09020* (2020).

[64] Khaled Yagoub, Peter Belknap, Benoit Dageville, Karl Dias, Shantanu Joshi, and Hailing Yu. 2008. Oracle's SQL Performance Analyzer. *IEEE Data Eng. Bull.* 31, 1 (2008), 51–58.

[65] Jiaqi Yan, Qiuye Jin, Shrainik Jain, Stratis D Viglas, and Allison Lee. 2018. Snowtrail: Testing with production queries on a cloud database. In *Proceedings of the Workshop on Testing Database Systems*. 1–6.

[66] Junwen Yang, Cong Yan, Chengcheng Wan, Shan Lu, and Alvin Cheung. 2019. View-Centric Performance Optimization for Database-Backed Web Applications. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 994–1004. https://doi.org/10.1109/ICSE.2019.00104

[67] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2021. NeuroCard: One Cardinality Estimator for All Tables. *Proceedings of the VLDB Endowment* 14, 1, 61–73.

[68] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep Unsupervised Cardinality Estimation. *Proceedings of the VLDB Endowment* 13, 3, 279–292.

[69] Andreas Zeller. 2009. *Why programs fail: a guide to systematic debugging*. Elsevier.

[70] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random sampling over joins revisited. In *Proceedings of the 2018 International Conference on Management of Data*. 1525–1539.

[71] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S Yang, Bill W Zhao, and Shashank Singh. 2014. Torturing databases for fun and profit. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 449–464.

[72] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. Squirrel: Testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 955–970.

[73] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2021. FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation. *VLDB* (2021).