理解spark sql 优化策略的最好方法就是自己实现一个

Original spark君 一生数据人 2019年01月20日 19:00

spark sql 的优化框架 Catalyst 博大精深，里面的精华是很多大牛一个pr一个pr积累起来的，仔细琢磨琢磨相关源码也是一件痛并快乐的事情，spark 逻辑优化就是在一个 AST 树上进行匹配，匹配到一定的规则，然后进行等价变换规则，从而使计算的成本更低，今天我带大家自己实现一个逻辑优化规则，帮助大家更快地理解spark sql 逻辑优化的底层原理，如果对 spark sql 总体架构不了解的，可以先看这篇文章 是时候学习真正的spark技术了 了解全貌。

我们的例子非常简单，先注册一个表，包含一个 a 字段：

```
scala>  Seq((10)).toDF(("a")).createOrReplaceGlobalTempView("t")

scala> spark.sql("select * from t").show
+---+
|  a|
+---+
| 10|
+---+
```

我们看下当前的执行计划：

```
scala> val queryExecution = spark.sql("select a * 1 from t").queryExecution
queryExecution: org.apache.spark.sql.execution.QueryExecution =
== Parsed Logical Plan ==
'Project [unresolvedalias(('a * 1), None)]
+- 'UnresolvedRelation `t`

== Analyzed Logical Plan ==
(a * 1): int
Project [(a#27 * 1) AS (a * 1)#91]
+- SubqueryAlias `t`
   +- Project [value#25 AS a#27]
      +- LocalRelation [value#25]

== Optimized Logical Plan ==
LocalRelation [(a * 1)#91]

== Physical Plan ==
LocalTableScan [(a * 1)#91]
```

可以看到这个执行计划是比较费的， 因为对于 (a * 1) 这个算式来讲，其实就等于a本身，我们针对这种规则自定义一个 逻辑优化规则

```
scala> import  org.apache.spark.sql.catalyst.rules._
import org.apache.spark.sql.catalyst.rules._

scala> import org.apache.spark.sql.catalyst.plans.logical
import org.apache.spark.sql.catalyst.plans.logical

scala>

scala> object MyMultiplyOpetimizationRule extends Rule[LogicalPlan] {
     |   def apply(plan: LogicalPlan): LogicalPlan = plan transform {
     |     case q: LogicalPlan => q transformExpressionsDown {
     |       case mul @ Multiply(left, right) if right.isInstanceOf[Literal] &&
     |         right.asInstanceOf[Literal].value.asInstanceOf[Integer] == 1 =>
     |         left
     |     }
     |   }
     | }
defined object MyMultiplyOpetimizationRule
```

```
scala> queryExecution.analyzed
res39: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
Project [(a#27 * 1) AS (a * 1)#91]
+- SubqueryAlias `t`
   +- Project [value#25 AS a#27]
      +- LocalRelation [value#25]

scala>

scala>

scala>  MyMultiplyOpetimizationRule(queryExecution.analyzed)
res40: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
Project [a#27 AS (a * 1)#91]
+- SubqueryAlias `t`
   +- Project [value#25 AS a#27]
      +- LocalRelation [value#25]
```

上面的代码很好理解，如果匹配到一个变量乘以1的表达式，就直接变换为变量本身, 应用完这个规则 (a#27 * 1) 就变为了 a#27：

这样就少了一次乘法运算，从而提高了性能。

上面我们是从内部测试，如果你在应用中要定义一个基于规则的优化，然后让这个优化策略自动应用到你写的sql中，可以如下方式定义

```
scala>  import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.SparkSession

scala> import org.apache.spark.sql.SparkSessionExtensions
import org.apache.spark.sql.SparkSessionExtensions

scala> case class  MyMultiplyOpetimizationRule(spark: SparkSession) extends Rule[LogicalPlan] {
     |   def apply(plan: LogicalPlan): LogicalPlan = plan transform {
     |     case q: LogicalPlan => q transformExpressionsDown {
     |       case mul @ Multiply(left, right) if right.isInstanceOf[Literal] &&
     |         right.asInstanceOf[Literal].value.asInstanceOf[Double] == 1 =>
     |         left
     |     }
     |   }
     | }
defined class MyMultiplyOpetimizationRule

scala> val f: ExtensionsBuilder = { e => e.injectOptimizerRule(MyMultiplyOpetimizationRule)}
f: ExtensionsBuilder = <function1>

scala> val spark = SparkSession.builder().withExtensions(f).getOrCreate()
spark: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@36b81ff2
```

sparkSession 中给用户留了扩展点，Spark catalyst的扩展点在SPARK-18127中被引入，Spark用户可以在SQL处理的各个阶段扩展自定义实现，非常强大高效

- injectOptimizerRule – 添加optimizer自定义规则，optimizer负责逻辑执行计划的优化,我们例子中就是扩展了逻辑优化规则。
- injectParser – 添加parser自定义规则，parser负责SQL解析。
- injectPlannerStrategy – 添加planner strategy自定义规则，planner负责物理执行计划的生成。
- injectResolutionRule – 添加Analyzer自定义规则到Resolution阶段，analyzer负责逻辑执行计划生成。
- injectPostHocResolutionRule – 添加Analyzer自定义规则到Post Resolution阶段。
- injectCheckRule – 添加Analyzer自定义Check规则。

其他几种扩展我们可以也会举例说明，今天只讲解一下怎么扩展逻辑优化规则。