Testing Database Management Systems via Pivoted Query Synthesis

Manuel Rigger

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

@RiggerManuel @ast_eth
Database Management Systems

PostgreSQL

SQLite

MySQL
Database Management Systems

Who has heard about/used these Database Management Systems?
Databases are Used Ubiquitously

“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**

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>A cute toast cat</td>
<td><img src="image1" alt="Cat" /></td>
</tr>
<tr>
<td>Dog</td>
<td>Cute dog pic</td>
<td><img src="image2" alt="Dog" /></td>
</tr>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td><img src="image3" alt="Cat" /></td>
</tr>
</tbody>
</table>
## Relational Data Model

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

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>A cute toast cat</td>
<td><img src="image" alt="Cat Picture" /></td>
</tr>
<tr>
<td>Dog</td>
<td>Cute dog pic</td>
<td><img src="image" alt="Dog Picture" /></td>
</tr>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td><img src="image" alt="Cat Picture" /></td>
</tr>
</tbody>
</table>
Relational Data Model

animal_pictures

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>A cute toast cat</td>
<td>![Cat Picture]</td>
</tr>
<tr>
<td>Dog</td>
<td>Cute dog pic</td>
<td>![Dog Picture]</td>
</tr>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td>![Cat Picture]</td>
</tr>
</tbody>
</table>

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

```sql
SELECT picture, description
FROM animal_pictures
WHERE animal = 'Cat'
    AND description LIKE '%cute%'
```
SELECT * FROM <table>
WHERE <cond>
Database Management Systems

SELECT * FROM <table>
WHERE <cond>

Client Application  Database Management System (DBMS)  Database

row₁  <cond>  row₂  <cond>  row₃  ¬<cond>
Database Management Systems

SELECT * FROM <table>
WHERE <cond>
Goal

Aim: Detect logic bugs in DBMS
Database Management Systems

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

Client Application ↔ Database Management System (DBMS) → Database

- `row_1`
- `row_2`
- `row_3`
- `<cond>`
- `¬<cond>`
Database Management Systems

SELECT * FROM <table>
WHERE <cond>

Client Application

Database Management System (DBMS)

Database

row₁
<cond>

row₂
<cond>

row₃
¬<cond>

row₃
¬<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;
```
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

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>0</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>0</td>
<td>NULL</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>1</td>
<td>NULL</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>0</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>0</td>
<td>NULL</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>1</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Actual result set

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>0</td>
<td>1</td>
<td>NULL</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>0</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>0</td>
<td>NULL</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>1</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Actual result set

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>0</td>
<td>1</td>
<td>NULL</td>
</tr>
</tbody>
</table>

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

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

SQLite’s test cases achieve **100% branch test coverage**.

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

[https://www.sqlite.org/testing.html](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 is **extensively fuzzed** (e.g., by Google’s OS-Fuzz Project).

SQLite’s test cases achieve **100% branch test coverage**.

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


[https://www.sqlite.org/testing.html](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 → RS₁

PostgreSQL → RS₂

MySQL → RS₃

RS₁ = RS₂ = RS₃?
Differential Testing

Differential testing applies only when systems implement the same language.
Problem: Differential Testing

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”
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
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.
Idea: PQS

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

- **Simpler** conceptually and implementation-wise

<table>
<thead>
<tr>
<th>Column(_0)</th>
<th>Column(_1)</th>
<th>Column(_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Value(_{i,0})</td>
<td>Value(_{i,1})</td>
<td>Value(_{i,2})</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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

`<cond>`?
Intuition

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

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

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

```
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

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>A cute toast cat</td>
<td></td>
</tr>
<tr>
<td>Dog</td>
<td>Cute dog pic</td>
<td></td>
</tr>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td></td>
</tr>
</tbody>
</table>
Database Generation

Randomly Generate Database

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

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>A cute toast cat</td>
<td><img src="cat.jpg" alt="Cat Image" /></td>
</tr>
<tr>
<td>Dog</td>
<td>Cute dog pic</td>
<td><img src="dog.jpg" alt="Dog Image" /></td>
</tr>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td><img src="catplants.jpg" alt="Cat Image" /></td>
</tr>
</tbody>
</table>
## Pivot Row Selection

1. **Randomly Generate Database**
2. **Select Pivot Row**

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>A cute toast cat</td>
<td>![Cat Image]</td>
</tr>
<tr>
<td>Dog</td>
<td>Cute dog pic</td>
<td>![Dog Image]</td>
</tr>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td>![Cat with Plants Image]</td>
</tr>
</tbody>
</table>
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%'
```

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td><img src="image" alt="Cat image" /></td>
</tr>
</tbody>
</table>
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%'

**pivot row**

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td></td>
</tr>
</tbody>
</table>

**result set**

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>A cute toast cat</td>
<td></td>
</tr>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td></td>
</tr>
</tbody>
</table>
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%'

result set

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>A cute toast cat</td>
<td>![Cat]</td>
</tr>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td>![Cat plants]</td>
</tr>
</tbody>
</table>

pivot row

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td>![Cat plants]</td>
</tr>
</tbody>
</table>

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%'

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>A cute toast cat</td>
<td></td>
</tr>
<tr>
<td>Dog</td>
<td>Cute dog pic</td>
<td></td>
</tr>
</tbody>
</table>
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%'

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>A cute toast cat</td>
<td><img src="image" alt="Cat" /></td>
</tr>
<tr>
<td>Dog</td>
<td>Cute dog pic</td>
<td><img src="image" alt="Dog" /></td>
</tr>
</tbody>
</table>

pivot row

pivot row $\notin$ result set

result set

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td><img src="image" alt="Cat" /></td>
</tr>
<tr>
<td>Dog</td>
<td>Cute dog pic</td>
<td><img src="image" alt="Dog" /></td>
</tr>
</tbody>
</table>
Verifying the Result

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

pivot row \notin \text{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

1. Randomly Generate Database
2. Select Pivot Row
3. Generate Query for the Pivot Row
4. 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?

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

Generate an expression that yields TRUE for the pivot row.
How do we Generate Queries?

1. Randomly Generate Expression
2. Evaluate Expression on Pivot Row
3. Modify expression to yield TRUE
4. Use in WHERE clause
Random Expression Generation

We first generate a random expression.
Random Expression Generation

```
animal = 'Cat'
AND description LIKE '%cute%'
```
Random Expression Generation

\[
\text{animal} = \text{'Cat'} \quad \text{AND} \quad \text{description LIKE 'cute%'}
\]

Evaluate the tree based on the pivot row
Random Expression Evaluation

```
AND

=  'Cat'

animal  'Cat'

LIKE

description  'cute%'  '

Constant nodes return their assigned literal values

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>animal</td>
<td>description</td>
<td>picture</td>
</tr>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td></td>
</tr>
</tbody>
</table>
```
Random Expression Evaluation

\[
\text{AND} = \text{description} \text{ LIKE} \%\text{cute}\%'
\]

Column references return the values from the pivot row

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td><img src="image" alt="Picture" /></td>
</tr>
</tbody>
</table>
Random Expression Evaluation

Compound nodes compute their result based on their children

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td><img src="image.png" alt="Picture" /></td>
</tr>
</tbody>
</table>

```
= 'Cat plants (cute!)' LIKE 'Cat' != TRUE
```

```
= 'Cat' != TRUE
```

```
LIKE 'description' != TRUE
```

```
LIKE '%'cute%' != TRUE
```

```
AND TRUE
```

```
AND TRUE
```

```
AND TRUE
```

```
AND TRUE
```

```
AND TRUE
```

```
AND TRUE
```

```
AND TRUE
```

```
AND TRUE
```

```
AND TRUE
```

```
AND TRUE
```

```
AND TRUE
```

```
AND TRUE
```

```
AND TRUE
```
Random Expression Evaluation

```
<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td><img src="image_url" alt="picture" /></td>
</tr>
</tbody>
</table>
```

```
AND

TRUE

TRUE

TRUE

=  

'Cat'

'Cat (cute!)

LIKE

'\%cute\%'

animal

'C\at'

description

'\%cute\%'
```
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?

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td><img src="image" alt="Picture of cat" /></td>
</tr>
</tbody>
</table>
Random Expression Evaluation

What about when the expression does not evaluate to TRUE?

animal = 'Dog'

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td>![Cat image]</td>
</tr>
</tbody>
</table>
Random Expression Rectification

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

```java
switch (result) {
    case TRUE:
        result = randexpr;
        break;  // Move the case keyword here
    case FALSE:
        result = NOT randexpr;
        break;  // Move the case keyword here
    case NULL:
        result = randexpr ISNULL;
        break;  // Move the case keyword here
}
```

animal = 'Dog'

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td><img src="image" alt="Cat" /></td>
</tr>
</tbody>
</table>
Random Expression Rectification

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

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td><img src="image" alt="Cat" /></td>
</tr>
</tbody>
</table>
How do we Generate Queries?

```sql
SELECT picture, description
FROM animal_pictures
WHERE NOT(animal = 'Dog')
```
Evaluation
Tested DBMS

PostgreSQL

SQLite

MySQL
Tested DBMS

PostgreSQL

SQLite

MySQL

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

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Popularity Rank</th>
<th>LOC</th>
<th>First Release</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DB-Engines</td>
<td>Stack Overflow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLite</td>
<td>11</td>
<td>4</td>
<td>0.3M</td>
<td>2000</td>
</tr>
<tr>
<td>MySQL</td>
<td>2</td>
<td>1</td>
<td>3.8M</td>
<td>1995</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>4</td>
<td>2</td>
<td>1.4M</td>
<td>1996</td>
</tr>
</tbody>
</table>
# DBMS

<table>
<thead>
<tr>
<th>DBMS</th>
<th>DB-Engines</th>
<th>Stack Overflow</th>
<th>LOC</th>
<th>First Release</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>11</td>
<td>4</td>
<td>0.3M</td>
<td>2000</td>
<td>19 years</td>
</tr>
<tr>
<td>MySQL</td>
<td>2</td>
<td>1</td>
<td>3.8M</td>
<td>1995</td>
<td>24 years</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>4</td>
<td>2</td>
<td>1.4M</td>
<td>1996</td>
<td>23 years</td>
</tr>
<tr>
<td>DBMS</td>
<td>DB-Engines</td>
<td>Stack Overflow</td>
<td>LOC</td>
<td>First Release</td>
<td>Age</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>----------------</td>
<td>------</td>
<td>---------------</td>
<td>---------</td>
</tr>
<tr>
<td>SQLite</td>
<td>11</td>
<td>4</td>
<td>0.3M</td>
<td>2000</td>
<td>19 years</td>
</tr>
<tr>
<td>MySQL</td>
<td>2</td>
<td>1</td>
<td>3.8M</td>
<td>1995</td>
<td>24 years</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>4</td>
<td>2</td>
<td>1.4M</td>
<td>1996</td>
<td>23 years</td>
</tr>
</tbody>
</table>
## Bugs Overview

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Fixed</th>
<th>Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>65</td>
<td>0</td>
</tr>
<tr>
<td>MySQL</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td><strong>Sum</strong></td>
<td><strong>85</strong></td>
<td><strong>14</strong></td>
</tr>
</tbody>
</table>
## Bugs Overview

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Fixed</th>
<th>Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>65</td>
<td>0</td>
</tr>
<tr>
<td>MySQL</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Sum</td>
<td>85</td>
<td>14</td>
</tr>
</tbody>
</table>

**99 real bugs**: addressed by code or documentation fixes, or verified as bugs.
Bugs Overview

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

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Fixed</th>
<th>Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>65</td>
<td>0</td>
</tr>
<tr>
<td>MySQL</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Sum</td>
<td>85</td>
<td>14</td>
</tr>
</tbody>
</table>
Bugs Overview

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Fixed</th>
<th>Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>65</td>
<td>0</td>
</tr>
<tr>
<td>MySQL</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Sum</td>
<td>85</td>
<td>14</td>
</tr>
</tbody>
</table>

All MySQL bug reports were verified quickly
### Bugs Overview

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

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Fixed</th>
<th>Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>65</td>
<td>0</td>
</tr>
<tr>
<td>MySQL</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td><strong>Sum</strong></td>
<td><strong>85</strong></td>
<td><strong>14</strong></td>
</tr>
</tbody>
</table>
## Bugs Overview

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

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Fixed</th>
<th>Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>65</td>
<td>0</td>
</tr>
<tr>
<td>MySQL</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Sum</td>
<td>85</td>
<td>14</td>
</tr>
</tbody>
</table>
## Oracles

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Containment</th>
<th>Error</th>
<th>SEGFAULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>46</td>
<td>17</td>
<td>2</td>
</tr>
<tr>
<td>MySQL</td>
<td>14</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Sum</td>
<td>61</td>
<td>34</td>
<td>4</td>
</tr>
</tbody>
</table>
## Oracles

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Containment</th>
<th>Error</th>
<th>SEGFAULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>46</td>
<td>17</td>
<td>2</td>
</tr>
<tr>
<td>MySQL</td>
<td>14</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Sum</td>
<td>61</td>
<td>34</td>
<td>4</td>
</tr>
</tbody>
</table>

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

An index is an auxiliary data structure that should not affect the query's result.
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');
```

<table>
<thead>
<tr>
<th>c1</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A'</td>
</tr>
<tr>
<td>'a'</td>
</tr>
</tbody>
</table>
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');

SELECT * FROM t0;
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');

SELECT * FROM t0;

SQLite failed to fetch 'a'!
Result: Bug in PostgreSQL

CREATE TABLE t0(c0 INT PRIMARY KEY, c1 INT);
CREATE TABLE t1(c0 INT) INHERITS (t0);
## Result: Bug in PostgreSQL

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c0</td>
<td>c1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

```sql
CREATE TABLE t0(c0 INT PRIMARY KEY, c1 INT);
CREATE TABLE t1(c0 INT) INHERITS (t0);
INSERT INTO t0(c0, c1) VALUES(0, 0);
```
Result: Bug in PostgreSQL

```
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);
```

<table>
<thead>
<tr>
<th>c0</th>
<th>c1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c0</th>
<th>c1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Result: Bug in PostgreSQL

**t0**

<table>
<thead>
<tr>
<th>c0</th>
<th>c1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**t1**

<table>
<thead>
<tr>
<th>c0</th>
<th>c1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>c0</th>
<th>c1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

```sql
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);

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

---

Real Bugs

Containment
Oracle

---

<table>
<thead>
<tr>
<th>c0</th>
<th>c1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Result: Bug in PostgreSQL

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);

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

PostgreSQL failed to fetch the row 0 | 1
CREATE TABLE t0(c0 TINYINT);
INSERT INTO t0(c0) VALUES(NULL);
Result: Bug in MySQL

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

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

Real Bugs
Containment Oracle

MySQL™

FALSE
Result: Bug in MySQL

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

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

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

We also found many bugs using an *Error* oracle

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Containment</th>
<th>Error</th>
<th>SEGFAULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>46</td>
<td>17</td>
<td>2</td>
</tr>
<tr>
<td>MySQL</td>
<td>14</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Sum</td>
<td>61</td>
<td>34</td>
<td>4</td>
</tr>
</tbody>
</table>
SQLite3 Bug

```sql
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);
```
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);

Database disk image is malformed
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);

The INSERT and UPDATE statements corrupted the database

Database disk image is malformed
Oracles

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Containment</th>
<th>Error</th>
<th>SEGFAULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>46</td>
<td>17</td>
<td>2</td>
</tr>
<tr>
<td>MySQL</td>
<td>14</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Sum</td>
<td>61</td>
<td>34</td>
<td>4</td>
</tr>
</tbody>
</table>

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

Real Bugs ➔ SEGFAULTs
Average Number of Statements

Half of all bugs can be reproduced with only 4 SQL statements
SQLite3 Bug with a Single Statement

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?

<table>
<thead>
<tr>
<th>Severity Level</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical</td>
<td>14</td>
</tr>
<tr>
<td>Severe</td>
<td>8</td>
</tr>
<tr>
<td>Important</td>
<td>14</td>
</tr>
</tbody>
</table>

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 $\rightarrow$ 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
PQSY is one of multiple DBMS testing approaches we have been working on.
Larger Picture

We have found about **15 bugs** by a novel *metamorphic testing* approach that can compute a *precise result set*.
Pivoted Query Synthesis (PQS) is **not applicable** for testing aggregate and window functions.
Aim: Find Logic Bugs in DBMS

Challenge: Precise Oracle is Difficult to Construct

Idea: Consider Only a Single Row

Create Expressions that Yield TRUE for the Pivot Row

PQS is Highly Effective

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Fixed</th>
<th>Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>65</td>
<td>0</td>
</tr>
<tr>
<td>MySQL</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Sum</td>
<td>85</td>
<td>14</td>
</tr>
</tbody>
</table>

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