



# 数据库系统测试方法介绍


---

- 2022.02.25



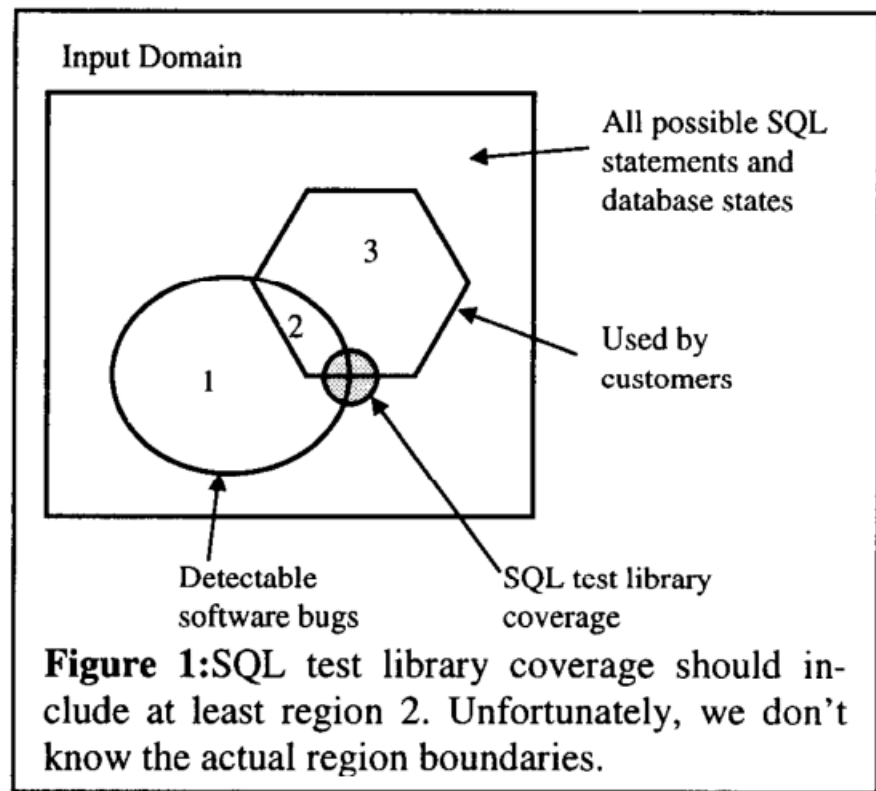


# 目录

- 
- A horizontal bar consisting of two segments: a blue segment on the left and an orange segment on the right.
- 研究背景与意义
  - 测试数据库
  - 测试优化器
  - 总结

## ● 测试目的

- 找到代码中的bug
- 验证数据库系统的功能正确性，确保系统性能
- Regression test, 回归测试，避免开发过程出现功能或性能回滚





# 研究背景与意义

- 测试分类

对象	类型	测试工具（系统）
测试数据库	Fuzzing 模糊测试	<a href="#">SQLsmith</a> , <a href="#">Google AFL fuzzer</a> , <a href="#">SQUIRREL</a>
	系统逻辑正确性测试	<a href="#">RAGS</a> (SQL-server,1998) , <a href="#">SQLancer</a>

对象	关键技术点	测试工具（系统）
测试优化器	优化器性能指标	<a href="#">OptMark</a> , <a href="#">TAQO</a> (orca)
	遍历执行计划空间	<a href="#">OptMark</a> , <a href="#">TAQO</a> (orca), <a href="#">SQL Server测试工具</a> , <a href="#">Horoscope</a> (TiDB)



# 目录

- 测试数据库
- 测试优化器
- 总结



# 测试数据库-模糊测试

- 模糊测试
  - 模糊测试（fuzz testing, fuzzing）是一种软件测试技术。其核心思想是将自动或半自动生成的随机数据输入到一个程序中，并监视程序异常，如崩溃，断言（assertion）失败，以发现可能的程序错误，比如内存泄漏。

类型	测试工具	类型
模糊测试	SQLsmith	Generation-based
	Google AFL	Mutation-based
	SQUIRREL	Mixed

**Table 2: Compatibility between fuzzers and DBMSs.** MySQL only permits C/S mode, which is not supported by the last three fuzzers. SQLsmith does not support MySQL’s grammar. QSYM supports fuzzing PostgreSQL in the single mode, GRIMOIRE cannot compile it, while Angora cannot run it.

	SQUIRREL	AFL	SQLsmith	QSYM	Angora	GRIMOIRE
SQLite	✓	✓	✓	✓	✓	✓
PostgreSQL	✓	✓	✓	✓ (single)	✗ (execute)	✗ (compile)
MySQL	✓	✓	✗ (interface)	✗ (C/S)	✗ (C/S)	✗ (C/S)

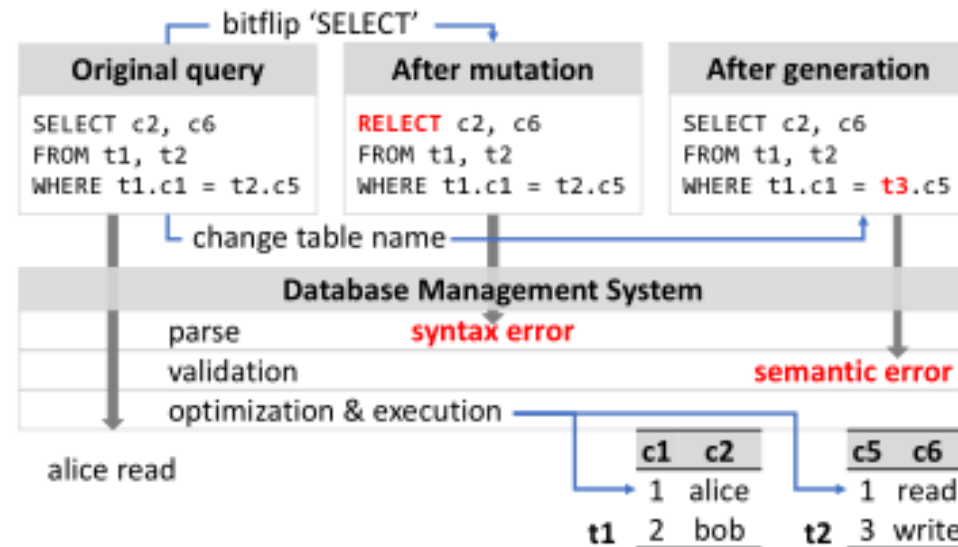
# 测试数据库-模糊测试

## ● SQLsmith Genaration-based

- SQLsmith 的灵感来自于C编译器测试工具Csmith。
- SQLsmith 基于AST来生成测试的SQL表达式，生成的SQL符合SQL语法，但是不符合SQL语义。
- 没有设计反馈机制，大多数SQL语句往往测试的代码执行相同，测试不到系统深层逻辑。

## ● Google AFL Mutation-based

- Mutation-based的方法随机生成SQL语句，并对SQL语句进行修改，如果修改能提升一些指标，如code coverage，那么会将这个SQL加入一个set，并后续基于此去继续mutation。
- AFL不是针对DBMS的，只有4%的SQL语句能够通过语法和语义检查。



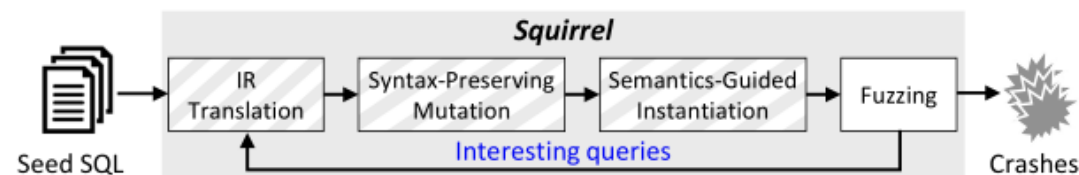
**Figure 1: Challenges of testing DBMSs.** A DBMS takes four steps to process one SQL query. Among them, *parse* checks syntactic correctness, and *validation* examines semantic validity. Random mutation unlikely guarantees the syntactic correctness, while grammar-based generation may fail to enforce semantic correctness.



# 测试数据库-模糊测试

## • SQUIRREL Mixed

- 针对DBMS的 Generation-based & Mutation-based 相结合
- 能确保生成SQL的语法是正确的，同时依靠数据依赖等提高语义检查的正确性。
- 同时设计了基于反馈的Mutation策略，生成更有针对性的SQL语句以引起系统的崩溃。



**Figure 2: Overview of SQUIRREL.** SQUIRREL aims to find queries that crash the DBMS. SQUIRREL first lifts queries from SQL to IR; then, it mutates IR to generate new skeletons; next, it fills the skeleton with concrete operands; finally, it runs the new query and detects bugs.



# 测试数据库-模糊测试

- **SQUIRREL Mixed**
- IR Translator 中间结果
- Syntax-Preserving Mutation 语法保持mutation

```

1 // l: left child, r: right child, d: data, t: data type
2 V1 = (Column, l=0, r=0, op=0, d="c2", t=ColumnName);
3 V2 = (ColumnRef, l=V1, r=0, op=0, d=0);
4 V3 = (Expr, l=V2, r=0, op=0, d=0);
5 V4 = (Column, l=0, r=0, op=0, d="c6", t=ColumnName);
6 V5 = (ColumnRef, l=V4, r=0, op=0, d=0);
7 V6 = (Expr, l=V5, r=0, op=0, d=0);
8 V7 = (SelectList, l=V3, r=V6, op=0, d=0);
9 // the optional left child can be DINSTRICT
10 V8 = (SelectClause, l=0, r=V6, op.prefix="SELECT", d=0);
11 ...
12 //Unknown type for intermediate IRs
13 Va = (Unknown, l=V8, r=V14, op=0, d=0);
14 Vb = (Unknown, l=Va, r=V25, op=0, d=0);
15 // the optional right child can be an ORDER clause
16 V26 = (SelectStmt, l=Vb, r=0, op=0, d=0);

```

Figure 3: IR of the running example SQL query. The corresponding AST tree is shown in Figure 4.

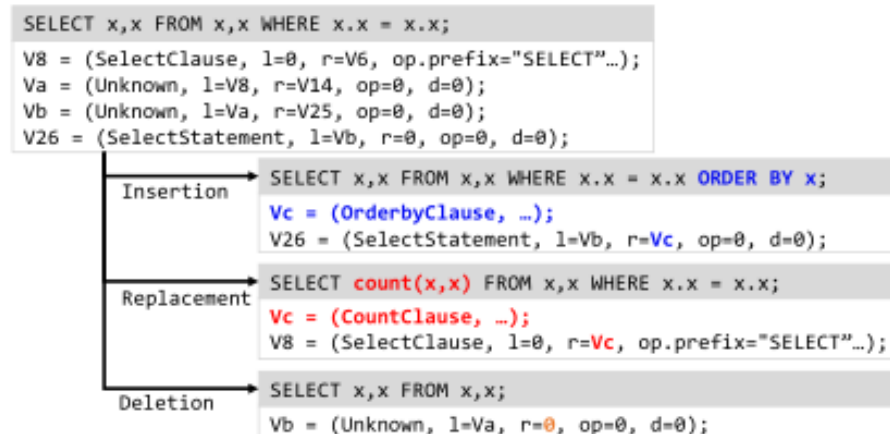


Figure 5: Mutation strategies on IR programs, including type-based insertion and replacement, and deletion of optional operands.

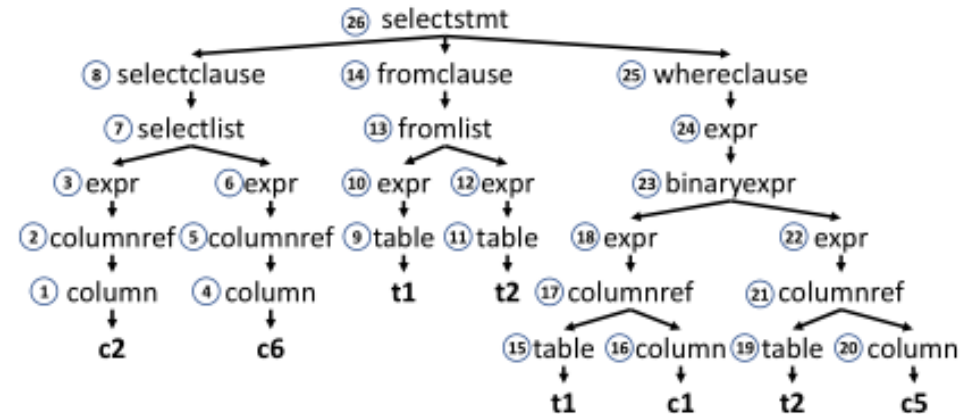
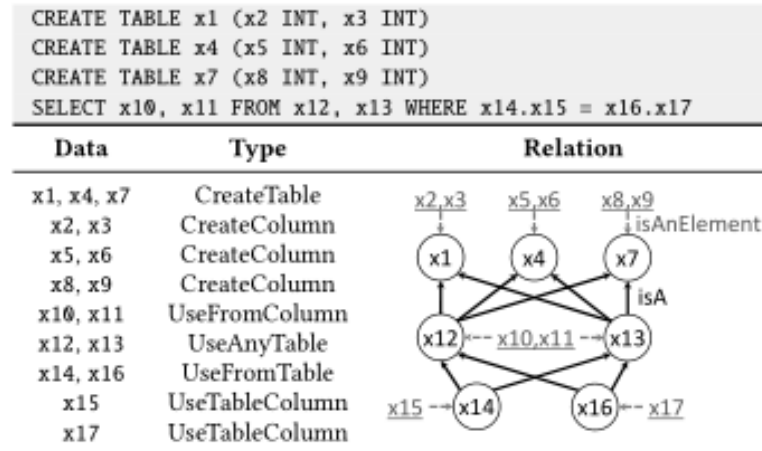


Figure 4: AST of the running example. SQUIRREL parses the SQL query and represents it in AST, and finally translate AST to IR.

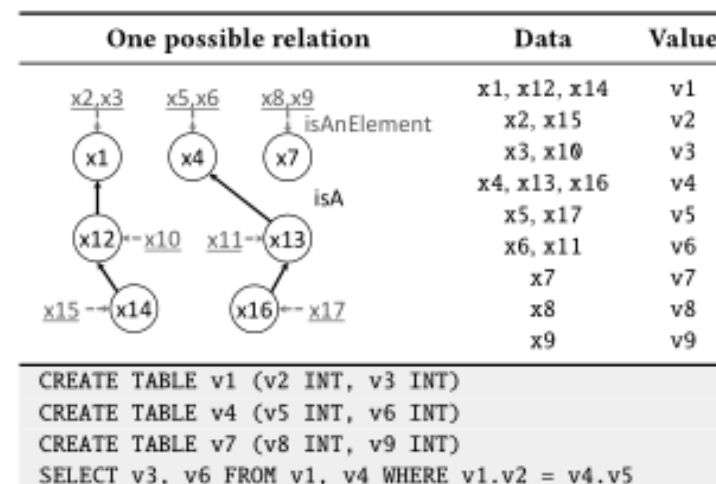
# 测试数据库-模糊测试

## • SQUIRREL Mixed

- Semantics-Guided Instantiation  
语义导向实例化
- 经过mutation后的IR还没有指定具体的值, 如何进一步确定这些测试用例以提升生成query语义正确的概率。
- 根据分析数据依赖从而进行实例化
  1. 根据几条SQL语句形成数据直接的依赖图
  2. 选择一条路径, 用具体的表格和数据来实例化从而生成语义正确的SQL语句



**Figure 6: Data dependency example.** This example consists of three new CREATE statements and our running example. In "Relation", we show two types of relations: "isAnElement" (dashed line) and "isA" (solid line).



**Figure 7: Instantiation of IR structure.** We create one concrete dependency graph from the dependencies of Figure 6, replace all place-holder x and finally get one concrete new query.



# 测试数据库-模糊测试

## ● 小结

类型	测试工具	类型	特点
模糊测试	SQLsmith	Generation-based	1. SQL语法确保正确, 语义不确保正确 2. 没有反馈机制
	Google AFL	Mutation-based	1. 通用, 不只针对DBMS, SQL语法语义正确率极低 2. 具有反馈的Mutation机制, SQL语句更有针对性
	SQUIRREL	Mixed	1. 针对DBMS, SQL语法确保正确, 语义正确率高 2. 具有反馈的Mutation机制, SQL语句更有针对性

- 问题: 通过监控数据库发生的crash来发现bug, 不能检查逻辑bug, 如一个SQL语句的结果是否正确。



# 测试数据库-系统正确性测试

- 系统正确性测试
  - 测试验证一个数据库获取的数据是否正确，符合逻辑。

Listing 1: Illustrative example, based on a *critical* SQLite bug. The check symbol denotes the expected, correct result, while the bug symbol denotes the actual, incorrect one.

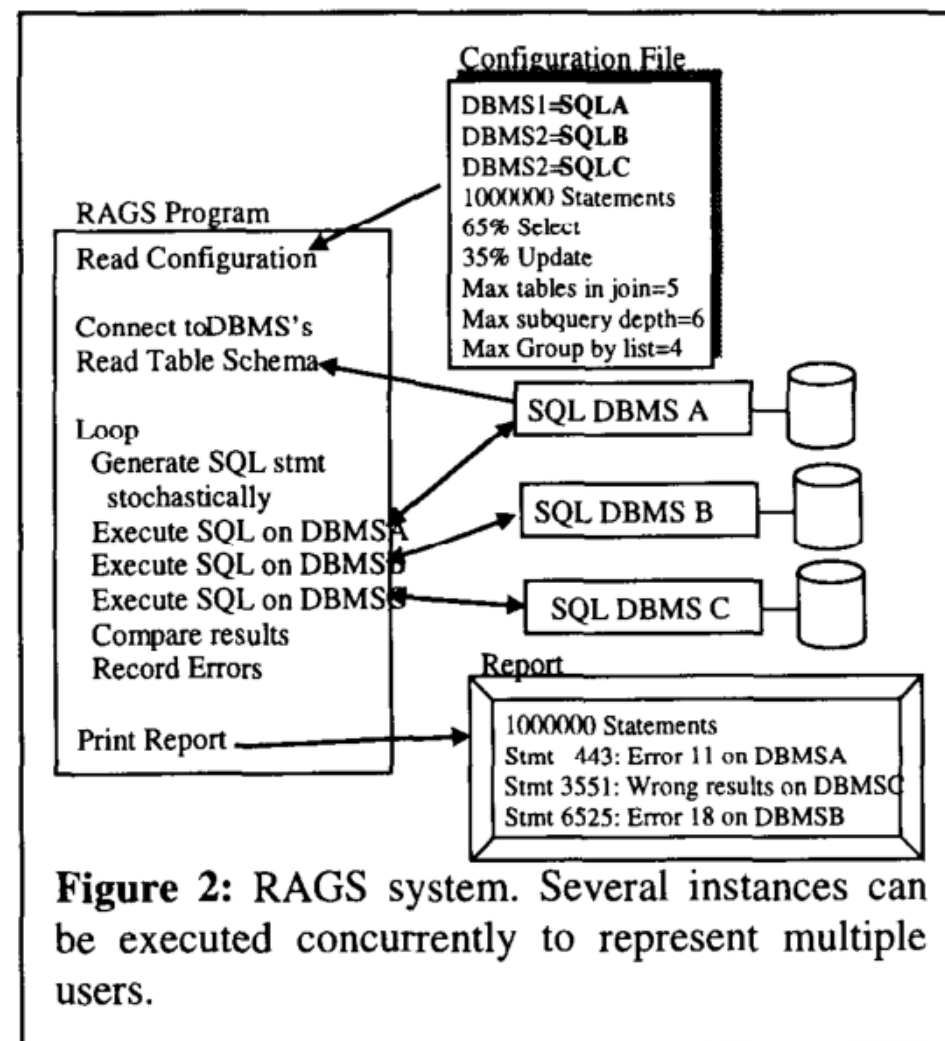
```
CREATE TABLE t0(c0);
CREATE INDEX i0 ON t0(1) WHERE c0 NOT NULL;
INSERT INTO t0(c0) VALUES (0), (1), (NULL);
SELECT c0 FROM t0 WHERE c0 IS NOT 1; -- {0}  {0,
NULL} 
```

类型	测试工具	类型
系统逻辑正确性测试	RAQS	一个SQL语句在多个DBMS上结果的比较
	SQLancer	一个SQL在同一个DBMS结果的比较

# 测试数据库-系统正确性测试

- **RAGS** Massive Stochastic Testing of SQL

- 1998年被微软用于测试SQL server.
- **思想**: 一个SQL语句在多个DBMS上结果的比较, 如果有结果不一致则其中有一个DBMS可能出现bug.
- **缺点**: 不同的DBMS存在方言等system-specify的特性, 这个方法只能测试那些数据库系统中共有的SQL语法.





# 测试数据库-系统正确性测试

- SQLancer
  - **PQS**: [Testing Database Engines via Pivoted Query Synthesis](#)
  - **NoREC**: [Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction](#)
  - **TLP**: [Ternary Logic Partitioning: Detecting Logic Bugs in Database Management Systems](#)



# 测试数据库-系统正确性测试

## • PQS Testing Database Engines via Pivoted Query Synthesis

思想：以结果为导向，生成一定会包含某一行的SQL语句，在数据库中执行，如果不包含这行，则说明数据库有bug。

1. 随机生成 table 和插入数据
2. 从数据库中随机选择一行数据
3. 根据这行数据，随机构造一个 expression
4. 执行 expression，如果不为 TRUE，调整为 TRUE
5. 将这个 expression 放到 where 或者 join 里面
6. 执行这条查询语句
7. 看最新的返回结果是不是还包含之前的那行数据，如果没有，则表明有 bug

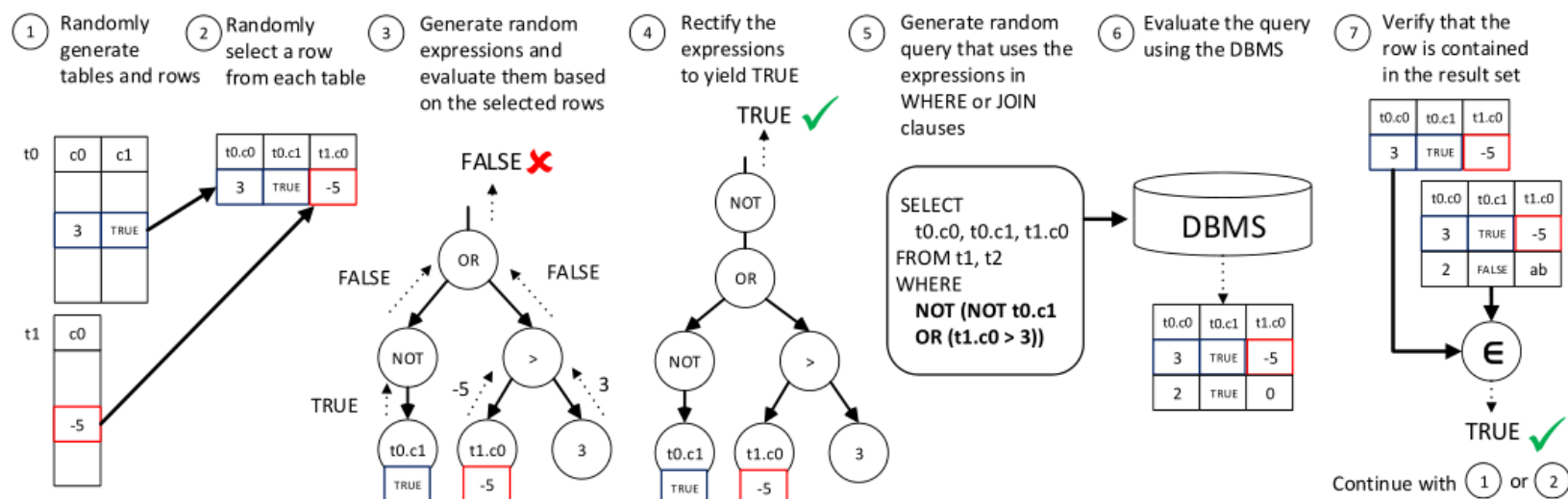


Figure 1: Overview of the approach implemented in SQLancer. Dotted lines indicate that a result is generated.



- PQS Testing Database Engines via Pivoted Query Synthesis

Listing 2: Checking containment using the **INTERSECT** operator in SQLite.

```
SELECT (3, TRUE, -5) INTERSECT SELECT t0.c0, t0.c1  
      , t1.c0 FROM t1, t2 WHERE NOT(NOT(t0.c1 OR (t1  
      .c0 > 3))));
```

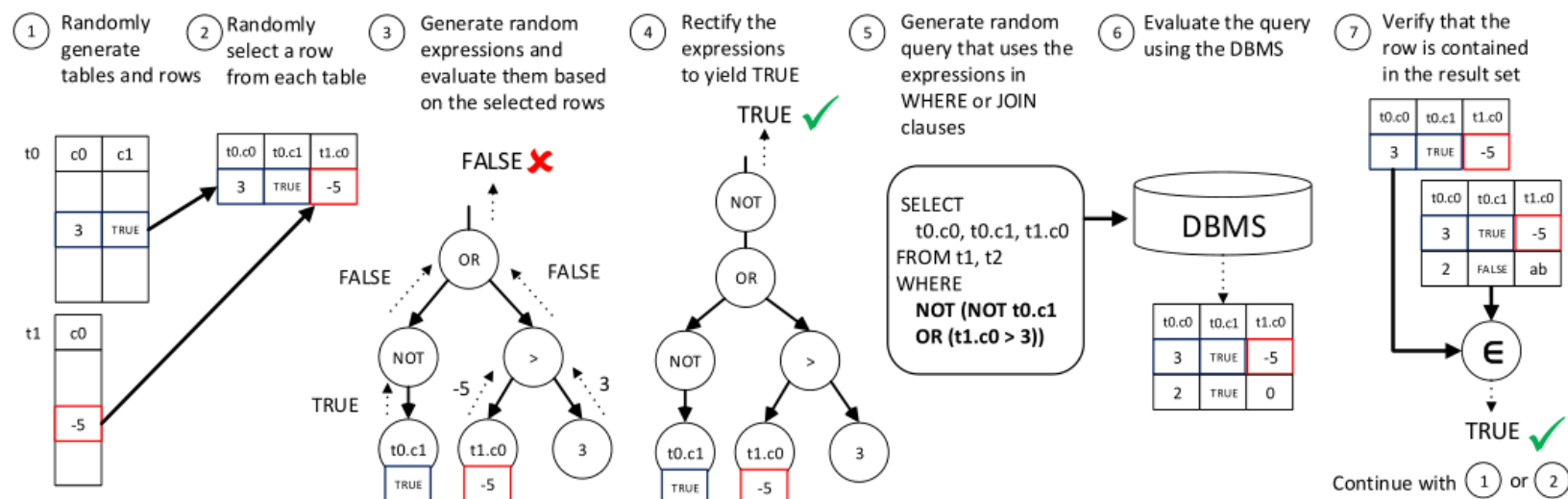


Figure 1: Overview of the approach implemented in SQLancer. Dotted lines indicate that a result is generated.





# 测试数据库-系统正确性测试

## • NoREC Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction

思想：将一条优化的 Query，强制变成非优化的方式，然后对比查询结果，如果两种执行方式不一致，那就是有 bug。

1. 优化过的执行计划
2. 非优化的执行计划
3. 比较两者的结果，不一样则有bug

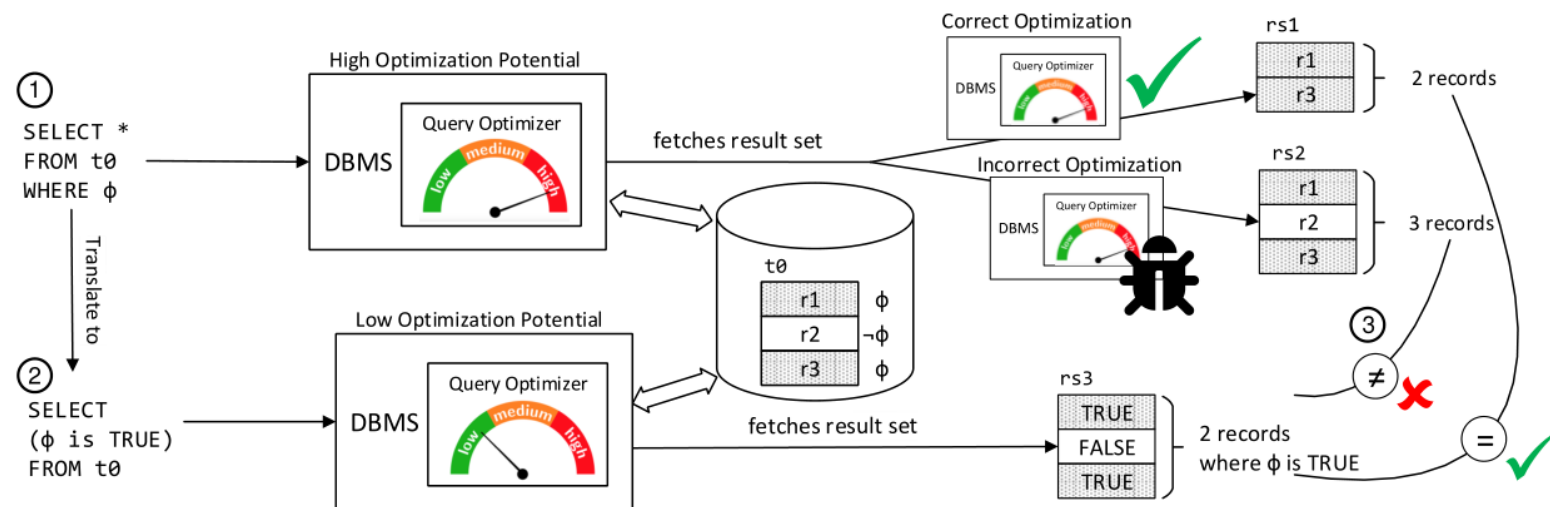


Figure 1: The core of the approach is the translation of an optimized query (step ①) to an unoptimized one (step ②), which allows the automatic detection of optimization bugs (step ③).  $t_0$  is a table contained in the database, and  $rs_1$ ,  $rs_2$ , as well as  $rs_3$  are result sets returned by the DBMS. Predicate  $\phi$  is random, but fixed.

# 测试数据库-系统正确性测试

- **NoREC** Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction

**Listing 1: Illustrative example where a bug in SQLite's *LIKE* optimization caused a record to mistakenly be omitted.**

```
CREATE TABLE t0(c0 UNIQUE);
INSERT INTO t0 VALUES (-1);
```

```
① SELECT * FROM t0 WHERE t0.c0 GLOB '-*'; -- {} 🐛
② SELECT t0.c0 GLOB '-*' FROM t0; -- {TRUE} ✓
```

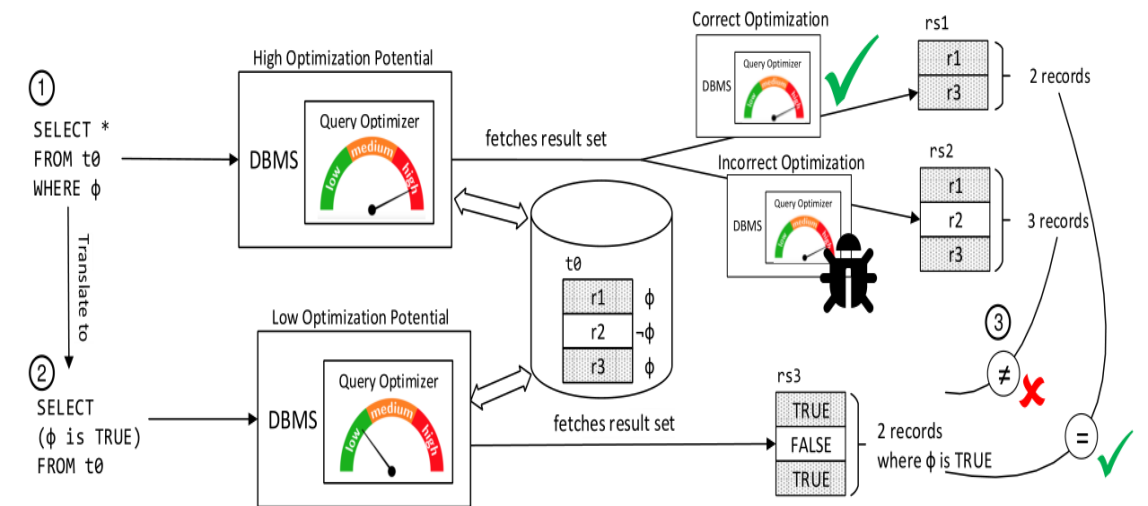


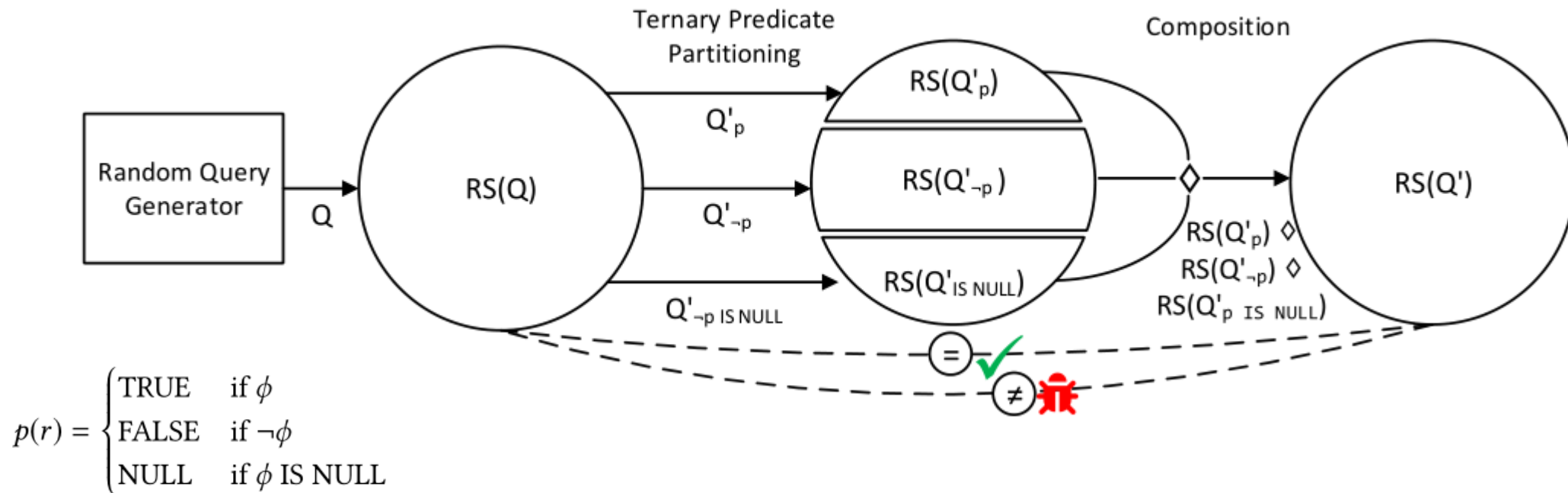
Figure 1: The core of the approach is the translation of an optimized query (step ①) to an unoptimized one (step ②), which allows the automatic detection of optimization bugs (step ③). t0 is a table contained in the database, and rs1, rs2, as well as rs3 are result sets returned by the DBMS. Predicate  $\phi$  is random, but fixed.



# 测试数据库-系统正确性测试

- **TLP Ternary Logic Partitioning: Detecting Logic Bugs in Database Management Systems**

思想：将一个 Query 分成了三个 Query，执行结果分别为 TRUE，FALSE 和 NULL，然后再将这三个 Query 的结果合并，并且保证结果为 TRUE。





# 测试数据库-系统正确性测试

- **TLP** Ternary Logic Partitioning: Detecting Logic Bugs in Database Management Systems

$$p(r) = \begin{cases} \text{TRUE} & \text{if } \phi \\ \text{FALSE} & \text{if } \neg\phi \\ \text{NULL} & \text{if } \phi \text{ IS NULL} \end{cases}$$

Listing 1. A logic bug in MySQL caused a predicate  $0=-0$  to incorrectly evaluate to **FALSE**. The check symbol denotes the expected, correct result, while the bug symbol denotes the actual, incorrect result.

```
CREATE TABLE t0(c0 INT);
CREATE TABLE t1(c0 DOUBLE);
INSERT INTO t0 VALUES(0);
INSERT INTO t1 VALUES('-0');
① SELECT * FROM t0, t1 WHERE t0.c0 = t1.c0; -- {}
② SELECT * FROM t0, t1; -- {0, -0}
③ SELECT * FROM t0, t1 WHERE t0.c0 = t1.c0
    UNION ALL SELECT * FROM t0, t1 WHERE NOT(t0.c0 = t1.c0)
    UNION ALL SELECT * FROM t0, t1 WHERE (t0.c0 = t1.c0) IS NULL; -- {}
```



# 测试数据库-系统正确性测试

- 小结

类型	工具	方法	特点
系统正确性测试	SQLancer	PQS	根据结果行生成 query
		NoREC	比较优化和非优化的两个执行方案
		TLP	将一个query拆成几部分执行
	RAGS	RAGS	一条SQL不同DBMS执行，适用范围小



# 目录

- 测试数据库
- 测试优化器
- 总结



# 测试优化器

- 为什么要测试优化器

- 整体测试的性能变化的原因可能是不同的组件，如存储层
- 优化器是DBMS中重要而又复杂的组件
- 需要确定当前选择的执行计划是否最优？选择率估计是否准确？

- 如何测试优化器

- 定义优化器的性能指标
- 遍历执行计划空间
- 数据集以及查询生成

对象	关键技术点	测试工具（系统）
测试优化器	优化器性能指标	OptMark, TAQO(orca)
	遍历执行计划空间	OptMark, TAQO(orca), SQL Server测试工具, Horoscope(TiDB)



# 测试优化器

- 优化器性能指标

	指标目标	测试工具	指标
优化器性能指标	Effectiveness	OptMark	Performance Factor, Optimality Frequency
	Effectiveness	TAQO(orca)	Rank 指标
	Efficiency	OptMark	logical plan, join的数量等





# 测试优化器

- **优化器性能指标 OptMark Effectiveness**

- Performance Factor

优化器所选执行计划比其他潜在计划代价小的比例

$$\mathbf{PF}(O_D, q) = \frac{|\{p | p \in P_D(q), \mathbf{r}(D, p) \geq \mathbf{r}(D, O_D(q))\}|}{|P_D(q)|} \quad (1)$$

- Optimality Frequency

PF等于1的query的数量



# 测试优化器

## • 优化器性能指标 TAQO rank指标

- 同样使用比较大小避免了直接对 execution time 的直接使用

$$\forall i, j : e_i \leq e_j \iff a_i \leq a_j$$

$$\tau = \sum_{i < j} \text{sgn}(e_j - e_i) \quad (1)$$

- 性能越好的执行计划的权重应该越高

$$w_m = \frac{a_1}{a_m} \quad (2)$$

- 开销相近的执行计划之间权重应该较小

$$d_{ij} = \sqrt{\left(\frac{a_j - a_i}{a_n - a_1}\right)^2 + \left(\frac{e_j - e_i}{\max_k(e_k) - \min_k(e_k)}\right)^2} \quad (3)$$

- 最终的总公式

$$s = \sum_{i < j} w_i w_j d_{ij} \cdot \text{sgn}(e_j - e_i) \quad (4)$$



# 测试优化器

- 优化器性能指标 OptMark Efficiency

- Efficiency
- 传统
  - 记录优化器优化的平均时间开销
  - 与系统有关，没有考虑到memory
- OptMark
  - #LP - 枚举的 logical plan个数
  - #JO - 枚举的 join 顺序个数
  - #PP - 总的有开销的 physical plan 的个数
  - #PJ - 总的有开销的 physical join plan 的个数

System	#LP	#JO	#PP	#PJ
MySQL	0.92	0.93	<b>0.94</b>	<b>0.94</b>
PostgreSQL	0.72	0.72	<b>0.97</b>	<b>0.97</b>
System X	<b>0.81</b>	<b>0.81</b>	0.71	0.72
System Y	0.77	0.75	<b>0.85</b>	<b>0.85</b>

Table 1: Correlation of efficiency metrics and optimization times over four DBMSs



# 测试优化器

- 遍历执行计划方式

	测试工具	方法
遍历执行计划方式	OptMark	随机生成
	TAQO(orca)	使用不同的hint
	Horoscope (TiDB)	nth_plan hint
	SQL server测试工具	基于Memo数据结构的

# 测试优化器

## • OptMark 遍历执行计划空间

- **信心指数**: 利用统计学上的Cochran公式计算要达到一定置信度的准确率需要采样的执行计划的数量。
- 对于一条 Query, 对里面的 Join **随机** 进行重新排序。
- 对于 join 的两个 table, 如果没有指定 join 方式, 则使用 cross join, 否则则随机从 joinType() 里面选择一个 physical join, 譬如 hash, index merge 等。
- 对任何 table, 随机选择一种扫描方式, 譬如使用某个 index, 或者全表扫描。
- 生成一条 plan, 去执行。然后重复执行上述操作, 直到满足我们之前说的信心指数。

$$n = \frac{Z^2 p(1-p)}{e^2}$$

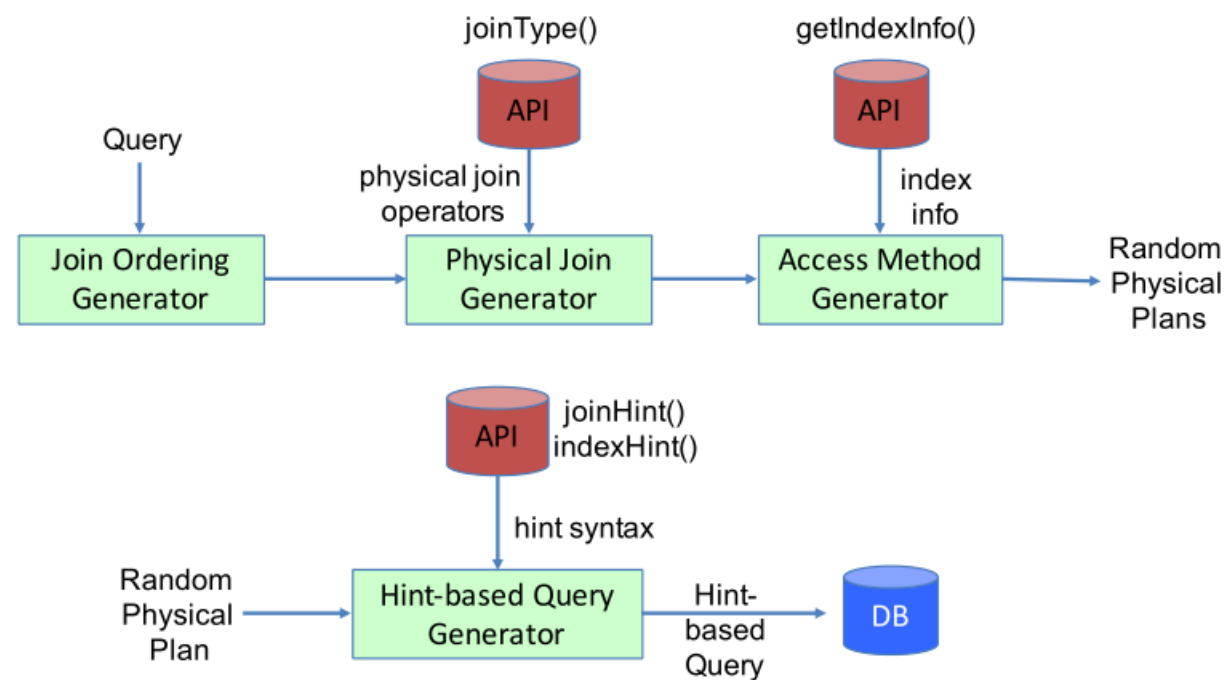


Figure 3: Effectiveness Benchmarking Workflow



# 测试优化器

- TAQO 遍历执行计划方式

- 思想：使用不同的hint，如是否使用 nested loop join等开关让现有数据库的优化器生成不同的执行计划
- 这种方法生成的物理执行计划都是有可能被优化器选择的，不过正如右图显示的，其数量可能还不够多。

Query	Opt-A		Opt-B		Opt-C		Opt-D	
	OK	T/O	OK	T/O	OK	T/O	OK	T/O
Q1	1	5	1	1	31	2	6	10
Q2	73	6	161	40	236	120	87	14
Q3	24	8	19	37	27	67	49	15
Q4	26	2	2	2	24	64	49	15
Q5	34	8	22	133	32	57	140	36
Q6	3	0	1	1	16	0	4	0
Q7	34	6	45	105	36	57	126	26
Q8	20	18	25	163	18	44	65	32
Q9	4	46	14	144	3	50	96	23
Q10	34	6	65	98	29	64	56	9
Q11	36	2	173	23	1	0	33	12
Q12	15	3	8	13	60	31	80	24
Q13	19	13	14	2	28	6	60	24
Q14	13	1	11	4	29	14	27	4
Q15	6	0	2	0	1	0	69	8
Q16	54	6	62	9	1	0	24	23
Q17	6	18	10	14	1	0	24	23
Q18	31	19	28	84	1	0	53	13
Q19	12	2	11	13	86	1	26	6
Q20	83	5	61	87	178	58	94	49
Q21	18	38	75	59	98	70	98	72
Q22	12	8	28	4	1	0	37	12

Table 1: Number of plans generated for TPC-H queries



# 测试优化器

## • TiDB Horoscope 遍历执行计划

- **思想：**在优化器遍历物理执行计划时对每个遍历过的物理执行计划记录编号。
- 通过 `nth_plan` 语法提供指定使用哪个物理执行计划。

```
TiDB(root@127.0.0.1:test) > explain select /*+ nth_plan(1) */ * from t where a = 1 and b > 0 and b < 10;
```

id	estRows	task	access object	operator info
TableReader_7	0.25	root		data:Selection_6
└─Selection_6	0.25	cop[tikv]		eq(hehe.t.a, 1), gt(hehe.t.b, 0), lt(hehe.t.b, 10)
└─┬─TableFullScan_5	10000.00	cop[tikv]	table:t	keep order:false, stats:pseudo

```
3 rows in set (0.00 sec)
```

```
// findBestTask converts the logical plan to the physical plan. It's a new interface.  
// It is called recursively from the parent to the children to create the result physical plan.  
// Some logical plans will convert the children to the physical plans in different ways, and return the one  
// With the lowest cost and how many plans are found in this function.  
// planCounter is a counter for planner to force a plan.  
// If planCounter > 0, the clock_th plan generated in this function will be returned.  
// If planCounter = 0, the plan generated in this function will not be considered.  
// If planCounter = -1, then we will not force plan.  
findBestTask(prop *property.PhysicalProperty, planCounter *PlanCounterTp) (task, int64, error)
```



# 测试优化器

## • SQL server 遍历执行计划方式

- **思想：**与前者类似，相比于前者，编号是基于Memo结构递增的。

- 将初始的逻辑执行计划装到Memo结构中。
- 左下角的7.1这样的编号是 Momo编号  
右下角的是Memo的子节点编号，子节点之间有顺序，前面的是左孩子。

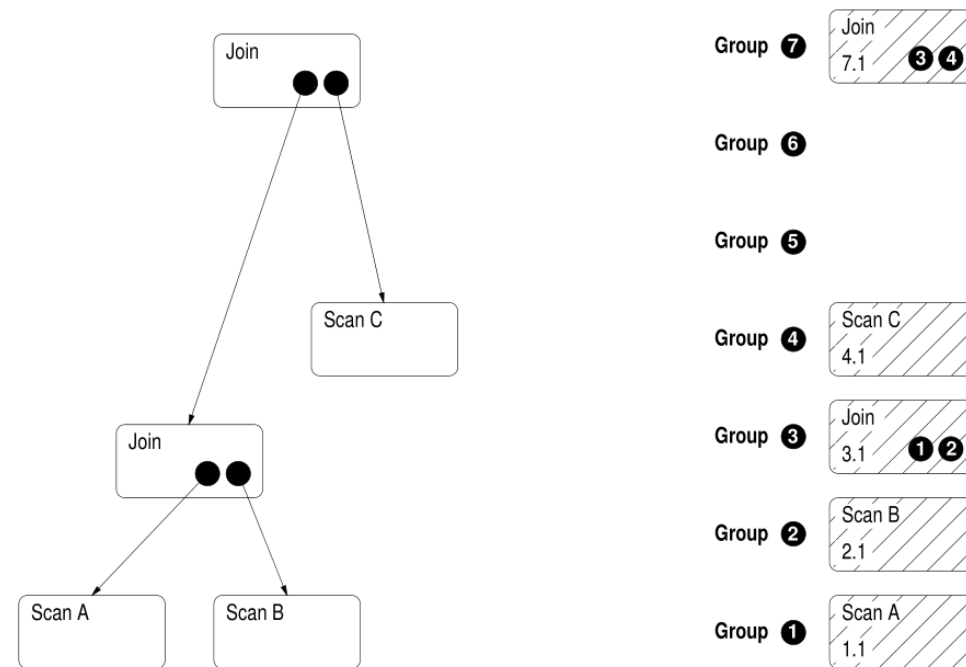


Figure 1: Copying the initial plan into the MEMO structure.





# 测试优化器

## • SQL server 遍历执行计划方式

- Figure2: 经过优化遍历后的物理执行计划空间
- Figure3: 编号计算示例

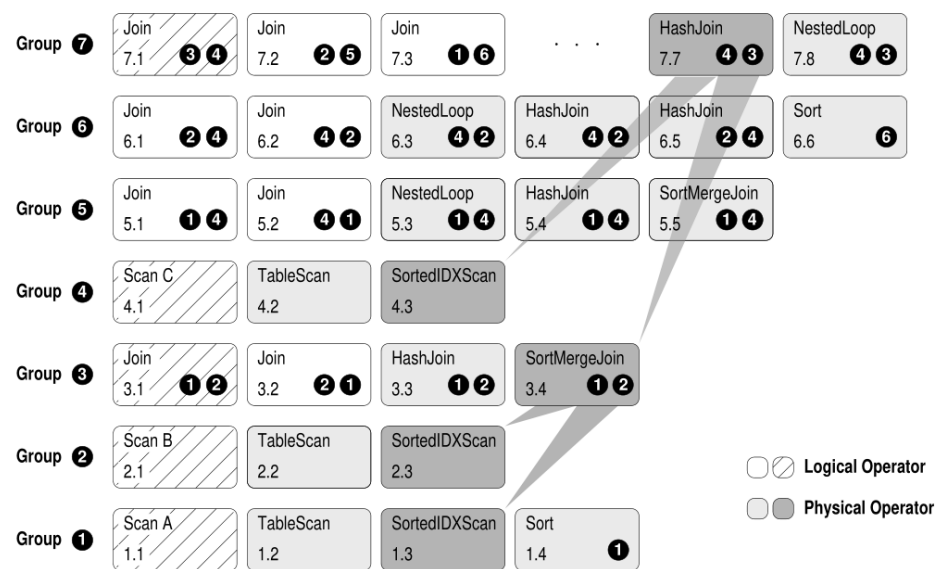


Figure 2: MEMO structure representing alternative solutions.

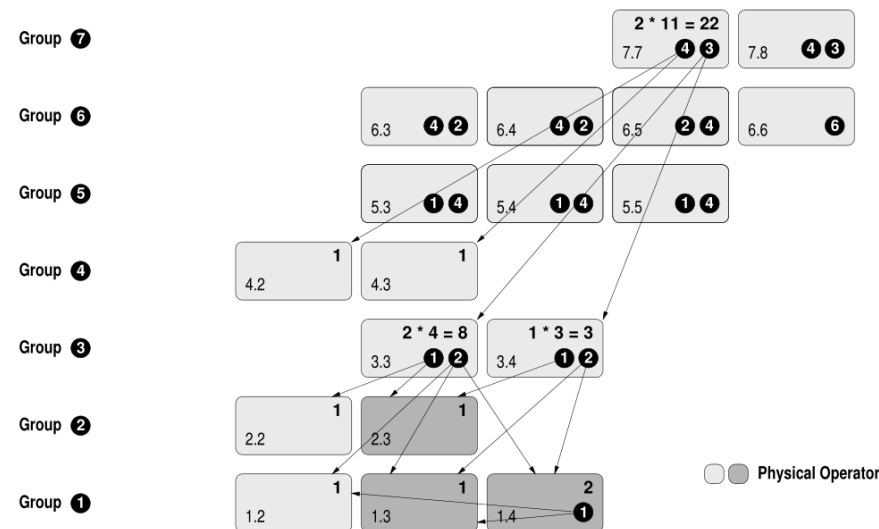
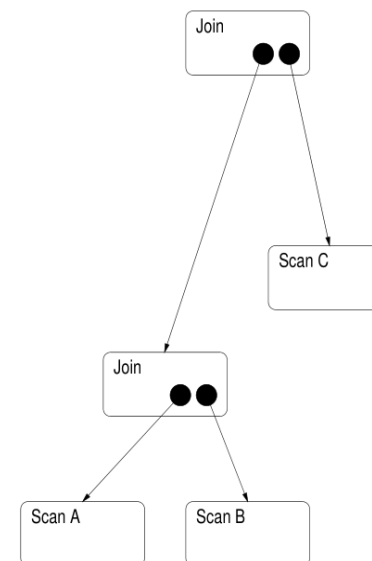


Figure 3: MEMO Structure with materialized links between operators and children, for possible plans rooted in operator 7.7.



# 测试优化器

- 优化器性能指标

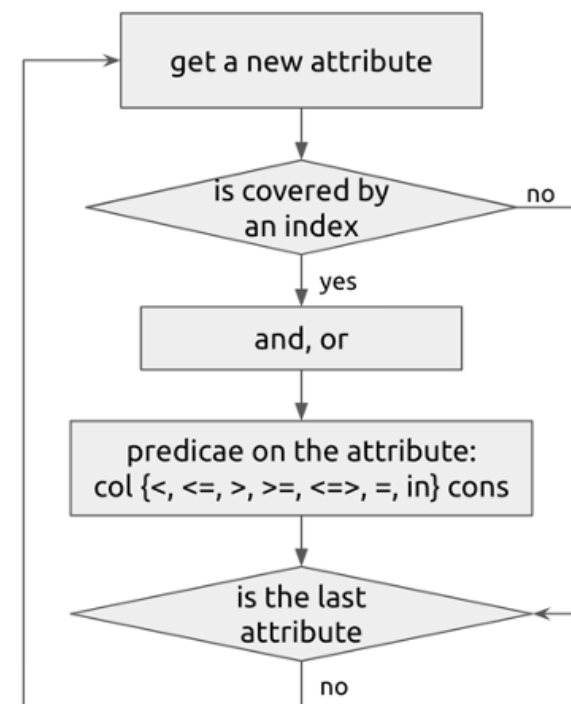
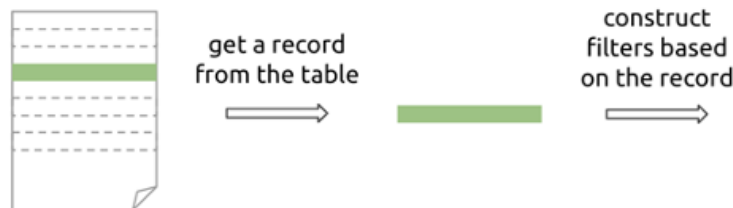
	指标目标	测试工具	指标
优化器性能指标	Effectiveness	OptMark	Performance Factor, Optimality Frequency
	Effectiveness	TAQO(orca)	Rank 指标
	Efficiency	OptMark	logical plan, join的数量等
	测试工具	方法	特点
遍历执行计划方式	OptMark	随机生成	生成的部分执行计划可能不会被优化器选择
	TAQO(orca)	使用不同的hint	会被优化器选择，但是数量少
	Horoscope (TiDB)	nth_plan hint	记录遍历过的物理执行计划
	SQL server测试工具	基于Memo数据结构的	



# 测试优化器

## • TiDB Horoscope 介绍

- 定义优化器的性能指标  
使用了OptMark中的Performance factor
- 遍历执行计划空间  
使用了 **nth\_plan** hint
- 数据集以及查询生成  
数据集使用的imdb真实数据集  
查询生成使用的SQLancer的PQS思想





# 测试优化器

## ● TiDB Horoscope 介绍

### ○ 效果展示

ID	#PLAN SPACE	DEFAULT EXECUTION TIME	BEST PLAN EXECUTION TIME	EFFECTIVENESS	BETTER OPTIMAL PLANS
10a.sql	53	5182.2ms ± 4%	4549.2ms ± 6%	94.3%	#22(89.7%),#23(87.8%),#26(88.9%)
11a.sql	65	2467.2ms ± 8%	2467.2ms ± 8%	100.0%	
11b.sql	73	1478.2ms ±23%	1478.2ms ±23%	100.0%	
11c.sql	62	21241.2ms ± 7%	21241.2ms ± 7%	100.0%	
15a.sql	87	4204.5ms ±165%	4204.5ms ±165%	100.0%	
15b.sql	85	85.0ms ±30%	85.0ms ±30%	100.0%	
15c.sql	87	3181.5ms ± 2%	1711.5ms ± 2%	60.9%	#40(54.6%),#41(53.9%),#42(53.8%),#43(60.0%),#44(66.5%)
15d.sql	85	2600.2ms ± 3%	2110.0ms ± 3%	87.1%	#51(81.1%),#52(82.8%),#53(81.4%),#57(82.8%),#58(82.1%)
16a.sql	67	349.8ms ±159%	349.8ms ±159%	100.0%	
16b.sql	66	7869.5ms ± 5%	4409.8ms ± 5%	81.8%	#23(87.5%),#24(85.1%),#25(85.8%),#26(84.1%),#31(86.7%)
16c.sql	67	1039.8ms ± 6%	1039.8ms ± 6%	100.0%	
16d.sql	67	858.5ms ± 4%	858.5ms ± 4%	100.0%	
17a.sql	63	3673.0ms ± 5%	2703.2ms ± 3%	76.2%	#23(81.3%),#24(79.1%),#25(78.4%),#26(77.7%),#31(86.3%)
17b.sql	68	4042.8ms ± 8%	4042.8ms ± 8%	100.0%	
17c.sql	68	4002.0ms ± 4%	4002.0ms ± 4%	100.0%	
17d.sql	67	5679.2ms ± 5%	4079.8ms ± 8%	85.1%	#24(88.7%),#25(88.2%),#31(89.9%),#32(84.0%),#33(84.8%)
17e.sql	62	3731.2ms ± 7%	2920.5ms ± 4%	80.6%	#23(86.7%),#24(83.0%),#25(81.4%),#26(82.2%),#32(79.9%)
17f.sql	67	5949.2ms ± 5%	4352.8ms ± 9%	79.1%	#23(88.3%),#24(86.3%),#25(85.9%),#26(88.7%),#31(87.6%)
19a.sql	83	7315.0ms ± 8%	5936.2ms ± 1%	94.0%	#29(82.7%),#30(82.3%),#31(81.2%),#32(83.5%),#33(83.0%)
19b.sql	89	6522.0ms ± 5%	1901.5ms ± 3%	79.8%	#32(29.6%),#33(29.5%),#34(29.2%),#35(32.3%),#36(37.4%)
19c.sql	84	7108.8ms ± 4%	6266.2ms ± 3%	82.1%	#42(89.9%),#43(89.7%),#44(89.6%),#45(89.5%),#46(88.9%)
19d.sql	83	13492.5ms ± 3%	13492.5ms ± 3%	100.0%	



# 目录

- 测试数据库
- 测试优化器
- 总结



# 总结

对象	类型	测试工具（系统）
测试数据库	Fuzzing 模糊测试	SQLsmith, Google AFL fuzzer, SQUIRREL
	系统逻辑正确性测试	RAGS (SQL-server,1998) , SQLancer

对象	关键技术点	测试工具（系统）
测试优化器	优化器性能指标	OptMark, TAQO(orca)
	遍历执行计划空间	OptMark, TAQO(orca), SQL Server测试工具, Horoscope(TiDB)



# 总结

---

- 分布式数据库系统其他测试方向
  - Query 生成 SQLFUZZ
  - Database生成 QAGen
  - 一致性检查 Jepsen, Elle (go-elle)
  - ...



谢谢