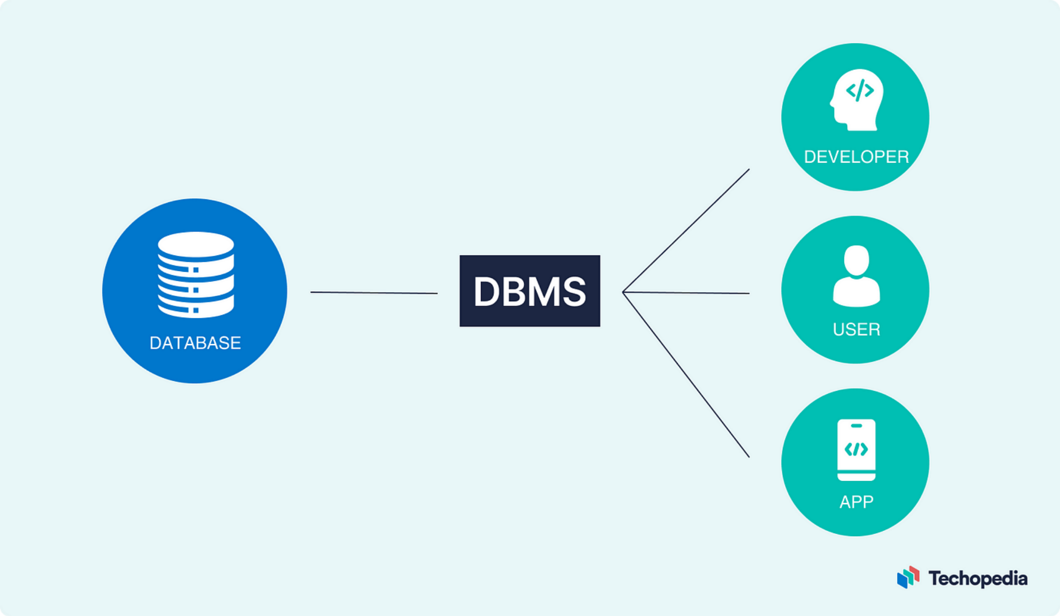
Testing Database Engines via Query Plans

Jinsheng Ba

National University of Singapore

Database Management Systems (DBMSs)

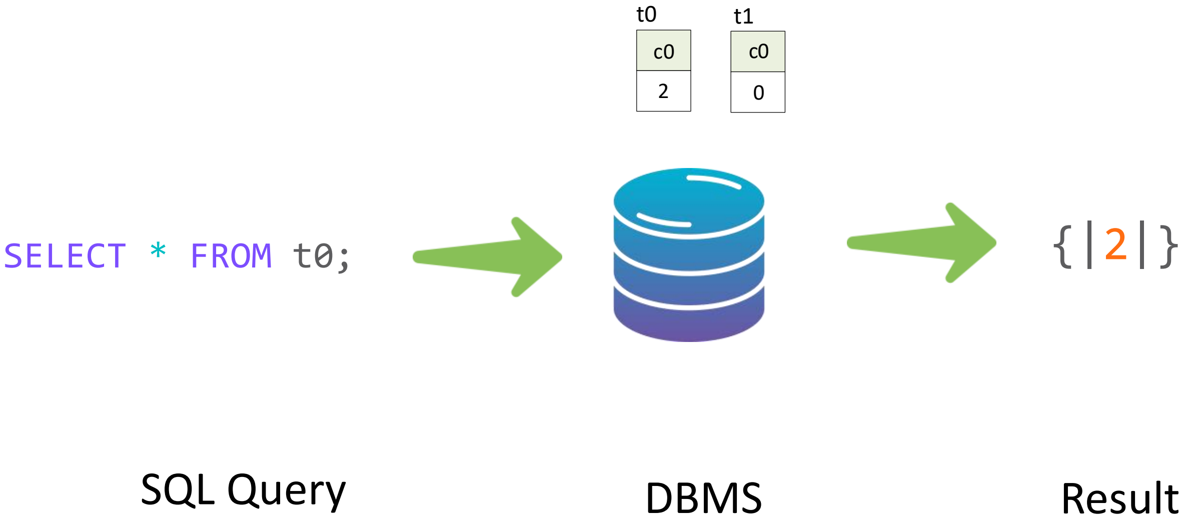


Systems that store, process, manipulate, and query data.

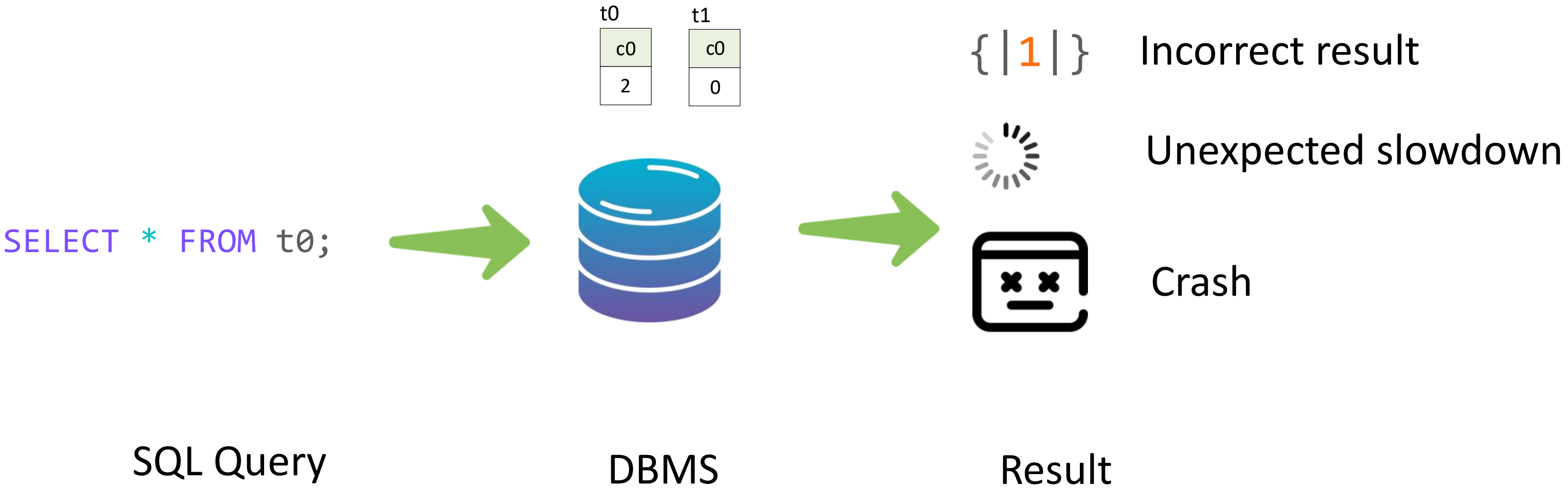
The global DBMSs market has grown to $163.93 billion in 2023 at a high compound annual growth rate of 15.4%*.

* https://www.researchandmarkets.com/reports/5735140/database-software-global-market-report#product--related-products.
Figure: https://medium.com/@sewwandithilakarathna2000/dbms-database-management-system-bc2d86bdba2

Database Management Systems (DBMSs)



Database Management Systems (DBMSs)



Problem: How to efficiently and effectively find bugs in DBMSs?

Core Challenges For Automatic Testing

- Test oracle (bug identification)
  - Logic bugs: How to know the result is incorrect?
  - Performance issues: How to know an execution time is unexpected?

{ **0** }

1s? 5s?

- Test case generation
  - How to automatically explore huge states of target systems?

State-of-the-art Research

Fuzzing for crash bugs

Zhong et al. Squirrel (CCS’20)
Liang et al. LEGO (ICDE’23)
Jiang et al. DynSQL (SEC’23)
Fu et al. Sedar (ICSE’24)

Cannot find logic bugs and performance issues.

Grammar-based Test Cases Generation

Seltenreich et al. (SQLSmith)
Fu et al. Griffin (ASE’22)
Liang et al. SQLRight (SEC’23)

Restricted test cases.

Differential/Metamorphic testing for logic bugs

Slutz RAGS (VLDB’98)
Rigger et al. SQLancer (OSDI’20, ESEC/FSE’20, OOPSLA’20)
Song et al. DQE (ICSE’23)

Cannot generate diverse test cases.

DBMS internal states are not considered.

neering for logic bugs

SIGMOD’23)

Not intuitive to understand.

Differential/Metamorphic testing for performance bugs

Liu et al. AMOEBA (ICSE’22)
Jung et al. APOLLO (VLDB’22)

Find regression bugs only or has a high false alarm rate.

Thesis Statement

Efficient and effective testing of database engines can be achieved by utilizing the internal execution information provided by query plans.

What is a Query Plan?

- A query plan is a tree of operations that specifies how a SQL statement is executed by a specific DBMS.

t0

c0
1
2
3
4
5
6
7
8
9
10

t1

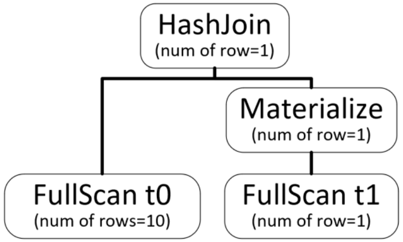
c0
1

```
EXPLAIN SELECT * FROM t0 LEFT JOIN t1
ON t0.c0=t1.c0 WHERE t0.c0=1;
```

SQL

```
HashJoin
 └─TableFullScan
 └─Materialize
    └─TableFullScan
```

Textual Query Plan (Simplified)



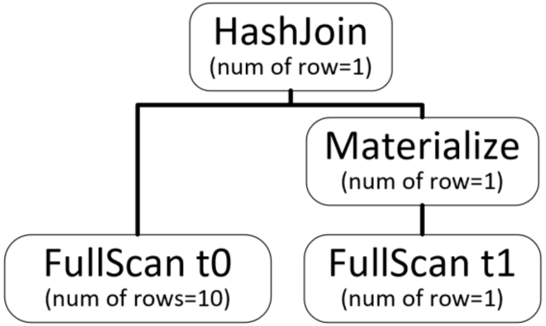
Query Plan

{1|1}

Result

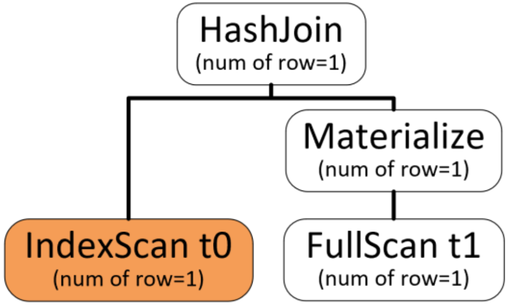
What is a Query Plan?

- A query plan is a tree of operations that describes how a SQL statement is executed by a specific DBMS.
- DBMSs typically expose query plans to users for tuning the performance of queries.

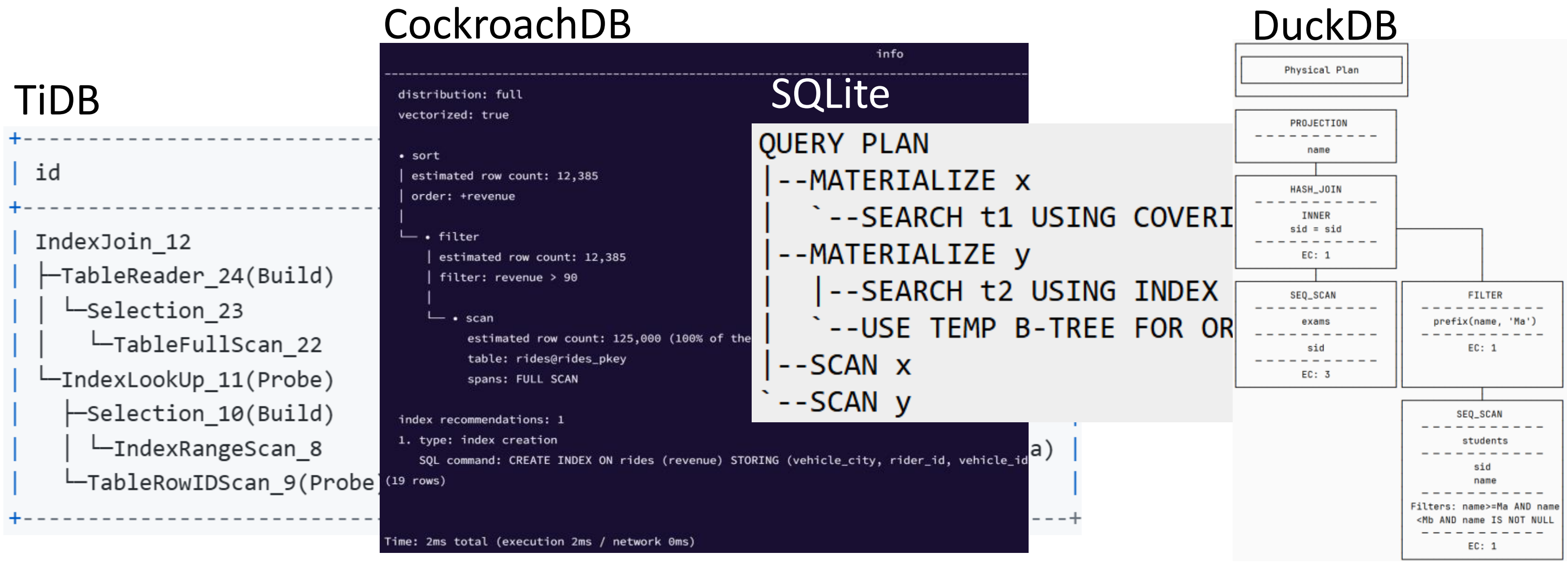



```sql
CREATE INDEX i0 ON t0(c0);

SELECT * FROM t0 LEFT JOIN t1
ON t0.c0=t1.c0 WHERE t0.c0=1;
```



Query Plan Representations



Query plans are represented in DBMS-specific ways, and we empirically studied them*.

* Jinsheng Ba & Manuel Rigger. (2024). Towards a Unified Query Plan Representation.

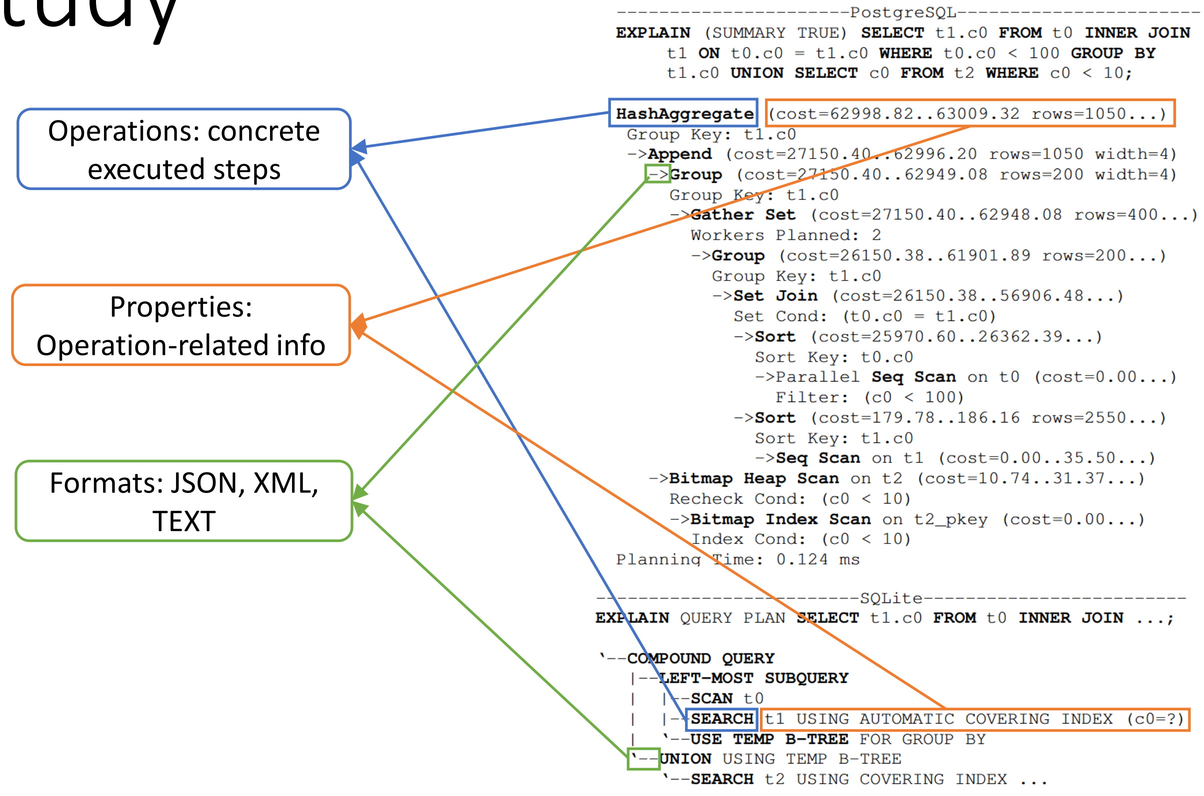
Studied Target DBMSs

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

DBMS	Version	Data Model	Release	Rank
InfluxDB	2.7.0	Time-series	2013	28
MongoDB	6.0.5	Document	2009	5
MySQL	8.0.32	Relational	1995	2
Neo4j	5.6.0	Graph	2007	22
PostgreSQL	14.7	Relational	1989	4
SQL Server	16.0.4015.1	Relational	1989	3
SQLite	3.41.2	Relational	1990	10
SparkSQL	3.3.2	Relational	2014	37
TiDB	6.5.1	Relational	2016	84

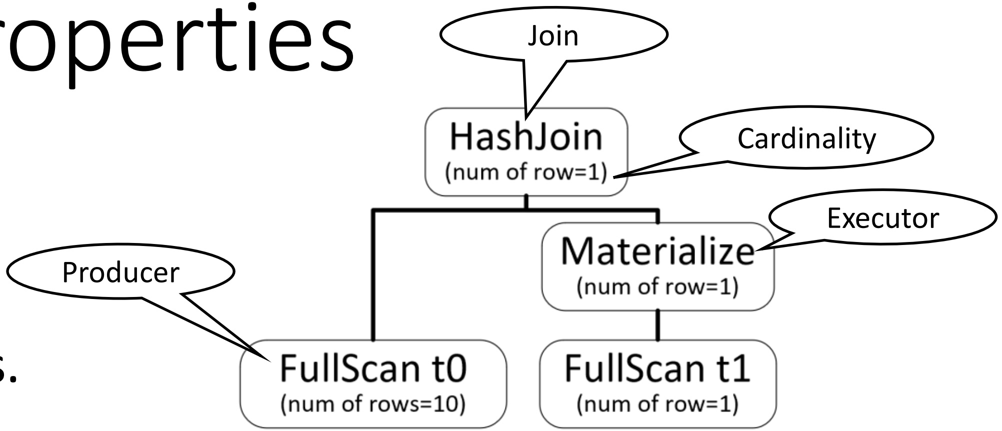
Query Plan Study

- Query plan representations share three conceptual components



Operations and Properties

- According to function signatures and semantic, we classified operations into seven categories and properties into four categories.



	Operations								Properties				
DBMS	Producer	Bag	Join	Folder	Projector	Executor	Consumer	Total	Cardinality	Cost	Configuration	Status	Total
InfluxDB	0	0	0	0	0	0	0	0	5	0	0	1	6
MongoDB	14	9	0	5	3	10	3	44	16	5	18	12	51
MySQL	15	3	2	1	0	2	0	23	3	6	3	10	22
Neo4j	18	11	43	6	3	17	13	111	3	3	12	7	25
PostgreSQL	18	8	3	3	0	9	1	42	8	17	42	40	107
SQL Server	15	3	3	3	0	16	19	59	4	4	7	3	18
SQLite	3	6	3	0	0	5	0	17	0	0	3	0	3
SparkSQL												0	22
TiDB												1	12
Avg:												8	30

Query plan representations are commonly supported and share common components, so we can develop general testing approaches.

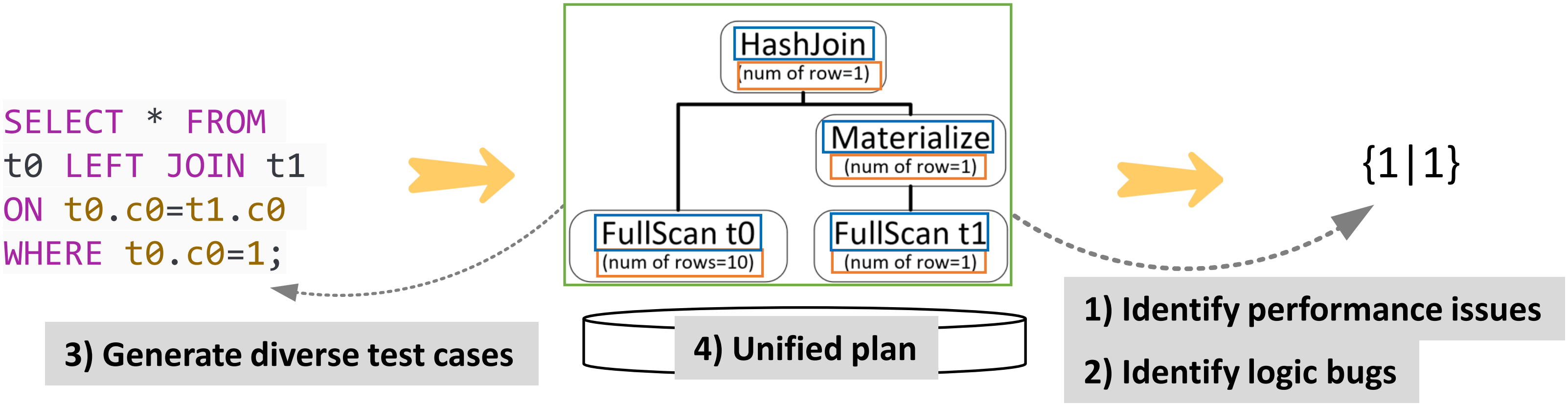
Research Overview

1) Test oracle: identifying performance issues: **Jinsheng Ba** & Manuel Rigger. (2024). Finding Performance Issues in Database Engines via Cardinality Estimation Testing. In Proceedings of International Conference on Software Engineering (ICSE).

2) Test oracle: identifying logic bugs in a simple way: **Jinsheng Ba** & Manuel Rigger. (2024). Keep It Simple: Testing Databases via Differential Query Plans. In Proceeding of ACM Management of Data (SIGMOD)

3) Test case generation: generating diverse test cases: **Jinsheng Ba** & Manuel Rigger. (2023). Testing Database Engines via Query Plan Guidance. In Proceedings of International Conference on Software Engineering (ICSE). 🏆

4) Building general applications on query plans: **Jinsheng Ba** & Manuel Rigger. (2024). Towards a Unified Query Plan Representation. (Under submission).



Cardinality Estimation Restriction Testing (CERT)

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.

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

Problem: How to Identify Performance Issues?

```sql
SELECT * FROM t0 LEFT JOIN t1 ON t0.c0=t1.c0 WHERE t0.c0==1;
```

Query

3 seconds

{|1|2|}

Results

Is it an unexpected bad performance?

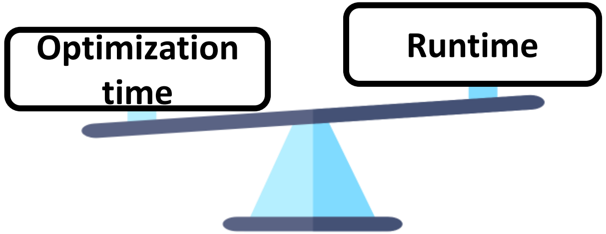
Challenge: No Ground Truth

- No ground truth (test oracle) of a reasonable execution time
- Cannot be expected to achieve optimal efficiency as they make various **tradeoffs** to balance **optimization time** and **execution time**

0.1s – 1s ?

1s – 5s ?

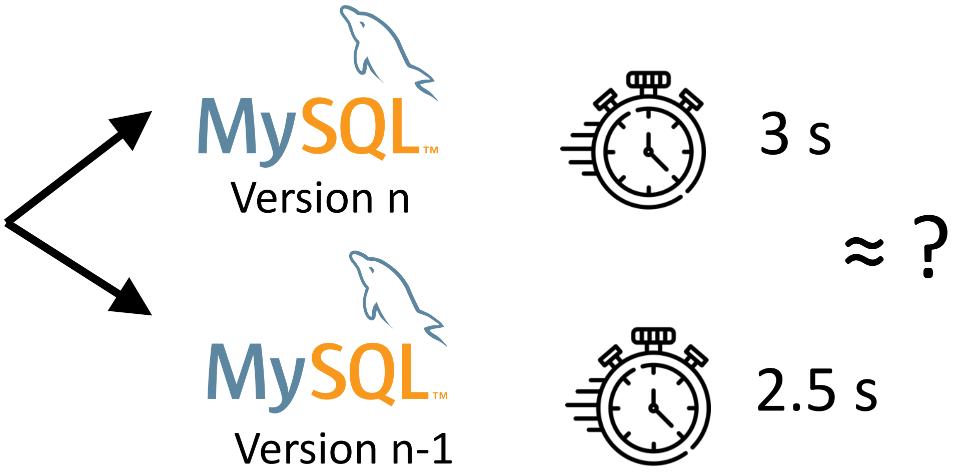
5s – 20s ?



Existing Solution: Differential Testing

- APOLLO[1]
- Selecting an old and new version of a DBMS enables finding **only regression bugs**

```
SELECT * FROM t0
WHERE c0=0;
```



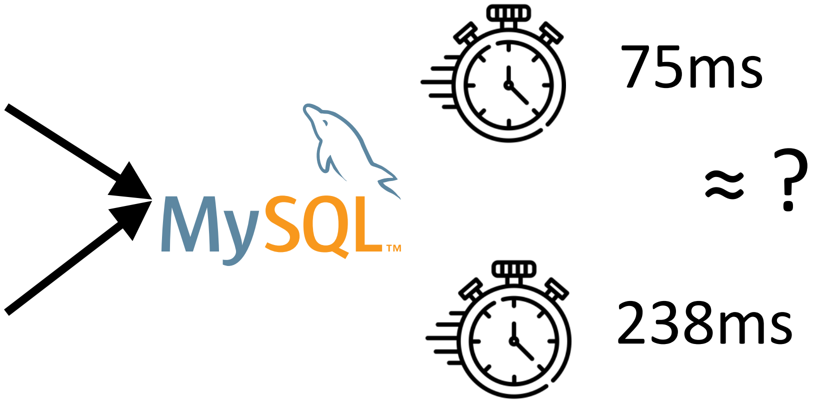
[1] Jung, J., Hu, H., Arulraj, J., Kim, T., & Kang, W. Apollo: Automatic detection and diagnosis of performance regressions in database systems. VLDB Endowment, 13(1), 57-70.

Existing Solution: Equivalent Queries

- AMOEBA[1]
- Generating equivalent queries (or programs) might result in many **false alarms as** only 6/39 reported issues are confirmed.

```
SELECT Max(emp.sal)FROM dept INNER JOIN emp
ON ename NOT LIKE nameWHERE name ='ACCT';
```

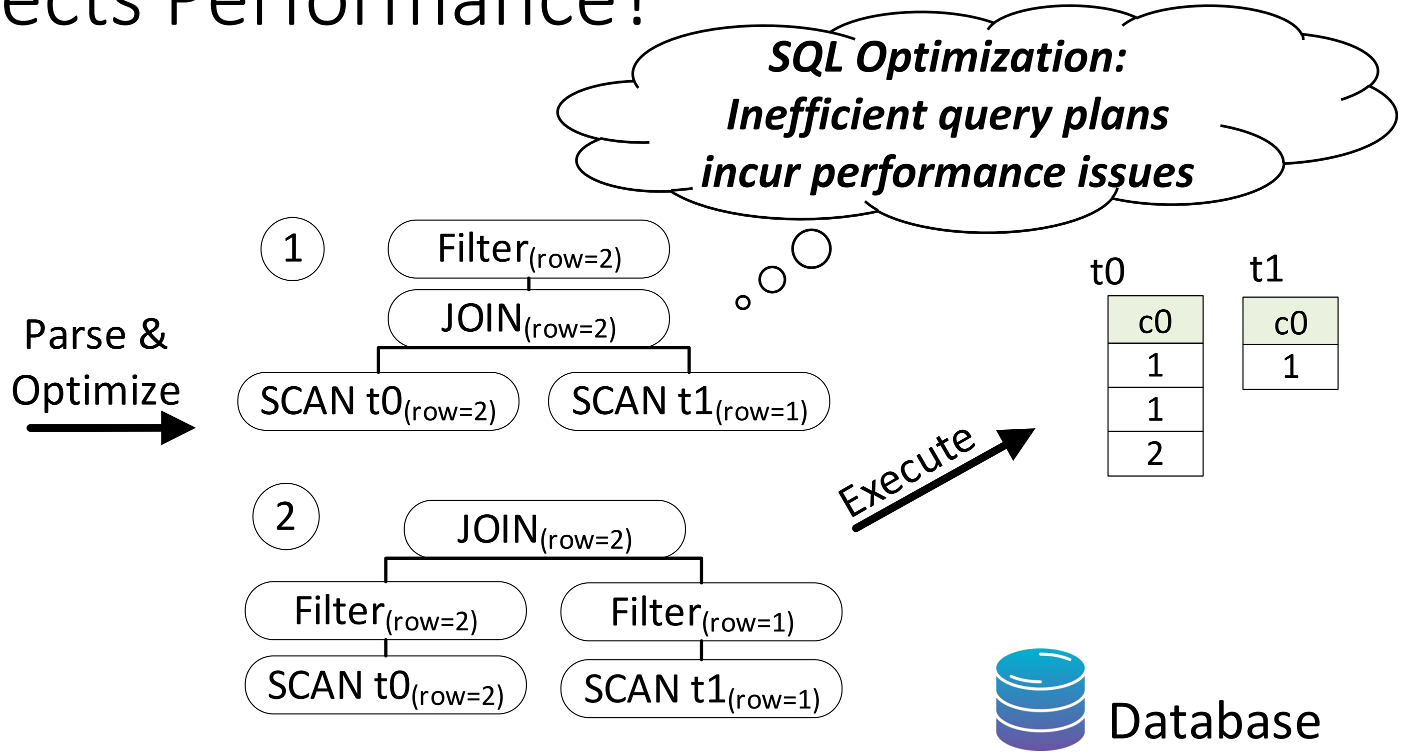
```
SELECT Max(emp.sal)FROM dept INNER JOIN emp ON
ename NOT LIKE nameWHERE name ='ACCT'IS TRUE;
```



[1] Liu, X., Zhou, Q., Arulraj, J., & Orso, A. (2022, May). Automatic detection of performance bugs in database systems using equivalent queries. In Proceedings of the 44th International Conference on Software Engineering (pp. 225-236).

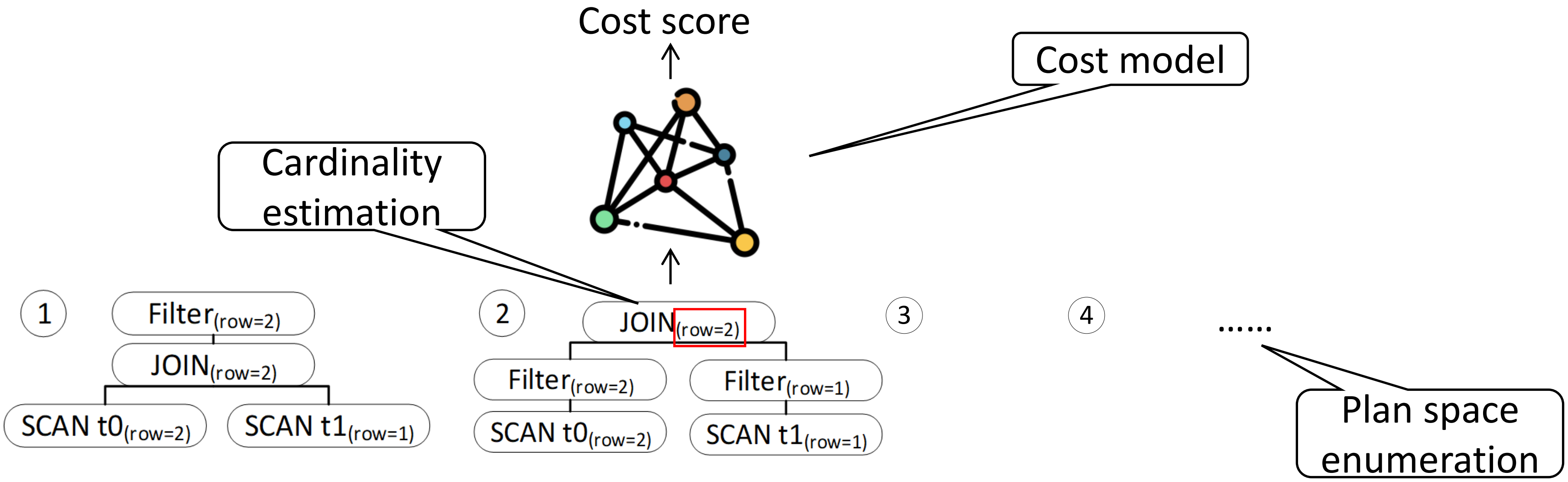
What Affects Performance?

```sql
SELECT * FROM t0
LEFT JOIN t1 ON
t0.c0=t1.c0 WHERE
t0.c0==1;
```



Simplified Query Plans

What Constitute SQL Optimization?



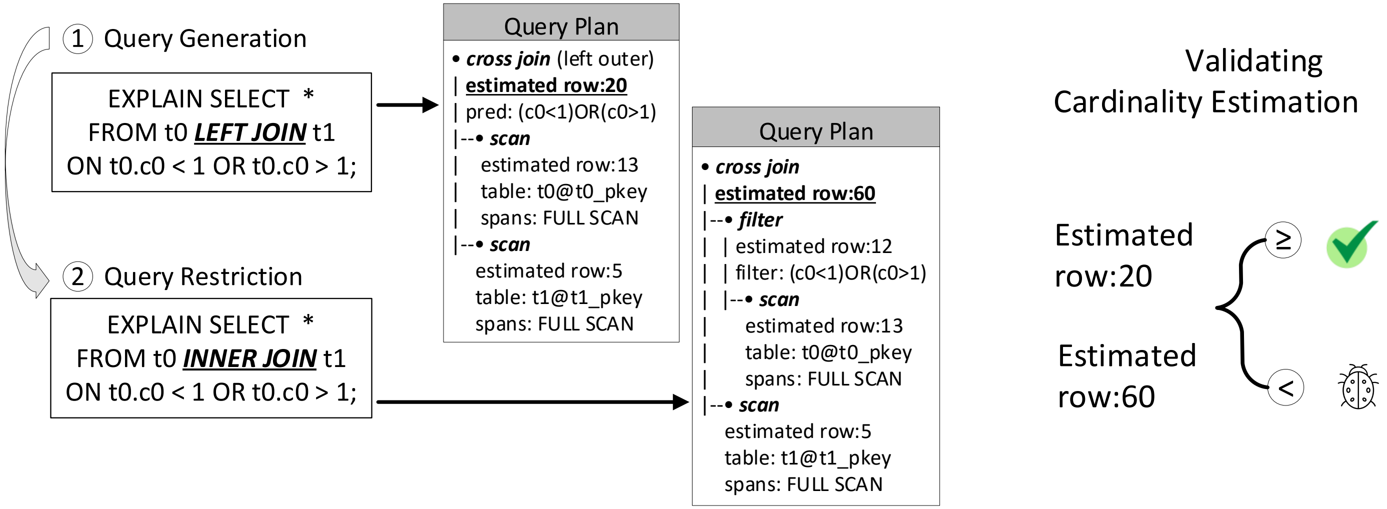
Cardinality estimation is the **most important part of** query optimization[1]

[1] Leis, V., Gubichev, A., Mirchev, A., Boncz, P., Kemper, A., & Neumann, T. (2015). How good are query optimizers, really?. Proceedings of the VLDB Endowment, 9(3), 204-215.

Idea

Cardinality Estimation Restriction Testing (CERT) focuses on the most relevant SQL optimization component and eschews executing queries

Cardinality Estimation Restriction Testing (CERT)



<u>Cardinality Restriction Monotonicity Property</u>: a given query should not fetch fewer rows than a more restrictive query derived from it.

How to restrict queries?

- We propose **12 rules** covering the common clauses of a query.

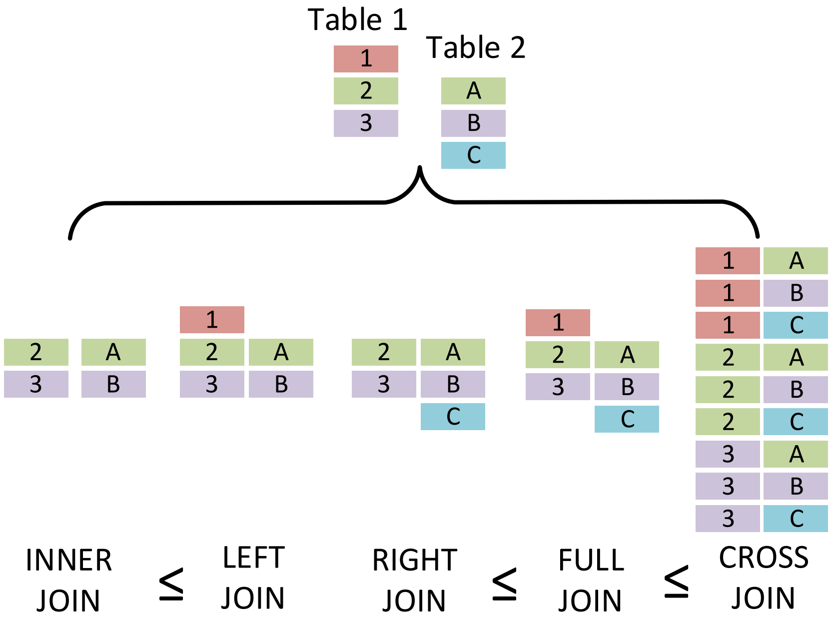
```
SELECT
[ALL | DISTINCT]
select_expression [,
select_expression ...]
FROM table_reference [INNER | LEFT
| RIGHT | FULL | CROSS JOIN
table_reference ...]*
[WHERE where_condition ]
[GROUP BY column_expression
[HAVING where_condition ]]
[LIMIT row_count ];
```

How to restrict queries?

SELECT ~~ALL~~ DISTINCT * FROM t0;

```
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 ];
```

How to restrict queries?




```
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 ];
```

How to restrict queries?

SELECT * FROM t0 WHERE c0>0;

SELECT * FROM t0 WHERE c0>0 AND c0!=8;

SELECT * FROM t0 WHERE c0>0 ~~OR c0!=8~~;

```
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 ];
```

How to restrict queries?

```
SELECT * FROM t0 GROUP BY c0;
```

```
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 ];
```

How to restrict queries?

SELECT * FROM t0 GROUP BY c0
HAVING c0>0;

```
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 ];
```

How to restrict queries?

SELECT * FROM t0 LIMIT ~~10~~ 5;

```
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 ];
```

* These rules are not exhaustive, and we just propose several promising rules to cover common SQL clauses.

How to Avoid False Positive?

```
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
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
```

```
`  filter
|  estimated row:2
|-`  cross join(full)
     |  estimated row:6
     |-`  scan (t1)
     |      estimated row:4
     |-`  scan (t0)
            estimated row:2
```

```
`  cross join(right)
|  estimated row:3
|-`  scan (t0)
|      estimated row:2
|-`  filter
     |  estimated row:1
     |-`  scan (t1)
            estimated row:4
```

The two query plans are **significantly different**, so developers consider their query plans **incomparable**

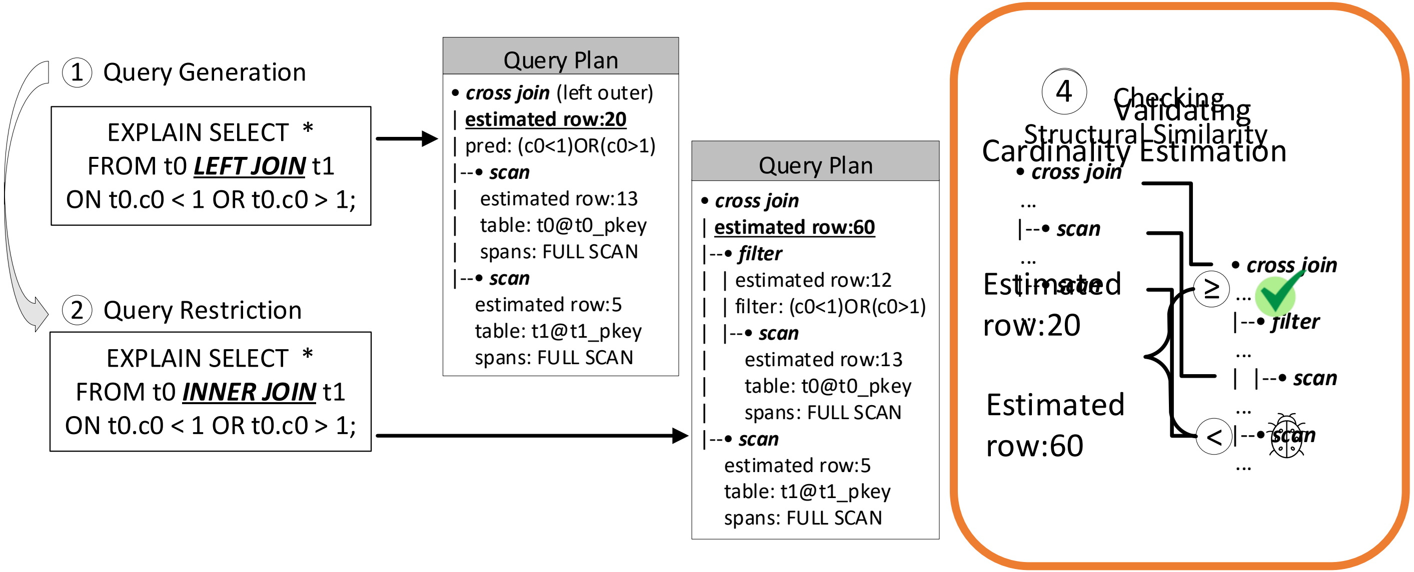
Comparable Query Plans

```
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
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
```

```
`  filter                          `  cross join(right)
|  estimated row:2                 |  estimated row:3
|-`  cross join(full)              |-`  scan (t0)
     |  estimated row:6            |      estimated row:2
     |-`  scan (t1)                |-`  filter
     |       estimated row:4           |  estimated row:1
     |-`  scan (t0)                    |-`  scan (t1)
             estimated row:2                   estimated row:4
```

Two query plans are comparable only when the ***edit distance*** of the two query plans' operation sequences is no more than one

Checking Structural Similarity



Evaluation: Issues Found

	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

We reported 13 unique performance issues, in which 11 were confirmed or fixed.

Evaluation: Performance Analysis

```
CREATE TABLE t0 (c0 INT);
CREATE TABLE t1 (c0 INT);
CREATE TABLE t2 (c0 INT);
INSERT INTO t0 SELECT * FROM generate_series(1,1000);
INSERT INTO t1 SELECT * FROM generate_series(1001,2000);
INSERT INTO t2 SELECT * FROM generate_series(1,333100);
```

ISSUE 88455: `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`

ISSUE 89161: `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`

The fixes improves query performance by 19% for CockroachDB on average.

Bug Analysis

```
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[2]
```

The hash join[1] loads into memory the second child's data, which is expected smaller than second child.

```
• group (scalar)
│ estimated row count: 1
└── • hash join (full outer)
    │ estimated row count: 335,603
    ├── • scan
    │     estimated row count: 333,100
    │     table: t2@t2_pkey
    └── • cross join (left outer)
        │ estimated row count: 333,000
        ├── • scan
        │     estimated row count: 1,000
        │     table: t0@t0_pkey
        └── • scan
              estimated row count: 1,000
              table: t1@t1_pkey
```

```
• group (scalar)
│ estimated row count: 1
└── • hash join (full outer)
    │ estimated row count: 1,006,808
    ├── • cross join (left outer)
    │   │ estimated row count: 999,001
    │   ├── • scan
    │   │     estimated row count: 1,000
    │   │     table: t0@t0_pkey
    │   └── • scan
    │         estimated row count: 1,000
    │         table: t1@t1_pkey
    └── • scan
          estimated row count: 333,100
          table: t2@t2_pkey
```

[1] Website, CockroachDB Hash Join, https://www.cockroachlabs.com/docs/stable/joins.html#hash-joins
[2] Website, CockroachDB Issue 88455, https://github.com/cockroachdb/cockroach/issues/88455

Query Plan Guidance (QPG)

Testing Database Engines via Query Plan Guidance

Jinsheng Ba
National University of Singapore

Manuel Rigger
National University of Singapore

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

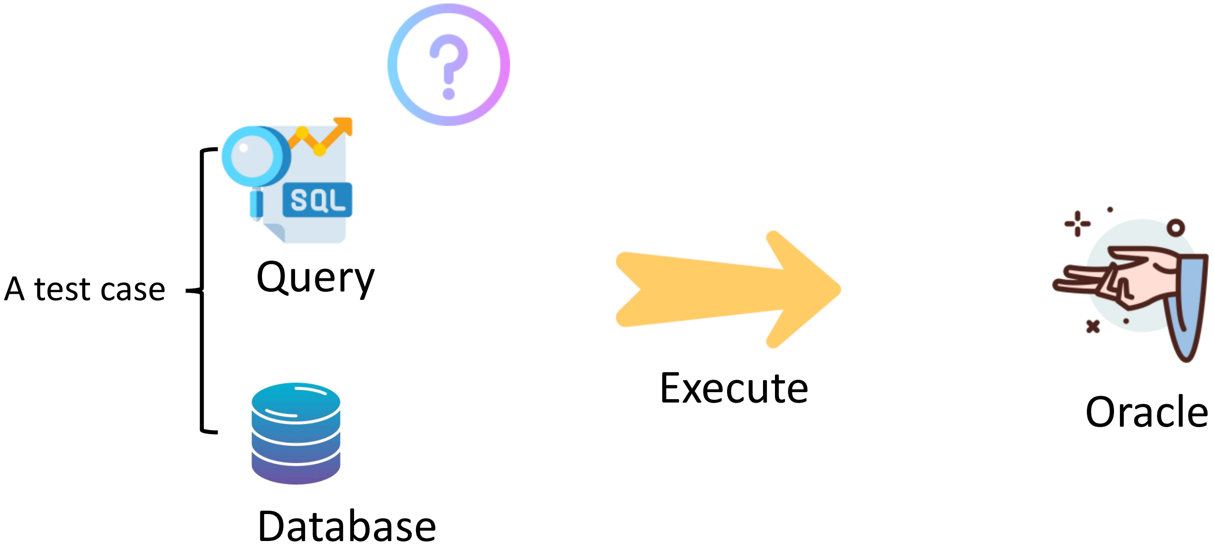
***Index Terms*—automated testing, test case generation**

DBMS to increase the chance of finding bugs in them. No clear definition or metric on what an interesting test case constitutes exists, as it is unknown in advance by which logic bugs a DBMS is affected. Second, the test cases should be valid both syntactically and semantically while also corresponding to the structure imposed by the test oracle; for example, the NoREC oracle requires a query with a `WHERE` clause, but no more complex clauses (e.g., `HAVING` clauses) [7] while also forbidding various functions and keywords from being used (e.g., aggregate functions).

Both generation-based and mutation-based approaches have been proposed to be paired with the above test oracles [6]–[8]. SQLancer uses a generation-based approach in which test cases are generated adhering to the grammar of the respective SQL dialects as well as the constraints imposed by the test oracles. Overall, this approach makes it likely to generate valid test cases; we observed that about 90% of the queries generated by SQLancer for SQLite are valid. However, the test case generation approach receives no guidance that could steer it towards producing interesting test cases. Recently, SQL-Right [9] was proposed to address this shortcoming. SQLRight mutates test cases aiming to maximize the DBMS' covered code, thus building on the success of grey-box fuzzing [10], [11]. While SQLRight improved on SQLancer's test case generation in various metrics, code coverage alone was shown

Problem: How To Generate Test Cases?

- How do we generate diverse test cases to test DBMSs?



Previous Test Case Generation Methods

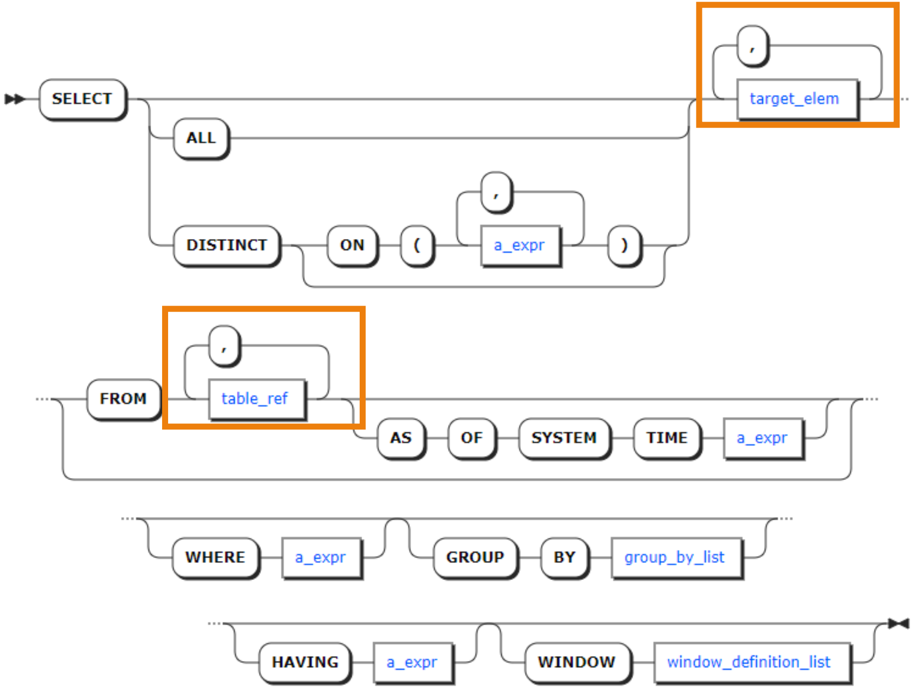
- Generation-based methods.
  - Restricted to the grammar and hard to generate diverse test cases.

```
SELECT c0 FROM t0;

SELECT c0, c1+5 FROM t0;

SELECT c0, c1+5 FROM t0, t1;
```

- Examples: SQLSmith[2], SQLancer[3]



The SQL grammar[1] for CockroachDB.

[1] Website, "CockroachDB SELECT Clause", https://www.cockroachlabs.com/docs/stable/select-clause.html
[2] Website, "SQLSmith", https://github.com/anse1/sqlsmith
[3] Website, "SQLancer", https://github.com/sqlancer/sqlancer

Previous Test Case Generation Methods

- Mutation-based methods (Coverage-guided Grey-box fuzzing).
  - Insufficient proportion of valid test cases. (SQLRight[1]: 40%)
  - Code coverage is insufficient to explore DBMSs' bugs.

```
SELECT * FROM t0;

SEL?CT * FROM t0;

SEL?CT * FROM t0EOFEOF;
```

100% code coverage does not cover 100% execution logic, such as database states are not accounted for.

SQLite uses `testcase()` macros as described in the previous subsection to make sure that every condition in a bit-vector decision takes on every possible outcome. In this way, SQLite also achieves 100% MC/DC in addition to 100% branch coverage.

SQLite Documents[2].

- Example: SQLRight

[1] Liang, Y., Liu, S., & Hu, H. (2022). Detecting Logical Bugs of {DBMS} with Coverage-based Guidance. In 31st USENIX Security Symposium (USENIX Security 22) (pp. 4309-4326).
[2] Website, "How SQLite Is Tested", https://www.sqlite.org/testing.html#mcdc

Idea

Query Plan Guidance (QPG) steers the test case generation process towards exploring diverse query plans

Query Plan Guidance

```
SELECT * FROM t0 LEFT JOIN t1
WHERE t1.c0==0;
```

```
|--SCAN t0
`--SCAN t1 LEFT-JOIN
    `--Filter
```

{2|0}

Query A

```
|--Filter
|--SCAN t
```

Query B

```
|--SCAN t0
`--USE TEMP B-TREE FOR ORDER BY
```

Query C

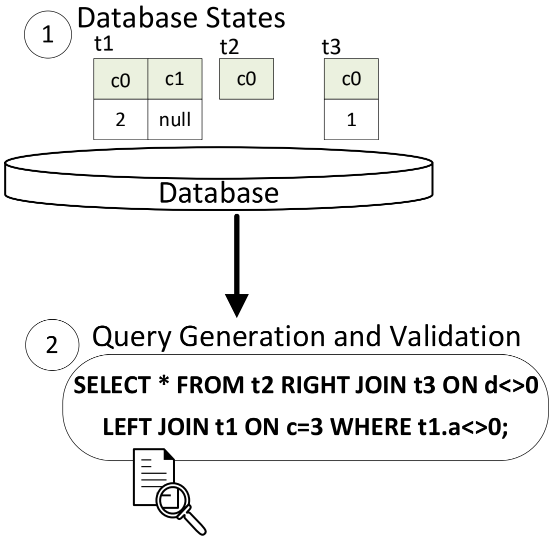
```
|--Filter
|--SCAN t0
`--SCAN t1 LEFT-JOIN
```

…

…

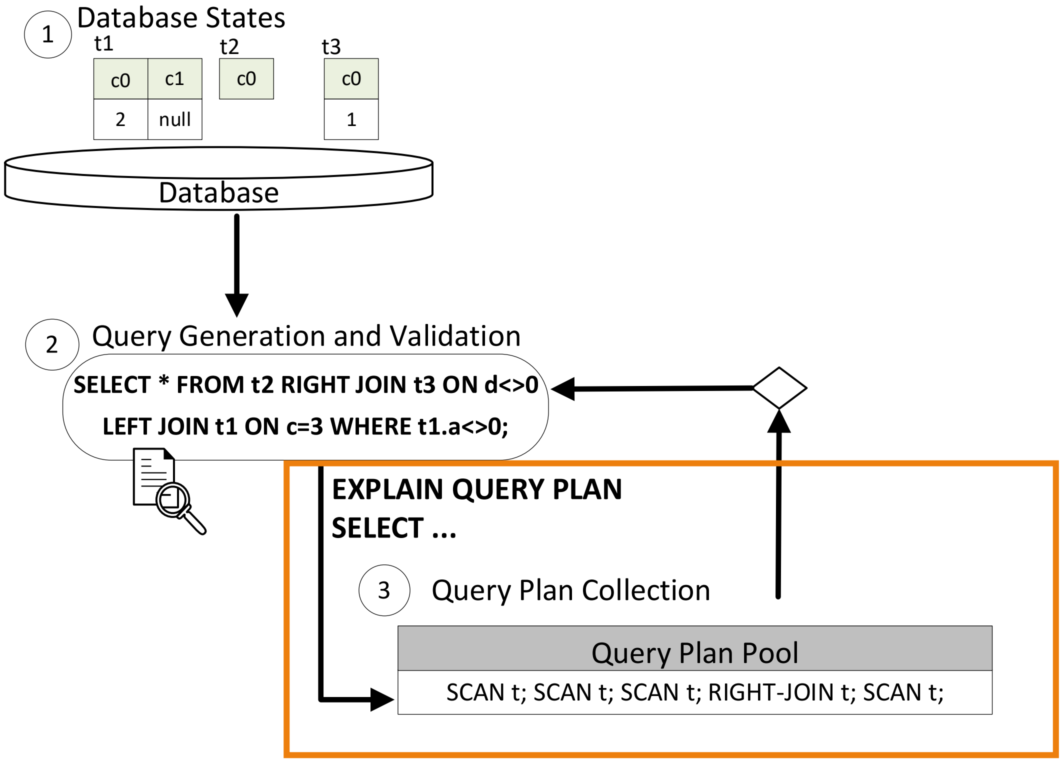
Step 1 & 2: Query Generation and Validation

Reuse the existing database and query generation approaches



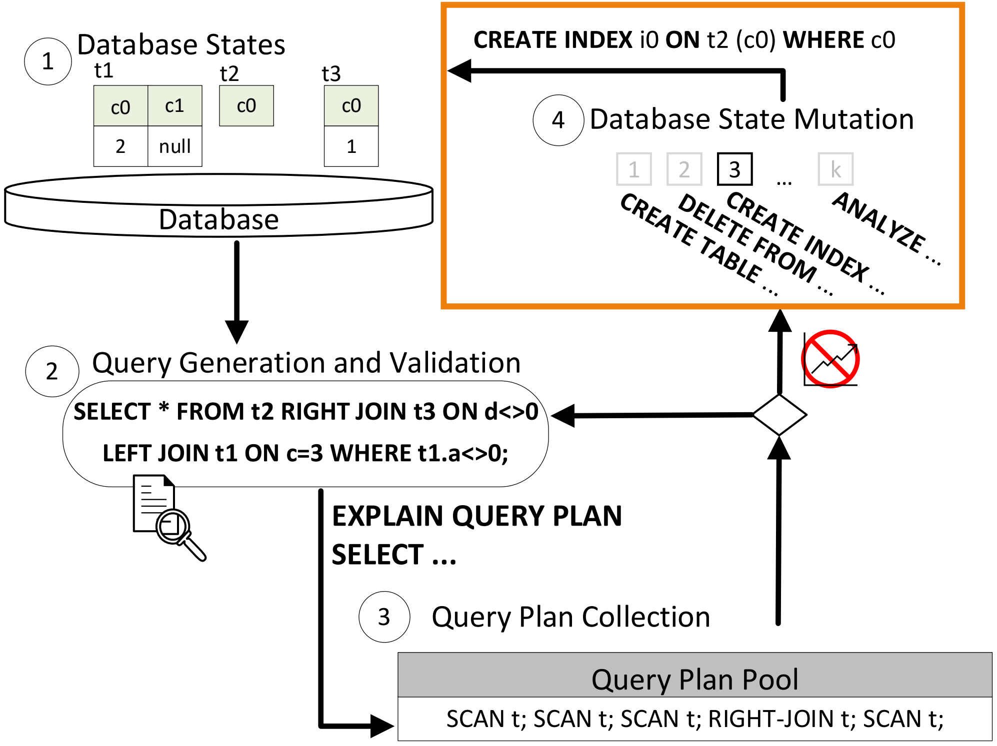
Step 3: Query Plan Collection

Record newly seen query plans



Step 4: Database State Mutation

Mutate the database state if no query plan has been observed for a certain number of iterations



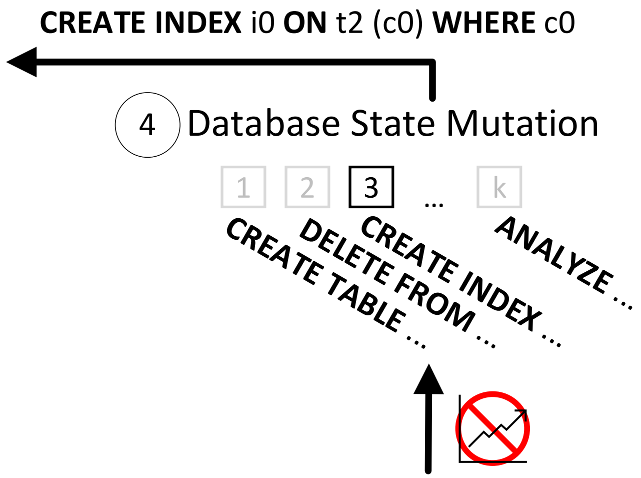
Step 4: Database State Mutation

- Challenge:

  How to apply **promising mutations** that likely result in queries triggering new query plans?

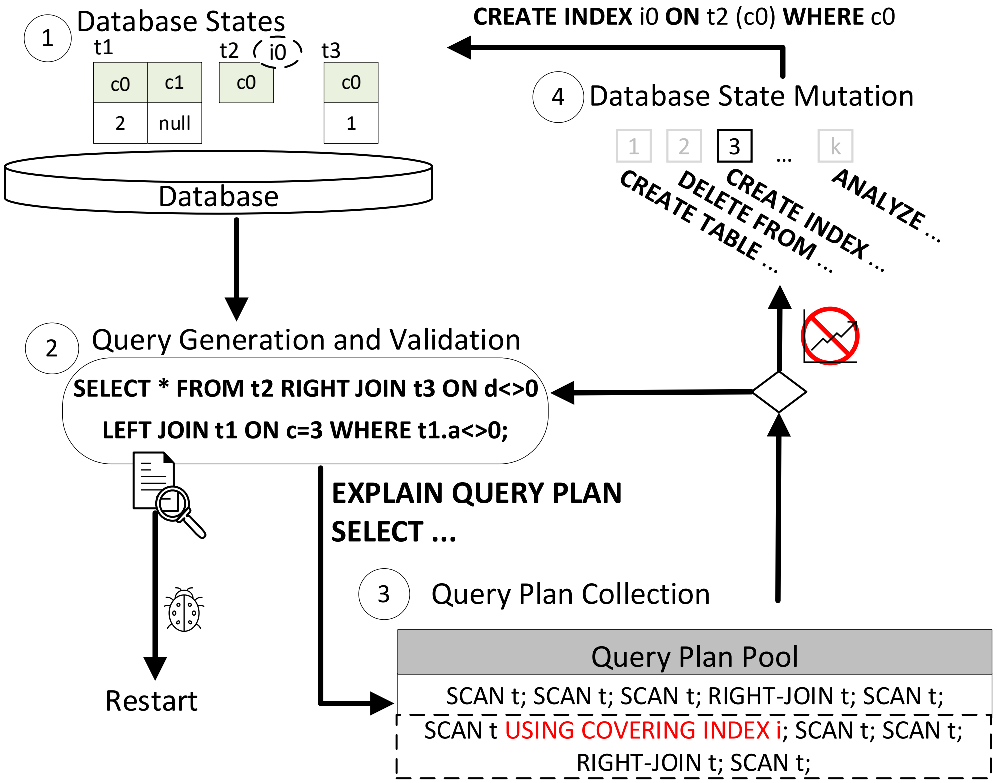
- Solution:

  model as a **multi-armed bandit** problem



Query Plan Guidance (QPG)

New query plans are able to be observed, and new bugs may be found



Evaluation: New Bugs

(2) By Richard Hipp (drh) on 2022-07-15 12:59:59 in reply to 1 [link] [source]

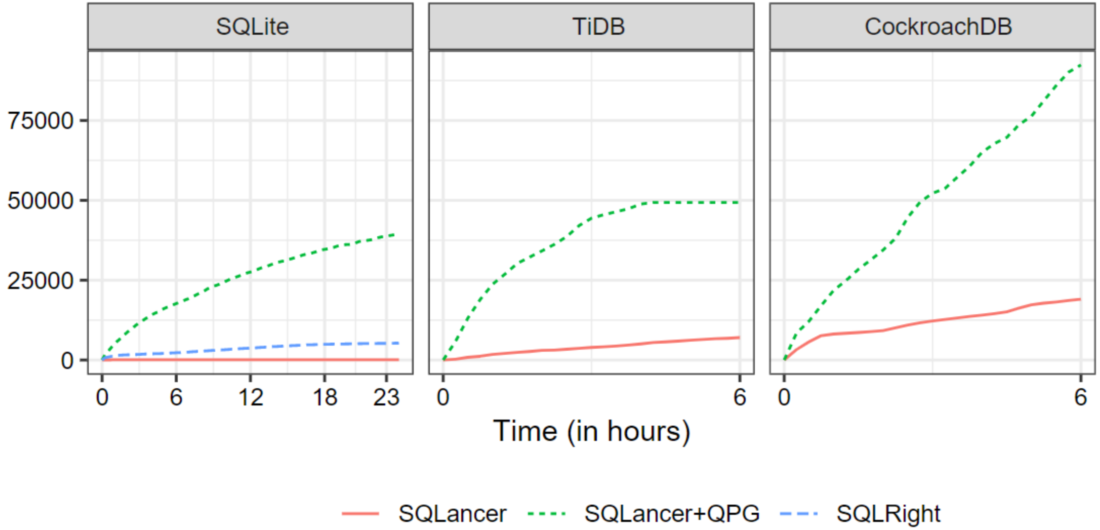
This bug goes back almost 8 years to check-in ddb5f0558c445699 on 2016-09-07, ve

Several bugs had been hidden for more than six years!

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

With the help of QPG, we found 53 unique, previously unknown bugs.

Evaluation: Covering unique query plans



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

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

Differential Query Plans (DQP)

Keep It Simple: Testing Databases via Differential Query Plans

JINSHENG BA, National University of Singapore, Singapore
MANUEL RIGGER, National University of Singapore, Singapore

Query optimizers perform various optimizations, many of which have been proposed to optimize joins. It is pivotal that these optimizations are correct, meaning that they should be extensively tested. Besides manually written tests, automated testing approaches have gained broad adoption. Such approaches semi-randomly generate databases and queries. More importantly, they provide a so-called *test oracle* that can deduce whether the system's result is correct. Recently, researchers have proposed a novel testing approach called *Transformed Query Synthesis* (*TQS*) specifically designed to find logic bugs in join optimizations. *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, in this work, we propose a simple alternative approach to *TQS*. Our approach enforces different query plans for the same query and validates that the results are consistent. We refer to this approach as *Differential Query Plan* (*DQP*) testing. *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. Additionally, *DQP* is complementary to other testing approaches for finding logic bugs. 81% of the logic bugs found by *DQP* cannot be found by *NoREC* and *TLP*, whereas *DQP* overlooked 86% of the bugs found by *NoREC* and *TLP*. We hope that the practicality of our approach—we implemented in less than 100 lines of code per system—will lead to its wide adoption.

CCS Concepts: • **Information systems → Query optimization**; • **Security and privacy → Database and storage security**.

Additional Key Words and Phrases: Join, logic bug

ACM Reference Format:
Jinsheng Ba and Manuel Rigger. 2024. Keep It Simple: Testing Databases via Differential Query Plans. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 188 (June 2024), 26 pages. https://doi.org/10.1145/3654991

Transformed Query Synthesis (TQS)

TQS* is the state-of-the-art approach to realize a test oracle.

(a) Wide Table T_W

RowID	orderId	goodsId	goodsName	userId	userName	price
0	0001	1111	book	str1	Tom→*null*	15
1	0001	1112	food	str1	Tom→*null*	5
2	0002	1111	book	str1	Tom→*null*	15
3	0003	1111	book	str2	Peter	15
4	0003	1112	food	str2	Peter	5
5	0003	1113	flower	str2	Peter	10
6	0004	1111→*noise*	book→*null*	str3	Bob	15→*null*
7	0004	1112	food	str3	Bob	5
8	*null*	*null*	*null*	*noise*	*Tom*	*null*
9	*null*	*1111*	*book*	*null*	*null*	*15*

Validate correctness according to T_w

Result

Split

(b) Schema Tables

Table T_1

RowID	orderId	goodsId	userId
0	0001	1111	str1
1	0001	1112	str1
2	0002	1111	str1
3	0003	1111	str2
4	0003	1112	str2
5	0003	1113	str2
6	0004	1111→*noise*	str3
7	0004	1112	str3

Noise Injection

Table T_2

RowID	userId	userName
0	str1→*noise*	Tom
1	str2	Peter
2	str3	Bob

Table T_3

RowID	goodsId	goodsName
0	1111	book
1	1112	food
2	1113	flower

Table T_4

RowID	goodsName	price
0	book	15
1	food	5
2	flower	10

Query

T1 Join T2 Join T3

*** Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. 2023. Detecting Logic Bugs of Join Optimizations in DBMS. Proc. ACM Manag. Data 1, 1, Article 55.**

TQS Study

We observed that

1) TQS claimed 100+ found bugs, but we only found 21 bug reports and

2) most bugs were reported in a different manner as TQS.

```
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}
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
```

***https://bugs.mysql.com/bug.php?id=106713**

Differential Query Plans (DQP)

1 Database State Generation

user

user_id
1
2

transaction

transaction_id	amount
1_c12934	100000
1_e3b664	-10

i0

Database

2 Query Generation

```
SELECT
IFNULL(SUM(amount), 0) AS balance
FROM user JOIN transaction
ON transaction.transaction_id =
user.user_id;
```

Query Plan

```
nested_loop
+- table
|  table_name: user
|  access_type: index
+- table
|  table_name: transaction
|  access_type: all
```

3 Query Plan Enforcement

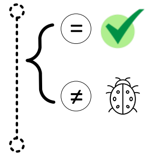
```
SELECT
/*+ JOIN_ORDER(transaction, user)*/
IFNULL(SUM(amount), 0) as balance
FROM user JOIN transaction
ON transaction.transaction_id =
user.user_id;
```

Query Plan

```
nested_loop
+- table
|  table_name: transaction
|  access_type: all
+- table
|  table_name: user
|  access_type: eq_ref
```

4 Result Validation

balance
99990.00



balance
0.00

Evaluation

- 14 of 15 unique bugs found by TQS can be reproduced by our method DQP.

DBMS	Bug	Unique	Join	Query Plan
MySQL	106713	√		√
MySQL	106715	√	√	√
MySQL	106716	√	√	√
MySQL	106717	√		√
MySQL	106718	√		√
MySQL	106611			√
MySQL	106710	√		√
MySQL	99273	√		
MySQL	109211	√	√	√
MySQL	109212	√	√	√
MariaDB	28214	√	√	√
MariaDB	28215	√	√	√
MariaDB	28216	√	√	√
MariaDB	28217	√	√	√
MariaDB	29695	√	√	√
TiDB	33039		√	√
TiDB	33041		√	√
TiDB	33042	√	√	√
TiDB	33045		√	√
TiDB	33046		√	√

Evaluation

- DQP additionally found 26 previously unknown and unique bugs.

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

Unified Query Plan Representation (Uplan)

Towards a Unified Query Plan Representation

Jinsheng Ba
National University of Singapore

Manuel Rigger
National University of Singapore

***Abstract*—In database systems, a query plan is a series of concrete internal steps to execute a query. Multiple testing approaches utilize query plans for finding bugs. However, query plans are represented in a database-specific manner, so implementing these testing approaches requires a non-trivial effort, hindering their adoption. We envision that a unified query plan representation can facilitate the implementation of these approaches. In this paper, we present an exploratory case study to investigate query plan representations in nine widely-used database systems. Our study shows that query plan representations consist of three conceptual components: operations, properties, and formats, which enable us to design a unified query plan representation. Based on it, existing testing methods can be efficiently adopted, finding 17 previously unknown and unique bugs. Additionally, the unified query plan representation can facilitate other applications. Existing visualization tools can support multiple database systems based on the unified query plan representation with moderate implementation effort, and comparing unified query plans across database systems provides actionable insights to improve their performance. We expect that the unified query plan representation will enable the exploration of additional application scenarios.**

***Index Terms*—Case Study, Database, Query Plan, Unified Representation**

of TiDB [10], but corresponds to a property of another step to scan tables in the query plans of PostgreSQL [11]. We refer to the different ways in which serialized query plans are represented as *query plan representations*. Considering that hundreds of DBMSs exist,[1] implementing the above testing methods requires significant effort as they need to account for differences in query plan representations, thus significantly hindering the effectiveness of the above approaches.

We envision that a unified query plan representation would remove the roadblock to implementing the above testing approaches. In this work, we systematically study query plan representations. We present an exploratory case study [12], 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

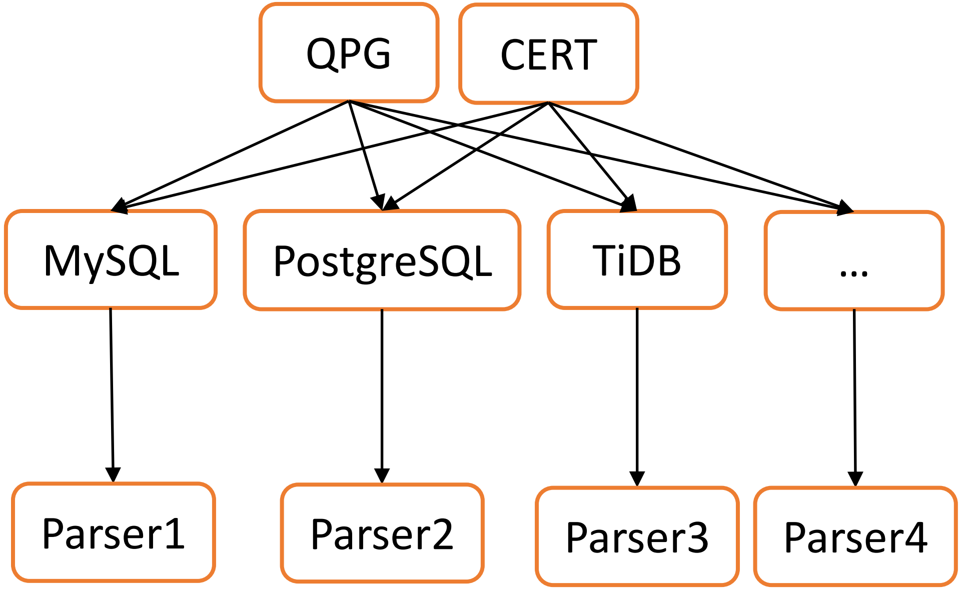
Unified Query Plan Representation (Uplan)

- We define **plan** as a **tree** that can have plan-associated **properties**.

Listing 2. The unified query plan representation in EBNF.

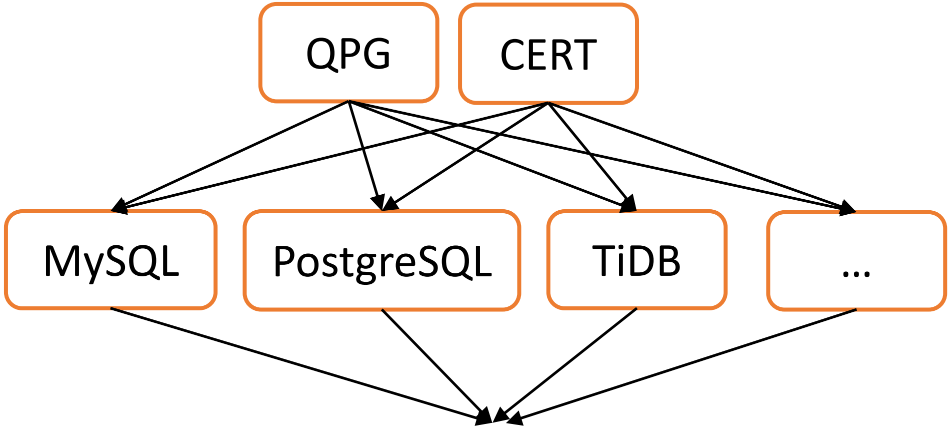
```
plan ::= ( tree )? properties
tree ::= node ( '--children-->' '{' tree (',' tree)*
        '}' )?
node ::= operation properties
operation ::= 'Operation' ':' operation_category '->'
        operation_identifier
properties ::= ( property ( ',' property )* )?
property ::= property_category '->' property_identifier
        ':' value
operation_category ::= 'Producer' | 'Bag' | 'Join' |
        'Folder' | 'Executor' | 'Projector' | 'Consumer'
property_category ::= 'Cardinality' | 'Cost' |
        'Configuration' | 'Status'
operation_identifier ::= keyword
property_identifier ::= keyword
keyword ::= letter ( letter | digit | '_' )*
value ::= string | number | boolean | 'null'
string ::= '"' ( letter | digit )* '"'
number ::= '-'? digit+
boolean ::= 'true' | 'false'
letter ::= [a-zA-Z]
digit ::= [0-9]
```

Application: Testing



- We can easily extend QPG and CERT to support more DBMSs reusing the same query plan parser.

Application: Testing




```
public static String
parseQueryPlan(String queryPlan);
```

- We can easily extend QPG and CERT to support more DBMSs reusing the same query plan parser.

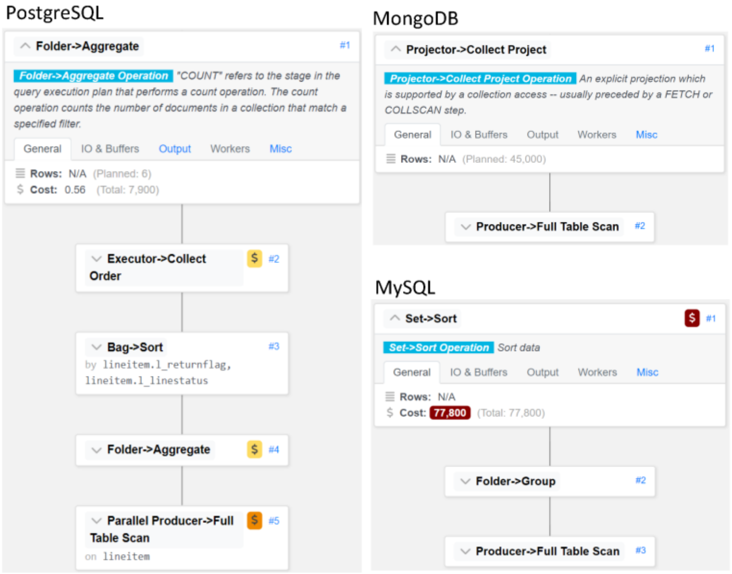
PREVIOUSLY UNKNOWN AND UNIQUE BUGS FOUND WITH *UPlan*.

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

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

Application: Visualization

- We implemented a visualization tool for serialized query plans by modifying PEV2, a customized query plan visualization tool for PostgreSQL, to use Uplan.



https://unifiedqueryplan.github.io/pev2.html

Existing DBMS-specific visualization tools could support more DBMSs if they supported our unified query plan representation.

Application: Benchmarking

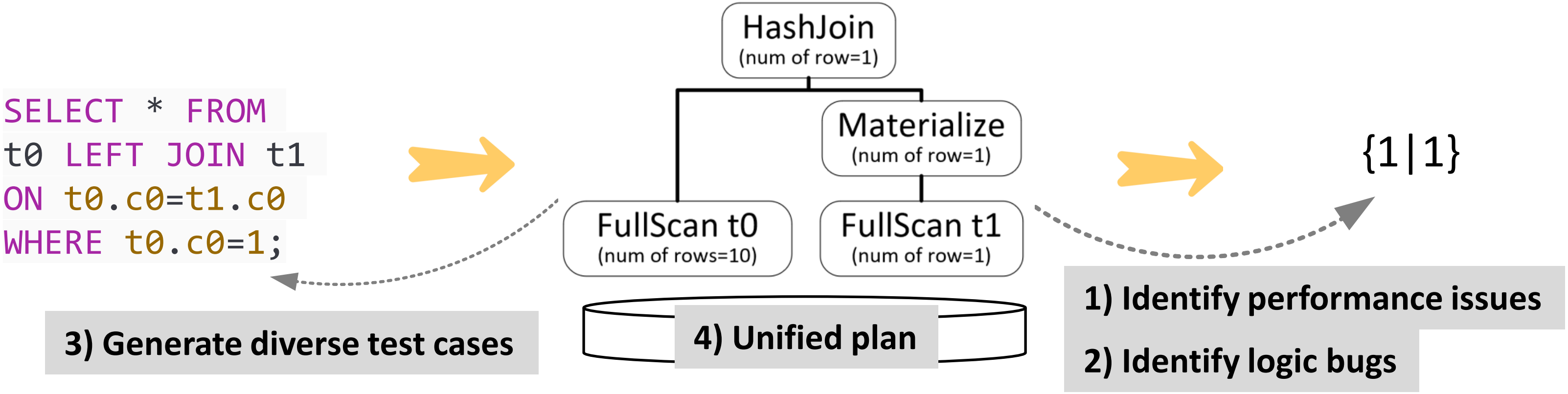
- Uplan enables comparing query plans across DBMSs.
- A potential efficiency issue that PostgreSQL requires six table scanning operations, while TiDB only requires four table scanning operations for the same query.

Comparing the unified query plan representation provides actionable insights.

```
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...
      name object: nation          name object: supplier
 Folder->Aggregate              Executor->Collect Order
  Join->Hash Join                Producer->Index-only...
   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
```

Review: Our Methods

- Challenges:
  - 1) Test oracle
  - 2) Test case generation



Impact

Pavel Klinov @klinovp · Oct 12

Cool stuff! Cardinality estimation in many systems starts well-founded at

Thuan Pham @thuanpv_ · Jun 2

Jordan Lewis @largedatabank · Dec 9, 2022

131 unique and previously unknown bugs found, and 12 CVEs assigned.

Figure 19 (d), (e) and (f) show the execution results on MySQL. Since NoREC does not support MySQL, we only compared the other tools. It can be observed that TLP and QPG are quite exceptional, as they quickly detected bugs in MySQL. They discovered 12 bugs in just 40 minutes, after which the system crashed (crash detected), leading to

automated testing towards unseen query plans for finding logic bugs. We found over 50 unique, previously unknown bugs. Stay tuned for the preprint!

This site uses Just the Docs, a documentation theme for Jekyll.

[1] Gao, X., Liu, Z., Cui, J., Li, H., Zhang, H., Wei, K., & Zhao, K. (2023). A Comprehensive Survey on Database Management System Fuzzing: Techniques, Taxonomy and Experimental Comparison. arXiv preprint arXiv:2311.06728.

Research Scope and Limitations

- Target bugs
  - Logic bugs: Incorrect results.
  - Performance issues: Unexpected slowdown.
- Query plans
  - The proposed methods require target DBMSs expose query plans.
  - Top-10 DBMSs* support exposing query plans.
- Advancing automated testing technique
  - The proposed methods can efficiently find bugs, but cannot demonstrate the absence of bugs.

*https://db-engines.com/en/ranking/relational+dbms

Discussion: Bug-finding Techniques

- The methods we covered
  - Metamorphic testing
  - Differential testing
- The methods we did not cover
  - Fuzzing
  - Test suites and benchmarking
  - Verification

Fuzzing

Squirrel[1]

```
SELECT * FROM t0;
 …
SEL?CT * FROM t0;
 …
SEL?CT * FROM t0EOFEOF;
```

Memory corrupt?

Generating Test Cases

Validating

Fuzzing can only find memory-related bugs.
We aim to efficiently find logic bugs and performance issues.

[1] Zhong, Rui, et al. "Squirrel: Testing database management systems with language validity and coverage feedback." Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. 2020.

Test Suites and Benchmarking

```
do_execsql_test join8-14020 {
  SELECT * FROM y0 RIGHT JOIN y1 ON true INNER JOIN y2 ON true WHERE y2.c!=99 AND y2.c!=98;
} {
  - 1 3
  - 1 4
  - 2 3
  - 2 4
}
```

Test case

Expected result

SQLite test suite[1]

We aim to automatically construct such test cases for finding bugs.

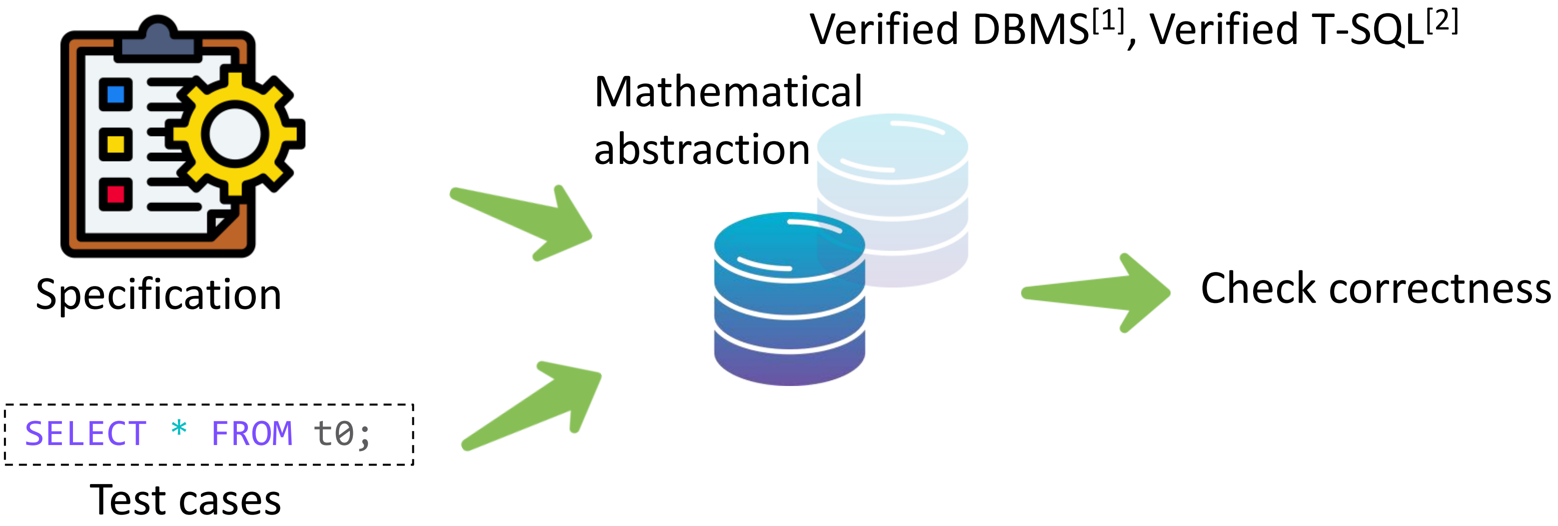
Performance benchmark TPC-H[2]

1,000 GB Results

Company	System	Performance (QphH)	Price/kQphH	Watts/KQphH	System Availability	Database	Operating System	Date Submitted
Hewlett Packard Enterprise	HPE DL325 Gen10	6,145,628	50.40 USD	NR	08/26/19	EXASOL 6.2	CentOS 7.6	07/31/19
Hewlett Packard Enterprise	HPE DL325 Gen10	3,635,443	57.91 USD	NR	08/26/19	EXASOL 6.2	CentOS 7.6	07/31/19
Hewlett Packard Enterprise	HPE ProLiant DL385 Gen11	1,156,627	265.09 USD	NR	12/05/22	Microsoft SQL Server 2022 Enterprise Edition 64 bit	Microsoft Windows Server 2022 Datacenter Edition	11/03/22
Dell Technologies	Dell PowerEdge R7515	979,335	269.23 USD	NR	05/03/21	Microsoft SQL Server 2019 Enterprise Edition 64 bit	Red Hat Enterprise Linux 8	05/03/21
Dell Technologies	PowerEdge MX740c Server	824,693	459.50 USD	NR	03/03/21	Microsoft SQL Server 2019 Enterprise Edition	Red Hat Enterprise Linux 8	03/03/21
Hewlett Packard Enterprise	HPE ProLiant DL325 Gen10	743,750	339.21 USD	NR	08/07/19	Microsoft SQL Server 2017 Enterprise Edition	Red Hat Enterprise Linux 8	08/05/19

[1] Website, "SQLite Test Suite", https://github.com/sqlite/sqlite/tree/5cc4ab93/test
[2] Website, "TPC-H", https://www.tpc.org/tpch/

Verification

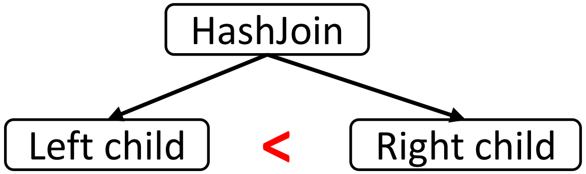


Verification can prove the target program is theoretical bug-free, but suffer from scalability problem.
We aim to efficiently find bugs in practice.

[1] Malecha, Gregory, et al. "Toward a verified relational database management system." POPL 2010.
[2] Diana, Rodrigo, et al. "A symbolic model checking appproach to verifying transact-SQL." SMC 2012.

Future Work

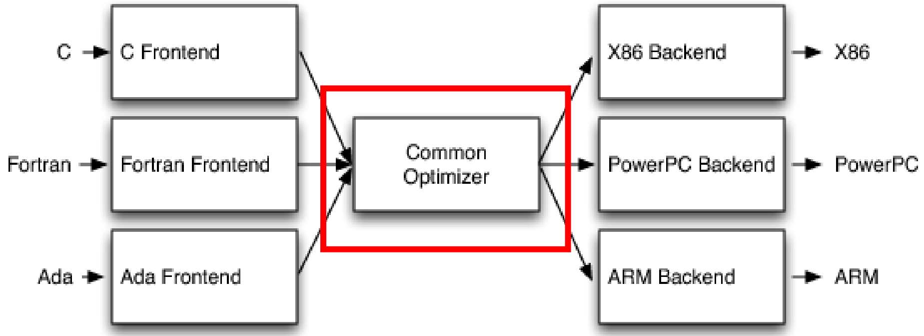
- How to simulate real-world database workload?
  - **Query plans** approximate queries and data distribution.
    - The table has an index -> **IndexScan** in the query plan
    - The table **does not** have an index -> **FullScan** in the query plan
- How to verify DBMSs in a practical way?
  - **Query plans** are suitable abstractions of DBMSs with limited states.



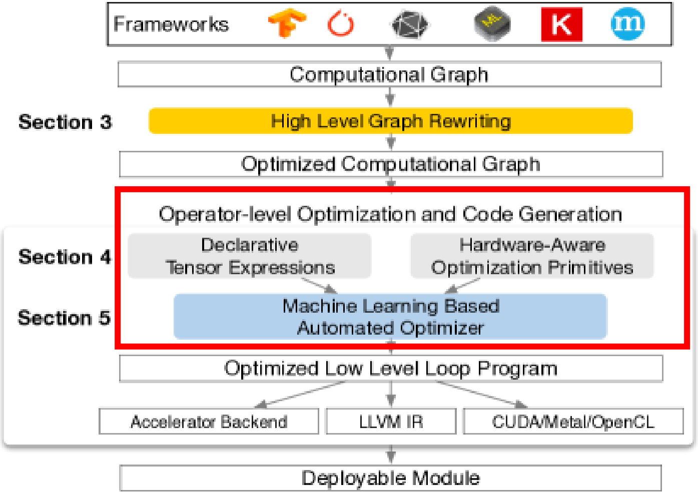
- How to detect bugs in various DBMSs?

Future Work: Beyond Query Plans

- Understand and utilize intermediate representations in testing



Compiler: LLVM



AI system: TVM

Publications During PhD Study

1) **Jinsheng Ba**, Manuel Rigger. (2024). Finding Performance Issues in Database Engines via Cardinality Estimation Testing. In Proceedings of International Conference on Software Engineering (ICSE).

2) **Jinsheng Ba**, Manuel Rigger. (2024). Keep It Simple: Testing Databases via Differential Query Plans. In Proceeding of ACM Management of Data (SIGMOD)

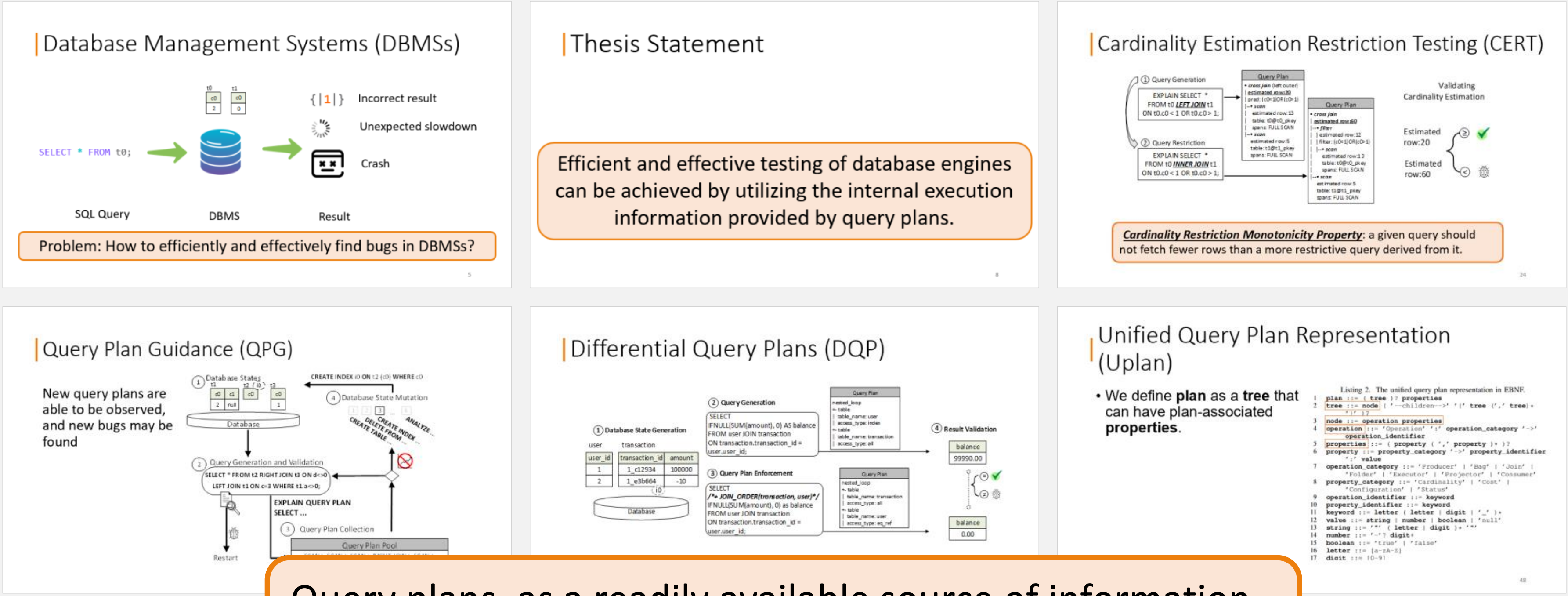
3) **Jinsheng Ba**, Manuel Rigger. (2023). Testing Database Engines via Query Plan Guidance. In Proceedings of International Conference on Software Engineering (ICSE) 🏆 **Distinguished Paper Award**

4) **Jinsheng Ba**, Manuel Rigger. (2024). Towards a Unified Query Plan Representation. (In submission).

5) **Jinsheng Ba**, Gregory J Duck, and Abhik Roychoudhury. (2022). Efficient Greybox Fuzzing to Detect Memory Errors. In The 37th IEEE/ACM International Conference on Automated Software Engineering (ASE) 🏆 **Distinguished Paper Award**

6) **Jinsheng Ba**, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. (2022). Stateful Greybox Fuzzing. In 31st USENIX Security Symposium (SEC)

Conclusion



Query plans, as a readily available source of information, can make testing more efficient and effective.