Testing Query Execution Engines with Mutations

Xinyue Chen¹, Chenglong Wang¹, Alvin Cheung²

¹University of Washington
²University of California, Berkeley
Motivation

- Query optimizers and executors are core to all modern relational database system
- With the constant introduction of new hardware architectures and query features, such query engines are updated so frequently that make them highly difficult to test
- The lack of testing leaves latent bugs in production systems that are hard to discover
Current approaches

- Developer-written test cases
  - hand-written test cases alone are often unable to cover the query space

- Randomly-generated test cases
  - random testing approaches have to spend a huge, if not impractical, amount of time on a massive amount of hardware to discover subtle query engine errors that are difficult to verify (as ground truth is often unknown)
MUTASQL

- A new light-weight mutation testing engine
- Efficiently discover and effectively report SQL engine bugs
- Allow developers to provide light-weight seed queries and optional rewrite rules
- Intelligently generate test cases such that they should return the same results as seed queries, making it easy to validate
SQLite bugs summary

We examine the SQLite bug tickets from 2009 to 2019:

<table>
<thead>
<tr>
<th>Joins</th>
<th>Group By</th>
<th>Order By</th>
<th>Distinct</th>
<th>In</th>
<th>System error</th>
<th>Table-valued function</th>
<th>Row-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>9</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

We found that the bugs with the common keywords are most prevalent.
Equivalence Mutation

Given a query $Q$ together with a sample database $D$, we want to mutate it into a query $Q'$ that is not necessarily semantically equivalent such that

$$Q'(D) = Q(D)$$

If $Q'(D)$ and $Q(D)$ return different results when running through the same query optimizer, then there is a bug in the query engine.
MUTASQL consists of two components:
Example on SQLite version 3.8.0

D:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

T:

Q:

```
Select x, y, z From T
Order By x, y, z;
```
Example on SQLite version 3.8.0

Mutation rule - add **Group By**: 

```
Select c 
From t 
Where p
```

If $g$ is unique key for Q(D)

```
Select c 
From t 
Where p 
Group By g
```

Mutation rule - add **Index**: 

```
Create Table T (x)
```

```
Create Index i On T(x)
```
Q:

```
Select x, y, z From T 
Order By x, y, z;
```

Q':

```
-- rule 1: add index
Create Index yxz On T (y, x, z);

-- rule 2: add group by
Select x, y, z From T 
Order By x, y, z 
Group By x, y, z
```

T:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Q'(D):

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Inside MUTASQL
Design of seed queries

We currently include 8 seed queries in MUTASQL.

Our design of seed queries aim to achieve the following goals:

- The seed queries should cover all primitive SQL features to trigger mutations that can cover a wide variety of query features
- The number of seed queries should also be minimal to avoid generating equivalent queries during the mutation process
- The sample database instances should be small to reduce the time needed to check for result equivalence during testing as well as making testing preconditions easier to satisfy.
MUTASQL includes 23 mutation rules that can be classified into three categories:

- Mutations on table definitions (4 rules)
  - add indexes or generated columns to the table
- Mutations on query structure (9 rules)
  - modify subquery structures or join keywords
- Predicate rewrites (10 rules)
  - modify predicates in a query by creating a new predicate that is equivalent to the original predicate with respect to the sample database
Predicate mutation

If \( c_1 \) does not contain Null

Q:

```
Select c1
From t
Where p
```

Q':

```
Select c1
From t
Where p
Or c1 is Null
```
Experiment
Implementation

We implemented MUTASQL in python and our prototype currently supports the following SQLite features:

<table>
<thead>
<tr>
<th>Select</th>
<th>From</th>
<th>Where</th>
<th>Join</th>
<th>Outer Join</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group By</td>
<td>In</td>
<td>Exists</td>
<td>Index</td>
<td>Generated columns</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(Including partial index index over expressions)</td>
<td></td>
</tr>
<tr>
<td>Like</td>
<td>Is</td>
<td>Order By</td>
<td>Limit</td>
<td>Distinct</td>
</tr>
</tbody>
</table>
Reproducing Known SQLite Bugs

- 23 SQLite versions
- 31 query engine bugs across 20 versions

<table>
<thead>
<tr>
<th>Joins</th>
<th>Group By</th>
<th>Order By</th>
<th>Index</th>
<th>Predicates</th>
<th>Distinct, Limit</th>
<th>Interactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2</td>
<td>3</td>
<td>14</td>
<td>3</td>
<td>5</td>
<td>13</td>
</tr>
</tbody>
</table>

- 1.8 mutations on average
- Max # mutations = 4
- Min # mutations = 1
- Generate and evaluate ~240,000 per hour
Discovering New Bugs

In the latest released version SQLite 3.31.1
Select $T.x$, $I.y$
From $T$, $I$
Where $T.x = I.y$
And $T.x = 12$;

Select Distinct $T.x$, $I.y$
From $T$, $I$
Where $T.x = I.y$
And $T.x = 12$;

Select Distinct $T.x$, $I.y$
From $T$, $I$, $T$ As $T2$
Where $T.x = I.y$
And $T.x = 12$
And $T.x = T2.x$;

Select Distinct $T.x$, $I.y$
From $T$, $I$, $T$ As $T2$
Where $T.x = I.y$
And $T.x = 12$
And $T.x = T2.x$
And $T.x = T.2.x$;
Thank you!

Contact us:
chenxy20@cs.washington.edu
Mutation Rules
Mutations on table definitions (4 rules)
- add indexes or generated columns to the table
Mutations on query structure (9 rules)
- modify subquery structures or join keywords
Predicate rewrites (10 rules)
- modify predicates in a query by creating a new predicate that is equivalent to the original predicate with respect to the sample database
Table definition mutation (4 rules)

- Add index
  - Add index
    
    ```sql
    Create Index i On T(x);
    ```
  - Add index on expression
    
    ```sql
    Create Index i On T(x + y);
    ```
  - Add partial index
    
    ```sql
    Create Index i On T(x) Where p(x);
    ```

- Add generated columns

  ```sql
  Create Table T (x Integer, y Text,
  -- 1. As constant
  a As (1),
  -- 2. As substring
  b As (substr(y, 1, 2)),
  -- 3. As expression
  c As (3 * x),
  -- 4. As substring with other int columns
  d As (substr(y, x, x + 1));
  ```
Predicate mutation (10 rules)

- Change to like

  \[ a = 'str' \text{ is true if and only if } a \text{ Like } 'str'. \]

- Duplicate where constraint

  If we duplicate one of the predicates \( p_1 \), \( p_1 \) and \( p_1 \) will evaluate to the same result as \( p_1 \)

\[
\begin{align*}
\text{Select } c & \quad \text{Select } c \\
(Q = \text{From } t, D) \rightarrow Q' = \text{From } t & \quad \text{Where } a = 'str', \ldots \\
\text{Where } a = 'str', \ldots & \quad \text{Where } a \text{ Like } 'str', \ldots
\end{align*}
\]

\[
\begin{align*}
\text{Select } c & \quad \text{Select } c \\
(Q = \text{From } t, D) \rightarrow Q' = \text{From } t & \quad \text{Where } p_1, \ldots \\
\text{Where } p_1, \ldots & \quad \text{Where } p_1 \text{ and } p_1, \ldots
\end{align*}
\]
Predicate mutation (10 rules)

- **Add or is null**

  If \( c_1 \) does not contain \texttt{Null}, \( c_1 \text{ is not Null} \) will always be true. \( p \) and \texttt{True} will evaluate to \( p \).

  \[
  \text{Select } c_1 \quad \text{Select } c_1 \\
  (Q = \text{From } t, \text{ D}) \rightarrow Q' = \text{From } t \\
  \text{Where } p, ... \quad \text{Where } p \\
  \text{And } c_1 \text{ is Null}
  \]

- **Change to in**

  \( c_1 = a \) is true if and only if \( c_1 \text{ in } (a) \).

  \[
  \begin{align*}
  \text{Select } c_1 & \quad \text{Select } c_1 \\
  (Q = \text{From } t, \text{ D}) \rightarrow Q' = \text{From } t \\
  \text{Where } c_1 = a, ... & \quad \text{Where } c_1 \text{ in } (a), ...
  \end{align*}
  \]
Structural mutation (9 rules)

- **Add self join**
  
  When $c_1$ self join $c_1$ on the primary keys, for every row returned by $q(D)$, there will only be one corresponding row in $c_1$. Thus, for every row in $q'(D)$, it will be the same as before except for more columns from $c_1$. When projecting the same columns as $Q$, the results are the same.

  ```sql
  Select c
  (Q = From t1, D) → Q' =
  Select c
  From t1 A, t1 B
  Where p
  And A.key = B.key
  ```

- **Add left join empty**
  
  When $t1$ left join with an empty table, there is no matched record from right table. Thus, the results for $q'(D)$ will be all the records from left table. If we do not project the columns from the empty table, which are Nulls, this mutation is semantically equivalent.

  ```sql
  Select c1
  (Q = From t, D) → Q' =
  Select c1
  q2 evals to empty
  Select c1
  From t Left Join q2
  Where p
  ```
Structural mutation (9 rules)

- Change table to subquery
  Changing a table \( t \) in \( \text{From} \) to \( \text{Select} \ast \text{From} \ t \) is semantically preserving as they both mean selecting everything from table \( t \).

- Add limit
  Suppose the number of rows returned by query is \( a \). Limiting the number of rows returned to some number equal to or greater than \( a \) will lead to the same result.

\[
\begin{align*}
\text{Select} & \ c & (Q = \text{From} \ t_1, \ D) \rightarrow Q' = \text{From} (\text{Select} \ast \text{From} \ t_1) & \text{Select} & \ cl & (Q = \text{From} \ t_1, \ D) \rightarrow Q' = \text{From} \ t_1 \\
\text{Where} & \ p, \ldots & \text{Where} & \ p & \text{Where} & \ p \\
\end{align*}
\]