Towards Correct and Reliable Data-centric Systems

Manuel Rigger

National University of Singapore

Structure

- Wednesday
  - Challenges
  - Test oracle
  - SQLancer Overview
- Thursday
  - Test-case generation
  - Hands-on coding

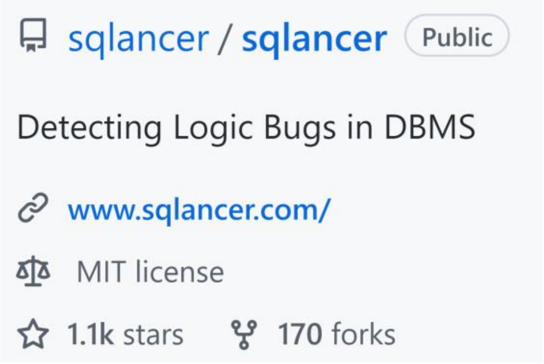
Structure

- Wednesday
  - Challenges
  - Test oracle
  - SQLancer Overview
- Thursday
  - Test-case generation
  - Hands-on coding

Establish understanding of the context, general problem, and existing solutions

Structure

- Wednesday
  - Challenges
  - Test oracle
  - SQLancer Overview
- Thursday
  - Test-case generation
  - Hands-on coding



Understand the tool to extend it (or design other tools)

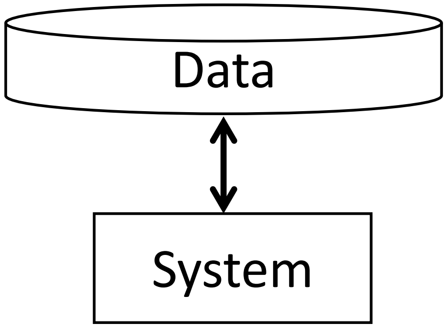
Structure

- Wednesday
  - Challenges
  - Test oracle
  - SQLancer Overview
- Thursday
  - Test-case generation
  - Hands-on coding

Open problems along the way!

What are Data-Centric Systems?

Data-centric systems: system in which **data** is an important asset



Data-Centric Systems

- Relational database systems
- Datalog engines
- Graph stores
- Document stores
- Data wrangling libraries
- Big-data processing platforms
- …

Database Management Systems (DBMS)

Structured Query Language (SQL)

Interact with

Database Management System

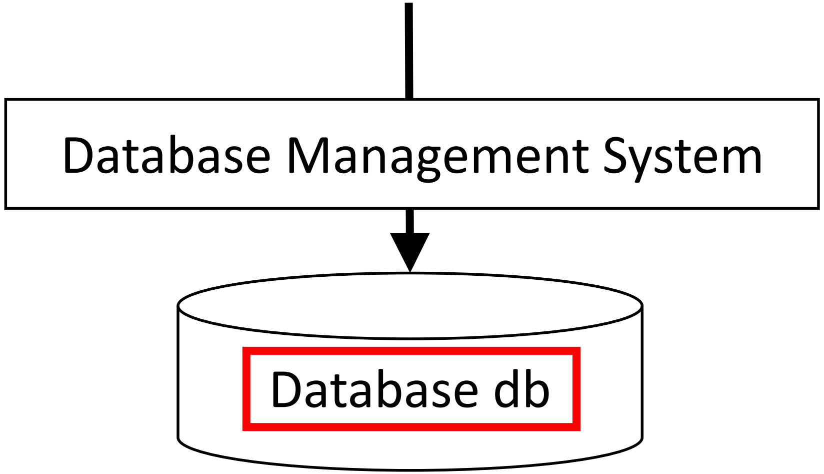
Database Management Systems (DBMS)

```sql
CREATE DATABASE db;
```

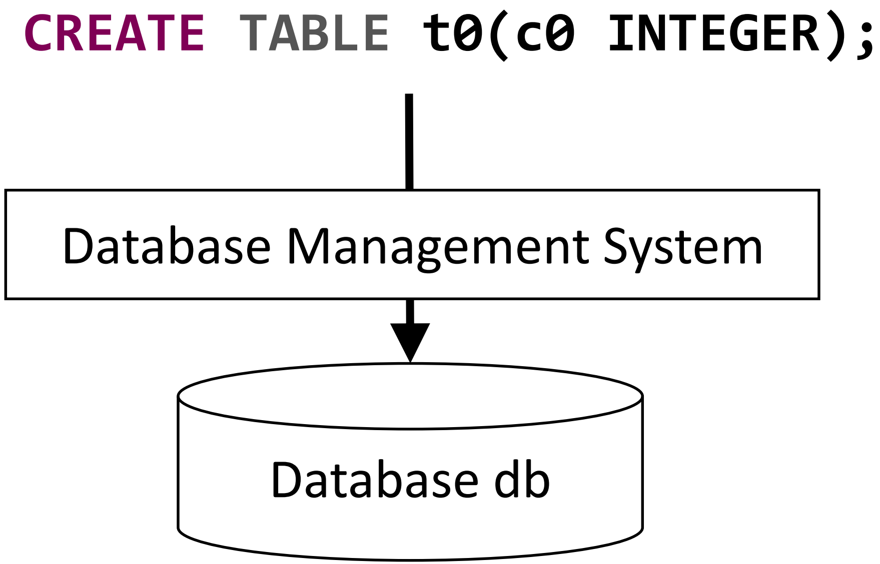
Database Management System

Database Management Systems (DBMS)

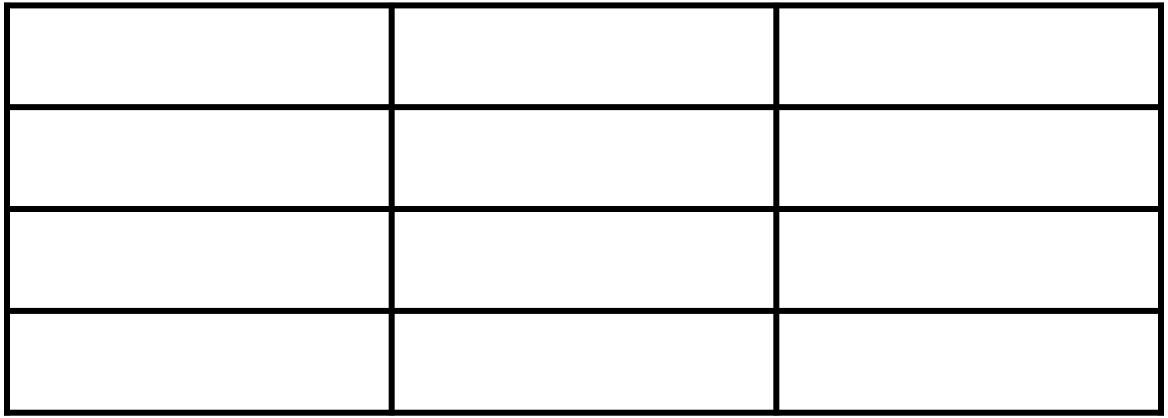
```sql
CREATE DATABASE db;
```



Database Management Systems (DBMS)



Relational Model

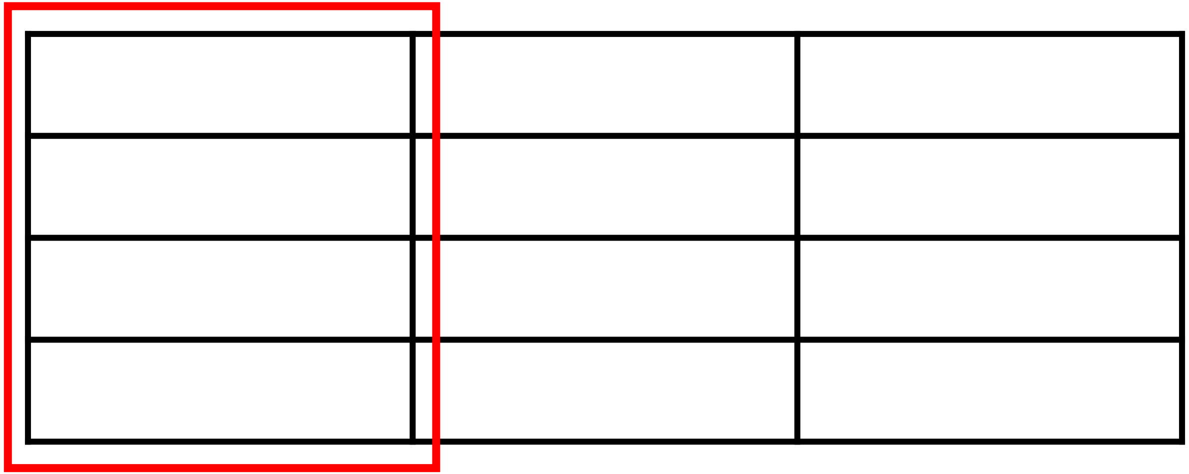


Relational Model

Table or Relation

Relational Model

Column or Attribute

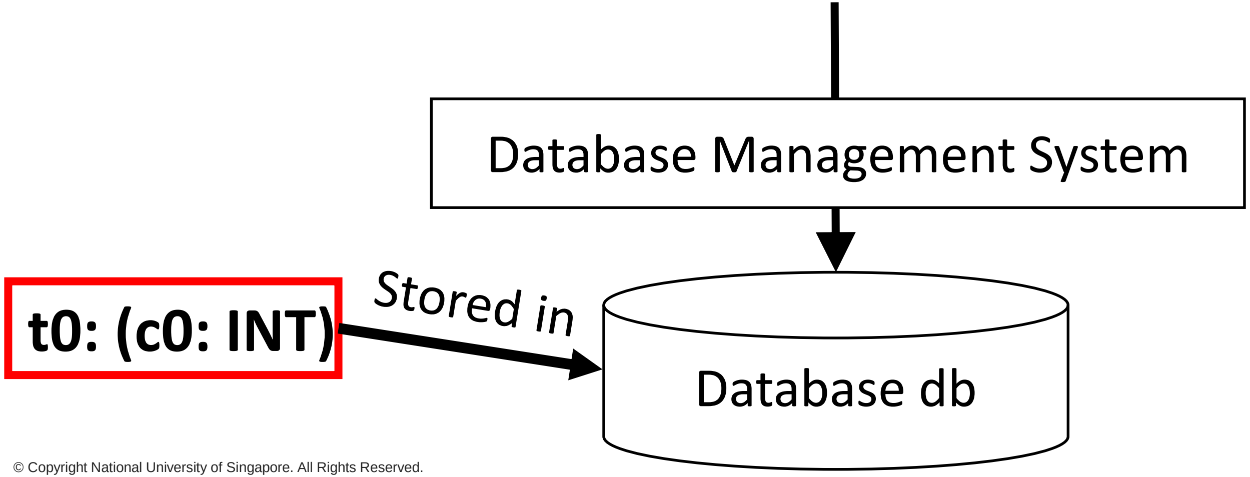


Relational Model

Row or Tuple

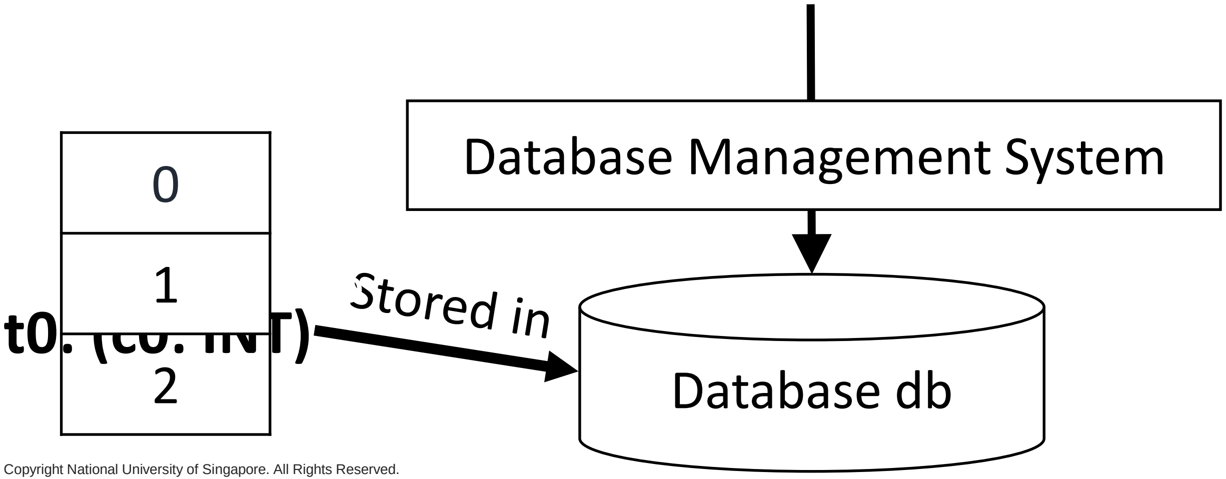
Database Management Systems (DBMS)

```sql
CREATE TABLE t0(c0 INTEGER);
```

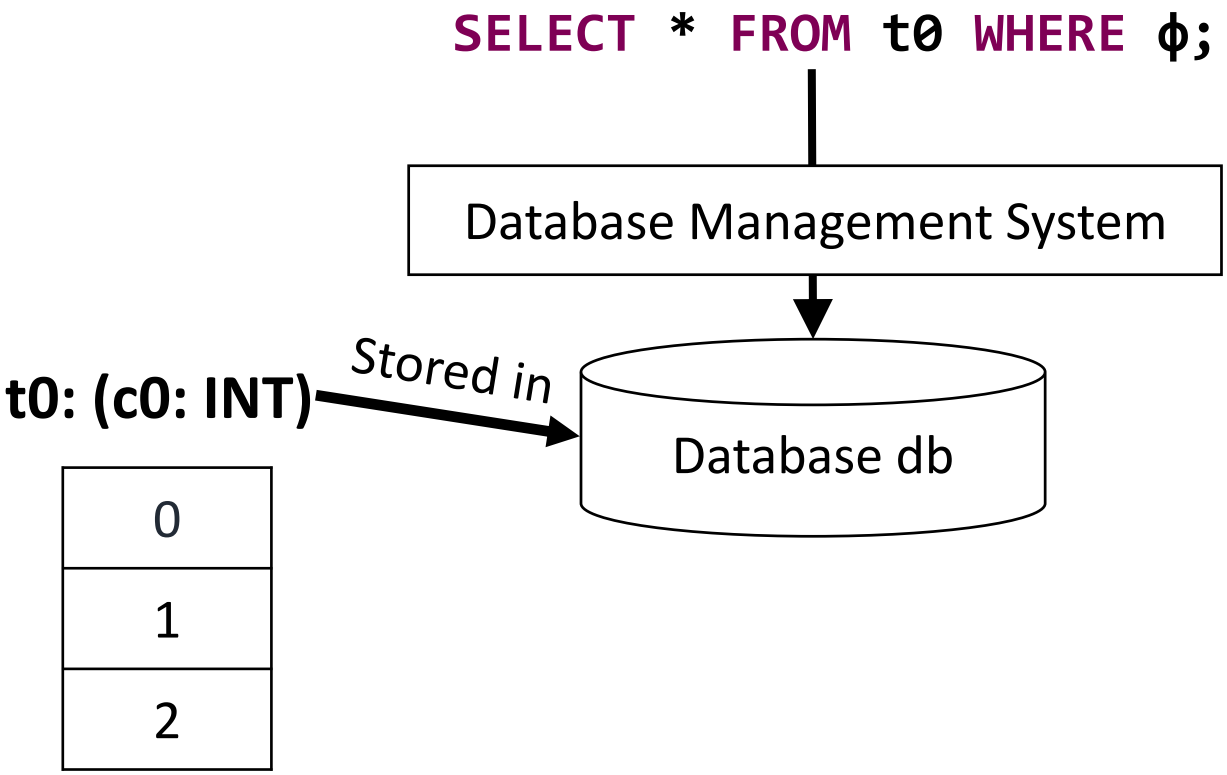


Database Management Systems (DBMS)

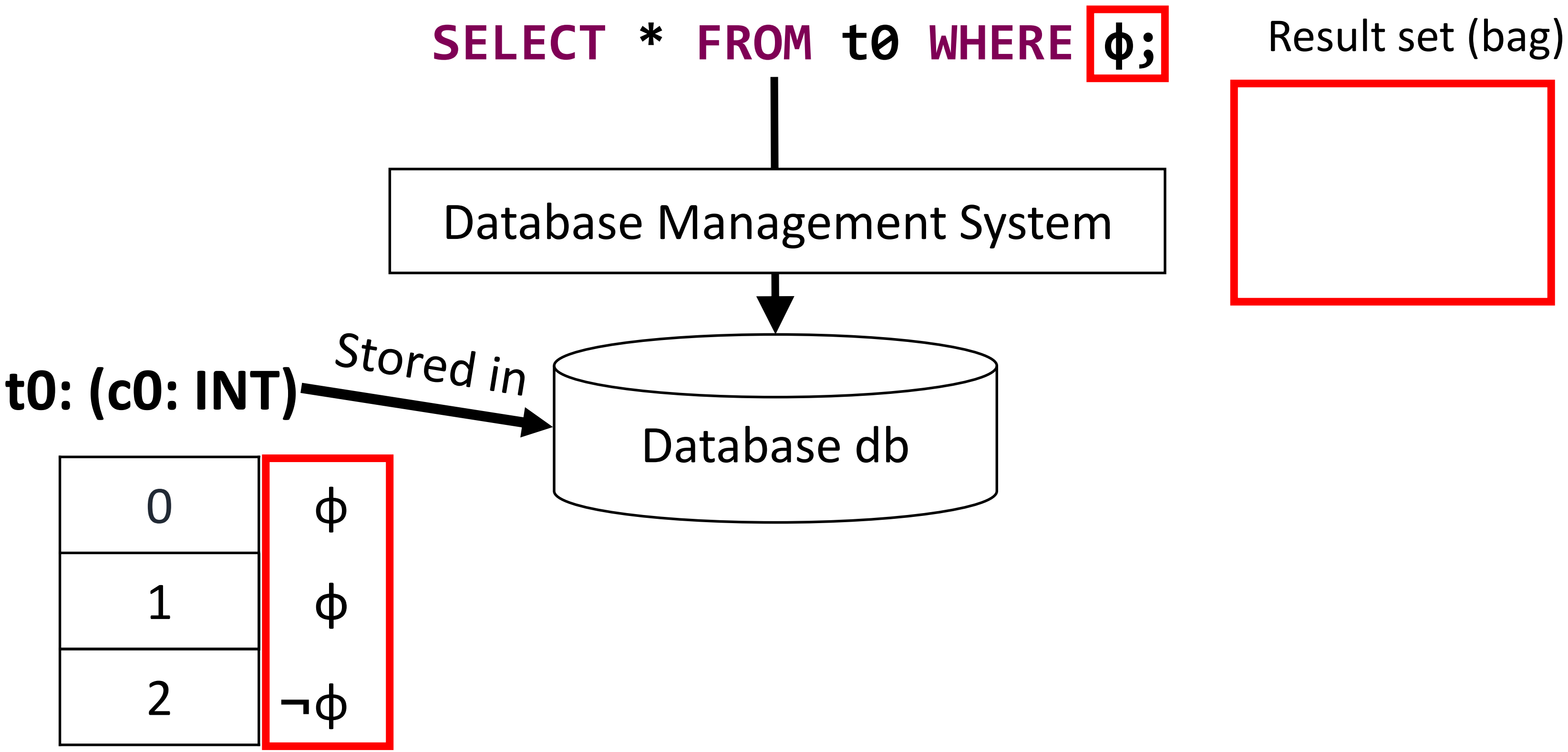
```sql
INSERT INTO t0(c0) VALUES (0), (1), (2);
```



Database Management Systems (DBMS)



Database Management Systems (DBMS)



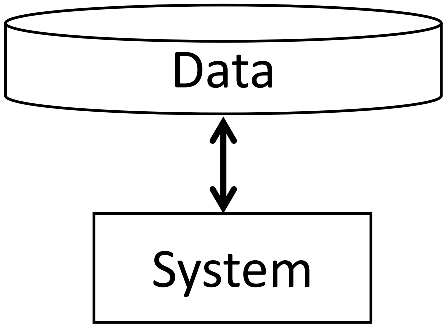
Importance of Data-centric Systems

M Motivate in a couple of bullet points why data-centric systems are important

- Data-driven decision-making: Data-centric systems provide organizations with the necessary tools to gather, analyze, and interpret data, enabling informed and data-driven decision-making.
- Operational efficiency and optimization: By effectively managing and utilizing data, data-centric systems help organizations streamline processes, improve efficiency, and identify areas for optimization, leading to cost savings and improved performance.
- Insights and innovation: Data-centric systems unlock valuable insights, patterns, and trends hidden within vast amounts of data, empowering organizations to drive innovation, identify new opportunities, and stay ahead of the competition.
- Customer-centric approach: By leveraging data-centric systems, organizations can gain a deeper understanding of their customers, personalize experiences, and tailor products and services to meet customer needs and preferences.
- Compliance and risk management: Data-centric systems enable organizations to effectively manage and mitigate risks, ensuring compliance with regulatory requirements and safeguarding sensitive information.
- Scalability and adaptability: Data-centric systems can scale to handle large volumes of data, adapt to changing business needs, and support future growth, providing a flexible and resilient foundation for organizational success.

Training ChatGPT likely involved data-centric systems for collecting, cleaning, storing, and processing data

What makes testing them challenging?



Challenges

- No ground truth
- Heterogeneous landscape
- Inputs are complex (creating databases + queries)
- Traditional coverage metrics do not capture the state

Challenges

- $\boxed{\text{No ground truth}}$
- Heterogeneous landscape
- Inputs are complex (creating databases + queries)
- Traditional coverage metrics do not capture the state

No Ground Truth

t0

c0
0.0

t1

c0
-0.0

MySQL

```
SELECT * FROM t0, t1
WHERE t0.c0 = t1.c0;
```

→ ?

It might seem **disputable** whether the predicate should evaluate to `true`

No Ground Truth

t0

c0
0.0

t1

c0
-0.0

0	0	0	0	0	...	0	0	0	0
-0	1	0	0	0	...	0	0	0	0

Different binary representation

false?

```
SELECT * FROM t0, t1
WHERE t0.c0 = t1.c0;
```

t0.c0	t1.c0

No Ground Truth

t0

c0
0.0

t1

c0
-0.0

true?

Evaluates to `true` for **most programming languages**

```
SELECT * FROM t0, t1
WHERE t0.c0 = t1.c0;
```

MySQL

t0.c0	t1.c0
0	-0

No Ground Truth

t0

c0
0.0

t1

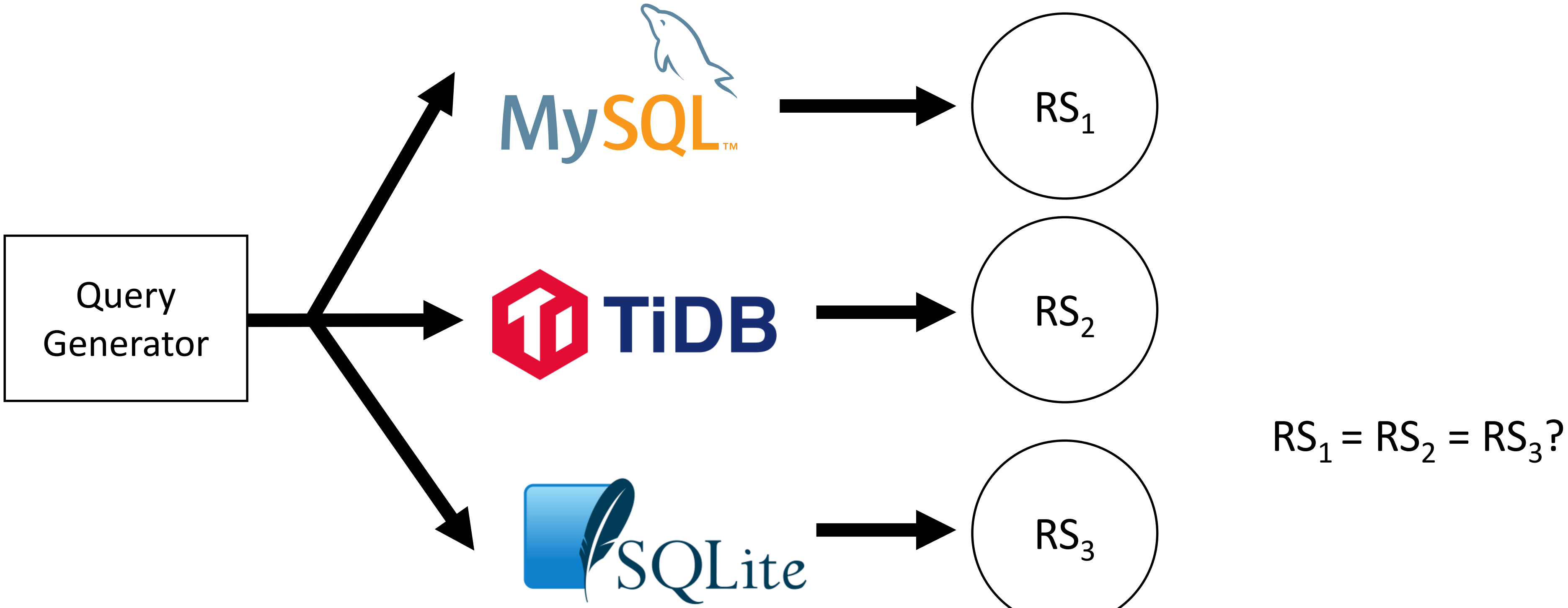
c0
-0.0

```
SELECT * FROM t0, t1
WHERE t0.c0 = t1.c0;
```

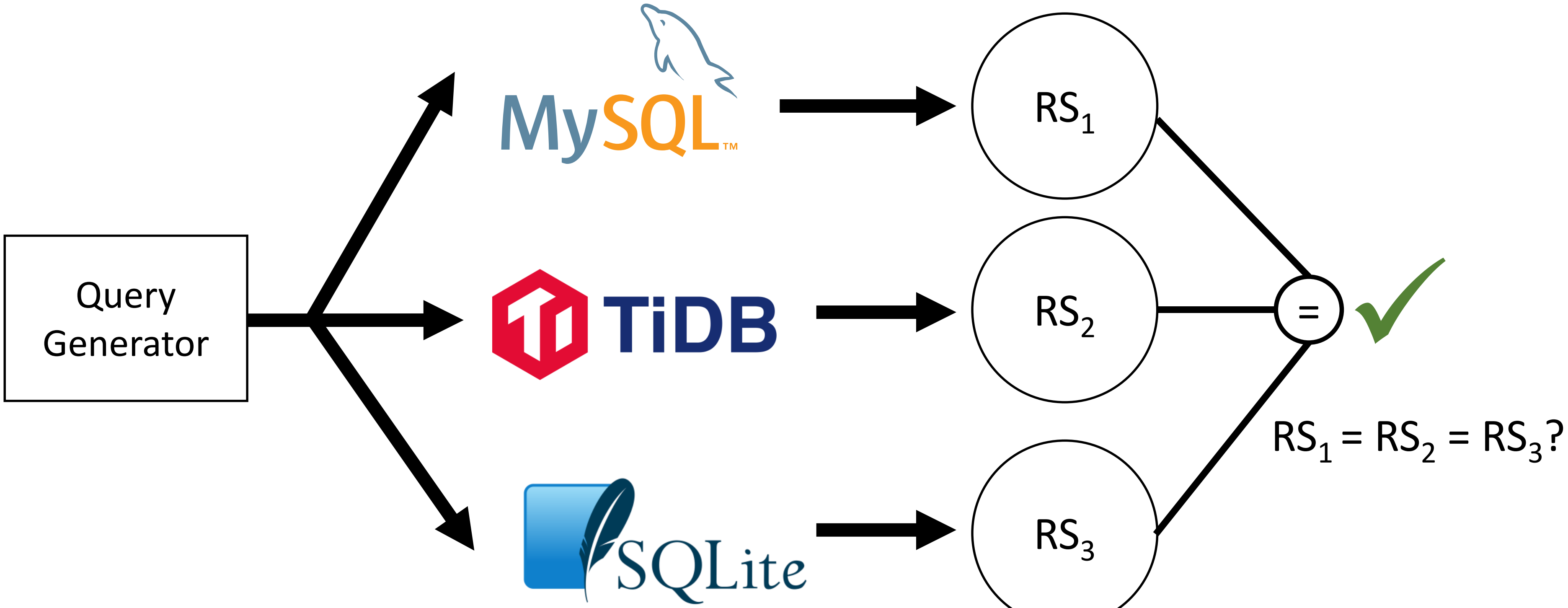
MySQL

t0.c0	t1.c0
0	-0

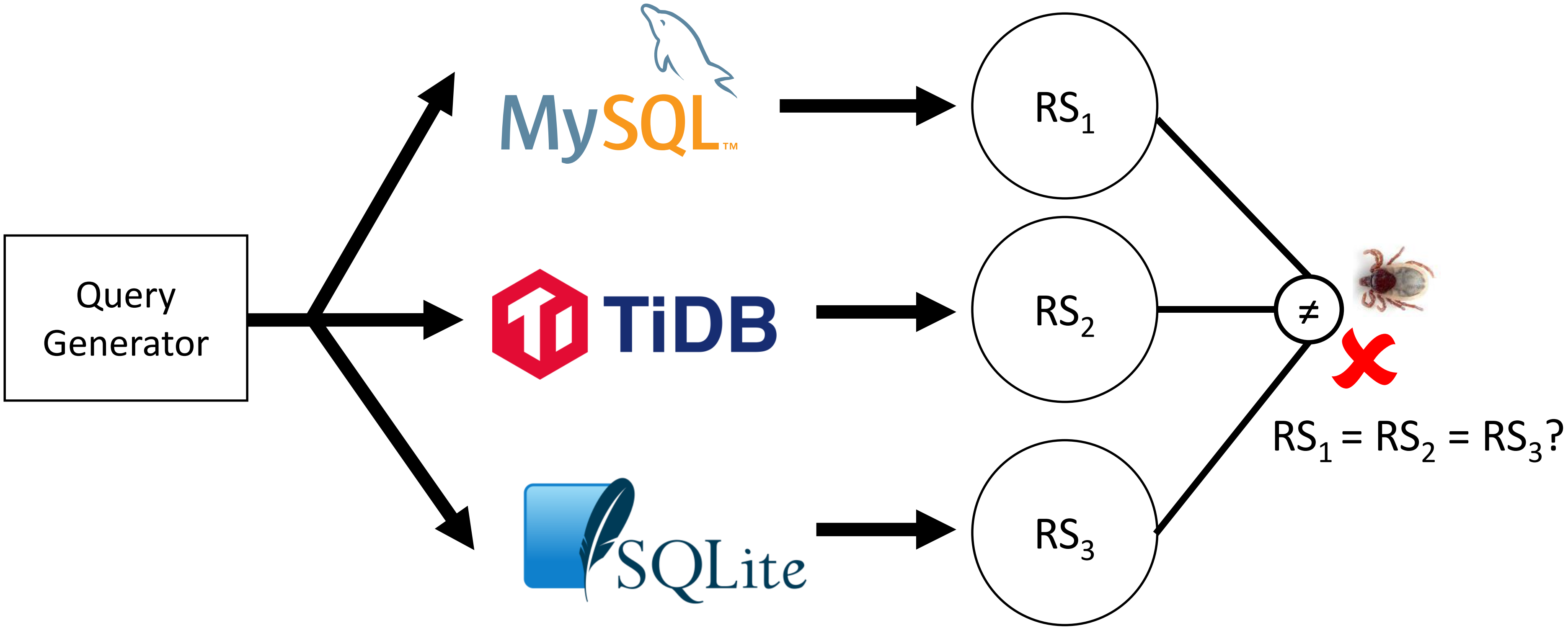
Differential Testing?



Differential Testing?



Differential Testing?



Slutz, VLDB 1998

Massive Stochastic Testing of SQL

Don Slutz
Microsoft Research
dslutz@Microsoft.com

Abstract

Deterministic testing of SQL database systems is human intensive and cannot adequately cover the SQL input domain. A system (RAGS), was built to stochastically generate valid SQL statements 1 million times faster than a human and execute them.

1 Testing SQL is Hard

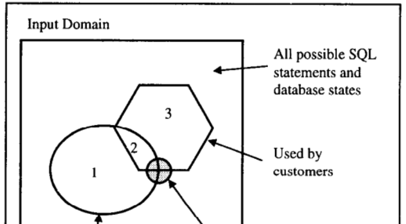
Good test coverage for commercial SQL database systems is very hard. The *input domain*, all SQL statements, from any number of users, combined with all states of the database, is gigantic. It is also difficult to verify output for positive tests because the semantics of SQL are complicated.

Software engineering technology exists to predictably improve quality ([Bei90] for example). The techniques involve a software development process including unit tests and final system validation tests (to verify the absence of bugs). This process requires a substantial investment so commercial SQL vendors with tight schedules tend to use a more ad hoc proc-

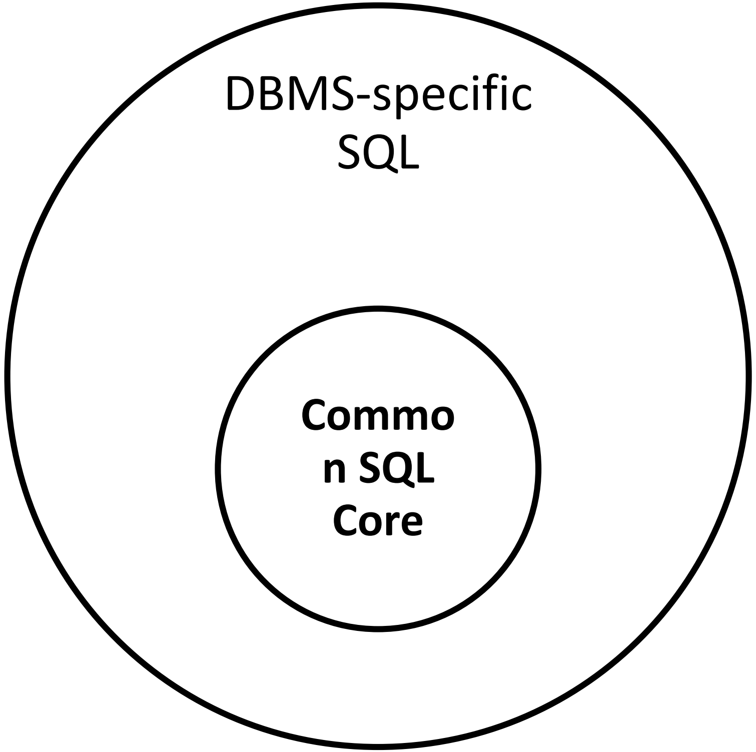
distribute the SQL statements in useful regions of the input domain. If the distribution is adequate, stochastic testing has the advantage that the quality of the tests improves as the test size increases [TFW93].

A system called RAGS (Random Generation of SQL) was built to explore automated testing. RAGS is currently used by the Microsoft SQL Server [MSS98] testing group. This paper describes RAGS and some illustrative test results.

Figure 1 illustrates the test coverage problem. Customers use the hexagon, bugs are in the oval, and the test libraries cover the shaded circle.



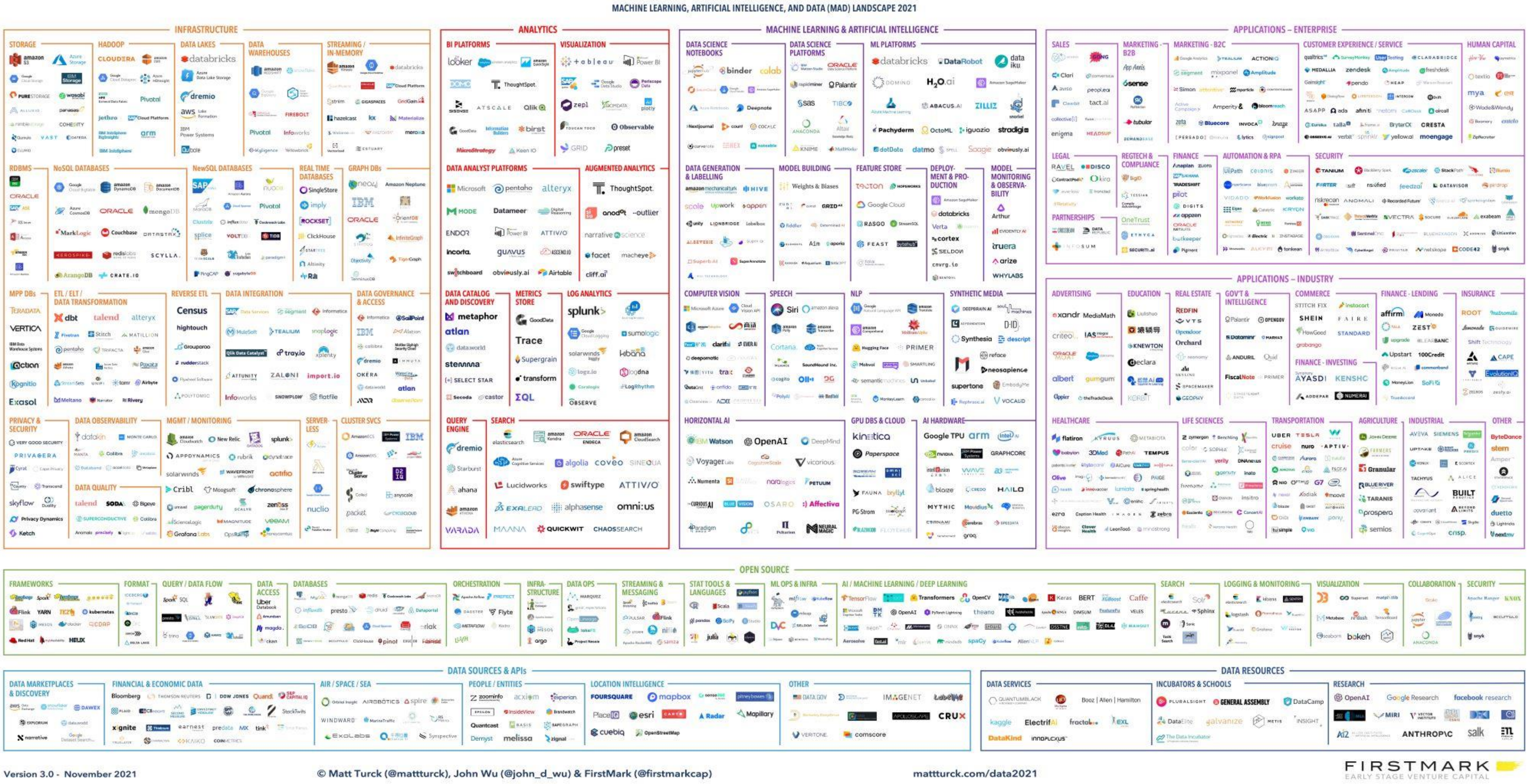
Slutz, VLDB 1998



“[…] proved to be extremely useful, but only for the **small set of common SQL**”

Challenges

- No ground truth
- Heterogeneous landscape
- Inputs are complex (creating databases + queries)
- Traditional coverage metrics do not capture the state


MACHINE LEARNING, ARTIFICIAL INTELLIGENCE, AND DATA (MAD) LANDSCAPE 2021
INFRASTRUCTURE
STORAGE
HADOOP
DATA LAKES
DATA WAREHOUSES
STREAMING / IN-MEMORY
RDBMS
NoSQL DATABASES
NewSQL DATABASES
REAL TIME DATABASES
GRAPH DBs
MPP DBs
ETL / ELT / DATA TRANSFORMATION
REVERSE ETL
DATA INTEGRATION
DATA GOVERNANCE & ACCESS
PRIVACY & SECURITY
DATA OBSERVABILITY
DATA QUALITY
MGMT / MONITORING
SERVER-LESS
CLUSTER SVCS
ANALYTICS
BI PLATFORMS
VISUALIZATION
DATA ANALYST PLATFORMS
AUGMENTED ANALYTICS
DATA CATALOG AND DISCOVERY
METRICS STORE
LOG ANALYTICS
QUERY ENGINE
SEARCH
MACHINE LEARNING & ARTIFICIAL INTELLIGENCE
DATA SCIENCE NOTEBOOKS
DATA SCIENCE PLATFORMS
ML PLATFORMS
DATA GENERATION & LABELLING
MODEL BUILDING
FEATURE STORE
DEPLOYMENT & PRODUCTION
MODEL MONITORING & OBSERVABILITY
COMPUTER VISION
SPEECH
NLP
SYNTHETIC MEDIA
HORIZONTAL AI
GPU DBS & CLOUD
AI HARDWARE
APPLICATIONS – ENTERPRISE
SALES
MARKETING - B2B
MARKETING - B2C
CUSTOMER EXPERIENCE / SERVICE
HUMAN CAPITAL
LEGAL
REGTECH & COMPLIANCE
FINANCE
AUTOMATION & RPA
SECURITY
PARTNERSHIPS
APPLICATIONS – INDUSTRY
ADVERTISING
EDUCATION
REAL ESTATE
GOV'T & INTELLIGENCE
COMMERCE
FINANCE - LENDING
INSURANCE
FINANCE - INVESTING
HEALTHCARE
LIFE SCIENCES
TRANSPORTATION
AGRICULTURE
INDUSTRIAL
OTHER
OPEN SOURCE
FRAMEWORKS
FORMAT
QUERY / DATA FLOW
DATA ACCESS
DATABASES
ORCHESTRATION
INFRASTRUCTURE
DATA OPS
STREAMING & MESSAGING
STAT TOOLS & LANGUAGES
ML OPS & INFRA
AI / MACHINE LEARNING / DEEP LEARNING
SEARCH
LOGGING & MONITORING
VISUALIZATION
COLLABORATION
SECURITY
DATA SOURCES & APIs
DATA MARKETPLACES & DISCOVERY
FINANCIAL & ECONOMIC DATA
AIR / SPACE / SEA
PEOPLE / ENTITIES
LOCATION INTELLIGENCE
OTHER
DATA RESOURCES
DATA SERVICES
INCUBATORS & SCHOOLS
RESEARCH
Version 3.0 - November 2021
© Matt Turck (@mattturck), John Wu (@john_d_wu) & FirstMark (@firstmarkcap)
mattturck.com/data2021
FIRSTMARK
EARLY STAGE VENTURE CAPITAL

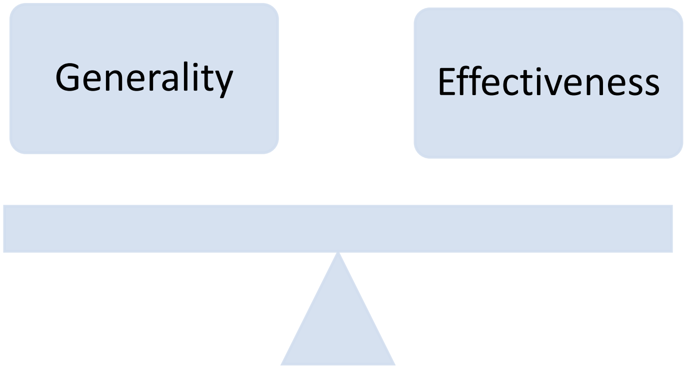
The “Data Processing Kaleidoscope”

Heterogeneous landscape of approaches, systems, and services

The “Data Processing Kaleidoscope”

Tradeoff: Generality and Effectiveness

Inherent conflict between creating **general** and **effective** techniques



Challenges

- No ground truth
- Heterogeneous landscape
- Inputs are complex (creating databases + queries)
- Traditional coverage metrics do not capture the state

Complex Inputs

```
CREATE TABLE t0(c0 INT UNIQUE);
CREATE TABLE t1(c0 INT, c1 TEXT);
INSERT INTO t1(c0, c1) VALUES (0, '');
UPDATE t1 SET c0 = 3;
CREATE INDEX i0 ON t1(c0, c1) WHERE c0 > 0;
INSERT INTO t0(c0) VALUES (0), (0);
SELECT * FROM t0, t1, t2, t3;
```

Validity of statements depends on the objects that exist in the database

Complex Inputs

```sql
CREATE TABLE t0(c0 INT UNIQUE);
CREATE TABLE t1(c0 INT, c1 TEXT);
INSERT INTO t1(c0, c1) VALUES (0, '');
UPDATE t1 SET c0 = 3;
CREATE INDEX i0 ON t1(c0, c1) WHERE c0 > 0;
INSERT INTO t0(c0) VALUES (0), (0);
SELECT * FROM t0, t1, t2, t3;
```

Statements can fail or be redundant

Complex Inputs

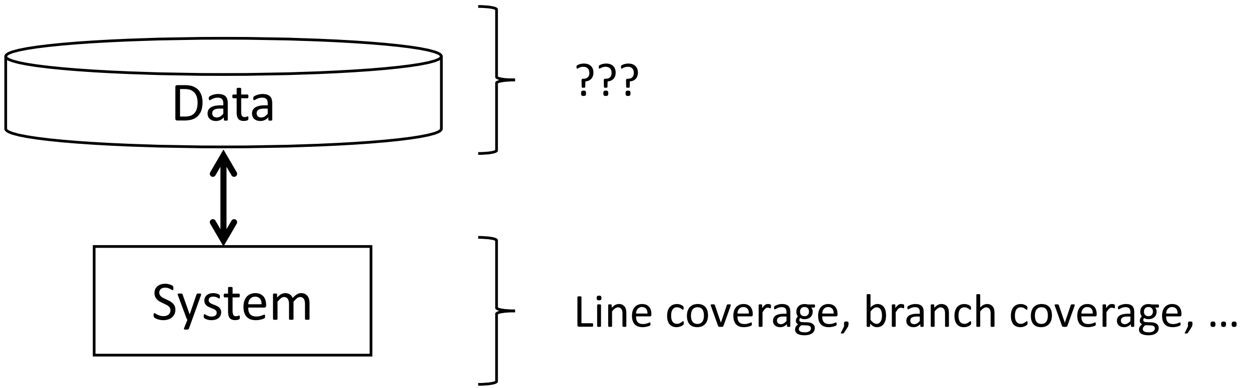
```
CREATE TABLE t0(c0 INT UNIQUE);
CREATE TABLE t1(c0 INT, c1 TEXT);
INSERT INTO t1(c0, c1) VALUES (0, '');
UPDATE t1 SET c0 = 3;
CREATE INDEX i0 ON t1(c0, c1) WHERE c0 > 0;
INSERT INTO t0(c0) VALUES (0), (0);
SELECT * FROM t0, t1, t2, t3;
```

Queries can "explode" in terms of their result size

Challenges

- No ground truth
- Heterogeneous landscape
- Inputs are complex (creating databases + queries)
- $\boxed{\text{Traditional coverage metrics do not capture the state}}$

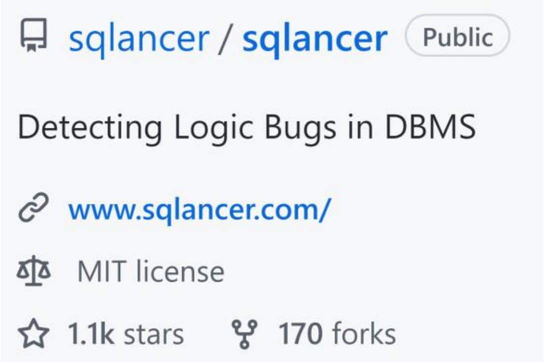
Coverage



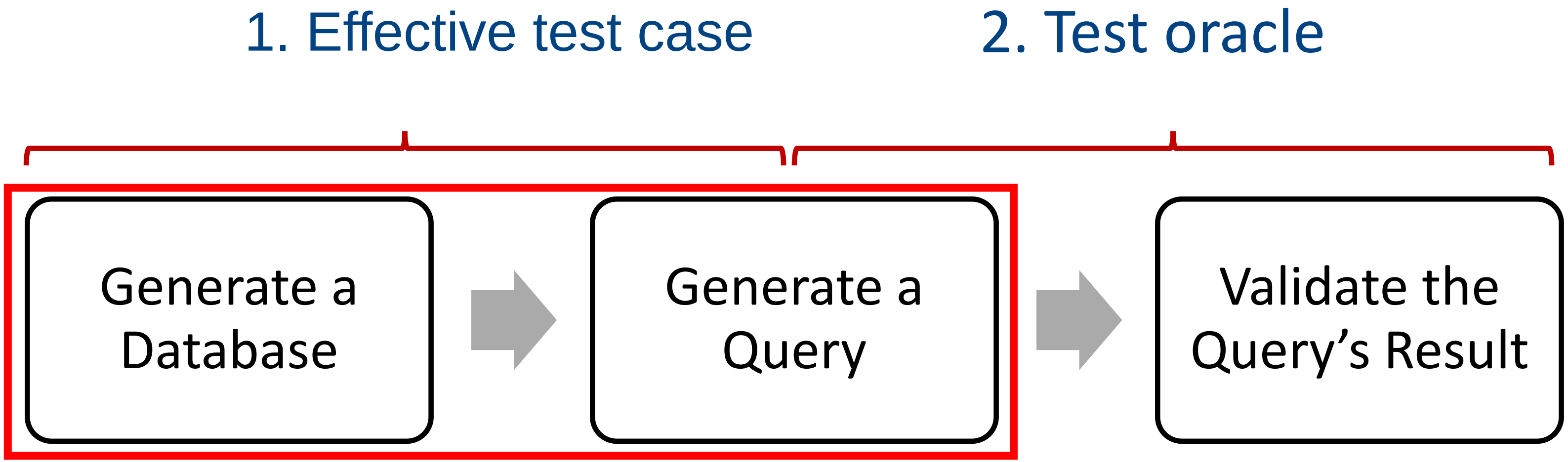
SQLancer

- Automated testing tool for finding bugs in database systems
- Hundreds of bugs found in mature, widely used DBMSs

SQLANCER



Automatic Testing Core Challenges



Query Plan Guidance (ICSE 2023)

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

http://jinshengba.me/

Automatic Testing Core Challenges

1. Effective test case

2. Test oracle

Generate a Database → Generate a Query → Validate the Query's Result

Ternary Logic Partitioning (OOPSLA 2020)

Finding Bugs in Database Systems via Query Partitioning

MANUEL RIGGER, ETH Zurich, Switzerland
ZHENDONG SU, ETH Zurich, Switzerland

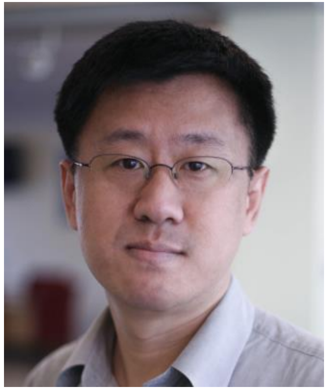
Logic bugs in Database Management Systems (DBMSs) are bugs that cause an incorrect result for a given query, for example, by omitting a row that should be fetched. These bugs are critical, since they are likely to go unnoticed by users. We propose *Query Partitioning*, a general and effective approach for finding logic bugs in DBMSs. The core idea of Query Partitioning is to, starting from a given original query, derive multiple, more complex queries (called *partitioning queries*), each of which computes a *partition* of the result. The individual partitions are then composed to compute a result set that must be equivalent to the original query's result set. A bug in the DBMS is detected when these result sets differ. Our intuition is that due to the increased complexity, the partitioning queries are more likely to stress the DBMS and trigger a logic bug than the original query. As a concrete instance of a partitioning strategy, we propose Ternary Logic Partitioning (TLP), which is based on the observation that a boolean predicate `p` can either evaluate to `TRUE`, `FALSE`, or `NULL`. Accordingly, a query can be decomposed into three partitioning queries, each of which computes its result on rows or intermediate results for which `p`, `NOT p`, and `p IS NULL` hold. This technique is versatile, and can be used to test `WHERE`, `GROUP BY`, as well as `HAVING` clauses, aggregate functions, and `DISTINCT` queries. As part of an extensive testing campaign, we found 175 bugs in widely-used DBMSs such as MySQL, TiDB, SQLite, and CockroachDB, 125 of which have been fixed. Notably, 77 of these were logic bugs, while the remaining were error and crash bugs. We expect that the effectiveness and wide applicability of Query Partitioning will lead to its broad adoption in practice, and the formulation of additional partitioning strategies.

211

CCS Concepts: • **Information systems → Database query processing**; • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: database testing, DBMS testing, test oracle, three-valued logic

ACM Reference Format:
Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 211 (November 2020), 30 pages. https://doi.org/10.1145/3428279



Hands-on Part

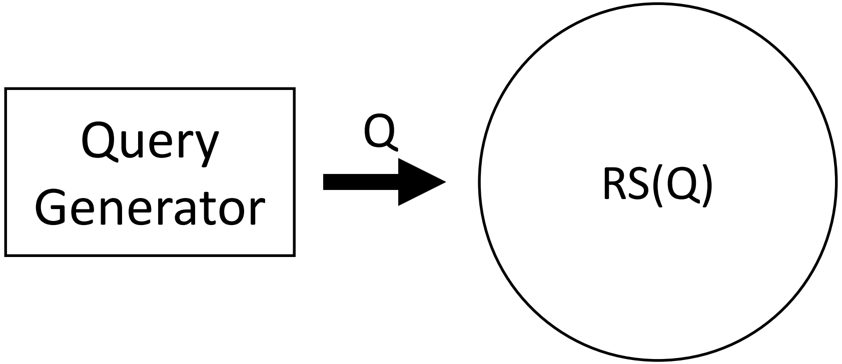
- SQLancer hands-on
  - New test oracle
  - New test case generation approach

Test Oracle
Ternary Logic Partitioning (TLP)

Query Partitioning

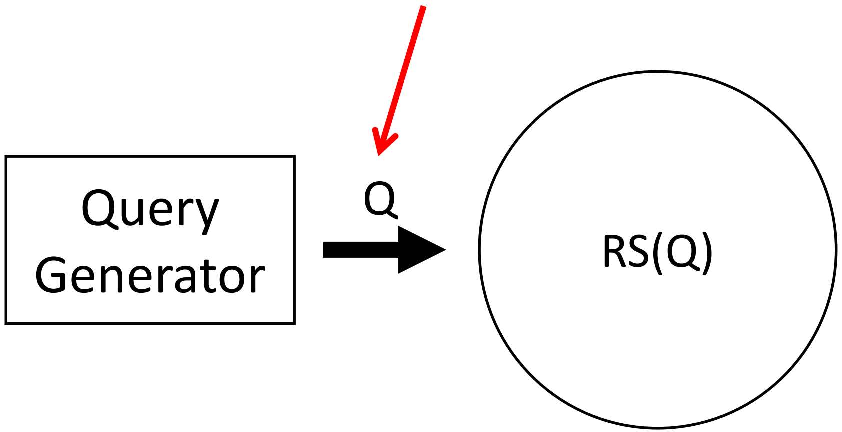
Ternary Logic Partitioning (TLP) is based on a conceptual framework called **Query Partitioning**

Query Partitioning

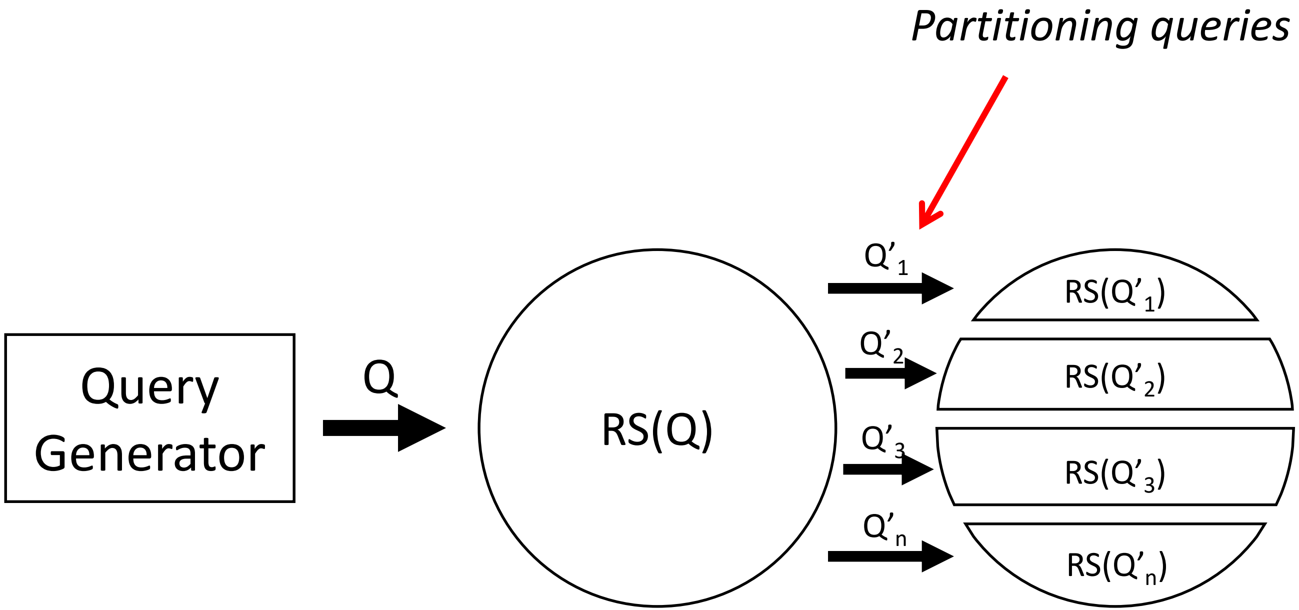


Query Partitioning

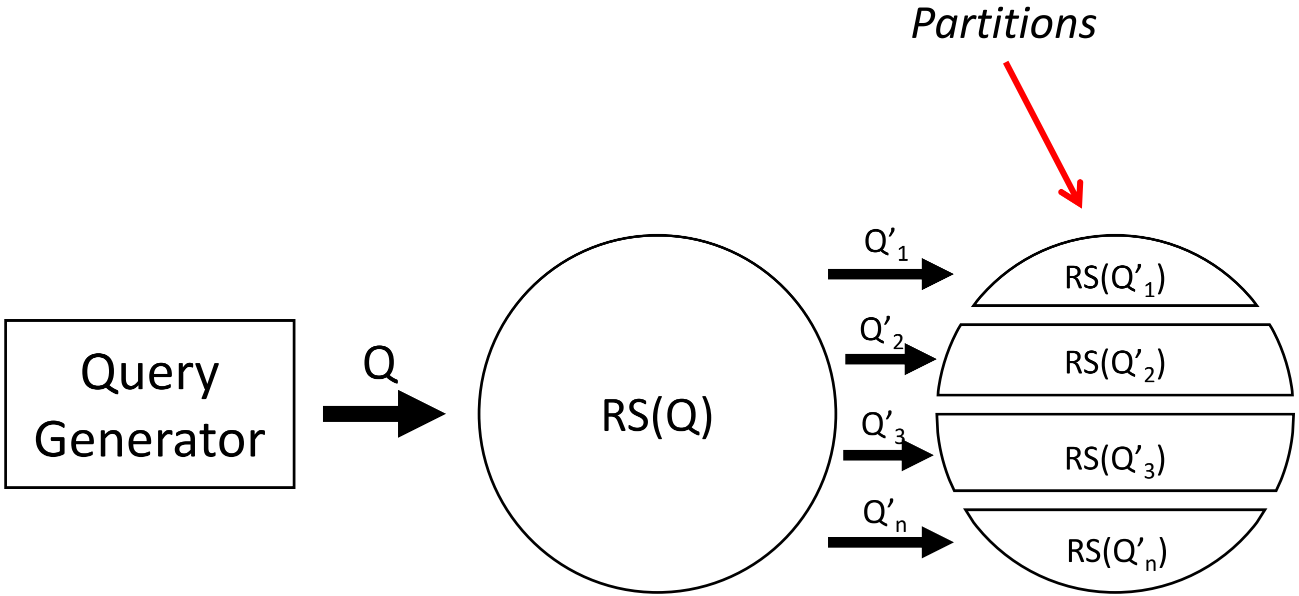
Q denotes the (randomly-generated) *original query*



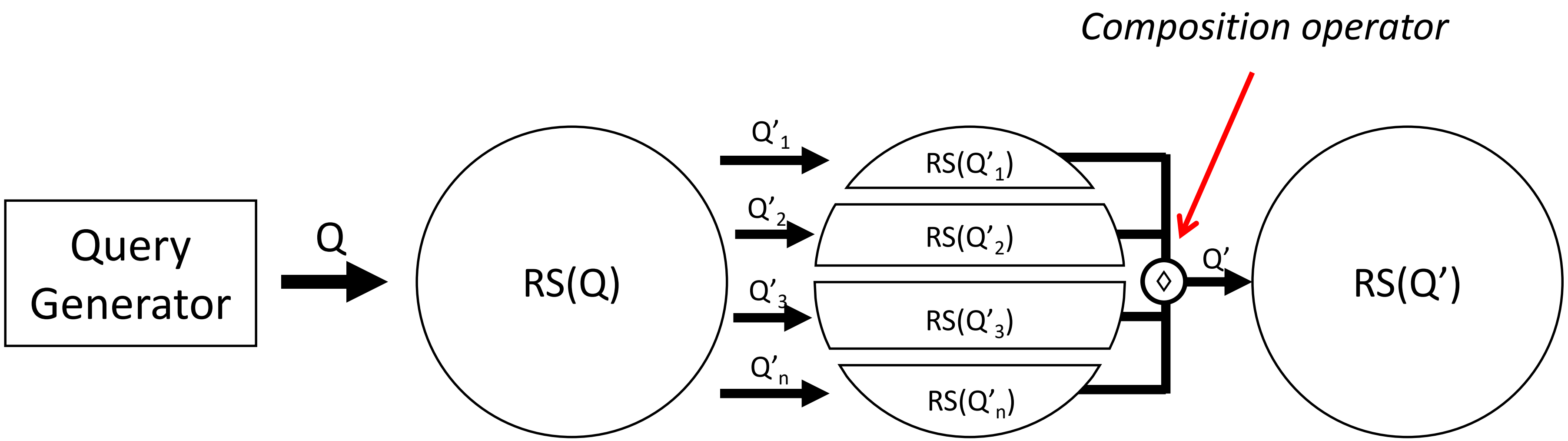
Query Partitioning



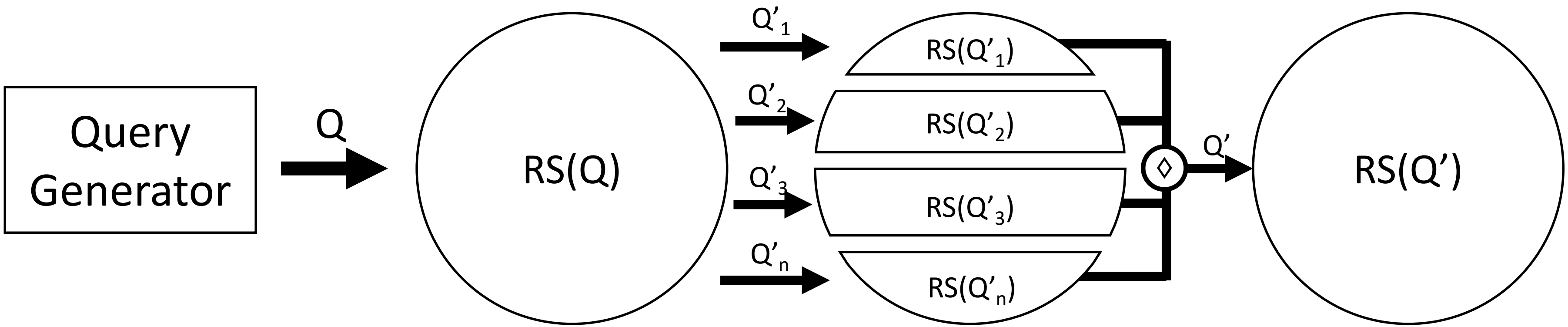
Query Partitioning



Query Partitioning

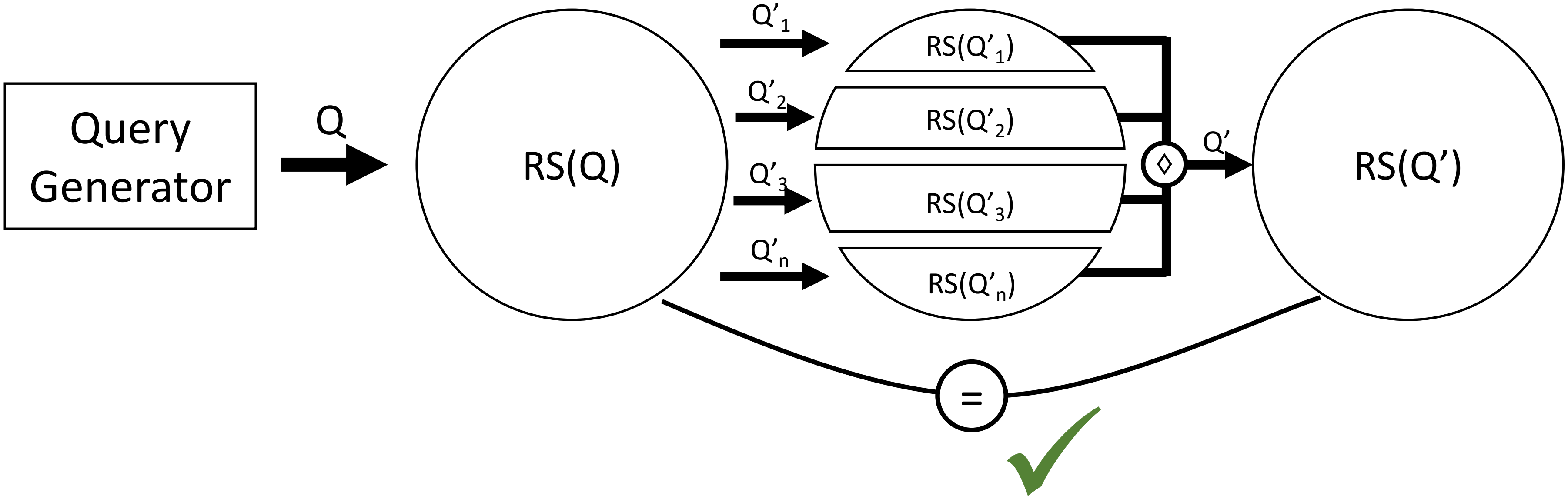


Query Partitioning

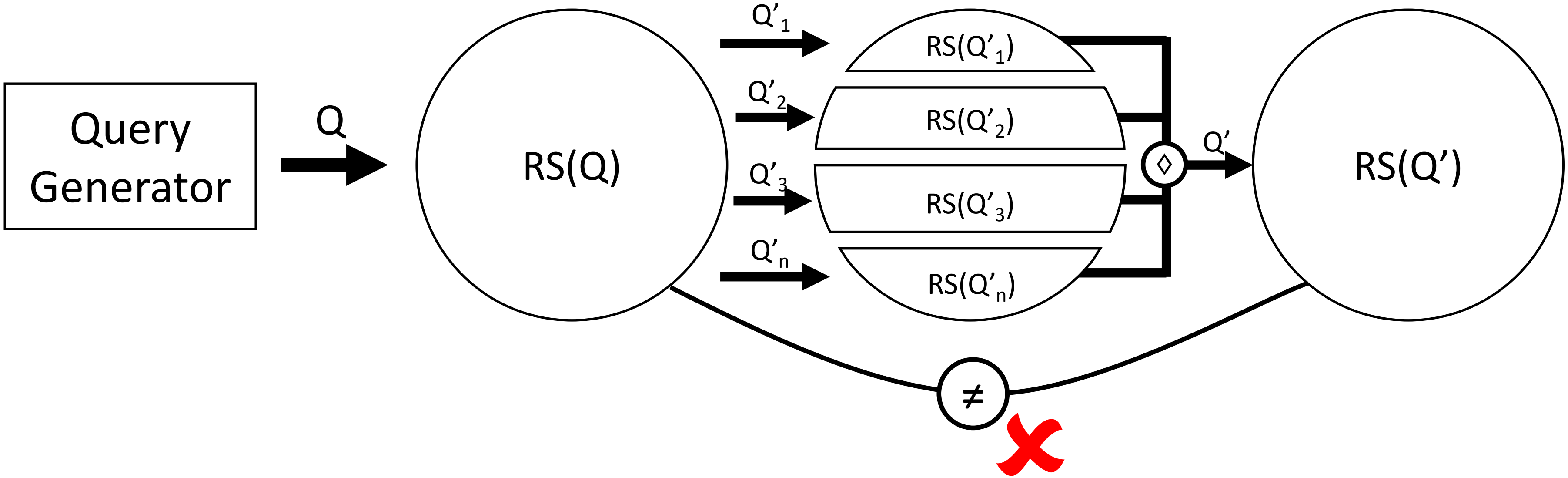


Combine the results so that RS(Q)=RS(Q')

Query Partitioning



Query Partitioning



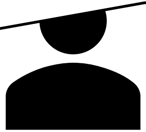
How to Realize This Idea?

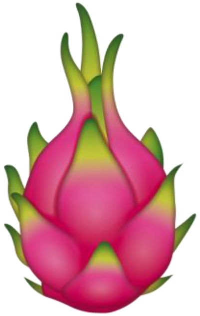
Key challenge: find a **valid partitioning strategy** that **stresses the DBMS**

Scenario: Dragon Fruits

White vs. Red Dragon Fruits

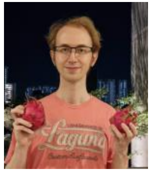
Red dragon fruits are tastier than white ones, but it is impossible to tell them apart

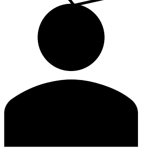




White vs. Red Dragon Fruits

Easy Lah! Just browse the web





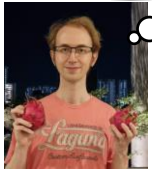
In Summary: Comparison Table

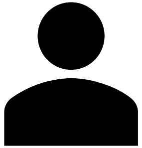
FEATURE	RED	WHITE
Scales	Narrower, more	Wider, fewer
Skin	Deep dark red	Bright pinkish
Flower margins	Red-purple	Green-yellow
Branches	Wavy, spiky	Milder, less spiky
Anti-oxidants	More	Less
Sugar content	Usually more	Less

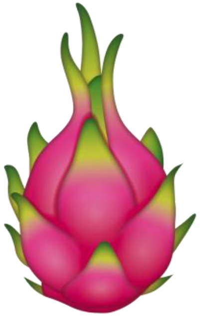
https://zenyrgarden.com/what-is-the-difference-between-red-dragon-fruit-and-white-dragon-fruit/

White vs. Red Dragon Fruits

How can I test whether you can really tell the difference?

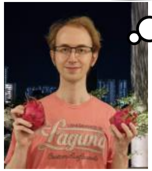


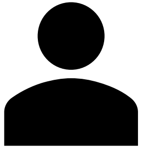


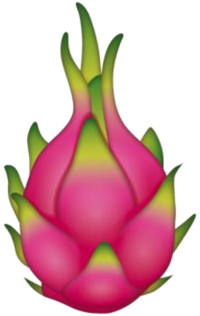


White vs. Red Dragon Fruits

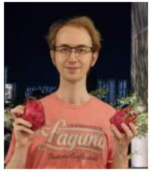
How can I test whether you can
really tell the difference?

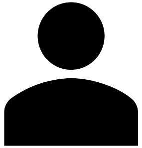


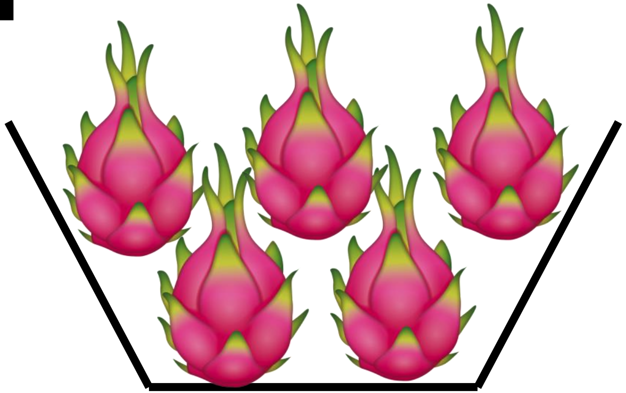




White vs. Red Dragon Fruits



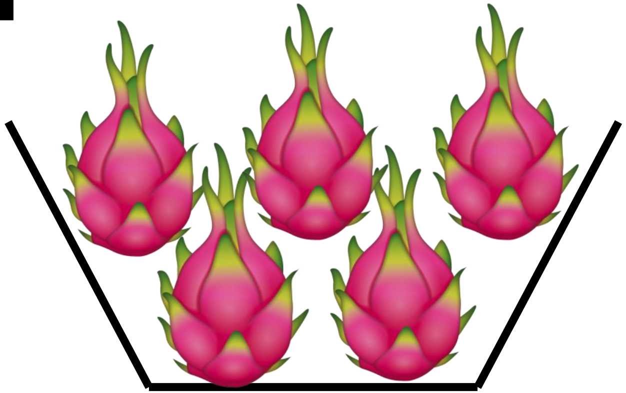




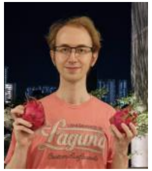
White vs. Red Dragon Fruits

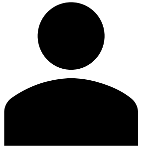
Bring me all red ones!

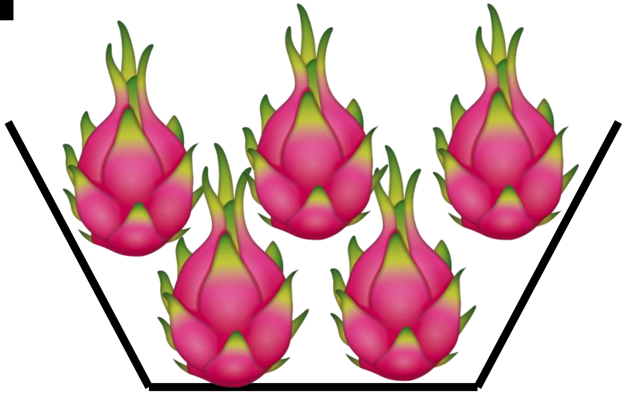
2 fruits



White vs. Red Dragon Fruits

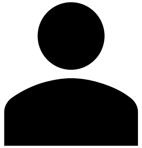




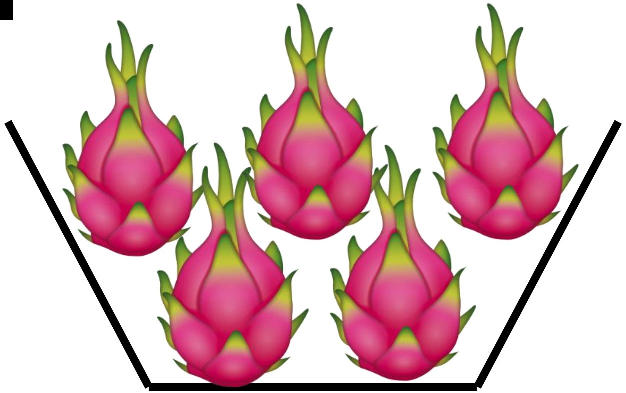


White vs. Red Dragon Fruits

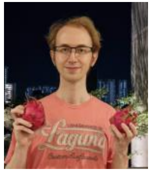
Bring me all **but** the red ones!

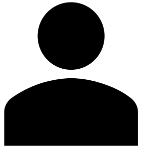


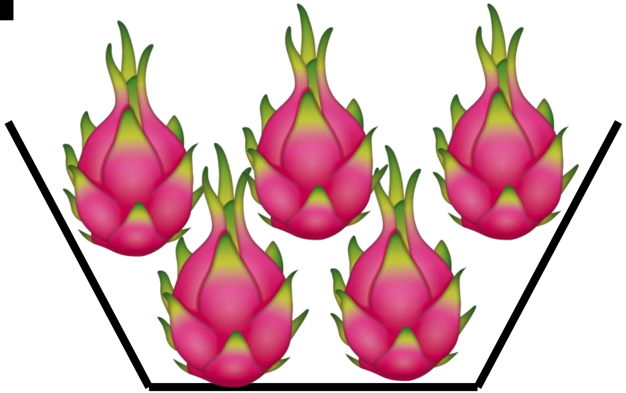
4 fruits



White vs. Red Dragon Fruits







2 fruits

4 fruits

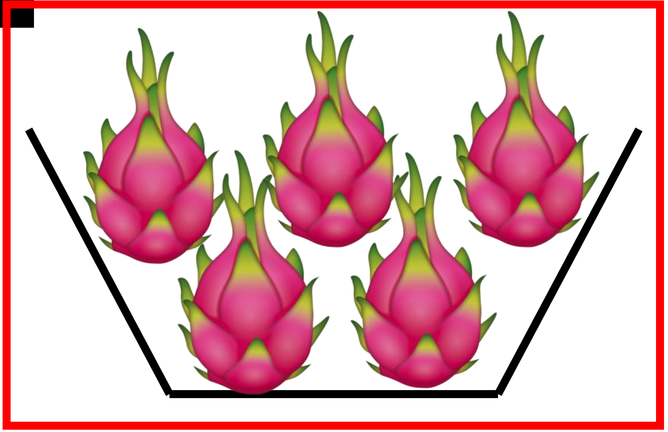
---

6 fruits

White vs. Red Dragon Fruits

You likely classified a dragon fruit as **both** a red and white!

5 fruits



2 fruits

4 fruits

6 fruits

Insight

Insight: Every object in a (mathematical) universe is either **a red dragon fruit** or **not**

Ternary Logic

Consider a predicate φ and a given row r.
Exactly **one** of the following must hold:

- φ
- NOT φ
- φ IS NULL

Ternary Logic

Consider a predicate φ and a given row r.
Exactly **one** of the following must hold:

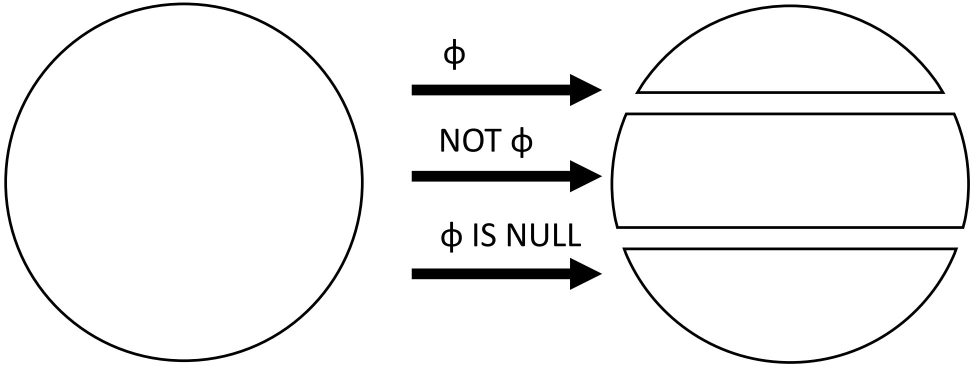
- φ
- NOT φ
- φ IS NULL

ternary predicate variants

Ternary Logic

Consider a predicate φ and a given row r.
Exactly **one** of the following must hold:

- φ
- NOT φ
- φ IS NULL



Motivating Example

t0

c0
0.0

t1

c0
-0.0

How did this insight allow us to detect this bug?

MySQL

```sql
SELECT * FROM t0, t1
WHERE t0.c0 = t1.c0;
```

t0.c0	t1.c0

https://bugs.mysql.com/bug.php?id=99122

Example: MySQL

```sql
SELECT * FROM t0, t1;
```

t0.c0	t1.c0
0.0	-0.0

ϕ

```sql
SELECT * FROM t0, t1 WHERE t0.c0=t1.c0
UNION ALL
SELECT * FROM t0, t1 WHERE NOT (t0.c0=t1.c0)
UNION ALL
SELECT * FROM t0, t1 WHERE (t0.c0=t1.c0) IS NULL;
```

t0.c0	t1.c0

≠

https://bugs.mysql.com/bug.php?id=99122

Metamorphic Testing

```
SELECT * FROM t0, t1;
```

t0.c0	t1.c0
0.0	-0.0

Derive

```
SELECT * FROM t0, t1 WHERE t0.c0=t1.c0
UNION ALL
SELECT * FROM t0, t1 WHERE NOT (t0.c0=t1.c0)
UNION ALL
SELECT * FROM t0, t1 WHERE (t0.c0=t1.c0) IS NULL;
```

Metamorphic Testing

Test Case → Execute → Result

Derive

```sql
SELECT * FROM t0, t1 WHERE t0.c0=t1.c0
UNION ALL
SELECT * FROM t0, t1 WHERE NOT (t0.c0=t1.c0)
UNION ALL
SELECT * FROM t0, t1 WHERE (t0.c0=t1.c0) IS NULL;
```

Metamorphic Testing

Test Case

Execute

Result

Derive

Derived Test Case

Scope

- WHERE
- GROUP BY
- HAVING
- DISTINCT queries
- Aggregate functions

Similar insights can be used to test **other SQL features**

Testing WHERE Clauses

Q	Q'_{ptern}	$\Diamond(Q'_p, Q'_{\neg p}, Q'_{p\ \text{IS NULL}})$
SELECT <columns> FROM <tables> [<joins>]	SELECT <columns> FROM <tables> [<joins>] WHERE p_{tern}	$Q'_p \uplus Q'_{\neg p} \uplus Q'_{p\ \text{IS NULL}}$

Testing WHERE Clauses

Q	Q'_{ptern}	$\Diamond(Q'_p, Q'_{\neg p}, Q'_{p\text{ IS NULL}})$
SELECT <columns> **FROM** <tables> [<joins>]	**SELECT** <columns> **FROM** <tables> [<joins>] **WHERE** p_{tern}	$Q'_p \uplus Q'_{\neg p} \uplus Q'_{p\text{ IS NULL}}$

Testing WHERE Clauses

Q	Q'_{ptern}	$\lozenge(Q'_p, Q'_{\neg p}, Q'_{p \text{ IS NULL}})$
SELECT <columns> FROM <tables> [<joins>]	SELECT <columns> FROM <tables> [<joins>] WHERE p_{tern}	$Q'_p \uplus Q'_{\neg p} \uplus Q'_{p \text{ IS NULL}}$

Testing WHERE Clauses

Q	Q'_{ptern}	$\Diamond(Q'_p, Q'_{\neg p}, Q'_{p\text{ IS NULL}})$
SELECT <columns> FROM <tables> [<joins>]	SELECT <columns> FROM <tables> [<joins>] WHERE p_{tern}	$Q'_p \uplus Q'_{\neg p} \uplus Q'_{p\text{ IS NULL}}$

Testing WHERE Clauses

Q	Q'_{ptern}	$\Diamond(Q'_p, Q'_{\neg p}, Q'_{p\ \text{IS NULL}})$

`SELECT <columns>`
`FROM <tables>`
`[<joins>]`
	`SELECT <columns>`
	`FROM <tables>`
	`[<joins>]`
	`WHERE` p_{tern}
		$Q'_p \uplus Q'_{\neg p} \uplus Q'_{p\ \text{IS NULL}}$

The multiset addition can be implemented using **`UNION ALL`**

Scope

- WHERE
- GROUP BY
- HAVING
- DISTINCT queries
- $\boxed{\text{Aggregate functions}}$

Testing Self-decomposable Aggregate Functions

Q	Q'_{ptern}	$\Diamond(Q'_p, Q'_{\neg p}, Q'_{p \text{ IS NULL}})$
SELECT MAX(<e>) FROM <tables> [<joins>]	SELECT MAX(<e>) FROM <tables> [<joins>] WHERE p_{tern};	$\text{MAX}(Q'_p \uplus Q'_{\neg p} \uplus Q'_{p \text{ IS NULL}})$

A partition is an **intermediate result**, rather than
a subset of the result set

Testing Self-decomposable Aggregate Functions

Q	Q'_{ptern}	$\Diamond(Q'_p, Q'_{\neg p}, Q'_{p\ \text{IS NULL}})$

`SELECT MAX(<e>)`
`FROM <tables>`
`[<joins>]`
	`SELECT MAX(<e>)`
	`FROM <tables>`
	`[<joins>]`
	`WHERE` p_{tern}`;`
		$MAX(Q'_p \uplus Q'_{\neg p} \uplus Q'_{p\ \text{IS NULL}})$

We use **MAX** in the **composition operator** to compute the overall maximum value

Bug Example: CockroachDB

```sql
SET vectorize=experimental_on;
CREATE TABLE t0(c0 INT);
CREATE TABLE t1(c0 BOOL) INTERLEAVE IN PARENT t0(rowid);
INSERT INTO t0(c0) VALUES (0);
INSERT INTO t1(rowid, c0) VALUES(0, TRUE);
```

```sql
SELECT MAX(t1.rowid)
FROM t1;
```

```sql
SELECT MAX(aggr) FROM (
    SELECT MAX(t1.rowid) as aggr FROM t1 WHERE '+' >= t1.c0 UNION ALL
    SELECT MAX(t1.rowid) as aggr FROM t1 WHERE NOT('+' >= t1.c0) UNION ALL
    SELECT MAX(t1.rowid) as aggr FROM t1 WHERE ('+' >= t1.c0) IS NULL
);
```

CockroachDB → NULL

CockroachDB → 0

NULL ≠ 0

Bug Example: CockroachDB

```sql
SET vectorize=experimental_on;
CREATE TABLE t0(c0 INT);
CREATE TABLE t1(c0 BOOL) INTERLEAVE IN PARENT t0(rowid);
INSERT INTO t0(c0) VALUES (0);
INSERT INTO t1(rowid, c0) VALUES(0, TRUE);
```

```sql
SELECT MAX(t1.rowid)
FROM t1;
```

CockroachDB

NULL

```sql
SELECT MAX(aggr) FROM (
    SELECT MAX(t1.rowid) as aggr FROM t1 WHERE '+' >= t1.c0 UNION ALL
    SELECT MAX(t1.rowid) as aggr FROM t1 WHERE NOT('+' >= t1.c0) UNION ALL
    SELECT MAX(t1.rowid) as aggr FROM t1 WHERE ('+' >= t1.c0) IS NULL
);
```

CockroachDB

0

NULL ≠ 0

Testing Decomposable Aggregate Functions

Q	Q'_{ptern}	$\Diamond(Q'_p, Q'_{\neg p}, Q'_{p \text{ IS NULL}})$

`SELECT AVG(<e>)`
`FROM <tables>`
`[<joins>];`
	`SELECT SUM(<e>) as s,`
	`COUNT(<e>) as c`
	`FROM <tables>`
	`[<joins>];`
		$\frac{SUM(s(Q'_p \uplus Q'_{\neg p} \uplus Q'_{p \text{ IS NULL}}))}{SUM(c(Q'_p \uplus Q'_{\neg p} \uplus Q'_{p \text{ IS NULL}}))}$

A **single value** to represent a partition is **insufficient**

Evaluation

We **evaluated the effectiveness of** our approach in a three-month period on **seven** widely-used DBMSs

Evaluation: Found Bugs

			Closed	
DBMS	Fixed	Verified	Intended	Duplicate
SQLite	4	0	0	0
MySQL	1	6	3	0
H2	16	2	0	1
CockroachDB	23	8	0	0
TiDB	26	35	0	1
DuckDB	72	0	0	2

We found **193 unique, previously unknown bugs**, 142 of which have been fixed!

Evaluation: Found Bugs

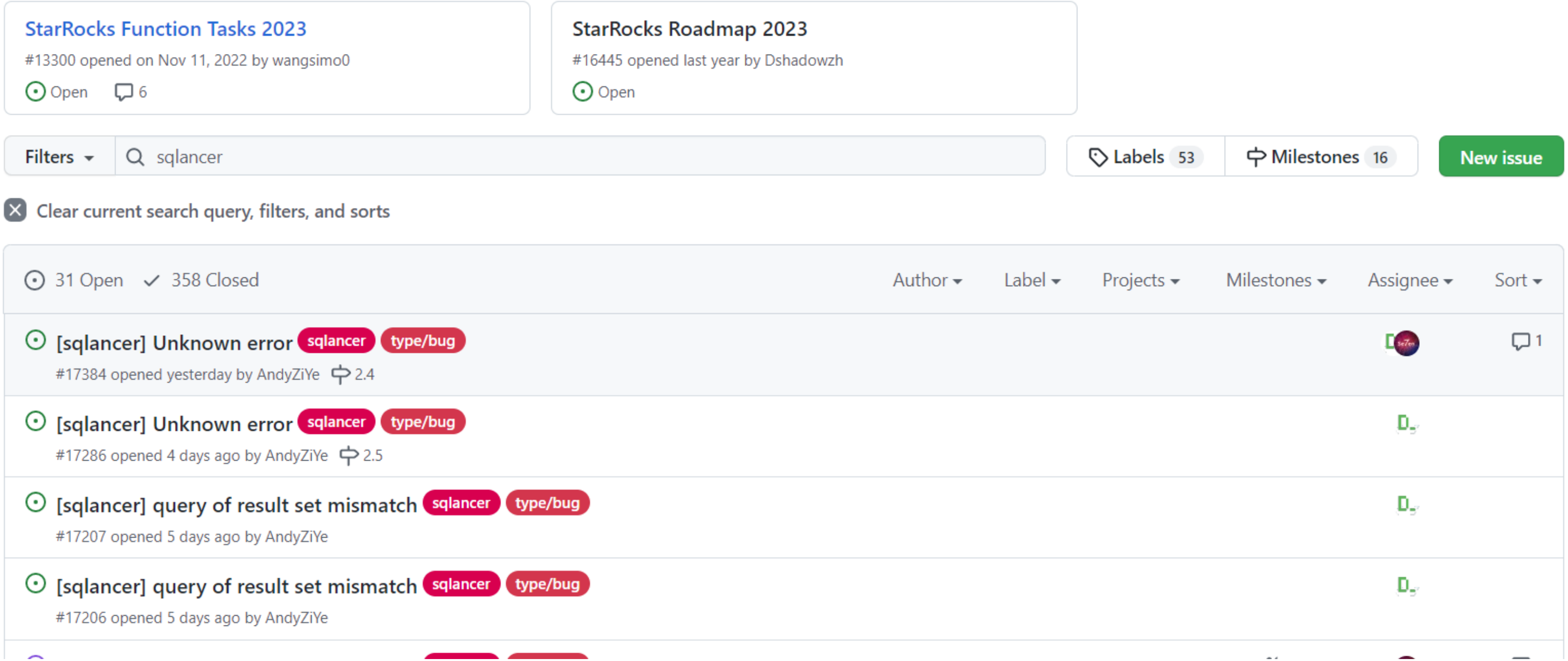
	Query Partitioning Oracle						
DBMS	WHERE	Aggregate	GROUP BY	HAVING	DISTINCT	Error	Crash
SQLite	0	3	0	0	1	0	0
CockroachDB	3	3	0	1	0	22	2
TiDB	29	0	1	0	0	27	4
MySQL	7	0	0	0	0	0	0
DuckDB	21	4	1	2	1	13	19
H2	2	0	0	0	0	16	0

Evaluation: Found Bugs

	Query Partitioning Oracle						
DBMS	WHERE	Aggregate	GROUP BY	HAVING	DISTINCT	Error	Crash
SQLite	0	3	0	0	1	0	0
CockroachDB	3	3	0	1	0	22	2
TiDB	29	0	1	0	0	27	4
MySQL	7	0	0	0	0	0	0
DuckDB	21	4	1	2	1	13	19
H2	2	0	0	0	0	16	0

The WHERE oracle is the **simplest**, but **most effective** oracle

TLP in Production: Example StarRocks



TLP in Production: Example CockroachDB

cockroachdb / cockroach Public

Code | Issues 5k+ | Pull requests 895 | Discussions | Actions

✓ **roachtest: add ternary logic partitioning (TLP) test**

```
This commit adds a roachtest that performs ternary logic partitioning
(TLP) testing. TLP is a method for logically testing a database which is
based on the logical guarantee that for a given predicate `p`, all rows
must satisfy exactly one of the following three predicates: `p`, `NOT
p`, `p IS NULL`. Unioning the results of all three "partitions" should
yield the same result as an "unpartitioned" query with a `true`
predicate.

TLP is implemented in [SQLancer](https://github.com/sqlancer/sqlancer)
and more information can be found at
https://www.manuelrigger.at/preprints/TLP.pdf.

We currently implement a limited form of TLP that only runs queries of
the form `SELECT * FROM table WHERE <predicate>` where `<predicate>` is
randomly generated. We also only verify that the number of rows returned
by the unpartitioned and partitioned queries are equal, not that the
values of the rows are equal. See the documentation for
`Smither.GenerateTLP` for more details.
```

SQLancer Retweeted

Isaac Wong
@itemujinwong

Fun thread from @largedatabank on a cool testing strategy we use at @CockroachDB called TLP (Ternary Logic Partitioning). It's a fancy name for a brilliant but simple testing technique.

> **Jordan Lewis** @largedatabank · Feb 4, 2022
> Time to merge a small enhancement to @CockroachDB's TLP implementation that I developed a few weeks ago on my stream!
>
> TLP is so neat that I want to talk about it a bit, and explain the enhancement as well.
>
> Here's the PR: github.com/cockroachdb/co...
>
> 👇
>
> 1/🧵
>
> Show this thread

5:25 AM · Feb 5, 2022

TLP for Testing Graph Database Systems

ISSTA 2023

Testing Graph Database Engines via Query Partitioning

Matteo Kamm
ETH Zurich
Switzerland
matkamm@student.ethz.ch

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

Chengyu Zhang
ETH Zurich
Switzerland
chengyu.zhang@inf.ethz.ch

Zhendong Su
ETH Zurich
Switzerland
zhendong.su@inf.ethz.ch

ABSTRACT

Graph Database Management Systems (GDBMSs) store data as graphs and allow the efficient querying of nodes and their relationships. Logic bugs are bugs that cause a GDBMS to return an incorrect result for a given query (*e.g.*, by returning incorrect nodes or relationships). The impact of such bugs can be severe, as they often go unnoticed. The core insight of this paper is that Query Partitioning, a test oracle that has been proposed to test Relational Database Systems, is applicable to testing GDBMSs as well. The core idea of Query Partitioning is that, given a query, multiple queries are derived whose results can be combined to reconstruct the given query's result. Any discrepancy in the result indicates a logic bug. We have implemented this approach as a practical tool named GDBMeter and evaluated GDBMeter on three popular GDBMSs and found a total of 41 unique, previously unknown bugs. We consider 14 of them to be logic bugs, the others being error or crash bugs. Overall, 27 of the bugs have been fixed, and 35 confirmed. We compared our approach to the state-of-the-art approach to testing GDBMS, which relies on differential testing; we found that it results in a high number of false alarms, while Query Partitioning reported actual logic bugs without any false alarms. Furthermore, despite the previous efforts in testing Neo4j and JanusGraph, we found 13 additional bugs. The developers appreciate our work and plan to integrate GDBMeter into their testing process. We expect that this

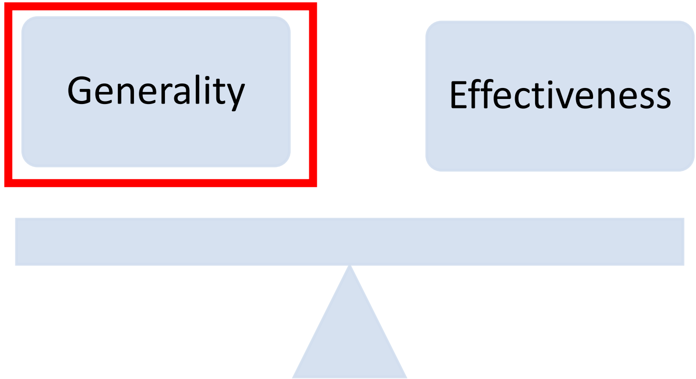
1 INTRODUCTION

Graph Database Management Systems (GDBMS) [21, 28, 31] allow storing and querying data as graphs. In recent years, the popularity of such systems has increased drastically due to their applicability in social networks, knowledge graphs [16], and fraud detection [35]. Examples of the most popular GDBMSs are Neo4j [10], JanusGraph [6], RedisGraph [12], and Memgraph [9].

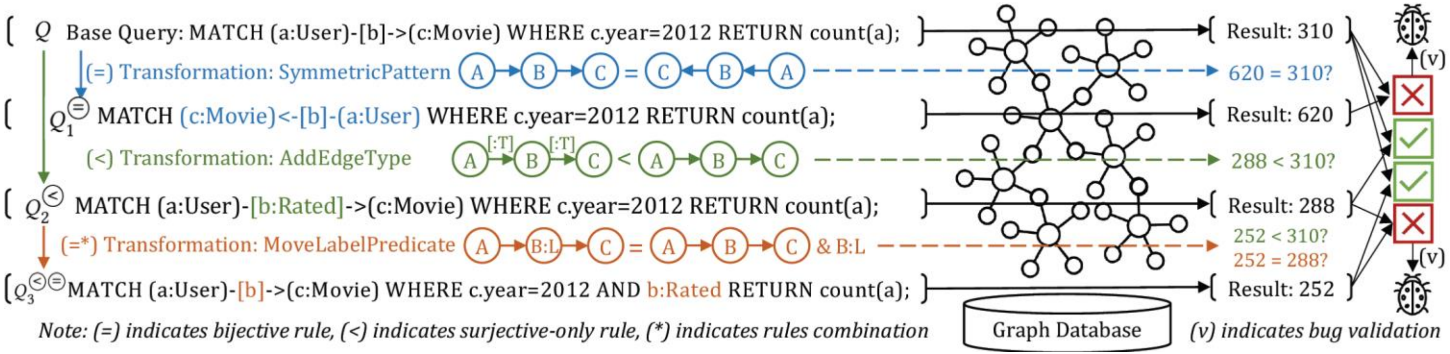
As with any other software, GDBMSs can be affected by various kinds of bugs. A notorious category of bugs are logic bugs, which are bugs that cause the GDBMS to compute an incorrect result. For example, for a given query, a GDBMS might mistakenly omit a vertex from the result set or include an edge that should not be part of the result. Such bugs are difficult to detect by users and might go unnoticed, especially considering the complexity of modern GDBMSs (*e.g.*, Neo4j has 468k LOC).

The state-of-the-art approach to testing GDBMSs, Grand [38], is based on *differential testing* [27]. It generates a test case that is sent to multiple GDBMSs; if the outputs disagree, at least one of the systems is assumed to be affected by a bug. Grand found 21 previously unknown bugs in six GDBMSs, of which 18 bugs were confirmed, 7 were fixed, and 2 were logic bugs. Despite its success in finding bugs, differential testing has major drawbacks in this context. GDBMSs support various query languages that differ in syntax and semantics. Grand was realized for Gremlin, which many,

Matteo Kamm

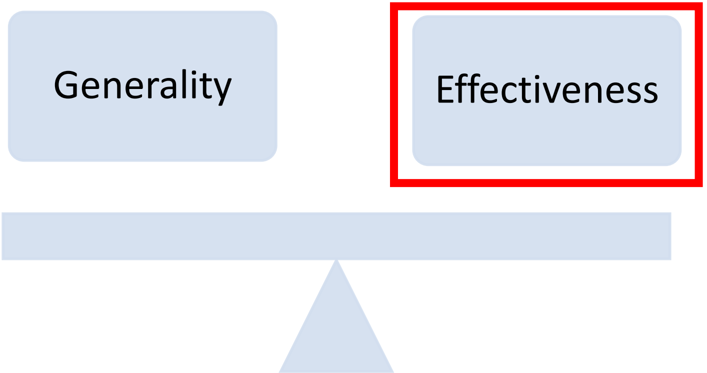


Utilizing Graph Properties

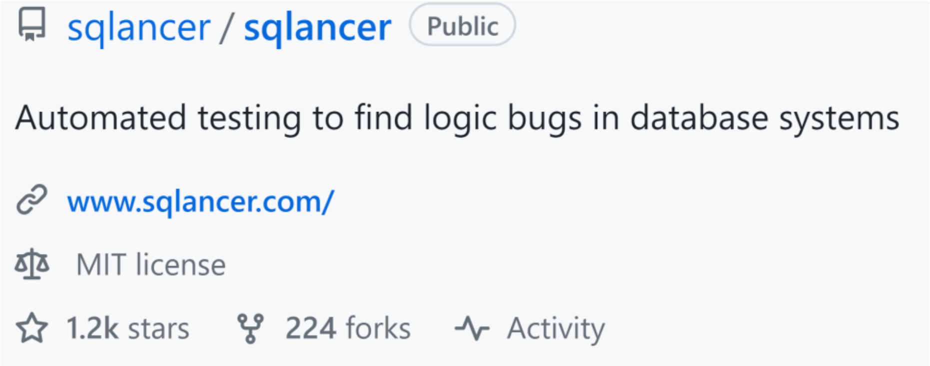


(Under submission)

Yuancheng Jiang
https://yuanchengjiang.github.io/



SQLancer



https://github.com/sqlancer/sqlancer

Over 20 Supported Database Systems!

Materialize

QuestDB

mongoDB®

DORIS

yugabyteDB

OCEANBASE

citusdata

DuckDB

TiDB

H2

SQLite

HyperSQL

MySQL™

Azure Cosmos DB

Databend

ArangoDB

ClickHouse

CockroachDB

MariaDB

CnosDB

Keeping up With Evolving DBMSs

- ✖ DBMS Tests (MariaDB)
- ✔ DBMS Tests (Materialize)
- ✔ QPG Tests (Materialize)
- ✖ DBMS Tests (MySQL)
- ✔ DBMS Tests (PostgreSQL)
- ✔ DBMS Tests (SQLite)
- ✔ QPG Tests (SQLite)
- ✔ DBMS Tests (TiDB)
- ✖ QPG Tests (TiDB)
- ✔ DBMS Tests (YugabyteDB)
- ✖ DBMS Tests (Apache Doris)
- ✔ Java 13 Compatibility (Duck...
- ✔ Java 14 Compatibility (Duck...

New approaches to keep up with (evolving) systems?

Additional Organizations

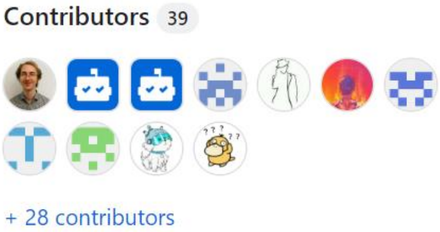
40 Open ✓ 366 Closed

Author ▾ Label ▾ Projects ▾ Milestones ▾ Assignee ▾ Sort ▾

- [spill] result error when turn on spill `type/bug` 1
  #23535 by colorfulu was closed last week 3.0
- Sqlancer report error: StarRocks process failed `bug` `sqlancer`
  #22349 opened on Apr 24 by satanson
- Sqlancer report error: Getting analyzing error from line 1, column 591 to line 1, column 632. Detail message: Column type ARRAY<varchar(65533)> does not support binary predicate operation. `bug` `sqlancer` 1
  #22341 by satanson was closed on Apr 26

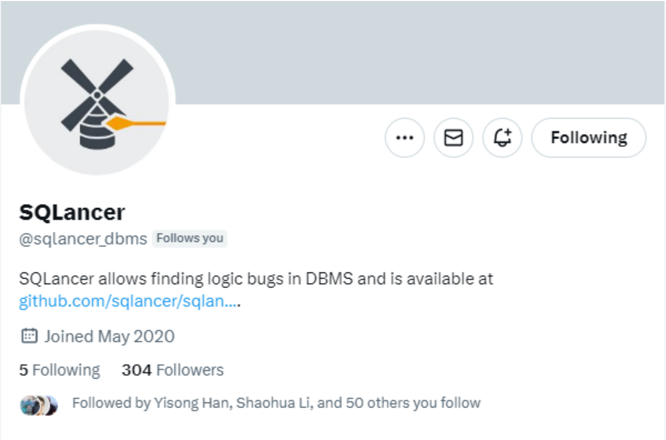
https://github.com/StarRocks/starrocks/issues?q=is%3Aissue+sqlancer+

Community

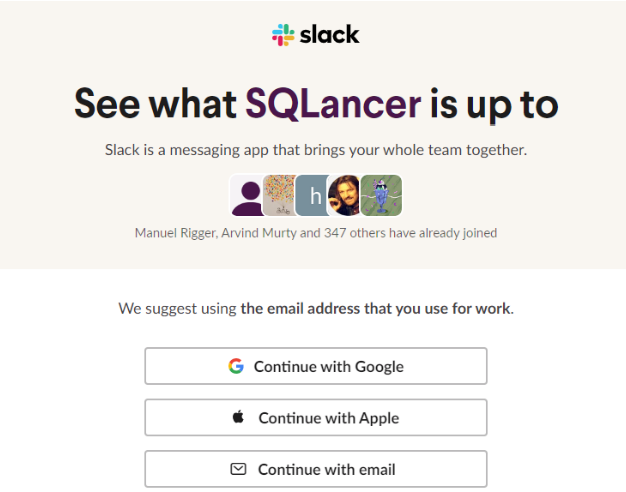


Google
Summer of Code

https://summerofcode.withgoogle.com/programs/2023/organizations/sqlancer



@sqlancer_dbms

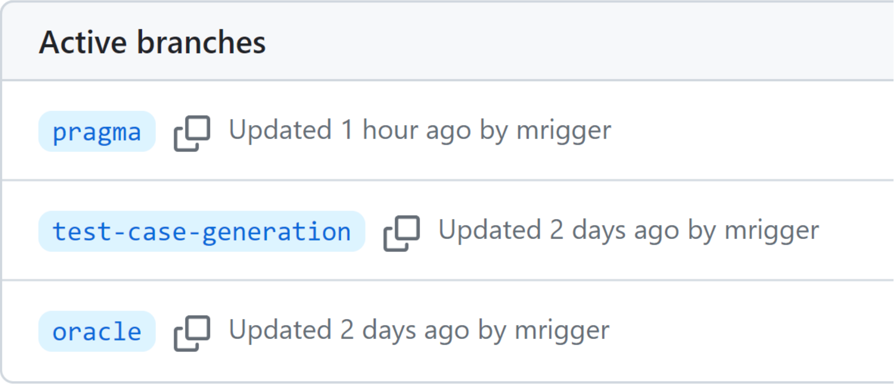


https://sqlancer.slack.com/intl/en-gb/join/shared_invite/zt-eozrcao4-ieG29w1LNaBDMF7OB_~ACg#/shared-invite/email

Installation

- Tested options
  - Ubuntu
  - WSL + Ubuntu
- Should work on any OS that supports Java (+ Maven and Git)

Repository



https://github.com/mrigger/sqlancer-tarot

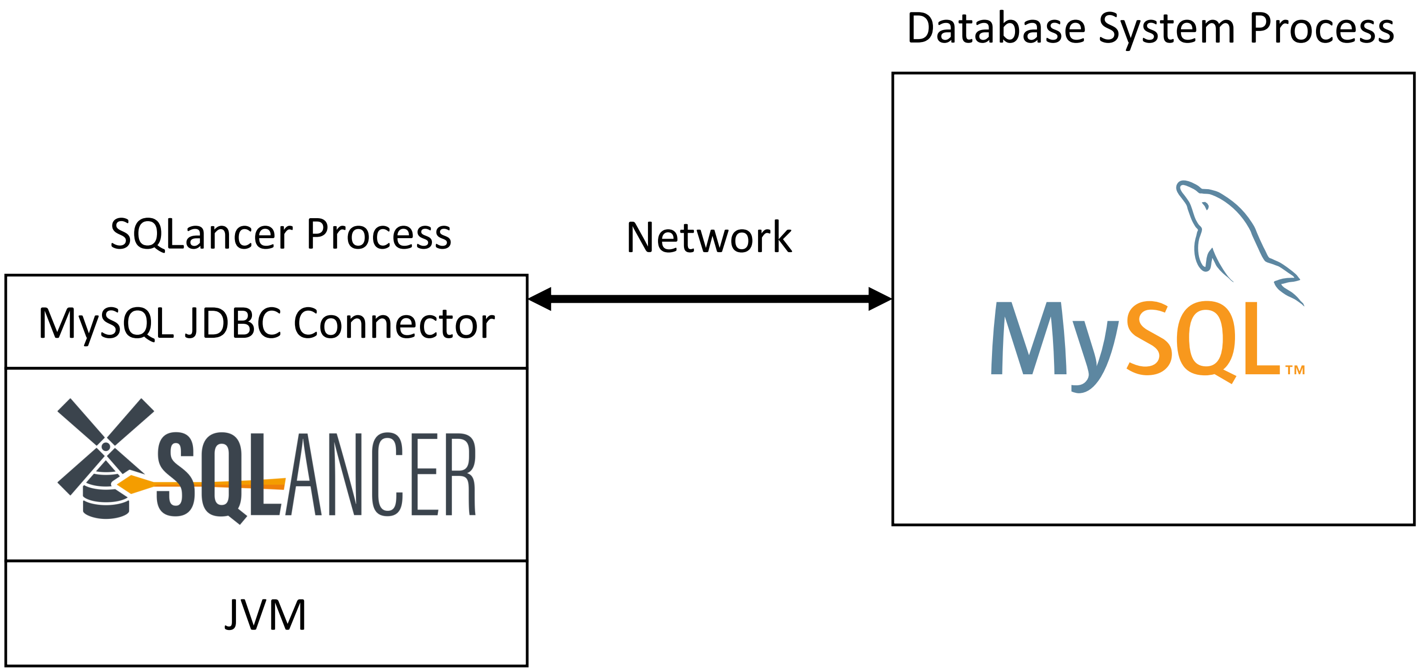
Installation

```
$ git clone https://github.com/mrigger/sqlancer-tarot
$ cd sqlancer-tarot
$ mvn package -DskipTests
$ cd target
$ java -jar sqlancer-*.jar duckdb
```

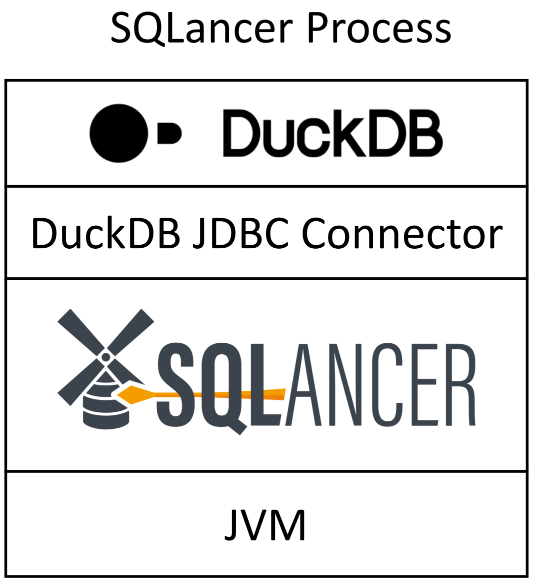
If you see such output, your setup works

```
[2023/06/07 08:24:00] Executed 38362 queries  (7632 queries/s; 2.19/s dbs, successful statements: 88%). Threads shut down: 0.
[2023/06/07 08:24:05] Executed 143039 queries (20973 queries/s; 0.80/s dbs, successful statements: 93%). Threads shut down: 0.
[2023/06/07 08:24:10] Executed 266483 queries (24867 queries/s; 0.20/s dbs, successful statements: 95%). Threads shut down: 0.
[2023/06/07 08:24:15] Executed 394076 queries (25615 queries/s; 0.00/s dbs, successful statements: 95%). Threads shut down: 0.
```

Client Server Database Systems



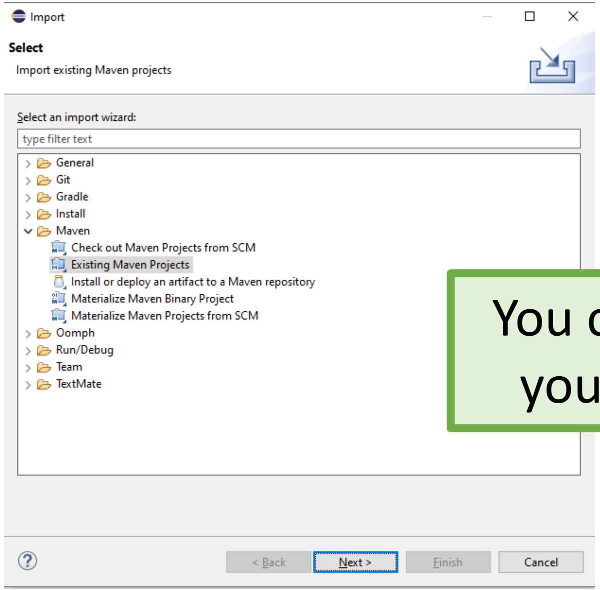
Embedded Database Systems



We will use an **embedded database** system, **DuckDB**, which is included in the JDBC driver and runs in the same process as the JVM

Working with Eclipse

File → Import →Existing Maven Projects



You can use any other editor you are comfortable with!

https://www.eclipse.org/downloads/

Architecture

DuckDB TLP Oracle	MySQL TLP Oracle	…
DuckDB Query Generator	MySQL Query Generator	…
DuckDB Database Generator	MySQL Database Generator	…
SQLancer Base (logging, thread handling, …)		

The DBMS-specific components are large and share less code than they should

DatabaseProvider

```
public class DuckDBProvider extends SQLProviderAdapter<DuckDBGlobalState, DuckDBOptions> {
    public enum Action implements AbstractAction<DuckDBGlobalState> {

        INSERT(DuckDBInsertGenerator::getQuery), //
        CREATE_INDEX(DuckDBIndexGenerator::getQuery), //
        VACUUM((g) -> new SQLQueryAdapter("VACUUM;")), //
        ANALYZE((g) -> new SQLQueryAdapter("ANALYZE;")), //
        DELETE(DuckDBDeleteGenerator::generate), //
        UPDATE(DuckDBUpdateGenerator::getQuery), //
        CREATE_VIEW(DuckDBViewGenerator::generate), //
        …
    });
}
```

The DatabaseProvider subclasses are the main entry points for a DBMS implementation

Statement Generators and Expected Errors

```java
private SQLQueryAdapter generate() {
    sb.append("INSERT INTO ");
    DuckDBTable table = globalState.getSchema().getRandomTable(t -> !t.isView());
    List<DuckDBColumn> columns = table.getRandomNonEmptyColumnSubset();
    sb.append(table.getName());
    sb.append("(");
    sb.append(columns.stream().map(c -> c.getName()).collect(Collectors.joining(", ")));
    sb.append(")");
    sb.append(" VALUES ");
    insertColumns(columns);
    DuckDBErrors.addInsertErrors(errors);
    return new SQLQueryAdapter(sb.toString(), errors);
}
```

```java
errors.add("PRIMARY KEY or
UNIQUE constraint violated");
```

Some semantic errors are difficult to prevent, while others might be unexpected (e.g., database corruptions)

Statement Generators and Expected Errors

```
private SQLQueryAdapter generate() {
    sb.append("INSERT INTO ");
    DuckDBTable table = globalState.getSchema().getRandomTable(t -> !t.isView());
    List<DuckDBColumn> columns = table.getRandomNonEmptyColumnSubset();
    sb.append(table.getName());
    sb.append("(");
    sb.append(columns.stream().map(c -> c.getName()).collect(Collectors.joining(", ")));
    sb.append(")");
    sb.append(" VALUES ");
    insertColumns(columns);
    DuckDBErrors.addInsertErrors(errors);
    return new SQLQueryAdapter(sb.toString(), errors);
}
```

```
errors.add("PRIMARY KEY or UNIQUE constraint violated");
```

Better approaches to automatically derive expected errors? How to increase the validity rate?