15721 - Project 3

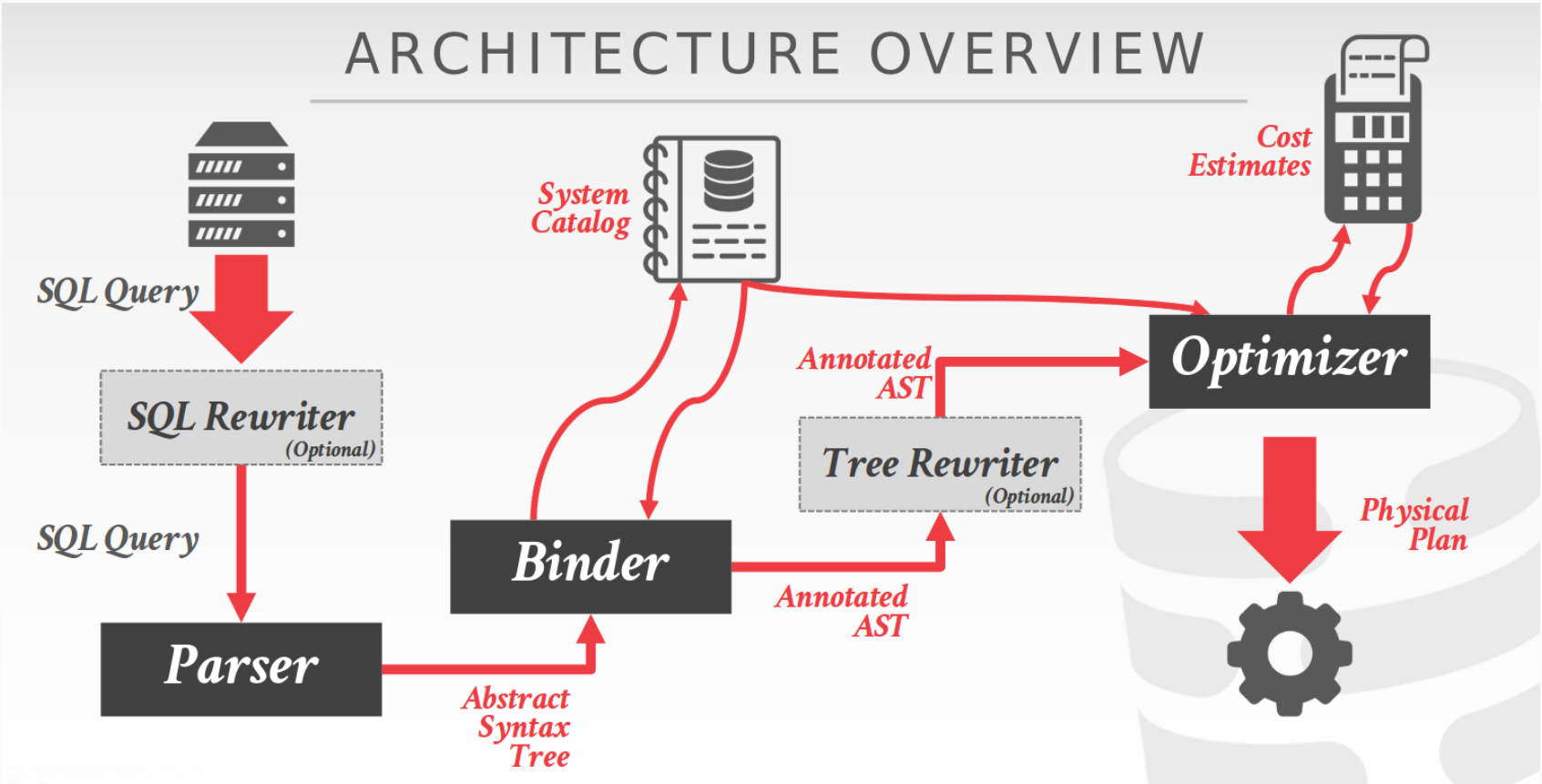
Query Optimizer

Guoquan Zhao
Pedro Paredes
Qiuwen Kai

May 14, 2018

Project overview

Query Optimizer



Goals

- **75% Goal:** Outer Joins;

Goals

- **75% Goal:** Outer Joins;
- **100% Goal:** Simple query rewrites;

Goals

- **75% Goal:** Outer Joins;
- **100% Goal:** Simple query rewrites;
- **125% Goal:** Expression rewrite;

Goals

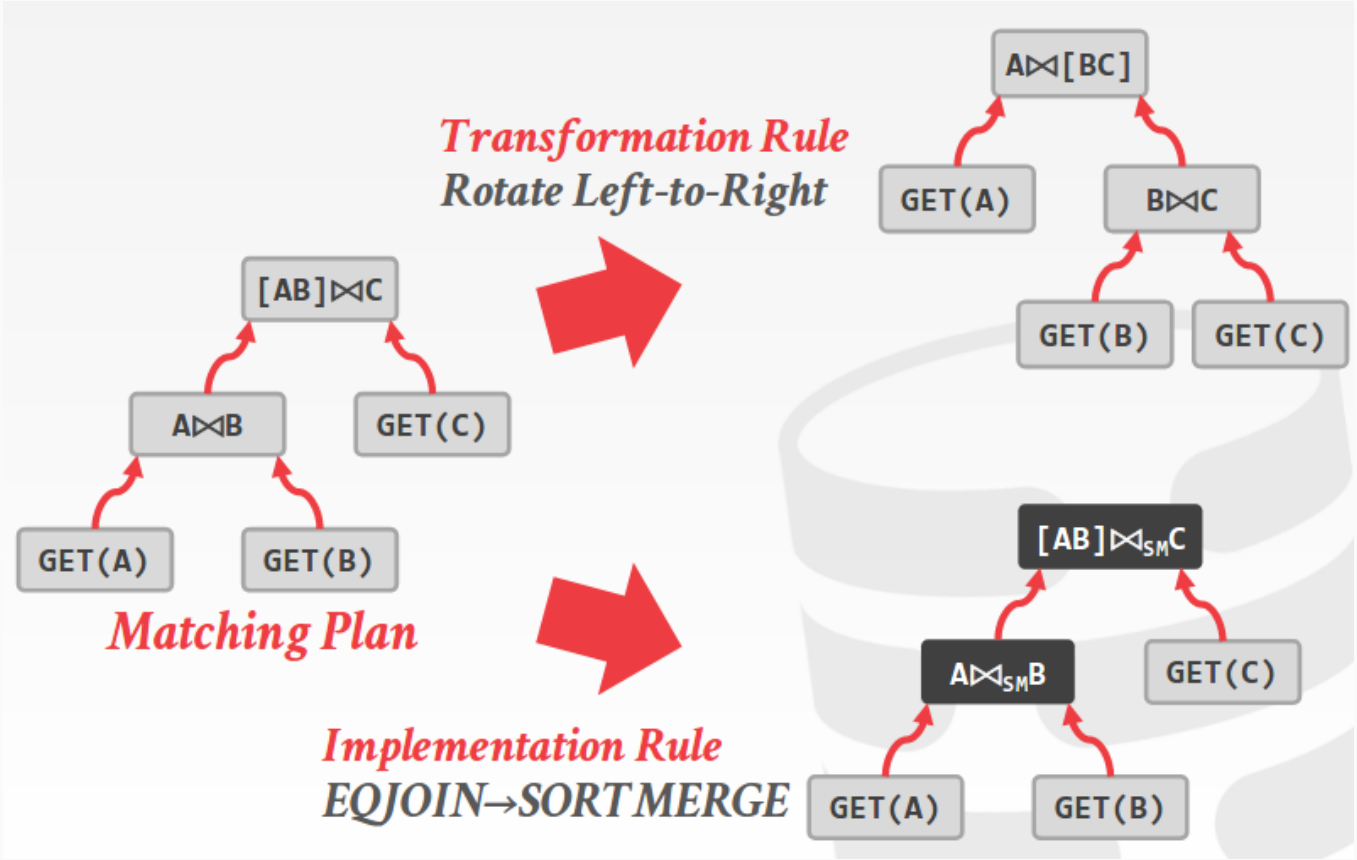
- **75% Goal:** Outer Joins;
- **100% Goal:** Simple query rewrites;
- **125% Goal:** Expression rewrite;

Goals

- **75% Goal:** Outer Joins; ✓
- **100% Goal:** Simple query rewrites; ✓
- **125% Goal:** Expression rewrite;

Outer joins

Outer joins - Optimizer



Summary of updates

```
+ // Associativity rules taken from
+ // http://15721.courses.cs.cmu.edu/spring2017/papers/14-optimizer1/p539-moerkotte.pdf
+ /*
+  * (A inner B) inner C = A inner (B inner C)
+  * (A right B) inner C = A right (B inner C)
+  * (A inner B) left  C = A inner (B left  C)
+  * (A left  B) left  C = A left  (B left  C) with Strong Predicates PBC
+  * (A full  B) left  C = A full  (B left  C) with Strong Predicate PBC
+  * (A right B) right C = A right (B right C) with Strong Predicate PAB
+  * (A right B) full  C = A right (B full  C) with Strong Predicate PAB
+  * (A full  B) full  C = A full  (B full  C) with Strong Predicate PAB & PBC
+  */
+ auto parent_join = expr->Op().As<LogicalJoin>();
+ std::vector<std::shared_ptr<OperatorExpression>> children = expr->Children();
+ auto child_join = children[0]->Op().As<LogicalJoin>();
+ if (parent_join->type == JoinType::INNER) {
+   if (child_join->type == JoinType::INNER || child_join->type == JoinType::RIGHT) {
+     return true;
+   }
+ } else if (parent_join->type == JoinType::LEFT) {
+   if (child_join->type == JoinType::INNER) {
+     return true;
+   } else if (child_join->type == JoinType::LEFT || child_join->type == JoinType::OUTER) {
+     return StrongPredicate(expr, context);
+   }
+ } else if (parent_join->type == JoinType::RIGHT) {
+   if (child_join->type == JoinType::RIGHT) {
+     return StrongPredicate(expr, context);
+   }
+ } else if (parent_join->type == JoinType::OUTER) {
```

- Add unified join operator

Summary of updates

```
+ // Associativity rules taken from
+ // http://15721.courses.cs.cmu.edu/spring2017/papers/14-optimizer1/p539-moerkotte.pdf
+ /*
+  * (A inner B) inner C = A inner (B inner C)
+  * (A right B) inner C = A right (B inner C)
+  * (A inner B) left  C = A inner (B left  C)
+  * (A left  B) left  C = A left  (B left  C) with Strong Predicates PBC
+  * (A full  B) left  C = A full  (B left  C) with Strong Predicate PBC
+  * (A right B) right C = A right (B right C) with Strong Predicate PAB
+  * (A right B) full  C = A right (B full  C) with Strong Predicate PAB
+  * (A full  B) full  C = A full  (B full  C) with Strong Predicate PAB & PBC
+  */
+ auto parent_join = expr->Op().As<LogicalJoin>();
+ std::vector<std::shared_ptr<OperatorExpression>> children = expr->Children();
+ auto child_join = children[0]->Op().As<LogicalJoin>();
+ if (parent_join->type == JoinType::INNER) {
+   if (child_join->type == JoinType::INNER || child_join->type == JoinType::RIGHT) {
+     return true;
+   }
+ } else if (parent_join->type == JoinType::LEFT) {
+   if (child_join->type == JoinType::INNER) {
+     return true;
+   } else if (child_join->type == JoinType::LEFT || child_join->type == JoinType::OUTER) {
+     return StrongPredicate(expr, context);
+   }
+ } else if (parent_join->type == JoinType::RIGHT) {
+   if (child_join->type == JoinType::RIGHT) {
+     return StrongPredicate(expr, context);
+   }
+ } else if (parent_join->type == JoinType::OUTER) {
```

- Add unified join operator
- Write outer join rules (Associativity/Commutativity)

Summary of updates

```
+ // Associativity rules taken from
+ // http://15721.courses.cs.cmu.edu/spring2017/papers/14-optimizer1/p539-moerkotte.pdf
+ /*
+  * (A inner B) inner C = A inner (B inner C)
+  * (A right B) inner C = A right (B inner C)
+  * (A inner B) left  C = A inner (B left  C)
+  * (A left  B) left  C = A left  (B left  C) with Strong Predicates PBC
+  * (A full  B) left  C = A full  (B left  C) with Strong Predicate PBC
+  * (A right B) right C = A right (B right C) with Strong Predicate PAB
+  * (A right B) full  C = A right (B full  C) with Strong Predicate PAB
+  * (A full  B) full  C = A full  (B full  C) with Strong Predicate PAB & PBC
+  */
+ auto parent_join = expr->Op().As<LogicalJoin>();
+ std::vector<std::shared_ptr<OperatorExpression>> children = expr->Children();
+ auto child_join = children[0]->Op().As<LogicalJoin>();
+ if (parent_join->type == JoinType::INNER) {
+   if (child_join->type == JoinType::INNER || child_join->type == JoinType::RIGHT) {
+     return true;
+   }
+ } else if (parent_join->type == JoinType::LEFT) {
+   if (child_join->type == JoinType::INNER) {
+     return true;
+   } else if (child_join->type == JoinType::LEFT || child_join->type == JoinType::OUTER) {
+     return StrongPredicate(expr, context);
+   }
+ } else if (parent_join->type == JoinType::RIGHT) {
+   if (child_join->type == JoinType::RIGHT) {
+     return StrongPredicate(expr, context);
+   }
+ } else if (parent_join->type == JoinType::OUTER) {
```

- Add unified join operator
- Write outer join rules (Associativity/Commutativity)
- Fix parse tree to operator tree transform

Summary of updates

```
+ // Associativity rules taken from
+ // http://15721.courses.cs.cmu.edu/spring2017/papers/14-optimizer1/p539-moerkotte.pdf
+ /*
+  * (A inner B) inner C = A inner (B inner C)
+  * (A right B) inner C = A right (B inner C)
+  * (A inner B) left  C = A inner (B left  C)
+  * (A left  B) left  C = A left  (B left  C) with Strong Predicates PBC
+  * (A full  B) left  C = A full  (B left  C) with Strong Predicate PBC
+  * (A right B) right C = A right (B right C) with Strong Predicate PAB
+  * (A right B) full  C = A right (B full  C) with Strong Predicate PAB
+  * (A full  B) full  C = A full  (B full  C) with Strong Predicate PAB & PBC
+  */
+ auto parent_join = expr->Op().As<LogicalJoin>();
+ std::vector<std::shared_ptr<OperatorExpression>> children = expr->Children();
+ auto child_join = children[0]->Op().As<LogicalJoin>();
+ if (parent_join->type == JoinType::INNER) {
+   if (child_join->type == JoinType::INNER || child_join->type == JoinType::RIGHT) {
+     return true;
+   }
+ } else if (parent_join->type == JoinType::LEFT) {
+   if (child_join->type == JoinType::INNER) {
+     return true;
+   } else if (child_join->type == JoinType::LEFT || child_join->type == JoinType::OUTER) {
+     return StrongPredicate(expr, context);
+   }
+ } else if (parent_join->type == JoinType::RIGHT) {
+   if (child_join->type == JoinType::RIGHT) {
+     return StrongPredicate(expr, context);
+   }
+ } else if (parent_join->type == JoinType::OUTER) {
```

- Add unified join operator
- Write outer join rules (Associativity/Commutativity)
- Fix parse tree to operator tree transform
- Add some tests

Query rewrites

Query rewrites - short description

SQL optimizations that **don't** need a cost model

- Constant folding / propagation

Query rewrites - short description

SQL optimizations that **don't** need a cost model

- Constant folding / propagation
- Transitive predicates

Query rewrites - short description

SQL optimizations that **don't** need a cost model

- Constant folding / propagation
- Transitive predicates
- Trivial predicate elimination

Constant folding

```
SELECT * FROM tb1
  WHERE tb1.a = 4 * 5
```

To

```
SELECT * FROM tb1
  WHERE tb1.a = 20
```

Transitive predicates

```
SELECT * FROM tb1, tb2
  WHERE tb1.a = tb2.b AND tb1.a = tb2.c
```

To

```
SELECT * FROM tb1, tb2
  WHERE tb1.a = tb2.b AND tb1.a = tb2.c
        AND tb2.b = tb2.c
```

Trivial elimination

```
SELECT * FROM tb1
  WHERE tb1.a = 5 AND tb1.b = tb1.b
```

To

```
SELECT * FROM tb1
  WHERE tb1.a = 5
```

Summary of updates

```
99  + // Associativity rules taken from
100 + // http://15721.courses.cs.cmu.edu/spring2017/papers/14-optimizer1/p539-moerkotte.pdf
101 + /*
102 +  * (A inner B) inner C = A inner (B inner C)
103 +  * (A right B) inner C = A right (B inner C)
104 +  * (A inner B) left  C = A inner (B left  C)
105 +  * (A left  B) left  C = A left  (B left  C) with Strong Predicates PBC
106 +  * (A full  B) left  C = A full  (B left  C) with Strong Predicate PBC
107 +  * (A right B) right C = A right (B right C) with Strong Predicate PAB
108 +  * (A right B) full  C = A right (B full  C) with Strong Predicate PAB
109 +  * (A full  B) full  C = A full  (B full  C) with Strong Predicate PAB & PBC
110 +  */
111 + auto parent_join = expr->Op().As<LogicalJoin>();
112 + std::vector<std::shared_ptr<OperatorExpression>> children = expr->Children();
113 + auto child_join = children[0]->Op().As<LogicalJoin>();
114 + if (parent_join->type == JoinType::INNER) {
115 +   if (child_join->type == JoinType::INNER || child_join->type == JoinType::RIGHT) {
116 +     return true;
117 +   }
118 + } else if (parent_join->type == JoinType::LEFT) {
119 +   if (child_join->type == JoinType::INNER) {
120 +     return true;
121 +   } else if (child_join->type == JoinType::LEFT || child_join->type == JoinType::OUTER) {
122 +     return StrongPredicate(expr, context);
123 +   }
124 + } else if (parent_join->type == JoinType::RIGHT) {
125 +   if (child_join->type == JoinType::RIGHT) {
126 +     return StrongPredicate(expr, context);
127 +   }
128 + } else if (parent_join->type == JoinType::OUTER) {
```

- Implemented constant folding as parser step;

Summary of updates

```
99  + // Associativity rules taken from
100 + // http://15721.courses.cs.cmu.edu/spring2017/papers/14-optimizer1/p539-moerkotte.pdf
101 + /*
102 +  * (A inner B) inner C = A inner (B inner C)
103 +  * (A right B) inner C = A right (B inner C)
104 +  * (A inner B) left  C = A inner (B left  C)
105 +  * (A left  B) left  C = A left  (B left  C) with Strong Predicates PBC
106 +  * (A full  B) left  C = A full  (B left  C) with Strong Predicate PBC
107 +  * (A right B) right C = A right (B right C) with Strong Predicate PAB
108 +  * (A right B) full  C = A right (B full  C) with Strong Predicate PAB
109 +  * (A full  B) full  C = A full  (B full  C) with Strong Predicate PAB & PBC
110 +  */
111 + auto parent_join = expr->Op().As<LogicalJoin>();
112 + std::vector<std::shared_ptr<OperatorExpression>> children = expr->Children();
113 + auto child_join = children[0]->Op().As<LogicalJoin>();
114 + if (parent_join->type == JoinType::INNER) {
115 +   if (child_join->type == JoinType::INNER || child_join->type == JoinType::RIGHT) {
116 +     return true;
117 +   }
118 + } else if (parent_join->type == JoinType::LEFT) {
119 +   if (child_join->type == JoinType::INNER) {
120 +     return true;
121 +   } else if (child_join->type == JoinType::LEFT || child_join->type == JoinType::OUTER) {
122 +     return StrongPredicate(expr, context);
123 +   }
124 + } else if (parent_join->type == JoinType::RIGHT) {
125 +   if (child_join->type == JoinType::RIGHT) {
126 +     return StrongPredicate(expr, context);
127 +   }
128 + } else if (parent_join->type == JoinType::OUTER) {
```

- Implemented constant folding as parser step;
- Implemented transitive predicates as optimizer rule;

Summary of updates

```
+ // Associativity rules taken from
+ // http://15721.courses.cs.cmu.edu/spring2017/papers/14-optimizer1/p539-moerkotte.pdf
+ /*
+  * (A inner B) inner C = A inner (B inner C)
+  * (A right B) inner C = A right (B inner C)
+  * (A inner B) left  C = A inner (B left  C)
+  * (A left  B) left  C = A left  (B left  C) with Strong Predicates PBC
+  * (A full  B) left  C = A full  (B left  C) with Strong Predicate PBC
+  * (A right B) right C = A right (B right C) with Strong Predicate PAB
+  * (A right B) full  C = A right (B full  C) with Strong Predicate PAB
+  * (A full  B) full  C = A full  (B full  C) with Strong Predicate PAB & PBC
+  */
+ auto parent_join = expr->Op().As<LogicalJoin>();
+ std::vector<std::shared_ptr<OperatorExpression>> children = expr->Children();
+ auto child_join = children[0]->Op().As<LogicalJoin>();
+ if (parent_join->type == JoinType::INNER) {
+   if (child_join->type == JoinType::INNER || child_join->type == JoinType::RIGHT) {
+     return true;
+   }
+ } else if (parent_join->type == JoinType::LEFT) {
+   if (child_join->type == JoinType::INNER) {
+     return true;
+   } else if (child_join->type == JoinType::LEFT || child_join->type == JoinType::OUTER) {
+     return StrongPredicate(expr, context);
+   }
+ } else if (parent_join->type == JoinType::RIGHT) {
+   if (child_join->type == JoinType::RIGHT) {
+     return StrongPredicate(expr, context);
+   }
+ } else if (parent_join->type == JoinType::OUTER) {
```

- Implemented constant folding as parser step;
- Implemented transitive predicates as optimizer rule;
- Implemented trivial elimination as optimizer rule;

Demo

Demo

```
postgres=# INSERT INTO t2 VALUES(1,2,3);
INSERT 0 1
postgres=# INSERT INTO t2 VALUES(2,3,4);
INSERT 0 1
postgres=# INSERT INTO t2 VALUES(3,4,5);
INSERT 0 1
postgres=# CREATE TABLE t3(d INT);
CREATE TABLE
postgres=# INSERT INTO t3 VALUES(3);
INSERT 0 1
postgres=# INSERT INTO t3 VALUES(4);
INSERT 0 1
postgres=# INSERT INTO t3 VALUES(5);
INSERT 0 1
postgres=# SELECT * FROM t3 LEFT OUTER JOIN (t1 INNER JOIN t2 ON t1.a=t2.d) on t3.d = t2.d WHERE t2.c = 2;
 b | c | d | a | b | c | d
---+---+---+---+---+---+---
 1 | 2 | 3 | 3 | 4 | 5 | 3
   |   |   |   |   |   | 5
   |   |   |   |   |   | 4
(3 rows)

postgres=# explain SELECT * FROM t3 LEFT OUTER JOIN (t1 INNER JOIN t2 ON t1.a=t2.d) on t3.d = t2.d WHERE t2.c = 2;
                        Query plan
-----------------------------------------------------------
 -----------------------------------------------------------
 -> Plan Type: HASHJOIN
    Info: HashJoin(Expression ::
  expression type = Conjunction,
  conjunction type = CONJUNCTION_AND
   Expression ::
    expression type = Comparison,
    comparison type = COMPARE_EQUAL
     Expression ::
      expression type = Tuple Value,
      table name: t2
      column name: c
     Expression ::
      expression type = Constant Value,
      value: INTEGER(2)
   Expression ::
    expression type = Comparison,
    comparison type = COMPARE_EQUAL
     Expression ::
      expression type = Tuple Value,
      table name: t3
      column name: d
     Expression ::
      expression type = Tuple Value,
      table name: t2
      column name: d
 )
    NumChildren: 2
    -> Plan Type: SEQSCAN
       Info: SeqScan()
       NumChildren: 0
    -> Plan Type: HASH
       Info: Hash
       NumChildren: 1
       -> Plan Type: HASHJOIN
          Info: HashJoin(Expression ::
  expression type = Comparison,
  comparison type = COMPARE_EQUAL
   Expression ::
    expression type = Tuple Value,
    table name: t1
    column name: a
   Expression ::
```

Thanks!