SparkSQL子查询源码阅读

笔者近期研究了一段时间Spark的源码，主要集中在SparkSQL上，写下一篇阅读笔记，记录SparkSQL中对子查询的处理源码，如果有同学同样在研究这一部分，可以作为部分参考，文中均为笔者个人对源码的理解，难免会有一些错误的解读，欢迎批评指正，一同学习(email: hanmingcong123@hotmail.com)。

目录

Spark源码版本2.4.x

笔者研究的是当前master分支的源码，也提交了几个相关的patch，但是本文还是要以2.4.x版本（目前最高是2.4.3版本，在2.4.0-2.4.3之间没有相关的大的改动）。

基础知识

先对本文阅读的对象做一个简单的介绍，子查询分为很多种，这里我们指的是FROM子句外出现的子查询（在Postgres里叫做子连接Sublink）。在SQL92里，对子查询分了三类：

- 标量子查询（Scalar-valued Subquery），指的是子查询返回一个标量，也就是一行一列，常见的是带有聚集算子的,如果返回值不是一行的话，会报error。

```
-- Simplified version of TPCDS-Q32
-- From spark/sql/core/test/resources/sql-tests/inputs/subquery/scalar-subquery/scalar-subquery-predicate.sql
SELECT pk, cv
FROM   p, c
WHERE  p.pk = c.ck
AND    c.cv = (SELECT avg(c1.cv)
               FROM   c c1
               WHERE  c1.ck = p.pk);
```

- 存在性检测（existential test），指的是Exists谓词的子查询，如果子查询内的结果不为空（即使只返回一行null）则返回true，否则返回false。值得指出的是Exists的结果只有T or F，不会出现null。

```
-- Simple correlated predicate in exist subquery
-- From spark/sql/core/test/resources/sql-tests/inputs/subquery/exists-subquery/exists-basic.sql
SELECT *
FROM   emp
WHERE  EXISTS (SELECT dept.dept_name
               FROM   dept
               WHERE  emp.dept_id = dept.dept_id);
```

- 集合比较（Quantified Comparision），指的是IN/ANY/ALL类型子查询，这里只介绍IN类型子查询，因为SparkSQL目前还**没有支持ANY/ALL谓词**（已有相关的PR在完善）。IN子查询的一般表达方式是`x in (subquery)`,并且是支持多列查询的`(x, y) in (select a,b from ...)`，IN子查询与Exists不同，它是可以返回null的，例如`3 in (1, 2, null)`返回的不是false而应该是null，因为null的语义是Unknown，那么3 到底等不等于这个null也是Unknown的，因此返回的是null，但是如果是`3 in (1, 2, 3, null)`那么有一个3是满足的，所以应该返回true。

```
-- From spark/sql/core/test/resources/sql-tests/inputs/subquery/in-subquery/simple-in.sql
SELECT *
FROM   t1
WHERE  t1b IN (SELECT t2b
               FROM   t2);
```

还有另外一种区分子查询的方式，即关联、非关联子查询（Correlated/Non-Correlated Subquery)，指的是子查询中是否有外表的引用(outer reference)，上面的前两个子查询都是关联子查询，最后一个则是非关联的子查询。

子查询的难点在与去关联化（Unnesting/Decorrelation)，在传统数据库中，提供了NestedLoop的执行方式，即对于父表的每一行执行一次子查询。但是在SparkSQL这种分布式场景下，NestedLoop显然代价就有些太高了，而且很难在逻辑上支持这种执行方式，因为他是Plan-Expression-Plan这种执行方式，在计算Expression时要执行一个Plan，在RDD这种模型下很难实现（即不能在一个RDD里计算另一个RDD）。下面我们看一下SparkSQL是如何把这些Expression重写成Plan的。

将子查询重写为Join

这里我们先看一下子查询应该如何等价转换成为Join，推荐一篇博文SQL子查询优化，其中主要讲了Microsoft SQLServer中Unnesting Subquery的方法。我们在这里给出几个比较典型的例子，以及SparkSQL转换的结果。

```
两个Table的Schema以及Data
scala> spark.sql("select * from t1").show()
+---+---+
|t1a|t1b|
+---+---+
|  1|  1|
|  2|  2|
|  3|  3|
```

```
+---+---+
scala> spark.sql("select * from t2").show()
+---+----+
|t2a| t2b|
+---+----+
|  1|   1|
|  2|null|
|  3|   3|
+---+----+
```

```sql
-- Q1 Correlated EXISTS
SELECT *
FROM   t1
WHERE  EXISTS (SELECT *
               FROM   t2
               WHERE  t2b = t1b)
```

```
== Analyzed Logical Plan ==
t1a: string, t1b: string
Project [t1a#35, t1b#36]
+- Filter exists#75 [t1b#36]
   :  +- Project [t2a#60, t2b#61]
   :     +- Filter (t2b#61 = outer(t1b#36))
   :        +- SubqueryAlias `t2`
   :           +- Relation[t2a#60,t2b#61] csv
   +- SubqueryAlias `t1`
      +- Relation[t1a#35,t1b#36] csv

这里很好理解，SemiJoin本身就是为EXISTS而生的，我们把Correlated的Condition提上来，然后将
父表和子表做一个SemiJoin即可，这个是最标准的形式。

== Optimized Logical Plan ==
Join LeftSemi, (t2b#61 = t1b#36) //原本是Filter里的Condition，我们把他提到Join里来
:- Relation[t1a#35,t1b#36] csv
+- Project [t2b#61]
   +- Relation[t2a#60,t2b#61] csv
```

```sql
-- Q2 Correlated IN with a Non-Correlated Subquery
-- 这里稍微解释一下这个查询，看上去子查询里是没有外引用的，
-- 但我一般把这个查询叫做关联的IN（Correlated IN）虽然它的子查询是一个非关联的，
-- 但是由于IN的语义是判断左边的值是否在右边的集合里，故存在关联性
SELECT *
FROM   t1
WHERE  t1b IN (SELECT t2b
               FROM   t2
               WHERE  t2a > 1)
-- 实际上我们可以认为这个查询等价于下面这个
```

```sql
SELECT *
FROM   t1
WHERE  EXISTS (SELECT t2b
               FROM   t2
               WHERE  t2a > 1 AND t1b = t2b)
-- 这样的话就和上面的处理方式一样了对吧？我们后面讲一下这个转换的正确性
```

```
== Analyzed Logical Plan ==
t1a: string, t1b: string
Project [t1a#35, t1b#36]
+- Filter t1b#36 IN (list#79 [])
   :  +- Project [t2b#61]
   :     +- Filter (cast(t2a#60 as int) > 1)
   :        +- SubqueryAlias `t2`
   :           +- Relation[t2a#60,t2b#61] csv
   +- SubqueryAlias `t1`
      +- Relation[t1a#35,t1b#36] csv

== Optimized Logical Plan ==
Join LeftSemi, (t1b#36 = t2b#61)
:- Relation[t1a#35,t1b#36] csv
+- Project [t2b#61]
   +- Filter (isnotnull(t2a#60) && (cast(t2a#60 as int) > 1))
      +- Relation[t2a#60,t2b#61] csv
```

那么上面这个转换到底对不对呢？我再推荐一篇文档，MySQL EXISTS Strategy，这篇文档讲了MySQL里何种情况的IN才能转换为EXISTS。我们上文提到过，InSubquery是nullable的，但Exists不是，所以这之间是有区别的，我们把上面这个查询改变一下。

```sql
SELECT *
FROM   t1
WHERE  (t1b IN (SELECT t2b
               FROM   t2
               WHERE  t2a > 1)) IS NULL
```

```
== Analyzed Logical Plan ==
t1a: string, t1b: string
Project [t1a#35, t1b#36]
+- Filter isnull(t1b#36 IN (list#83 []))
   :  +- Project [t2b#61]
   :     +- Filter (cast(t2a#60 as int) > 1)
   :        +- SubqueryAlias `t2`
   :           +- Relation[t2a#60,t2b#61] csv
   +- SubqueryAlias `t1`
      +- Relation[t1a#35,t1b#36] csv

== Optimized Logical Plan ==
```

```
Project [t1a#35, t1b#36]
+- Filter isnull(exists#87)
   +- Join ExistenceJoin(exists#87), (t1b#36 = t2b#61) 可以看到这里不再用SemiJoin
了，而是使用一个ExistenceJoin，我们show一下结果
      :- Relation[t1a#35,t1b#36] csv
      +- Project [t2b#61]
         +- Filter (isnotnull(t2a#60) && (cast(t2a#60 as int) > 1))
            +- Relation[t2a#60,t2b#61] csv

scala> spark.sql(q3).show()
+---+---+
|t1a|t1b|
+---+---+
+---+---+
```

很明显这个结果是不对的，外查询的t1b应该是[1, 2, 3]，子查询的结果应该是[null, 3],那么这个IN的结果应该分别是[null, null, true],所以最后应该有两行，实际上SparkSQL目前没有完全准确地处理InSubquery的null值，核心问题就是这个ExistenceJoin不会产生null值，SPARK-27572提出了这个问题，但在实际中这个问题一般不会影响结果，因为很少会有人去针对InSubquery的返回值做null 和 false的区分，我们只要对NOT IN做特殊处理就好（因为Not null = null而Not false = true），在不区分null值时In的确可以转换为Exists。

```sql
-- Correlated IN with Aggregation
SELECT t1a
FROM   t1
WHERE  t1a IN (SELECT MAX(t2a)
               FROM   t2
               WHERE  t2b = t1b)
```

上面这个查询是带聚集的关联子查询，由于在聚集之前有一个带关联条件的Filter，因此不能提前聚集再做Join，那怎么办呢？我们可以看到实际上我们只要针对不同的t2b做聚集就好，也就是Group By t2b，于是这个查询等价于下面这个SQL。

```sql
SELECT t1a
FROM   t1
WHERE  (t1a, t1b) IN (SELECT MAX(t2a), t2b
                      FROM t2
                      GROUP BY t2b)
```

我们看下之前那个SQL的查询计划：

```
== Analyzed Logical Plan ==
t1a: string
Project [t1a#35]
+- Filter t1a#35 IN (list#103 [t1b#36])
   :  +- Aggregate [max(t2a#60) AS max(t2a)#109]
   :     +- Filter (t2b#61 = outer(t1b#36))
```

```
   :        +- SubqueryAlias `t2`
   :           +- Relation[t2a#60,t2b#61] csv
   +- SubqueryAlias `t1`
      +- Relation[t1a#35,t1b#36] csv

== Optimized Logical Plan ==
Project [t1a#35]
+- Join LeftSemi, ((t1a#35 = max(t2a)#109) && (t2b#61 = t1b#36))
   :- Relation[t1a#35,t1b#36] csv
   +- Aggregate [t2b#61], [max(t2a#60) AS max(t2a)#109, t2b#61] // 前面是GroupBy，
可以看到增加了t2b
      +- Relation[t2a#60,t2b#61] csv
```

上面给出了Exists和InSubquery的例子，下面们继续看一下ScalarSubquery的例子，SparkSQL中只支持带聚集的ScalarSubquery，同时子查询里如果包含GroupBy的话，GroupBy后面必须是外引用。换句话说，SparkSQL在编译阶段就需要确保ScalarSubquery只返回一行一列。

```
SELECT t1a
FROM   t1
WHERE  t1a > (SELECT max(t2a)
              FROM   t2
              WHERE  t2b = t1b)
```

```
== Analyzed Logical Plan ==
t1a: string
Project [t1a#35]
+- Filter (t1a#35 > scalar-subquery#112 [t1b#36])
   :  +- Aggregate [max(t2a#60) AS max(t2a)#118]
   :     +- Filter (t2b#61 = outer(t1b#36))
   :        +- SubqueryAlias `t2`
   :           +- Relation[t2a#60,t2b#61] csv
   +- SubqueryAlias `t1`
      +- Relation[t1a#35,t1b#36] csv

== Optimized Logical Plan ==
Project [t1a#35]
// 这里把子查询重写成了一个 Inner Join，我们看到了t2b = t1b被提到的Join的Condition里，
同时Aggregate的GroupBy也加入了t2b，这个查询和上面的InSubquery的区别就是这里使用了Inner
Join
+- Join Inner, ((t1a#35 > max(t2a)#118) && (t2b#61 = t1b#36))
   :- Filter (isnotnull(t1a#35) && isnotnull(t1b#36))
   :  +- Relation[t1a#35,t1b#36] csv
   +- Filter isnotnull(max(t2a)#118)
      +- Aggregate [t2b#61], [max(t2a#60) AS max(t2a)#118, t2b#61]
         +- Filter isnotnull(t2b#61)
            +- Relation[t2a#60,t2b#61] csv
```

在重写ScalarSubquery时，有个很出名的问题叫做**COUNT BUG**，我们后面遇到的时候再介绍。

下面我们阅读以下SparkSQL2.4.x版本里对三类子查询的处理。 我假设读者对Catalyst的整体框架是有所了解的，包括对Rule的定义，Expression的定义，LogicalPlan的定义。

三个子查询Expression的定义

首先看这三个case class的定义。

```
case class Exists(
    plan: LogicalPlan, // 子查询的LogicalPlan
    children: Seq[Expression] = Seq.empty, // 子查询中的outer reference，在analyze阶段会被记录在里面
    exprId: ExprId = NamedExpression.newExprId)
  extends SubqueryExpression(plan, children, exprId) with Predicate with Unevaluable {
  override def nullable: Boolean = false
  override def withNewPlan(plan: LogicalPlan): Exists = copy(plan = plan)
  override def toString: String = s"exists#${exprId.id} $conditionString"
  override lazy val canonicalized: Expression = {
    Exists(
      plan.canonicalized,
      children.map(_.canonicalized),
      ExprId(0))
  }
}
```

Exists里面也没什么太值得提的，需要说的一点是这个children，这个没找到有注释解释，刚开始看源码的时候在思考，Exists为什么会把children放在参数里？看到后面的Rule之后，才意识到**children里存的是plan里的outer reference**，也就是子查询里对外表的引用会被暂存在children里方便做一些操作。

```
case class ScalarSubquery(
    plan: LogicalPlan,
    children: Seq[Expression] = Seq.empty,
    exprId: ExprId = NamedExpression.newExprId)
  extends SubqueryExpression(plan, children, exprId) with Unevaluable {
  override def dataType: DataType = plan.schema.fields.head.dataType
  override def nullable: Boolean = true
  override def withNewPlan(plan: LogicalPlan): ScalarSubquery = copy(plan = plan)
  override def toString: String = s"scalar-subquery#${exprId.id} $conditionString"
  override lazy val canonicalized: Expression = {
    ScalarSubquery(
      plan.canonicalized,
      children.map(_.canonicalized),
      ExprId(0))
  }
}
```

ScalarSubquery也没有特殊的地方，与Exists是差不多的。

```
case class InSubquery(values: Seq[Expression], query: ListQuery)
  extends Predicate with Unevaluable {

  @transient lazy val value: Expression = if (values.length > 1) {
    CreateNamedStruct(values.zipWithIndex.flatMap {
      case (v: NamedExpression, _) => Seq(Literal(v.name), v)
      case (v, idx) => Seq(Literal(s"_$idx"), v)
    })
  } else {
    values.head
  } // 如果InSubquery的left value是多列的话,比如(a, b)这样的，变量value应该是一个
NamedStruct


  override def checkInputDataTypes(): TypeCheckResult = {
      // 检查values和query的输出类型是否一致
  }

 // 这里的children可就不是outer reference了
  override def children: Seq[Expression] = values :+ query
  override def nullable: Boolean = children.exists(_.nullable)
  override def foldable: Boolean = children.forall(_.foldable)
  override def toString: String = s"$value IN ($query)"
  override def sql: String = s"(${value.sql} IN (${query.sql}))"
}
```

InSubquery和那两个类型有区别，InSubquery在文件predicates.scala里，但是另外两个在subquery.scala里，很容易想到原因，因为InSubquery除了子查询还有一个left value作为输入参数，所以InSubquery实际上是个Binary的表达式，而另外两个都是Unary的（甚至可以认为是LeafExpression，这不重要）。因此InSubquery的子查询被ListQuery这个类给包装起来了，我们看这个类的定义。

```
 // A [[ListQuery]] expression defines the query which we want to search in an IN
subquery
 // expression. It should and can only be used in conjunction with an IN
expression.
 // ListQuery是给InSubquery专用的
case class ListQuery(
    plan: LogicalPlan,
    children: Seq[Expression] = Seq.empty, //这里就是outer reference了
    exprId: ExprId = NamedExpression.newExprId,
    childOutputs: Seq[Attribute] = Seq.empty)
  extends SubqueryExpression(plan, children, exprId) with Unevaluable {
  override def dataType: DataType = if (childOutputs.length > 1) {
    childOutputs.toStructType
  } else {
    childOutputs.head.dataType
  }
  override lazy val resolved: Boolean = childrenResolved && plan.resolved &&
childOutputs.nonEmpty
  override def nullable: Boolean = false // 注意这里，实际上是不对的，如果这个
```

```
nullable是false的话，那么InSubquery的nullable只取决于values，因此SparkSQL没有正确的实
现InSubquery，我们后文详细讨论这件事情
  override def withNewPlan(plan: LogicalPlan): ListQuery = copy(plan = plan)
  override def toString: String = s"list#${exprId.id} $conditionString"
  override lazy val canonicalized: Expression = {
    ListQuery(
      plan.canonicalized,
      children.map(_.canonicalized),
      ExprId(0),
      childOutputs.map(_.canonicalized.asInstanceOf[Attribute]))
  }
}
```

Analyzer：ResolveSubquery

这里关注Analyzer Rule: ResolveSubquery。先概述一下这个规则，这个规则有两个作用，一个是将子查询里的外引用解析出来，转换成OuterReference；另一个作用就是将OuterReference放在各个SubqueryExpression的children里，方便后面使用。

代码顺序有所调整，为了阅读方便。

```
  object ResolveSubquery extends Rule[LogicalPlan] with PredicateHelper {
    ...
    /**
     * Resolve and rewrite all subqueries in an operator tree..
     */
    def apply(plan: LogicalPlan): LogicalPlan = plan.resolveOperatorsUp {
      // In case of HAVING (a filter after an aggregate) we use both the aggregate
and
      // its child for resolution.
      case f @ Filter(_, a: Aggregate) if f.childrenResolved =>
        resolveSubQueries(f, Seq(a, a.child))
      // Only a few unary nodes (Project/Filter/Aggregate) can contain subqueries.
      case q: UnaryNode if q.childrenResolved =>
        resolveSubQueries(q, q.children)
    }
  }
```

```
    /**
     * Resolves the subquery. Apart of resolving the subquery and outer references
(if any)
     * in the subquery plan, the children of subquery expression are updated to
record the
     * outer references. This is needed to make sure
     * (1) The column(s) referred from the outer query are not pruned from the
plan during
     *     optimization.
     * (2) Any aggregate expression(s) that reference outer attributes are pushed
down to
```

```
     *     outer plan to get evaluated.
     */
    private def resolveSubQueries(plan: LogicalPlan, plans: Seq[LogicalPlan]):
LogicalPlan = {
      plan transformExpressions {
        case s @ ScalarSubquery(sub, _, exprId) if !sub.resolved =>
          // 解析外引用后返回新的Expression, 下同
          resolveSubQuery(s, plans)(ScalarSubquery(_, _, exprId))
        case e @ Exists(sub, _, exprId) if !sub.resolved =>
          resolveSubQuery(e, plans)(Exists(_, _, exprId))
        case InSubquery(values, l @ ListQuery(_, _, exprId, _))
            if values.forall(_.resolved) && !l.resolved =>
          //上文提到过InSubquery不是一个SubqueryExpression, 所以要解析它的ListQuery,
然后重新构造一个InSubquery
          val expr = resolveSubQuery(l, plans)((plan, exprs) => {
            ListQuery(plan, exprs, exprId, plan.output)
          })
          val subqueryOutput = expr.plan.output
          val resolvedIn = InSubquery(values, expr.asInstanceOf[ListQuery])
          // 这里经过resolve OuterReference之后, 需要检查InSubquery两边的长度
          // 实际上在InSubquery的checkInputDataTypes里有一个完全一样的check
          // 因此这个check在3.0版本已经移除[SPARK-24341]
          if (values.length != subqueryOutput.length) {
            throw new AnalysisException(...)
          }
          resolvedIn
      }
    }
```

```
  /**
     * Resolves the subquery plan that is referenced in a subquery expression. The
normal
     * attribute references are resolved using regular analyzer and the outer
references are
     * resolved from the outer plans using the resolveOuterReferences method.
     *
     * Outer references from the correlated predicates are updated as children of
     * Subquery expression.
     */
    private def resolveSubQuery(
        e: SubqueryExpression,
        plans: Seq[LogicalPlan])(
        f: (LogicalPlan, Seq[Expression]) => SubqueryExpression):
SubqueryExpression = {
      // Step 1: Resolve the outer expressions.
      // 第一步先把outer reference给解析了
      var previous: LogicalPlan = null
      var current = e.plan
      do {
        // Try to resolve the subquery plan using the regular analyzer.
        previous = current
```

```
          // 要先把子查询这个plan调用一次Analyzer的execute,
          // 宏观来看就是一个递归的Analysis
          current = executeSameContext(current)

          // Use the outer references to resolve the subquery plan if it isn't
resolved yet.
          val i = plans.iterator
          val afterResolve = current
          while (!current.resolved && current.fastEquals(afterResolve) && i.hasNext)
{
            // 在这个循环里，依次使用plans里的LogicalPlan作为参照来解析outerreference
            current = resolveOuterReferences(current, i.next())
          }
          // 迭代到子查询里的外引用全部被解析或者不再产生变化
        } while (!current.resolved && !current.fastEquals(previous))

        // Step 2: If the subquery plan is fully resolved, pull the outer references
and record
        // them as children of SubqueryExpression.
        // 第二步就是要把外引用放到children里
        if (current.resolved) {
          // Record the outer references as children of subquery expression.
          // 这个f就是SubqueryExpression的构造函数
          f(current, SubExprUtils.getOuterReferences(current))
        } else {
          e.withNewPlan(current)
        }
      }
```

```
    /**
     * Resolve the correlated expressions in a subquery by using the an outer
plans' references. All
     * resolved outer references are wrapped in an [[OuterReference]]
     */
    private def resolveOuterReferences(plan: LogicalPlan, outer: LogicalPlan):
LogicalPlan = {
        plan resolveOperatorsDown {
          case q: LogicalPlan if q.childrenResolved && !q.resolved =>
            q transformExpressions {
              // 把能够通过outerPlan resolve的UnresolvedAttribute换成OuterReference
              case u @ UnresolvedAttribute(nameParts) =>
                withPosition(u) {
                  try {
                    outer.resolve(nameParts, resolver) match {
                      case Some(outerAttr) => OuterReference(outerAttr)
                      case None => u
                    }
                  } catch {
                    case _: AnalysisException => u
                  }
                }
```

```
            }
        }
    }
```

Analyze: CheckAnalysis

熟悉Catalyst的同学都应该知道，经过Analyzer之后需要通过一个CheckAnalysis的过程来检查这个算子树是否有效。在CheckAnalysis里有一部分checkSubqueryExpression用于检查Subqeury的有效性，我们先简单地把几个限制总结一下：

1. ScalarSubquery只能返回一行一列
2. Correlated ScalarSubquery的output必须是一个聚集值(Aggregated value)
3. Correlated ScalarSubquery如果存在GroupBy，则GroupBy后面的值必须是outer reference
4. ScalarSubquery只能出现在Project,Filter,Aggregate里
5. InSubquery/Exists只能出现在Filter里
6. InSubquery中的OuterReference只能出现在Filter里

限制1和4可以理解，绝大部分的SQL引擎都是这样的。

但是限制2、3、5是SparkSQL的Optimizer的实现问题，以及受限于Spark的引擎。比如限制2，传统数据库是允许Correlated ScalarSubquery的子查询里不是聚集的，但是这样怎么满足限制1呢？传统数据库的执行引擎在Runtime的时候可以报运行时异常，如果使用NestedLoop的子查询返回的不是一行的话那就报运行时Error。而对于Spark来说，Spark是不能支持NestedLoop方式执行子查询的，因此只能把子查询写成Join（后面会详细讲），这样一来就必须在编译期确定Correlated ScalarSubquery必须只返回一行，能做到如此的也就只有聚集函数了。

```
/**
   * Validates subquery expressions in the plan. Upon failure, returns an user
facing error.
   */
  private def checkSubqueryExpression(plan: LogicalPlan, expr:
SubqueryExpression): Unit = {
    ...
    // Validate the subquery plan.
    // 同样递归的检查子查询的plan
    checkAnalysis(expr.plan)

    expr match {
      case ScalarSubquery(query, conditions, _) =>
        // Scalar subquery must return one column as output.
        // 子查询只能返回一列
        if (query.output.size != 1) {
          failAnalysis(
            s"Scalar subquery must return only one column, but got
${query.output.size}")
        }
        // 这里的目的是限制Correlated ScalarSubquery的最后Project 必须是一个聚集值
（aggregated value)
        if (conditions.nonEmpty) {
          // 这个函数返回的是一个查询树最上方的非SubqueryAlias和Project的算子
```

```
          // 例如一个树是 SubqueryAlias-Project-Filter-Projcet-Scan
          // 那么返回的就是 Filter-Project-Scan
          cleanQueryInScalarSubquery(query) match {
            // 这里面检查关联子查询中是否有Aggregate，如果同时有Group By的话，Group By
后面的值
            // 必须全部都是OuterReference，可以考虑一下这有什么含义呢？对于一个子查询来
说，OuterReference
            // 就是一个常量，与GroupBy 1 等价？
            case a: Aggregate => checkAggregateInScalarSubquery(conditions, query,
a)
            case Filter(_, a: Aggregate) =>
checkAggregateInScalarSubquery(conditions, query, a)
            case fail => failAnalysis(s"Correlated scalar subqueries must be
aggregated: $fail")
          }

          // Only certain operators are allowed to host subquery expression
containing
          // outer references.
          // 这里限制Correlated ScalarSubquery只能出现在Filter,Aggregate,Project算子
里
          plan match {
            case _: Filter | _: Aggregate | _: Project => // Ok
            case other => failAnalysis(
              "Correlated scalar sub-queries can only be used in a " +
                s"Filter/Aggregate/Project: $plan")
          }
        }

      // 这里限制InSubquery和Exists只能出现在Filter算子里，无论是否Correlated
      case inSubqueryOrExistsSubquery =>
        plan match {
          case _: Filter => // Ok
          case _ =>
            failAnalysis(s"IN/EXISTS predicate sub-queries can only be used in a
Filter: $plan")
        }
    }

    // Validate to make sure the correlations appearing in the query are valid and
    // allowed by spark.
    // 这个方法相当的长，里面有非常详细的注释，我们不展开分析源码，但我们给出主要的注释：
    // Whitelist operators allowed in a correlated InSubquery
    // There are 4 categories:
    // 1. Operators that are allowed anywhere in a correlated subquery, and,
    //    by definition of the operators, they either do not contain
    //    any columns or cannot host outer references.
    // 2. Operators that are allowed anywhere in a correlated subquery
    //    so long as they do not host outer references.
    // 3. Operators that need special handlings. These operators are
    //    Filter, Join, Aggregate, and Generate.
    // 4. Any operators that are not in the above list are allowed
    //    in a correlated subquery only if they are not on a correlation path.
    //    In other word, these operators are allowed only under a correlation
```

```
point.
    checkCorrelationsInSubquery(expr.plan)
  }
```

Optimize: OptimizeSubqueries

现在我们进入Optimize阶段，一共有四个比较重要的与子查询相关的Rule。第一个Rule就是OptimizeSubqueries，做了两件事，一个是递归的Optimize子查询的Plan，另一个是消除子查询顶部的Sort（因为最终结果的Sort不会影响子查询的结果对外部的影响，注意这里说的是顶部的Sort，也就是对最终结果的Sort）。

```
  /**
   * Optimize all the subqueries inside expression.
   */
  object OptimizeSubqueries extends Rule[LogicalPlan] {
    private def removeTopLevelSort(plan: LogicalPlan): LogicalPlan = {
      // 在这里消除最顶部的Sort，很容易理解，如果一个Sort上面只有Project或者没有别的算
子的话   就消除
      // 为什么只限定Project呢？为什么不限定Filter之类的算子呢？
      // 原因就是 这里已经是Optimized Plan了，也就是Sort上面不会有Filter的，Filter一定
已经被下推到Sort下面了
      plan match {
        case Sort(_, _, child) => child
        case Project(fields, child) => Project(fields, removeTopLevelSort(child))
        case other => other
      }
    }
    def apply(plan: LogicalPlan): LogicalPlan = plan transformAllExpressions {
      case s: SubqueryExpression =>
        // 递归的优化子查询树
        val Subquery(newPlan) = Optimizer.this.execute(Subquery(s.plan))
        // At this point we have an optimized subquery plan that we are going to
attach
        // to this subquery expression. Here we can safely remove any top level
sort
        // in the plan as tuples produced by a subquery are un-ordered.
        s.withNewPlan(removeTopLevelSort(newPlan))
    }
  }
```

接下来的三个Rule分别是PullupCorrelatedPredicates, RewritePredicateSubqeury,RewriteCorrelatedScalarSubquery,要记住我们现在的目的是，把所有能支持的子查询重写成Join，只有NonCorrelated ScalarSubquery可以使用一个SubPlan来执行，这个我们放到最后看。

Optimize: PullupCorrelatedPredicaets

这个Rule的目的有两点，一个是把子查询Filter里所有包含OuterReference的子式消除并提到SubqueryExpression里的children（回顾上文，这里的children是子查询里的OuterReference），另一个是把下层出现的所有与OuterReference有关系的LocalReference的都保留在上层的Project和Aggregate算子里，比如我

们上面给出里例子，在子查询是Aggregate的时候，需要把子查询Filter里的与OuterReference有关系的LocalReference加到GroupBy条件里，同样要放到最终的Project里，这样后面可以保证子查询会产生这一列，不然怎么拿它做Join条件呢？可能讲的有点乱，我们举个例子，比如：

```
SELECT *
FROM   t1
WHERE  t1a IN (SELECT t2a
               FROM t2
               WHERE t2b = t1b AND t2b > 10)
```

在这里面的SemiJoin条件应该是(t1a = t2a AND t1b = t2b)，我们第一步是把子查询里的Filter中t1b = t2b提到ListQuery的children里，保证子查询里不包含OuterReference，第二步就是把SELECT后面加上t2b，保证最后子查询的返回列是(t2a, t2b)，这样才能保证SemiJoin条件都合法。

```
/**
  * Pull out all (outer) correlated predicates from a given subquery. This method removes the
  * correlated predicates from subquery [[Filter]]s and adds the references of these predicates
  * to all intermediate [[Project]] and [[Aggregate]] clauses (if they are missing) in order to
  * be able to evaluate the predicates at the top level.
  *
  * TODO: Look to merge this rule with RewritePredicateSubquery.
  */
object PullupCorrelatedPredicates extends Rule[LogicalPlan] with PredicateHelper {
  ...
    /**
   * Pull up the correlated predicates and rewrite all subqueries in an operator tree..
   */
  def apply(plan: LogicalPlan): LogicalPlan = plan transformUp {
    case f @ Filter(_, a: Aggregate) =>
    // 这里和Analyzer里一样要单独处理Filter-Aggregate这种Pattern
      rewriteSubQueries(f, Seq(a, a.child))
    // Only a few unary nodes (Project/Filter/Aggregate) can contain subqueries.
    case q: UnaryNode =>
      rewriteSubQueries(q, q.children)
  }
}
```

```
  private def rewriteSubQueries(plan: LogicalPlan, outerPlans: Seq[LogicalPlan]): LogicalPlan = {
    plan transformExpressions {
      // 这里就是重写所有的SubqueryExpression，然后构造新的Expression
      case ScalarSubquery(sub, children, exprId) if children.nonEmpty =>
        val (newPlan, newCond) = pullOutCorrelatedPredicates(sub, outerPlans)
```

```
        ScalarSubquery(newPlan, newCond, exprId)
      case Exists(sub, children, exprId) if children.nonEmpty =>
        val (newPlan, newCond) = pullOutCorrelatedPredicates(sub, outerPlans)
        Exists(newPlan, newCond, exprId)
      case ListQuery(sub, _, exprId, childOutputs) =>
        val (newPlan, newCond) = pullOutCorrelatedPredicates(sub, outerPlans)
        ListQuery(newPlan, newCond, exprId, childOutputs)
    }
  }
```

```
  /**
   * Returns the correlated predicates and a updated plan that removes the outer
references.
   */
  private def pullOutCorrelatedPredicates(
      sub: LogicalPlan,
      outer: Seq[LogicalPlan]): (LogicalPlan, Seq[Expression]) = {
    // 一个mutable的Map用来保存子树带有OuterReference的Expression
    val predicateMap = scala.collection.mutable.Map.empty[LogicalPlan,
Seq[Expression]]

    /** Determine which correlated predicate references are missing from this
plan. */
    def missingReferences(p: LogicalPlan): AttributeSet = {
      // 这个函数提取出p的子树中的带有OuterReference的表达式中的LocalReference
      // 例如 t2a是个OuterReference，t1a = t2a是predicateMap里保存的一个Value
      // 那t1a就要保存在localPredicateReferences里，如果当前p里没有t1a的话，就要给p添
加这个t1a，保证外面可以拿t1a当做Join Condition
      // 这里这个p.collect(predicatemMap)是个很优雅的写法！！！
      // 大家可以思考为什么要用一个Map来存？ 因为我只希望补全我的子树中
      // 出现的Reference，比如子查询里有Join的话，我不希望看到另一颗子树中的Reference。
      val localPredicateReferences = p.collect(predicateMap)
        .flatten // 收集所有的Expression
        .map(_.references) // 提取出其中的AttributeSet
        .reduceOption(_ ++ _) // 再合并成一个AttributeSet
        .getOrElse(AttributeSet.empty)
      localPredicateReferences -- p.outputSet
    }

    // Simplify the predicates before pulling them out.
    // BooleanSimplification是个常见的Optimize Rule，可以认为是优化了子树
    val transformed = BooleanSimplification(sub) transformUp {
      case f @ Filter(cond, child) =>
      // Filter里要消除OuterReference存在的子式，同时把这个子式记录在predicateMap里
        val (correlated, local) =
          splitConjunctivePredicates(cond).partition(containsOuter)

        // Rewrite the filter without the correlated predicates if any.
        correlated match {
          case Nil => f
          case xs if local.nonEmpty =>
```

```
            val newFilter = Filter(local.reduce(And), child)
            predicateMap += newFilter -> xs
            newFilter
          case xs =>
            predicateMap += child -> xs
            child
        }
      case p @ Project(expressions, child) =>
      // Project里要添加Join需要的Reference
        val referencesToAdd = missingReferences(p)
        if (referencesToAdd.nonEmpty) {
          // 这里有个细节很重要，就是要用expressions ++ referencesToAdd而不能返过来
          // 因为后面的Rule在InSubquery里需要添加InSubquery的Condition，例如我们现在
          // 会把`t1a IN (select t2a from t2 where t1b = t2b)`转换成 `t1a IN
(select t2a, t2b from t2)`
          // 而我们后面的规则会把 t1a = t2a添加到Join Condition里，所以需要保证
Subquery前几个输出列不能变，新的列只能append到后面
          // 同样的，在ScalarSubquery里，我们会使用plan.output.head，所以要保证第一列
是原本的输出列
          Project(expressions ++ referencesToAdd, child)
        } else {
          p
        }
      case a @ Aggregate(grouping, expressions, child) =>
      // Aggregate要在GroupBy和Output里都添加Join需要的Reference
        val referencesToAdd = missingReferences(a)
        if (referencesToAdd.nonEmpty) {
          Aggregate(grouping ++ referencesToAdd, expressions ++ referencesToAdd,
child)
        } else {
          a
        }
      case p =>
        p
    }

    // Make sure the inner and the outer query attributes do not collide.
    // In case of a collision, change the subquery plan's output to use
    // different attribute by creating alias(s).
    ...
    // 这里是一些trivial code，目标是避免self join的时候出现冲突，我们暂不展开
```

至此，我们对SubqueryExpression的预处理全部结束了，我们保证了Subquery的Plan里不再包含OuterReference，所有的OuterReference都保存在Children里，下面我们可以直接将Subquery转换为Join。

Optimize: RewritePredicateSubquery

在这个Rule里，我们把Exists和InSubquery重写为Semi/Anti/Existential Join, 我们先解释一下这几个Join的功能。

- R semi-join(p) S，对于R的每一行若存在S中的一行满足p，则返回R的这一行。

- R anti-join(p) S, 对于R的每一行若存在S中的一行满足p，则拒绝R的这一行。换句话说，就是对于S中的每一行都**不满足p**时，才返回R的这一行。
- R existential-join(p) S, 与semi-join一样，只不过当满足时返回(R, true)，否则返回(R, false)，也就是把semi-join的结果使用一个新的Attribute标记。

```
/**
 * This rule rewrites predicate sub-queries into left semi/anti joins. The
following predicates
 * are supported:
 * a. EXISTS/NOT EXISTS will be rewritten as semi/anti join, unresolved conditions
in Filter
 *    will be pulled out as the join conditions.
 * b. IN/NOT IN will be rewritten as semi/anti join, unresolved conditions in the
Filter will
 *    be pulled out as join conditions, value = selected column will also be used
as join
 *    condition.
 */
object RewritePredicateSubquery extends Rule[LogicalPlan] with PredicateHelper {

  private def buildJoin(
      outerPlan: LogicalPlan,
      subplan: LogicalPlan,
      joinType: JoinType,
      condition: Option[Expression]): Join = {
    // Deduplicate conflicting attributes if any.
    val dedupSubplan = dedupSubqueryOnSelfJoin(outerPlan, subplan, None,
condition)
    Join(outerPlan, dedupSubplan, joinType, condition, JoinHint.NONE)
  }

  private def dedupSubqueryOnSelfJoin(...) = {
    // 这里也是一些trivial Code，也是在SelfJoin时会出现一些冲突，我们需要消解这些冲突
  }

  def apply(plan: LogicalPlan): LogicalPlan = plan transform {
    case Filter(condition, child) => // 我们上文提到过，目前只允许出现在Filter里
      // 把Condition中的合取式（AND连接的子式）分解，然后提取出包含IN/Exists的部分
      val (withSubquery, withoutSubquery) =

splitConjunctivePredicates(condition).partition(SubqueryExpression.hasInOrExistsSu
bquery)

      // 使用不包含Subquery的部分构造新的Filter
      // Construct the pruned filter condition.
      val newFilter: LogicalPlan = withoutSubquery match {
        case Nil => child
        case conditions => Filter(conditions.reduce(And), child)
      }

      // Filter the plan by applying left semi and left anti joins.
      // 每个Subquery都构造成一个Join，然后Fold成一颗树
```

```
      withSubquery.foldLeft(newFilter) {
        case (p, Exists(sub, conditions, _)) =>
          // 这个rewriteExistentialExpr的作用我们后文介绍，这里可以认为是产生
JoinCondition以及处理在父查询的查询树上先追加上一些ExistentialJoin
          val (joinCond, outerPlan) = rewriteExistentialExpr(conditions, p)
          // 构造成一个 outerPlan LeftSemi-Join(joinCond) sub的查询树
          buildJoin(outerPlan, sub, LeftSemi, joinCond)
        case (p, Not(Exists(sub, conditions, _))) =>
          val (joinCond, outerPlan) = rewriteExistentialExpr(conditions, p)
          // 上同，不过在Not Exists的情况下要重写成Anti-Join
          buildJoin(outerPlan, sub, LeftAnti, joinCond)
        case (p, InSubquery(values, ListQuery(sub, conditions, _, _))) =>
          // Deduplicate conflicting attributes if any.
          val newSub = dedupSubqueryOnSelfJoin(p, sub, Some(values))
          // inConditions是把InSubquery的leftValues和子查询里前几列用EqualTo组合起来,
          // 这里直接使用zip，是因为在PullupCorrelatedPredicate里我们只是append了新
列,
          // 子查询原本的输出列保证在最前面没有变化，这样才能用zip
          val inConditions = values.zip(newSub.output).map(EqualTo.tupled)
          val (joinCond, outerPlan) = rewriteExistentialExpr(inConditions ++
conditions, p)
          Join(outerPlan, newSub, LeftSemi, joinCond, JoinHint.NONE)
        case (p, Not(InSubquery(values, ListQuery(sub, conditions, _, _)))) =>
          // This is a NULL-aware (left) anti join (NAAJ) e.g. col NOT IN expr
          // Construct the condition. A NULL in one of the conditions is regarded
as a positive
          // result; such a row will be filtered out by the Anti-Join operator.

          // Note that will almost certainly be planned as a Broadcast Nested Loop
join.
          // Use EXISTS if performance matters to you.

          // Deduplicate conflicting attributes if any.
          val newSub = dedupSubqueryOnSelfJoin(p, sub, Some(values))
          val inConditions = values.zip(newSub.output).map(EqualTo.tupled)
          val (joinCond, outerPlan) = rewriteExistentialExpr(inConditions, p)
          // Not(InSubquery)和Not(Exists)就不一样了，还记得我们上文提到过InSubquery是
可以取null的,
          // 而Not null = null, Not false = true, 这样就和Exists产生区别了
          // 因此这里在构造Join Condition时我们要把null值也考虑进去，下面的例子很具体
          // Expand the NOT IN expression with the NULL-aware semantic
          // to its full form. That is from:
          //   (a1,a2,...) = (b1,b2,...)
          // to
          //   (a1=b1 OR isnull(a1=b1)) AND (a2=b2 OR isnull(a2=b2)) AND ...
          val baseJoinConds = splitConjunctivePredicates(joinCond.get)
          val nullAwareJoinConds = baseJoinConds.map(c => Or(c, IsNull(c)))
          // After that, add back the correlated join predicate(s) in the subquery
          // Example:
          // SELECT ... FROM A WHERE A.A1 NOT IN (SELECT B.B1 FROM B WHERE B.B2 =
A.A2 AND B.B3 > 1)
          // will have the final conditions in the LEFT ANTI as
          // (A.A1 = B.B1 OR ISNULL(A.A1 = B.B1)) AND (B.B2 = A.A2) AND B.B3 > 1
          val finalJoinCond = (nullAwareJoinConds ++ conditions).reduceLeft(And)
```

```
          Join(outerPlan, newSub, LeftAnti, Option(finalJoinCond), JoinHint.NONE)
        case (p, predicate) =>
          // 大家可以看上面的case都是对单独的Subquery做的，如果Subquery外面带着别的表达
式呢？
          // 例如 (Exists(..) Or A) And InSubquery(..)分割之后的Seq就是
          // [(Exists(..) Or A), InSubquery]，我们换成Tree的方式表达：
          // [Or(Exists(..), A), InSubquery] 这样的话这个Exists不会被上面的match到，
只有这个
          // InSubquery会被处理成Semi-Join，因此这个Exists就需要替换成一个
ExistentialJoin，
          // 然后对这个ExistentialJoin做一个Filter，过滤掉不满足的后，再做一次Project
把这个新加的列删除掉。
          val (newCond, inputPlan) = rewriteExistentialExpr(Seq(predicate), p)
          Project(p.output, Filter(newCond.get, inputPlan))
      }
  }
  /**
   * Given a predicate expression and an input plan, it rewrites any embedded
existential sub-query
   * into an existential join. It returns the rewritten expression together with
the updated plan.
   * Currently, it does not support NOT IN nested inside a NOT expression. This
case is blocked in
   * the Analyzer.
   */
  private def rewriteExistentialExpr(
      exprs: Seq[Expression],
      plan: LogicalPlan): (Option[Expression], LogicalPlan) = {
    var newPlan = plan
    // 这里两个作用，一个是把套在其他Expression里的Subquery重写成Existential，例如(A =
1 OR Exists(...))
    // 还有一个作用就是，重写嵌套的InSubquery，例如：
    // SELECT *
    // FROM   t1
    // WHERE  EXISTS (SELECT t2a
    //                FROM   t2
    //                WHERE  t1b IN (SELECT t3b FROM t3)
    // 这个查询里(t1b IN (...))会被提到上层的children里，因为t1b是一个
OuterReference，但是这个OuterReference
    // 不能写到JoinCondition里的，因为它是一个子查询，所以我们先拿一个ExistentialJoin
把这个OuterReference标记一下
    // 然后把ExistentialJoin的结果列放到Semi/Anti Join的Condition里。
    val newExprs = exprs.map { e =>
      e transformUp {
        case Exists(sub, conditions, _) =>
          val exists = AttributeReference("exists", BooleanType, nullable = false)
()
          newPlan =
            buildJoin(newPlan, sub, ExistenceJoin(exists),
conditions.reduceLeftOption(And))
          exists
        case InSubquery(values, ListQuery(sub, conditions, _, _)) =>
          val exists = AttributeReference("exists", BooleanType, nullable = false)
()
```

```
          // Deduplicate conflicting attributes if any.
          val newSub = dedupSubqueryOnSelfJoin(newPlan, sub, Some(values))
          val inConditions = values.zip(newSub.output).map(EqualTo.tupled)
          val newConditions = (inConditions ++ conditions).reduceLeftOption(And)
          newPlan = Join(newPlan, newSub, ExistenceJoin(exists), newConditions,
JoinHint.NONE)
          exists
        }
      }
      (newExprs.reduceOption(And), newPlan)
    }
}
```

至此，所有的Exists以及InSubquery都被重写为了Semi/Anti/Existential Join。

Optimize: RewriteCorrelatedScalarSubquery

这是最后一个Optimize Rule了，目标是将Correlated ScalarSubquery重写成LeftOuter Join，我们依旧自顶向下阅读。

```
/**
 * This rule rewrites correlated [[ScalarSubquery]] expressions into LEFT OUTER
joins.
 */
object RewriteCorrelatedScalarSubquery extends Rule[LogicalPlan] {
  ...
  /**
   * Rewrite [[Filter]], [[Project]] and [[Aggregate]] plans containing correlated
scalar
   * subqueries.
   */
  def apply(plan: LogicalPlan): LogicalPlan = plan transform {
    // 这里的主要任务分两点:
    // 1. 把ScalarSubquery替换为子查询结果的第一列的Reference
    // 2. 把ScalarSubquery重写为LeftOuter Join
    case a @ Aggregate(grouping, expressions, child) =>
      val subqueries = ArrayBuffer.empty[ScalarSubquery]
      // extractCorrelatedScalarSubqueries做两件事，一个是提出来ScalarSubquery，另一
个是把它替换成输出的列
      val newExpressions = expressions.map(extractCorrelatedScalarSubqueries(_,
subqueries))
      if (subqueries.nonEmpty) {
        // We currently only allow correlated subqueries in an aggregate if they
are part of the
        // grouping expressions. As a result we need to replace all the scalar
subqueries in the
        // grouping expressions by their result.
        // 这里说的是在父查询里如果Aggregate算子有ScalarSubquery，那么只允许出现在
GroupBy语句里，
        // 因此替换GroupBy里的ScalarSubquery
        val newGrouping = grouping.map { e =>
```

```
          //这里用了plan.output.head 这就是我们前文讲过的为什么在
PullupCorrelatedPredicate时只能Append新的列

subqueries.find(_.semanticEquals(e)).map(_.plan.output.head).getOrElse(e)
        }
        // constructLeftJoins是根据子查询构建LeftOuterJoin，返回新的查询树
        Aggregate(newGrouping, newExpressions, constructLeftJoins(child,
subqueries))
      } else {
        a
      }
      // 下面类似
    case p @ Project(expressions, child) =>
      val subqueries = ArrayBuffer.empty[ScalarSubquery]
      val newExpressions = expressions.map(extractCorrelatedScalarSubqueries(_,
subqueries))
      if (subqueries.nonEmpty) {
        Project(newExpressions, constructLeftJoins(child, subqueries))
      } else {
        p
      }
    case f @ Filter(condition, child) =>
      val subqueries = ArrayBuffer.empty[ScalarSubquery]
      val newCondition = extractCorrelatedScalarSubqueries(condition, subqueries)
      if (subqueries.nonEmpty) {
        Project(f.output, Filter(newCondition, constructLeftJoins(child,
subqueries)))
      } else {
        f
      }
  }
}
```

```
  /**
   * Extract all correlated scalar subqueries from an expression. The subqueries
are collected using
   * the given collector. The expression is rewritten and returned.
   */
  // 这个函数还是蛮好理解的，不多做解释
  private def extractCorrelatedScalarSubqueries[E <: Expression](
      expression: E,
      subqueries: ArrayBuffer[ScalarSubquery]): E = {
    val newExpression = expression transform {
      case s: ScalarSubquery if s.children.nonEmpty =>
        subqueries += s
        s.plan.output.head
    }
    newExpression.asInstanceOf[E]
  }
```

在看constructLeftJoins之前，我们先了解一下上文提到过的CountBugSPARK-15370，我们引用文章的例子：这个SQL在语义上是要统计用户的订单数，如果没有订单的话子查询应该返回0。

```
SELECT c_custkey, (
    SELECT COUNT(*)
    FROM ORDERS
    WHERE o_custkey = c_custkey
) AS count_orders
FROM CUSTOMER
```

但是我们根据一般Rewrite的方法，产生以下计划：

```
== Wrong Plan ==
Project [c_custkey, scalar_subquery() AS count_orders]
+- Join LeftOuter, (o_custkey = c_custkey) // 重写成 LeftOuter Join
   :- Project [c_custkey]
   :  +- Relation CUSTOMER
   +- Aggregate [o_custkey], [count(1)] // 子查询groupby o_custkey,求count
      +- Relation ORDERS
```

这个计划的语义显然是不对的，用户没有订单的时候LeftOuterJoin会在子查询的结果里填上null，显然这个时候我们需要的是0。（熟悉的同学会发现这个COUNT BUG和论文里讨论的不太一样，是因为别的系统会先LeftOuterJoin再Aggregate，而我们先Aggregate再LeftOuterJoin，所以对COUNT BUG的定义不一样，但他们的本质都是一样的），解决这个问题其实不难，我们只要在子查询中添加一列恒为真的属性（alwaysTrue），Join之后如果它为True说明Join成功了，如果返回了null说明没有Join上，那么子查询的返回值应该是Count(empty relation)的值。

```
  /**
   * Construct a new child plan by left joining the given subqueries to a base
plan.
   */
  private def constructLeftJoins(
      child: LogicalPlan,
      subqueries: ArrayBuffer[ScalarSubquery]): LogicalPlan = {
    subqueries.foldLeft(child) {
      case (currentChild, ScalarSubquery(query, conditions, _)) =>
        // 在Pullup之前的子查询的结果列
        val origOutput = query.output.head

        // 这个方法静态地用空的输入把子查询求一次值
        // 把Aggregate的结果用AggregateExpression的defaultResult来代替，剩下的列用
null来代替
        // 然后依次向上传递求值，最后得到一个Option
        val resultWithZeroTups = evalSubqueryOnZeroTups(query)
        if (resultWithZeroTups.isEmpty) {
          // 如果子查询为空的时候，得到的结果为null的话，那就不存在COUNT BUG了
          // 常见的聚集函数MAX,MIN等都满足这个条件
```

```
          // 因此我们直接重写成LeftOuterJoin，之后再加Project（因为只需要保留第一列）
          // CASE 1: Subquery guaranteed not to have the COUNT bug
          Project(
            currentChild.output :+ origOutput,
            Join(currentChild, query, LeftOuter, conditions.reduceOption(And),
JoinHint.NONE))
        } else {
          // Subquery might have the COUNT bug. Add appropriate corrections.
          // 这个方法把子查询根据 plans-having?-aggreate的方式分割
          // 第一个返回值是having-aggreate之上的算子
          // 第二个返回值是Option[Filter]代表是否包含having
          // 第三个返回值是Aggregate（再次强调，这个算子是一定包含的）
          val (topPart, havingNode, aggNode) = splitSubquery(query)

          // The next two cases add a leading column to the outer join input to
make it
          // possible to distinguish between the case when no tuples join and the
case
          // when the tuple that joins contains null values.
          // The leading column always has the value TRUE.
          // 这里就是我们讲过的，要添加一个alwaysTrue列，通过这一列来解决COUNT BUG
          val alwaysTrueExprId = NamedExpression.newExprId
          val alwaysTrueExpr = Alias(Literal.TrueLiteral,
            ALWAYS_TRUE_COLNAME)(exprId = alwaysTrueExprId)
          val alwaysTrueRef = AttributeReference(ALWAYS_TRUE_COLNAME,
            BooleanType)(exprId = alwaysTrueExprId)

          val aggValRef = query.output.head

          if (havingNode.isEmpty) {
            // CASE 2: Subquery with no HAVING clause
            // 没有Having的话，Join之后增加的Project的新列应该是:
            // if (alwaysTrue is null) then defaultValueOfAgg else subqueryResult
            Project(
              currentChild.output :+
                Alias(
                  If(IsNull(alwaysTrueRef),
                    Literal.create(resultWithZeroTups.get, origOutput.dataType),
                    aggValRef), origOutput.name)(exprId = origOutput.exprId),
              Join(currentChild,
                Project(query.output :+ alwaysTrueExpr, query),
                LeftOuter, conditions.reduceOption(And), JoinHint.NONE))

          } else {
            // CASE 3: Subquery with HAVING clause. Pull the HAVING clause above
the join.
            // Need to modify any operators below the join to pass through all
columns
            // referenced in the HAVING clause.

            // 有having的时候需要把having单独提到父查询中处理
            // 因为having会把aggregate之后的结果再做一次过滤，如果是因为这个原因导致的
            // 子查询结果为空，那么我们应该保留null而不是使用aggDefaultValue，保证语义
正确
```

```
            var subqueryRoot: UnaryNode = aggNode
            val havingInputs: Seq[NamedExpression] = aggNode.output

            // topPart只能包含Project和SubqueryAlias这两个算子，我们把having提上来
            // 因此需要把Project的projectList加上having算子依赖的属性
            topPart.reverse.foreach {
              case Project(projList, _) =>
                subqueryRoot = Project(projList ++ havingInputs, subqueryRoot)
              case s @ SubqueryAlias(alias, _) =>
                subqueryRoot = SubqueryAlias(alias, subqueryRoot)
              case op => sys.error(s"Unexpected operator $op in corelated
subquery")
            }

            // 下面这个就很好理解了，我们把having conditon放到case when里过滤
            // CASE WHEN alwayTrue IS NULL THEN resultOnZeroTups
            //      WHEN NOT (original HAVING clause expr) THEN CAST(null AS <type
of aggVal>)
            //      ELSE (aggregate value) END AS (original column name)
            val caseExpr = Alias(CaseWhen(Seq(
              (IsNull(alwaysTrueRef), Literal.create(resultWithZeroTups.get,
origOutput.dataType)),
              (Not(havingNode.get.condition), Literal.create(null,
aggValRef.dataType))),
              aggValRef),
              origOutput.name)(exprId = origOutput.exprId)

            Project(
              currentChild.output :+ caseExpr,
              Join(currentChild,
                Project(subqueryRoot.output :+ alwaysTrueExpr, subqueryRoot),
                LeftOuter, conditions.reduceOption(And), JoinHint.NONE))

          }
        }
      }
    }
```

至此，Correlated Subquery的转换规则全部结束了。

Execute: Non-Correlated ScalarSubquery

非关联的ScalarSubquery（non-correlated ScalarSubquery)我们使用一个Physical Plan来处理，然后在代码生成阶段把预先执行完的子查询的值填进去，这里用了一个Rule来把ScalarSubquery用一个物理算子替换，然后预执行。

```
/**
 * Plans scalar subqueries from that are present in the given [[SparkPlan]].
 */
```

```
case class PlanSubqueries(sparkSession: SparkSession) extends Rule[SparkPlan] {
  def apply(plan: SparkPlan): SparkPlan = {
    plan.transformAllExpressions {
      case subquery: expressions.ScalarSubquery =>
        // 获得subquery的executedPlan，然后组装成一个ScalarSubquery Physical Plan
        val executedPlan = new QueryExecution(sparkSession,
subquery.plan).executedPlan
        ScalarSubquery(
          SubqueryExec(s"subquery${subquery.exprId.id}", executedPlan),
          subquery.exprId)
    }
  }
}
```

这一部分的代码在master（3.0）分支有部分改动，但是整体过程没有变化。

```
/**
 * A subquery that will return only one row and one column.
 *
 * This is the physical copy of ScalarSubquery to be used inside SparkPlan.
 */
case class ScalarSubquery(
    plan: SubqueryExec,
    exprId: ExprId)
  extends ExecSubqueryExpression {

  override def dataType: DataType = plan.schema.fields.head.dataType
  override def children: Seq[Expression] = Nil
  override def nullable: Boolean = true
  override def toString: String = plan.simpleString(SQLConf.get.maxToStringFields)
  override def withNewPlan(query: SubqueryExec): ScalarSubquery = copy(plan =
query)

  override def semanticEquals(other: Expression): Boolean = other match {
    case s: ScalarSubquery => plan.sameResult(s.plan)
    case _ => false
  }

  // the first column in first row from `query`.
  @volatile private var result: Any = _
  @volatile private var updated: Boolean = false

  // 这个方法在SparkPlan的waitForSubqueries里调用，等待子查询结果
  def updateResult(): Unit = {
    // collect子查询结果，做runtime check
    val rows = plan.executeCollect()
    if (rows.length > 1) {
      sys.error(s"more than one row returned by a subquery used as an
expression:\n$plan")
    }
    if (rows.length == 1) {
      assert(rows(0).numFields == 1,
```

```
        s"Expects 1 field, but got ${rows(0).numFields}; something went wrong in
analysis")
      result = rows(0).get(0, dataType)
    } else {
      // If there is no rows returned, the result should be null.
      result = null
    }
    updated = true
  }

  override def eval(input: InternalRow): Any = {
    require(updated, s"$this has not finished")
    result
  }

  override def doGenCode(ctx: CodegenContext, ev: ExprCode): ExprCode = {
    require(updated, s"$this has not finished")
    Literal.create(result, dataType).doGenCode(ctx, ev)
  }
}
```

总结

本文从Spark2.4.x版本的源码入手，分析了SparkSQL当前处理子查询的方法，认真读完这篇文章的同学应该会感慨，SparkSQL对子查询的支持竟然这么简单，功能也不是很完善（但在实际生产中应该能满足大部分需求了），同样存在了很多Bug。

笔者研究SparkSQL源码以及接近半年了，本文是对这半年里主要的研究内容做一个自我总结，在实验的过程中也发现了Bug并且合并了相应的PullRequest。笔者近期也在着手SparkSQL子查询功能的完善（ANY，ALL谓词），Bug的修复（InSubquery的null值），性能优化（Non-Correlated Subquery）等等，有兴趣的同学可以联系我一起讨论学习~

参考资料

1. GitHub: Spark
2. Subqueries in Apache Spark 2.0 - Davies Liu & Herman van Hövel
3. SQL 子查询的优化
4. Orthogonal Optimization of Subqueries and Aggregation - C Galindo-Legaria, M Joshi
5. Unnesting Arbitrary Queries - T Neumann, A Kemper
6. Complex Query Decorrelation - P. Seshadri; H. Pirahesh; T. Y. C. Leung
7. MySQL EXISTS Strategy