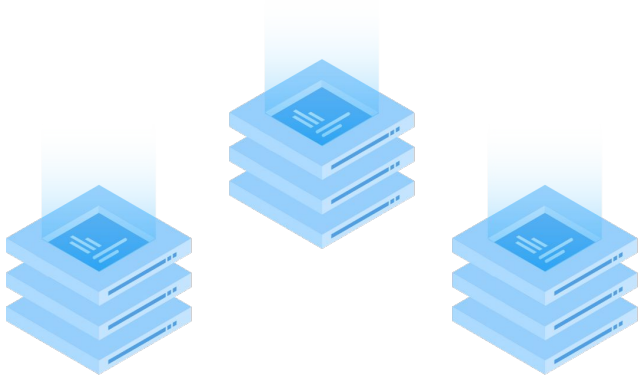
SQL Spider

Presented by Yuanjia Zhang/Zhongyang Guan/Xiangrui Meng

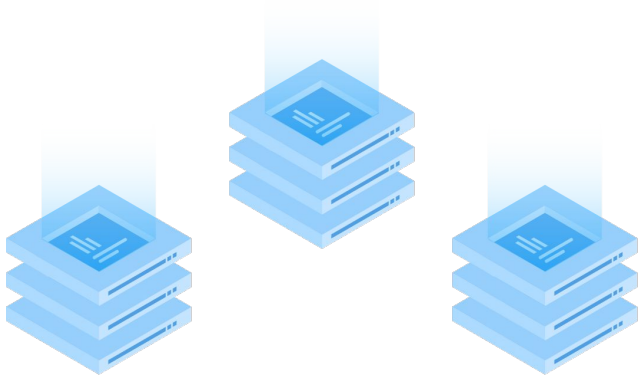
PingCAP TiDB



SQL Spider

Presented by Yuanjia Zhang/Zhongyang Guan/Xiangrui Meng

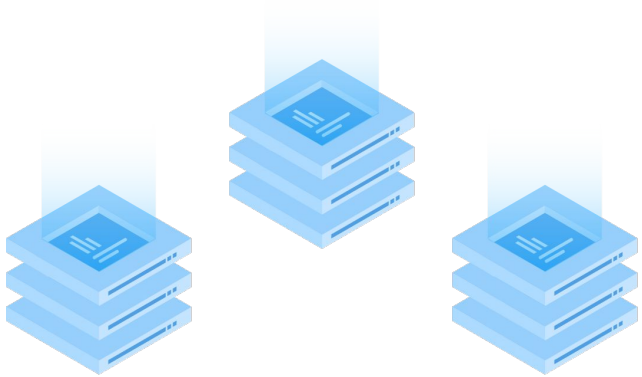
PingCAP TiDB



SQL Spider

Presented by Yuanjia Zhang/Zhongyang Guan/Xiangrui Meng

PingCAP TiDB



PingCAP

Part I - SQL Spider 介绍

项目介绍

项目名称：SQL Spider

项目介绍：提出两种 SQL 空间中的搜索策略（框架），提取有用的 SQL 进行测试

团队：祥瑞

团队成员：管仲洋，孟祥瑞，张原嘉

想法来源

ONCALL 中遇到的两个问题：

- 有 t1，t2，t1 有索引 (a, b, c)，在进行如下 join 时 t1.a=t2.a and t1.b=2 and t1.c=t2.c 会有 bug，原因总结：三列进行 IndexJoin 时，第二列为 constant 时会出错。
- 如果有某列 c 是 unsigned，表达式 IF(c>1, c, -c) 的返回类型会出错，原因总结：为 IF 判断返回类型时，如果第一列是 unsigned，但是其他参数又有可能是 signed 的时候会出错。

希望有工具能自动生成测试 Query，覆盖到诸如此类的 case。

已有方案

目前我们用的比较多的 Query 生成工具，RandGen：

- Example: SELECT _field[invariant] FROM _table[invariant] WHERE _table[invariant]._field[invariant] BETWEEN _digit[invariant] AND _digit[invariant] + _digit;
- Query 的模式需要提前定义好，随机能力有限，需要较多人为介入，无法在给定数据集上自动生成查询；
- 消耗人力去构造“可能出问题的” case，效率较低；

另一个出名的 Query 生成工具，SQLSmith：

- 官方不支持 MySQL（近期唐长老已经补充）；
- 生成的 Query 只包含最基本的运算符号比如 + - * / IsNull 等，不能包含较为复杂的 builtin-func 比如 TimeDiff 等；

预期目标

Table Schema + Database Schema ⇒ 合法的 SQL 空间
Table Schema + Database Schema + 一些约束 ⇒ 合法且有用的 SQL 空间

目标：实现一个工具，能在 Table 和 Database Schema 形成的合法 SQL 空间内，自动生成有用的 SQL，用于测试；

Table Schema 包括：

- Table Name
- Columns
  - Column Name
  - Column Type

Database Schema 包括：

- Operators(Join/Agg/Proj...)
- Expressions
  - ScalarFuncs
  - AggFuncs
  - Type system

实际产出

- 设计并实现了两种 SQL 空间中的搜索策略
  - 完全随机生成 Query
  - 广度优先搜索相邻 Query
- 在每种搜索策略上，提供了更通用的抽象，方便 扩展
- 发现多个 bug

PingCAP

Part II - 随机生成 Query

随机生成-初步设计

基本思路：随机出一个 LogicalPlan 或者 AST，然后反向改写成 Query；

- AST ⇒ SQL：不好感知数据类型，导致约束比较弱
- Logical Plan ⇒ SQL ✔️

Logical Plan 生成被拆解为：

1. 生成不带表达式的Logical Plan（空的算子树）
2. 为算子树填充表达式（完整的 Logical Plan）

再加上第三步：

3. 改写 Logcial Plan 为 Query

随机生成算子树-已经支持的算子

目前支持的算子：

- Filter{Where Expr}：多个条件用 And 链接，故只有一个 Expr
- Proj{Proj []Expr}
- Order{OrderBy []Expr}
- Limit{Limit int}
- Agg{AggExprs []Expr, GroupBy []Expr}
- Join{JoinCond Expr}：多个条件用 And 链接
- Table{Schema TableSchema, Selected []int}
    - Schema：用户输入的某个表结构
    - Selected：表示该表的哪几列被选中了

其他说明：

- Table 表示从表读数据，只能为叶节点，他没有儿子；
- Join 有两个儿子，其他算子除 Table 只有一个儿子；

随机生成算子树-生成算法

递归向下的生成：

1. 随机的选择当前的节点类型：
   a. 如果为 Table，则生成后直接返回；
   b. 如果不为 Table，生成后递归的生成其儿子节点；

2. 用概率表控制其基本结构：
   为每种算子设定出现的概率，用于大概控制算子树的结构；
   比如 Table 出现的概率为 20%，则某一条路径深度超过 5 的概率为 (1-20%)^5；

3. 动态的条件约束，产生更合理的结构：
   a. 根据当前的深度，动态的改变每种算子出现的概率，比如“当深度大于 6 时，Table 的概率为 100%”；
   b. 记录当前路径的已有算子信息，过滤掉无用的结构，比如“...Filter->Filter->...”这种结构；

随机生成表达式-已经支持的表达式

目前支持的表达式：

- 已支持的类型：Int, Real, Decimal, String, Datetime
- 已支持的表达式：
  - Constant
  - Column
  - Funcs
    - ScalarFunc: 82 个（剩下的简单调试也能轻松导入）
    - AggFunc: 5 个（AVG/SUM/COUNT/MAX/MIN）

随机生成表达式-生成算法

假设我们现在准备为某个算子生成对应的表达式，如为 Filter 生成 Where，已知 Filter 的子节点会返回 3 列（c0, c1, c2）（别名形式），生成表达式过程如下：

- 选择生成哪种类型表达式：
  - Constant：生成对应类型常数，返回；
  - Column：从（c0, c1, c2）选一个对应类型的列，返回；
  - Func：选一个对应返回类型的函数，然后递归为其生成儿子节点；

- 用动态的概率表来控制表达式树的结构，同生成算子树类似，不再赘述；

- 约束：生成过程感知类型，直接避免大部分无意义的表达式，如：
  pow(“abc”, “xxx”)，concat(23, 56)；
  接口：GenExpr(cols []Expr, tp TypeMask, validate ValidateExprFn) Expr
  TypeMask 直接表示需要的类型

随机生成表达式-更强的约束

动态的参数类型：比如 =, > 这些函数，通常两边的参数都为同一类类型，比如 string，number

接口：ArgTypeMask(i int, prvArgs []Expr) TypeMask

定义了 Validator 用来检测表达式是否有意义：

接口：type ValidateExprFn func(expr Expr) bool

已有的 Validator：

- MustContainsCols：表达式必须包含 Col，给 Filter.Where 和 Proj.Proj 用，用于过滤掉类似于 Select 1, 2, 3 from t where 4 and 5 and 6的Query；
- RejectConstants：如果某个 ScalarFunc 的所有参数都是常数，则拒绝，比如 where c0 > 10 and Ceil(25 * 0.88)；

可以为每个 Func 单独定义 Validator，如为 - 定义用来过滤 c0 - c0 这种无意义的表达式；

随机生成-算子表达式生成策略概览

- Filter{Where Expr}：常规方法 + MustContainsCols + RejectConstants 过滤
- Proj{Proj []Expr}：同上
- Order{OrderBy []Expr}：目前只考虑拿全部列排序，且不包裹函数
- Limit{Limit int}：随机生成一个 int
- Agg{AggExprs []Expr, GroupBy []Expr}：
  - AggExprs：随机选取 AggFunc 包裹在 Column 上
  - GroupBy：同 Order.OrderBy
- Join{JoinCond Expr}：生成同时包含左右儿子列的表达式，目前实现比较简单，只会生成 LCol op Rcol，op 包括 >, <, =, !=, <=, >=

随机生成-转换为 SQL

目前实现较为简单，方法为直接把子树当做子查询给父亲；

比如 Filter(c0 > 2)->Proj(c0+c1 AS c0)->Agg(c0, sum(c1) by c0)->Table(c0, c1)，会被转换为 SQL:

```
Select * from (
  select c0+c1 as c0 from (
    select c0, sum(c1) as c1 from (
      select cxxx as c0, cyyy as c1 from t
    ) t group by c0
  ) t
) t where c0 > 2;
```

每个算子实现了 ToSQL() string 的方法，可以修改它实现更“易读”的转换方式；

比如 Filter{Where} 不嵌套子查询，而是把列替换后直接把 Where 贴在儿子的 SQL 后；

目前觉得比较繁琐又不是很重要，就先没做；

随机生成-扩展性强

扩展算子（比如 window）：

1. 定义 Window 实现 Node 接口，实现 ToSQL() string, Columns() []Expr 等方法
2. 把 Window 加到概率表内让他能够出现在 LogicalPlan 中
3. 实现一个填充 Window 内部表达式的函数 FillWindow(w *Window)

扩展函数（比如 IsTrue）：

1. 将函数的信息添加到我们的函数列表即可

```
var FuncInfos = map[string]FuncInfo{
    FuncEQ:          { Name: FuncEQ, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncGE:          { Name: FuncGE, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncLE:          { Name: FuncLE, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncNE:          { Name: FuncNE, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncLT:          { Name: FuncLT, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncGT:          { Name: FuncGT, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncIsTrue:      { Name: FuncIsTrue, MinArgs: 1, MaxArgs: 1, ArgsTypes: nil, ReturnType: TypeDefault, Validate: nil},
    FuncIf:          { Name: FuncIf, MinArgs: 3, MaxArgs: 3, ArgsTypes: nil, ReturnType: TypeDefault, Validate: nil},
    FuncIfnull:      { Name: FuncIfnull, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeDefault, Validate: nil},
    FuncLcase:       { Name: FuncLcase, MinArgs: 1, MaxArgs: 1, ArgsTypes: []TypeMask{TypeString}, ReturnType: TypeString, Validate: nil},
    FuncLeft:        { Name: FuncLeft, MinArgs: 2, MaxArgs: 2, ArgsTypes: []TypeMask{TypeString, TypeNumber}, ReturnType: TypeString, Validate: nil},
    FuncRight:       { Name: FuncRight, MinArgs: 2, MaxArgs: 2, ArgsTypes: []TypeMask{TypeString, TypeNumber}, ReturnType: TypeString, Validate: nil},
```

随机生成-测试结果

测试表结构：

```
CREATE TABLE t (
    col_int int default null,
    col_double double default null,
    col_decimal decimal(40, 20) default null,
    col_string varchar(40) default null,
    col_datetime datetime default null,
    key(col_int),
    key(col_double),
    key(col_decimal),
    key(col_string),
    key(col_datetime),
    key(col_int, col_double),
    key(col_int, col_decimal),
    key(col_int, col_string),
    key(col_double, col_decimal)
);
```

跑了 1000 条 SQL，目前产生的错误情况如下：

都报错 & ErrCode 不一致	8
MySQL 报错 & TiDB 不报错	4
MySQL 不报错 & TiDB 报错	92
TiDB Panic	7
都不报错 & 结果不一致	34

MySQL 不报错 & TiDB 报错大多为字面量 out of range，比如：

```
Error 1690: DOUBLE value is out of range in 'pow(2147483647, 2147483647)'
```

结果不一致错误较多，是由于同种原因的 Bug 可能被多次触发；

随机生成-测试结果分析

Case 1：

```
SELECT * FROM ( SELECT c0, MIN(c1) AS c1, MAX(c2) AS c2 FROM (
 SELECT c0, COUNT(c1) AS c1, SUM(c2) AS c2 FROM (
SELECT * FROM (   SELECT col_double AS c0, col_string AS c1, col_datetime AS c2 FROM t) t WHERE c0
 ) AS t GROUP BY c0
) AS t GROUP BY c0) t ORDER BY c0, c1, c2;
```

TiDB 返回 47 行，MySQL 返回 93 行；

定位发现是 Coprocesser 处理下推条件 'WHERE c0' 有问题，过滤错了数据；

随机生成-测试结果分析

Case 2：

```
SELECT * FROM ( SELECT ACOS(c0) AS c0, (TO_BASE64(IF((c1 * (c3 - (c1 ^ 2013464221461210368))), UPPER('XtS38pz2hu'),
ABS(0.143))) OR 'z81Z4W') AS c1, c2 AS c2, LOWER(LTRIM(c0)) AS c3,
TO_BASE64(REPLACE(LCASE((0.000 + FLOOR(LOG(0.000)))), c0, POWER((ROUND(c1) ^ LOG(LN(-0.000))),
4377038545950721024))) AS c4, (IFNULL(c0, c1) / -451.353) AS c5, (OCT(RTRIM(c0)) >= LCASE(IFNULL(0.410, c1))) AS c6 FROM (
 SELECT t1.c0 AS c0,t2.c0 AS c1,t2.c1 AS c2,t2.c2 AS c3,t2.c3 AS c4 FROM (
  SELECT col_string AS c0 FROM t) AS t1, (
  SELECT col_int AS c0, col_double AS c1, col_string AS c2, col_datetime AS c3 FROM t) AS t2
  WHERE (t1.c0 < t2.c2)
) AS t) t ORDER BY c0, c1, c2, c3, c4, c5, c6 LIMIT 98;
```

MySQL 返回 98 行，TiDB 报错 constant -4.3238814911258294e+27 overflows bigint；

应该是浮点数和整数计算边界有问题；

随机生成-测试结果分析

Case 3：

```
SELECT * FROM ( SELECT t1.c0 AS c0,t2.c0 AS c1 FROM (
 SELECT col_string AS c0 FROM t) AS t1, (
SELECT * FROM (  SELECT SIN(IFNULL(c0, DAYOFWEEK('1999-12-01 12:50:06'))) AS c0 FROM (
  SELECT col_decimal AS c0 FROM t
) AS t) t WHERE ((REVERSE((c0 OR WEEKOFYEAR('2017-04-18 19:15:51'))) < ('2007-09-03 02:37:55' + (SIGN(0.000) | LN(c0)))) <=
(ABS(c0) AND c0))) AS t2
 WHERE (t1.c0 != t2.c0)) t ORDER BY c0, c1;
```

MySQL 返回空结果无报错，TiDB panic；

定位后发现是处理 c0 OR WEEKOFYEAR('2017-04-18 19:15:51') 改写逻辑有问题，导致子表达式没有被赋值，出现空指针；

随机生成-测试结果分析

Case 4：

```
SELECT * FROM ( SELECT c0, c1, c0 AS c2, c1 AS c3, MAX(c4) AS c4 FROM (
 SELECT t1.c0 AS c0,t1.c1 AS c1,t2.c0 AS c2,t2.c1 AS c3,t2.c2 AS c4 FROM (
  SELECT col_double AS c0, col_string AS c1 FROM t) AS t1, (
  SELECT col_int AS c0, col_string AS c1, col_datetime AS c2 FROM t) AS t2
  WHERE (t1.c1 != t2.c2)
) AS t GROUP BY c0, c1, c0, c1) t ORDER BY c0, c1, c2, c3, c4 LIMIT 24;
```

化简后的复现 SQL 如下：

```
SELECT count(*) from t t1, t t2 where t1.col_string != t2.col_datetime;
```

TiDB 返回 0，MySQL 返回 8184；

应该是计算 Hash 时 Join Key 在 string 和 datetime 类型下计算结果有误，导致没 Join 上；

随机生成-测试结果分析

Case 5：

```
SELECT * FROM ( SELECT c0, c1, MIN(c2) AS c2, MIN(c3) AS c3 FROM (
SELECT * FROM (  SELECT c0, c1, c2, AVG(c3) AS c3 FROM (
  SELECT col_int AS c0, col_double AS c1, col_decimal AS c2, col_datetime AS c3 FROM t
 ) AS t GROUP BY c0, c1, c2) t WHERE FLOOR((LN(IF(c1, -0.000, (c1 ^ COS(c2))))) < c2))
) AS t GROUP BY c0, c1) t ORDER BY c0, c1, c2, c3
```

MySQL 无结果，TiDB 输出 41 行；

定位原因在于 IF(c1, -0.000, (c1 ^ COS(c2)) 结果和 MySQL 不一致，主要为 IF 的第一个参数为浮点数时有问题；

随机生成-测试结果分析

Case 6：

```
SELECT * FROM (SELECT * FROM ( SELECT AVG(c0) AS c0 FROM (
 SELECT (c0 >= UPPER(REPEAT(UCASE(LOWER('UebLHwFSV')), ATAN(IFNULL(13.498, IF(0.000, '2026-08-14 10:51:21', c0)))))) AS c0
FROM (
  SELECT col_string AS c0 FROM t
 ) AS t
) AS t ) t WHERE CEIL(c0)) t ORDER BY c0
```

MySQL 返回 0.2473，TiDB 返回 1.0000;

定位是 c0 和 UPPER(...) 字符串比较有问题，导致结果和 MySQL 不一致；

PingCAP

Part III - 广度优先搜索 Query

广度优先搜索-想法来源

问题：怎么和已经有的错误 Query 结合起来起来搜索？

想法：把已知的错误 Query 当做搜索的起点，找到其附近相似的 Query？

⇒ 找到离错误 Query 最“近”（相似）的 N 个 Query

⇒ 这 N 个 Query 可以帮助我们：

1. 确认原错误 Query “真的”被修复了，增加信心 :D
2. 更有针对性，或许能在错误 Query 附近发现新的问题

⇒ 实现：Query + Transform Rules + BFS

广度优先搜索-初步设计

定义转换规则接口：

```
type TransformRule interface {
    OneStep(node Expr, ctx TransformContext) []Expr
}
```

表示某个表达式在该条规则作用下，向某些方向“走一步”能够到达的状态；

有了转换规则，我们就可以进行 BFS 了，很普通的 BFS 方法：

```
queue.PushBack(startNode)
for queue.Len() > 0 {
    node := queue.PopFront()
    for _, rule := range rules {
        queue.PushBack(rule.OneStep(node)...)
    }
}
```

当然还需要状态去重，判断结束条件等；

广度优先搜索-转换规则

规则设计的还比较简陋，不过框架是完整的，可以轻易添加删除规则；

目前规则有：

1. ConstantToColumn：把表达式树中的某个常量替换为列
2. ColumnToConstant：把表达式树中的某个列替换为常量
3. ReplaceChildToConstant：将表达式树中的某个子树替换为常量
4. ReplaceChildToColumn：将表达式树中的某个子树替换为列
5. ReplaceChildToFunc：将表达式树中的某个子树替换为新的 function

广度优先搜索-目前测试

以刚刚发现的错误 Query 作为 BFS 起点：

```
SELECT * FROM ( SELECT CEIL(TAN(IF(c1, c1, c1))) AS c0 FROM (
   SELECT col_string AS c0, col_datetime AS c1 FROM t
) AS t) t ORDER BY c0;
```

得到了如下的几个 Query，时间有限，目前还没发现新的问题 Query :(

只能作为 Demo 演示一下思想...

```
SELECT * FROM ( SELECT CEIL(-4824926468.651) AS c0 FROM (
  SELECT col_string AS c0, col_datetime AS c1 FROM t
 ) AS t) t ORDER BY c0
```

```
SELECT * FROM ( SELECT CEIL(c0) AS c0 FROM (
  SELECT col_string AS c0, col_datetime AS c1 FROM t
 ) AS t) t ORDER BY c0
```

```
SELECT * FROM ( SELECT CEIL(IF(c1, c1, c0)) AS c0 FROM (
  SELECT col_string AS c0, col_datetime AS c1 FROM t
 ) AS t) t ORDER BY c0
```

PingCAP

Part IV - 总结 & 未来工作

总结

- 随机生成 Query
  - 随机性强，覆盖面广，发现人想象不到的 bug
  - 避免手动构造 case 的枯燥劳动
  - 容易落地，已经发现不少问题
  - 框架完善，扩展性强，新增算子/表达式/约束都很容易
- 广度优先生成 Query
  - 框架完善，扩展性强，轻松添加规则
  - 针对性更强，更容易构造相似归因的 Query 集合进行有针对的测试

未来工作

- 随机生成 Query
  - 支持更多的算子 (window, view...)
  - 支持更多的表达式
  - 支持DML
- 广度优先生成 Query
  - 定义更多的 Rule，加大搜索空间
- 在搜索空间引入更多的搜索策略...(比如参考 gofuzz-testing 设计带反馈的 DFS 搜索算法)
- 拓宽框架应用场景，不仅局限于正确性验证，比如和 MySQL 对比进行性能测试，分析执行计划等

PingCAP

Thank You !

PingCAP

Part I - SQL Spider 介绍

项目介绍

项目名称：SQL Spider

项目介绍：提出两种 SQL 空间中的搜索策略（框架），提取有用的 SQL 进行测试

团队：祥瑞

团队成员：管仲洋，孟祥瑞，张原嘉

想法来源

ONCALL 中遇到的两个问题：

- 有 t1，t2，t1 有索引 (a, b, c)，在进行如下 join 时 t1.a=t2.a and t1.b=2 and t1.c=t2.c 会有 bug，原因总结：三列进行 IndexJoin 时，第二列为 constant 时会出错。
- 如果有某列 c 是 unsigned，表达式 IF(c>1, c, -c) 的返回类型会出错，原因总结：为 IF 判断返回类型时，如果第一列是 unsigned，但是其他参数又有可能是 signed 的时候会出错。

希望有工具能自动生成测试 Query，覆盖到诸如此类的 case。

已有方案

目前我们用的比较多的 Query 生成工具，RandGen：

- Example: SELECT _field[invariant] FROM _table[invariant] WHERE _table[invariant]._field[invariant] BETWEEN _digit[invariant] AND _digit[invariant] + _digit;
- Query 的模式需要提前定义好，随机能力有限，需要较多人为介入，无法在给定数据集上自动生成查询；
- 消耗人力去构造“可能出问题的” case，效率较低；

另一个出名的 Query 生成工具，SQLSmith：

- 官方不支持 MySQL（近期唐长老已经补充）；
- 生成的 Query 只包含最基本的运算符号比如 + - * / IsNull 等，不能包含较为复杂的 builtin-func 比如 TimeDiff 等；

预期目标

Table Schema + Database Schema ⇒ 合法的 SQL 空间
Table Schema + Database Schema + 一些约束 ⇒ 合法且有用的 SQL 空间

目标：实现一个工具，能在 Table 和 Database Schema 形成的合法 SQL 空间内，自动生成有用的 SQL，用于测试；

Table Schema 包括：

- Table Name
- Columns
  - Column Name
  - Column Type

Database Schema 包括：

- Operators(Join/Agg/Proj...)
- Expressions
  - ScalarFuncs
  - AggFuncs
  - Type system

实际产出

- 设计并实现了两种 SQL 空间中的搜索策略
  - 完全随机生成 Query
  - 广度优先搜索相邻 Query
- 在每种搜索策略上，提供了更通用的抽象，方便 扩展
- 发现多个 bug

PingCAP

Part II - 随机生成 Query

随机生成-初步设计

基本思路：随机出一个 LogicalPlan 或者 AST，然后反向改写成 Query；

- AST ⇒ SQL：不好感知数据类型，导致约束比较弱
- Logical Plan ⇒ SQL ✔️

Logical Plan 生成被拆解为：

1. 生成不带表达式的Logical Plan（空的算子树）
2. 为算子树填充表达式（完整的 Logical Plan）

再加上第三步：

3. 改写 Logcial Plan 为 Query

随机生成算子树-已经支持的算子

目前支持的算子：

- Filter{Where Expr}：多个条件用 And 链接，故只有一个 Expr
- Proj{Proj []Expr}
- Order{OrderBy []Expr}
- Limit{Limit int}
- Agg{AggExprs []Expr, GroupBy []Expr}
- Join{JoinCond Expr}：多个条件用 And 链接
- Table{Schema TableSchema, Selected []int}
    - Schema：用户输入的某个表结构
    - Selected：表示该表的哪几列被选中了

其他说明：

- Table 表示从表读数据，只能为叶节点，他没有儿子；
- Join 有两个儿子，其他算子除 Table 只有一个儿子；

随机生成算子树-生成算法

递归向下的生成：

1. 随机的选择当前的节点类型：
   a. 如果为 Table，则生成后直接返回；
   b. 如果不为 Table，生成后递归的生成其儿子节点；

2. 用概率表控制其基本结构：
   为每种算子设定出现的概率，用于大概控制算子树的结构；
   比如 Table 出现的概率为 20%，则某一条路径深度超过 5 的概率为 (1-20%)^5；

3. 动态的条件约束，产生更合理的结构：
   a. 根据当前的深度，动态的改变每种算子出现的概率，比如“当深度大于 6 时，Table 的概率为 100%”；
   b. 记录当前路径的已有算子信息，过滤掉无用的结构，比如“...Filter->Filter->...”这种结构；

随机生成表达式-已经支持的表达式

目前支持的表达式：

- 已支持的类型：Int, Real, Decimal, String, Datetime
- 已支持的表达式：
  - Constant
  - Column
  - Funcs
    - ScalarFunc: 82 个（剩下的简单调试也能轻松导入）
    - AggFunc: 5 个（AVG/SUM/COUNT/MAX/MIN）

随机生成表达式-生成算法

假设我们现在准备为某个算子生成对应的表达式，如为 Filter 生成 Where，已知 Filter 的子节点会返回 3 列（c0, c1, c2）（别名形式），生成表达式过程如下：

- 选择生成哪种类型表达式：
  - Constant：生成对应类型常数，返回；
  - Column：从（c0, c1, c2）选一个对应类型的列，返回；
  - Func：选一个对应返回类型的函数，然后递归为其生成儿子节点；

- 用动态的概率表来控制表达式树的结构，同生成算子树类似，不再赘述；

- 约束：生成过程感知类型，直接避免大部分无意义的表达式，如：
  pow(“abc”, “xxx”)，concat(23, 56)；
  接口：GenExpr(cols []Expr, tp TypeMask, validate ValidateExprFn) Expr
  TypeMask 直接表示需要的类型

随机生成表达式-更强的约束

动态的参数类型：比如 =, > 这些函数，通常两边的参数都为同一类类型，比如 string，number

接口：ArgTypeMask(i int, prvArgs []Expr) TypeMask

定义了 Validator 用来检测表达式是否有意义：

接口：type ValidateExprFn func(expr Expr) bool

已有的 Validator：

- MustContainsCols：表达式必须包含 Col，给 Filter.Where 和 Proj.Proj 用，用于过滤掉类似于 Select 1, 2, 3 from t where 4 and 5 and 6的Query；
- RejectConstants：如果某个 ScalarFunc 的所有参数都是常数，则拒绝，比如 where c0 > 10 and Ceil(25 * 0.88)；

可以为每个 Func 单独定义 Validator，如为 - 定义用来过滤 c0 - c0 这种无意义的表达式；

随机生成-算子表达式生成策略概览

- Filter{Where Expr}：常规方法 + MustContainsCols + RejectConstants 过滤
- Proj{Proj []Expr}：同上
- Order{OrderBy []Expr}：目前只考虑拿全部列排序，且不包裹函数
- Limit{Limit int}：随机生成一个 int
- Agg{AggExprs []Expr, GroupBy []Expr}：
  - AggExprs：随机选取 AggFunc 包裹在 Column 上
  - GroupBy：同 Order.OrderBy
- Join{JoinCond Expr}：生成同时包含左右儿子列的表达式，目前实现比较简单，只会生成 LCol op Rcol，op 包括 >, <, =, !=, <=, >=

随机生成-转换为 SQL

目前实现较为简单，方法为直接把子树当做子查询给父亲；

比如 Filter(c0 > 2)->Proj(c0+c1 AS c0)->Agg(c0, sum(c1) by c0)->Table(c0, c1)，会被转换为 SQL:

```
Select * from (
  select c0+c1 as c0 from (
    select c0, sum(c1) as c1 from (
      select cxxx as c0, cyyy as c1 from t
    ) t group by c0
  ) t
) t where c0 > 2;
```

每个算子实现了 ToSQL() string 的方法，可以修改它实现更“易读”的转换方式；

比如 Filter{Where} 不嵌套子查询，而是把列替换后直接把 Where 贴在儿子的 SQL 后；

目前觉得比较繁琐又不是很重要，就先没做；

随机生成-扩展性强

扩展算子（比如 window）：

1. 定义 Window 实现 Node 接口，实现 ToSQL() string, Columns() []Expr 等方法
2. 把 Window 加到概率表内让他能够出现在 LogicalPlan 中
3. 实现一个填充 Window 内部表达式的函数 FillWindow(w *Window)

扩展函数（比如 IsTrue）：

1. 将函数的信息添加到我们的函数列表即可

```
var FuncInfos = map[string]FuncInfo{
    FuncEQ:          { Name: FuncEQ, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncGE:          { Name: FuncGE, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncLE:          { Name: FuncLE, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncNE:          { Name: FuncNE, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncLT:          { Name: FuncLT, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncGT:          { Name: FuncGT, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncIsTrue:      { Name: FuncIsTrue, MinArgs: 1, MaxArgs: 1, ArgsTypes: nil, ReturnType: TypeDefault, Validate: nil},
    FuncIf:          { Name: FuncIf, MinArgs: 3, MaxArgs: 3, ArgsTypes: nil, ReturnType: TypeDefault, Validate: nil},
    FuncIfnull:      { Name: FuncIfnull, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeDefault, Validate: nil},
    FuncLcase:       { Name: FuncLcase, MinArgs: 1, MaxArgs: 1, ArgsTypes: []TypeMask{TypeString}, ReturnType: TypeString, Validate: nil},
    FuncLeft:        { Name: FuncLeft, MinArgs: 2, MaxArgs: 2, ArgsTypes: []TypeMask{TypeString, TypeNumber}, ReturnType: TypeString, Validate: nil},
    FuncRight:       { Name: FuncRight, MinArgs: 2, MaxArgs: 2, ArgsTypes: []TypeMask{TypeString, TypeNumber}, ReturnType: TypeString, Validate: nil},
```

随机生成-测试结果

测试表结构：

```
CREATE TABLE t (
    col_int int default null,
    col_double double default null,
    col_decimal decimal(40, 20) default null,
    col_string varchar(40) default null,
    col_datetime datetime default null,
    key(col_int),
    key(col_double),
    key(col_decimal),
    key(col_string),
    key(col_datetime),
    key(col_int, col_double),
    key(col_int, col_decimal),
    key(col_int, col_string),
    key(col_double, col_decimal)
);
```

跑了 1000 条 SQL，目前产生的错误情况如下：

都报错 & ErrCode 不一致	8
MySQL 报错 & TiDB 不报错	4
MySQL 不报错 & TiDB 报错	92
TiDB Panic	7
都不报错 & 结果不一致	34

MySQL 不报错 & TiDB 报错大多为字面量 out of range，比如：

```
Error 1690: DOUBLE value is out of range in 'pow(2147483647, 2147483647)'
```

结果不一致错误较多，是由于同种原因的 Bug 可能被多次触发；

随机生成-测试结果分析

Case 1：

```
SELECT * FROM ( SELECT c0, MIN(c1) AS c1, MAX(c2) AS c2 FROM (
 SELECT c0, COUNT(c1) AS c1, SUM(c2) AS c2 FROM (
SELECT * FROM (   SELECT col_double AS c0, col_string AS c1, col_datetime AS c2 FROM t) t WHERE c0
 ) AS t GROUP BY c0
) AS t GROUP BY c0) t ORDER BY c0, c1, c2;
```

TiDB 返回 47 行，MySQL 返回 93 行；

定位发现是 Coprocesser 处理下推条件 'WHERE c0' 有问题，过滤错了数据；

随机生成-测试结果分析

Case 2：

```
SELECT * FROM ( SELECT ACOS(c0) AS c0, (TO_BASE64(IF((c1 * (c3 - (c1 ^ 2013464221461210368))), UPPER('XtS38pz2hu'),
ABS(0.143))) OR 'z81Z4W') AS c1, c2 AS c2, LOWER(LTRIM(c0)) AS c3,
TO_BASE64(REPLACE(LCASE((0.000 + FLOOR(LOG(0.000)))), c0, POWER((ROUND(c1) ^ LOG(LN(-0.000))),
4377038545950721024))) AS c4, (IFNULL(c0, c1) / -451.353) AS c5, (OCT(RTRIM(c0)) >= LCASE(IFNULL(0.410, c1))) AS c6 FROM (
 SELECT t1.c0 AS c0,t2.c0 AS c1,t2.c1 AS c2,t2.c2 AS c3,t2.c3 AS c4 FROM (
  SELECT col_string AS c0 FROM t) AS t1, (
  SELECT col_int AS c0, col_double AS c1, col_string AS c2, col_datetime AS c3 FROM t) AS t2
  WHERE (t1.c0 < t2.c2)
) AS t) t ORDER BY c0, c1, c2, c3, c4, c5, c6 LIMIT 98;
```

MySQL 返回 98 行，TiDB 报错 constant -4.3238814911258294e+27 overflows bigint；

应该是浮点数和整数计算边界有问题；

随机生成-测试结果分析

Case 3：

```
SELECT * FROM ( SELECT t1.c0 AS c0,t2.c0 AS c1 FROM (
 SELECT col_string AS c0 FROM t) AS t1, (
SELECT * FROM (  SELECT SIN(IFNULL(c0, DAYOFWEEK('1999-12-01 12:50:06'))) AS c0 FROM (
  SELECT col_decimal AS c0 FROM t
) AS t) t WHERE ((REVERSE((c0 OR WEEKOFYEAR('2017-04-18 19:15:51'))) < ('2007-09-03 02:37:55' + (SIGN(0.000) | LN(c0)))) <=
(ABS(c0) AND c0))) AS t2
 WHERE (t1.c0 != t2.c0)) t ORDER BY c0, c1;
```

MySQL 返回空结果无报错，TiDB panic；

定位后发现是处理 c0 OR WEEKOFYEAR('2017-04-18 19:15:51') 改写逻辑有问题，导致子表达式没有被赋值，出现空指针；

随机生成-测试结果分析

Case 4：

```
SELECT * FROM ( SELECT c0, c1, c0 AS c2, c1 AS c3, MAX(c4) AS c4 FROM (
 SELECT t1.c0 AS c0,t1.c1 AS c1,t2.c0 AS c2,t2.c1 AS c3,t2.c2 AS c4 FROM (
  SELECT col_double AS c0, col_string AS c1 FROM t) AS t1, (
  SELECT col_int AS c0, col_string AS c1, col_datetime AS c2 FROM t) AS t2
  WHERE (t1.c1 != t2.c2)
) AS t GROUP BY c0, c1, c0, c1) t ORDER BY c0, c1, c2, c3, c4 LIMIT 24;
```

化简后的复现 SQL 如下：

```
SELECT count(*) from t t1, t t2 where t1.col_string != t2.col_datetime;
```

TiDB 返回 0，MySQL 返回 8184；

应该是计算 Hash 时 Join Key 在 string 和 datetime 类型下计算结果有误，导致没 Join 上；

随机生成-测试结果分析

Case 5：

```
SELECT * FROM ( SELECT c0, c1, MIN(c2) AS c2, MIN(c3) AS c3 FROM (
SELECT * FROM (  SELECT c0, c1, c2, AVG(c3) AS c3 FROM (
  SELECT col_int AS c0, col_double AS c1, col_decimal AS c2, col_datetime AS c3 FROM t
 ) AS t GROUP BY c0, c1, c2) t WHERE FLOOR((LN(IF(c1, -0.000, (c1 ^ COS(c2))))) < c2))
) AS t GROUP BY c0, c1) t ORDER BY c0, c1, c2, c3
```

MySQL 无结果，TiDB 输出 41 行；

定位原因在于 IF(c1, -0.000, (c1 ^ COS(c2)) 结果和 MySQL 不一致，主要为 IF 的第一个参数为浮点数时有问题；

随机生成-测试结果分析

Case 6：

```
SELECT * FROM (SELECT * FROM ( SELECT AVG(c0) AS c0 FROM (
 SELECT (c0 >= UPPER(REPEAT(UCASE(LOWER('UebLHwFSV')), ATAN(IFNULL(13.498, IF(0.000, '2026-08-14 10:51:21', c0)))))) AS c0
FROM (
  SELECT col_string AS c0 FROM t
 ) AS t
) AS t ) t WHERE CEIL(c0)) t ORDER BY c0
```

MySQL 返回 0.2473，TiDB 返回 1.0000;

定位是 c0 和 UPPER(...) 字符串比较有问题，导致结果和 MySQL 不一致；

PingCAP

Part III - 广度优先搜索 Query

广度优先搜索-想法来源

问题：怎么和已经有的错误 Query 结合起来起来搜索？

想法：把已知的错误 Query 当做搜索的起点，找到其附近相似的 Query？

⇒ 找到离错误 Query 最“近”（相似）的 N 个 Query

⇒ 这 N 个 Query 可以帮助我们：

1. 确认原错误 Query “真的”被修复了，增加信心 :D
2. 更有针对性，或许能在错误 Query 附近发现新的问题

⇒ 实现：Query + Transform Rules + BFS

广度优先搜索-初步设计

定义转换规则接口：

```
type TransformRule interface {
    OneStep(node Expr, ctx TransformContext) []Expr
}
```

表示某个表达式在该条规则作用下，向某些方向“走一步”能够到达的状态；

有了转换规则，我们就可以进行 BFS 了，很普通的 BFS 方法：

```
queue.PushBack(startNode)
for queue.Len() > 0 {
    node := queue.PopFront()
    for _, rule := range rules {
        queue.PushBack(rule.OneStep(node)...)
    }
}
```

当然还需要状态去重，判断结束条件等；

广度优先搜索-转换规则

规则设计的还比较简陋，不过框架是完整的，可以轻易添加删除规则；

目前规则有：

1. ConstantToColumn：把表达式树中的某个常量替换为列
2. ColumnToConstant：把表达式树中的某个列替换为常量
3. ReplaceChildToConstant：将表达式树中的某个子树替换为常量
4. ReplaceChildToColumn：将表达式树中的某个子树替换为列
5. ReplaceChildToFunc：将表达式树中的某个子树替换为新的 function

广度优先搜索-目前测试

以刚刚发现的错误 Query 作为 BFS 起点：

```
SELECT * FROM ( SELECT CEIL(TAN(IF(c1, c1, c1))) AS c0 FROM (
   SELECT col_string AS c0, col_datetime AS c1 FROM t
) AS t) t ORDER BY c0;
```

得到了如下的几个 Query，时间有限，目前还没发现新的问题 Query :(

只能作为 Demo 演示一下思想...

```
SELECT * FROM ( SELECT CEIL(-4824926468.651) AS c0 FROM (
  SELECT col_string AS c0, col_datetime AS c1 FROM t
 ) AS t) t ORDER BY c0
```

```
SELECT * FROM ( SELECT CEIL(c0) AS c0 FROM (
  SELECT col_string AS c0, col_datetime AS c1 FROM t
 ) AS t) t ORDER BY c0
```

```
SELECT * FROM ( SELECT CEIL(IF(c1, c1, c0)) AS c0 FROM (
  SELECT col_string AS c0, col_datetime AS c1 FROM t
 ) AS t) t ORDER BY c0
```

PingCAP

Part IV - 总结 & 未来工作

总结

- 随机生成 Query
  - 随机性强，覆盖面广，发现人想象不到的 bug
  - 避免手动构造 case 的枯燥劳动
  - 容易落地，已经发现不少问题
  - 框架完善，扩展性强，新增算子/表达式/约束都很容易
- 广度优先生成 Query
  - 框架完善，扩展性强，轻松添加规则
  - 针对性更强，更容易构造相似归因的 Query 集合进行有针对的测试

未来工作

- 随机生成 Query
  - 支持更多的算子 (window, view...)
  - 支持更多的表达式
  - 支持DML
- 广度优先生成 Query
  - 定义更多的 Rule，加大搜索空间
- 在搜索空间引入更多的搜索策略...(比如参考 gofuzz-testing 设计带反馈的 DFS 搜索算法)
- 拓宽框架应用场景，不仅局限于正确性验证，比如和 MySQL 对比进行性能测试，分析执行计划等

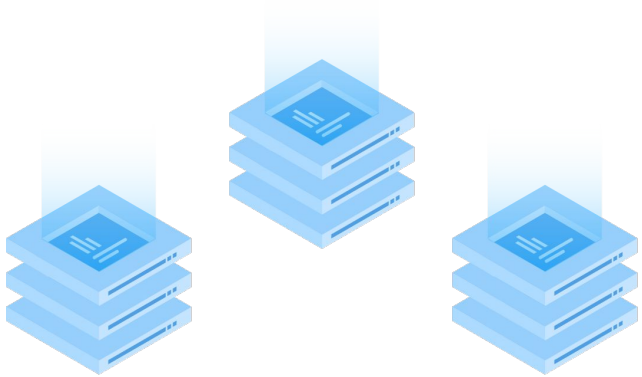
PingCAP

Thank You !

SQL Spider

Presented by Yuanjia Zhang/Zhongyang Guan/Xiangrui Meng

PingCAP TiDB



PingCAP

Part I - SQL Spider 介绍

项目介绍

项目名称：SQL Spider

项目介绍：提出两种 SQL 空间中的搜索策略（框架），提取有用的 SQL 进行测试

团队：祥瑞

团队成员：管仲洋，孟祥瑞，张原嘉

想法来源

ONCALL 中遇到的两个问题：

- 有 t1，t2，t1 有索引 (a, b, c)，在进行如下 join 时 t1.a=t2.a and t1.b=2 and t1.c=t2.c 会有 bug，原因总结：三列进行 IndexJoin 时，第二列为 constant 时会出错。
- 如果有某列 c 是 unsigned，表达式 IF(c>1, c, -c) 的返回类型会出错，原因总结：为 IF 判断返回类型时，如果第一列是 unsigned，但是其他参数又有可能是 signed 的时候会出错。

希望有工具能自动生成测试 Query，覆盖到诸如此类的 case。

已有方案

目前我们用的比较多的 Query 生成工具，RandGen：

- Example: SELECT _field[invariant] FROM _table[invariant] WHERE _table[invariant]._field[invariant] BETWEEN _digit[invariant] AND _digit[invariant] + _digit;
- Query 的模式需要提前定义好，随机能力有限，需要较多人为介入，无法在给定数据集上自动生成查询；
- 消耗人力去构造“可能出问题的” case，效率较低；

另一个出名的 Query 生成工具，SQLSmith：

- 官方不支持 MySQL（近期唐长老已经补充）；
- 生成的 Query 只包含最基本的运算符号比如 + - * / IsNull 等，不能包含较为复杂的 builtin-func 比如 TimeDiff 等；

预期目标

Table Schema + Database Schema ⇒ 合法的 SQL 空间
Table Schema + Database Schema + 一些约束 ⇒ 合法且有用的 SQL 空间

目标：实现一个工具，能在 Table 和 Database Schema 形成的合法 SQL 空间内，自动生成有用的 SQL，用于测试；

Table Schema 包括：

- Table Name
- Columns
  - Column Name
  - Column Type

Database Schema 包括：

- Operators(Join/Agg/Proj...)
- Expressions
  - ScalarFuncs
  - AggFuncs
  - Type system

实际产出

- 设计并实现了两种 SQL 空间中的搜索策略
  - 完全随机生成 Query
  - 广度优先搜索相邻 Query
- 在每种搜索策略上，提供了更通用的抽象，方便 扩展
- 发现多个 bug

PingCAP

Part II - 随机生成 Query

随机生成-初步设计

基本思路：随机出一个 LogicalPlan 或者 AST，然后反向改写成 Query；

- AST ⇒ SQL：不好感知数据类型，导致约束比较弱
- Logical Plan ⇒ SQL ✔️

Logical Plan 生成被拆解为：

1. 生成不带表达式的Logical Plan（空的算子树）
2. 为算子树填充表达式（完整的 Logical Plan）

再加上第三步：

3. 改写 Logcial Plan 为 Query

随机生成算子树-已经支持的算子

目前支持的算子：

- Filter{Where Expr}：多个条件用 And 链接，故只有一个 Expr
- Proj{Proj []Expr}
- Order{OrderBy []Expr}
- Limit{Limit int}
- Agg{AggExprs []Expr, GroupBy []Expr}
- Join{JoinCond Expr}：多个条件用 And 链接
- Table{Schema TableSchema, Selected []int}
    - Schema：用户输入的某个表结构
    - Selected：表示该表的哪几列被选中了

其他说明：

- Table 表示从表读数据，只能为叶节点，他没有儿子；
- Join 有两个儿子，其他算子除 Table 只有一个儿子；

随机生成算子树-生成算法

递归向下的生成：

1. 随机的选择当前的节点类型：
   a. 如果为 Table，则生成后直接返回；
   b. 如果不为 Table，生成后递归的生成其儿子节点；

2. 用概率表控制其基本结构：
   为每种算子设定出现的概率，用于大概控制算子树的结构；
   比如 Table 出现的概率为 20%，则某一条路径深度超过 5 的概率为 (1-20%)^5；

3. 动态的条件约束，产生更合理的结构：
   a. 根据当前的深度，动态的改变每种算子出现的概率，比如“当深度大于 6 时，Table 的概率为 100%”；
   b. 记录当前路径的已有算子信息，过滤掉无用的结构，比如“...Filter->Filter->...”这种结构；

随机生成表达式-已经支持的表达式

目前支持的表达式：

- 已支持的类型：Int, Real, Decimal, String, Datetime
- 已支持的表达式：
  - Constant
  - Column
  - Funcs
    - ScalarFunc: 82 个（剩下的简单调试也能轻松导入）
    - AggFunc: 5 个（AVG/SUM/COUNT/MAX/MIN）

随机生成表达式-生成算法

假设我们现在准备为某个算子生成对应的表达式，如为 Filter 生成 Where，已知 Filter 的子节点会返回 3 列（c0, c1, c2）（别名形式），生成表达式过程如下：

- 选择生成哪种类型表达式：
  - Constant：生成对应类型常数，返回；
  - Column：从（c0, c1, c2）选一个对应类型的列，返回；
  - Func：选一个对应返回类型的函数，然后递归为其生成儿子节点；

- 用动态的概率表来控制表达式树的结构，同生成算子树类似，不再赘述；

- 约束：生成过程感知类型，直接避免大部分无意义的表达式，如：
  pow(“abc”, “xxx”)，concat(23, 56)；
  接口：GenExpr(cols []Expr, tp TypeMask, validate ValidateExprFn) Expr
  TypeMask 直接表示需要的类型

随机生成表达式-更强的约束

动态的参数类型：比如 =, > 这些函数，通常两边的参数都为同一类类型，比如 string，number

接口：ArgTypeMask(i int, prvArgs []Expr) TypeMask

定义了 Validator 用来检测表达式是否有意义：

接口：type ValidateExprFn func(expr Expr) bool

已有的 Validator：

- MustContainsCols：表达式必须包含 Col，给 Filter.Where 和 Proj.Proj 用，用于过滤掉类似于 Select 1, 2, 3 from t where 4 and 5 and 6的Query；
- RejectConstants：如果某个 ScalarFunc 的所有参数都是常数，则拒绝，比如 where c0 > 10 and Ceil(25 * 0.88)；

可以为每个 Func 单独定义 Validator，如为 - 定义用来过滤 c0 - c0 这种无意义的表达式；

随机生成-算子表达式生成策略概览

- Filter{Where Expr}：常规方法 + MustContainsCols + RejectConstants 过滤
- Proj{Proj []Expr}：同上
- Order{OrderBy []Expr}：目前只考虑拿全部列排序，且不包裹函数
- Limit{Limit int}：随机生成一个 int
- Agg{AggExprs []Expr, GroupBy []Expr}：
  - AggExprs：随机选取 AggFunc 包裹在 Column 上
  - GroupBy：同 Order.OrderBy
- Join{JoinCond Expr}：生成同时包含左右儿子列的表达式，目前实现比较简单，只会生成 LCol op Rcol，op 包括 >, <, =, !=, <=, >=

随机生成-转换为 SQL

目前实现较为简单，方法为直接把子树当做子查询给父亲；

比如 Filter(c0 > 2)->Proj(c0+c1 AS c0)->Agg(c0, sum(c1) by c0)->Table(c0, c1)，会被转换为 SQL:

```
Select * from (
  select c0+c1 as c0 from (
    select c0, sum(c1) as c1 from (
      select cxxx as c0, cyyy as c1 from t
    ) t group by c0
  ) t
) t where c0 > 2;
```

每个算子实现了 ToSQL() string 的方法，可以修改它实现更“易读”的转换方式；

比如 Filter{Where} 不嵌套子查询，而是把列替换后直接把 Where 贴在儿子的 SQL 后；

目前觉得比较繁琐又不是很重要，就先没做；

随机生成-扩展性强

扩展算子（比如 window）：

1. 定义 Window 实现 Node 接口，实现 ToSQL() string, Columns() []Expr 等方法
2. 把 Window 加到概率表内让他能够出现在 LogicalPlan 中
3. 实现一个填充 Window 内部表达式的函数 FillWindow(w *Window)

扩展函数（比如 IsTrue）：

1. 将函数的信息添加到我们的函数列表即可

```
var FuncInfos = map[string]FuncInfo{
    FuncEQ:          { Name: FuncEQ, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncGE:          { Name: FuncGE, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncLE:          { Name: FuncLE, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncNE:          { Name: FuncNE, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncLT:          { Name: FuncLT, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncGT:          { Name: FuncGT, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncIsTrue:      { Name: FuncIsTrue, MinArgs: 1, MaxArgs: 1, ArgsTypes: nil, ReturnType: TypeDefault, Validate: nil},
    FuncIf:          { Name: FuncIf, MinArgs: 3, MaxArgs: 3, ArgsTypes: nil, ReturnType: TypeDefault, Validate: nil},
    FuncIfnull:      { Name: FuncIfnull, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeDefault, Validate: nil},
    FuncLcase:       { Name: FuncLcase, MinArgs: 1, MaxArgs: 1, ArgsTypes: []TypeMask{TypeString}, ReturnType: TypeString, Validate: nil},
    FuncLeft:        { Name: FuncLeft, MinArgs: 2, MaxArgs: 2, ArgsTypes: []TypeMask{TypeString, TypeNumber}, ReturnType: TypeString, Validate: nil},
    FuncRight:       { Name: FuncRight, MinArgs: 2, MaxArgs: 2, ArgsTypes: []TypeMask{TypeString, TypeNumber}, ReturnType: TypeString, Validate: nil},
```

随机生成-测试结果

测试表结构：

```
CREATE TABLE t (
    col_int int default null,
    col_double double default null,
    col_decimal decimal(40, 20) default null,
    col_string varchar(40) default null,
    col_datetime datetime default null,
    key(col_int),
    key(col_double),
    key(col_decimal),
    key(col_string),
    key(col_datetime),
    key(col_int, col_double),
    key(col_int, col_decimal),
    key(col_int, col_string),
    key(col_double, col_decimal)
);
```

跑了 1000 条 SQL，目前产生的错误情况如下：

都报错 & ErrCode 不一致	8
MySQL 报错 & TiDB 不报错	4
MySQL 不报错 & TiDB 报错	92
TiDB Panic	7
都不报错 & 结果不一致	34

MySQL 不报错 & TiDB 报错大多为字面量 out of range，比如：

```
Error 1690: DOUBLE value is out of range in 'pow(2147483647, 2147483647)'
```

结果不一致错误较多，是由于同种原因的 Bug 可能被多次触发；

随机生成-测试结果分析

Case 1：

```
SELECT * FROM ( SELECT c0, MIN(c1) AS c1, MAX(c2) AS c2 FROM (
 SELECT c0, COUNT(c1) AS c1, SUM(c2) AS c2 FROM (
SELECT * FROM (   SELECT col_double AS c0, col_string AS c1, col_datetime AS c2 FROM t) t WHERE c0
 ) AS t GROUP BY c0
) AS t GROUP BY c0) t ORDER BY c0, c1, c2;
```

TiDB 返回 47 行，MySQL 返回 93 行；

定位发现是 Coprocesser 处理下推条件 'WHERE c0' 有问题，过滤错了数据；

随机生成-测试结果分析

Case 2：

```
SELECT * FROM ( SELECT ACOS(c0) AS c0, (TO_BASE64(IF((c1 * (c3 - (c1 ^ 2013464221461210368))), UPPER('XtS38pz2hu'),
ABS(0.143))) OR 'z81Z4W') AS c1, c2 AS c2, LOWER(LTRIM(c0)) AS c3,
TO_BASE64(REPLACE(LCASE((0.000 + FLOOR(LOG(0.000)))), c0, POWER((ROUND(c1) ^ LOG(LN(-0.000))),
4377038545950721024))) AS c4, (IFNULL(c0, c1) / -451.353) AS c5, (OCT(RTRIM(c0)) >= LCASE(IFNULL(0.410, c1))) AS c6 FROM (
 SELECT t1.c0 AS c0,t2.c0 AS c1,t2.c1 AS c2,t2.c2 AS c3,t2.c3 AS c4 FROM (
  SELECT col_string AS c0 FROM t) AS t1, (
  SELECT col_int AS c0, col_double AS c1, col_string AS c2, col_datetime AS c3 FROM t) AS t2
  WHERE (t1.c0 < t2.c2)
) AS t) t ORDER BY c0, c1, c2, c3, c4, c5, c6 LIMIT 98;
```

MySQL 返回 98 行，TiDB 报错 constant -4.3238814911258294e+27 overflows bigint；

应该是浮点数和整数计算边界有问题；

随机生成-测试结果分析

Case 3：

```
SELECT * FROM ( SELECT t1.c0 AS c0,t2.c0 AS c1 FROM (
 SELECT col_string AS c0 FROM t) AS t1, (
SELECT * FROM (  SELECT SIN(IFNULL(c0, DAYOFWEEK('1999-12-01 12:50:06'))) AS c0 FROM (
  SELECT col_decimal AS c0 FROM t
) AS t) t WHERE ((REVERSE((c0 OR WEEKOFYEAR('2017-04-18 19:15:51'))) < ('2007-09-03 02:37:55' + (SIGN(0.000) | LN(c0)))) <=
(ABS(c0) AND c0))) AS t2
 WHERE (t1.c0 != t2.c0)) t ORDER BY c0, c1;
```

MySQL 返回空结果无报错，TiDB panic；

定位后发现是处理 c0 OR WEEKOFYEAR('2017-04-18 19:15:51') 改写逻辑有问题，导致子表达式没有被赋值，出现空指针；

随机生成-测试结果分析

Case 4：

```
SELECT * FROM ( SELECT c0, c1, c0 AS c2, c1 AS c3, MAX(c4) AS c4 FROM (
 SELECT t1.c0 AS c0,t1.c1 AS c1,t2.c0 AS c2,t2.c1 AS c3,t2.c2 AS c4 FROM (
  SELECT col_double AS c0, col_string AS c1 FROM t) AS t1, (
  SELECT col_int AS c0, col_string AS c1, col_datetime AS c2 FROM t) AS t2
  WHERE (t1.c1 != t2.c2)
) AS t GROUP BY c0, c1, c0, c1) t ORDER BY c0, c1, c2, c3, c4 LIMIT 24;
```

化简后的复现 SQL 如下：

```
SELECT count(*) from t t1, t t2 where t1.col_string != t2.col_datetime;
```

TiDB 返回 0，MySQL 返回 8184；

应该是计算 Hash 时 Join Key 在 string 和 datetime 类型下计算结果有误，导致没 Join 上；

随机生成-测试结果分析

Case 5：

```
SELECT * FROM ( SELECT c0, c1, MIN(c2) AS c2, MIN(c3) AS c3 FROM (
SELECT * FROM (  SELECT c0, c1, c2, AVG(c3) AS c3 FROM (
  SELECT col_int AS c0, col_double AS c1, col_decimal AS c2, col_datetime AS c3 FROM t
 ) AS t GROUP BY c0, c1, c2) t WHERE FLOOR((LN(IF(c1, -0.000, (c1 ^ COS(c2))))) < c2))
) AS t GROUP BY c0, c1) t ORDER BY c0, c1, c2, c3
```

MySQL 无结果，TiDB 输出 41 行；

定位原因在于 IF(c1, -0.000, (c1 ^ COS(c2)) 结果和 MySQL 不一致，主要为 IF 的第一个参数为浮点数时有问题；

随机生成-测试结果分析

Case 6：

```
SELECT * FROM (SELECT * FROM ( SELECT AVG(c0) AS c0 FROM (
 SELECT (c0 >= UPPER(REPEAT(UCASE(LOWER('UebLHwFSV')), ATAN(IFNULL(13.498, IF(0.000, '2026-08-14 10:51:21', c0)))))) AS c0
FROM (
  SELECT col_string AS c0 FROM t
 ) AS t
) AS t ) t WHERE CEIL(c0)) t ORDER BY c0
```

MySQL 返回 0.2473，TiDB 返回 1.0000;

定位是 c0 和 UPPER(...) 字符串比较有问题，导致结果和 MySQL 不一致；

PingCAP

Part III - 广度优先搜索 Query

广度优先搜索-想法来源

问题：怎么和已经有的错误 Query 结合起来起来搜索？

想法：把已知的错误 Query 当做搜索的起点，找到其附近相似的 Query？

⇒ 找到离错误 Query 最“近”（相似）的 N 个 Query

⇒ 这 N 个 Query 可以帮助我们：

1. 确认原错误 Query “真的”被修复了，增加信心 :D
2. 更有针对性，或许能在错误 Query 附近发现新的问题

⇒ 实现：Query + Transform Rules + BFS

广度优先搜索-初步设计

定义转换规则接口：

```
type TransformRule interface {
    OneStep(node Expr, ctx TransformContext) []Expr
}
```

表示某个表达式在该条规则作用下，向某些方向“走一步”能够到达的状态；

有了转换规则，我们就可以进行 BFS 了，很普通的 BFS 方法：

```
queue.PushBack(startNode)
for queue.Len() > 0 {
    node := queue.PopFront()
    for _, rule := range rules {
        queue.PushBack(rule.OneStep(node)...)
    }
}
```

当然还需要状态去重，判断结束条件等；

广度优先搜索-转换规则

规则设计的还比较简陋，不过框架是完整的，可以轻易添加删除规则；

目前规则有：

1. ConstantToColumn：把表达式树中的某个常量替换为列
2. ColumnToConstant：把表达式树中的某个列替换为常量
3. ReplaceChildToConstant：将表达式树中的某个子树替换为常量
4. ReplaceChildToColumn：将表达式树中的某个子树替换为列
5. ReplaceChildToFunc：将表达式树中的某个子树替换为新的 function

广度优先搜索-目前测试

以刚刚发现的错误 Query 作为 BFS 起点：

```
SELECT * FROM ( SELECT CEIL(TAN(IF(c1, c1, c1))) AS c0 FROM (
   SELECT col_string AS c0, col_datetime AS c1 FROM t
) AS t) t ORDER BY c0;
```

得到了如下的几个 Query，时间有限，目前还没发现新的问题 Query :(

只能作为 Demo 演示一下思想...

```
SELECT * FROM ( SELECT CEIL(-4824926468.651) AS c0 FROM (
  SELECT col_string AS c0, col_datetime AS c1 FROM t
 ) AS t) t ORDER BY c0
```

```
SELECT * FROM ( SELECT CEIL(c0) AS c0 FROM (
  SELECT col_string AS c0, col_datetime AS c1 FROM t
 ) AS t) t ORDER BY c0
```

```
SELECT * FROM ( SELECT CEIL(IF(c1, c1, c0)) AS c0 FROM (
  SELECT col_string AS c0, col_datetime AS c1 FROM t
 ) AS t) t ORDER BY c0
```

PingCAP

Part IV - 总结 & 未来工作

总结

- 随机生成 Query
  - 随机性强，覆盖面广，发现人想象不到的 bug
  - 避免手动构造 case 的枯燥劳动
  - 容易落地，已经发现不少问题
  - 框架完善，扩展性强，新增算子/表达式/约束都很容易
- 广度优先生成 Query
  - 框架完善，扩展性强，轻松添加规则
  - 针对性更强，更容易构造相似归因的 Query 集合进行有针对的测试

未来工作

- 随机生成 Query
  - 支持更多的算子 (window, view...)
  - 支持更多的表达式
  - 支持DML
- 广度优先生成 Query
  - 定义更多的 Rule，加大搜索空间
- 在搜索空间引入更多的搜索策略...(比如参考 gofuzz-testing 设计带反馈的 DFS 搜索算法)
- 拓宽框架应用场景，不仅局限于正确性验证，比如和 MySQL 对比进行性能测试，分析执行计划等

PingCAP

Thank You !

PingCAP

Part I - SQL Spider 介绍

项目介绍

项目名称：SQL Spider

项目介绍：提出两种 SQL 空间中的搜索策略（框架），提取有用的 SQL 进行测试

团队：祥瑞

团队成员：管仲洋，孟祥瑞，张原嘉

想法来源

ONCALL 中遇到的两个问题：

- 有 t1，t2，t1 有索引 (a, b, c)，在进行如下 join 时 t1.a=t2.a and t1.b=2 and t1.c=t2.c 会有 bug，原因总结：三列进行 IndexJoin 时，第二列为 constant 时会出错。
- 如果有某列 c 是 unsigned，表达式 IF(c>1, c, -c) 的返回类型会出错，原因总结：为 IF 判断返回类型时，如果第一列是 unsigned，但是其他参数又有可能是 signed 的时候会出错。

希望有工具能自动生成测试 Query，覆盖到诸如此类的 case。

已有方案

目前我们用的比较多的 Query 生成工具，RandGen：

- Example: SELECT _field[invariant] FROM _table[invariant] WHERE _table[invariant]._field[invariant] BETWEEN _digit[invariant] AND _digit[invariant] + _digit;
- Query 的模式需要提前定义好，随机能力有限，需要较多人为介入，无法在给定数据集上自动生成查询；
- 消耗人力去构造“可能出问题的” case，效率较低；

另一个出名的 Query 生成工具，SQLSmith：

- 官方不支持 MySQL（近期唐长老已经补充）；
- 生成的 Query 只包含最基本的运算符号比如 + - * / IsNull 等，不能包含较为复杂的 builtin-func 比如 TimeDiff 等；

预期目标

Table Schema + Database Schema ⇒ 合法的 SQL 空间
Table Schema + Database Schema + 一些约束 ⇒ 合法且有用的 SQL 空间

目标：实现一个工具，能在 Table 和 Database Schema 形成的合法 SQL 空间内，自动生成有用的 SQL，用于测试；

Table Schema 包括：

- Table Name
- Columns
  - Column Name
  - Column Type

Database Schema 包括：

- Operators(Join/Agg/Proj...)
- Expressions
  - ScalarFuncs
  - AggFuncs
  - Type system

实际产出

- 设计并实现了两种 SQL 空间中的搜索策略
  - 完全随机生成 Query
  - 广度优先搜索相邻 Query
- 在每种搜索策略上，提供了更通用的抽象，方便 扩展
- 发现多个 bug

PingCAP

Part II - 随机生成 Query

随机生成-初步设计

基本思路：随机出一个 LogicalPlan 或者 AST，然后反向改写成 Query；

- AST ⇒ SQL：不好感知数据类型，导致约束比较弱
- Logical Plan ⇒ SQL ✔️

Logical Plan 生成被拆解为：

1. 生成不带表达式的Logical Plan（空的算子树）
2. 为算子树填充表达式（完整的 Logical Plan）

再加上第三步：

3. 改写 Logcial Plan 为 Query

随机生成算子树-已经支持的算子

目前支持的算子：

- Filter{Where Expr}：多个条件用 And 链接，故只有一个 Expr
- Proj{Proj []Expr}
- Order{OrderBy []Expr}
- Limit{Limit int}
- Agg{AggExprs []Expr, GroupBy []Expr}
- Join{JoinCond Expr}：多个条件用 And 链接
- Table{Schema TableSchema, Selected []int}
    - Schema：用户输入的某个表结构
    - Selected：表示该表的哪几列被选中了

其他说明：

- Table 表示从表读数据，只能为叶节点，他没有儿子；
- Join 有两个儿子，其他算子除 Table 只有一个儿子；

随机生成算子树-生成算法

递归向下的生成：

1. 随机的选择当前的节点类型：
   a. 如果为 Table，则生成后直接返回；
   b. 如果不为 Table，生成后递归的生成其儿子节点；

2. 用概率表控制其基本结构：
   为每种算子设定出现的概率，用于大概控制算子树的结构；
   比如 Table 出现的概率为 20%，则某一条路径深度超过 5 的概率为 (1-20%)^5；

3. 动态的条件约束，产生更合理的结构：
   a. 根据当前的深度，动态的改变每种算子出现的概率，比如“当深度大于 6 时，Table 的概率为 100%”；
   b. 记录当前路径的已有算子信息，过滤掉无用的结构，比如“...Filter->Filter->...”这种结构；

随机生成表达式-已经支持的表达式

目前支持的表达式：

- 已支持的类型：Int, Real, Decimal, String, Datetime
- 已支持的表达式：
  - Constant
  - Column
  - Funcs
    - ScalarFunc: 82 个（剩下的简单调试也能轻松导入）
    - AggFunc: 5 个（AVG/SUM/COUNT/MAX/MIN）

随机生成表达式-生成算法

假设我们现在准备为某个算子生成对应的表达式，如为 Filter 生成 Where，已知 Filter 的子节点会返回 3 列（c0, c1, c2）（别名形式），生成表达式过程如下：

- 选择生成哪种类型表达式：
  - Constant：生成对应类型常数，返回；
  - Column：从（c0, c1, c2）选一个对应类型的列，返回；
  - Func：选一个对应返回类型的函数，然后递归为其生成儿子节点；

- 用动态的概率表来控制表达式树的结构，同生成算子树类似，不再赘述；

- 约束：生成过程感知类型，直接避免大部分无意义的表达式，如：
  pow(“abc”, “xxx”)，concat(23, 56)；
  接口：GenExpr(cols []Expr, tp TypeMask, validate ValidateExprFn) Expr
  TypeMask 直接表示需要的类型

随机生成表达式-更强的约束

动态的参数类型：比如 =, > 这些函数，通常两边的参数都为同一类类型，比如 string，number

接口：ArgTypeMask(i int, prvArgs []Expr) TypeMask

定义了 Validator 用来检测表达式是否有意义：

接口：type ValidateExprFn func(expr Expr) bool

已有的 Validator：

- MustContainsCols：表达式必须包含 Col，给 Filter.Where 和 Proj.Proj 用，用于过滤掉类似于 Select 1, 2, 3 from t where 4 and 5 and 6的Query；
- RejectConstants：如果某个 ScalarFunc 的所有参数都是常数，则拒绝，比如 where c0 > 10 and Ceil(25 * 0.88)；

可以为每个 Func 单独定义 Validator，如为 - 定义用来过滤 c0 - c0 这种无意义的表达式；

随机生成-算子表达式生成策略概览

- Filter{Where Expr}：常规方法 + MustContainsCols + RejectConstants 过滤
- Proj{Proj []Expr}：同上
- Order{OrderBy []Expr}：目前只考虑拿全部列排序，且不包裹函数
- Limit{Limit int}：随机生成一个 int
- Agg{AggExprs []Expr, GroupBy []Expr}：
  - AggExprs：随机选取 AggFunc 包裹在 Column 上
  - GroupBy：同 Order.OrderBy
- Join{JoinCond Expr}：生成同时包含左右儿子列的表达式，目前实现比较简单，只会生成 LCol op Rcol，op 包括 >, <, =, !=, <=, >=

随机生成-转换为 SQL

目前实现较为简单，方法为直接把子树当做子查询给父亲；

比如 Filter(c0 > 2)->Proj(c0+c1 AS c0)->Agg(c0, sum(c1) by c0)->Table(c0, c1)，会被转换为 SQL:

```
Select * from (
  select c0+c1 as c0 from (
    select c0, sum(c1) as c1 from (
      select cxxx as c0, cyyy as c1 from t
    ) t group by c0
  ) t
) t where c0 > 2;
```

每个算子实现了 ToSQL() string 的方法，可以修改它实现更“易读”的转换方式；

比如 Filter{Where} 不嵌套子查询，而是把列替换后直接把 Where 贴在儿子的 SQL 后；

目前觉得比较繁琐又不是很重要，就先没做；

随机生成-扩展性强

扩展算子（比如 window）：

1. 定义 Window 实现 Node 接口，实现 ToSQL() string, Columns() []Expr 等方法
2. 把 Window 加到概率表内让他能够出现在 LogicalPlan 中
3. 实现一个填充 Window 内部表达式的函数 FillWindow(w *Window)

扩展函数（比如 IsTrue）：

1. 将函数的信息添加到我们的函数列表即可

```
var FuncInfos = map[string]FuncInfo{
    FuncEQ:          { Name: FuncEQ, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncGE:          { Name: FuncGE, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncLE:          { Name: FuncLE, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncNE:          { Name: FuncNE, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncLT:          { Name: FuncLT, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncGT:          { Name: FuncGT, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeNumber, Validate: nil},
    FuncIsTrue:      { Name: FuncIsTrue, MinArgs: 1, MaxArgs: 1, ArgsTypes: nil, ReturnType: TypeDefault, Validate: nil},
    FuncIf:          { Name: FuncIf, MinArgs: 3, MaxArgs: 3, ArgsTypes: nil, ReturnType: TypeDefault, Validate: nil},
    FuncIfnull:      { Name: FuncIfnull, MinArgs: 2, MaxArgs: 2, ArgsTypes: nil, ReturnType: TypeDefault, Validate: nil},
    FuncLcase:       { Name: FuncLcase, MinArgs: 1, MaxArgs: 1, ArgsTypes: []TypeMask{TypeString}, ReturnType: TypeString, Validate: nil},
    FuncLeft:        { Name: FuncLeft, MinArgs: 2, MaxArgs: 2, ArgsTypes: []TypeMask{TypeString, TypeNumber}, ReturnType: TypeString, Validate: nil},
    FuncRight:       { Name: FuncRight, MinArgs: 2, MaxArgs: 2, ArgsTypes: []TypeMask{TypeString, TypeNumber}, ReturnType: TypeString, Validate: nil},
```

随机生成-测试结果

测试表结构：

```
CREATE TABLE t (
    col_int int default null,
    col_double double default null,
    col_decimal decimal(40, 20) default null,
    col_string varchar(40) default null,
    col_datetime datetime default null,
    key(col_int),
    key(col_double),
    key(col_decimal),
    key(col_string),
    key(col_datetime),
    key(col_int, col_double),
    key(col_int, col_decimal),
    key(col_int, col_string),
    key(col_double, col_decimal)
);
```

跑了 1000 条 SQL，目前产生的错误情况如下：

都报错 & ErrCode 不一致	8
MySQL 报错 & TiDB 不报错	4
MySQL 不报错 & TiDB 报错	92
TiDB Panic	7
都不报错 & 结果不一致	34

MySQL 不报错 & TiDB 报错大多为字面量 out of range，比如：

```
Error 1690: DOUBLE value is out of range in 'pow(2147483647, 2147483647)'
```

结果不一致错误较多，是由于同种原因的 Bug 可能被多次触发；

随机生成-测试结果分析

Case 1：

```
SELECT * FROM ( SELECT c0, MIN(c1) AS c1, MAX(c2) AS c2 FROM (
 SELECT c0, COUNT(c1) AS c1, SUM(c2) AS c2 FROM (
SELECT * FROM (   SELECT col_double AS c0, col_string AS c1, col_datetime AS c2 FROM t) t WHERE c0
 ) AS t GROUP BY c0
) AS t GROUP BY c0) t ORDER BY c0, c1, c2;
```

TiDB 返回 47 行，MySQL 返回 93 行；

定位发现是 Coprocesser 处理下推条件 'WHERE c0' 有问题，过滤错了数据；

随机生成-测试结果分析

Case 2：

```
SELECT * FROM ( SELECT ACOS(c0) AS c0, (TO_BASE64(IF((c1 * (c3 - (c1 ^ 2013464221461210368))), UPPER('XtS38pz2hu'),
ABS(0.143))) OR 'z81Z4W') AS c1, c2 AS c2, LOWER(LTRIM(c0)) AS c3,
TO_BASE64(REPLACE(LCASE((0.000 + FLOOR(LOG(0.000)))), c0, POWER((ROUND(c1) ^ LOG(LN(-0.000))),
4377038545950721024))) AS c4, (IFNULL(c0, c1) / -451.353) AS c5, (OCT(RTRIM(c0)) >= LCASE(IFNULL(0.410, c1))) AS c6 FROM (
 SELECT t1.c0 AS c0,t2.c0 AS c1,t2.c1 AS c2,t2.c2 AS c3,t2.c3 AS c4 FROM (
  SELECT col_string AS c0 FROM t) AS t1, (
  SELECT col_int AS c0, col_double AS c1, col_string AS c2, col_datetime AS c3 FROM t) AS t2
  WHERE (t1.c0 < t2.c2)
) AS t) t ORDER BY c0, c1, c2, c3, c4, c5, c6 LIMIT 98;
```

MySQL 返回 98 行，TiDB 报错 constant -4.3238814911258294e+27 overflows bigint；

应该是浮点数和整数计算边界有问题；

随机生成-测试结果分析

Case 3：

```
SELECT * FROM ( SELECT t1.c0 AS c0,t2.c0 AS c1 FROM (
 SELECT col_string AS c0 FROM t) AS t1, (
SELECT * FROM (  SELECT SIN(IFNULL(c0, DAYOFWEEK('1999-12-01 12:50:06'))) AS c0 FROM (
  SELECT col_decimal AS c0 FROM t
) AS t) t WHERE ((REVERSE((c0 OR WEEKOFYEAR('2017-04-18 19:15:51'))) < ('2007-09-03 02:37:55' + (SIGN(0.000) | LN(c0)))) <=
(ABS(c0) AND c0))) AS t2
 WHERE (t1.c0 != t2.c0)) t ORDER BY c0, c1;
```

MySQL 返回空结果无报错，TiDB panic；

定位后发现是处理 c0 OR WEEKOFYEAR('2017-04-18 19:15:51') 改写逻辑有问题，导致子表达式没有被赋值，出现空指针；

随机生成-测试结果分析

Case 4：

```
SELECT * FROM ( SELECT c0, c1, c0 AS c2, c1 AS c3, MAX(c4) AS c4 FROM (
 SELECT t1.c0 AS c0,t1.c1 AS c1,t2.c0 AS c2,t2.c1 AS c3,t2.c2 AS c4 FROM (
  SELECT col_double AS c0, col_string AS c1 FROM t) AS t1, (
  SELECT col_int AS c0, col_string AS c1, col_datetime AS c2 FROM t) AS t2
  WHERE (t1.c1 != t2.c2)
) AS t GROUP BY c0, c1, c0, c1) t ORDER BY c0, c1, c2, c3, c4 LIMIT 24;
```

化简后的复现 SQL 如下：

```
SELECT count(*) from t t1, t t2 where t1.col_string != t2.col_datetime;
```

TiDB 返回 0，MySQL 返回 8184；

应该是计算 Hash 时 Join Key 在 string 和 datetime 类型下计算结果有误，导致没 Join 上；

随机生成-测试结果分析

Case 5：

```
SELECT * FROM ( SELECT c0, c1, MIN(c2) AS c2, MIN(c3) AS c3 FROM (
SELECT * FROM (  SELECT c0, c1, c2, AVG(c3) AS c3 FROM (
  SELECT col_int AS c0, col_double AS c1, col_decimal AS c2, col_datetime AS c3 FROM t
 ) AS t GROUP BY c0, c1, c2) t WHERE FLOOR((LN(IF(c1, -0.000, (c1 ^ COS(c2))))) < c2))
) AS t GROUP BY c0, c1) t ORDER BY c0, c1, c2, c3
```

MySQL 无结果，TiDB 输出 41 行；

定位原因在于 IF(c1, -0.000, (c1 ^ COS(c2)) 结果和 MySQL 不一致，主要为 IF 的第一个参数为浮点数时有问题；

随机生成-测试结果分析

Case 6：

```
SELECT * FROM (SELECT * FROM ( SELECT AVG(c0) AS c0 FROM (
 SELECT (c0 >= UPPER(REPEAT(UCASE(LOWER('UebLHwFSV')), ATAN(IFNULL(13.498, IF(0.000, '2026-08-14 10:51:21', c0)))))) AS c0
FROM (
  SELECT col_string AS c0 FROM t
 ) AS t
) AS t ) t WHERE CEIL(c0)) t ORDER BY c0
```

MySQL 返回 0.2473，TiDB 返回 1.0000;

定位是 c0 和 UPPER(...) 字符串比较有问题，导致结果和 MySQL 不一致；

PingCAP

Part III - 广度优先搜索 Query

广度优先搜索-想法来源

问题：怎么和已经有的错误 Query 结合起来起来搜索？

想法：把已知的错误 Query 当做搜索的起点，找到其附近相似的 Query？

⇒ 找到离错误 Query 最“近”（相似）的 N 个 Query

⇒ 这 N 个 Query 可以帮助我们：

1. 确认原错误 Query “真的”被修复了，增加信心 :D
2. 更有针对性，或许能在错误 Query 附近发现新的问题

⇒ 实现：Query + Transform Rules + BFS

广度优先搜索-初步设计

定义转换规则接口：

```
type TransformRule interface {
    OneStep(node Expr, ctx TransformContext) []Expr
}
```

表示某个表达式在该条规则作用下，向某些方向“走一步”能够到达的状态；

有了转换规则，我们就可以进行 BFS 了，很普通的 BFS 方法：

```
queue.PushBack(startNode)
for queue.Len() > 0 {
    node := queue.PopFront()
    for _, rule := range rules {
        queue.PushBack(rule.OneStep(node)...)
    }
}
```

当然还需要状态去重，判断结束条件等；

广度优先搜索-转换规则

规则设计的还比较简陋，不过框架是完整的，可以轻易添加删除规则；

目前规则有：

1. ConstantToColumn：把表达式树中的某个常量替换为列
2. ColumnToConstant：把表达式树中的某个列替换为常量
3. ReplaceChildToConstant：将表达式树中的某个子树替换为常量
4. ReplaceChildToColumn：将表达式树中的某个子树替换为列
5. ReplaceChildToFunc：将表达式树中的某个子树替换为新的 function

广度优先搜索-目前测试

以刚刚发现的错误 Query 作为 BFS 起点：

```
SELECT * FROM ( SELECT CEIL(TAN(IF(c1, c1, c1))) AS c0 FROM (
   SELECT col_string AS c0, col_datetime AS c1 FROM t
) AS t) t ORDER BY c0;
```

得到了如下的几个 Query，时间有限，目前还没发现新的问题 Query :(

只能作为 Demo 演示一下思想...

```
SELECT * FROM ( SELECT CEIL(-4824926468.651) AS c0 FROM (
  SELECT col_string AS c0, col_datetime AS c1 FROM t
 ) AS t) t ORDER BY c0
```

```
SELECT * FROM ( SELECT CEIL(c0) AS c0 FROM (
  SELECT col_string AS c0, col_datetime AS c1 FROM t
 ) AS t) t ORDER BY c0
```

```
SELECT * FROM ( SELECT CEIL(IF(c1, c1, c0)) AS c0 FROM (
  SELECT col_string AS c0, col_datetime AS c1 FROM t
 ) AS t) t ORDER BY c0
```

PingCAP

Part IV - 总结 & 未来工作

总结

- 随机生成 Query
  - 随机性强，覆盖面广，发现人想象不到的 bug
  - 避免手动构造 case 的枯燥劳动
  - 容易落地，已经发现不少问题
  - 框架完善，扩展性强，新增算子/表达式/约束都很容易
- 广度优先生成 Query
  - 框架完善，扩展性强，轻松添加规则
  - 针对性更强，更容易构造相似归因的 Query 集合进行有针对的测试

未来工作

- 随机生成 Query
  - 支持更多的算子 (window, view...)
  - 支持更多的表达式
  - 支持DML
- 广度优先生成 Query
  - 定义更多的 Rule，加大搜索空间
- 在搜索空间引入更多的搜索策略...(比如参考 gofuzz-testing 设计带反馈的 DFS 搜索算法)
- 拓宽框架应用场景，不仅局限于正确性验证，比如和 MySQL 对比进行性能测试，分析执行计划等

PingCAP

Thank You !