Testing Query Execution Engines with Mutations

Xinyue Chen[1], Chenglong Wang[1], Alvin Cheung[2]

[1]University of Washington

[2]University 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:

Joins	Group By	Order By	Distinct	In	System error	Table-valued function	Row-value
11	2	3	6	4	9	1	3

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

Equivalence Mutation

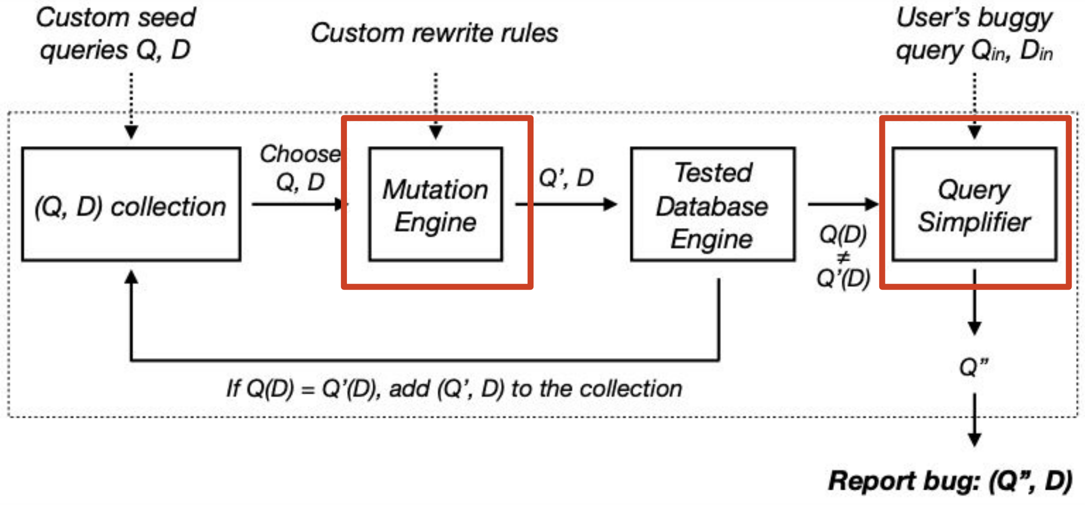
Given a query *Q* together with a sample database *D*, we want to mutate it into a query *Q′* that is **not** necessarily **semantically equivalent** such that

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

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.

System overview

MUTASQL consists of two components:



Example on SQLite version 3.8.0

D:

T:

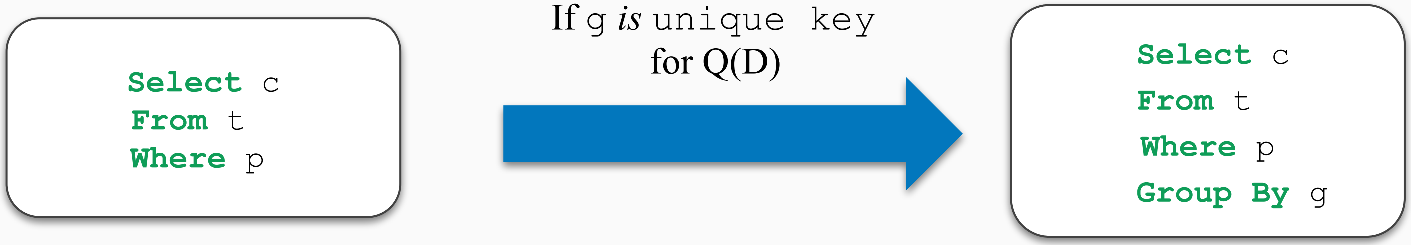
x	y	z
1	1	1
2	0	1

Q:

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

Example on SQLite version 3.8.0

Mutation rule - add **Group By**:

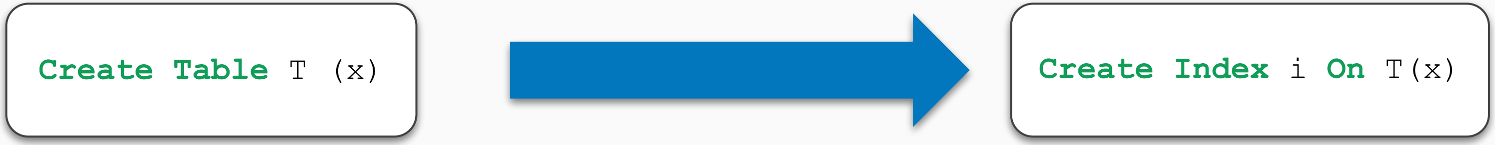


```
Select c
From t
Where p
```

If `g` *is* `unique key` for Q(D)

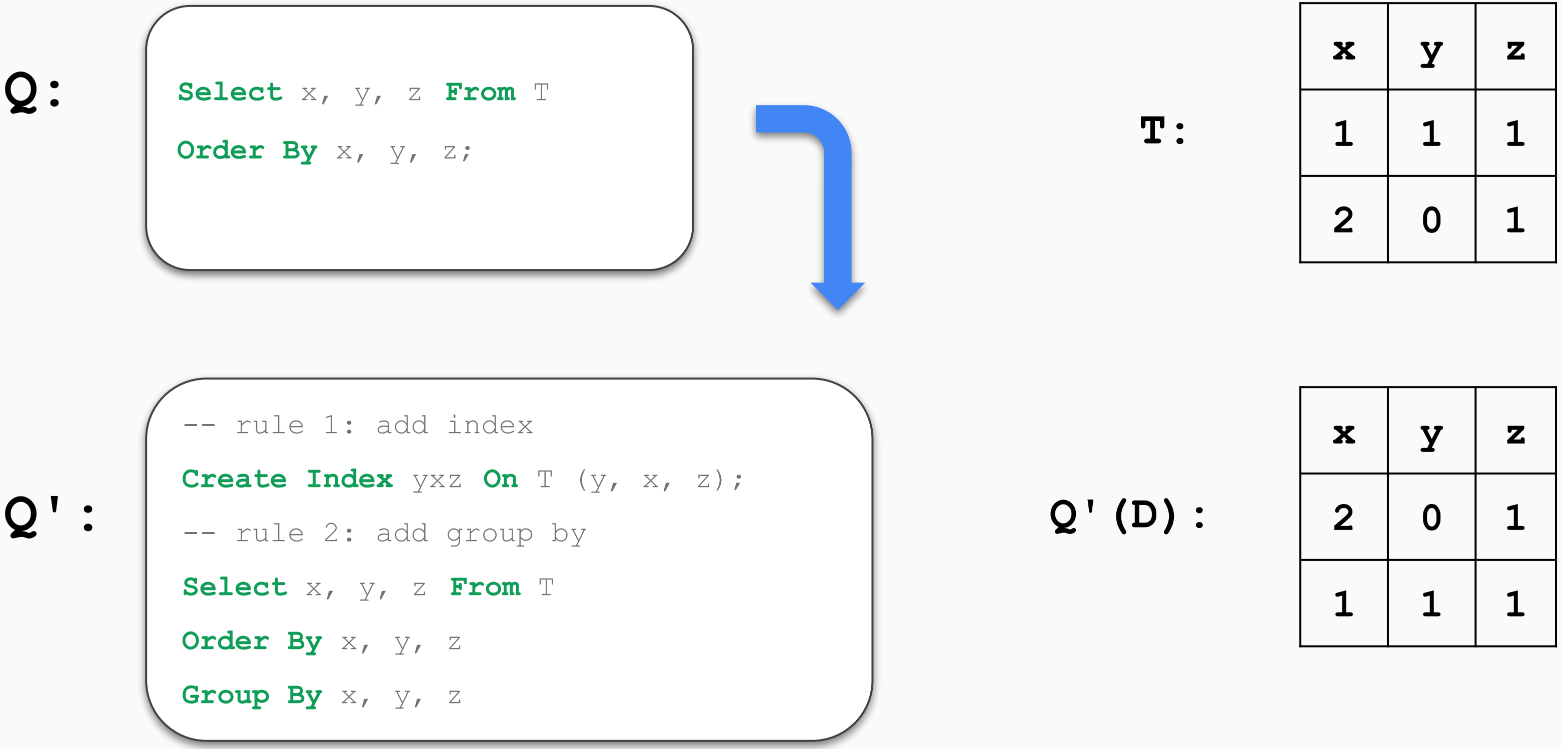
```
Select c
From t
Where p
Group By g
```

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:

x	y	z
1	1	1
2	0	1

Q'(D):

x	y	z
2	0	1
1	1	1

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:

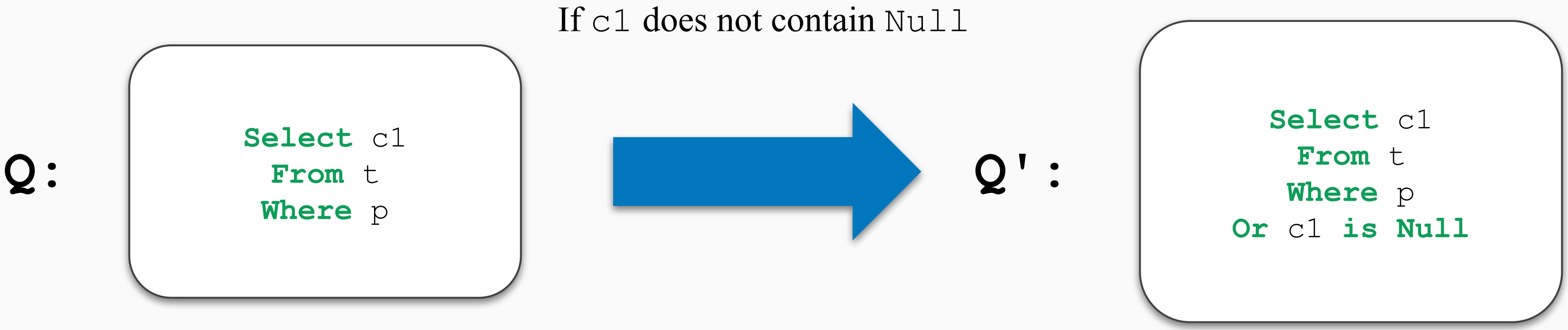
- The seed queries should cover all primitive SQL features to trigger mutations that can cover a wide variety of query features
- The number of seed queries should also be minimal to avoid generating equivalent queries during the mutation process
- The sample database instances should be small to reduce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

Mutation Rules

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



Q:

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

If c1 does not contain Null

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:

Select	From	Where	Join	Outer Join
Group By	In	Exists	Index (Including partial index index over expressions)	Generated columns
Like	Is	Order By	Limit	Distinct

Reproducing Known SQLite Bugs

Joins	Group By	Order By	Index	Predicates	Distinct, Limit	Interactions
10	2	3	14	3	5	13

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

- 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**

T:

x
'12'
'34'

I:

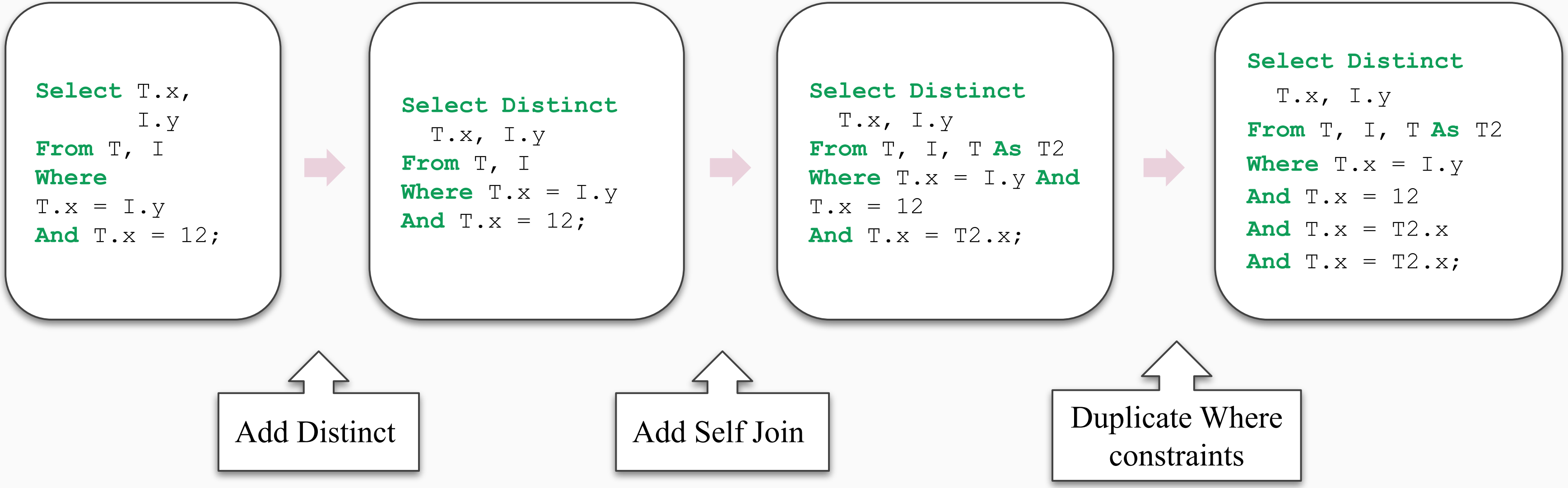
y
12
34

Q(D):

x	y
'12'	12

Q'(D):

x	y
'12'	12
'34'	12



```
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 = T2.x;
```

Add Distinct

Add Self Join

Duplicate Where constraints

Thank you!

Contact us:

chenxy20@cs.washington.edu

Questions?

Mutation Rules

Mutation Rules

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

Table definition mutation (4 rules)

- Add index
    - Add index

```
Create Index i On T(x);
```

    - Add index on expression

```
Create Index i On T(x + y);
```

    - Add partial index

```
Create Index i On T(x) Where p(x);
```

- Add generated columns

```
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'` is true if and only if `a Like 'str'`.

```
     Select c                 Select c
(Q = From t,       D) → Q′ =  From t
     Where a='str',…          Where a Like 'str',…
```

- Duplicate where constraint

If we duplicate one of the predicates p1, p1 and p1 will evaluate to the same result as p1

```
     Select c              Select c
(Q = From t,     D) → Q′ = From t
     Where p1,…            Where p1 and p1,…
```

Predicate mutation (10 rules)

- Add or is null

If `c1` does not contain `Null`, `c1 is not Null` will always be true. `p` and `True` will evaluate to `p`.

$$\left(Q = \begin{array}{l}\texttt{Select c1}\\ \texttt{From t,}\\ \texttt{Where p,}\ldots\end{array} D\right) \rightarrow Q' = \begin{array}{l}\texttt{Select c1}\\ \texttt{From t}\\ \texttt{Where p}\\ \texttt{And c1 is Null}\end{array}$$

- Change to in

`c1 = a` is true if and only if `c1 in (a)`

$$\left(Q = \begin{array}{l}\texttt{Select c1}\\ \texttt{From t,}\\ \texttt{Where c1 = a,}\ldots\end{array} D\right) \rightarrow Q' = \begin{array}{l}\texttt{Select c1}\\ \texttt{From t}\\ \texttt{Where c1 in (a),}\ldots\end{array}$$

Structural mutation (9 rules)

Add self join

When c1 self join c1 on the primary keys, for every row returned by q(D), there will only be one corresponding row in c1. Thus, for every row in q′(D), it will be the same as before except for more columns from c1. When projecting the same columns as Q, the results are the same.

```
     Select c                  Select c
(Q = From t1,   D) → Q′ = From t1 A, t1 B
     Where p,…                 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.

```
     Select c1     q2 evals to empty     Select c1
(Q = From t,     D) ──────────────→ Q′ = From t Left Join q2
     Where p                              Where p
```

Structural mutation (9 rules)

- Change table to subquery

Changing a table `t` in `From` to **`Select * From t`** is semantically preserving as they both mean selecting everything from table `t`.

$$\left(Q = \begin{array}{l}\texttt{Select c}\\ \texttt{From t1,}\\ \texttt{Where p,...}\end{array} \; D\right) \rightarrow Q' = \begin{array}{l}\texttt{Select c1}\\ \texttt{From (Select * From t1)}\\ \texttt{Where p}\end{array}$$

- 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.

$$\left(Q = \begin{array}{l}\texttt{Select c}\\ \texttt{From t1,}\\ \texttt{Where p}\end{array} \; D\right) \xrightarrow{a \geq \text{the number of rows of } q(D)} Q' = \begin{array}{l}\texttt{Select c1}\\ \texttt{From t1}\\ \texttt{Where p}\\ \texttt{Limit a}\end{array}$$