ETH zürich

Testing Database Management Systems via Pivoted Query Synthesis

Manuel Rigger

Oct 18., 2019
Workshop on Dependable and Secure Software Systems 2019

@RiggerManuel @ast_eth

AST
ADVANCED SOFTWARE TECHNOLOGIES

Database Management Systems

Database Management Systems

PostgreSQL

SQLite

MySQL™

Who has **heard about/used** these Database Management Systems?

Databases are Used Ubiquitously

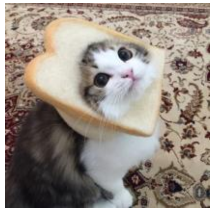
SQLite

“SQLite is the ***most used database*** *engine in the world. SQLite is built into* ***all mobile phones*** *and* ***most computers*** *and comes bundled inside* ***countless other applications*** *that people use every day.”*

https://www.sqlite.org

Relational Data Model

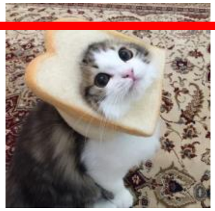
animal_pictures

animal	description	picture
Cat	A cute toast cat	
Dog	Cute dog pic	
Cat	Cat plants (cute!)	

Relational Data Model

A **database schema** describes the **tables (relations)** in the database

animal_pictures

animal	description	picture
Cat	A cute toast cat	
Dog	Cute dog pic	
Cat	Cat plants (cute!)	

Relational Data Model

animal_pictures

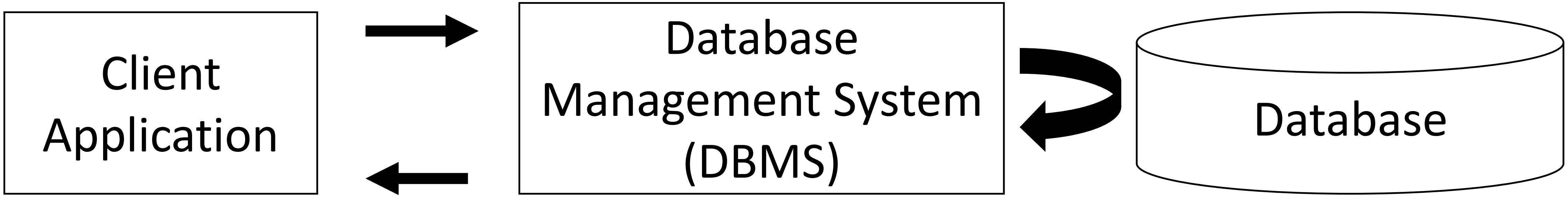
animal	description	picture
Cat	A cute toast cat	
Dog	Cute dog pic	
Cat	Cat plants (cute!)	

Structured Query Language (SQL) is a **declarative** DSL to **query and manipulate data**

```
SELECT picture, description
FROM animal_pictures
WHERE animal = 'Cat'
   AND description LIKE '%cute%'
```

Database Management Systems

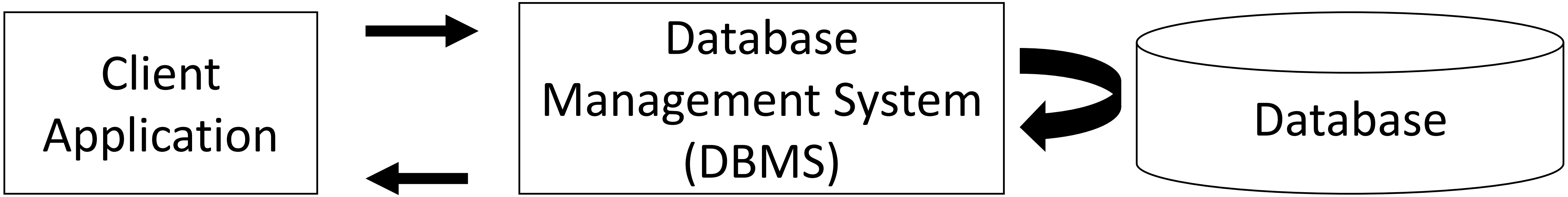
```
SELECT * FROM <table>
WHERE <cond>
```



Database Management Systems

```
SELECT * FROM <table>
WHERE <cond>
```

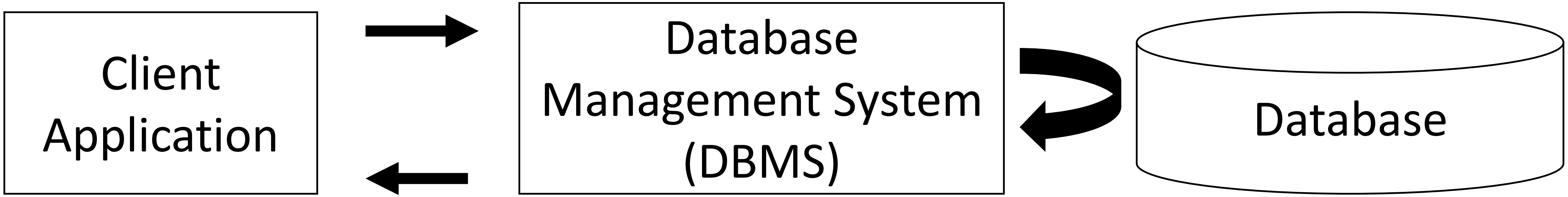
row_1	<cond>
row_2	<cond>
row_3	¬<cond>



Database Management Systems

row_1	<cond>
row_2	<cond>
row_3	¬<cond>

```
SELECT * FROM <table>
WHERE <cond>
```



row_1	<cond>
row_2	<cond> ✓

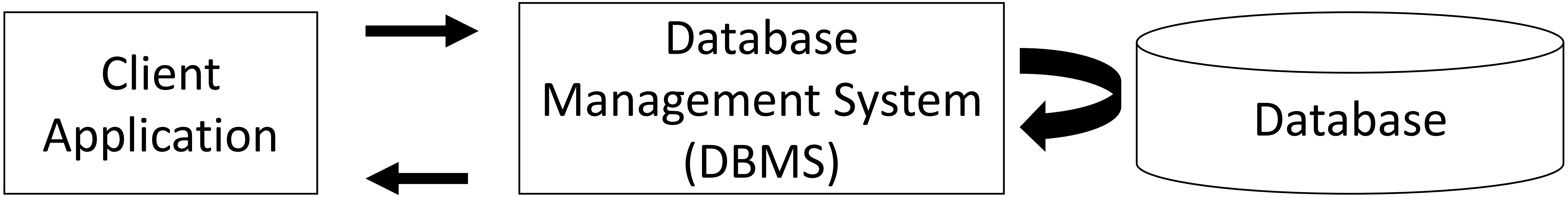
Goal

Aim: Detect **logic bugs** in DBMS

Database Management Systems

```
SELECT * FROM <table>
WHERE <cond>
```

row_1	<cond>
row_2	<cond>
row_3	¬<cond>

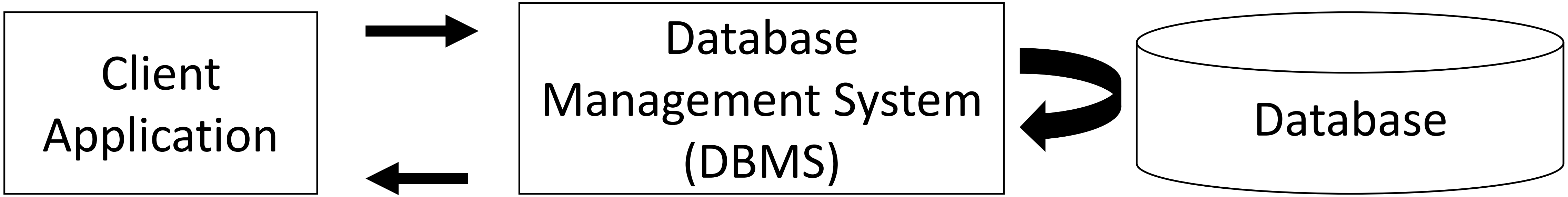


row_1	<cond>	
row_2	<cond>	✓

Database Management Systems

```
SELECT * FROM <table>
WHERE <cond>
```

row_1	**<cond>**
row_2	**<cond>**
row_3	**¬<cond>**



row_1	**<cond>**
row_3	**¬<cond>** ✘

Example Bug: SQLite

```
CREATE TABLE t1(c1, c2, c3, c4, PRIMARY KEY (c4, c3));
INSERT INTO t1(c3) VALUES (0), (0), (0), (0), (0), (0),
       (0), (0), (0), (0), (NULL), (1), (0);
UPDATE t1 SET c2 = 0;
INSERT INTO t1(c1) VALUES (0), (0), (NULL), (0), (0);
ANALYZE t1;
UPDATE t1 SET c3 = 1;
SELECT DISTINCT * FROM t1 WHERE t1.c3 = 1;
```

Example Bug: SQLite

```
CREATE TABLE t1(c1, c2, c3, c4, PRIMARY KEY (c4, c3));
INSERT INTO t1(c3) VALUES (0), (0), (0), (0), (0), (0),
       (0), (0), (0), (0), (NULL), (1), (0);
UPDATE t1 SET c2 = 0;
INSERT INTO t1(c1) VALUES (0), (0), (NULL), (0), (0);
ANALYZE t1;
UPDATE t1 SET c3 = 1;
SELECT DISTINCT * FROM t1 WHERE t1.c3 = 1;
```

ANALYZE gathers **statistics about tables**, which are then used for query planning

Example Bug: SQLite

```sql
CREATE TABLE t1(c1, c2, c3, c4, PRIMARY KEY (c4, c3));
INSERT INTO t1(c3) VALUES (0), (0), (0), (0), (0), (0),
      (0), (0), (0), (0), (NULL), (1), (0);
UPDATE t1 SET c2 = 0;
INSERT INTO t1(c1) VALUES (0), (0), (NULL), (0), (0);
ANALYZE t1;
UPDATE t1 SET c3 = 1;
SELECT DISTINCT * FROM t1 WHERE t1.c3 = 1;
```

Example Bug: SQLite

```
CREATE TABLE t1(c1, c2, c3, c4, PRIMARY KEY (c4, c3));
INSERT INTO t1(c3) VALUES (0), (0), (0), (0), (0), (0),
       (0), (0), (0), (0), (NULL), (1), (0);
UPDATE t1 SET c2 = 0;
INSERT INTO t1(c1) VALUES (0), (0), (NULL), (0), (0);
ANALYZE t1;
UPDATE t1 SET c3 = 1;
SELECT DISTINCT * FROM t1 WHERE t1.c3 = 1;
```

Expected result set

c1	c2	c3	c4
NULL	0	1	NULL
0	NULL	1	NULL
NULL	NULL	1	NULL

Example Bug: SQLite

```
CREATE TABLE t1(c1, c2, c3, c4, PRIMARY KEY (c4, c3));
INSERT INTO t1(c3) VALUES (0), (0), (0), (0), (0), (0),
      (0), (0), (0), (0), (NULL), (1), (0);
UPDATE t1 SET c2 = 0;
INSERT INTO t1(c1) VALUES (0), (0), (NULL), (0), (0);
ANALYZE t1;
UPDATE t1 SET c3 = 1;
SELECT DISTINCT * FROM t1 WHERE t1.c3 = 1;
```

Expected result set

c1	c2	c3	c4
NULL	0	1	NULL
0	NULL	1	NULL
NULL	NULL	1	NULL

Actual result set

c1	c2	c3	c4
NULL	0	1	NULL

Example Bug: SQLite

```
CREATE TABLE t1(c1, c2, c3, c4, PRIMARY KEY (c4, c3));
INSERT INTO t1(c3) VALUES (0), (0), (0), (0), (0), (0),
      (0), (0), (0), (0), (NULL), (1), (0);
UPDATE t1 SET c2 = 0;
INSERT INTO t1(c1) VALUES (0), (0), (NULL), (0), (0);
ANALYZE t1;
UPDATE t1 SET c3 = 1;
SELECT DISTINCT * FROM t1 WHERE t1.c3 = 1;
```

Expected result set

c1	c2	c3	c4
NULL	0	1	NULL
0	NULL	1	NULL
NULL	NULL	1	NULL

Actual result set

c1	c2	c3	c4
NULL	0	1	NULL

A bug in the **skip-scan optimization** caused this logic bug

Challenges

- DBMS are **tested well**

Databases are Tested Well

SQLite (~150,000 LOC) has **662 times** as much test code as source code

https://www.sqlite.org/testing.html

Databases are Tested Well

SQLite (~150,000 LOC) has **662 times** as much test code as source code

SQLite's test cases achieve **100% branch test coverage**

https://www.sqlite.org/testing.html

Databases are Tested Well

SQLite (~150,000 LOC) has **662 times** as much test code as source code

SQLite's test cases achieve **100% branch test coverage**

SQLite is **extensively fuzzed** (e.g., by Google's OS-Fuzz Project)

https://www.sqlite.org/testing.html

Databases are Tested Well

SQLite (~150,000 LOC) has **662 times** as much test code as source code

SQLite's test cases achieve **100% branch test coverage**

SQLite is **extensively fuzzed** (e.g., by Google's OS-Fuzz Project)

SQLite's performs **anomaly testing** (out-of-memory, I/O error, power failures)

https://www.sqlite.org/testing.html

Databases are Tested Well

SQLite (~150,000 LOC) has **662 times** as much test code as source code

SQLite's test cases achieve **100% branch test coverage**

SQLite is **extensively fuzzed** (e.g., by Google's OS-Fuzz Project)

SQLite's performs **anomaly testing** (out-of-memory, I/O error, power failures)

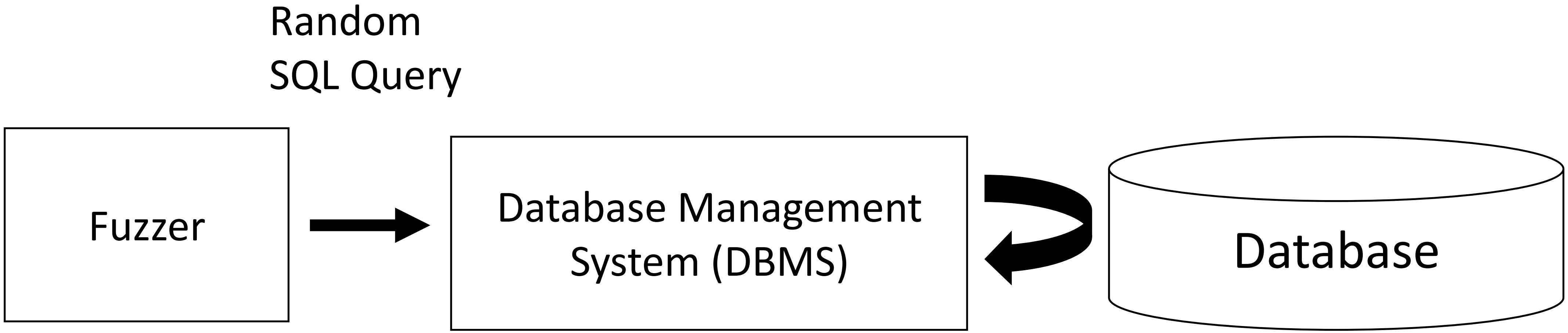
Small. Fast. **Reliable**. Choose any three.

https://www.sqlite.org/testing.html

Challenges

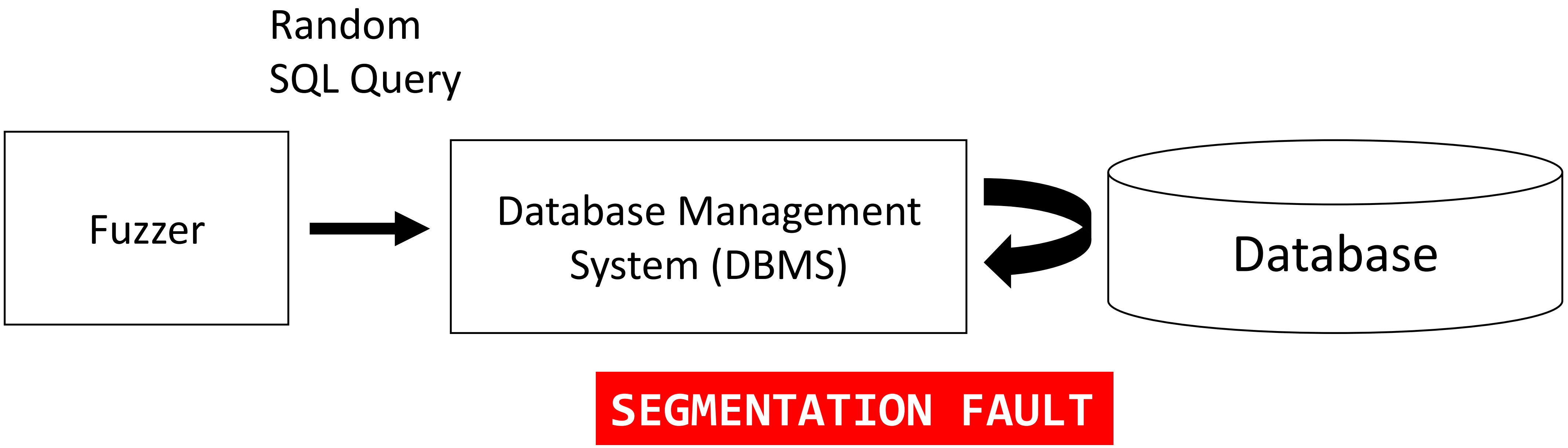
- DBMS are **tested well**
- Fuzzers are **ineffective** in **finding logic bugs**

Existing Work: Fuzzers and Query Generators



AFL, SQLSmith, QGEN (Poess et al. 2014), …

Existing Work: Fuzzers and Query Generators

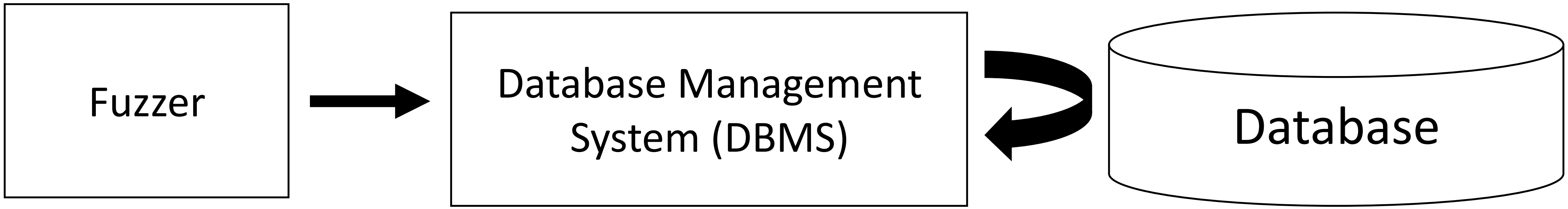


AFL, SQLSmith, QGEN (Poess et al. 2014), …

Existing Work: Fuzzers and Query Generators

Fuzzers are effective in detecting bugs that result in **crashes**

Random SQL Query



Fuzzer

Database Management System (DBMS)

Database

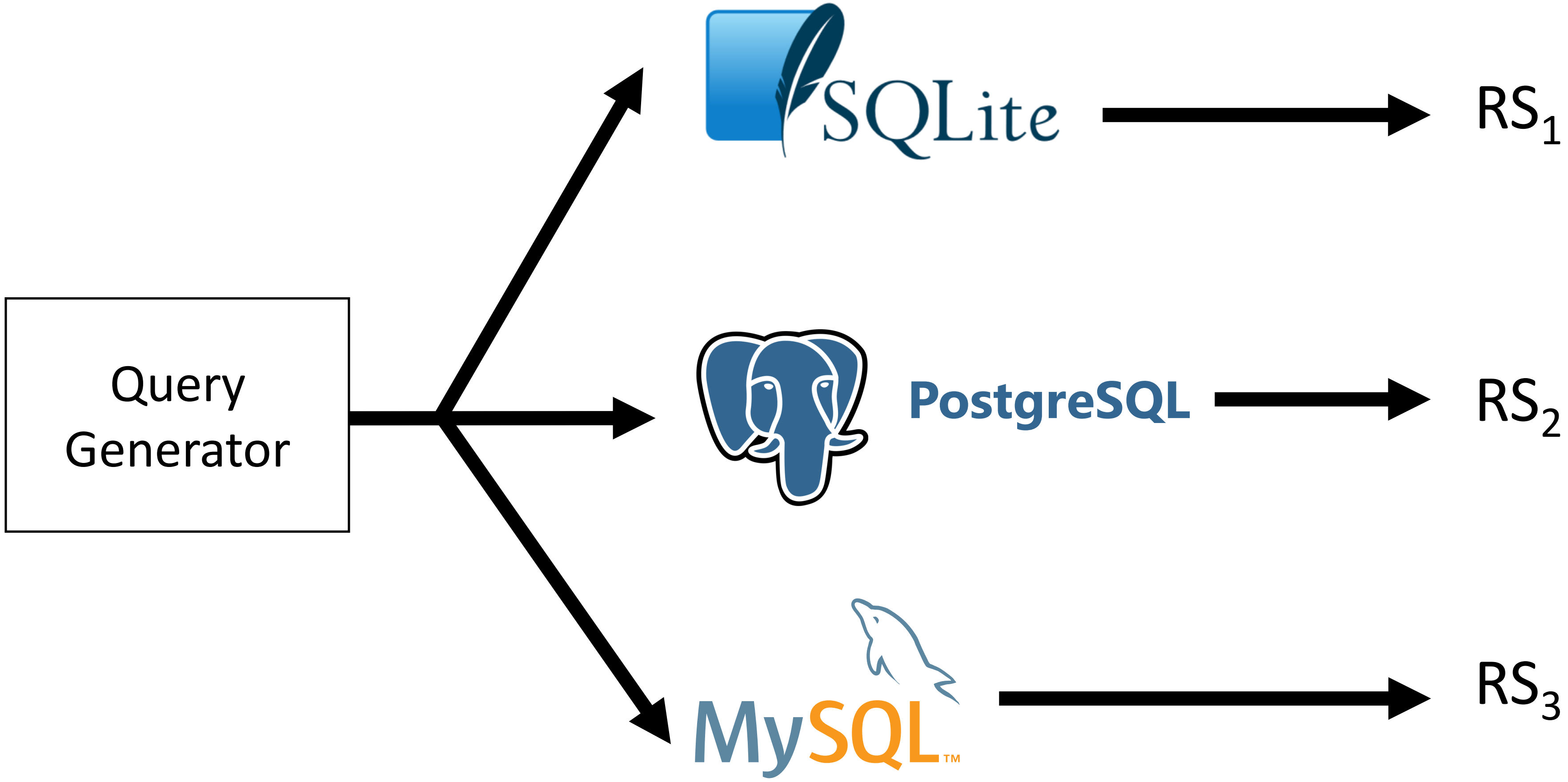
SEGMENTATION FAULT

AFL, SQLSmith, QGEN (Poess et al. 2014), …

Challenges

- DBMS are **tested well**
- Fuzzers are **ineffective** in **finding logic bugs**
- Knowing the **precise result set** for a query **is difficult**

Differential Testing



Query Generator

SQLite

PostgreSQL

MySQL

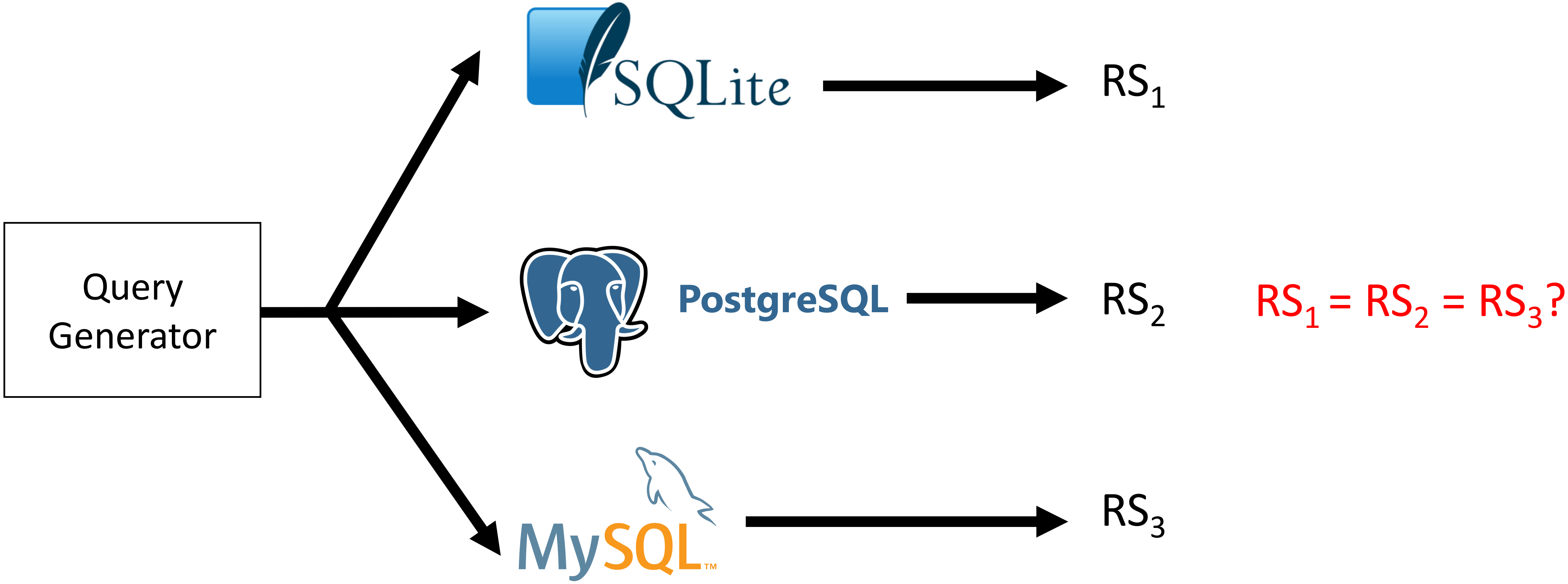
RS_1

RS_2

RS_3

$RS_1 = RS_2 = RS_3$?

Differential Testing



Query Generator

RS_1

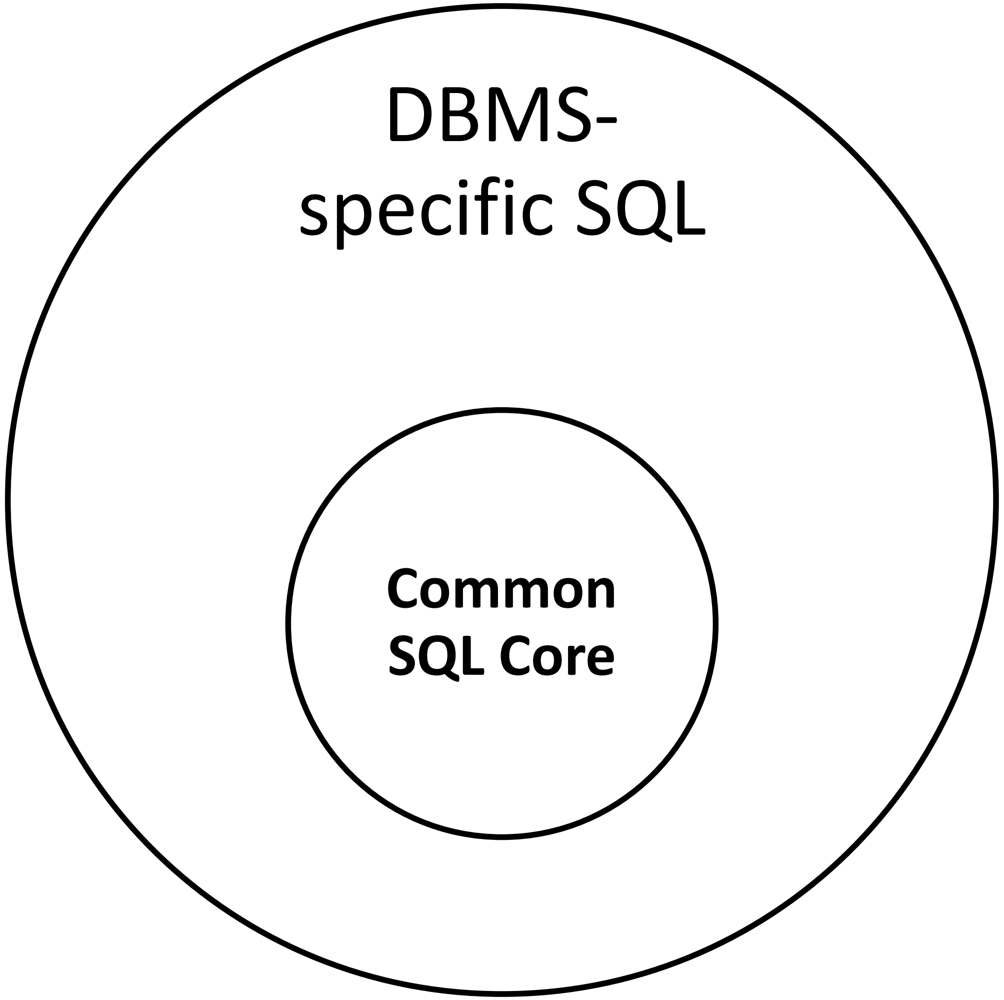
RS_2

RS_3

$RS_1 = RS_2 = RS_3$?

Differential testing applies only when systems implement the **same language**

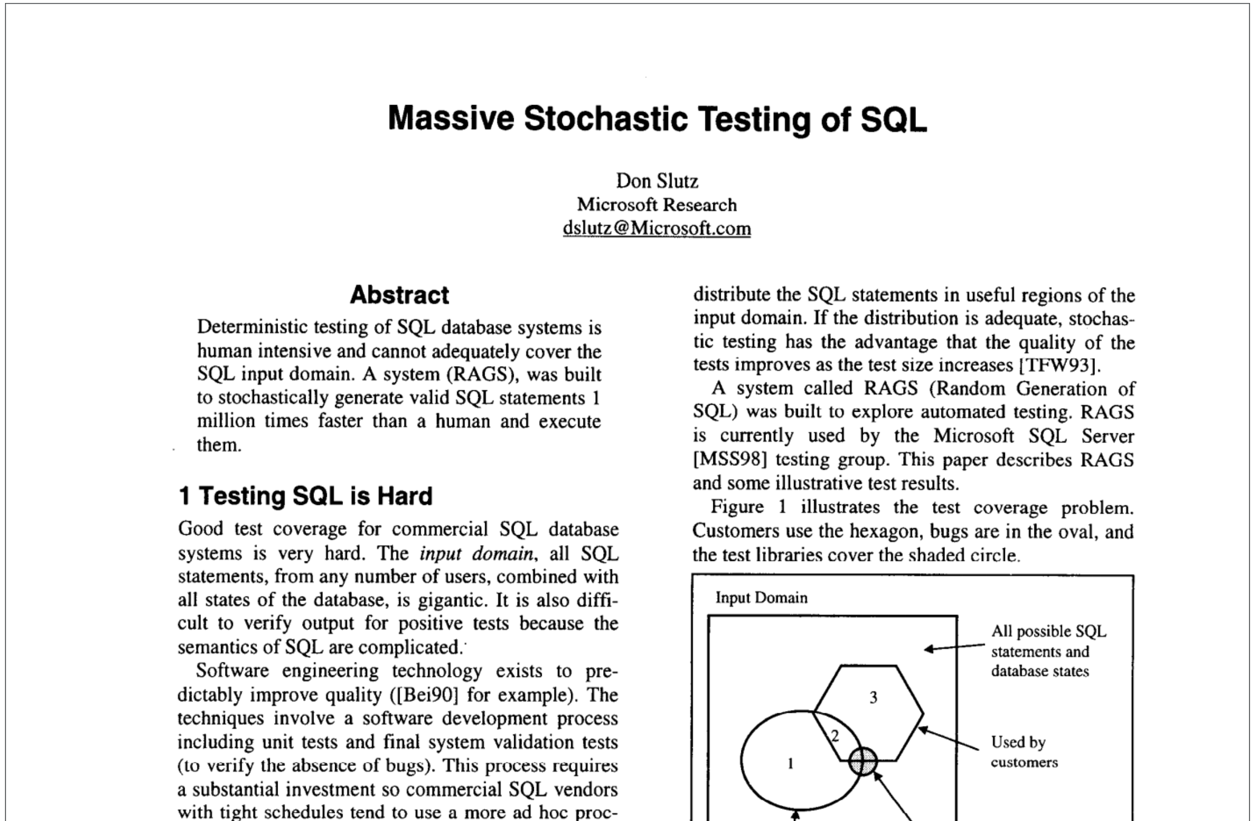
Problem: Differential Testing



Problem: The common SQL core is **small**

Differential Testing: RAGS (Slutz 1998)

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



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].

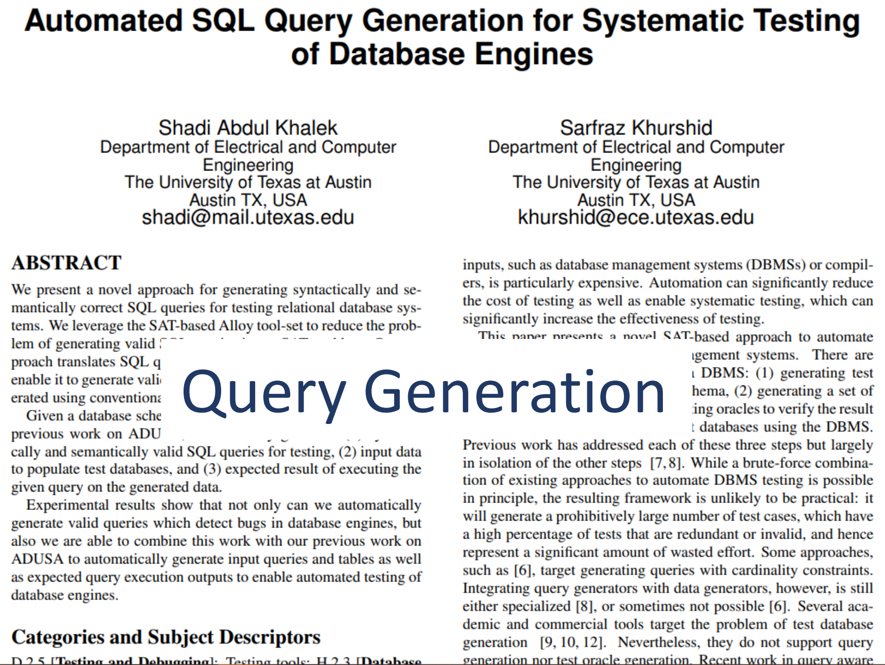
A system called RAGS (Random Generation of SQL) was built to explore automated testing. RAGS is currently used by the Microsoft SQL Server [MSS98] testing group. This paper describes RAGS and some illustrative test results.

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

Constraint Solving (Khalek et al. 2010)

Idea: Use a **solver** to generate queries, generate data, and provide a test oracle

Could reproduce already reported bugs, injected bugs, but **only one (potentially) new bug**



Query Generation

Automated SQL Query Generation for Systematic Testing of Database Engines

Shadi Abdul Khalek
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin TX, USA
shadi@mail.utexas.edu

Sarfraz Khurshid
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin TX, USA
khurshid@ece.utexas.edu

ABSTRACT

We present a novel approach for generating syntactically and semantically correct SQL queries for testing relational database systems. We leverage the SAT-based Alloy tool-set to reduce the problem of generating valid
proach translates SQL q
enable it to generate vali
erated using conventiona
Given a database sche
previous work on ADU
cally and semantically valid SQL queries for testing, (2) input data to populate test databases, and (3) expected result of executing the given query on the generated data.

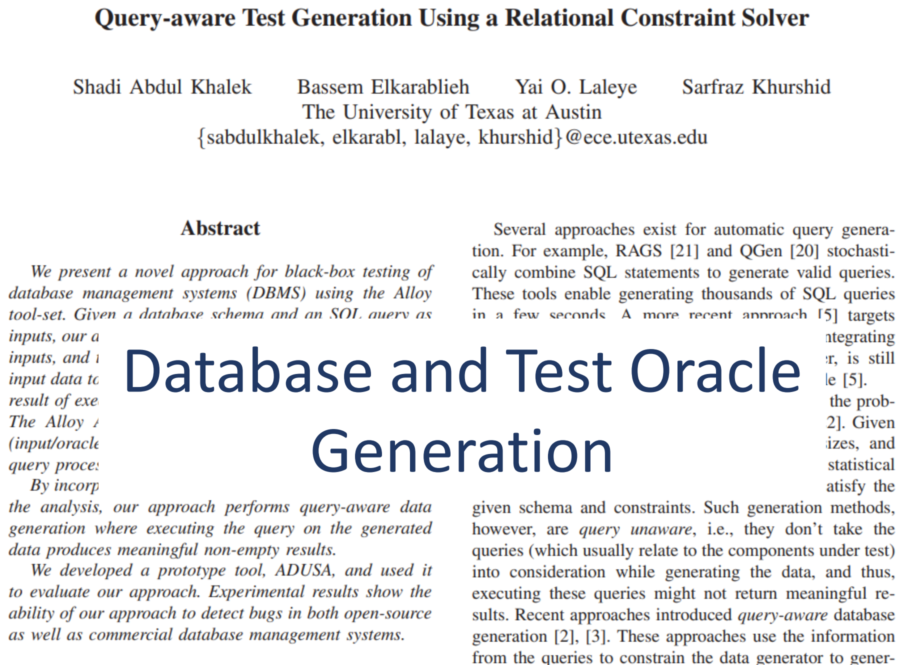
Experimental results show that not only can we automatically generate valid queries which detect bugs in database engines, but also we are able to combine this work with our previous work on ADUSA to automatically generate input queries and tables as well as expected query execution outputs to enable automated testing of database engines.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Testing tools; H.2.3 [**Database**

inputs, such as database management systems (DBMSs) or compilers, is particularly expensive. Automation can significantly reduce the cost of testing as well as enable systematic testing, which can significantly increase the effectiveness of testing.

This paper presents a novel SAT-based approach to automate
gement systems. There are
DBMS: (1) generating test
hema, (2) generating a set of
ing oracles to verify the result
databases using the DBMS.
Previous work has addressed each of these three steps but largely in isolation of the other steps [7, 8]. While a brute-force combination of existing approaches to automate DBMS testing is possible in principle, the resulting framework is unlikely to be practical: it will generate a prohibitively large number of test cases, which have a high percentage of tests that are redundant or invalid, and hence represent a significant amount of wasted effort. Some approaches, such as [6], target generating queries with cardinality constraints. Integrating query generators with data generators, however, is still either specialized [8], or sometimes not possible [6]. Several academic and commercial tools target the problem of test database generation [9, 10, 12]. Nevertheless, they do not support query generation nor test oracle generation. Recent work in query aware



Database and Test Oracle Generation

Query-aware Test Generation Using a Relational Constraint Solver

Shadi Abdul Khalek Bassem Elkarablieh Yai O. Laleye Sarfraz Khurshid
The University of Texas at Austin
{sabdulkhalek, elkarabl, lalaye, khurshid}@ece.utexas.edu

Abstract

We present a novel approach for black-box testing of database management systems (DBMS) using the Alloy tool-set. Given a database schema and an SQL query as
inputs, our a
inputs, and t
input data to
result of exe
The Alloy A
(input/oracle
query proces
By incorp
the analysis, our approach performs query-aware data generation where executing the query on the generated data produces meaningful non-empty results.

We developed a prototype tool, ADUSA, and used it to evaluate our approach. Experimental results show the ability of our approach to detect bugs in both open-source as well as commercial database management systems.

Several approaches exist for automatic query generation. For example, RAGS [21] and QGen [20] stochastically combine SQL statements to generate valid queries. These tools enable generating thousands of SQL queries in a few seconds. A more recent approach [5] targets
ntegrating
r, is still
le [5].
the prob-
2]. Given
izes, and
statistical
atisfy the
given schema and constraints. Such generation methods, however, are *query unaware*, i.e., they don't take the queries (which usually relate to the components under test) into consideration while generating the data, and thus, executing these queries might not return meaningful results. Recent approaches introduced *query-aware* database generation [2], [3]. These approaches use the information from the queries to constrain the data generator to gener-

Challenges

- DBMS are **tested well**
- Fuzzers are **ineffective** in **finding logic bugs**
- Knowing the **precise result set** for a query is **difficult**

The problem **of automatically testing DBMS has not** yet been **well addressed**

Approach: Pivoted Query Synthesis

Pivoted Query Synthesis is an **automatic testing approach** that can be used to **effectively test DBMS**

Pivoted Query Synthesis (PQS)

>100 bugs in widely used DBMS

Idea: PQS

Idea: Construct an automatic testing approach **considering only a single row**

Column$_0$	Column$_1$	Column$_2$
…	…	…
Value$_{i,0}$	Value$_{i,1}$	Value$_{i,2}$
…	…	…

Pivot Row

Intuition

- **Simpler** conceptually and implementation-wise

Column$_0$	Column$_1$	Column$_2$
…	…	…
Value$_{i,0}$	Value$_{i,1}$	Value$_{i,2}$
…	…	…

```
SELECT * FROM <table>
WHERE <cond>
```

<cond>?

Intuition

- **Simpler** conceptually and implementation-wise
- **Same effectiveness** as checking all rows

Column$_0$	Column$_1$	Column$_2$
…	…	…
Value$_{i,0}$	Value$_{i,1}$	Value$_{i,2}$
…	…	…

✓

```
SELECT * FROM <table>
WHERE <cond>
```

Intuition

- **Simpler** conceptually and implementation-wise
- **Same effectiveness** as checking all rows

Column_0	Column_1	Column_2
…	…	…
$\text{Value}_{i,0}$	$\text{Value}_{i,1}$	$\text{Value}_{i,2}$
…	…	…

```
SELECT * FROM <table>
WHERE <cond>
```

Intuition

- **Simpler** conceptually and implementation-wise
- **Same effectiveness** as checking all rows

Intuition

- **Simpler** conceptually and implementation-wise
- **Same effectiveness** as checking all rows
- **Precise** oracle for a single row

Approach

Database Generation

Randomly Generate Database

animal	description	picture
Cat	A cute toast cat	
Dog	Cute dog pic	
Cat	Cat plants (cute!)	

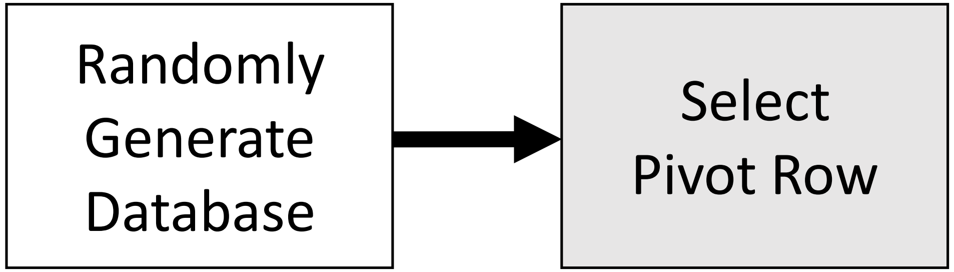
Database Generation

Randomly Generate Database

To explore “**all possible database states**” we randomly create databases

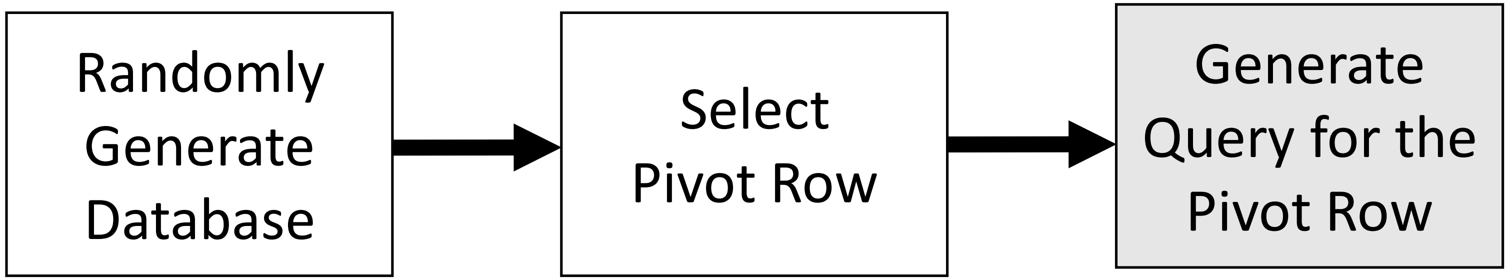
animal	description	picture
Cat	A cute toast cat	
Dog	Cute dog pic	
Cat	Cat plants (cute!)	

Pivot Row Selection



animal	description	picture
Cat	A cute toast cat	
Dog	Cute dog pic	
Cat	Cat plants (cute!)	

Query Generation



Randomly Generate Database

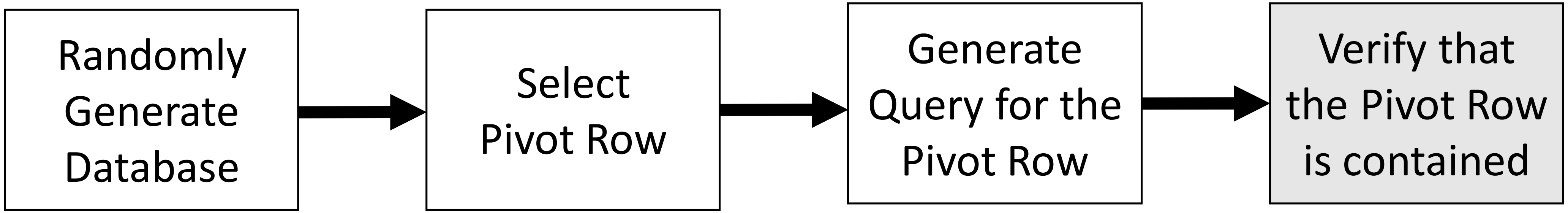
Select Pivot Row

Generate Query for the Pivot Row

```
SELECT picture, description
FROM animal_pictures
WHERE animal = 'Cat'
   AND description LIKE '%cute%'
```

animal	description	picture
Cat	Cat plants (cute!)	

Verifying the Result



Randomly Generate Database → Select Pivot Row → Generate Query for the Pivot Row → Verify that the Pivot Row is contained

```
SELECT picture, description
FROM animal_pictures
WHERE animal = 'Cat'
   AND description LIKE '%cute%'
```

DBMS

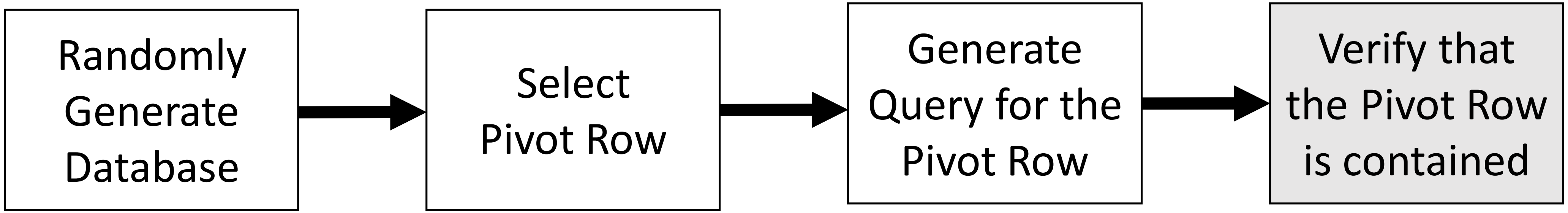
result set

animal	description	picture
Cat	A cute toast cat	
Cat	Cat plants (cute!)	

pivot row

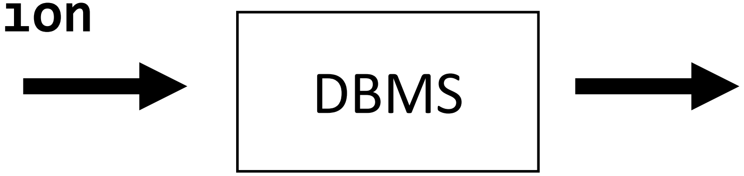
animal	description	picture
Cat	Cat plants (cute!)	

Verifying the Result



Randomly Generate Database → Select Pivot Row → Generate Query for the Pivot Row → Verify that the Pivot Row is contained

```
SELECT picture, description
FROM animal_pictures
WHERE animal = 'Cat'
   AND description LIKE '%cute%'
```



DBMS

result set

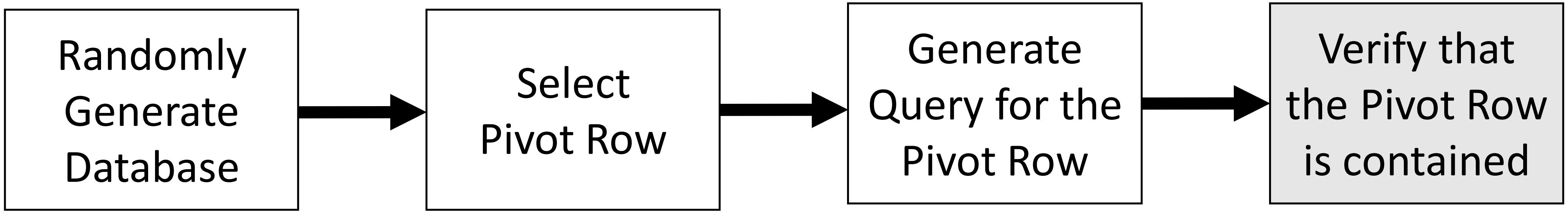
animal	description	picture
Cat	A cute toast cat	
Cat	Cat plants (cute!)	

pivot row

animal	description	picture
Cat	Cat plants (cute!)	

pivot row ∈ result set ✓

Verifying the Result


Randomly Generate Database
Select Pivot Row
Generate Query for the Pivot Row
Verify that the Pivot Row is contained

```
SELECT picture, description
FROM animal_pictures
WHERE animal = 'Cat'
   AND description LIKE '%cute%'
```

DBMS

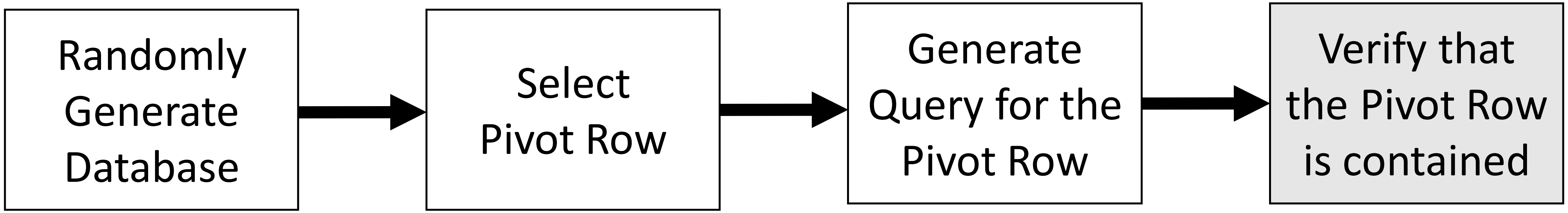
result set

animal	description	picture
Cat	A cute toast cat	
Dog	Cute dog pic	

pivot row

animal	description	picture
Cat	Cat plants (cute!)	

Verifying the Result


Randomly Generate Database → Select Pivot Row → Generate Query for the Pivot Row → Verify that the Pivot Row is contained

```
SELECT picture, description
FROM animal_pictures
WHERE animal = 'Cat'
   AND description LIKE '%cute%'
```

→ DBMS →

result set

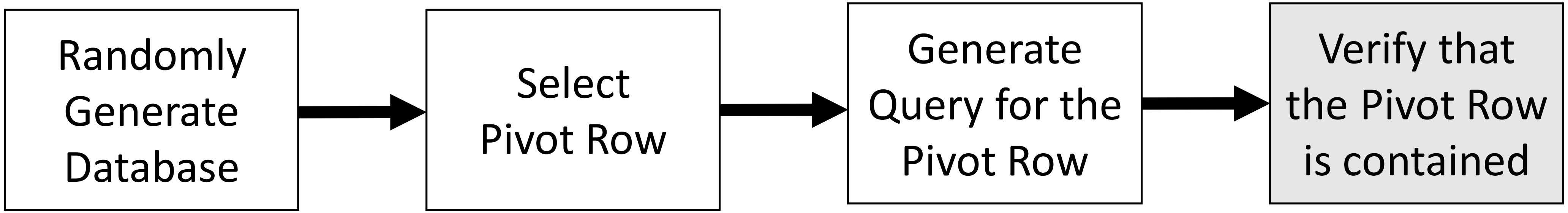
animal	description	picture
Cat	A cute toast cat	
Dog	Cute dog pic	

pivot row

animal	description	picture
Cat	Cat plants (cute!)	

pivot row ∉ result set ✗

Verifying the Result



Randomly Generate Database

Select Pivot Row

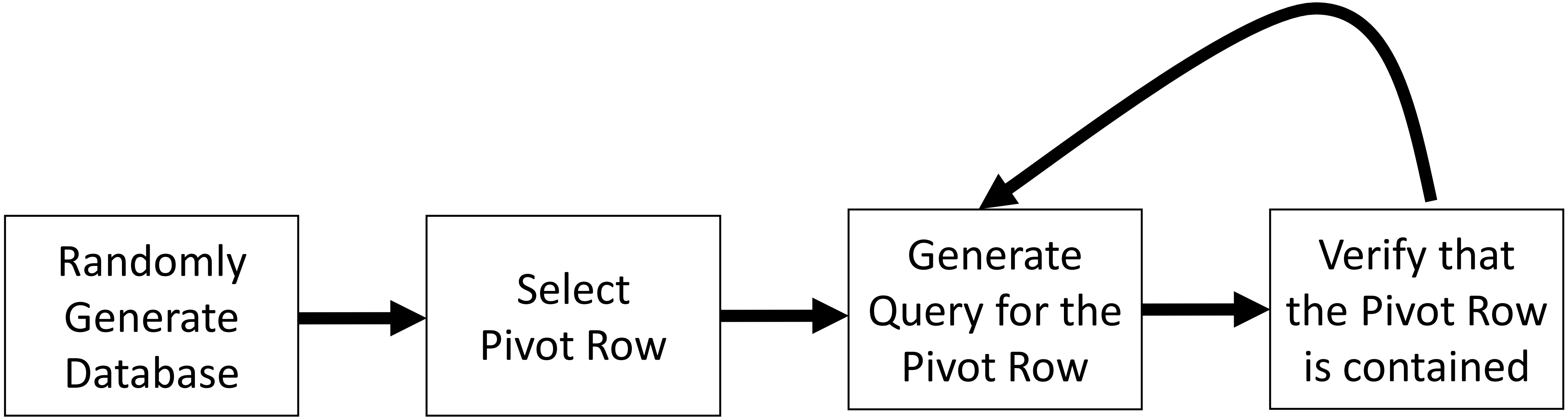
Generate Query for the Pivot Row

Verify that the Pivot Row is contained

pivot row ∉ result set ✘

The “**containment oracle**” is PQS’ primary oracle

Approach

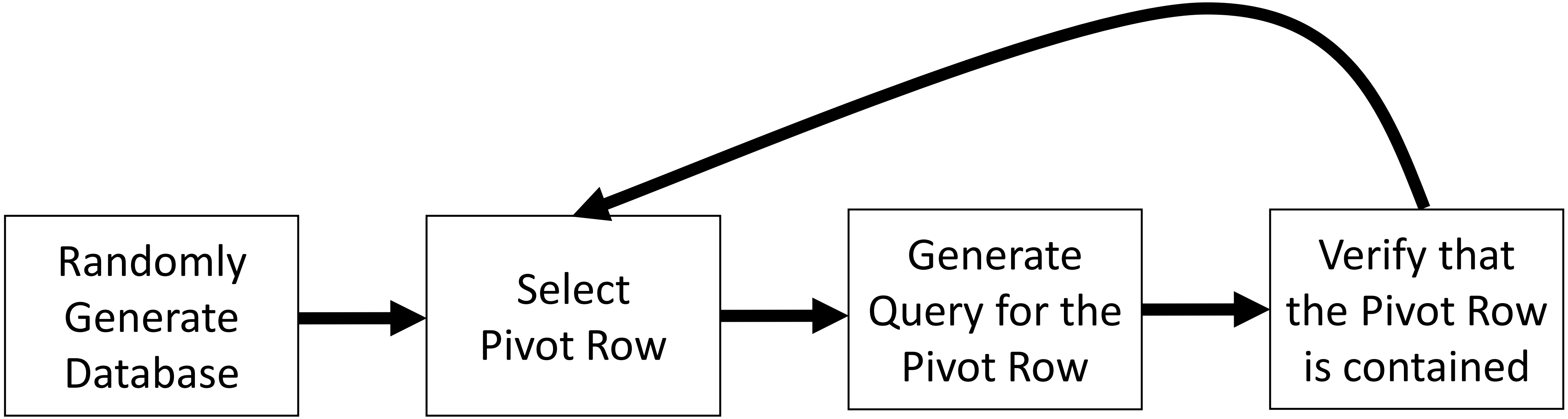

Randomly Generate Database

Select Pivot Row

Generate Query for the Pivot Row

Verify that the Pivot Row is contained

Approach



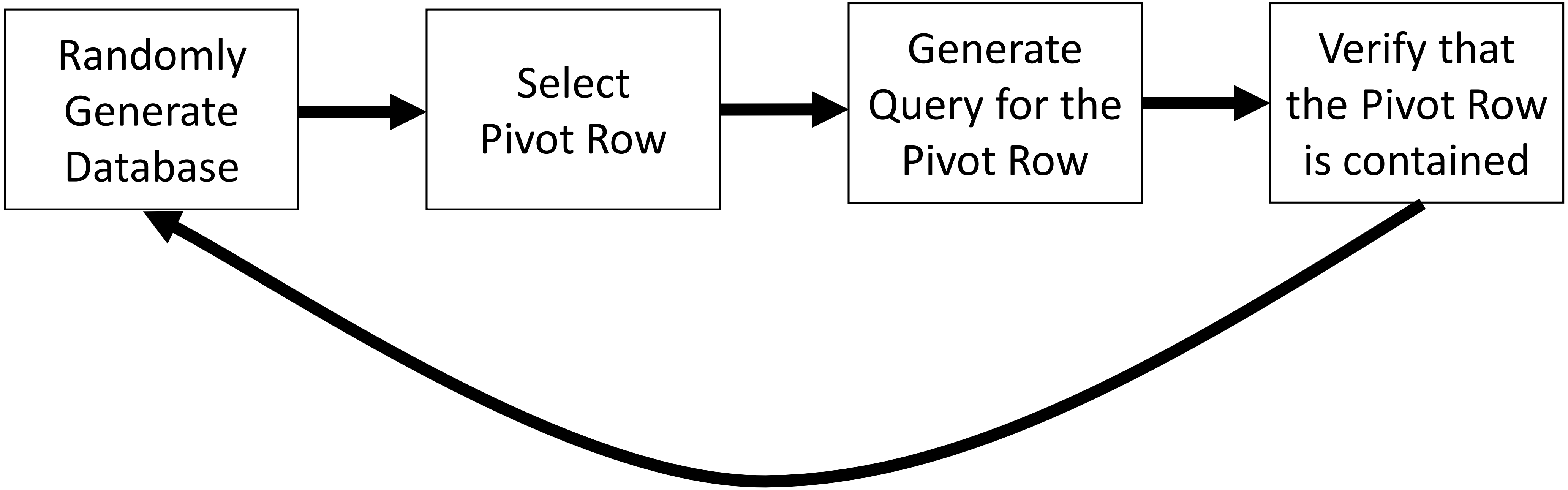
Randomly Generate Database

Select Pivot Row

Generate Query for the Pivot Row

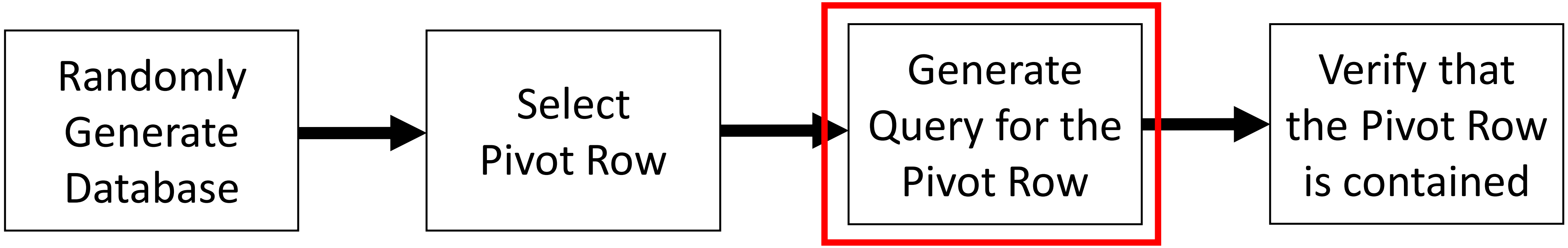
Verify that the Pivot Row is contained

Approach


Randomly Generate Database
Select Pivot Row
Generate Query for the Pivot Row
Verify that the Pivot Row is contained

Approach

How do we generate this query?



Randomly Generate Database

Select Pivot Row

Generate Query for the Pivot Row

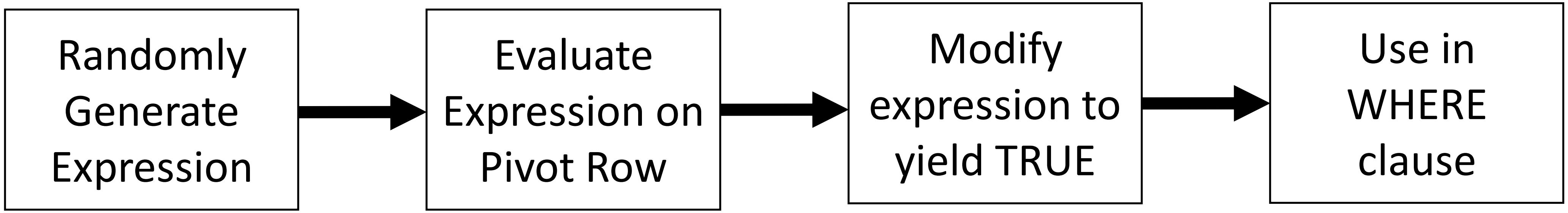
Verify that the Pivot Row is contained

How do we Generate Queries?

```
SELECT picture, description
FROM animal_pictures
WHERE
```

Generate an **expression** that **yields TRUE** for the pivot row

How do we Generate Queries?



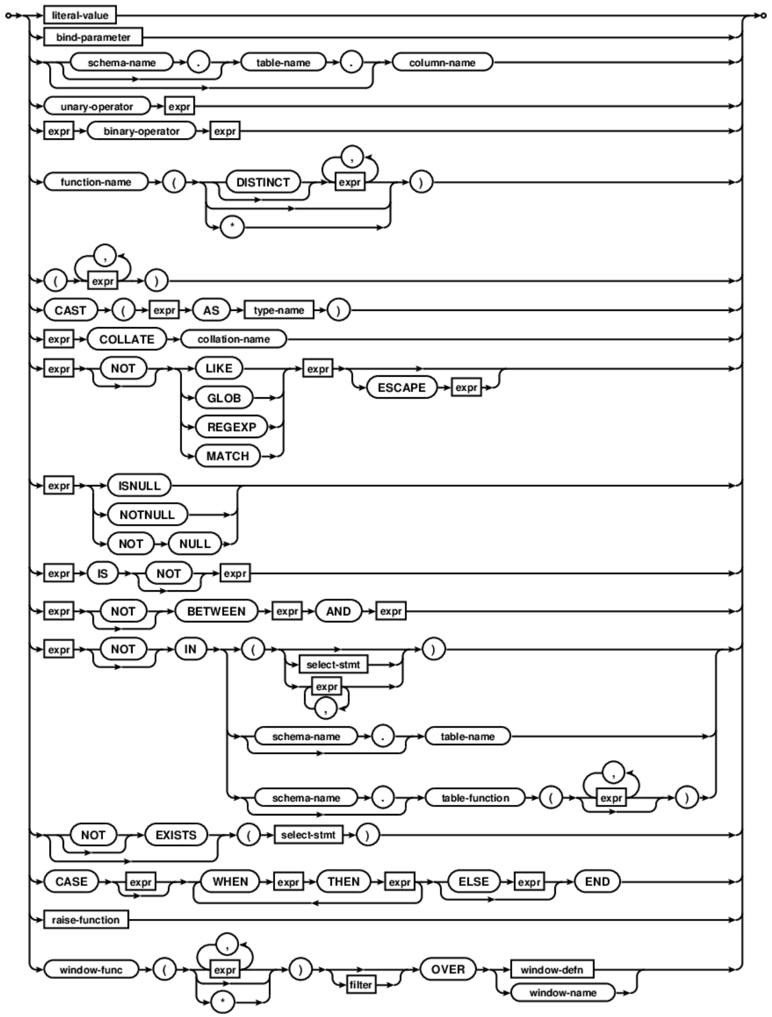
Randomly Generate Expression

Evaluate Expression on Pivot Row

Modify expression to yield TRUE

Use in WHERE clause

Random Expression Generation



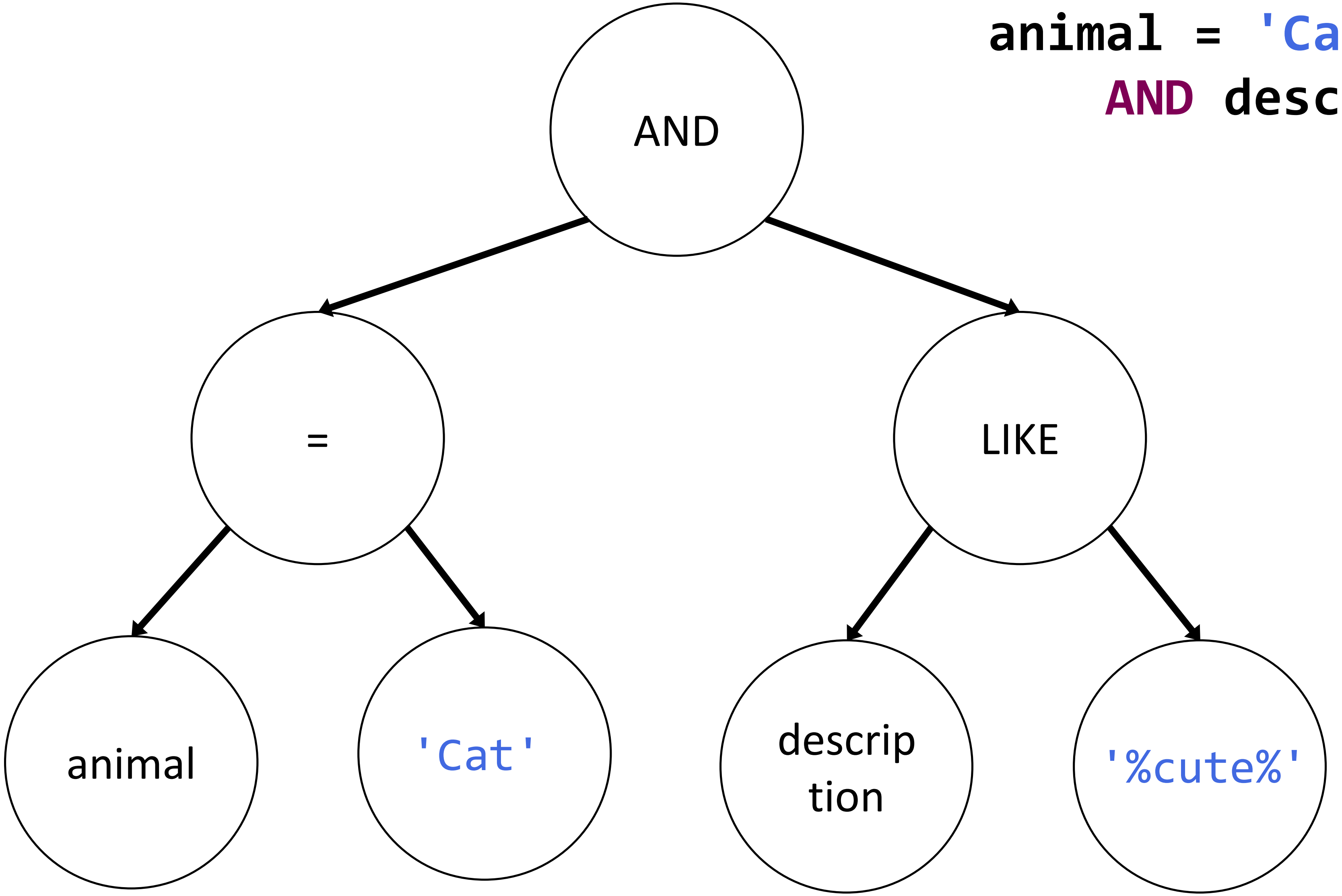
https://www.sqlite.org/syntax/expr.html

animal_pictures

animal	description	picture

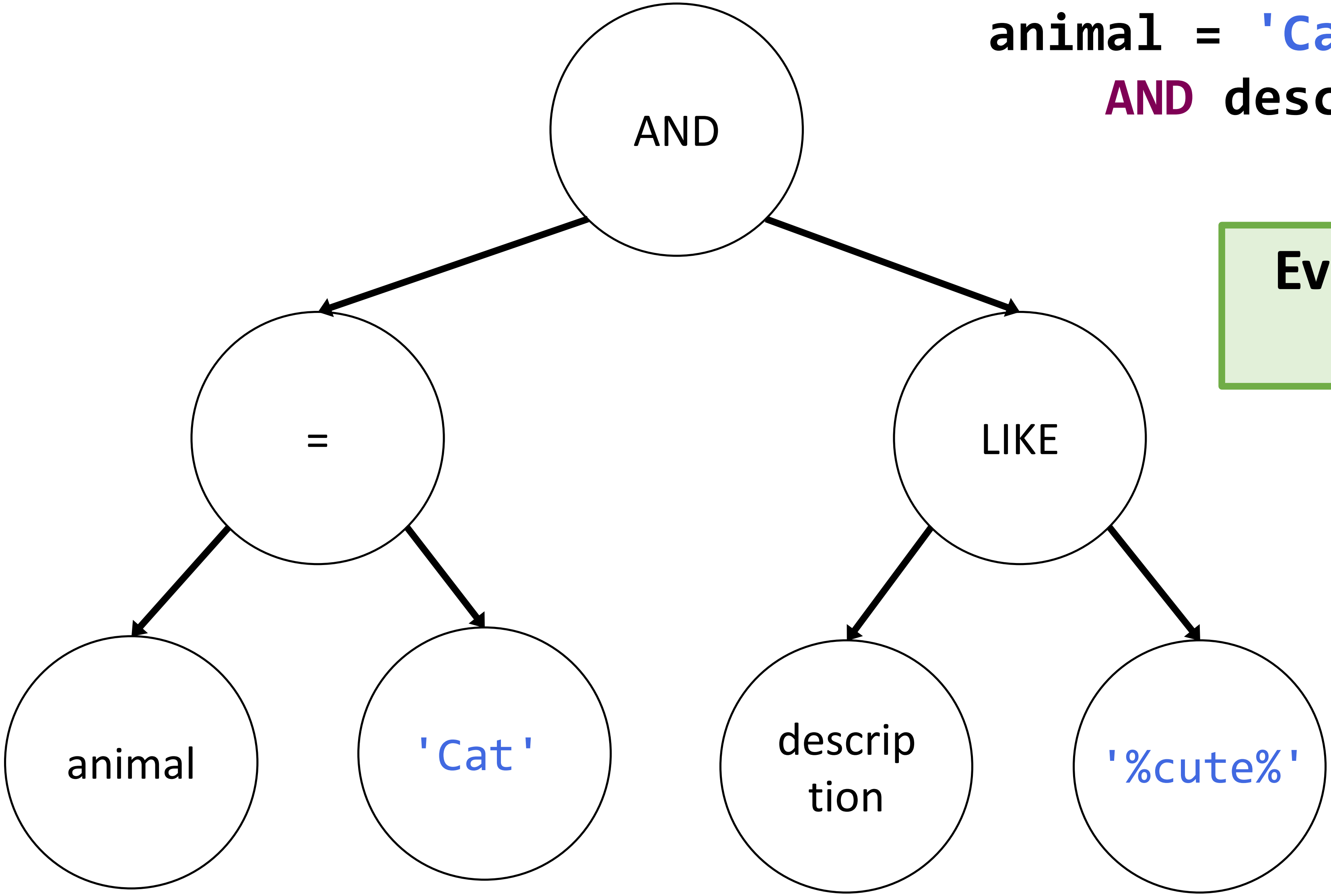
We **first** generate a **random expression**

Random Expression Generation



```
animal = 'Cat'
    AND description LIKE '%cute%'
```

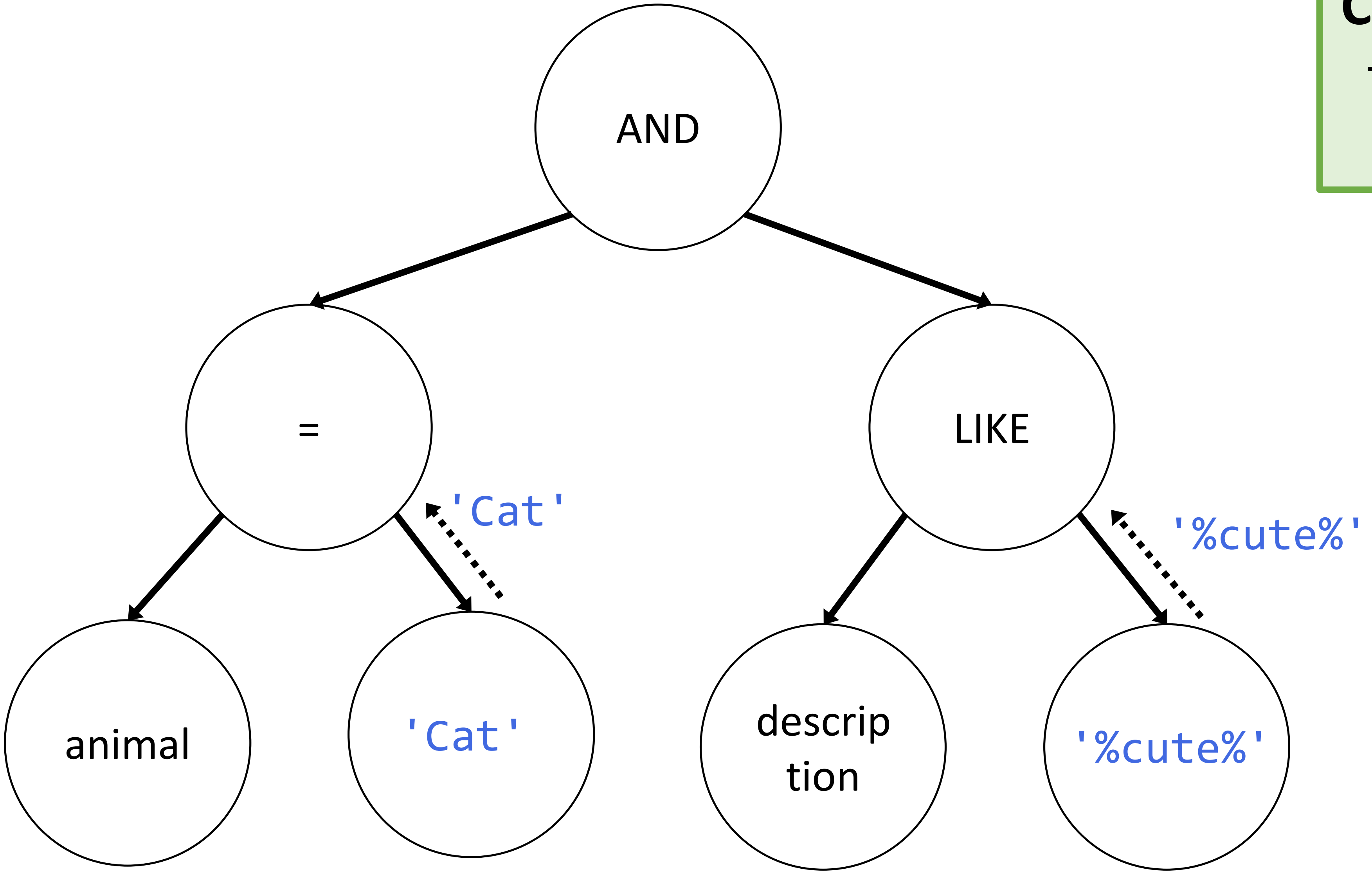
Random Expression Generation



```
animal = 'Cat'
    AND description LIKE '%cute%'
```

Evaluate the tree based on the pivot row

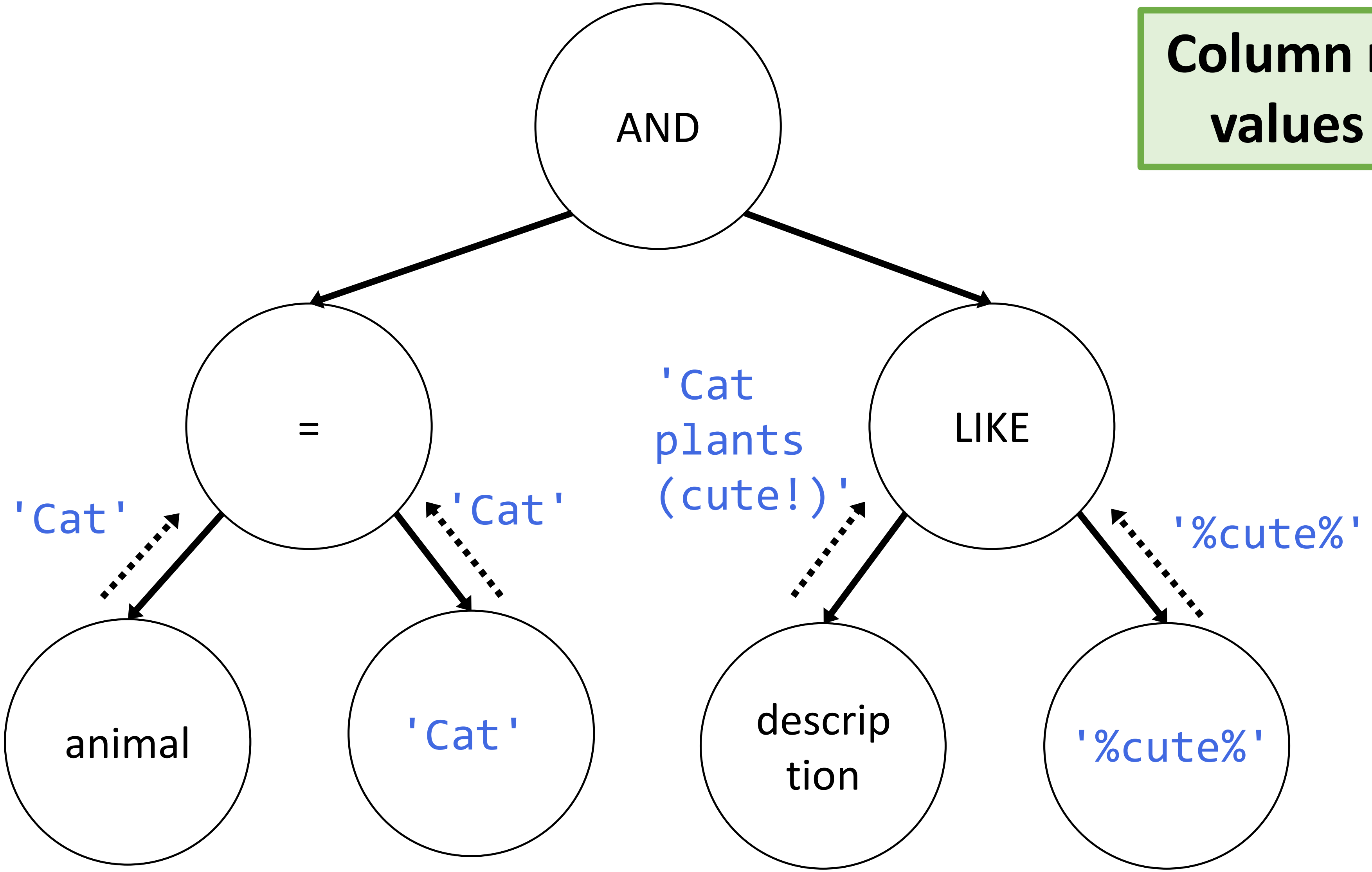
Random Expression Evaluation



Constant nodes return their assigned **literal values**

animal	description	picture
Cat	Cat plants (cute!)	

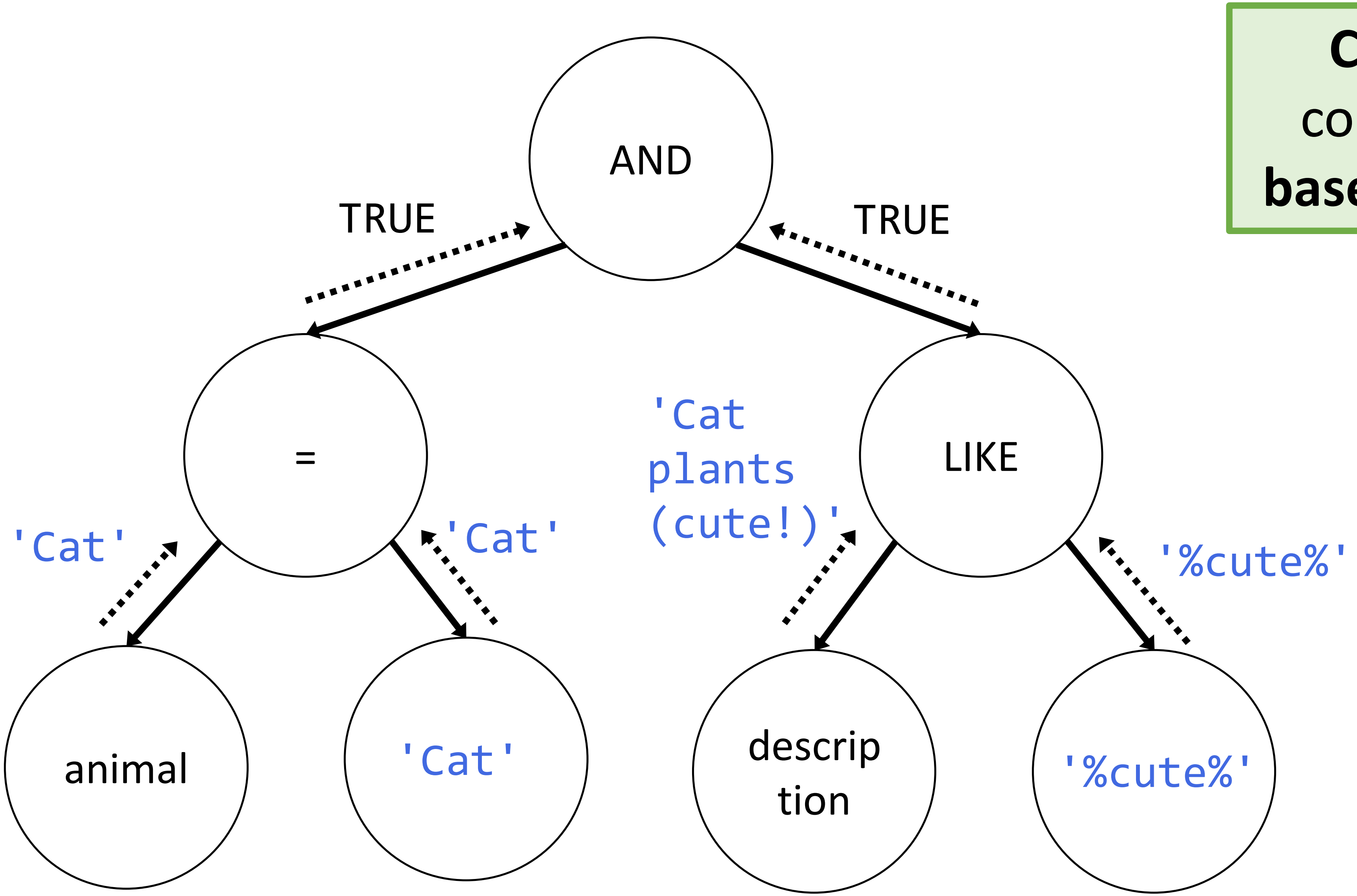
Random Expression Evaluation



Column references return the **values** from the **pivot row**

animal	description	picture
Cat	Cat plants (cute!)	

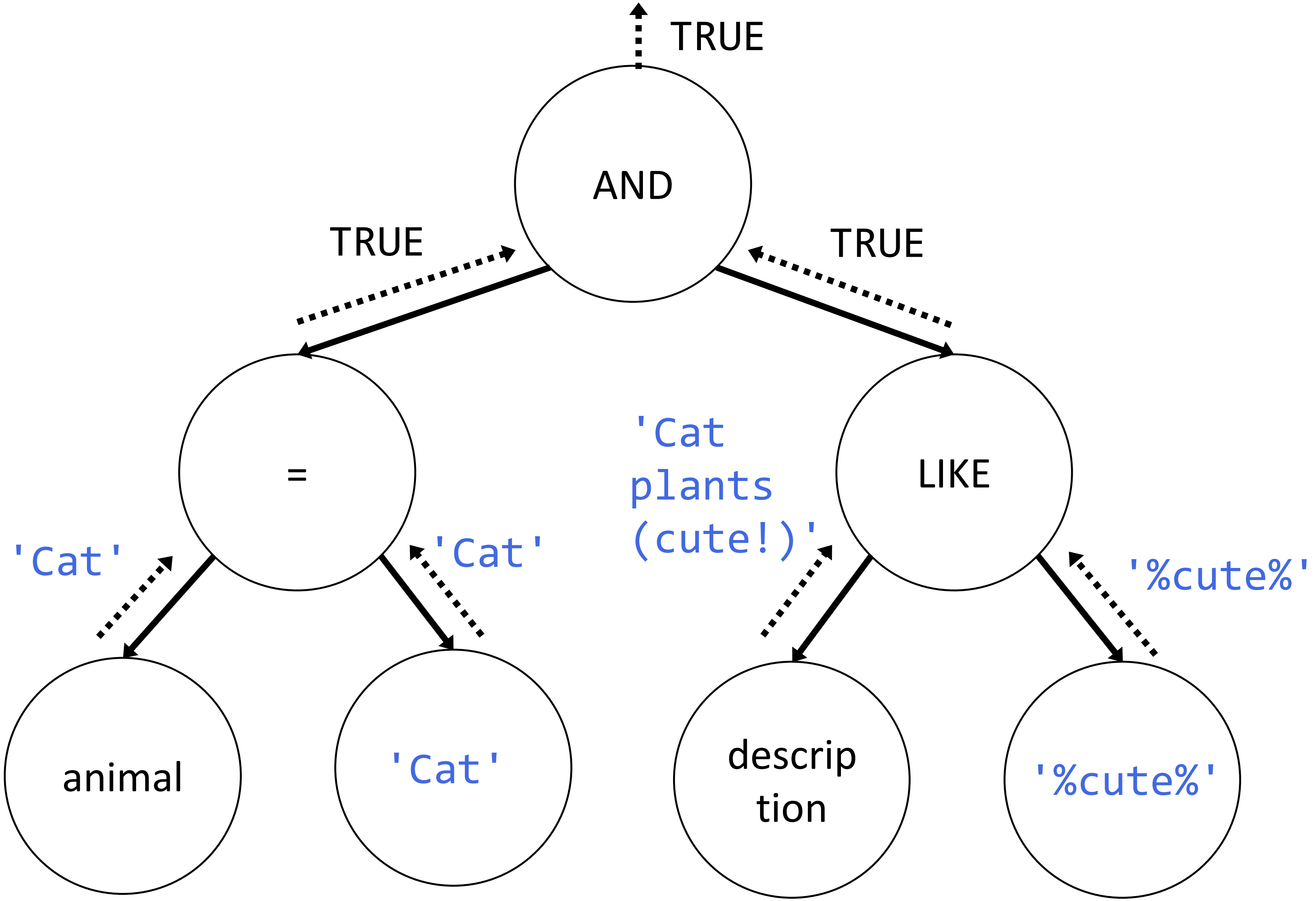
Random Expression Evaluation



Compound nodes compute their result **based on their children**

animal	description	picture
Cat	Cat plants (cute!)	

Random Expression Evaluation

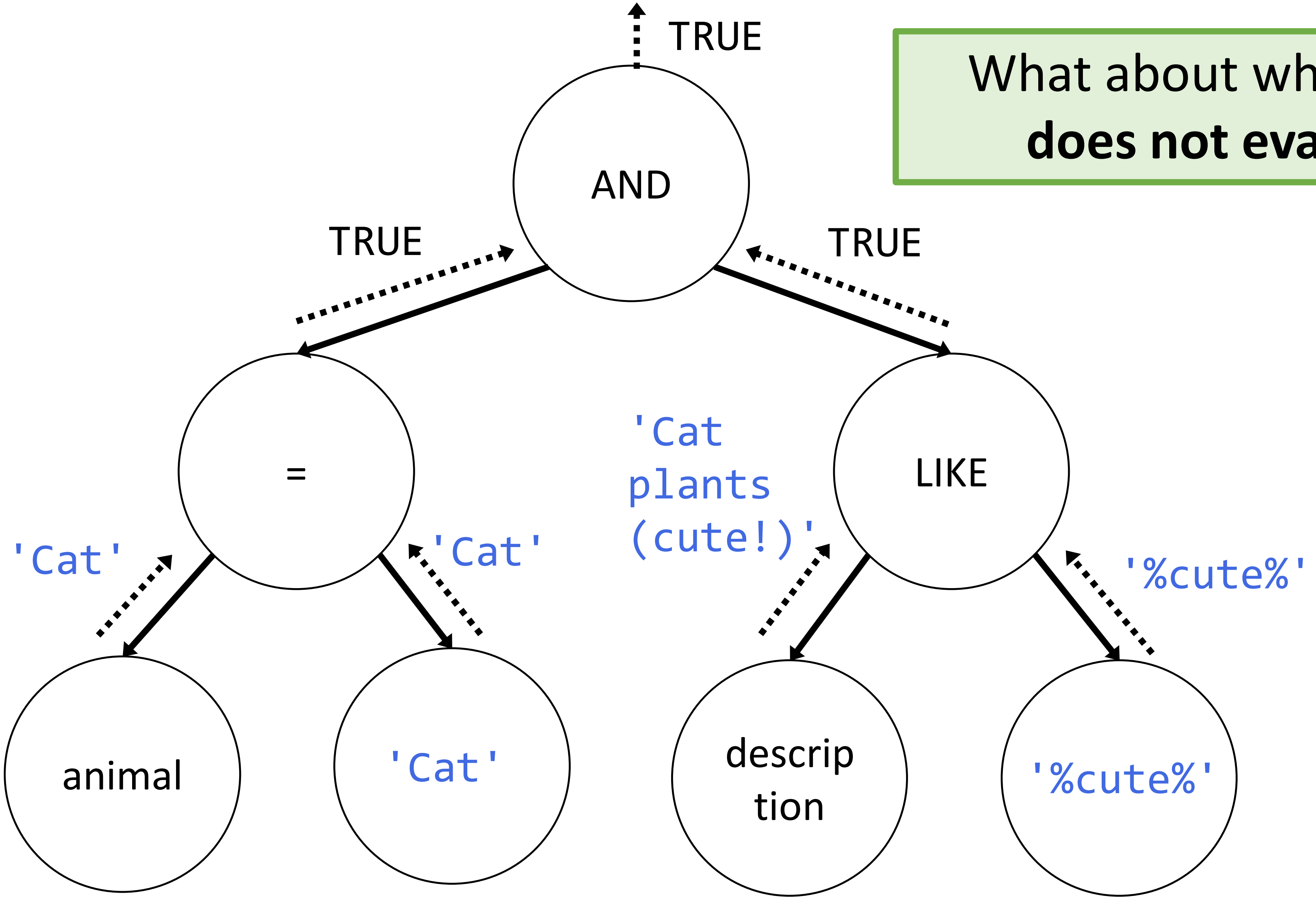


animal	description	picture
Cat	Cat plants (cute!)	

Query Synthesis

```sql
SELECT picture, description
FROM animal_pictures
WHERE animal = 'Cat' AND description LIKE '%cute%'
```

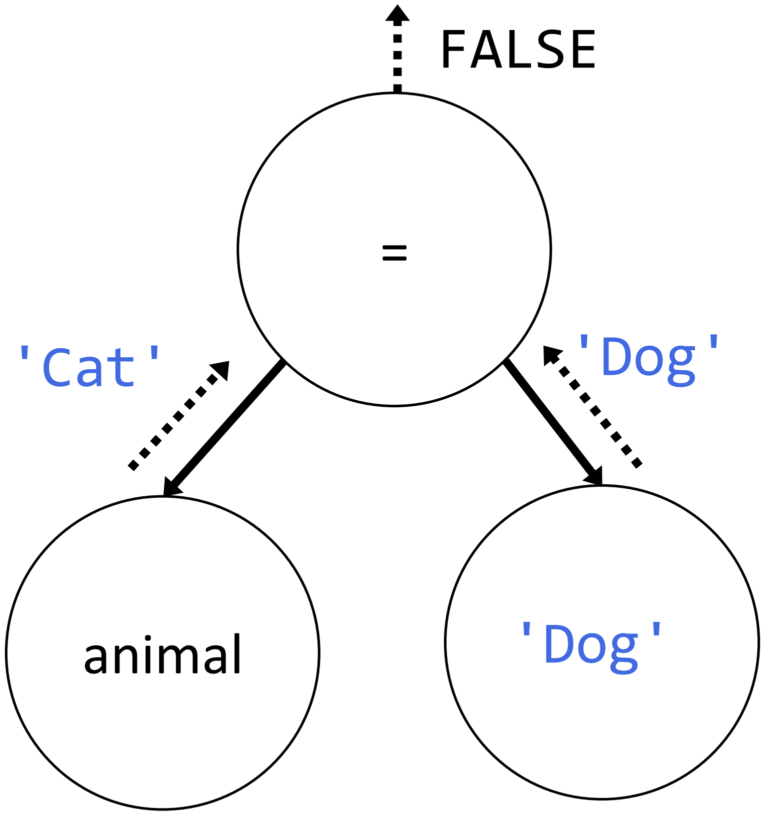
Random Expression Evaluation



What about when the expression **does not evaluate to TRUE**?

animal	description	picture
Cat	Cat plants (cute!)	

Random Expression Evaluation



What about when the expression **does not evaluate to TRUE**?

```
animal = 'Dog'
```

animal	description	picture
Cat	Cat plants (cute!)	

Random Expression Rectification

```
switch (result) {
    case TRUE:
        result = randexpr;
    case FALSE:
        result = NOT randexpr;
    case NULL:
        result = randexpr ISNULL;
}
```

Random Expression Rectification

```
switch (result) {
    case TRUE:
        result = randexpr;
    case FALSE:
        result = NOT randexpr;
    case NULL:
        result = randexpr ISNULL;
}
```

FALSE

`animal = 'Dog'`

animal	description	picture
Cat	Cat plants (cute!)	

Random Expression Rectification

```
switch (result) {
    case TRUE:
        result = randexpr;
    case FALSE:
        result = NOT randexpr;
    case NULL:
        result = randexpr ISNULL;
}
```

TRUE

`NOT(animal = 'Dog')`

animal	description	picture
Cat	Cat plants (cute!)	

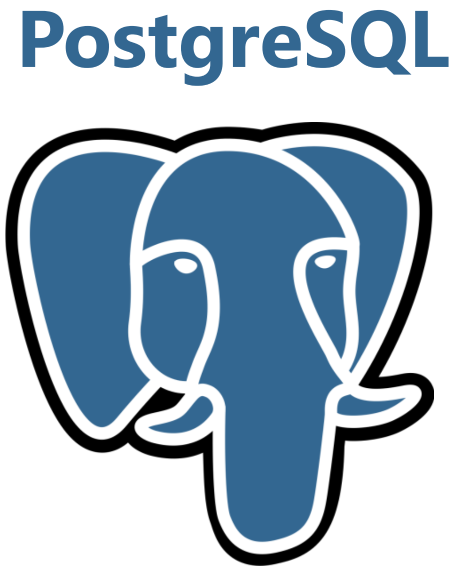
How do we Generate Queries?

```
SELECT picture, description
FROM animal_pictures
WHERE NOT(animal = 'Dog')
```

animal	description	picture
Cat	Cat plants (cute!)	

Evaluation

Tested DBMS



Tested DBMS

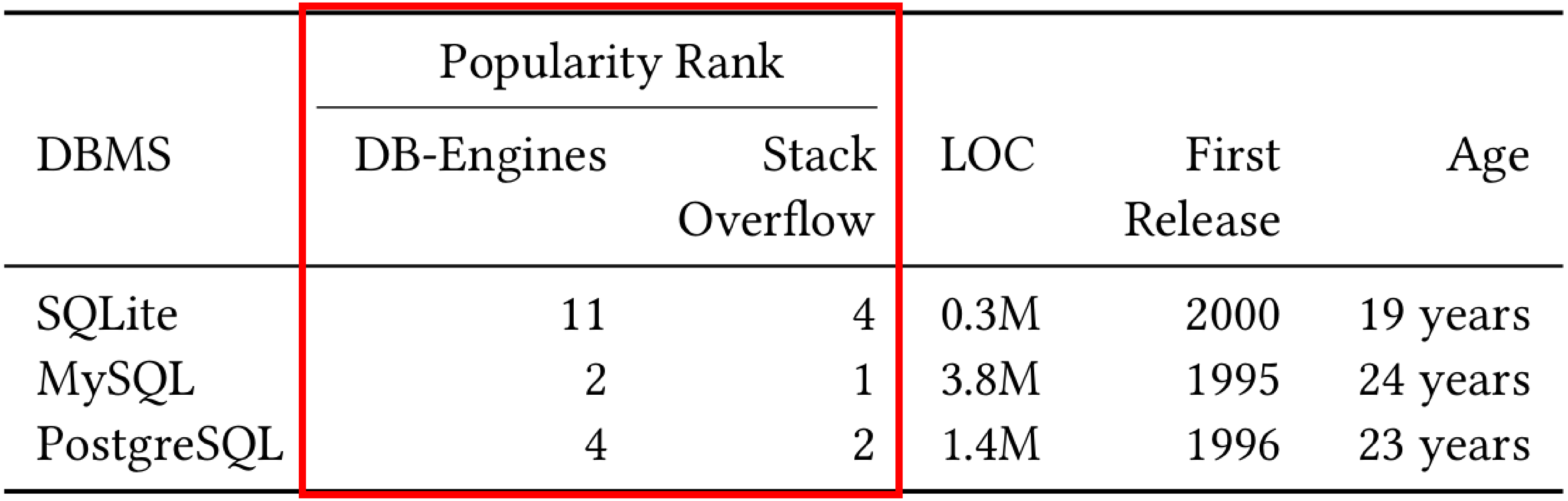
PostgreSQL

SQLite

MySQL™

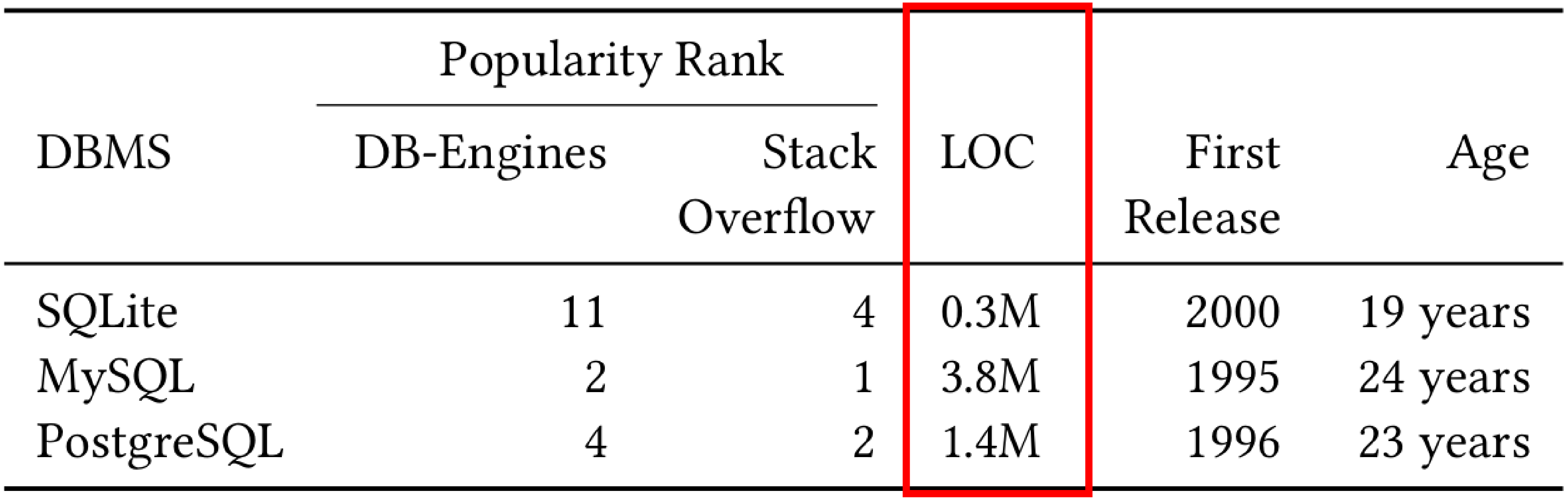
We tested these (and other DBMS) in a period of 3-4 months

DBMS



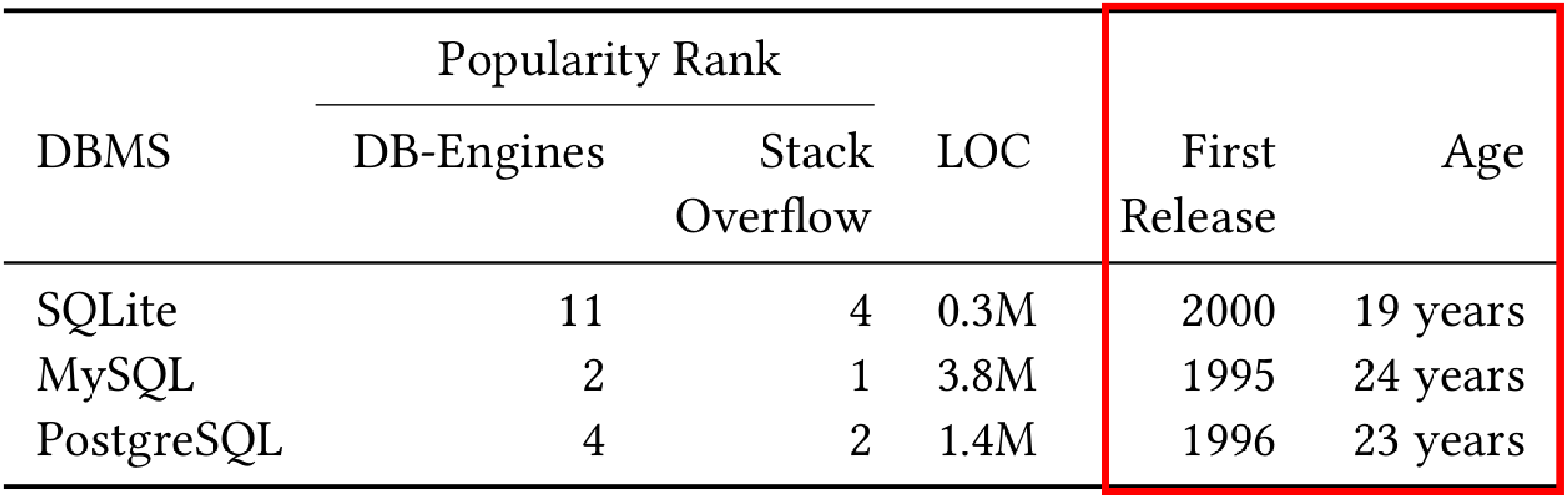
DBMS	Popularity Rank		LOC	First Release	Age
	DB-Engines	Stack Overflow			
SQLite	11	4	0.3M	2000	19 years
MySQL	2	1	3.8M	1995	24 years
PostgreSQL	4	2	1.4M	1996	23 years

DBMS



DBMS	Popularity Rank DB-Engines	 Stack Overflow	LOC	First Release	Age
SQLite	11	4	0.3M	2000	19 years
MySQL	2	1	3.8M	1995	24 years
PostgreSQL	4	2	1.4M	1996	23 years

DBMS



	Popularity Rank				
DBMS	DB-Engines	Stack Overflow	LOC	First Release	Age
SQLite	11	4	0.3M	2000	19 years
MySQL	2	1	3.8M	1995	24 years
PostgreSQL	4	2	1.4M	1996	23 years

Bugs Overview

Real Bugs

DBMS	Fixed	Verified
SQLite	65	0
MySQL	15	10
PostgreSQL	5	4
Sum	85	14

Bugs Overview

Real Bugs

DBMS	Fixed	Verified
SQLite	65	0
MySQL	15	10
PostgreSQL	5	4
Sum	85	14

99 real bugs: addressed by code or documentation fixes, or verified as bugs

Bugs Overview

Real Bugs

DBMS	Fixed	Verified
SQLite	65	0
MySQL	15	10
PostgreSQL	5	4
Sum	85	14

The SQLite developers **quickly responded** to all our bug reports → we focused on this DBMS

Bugs Overview

Real Bugs

DBMS	Fixed	Verified
SQLite	65	0
MySQL	15	10
PostgreSQL	5	4
Sum	85	14

All MySQL bug reports were **verified quickly**

Bugs Overview

Real Bugs

DBMS	Fixed	Verified
SQLite	65	0
MySQL	15	10
PostgreSQL	5	4
Sum	85	14

MySQL's trunk is **not available**, and it has a long release cycle

Bugs Overview

Real Bugs

DBMS	Fixed	Verified
SQLite	65	0
MySQL	15	10
PostgreSQL	5	4
Sum	85	14

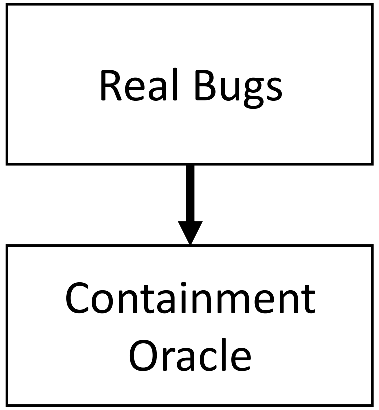
We found the **fewest bugs in PostgreSQL** and not all could be easily addressed

Oracles

Real Bugs

DBMS	Containment	Error	SEGFAULT
SQLite	46	17	2
MySQL	14	10	1
PostgreSQL	1	7	1
Sum	61	34	4

Oracles



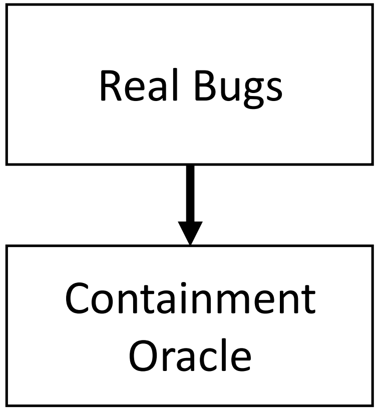
Real Bugs

Containment Oracle

DBMS	Containment	Error	SEGFAULT
SQLite	46	17	2
MySQL	14	10	1
PostgreSQL	1	7	1
Sum	61	34	4

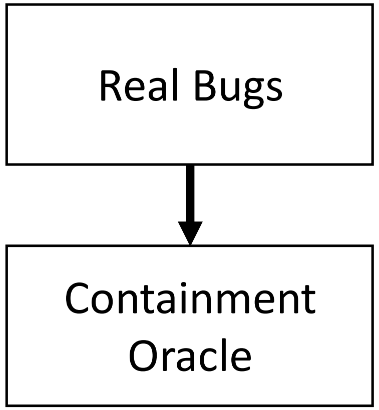
Our *Containment* oracle allowed us to detect **most errors**

Result: Bug in SQLite3



```
CREATE TABLE t0(c1 TEXT PRIMARY KEY) WITHOUT ROWID;
CREATE INDEX i0 ON t0(c1 COLLATE NOCASE);
INSERT INTO t0(c1) VALUES ('A');
INSERT INTO t0(c1) VALUES ('a');
```

Result: Bug in SQLite3



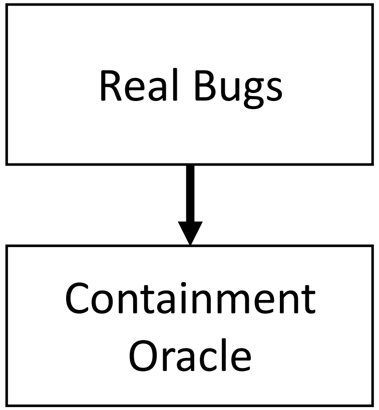
Real Bugs

Containment Oracle

```
CREATE TABLE t0(c1 TEXT PRIMARY KEY) WITHOUT ROWID;
CREATE INDEX i0 ON t0(c1 COLLATE NOCASE);
INSERT INTO t0(c1) VALUES ('A');
INSERT INTO t0(c1) VALUES ('a');
```

An index is an auxiliary data structure that **should not** affect the query’s result

Result: Bug in SQLite3



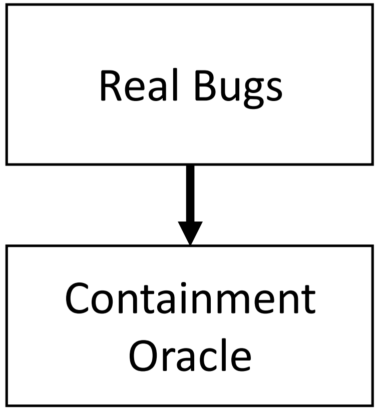
Real Bugs

Containment Oracle

```
CREATE TABLE t0(c1 TEXT PRIMARY KEY) WITHOUT ROWID;
CREATE INDEX i0 ON t0(c1 COLLATE NOCASE);
INSERT INTO t0(c1) VALUES ('A');
INSERT INTO t0(c1) VALUES ('a');
```

c1
'A'
'a'

Result: Bug in SQLite3



Real Bugs

Containment Oracle

```
CREATE TABLE t0(c1 TEXT PRIMARY KEY) WITHOUT ROWID;
CREATE INDEX i0 ON t0(c1 COLLATE NOCASE);
INSERT INTO t0(c1) VALUES ('A');
INSERT INTO t0(c1) VALUES ('a');
```

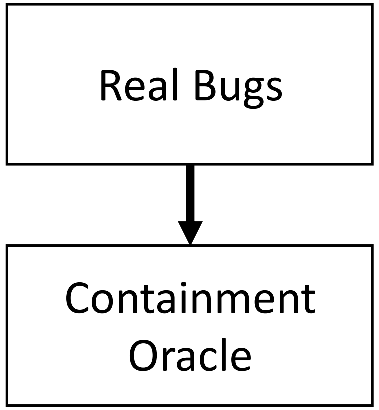
c1
'A'
'a'

```
SELECT * FROM t0;
```

→ SQLite →

c1
'A'

Result: Bug in SQLite3


Real Bugs

Containment Oracle

```
CREATE TABLE t0(c1 TEXT PRIMARY KEY) WITHOUT ROWID;
CREATE INDEX i0 ON t0(c1 COLLATE NOCASE);
INSERT INTO t0(c1) VALUES ('A');
INSERT INTO t0(c1) VALUES ('a');
```

c1
'A'
'a'

```
SELECT * FROM t0;
```

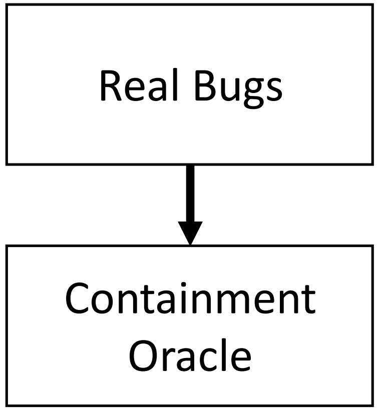
→ SQLite →

c1
'A'

✗

SQLite failed to fetch 'a'!

Result: Bug in PostgreSQL



t0

c0	c1

```
CREATE TABLE t0(c0 INT PRIMARY KEY, c1 INT);
CREATE TABLE t1(c0 INT) INHERITS (t0);
```

t1

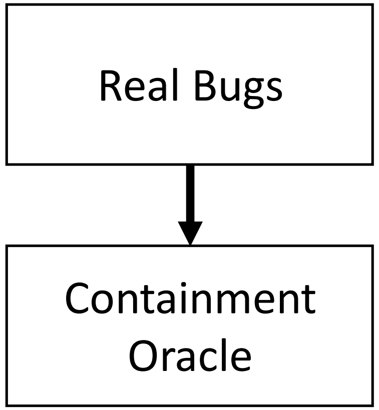
c0	c1

Result: Bug in PostgreSQL

t0

c0	c1
0	0

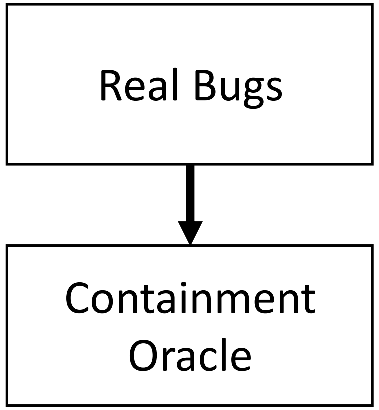
```
CREATE TABLE t0(c0 INT PRIMARY KEY, c1 INT);
CREATE TABLE t1(c0 INT) INHERITS (t0);
INSERT INTO t0(c0, c1) VALUES(0, 0);
```



t1

c0	c1

Result: Bug in PostgreSQL



t0

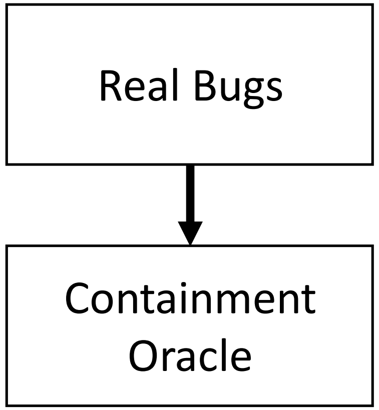
c0	c1
0	0
0	1

t1

c0	c1
0	1

```
CREATE TABLE t0(c0 INT PRIMARY KEY, c1 INT);
CREATE TABLE t1(c0 INT) INHERITS (t0);
INSERT INTO t0(c0, c1) VALUES(0, 0);
INSERT INTO t1(c0, c1) VALUES(0, 1);
```

Result: Bug in PostgreSQL



Real Bugs

Containment Oracle

t0

c0	c1
0	0
0	1

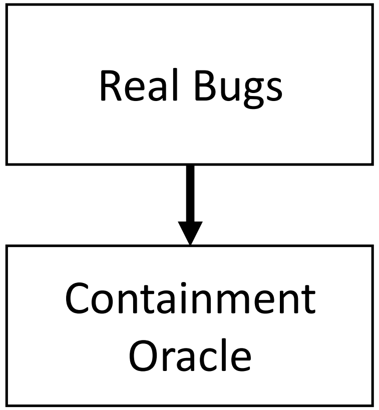
t1

c0	c1
0	1

```
CREATE TABLE t0(c0 INT PRIMARY KEY, c1 INT);
CREATE TABLE t1(c0 INT) INHERITS (t0);
INSERT INTO t0(c0, c1) VALUES(0, 0);
INSERT INTO t1(c0, c1) VALUES(0, 1);
```

The inheritance relationship causes the row to be **inserted both in t0 and t1**

Result: Bug in PostgreSQL



Real Bugs

Containment Oracle

t0

c0	c1
0	0
0	1

```
CREATE TABLE t0(c0 INT PRIMARY KEY, c1 INT);
CREATE TABLE t1(c0 INT) INHERITS (t0);
INSERT INTO t0(c0, c1) VALUES(0, 0);
INSERT INTO t1(c0, c1) VALUES(0, 1);
```

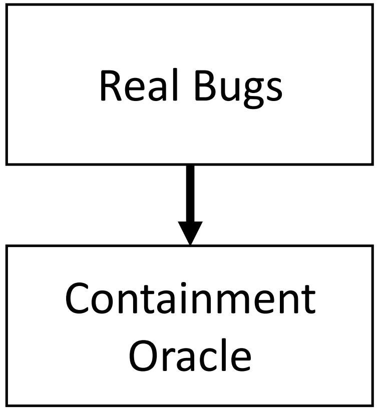
t1

c0	c1
0	1

```
SELECT c0, c1 FROM t0
GROUP BY c0, c1;
```

c0	c1
0	0

Result: Bug in PostgreSQL


Real Bugs

Containment Oracle

t0

c0	c1
0	0
0	1

```
CREATE TABLE t0(c0 INT PRIMARY KEY, c1 INT);
CREATE TABLE t1(c0 INT) INHERITS (t0);
INSERT INTO t0(c0, c1) VALUES(0, 0);
INSERT INTO t1(c0, c1) VALUES(0, 1);
```

t1

c0	c1
0	1

```
SELECT c0, c1 FROM t0
GROUP BY c0, c1;
```

c0	c1
0	0

✗

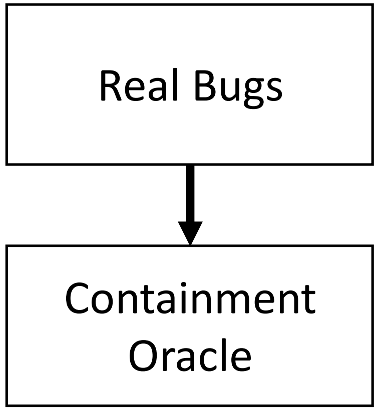
PostgreSQL failed to fetch the row 0 | 1

Result: Bug in MySQL

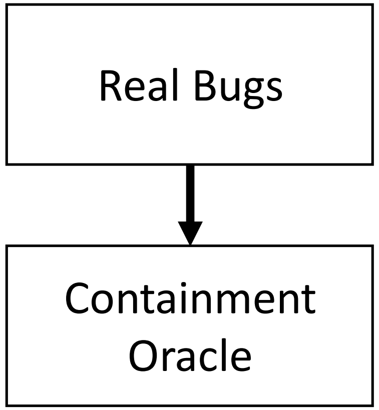
t0

c0
NULL

```
CREATE TABLE t0(c0 TINYINT);
INSERT INTO t0(c0) VALUES(NULL);
```



Result: Bug in MySQL



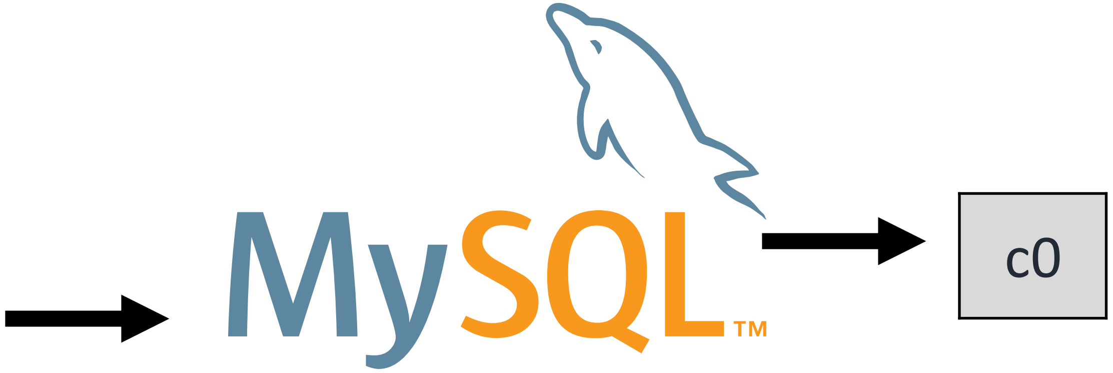
t0

c0
NULL

```
CREATE TABLE t0(c0 TINYINT);
INSERT INTO t0(c0) VALUES(NULL);
```

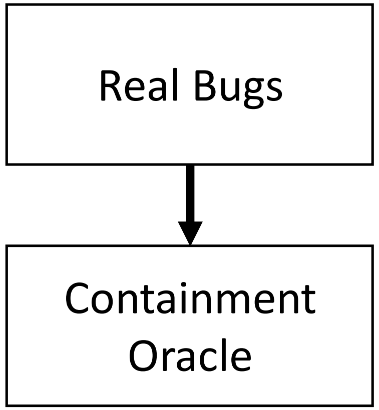
```
SELECT * FROM t0
WHERE
 NOT(t0.c0 <=> 2035382037);
```

$$\underbrace{\texttt{t0.c0 <=> 2035382037}}_{\texttt{FALSE}}$$



c0

Result: Bug in MySQL


Real Bugs

Containment Oracle

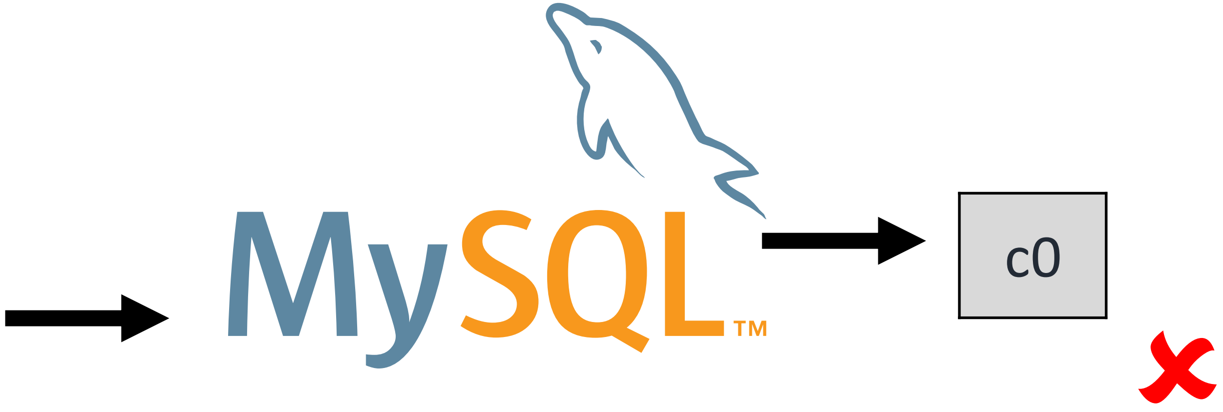
t0

c0
NULL

```
CREATE TABLE t0(c0 TINYINT);
INSERT INTO t0(c0) VALUES(NULL);
```

```
SELECT * FROM t0
WHERE
 NOT(t0.c0 <=> 2035382037);
```

FALSE

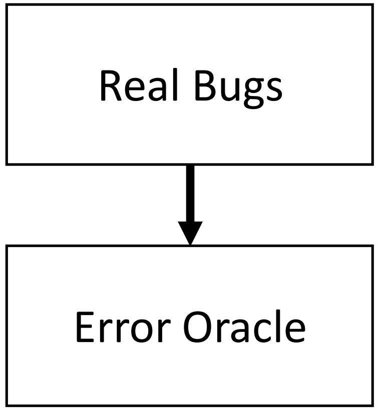


c0

✗

The **MySQL-specific equality operator** <=> malfunctioned for **large numbers**

Oracles

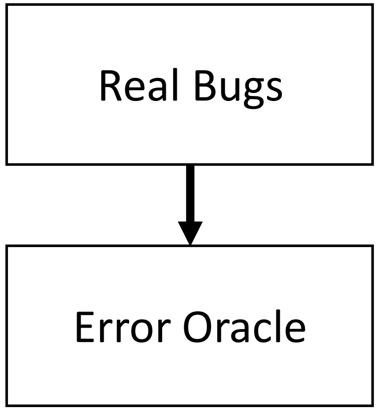


DBMS	Containment	Error	SEGFAULT
SQLite	46	17	2
MySQL	14	10	1
PostgreSQL	1	7	1
Sum	61	34	4

We also found many bugs using an *Error* oracle

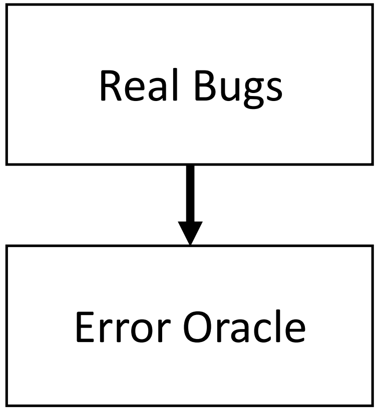
SQLite3 Bug

```
CREATE TABLE t1 (c0, c1 REAL PRIMARY KEY);
INSERT INTO t1(c0, c1) VALUES
(TRUE, 9223372036854775807), (TRUE, 0);
UPDATE t1 SET c0 = NULL;
UPDATE OR REPLACE t1 SET c1 = 1;
SELECT DISTINCT * FROM t1 WHERE (t1.c0 IS NULL);
```


Real Bugs

Error Oracle

SQLite3 Bug

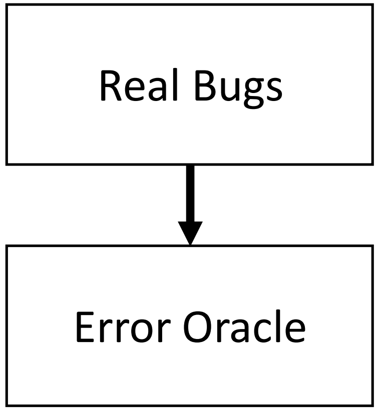

Real Bugs

Error Oracle

```
CREATE TABLE t1 (c0, c1 REAL PRIMARY KEY);
INSERT INTO t1(c0, c1) VALUES
(TRUE, 9223372036854775807), (TRUE, 0);
UPDATE t1 SET c0 = NULL;
UPDATE OR REPLACE t1 SET c1 = 1;
SELECT DISTINCT * FROM t1 WHERE (t1.c0 IS NULL);
```

```
Database disk image is malformed
```

SQLite3 Bug



Real Bugs

Error Oracle

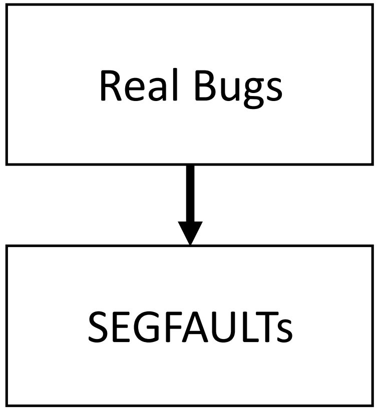
```
CREATE TABLE t1 (c0, c1 REAL PRIMARY KEY);
INSERT INTO t1(c0, c1) VALUES
(TRUE, 9223372036854775807), (TRUE, 0);
UPDATE t1 SET c0 = NULL;
UPDATE OR REPLACE t1 SET c1 = 1;
SELECT DISTINCT * FROM t1 WHERE (t1.c0 IS NULL);
```

SQLite →

```
Database disk image is malformed
```

The INSERT and UPDATE statements **corrupted the database**

Oracles



Real Bugs

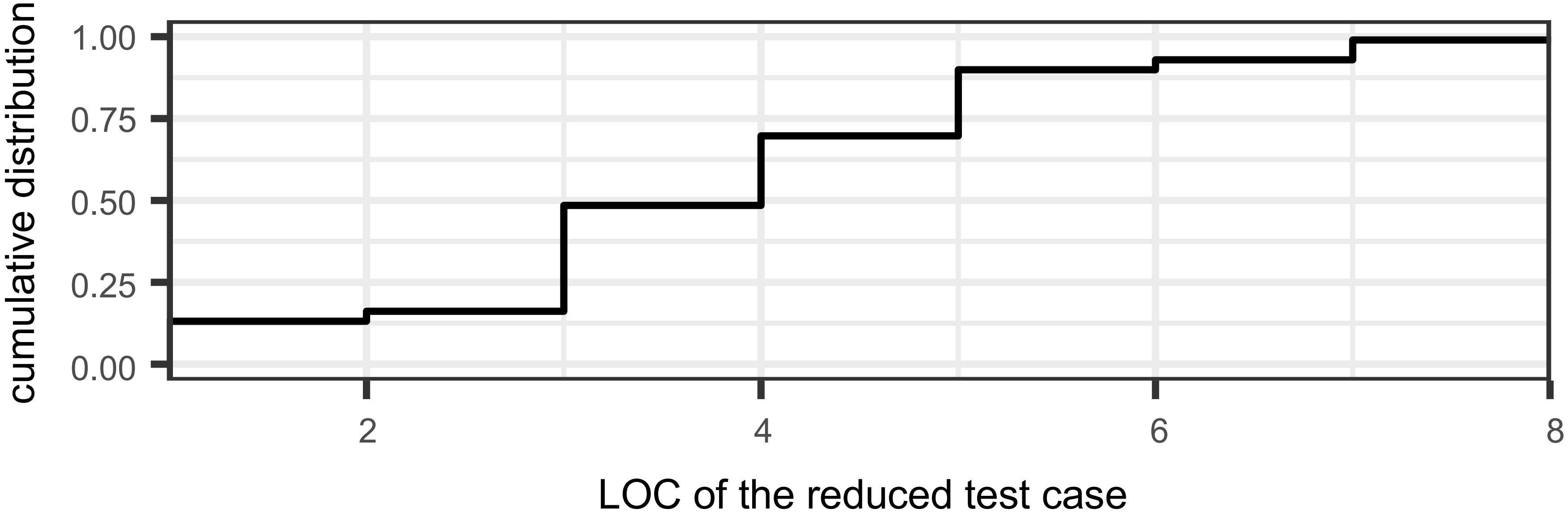
SEGFAULTs

DBMS	Containment	Error	SEGFAULT
SQLite	46	17	2
MySQL	14	10	1
PostgreSQL	1	7	1
Sum	61	34	4

We found only **a low number** of crash bugs, likely because DBMS are **fuzzed extensively**

Average Number of Statements

Real Bugs



Half of all bugs can be **reproduced** with **only 4 SQL statements**

ETH zürich

SQLite3 Bug with a Single Statement

Real Bugs

```
SELECT '' - 2851427734582196970;
```

```
-2851427734582196936
```

Subtracting a large integer from a string resulted in an **incorrect result**

Discussion

- Are the bugs relevant?

Discussion

- Are the bugs relevant?

Severity Level	#
Critical	14
Severe	8
Important	14

The SQLite developers (inconsistently) assigned **severity levels**

Discussion

- Are the bugs relevant?
- Statement coverage

Discussion

- Are the bugs relevant?
- Statement coverage

Low coverage 20%-50%, **DBMS provide a lot more** than pure database management

Discussion

- Are the bugs relevant?
- Statement coverage
- Implementation effort

Discussion

- Are the bugs relevant?
- Statement coverage
- Implementation effort

4,000-6,000 LOC per DBMS → **significantly smaller** than the DBMS

Discussion

- Are the bugs relevant?
- Statement coverage
- Implementation effort
- Limitations

Discussion

- Are the bugs relevant?
- Statement coverage
- Implementation effort
- Limitations

- Aggregate and window functions
- Difficult-to-implement functionality

Larger Picture

Pivoted Query Synthesis (PQS)

Larger Picture

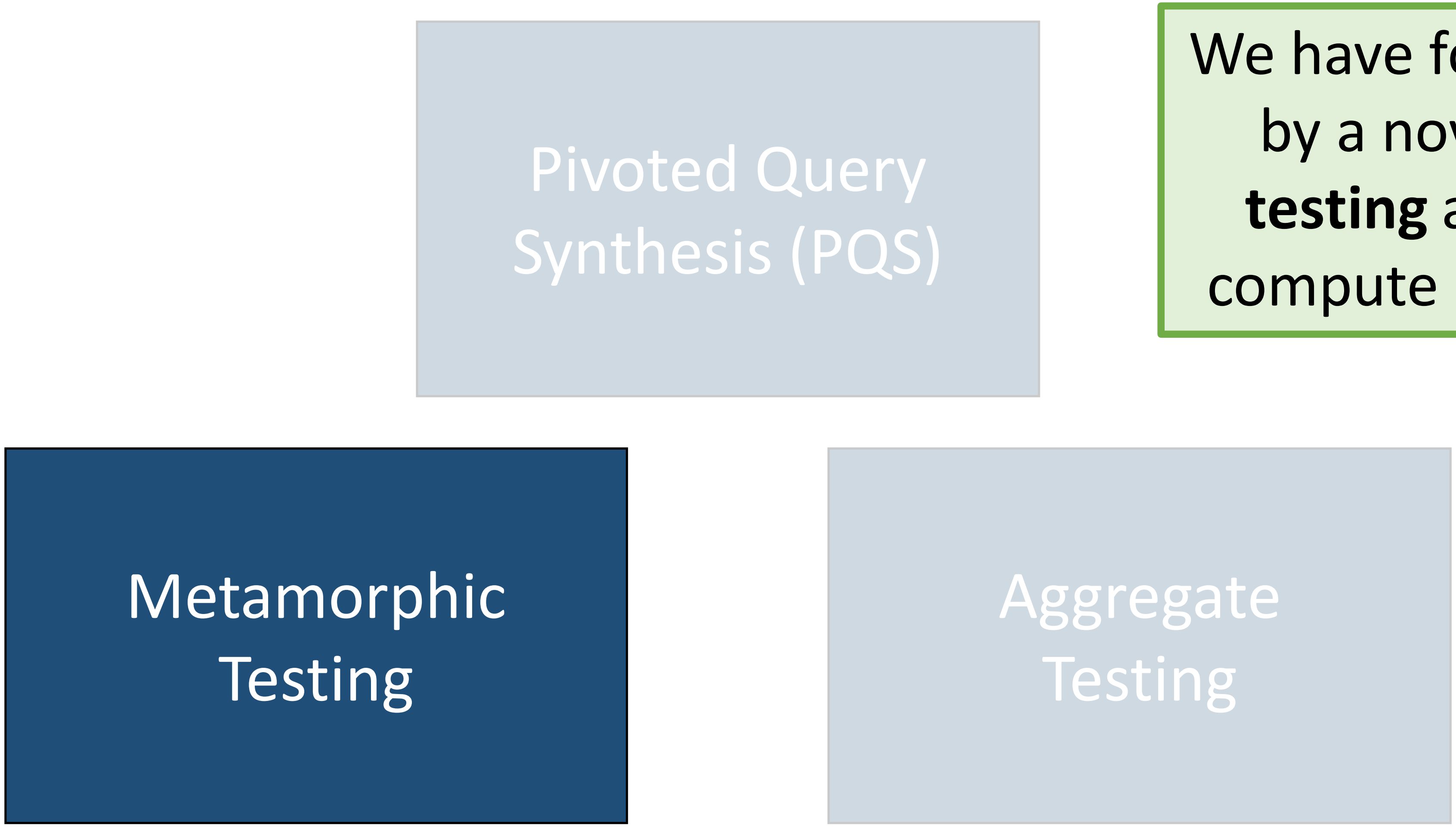
Pivoted Query Synthesis (PQS)

PQS is **one of multiple** DBMS testing approaches we have been working on

Metamorphic Testing

Aggregate Testing

Larger Picture


Pivoted Query Synthesis (PQS)

Metamorphic Testing

Aggregate Testing

We have found about **15 bugs** by a novel **metamorphic testing** approach that can compute a **precise result set**

Larger Picture

PQS is **not applicable** for testing **aggregate** and **window functions**

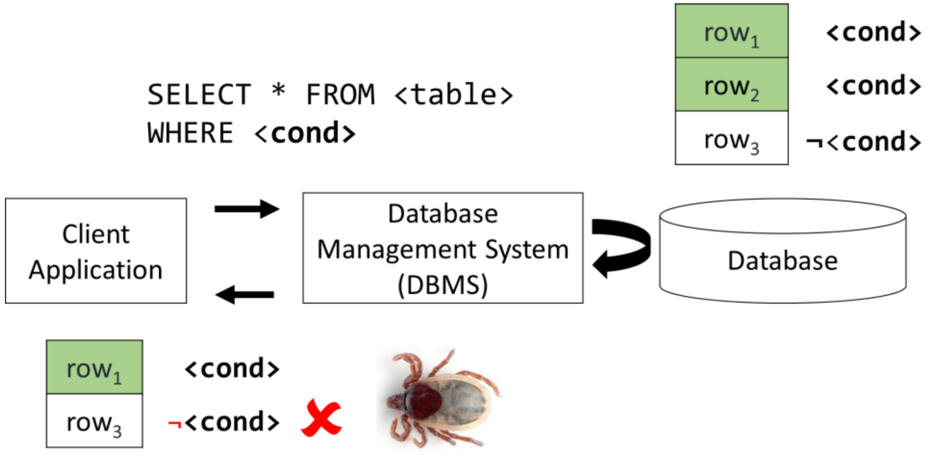
Pivoted Query Synthesis (PQS)

Metamorphic Testing

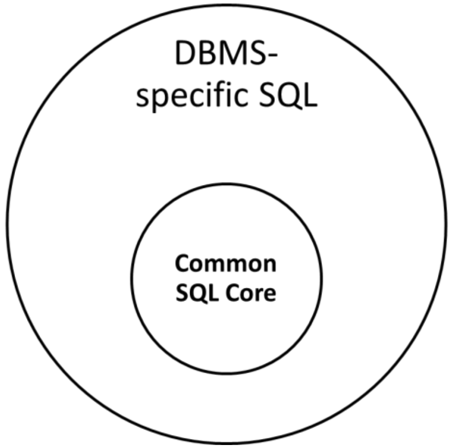
Aggregate Testing

Aim: Find Logic Bugs in DBMS

```
SELECT * FROM <table>
WHERE <cond>
```



Challenge: Precise Oracle is Difficult to Construct



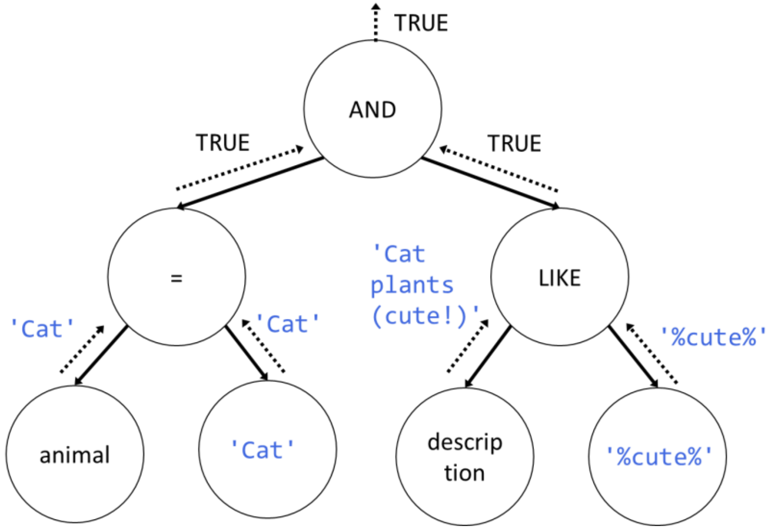
Problem: The common SQL core is **small**

Idea: Consider Only a Single Row

Idea: Construct an automatic testing approach **considering only a single row**

Column_0	Column_1	Column_2	
…	…	…	
$\text{Value}_{i,0}$	$\text{Value}_{i,1}$	$\text{Value}_{i,2}$	**Pivot Row**
…	…	…	

Create Expressions that Yield TRUE for the Pivot Row



PQS is Highly Effective

DBMS	Fixed	Verified
SQLite	65	0
MySQL	15	10
PostgreSQL	5	4
Sum	85	14

99 real bugs: addressed by code or documentation fixes, or verified as bugs