Spark 2.0介绍：在Spark SQL中定义查询优化规则

《Spark 2.0技术预览：更容易、更快速、更智能》文章中简单地介绍了Spark 2.0带来的新技术等。Spark 2.0是Apache Spark的下一个主要版本。此版本在架构抽象、API以及平台的类库方面带来了很大的变化，为该框架明年的发展奠定了方向，所以了解Spark 2.0的一些特性对我们能够使用它有着非常重要的作用。本博客将对Spark 2.0进行一序列的介绍（参见Spark 2.0分类），欢迎关注。

Catalyst优化器

Spark SQL使用Catalyst优化所有的查询，包括spark sql和dataframe dsl。这个优化器的使用使得查询比直接使用RDD要快很多。Spark在每个版本都会对Catalyst进行优化以便提高查询性能，而不需要用户修改他们的代码。

Catalyst是一个单独的模块类库，这个模块是基于规则的系统。这个框架中的每个规则都是针对某个特定的情况来优化的。比如：ConstantFolding规则用于移除查询中的常量表达式。

在Spark的早期版本，如果需要添加自定义的优化规则，我们需要修改Spark的源码，这在很多情况下是不太可取的，比如我们仅仅需要优化特定的领域或者场景。所以开发社区想有一种可插拔的方式在Catalyst中添加优化规则。

值得高兴的是，Spark 2.0提供了这种实验式的API，我们可以基于这些API添加自定义的优化规则。本文将介绍如何编写自定义的优化规则，并将这些规则添加到Catalyst中。

dataframe的优化计划(Optimized plan)

在编写我们自定义的优化规则之前，首先我们来理解如何在Spark中访问优化计划，下面代码片段就是展示访问优化计划的：

```
/**
 * User: 过往记忆
 * Date: 2016年07月14日
 * Time: 下午22:49
 * bolg: https://www.iteblog.com
 * 本文地址：https://www.iteblog.com/archives/1706
 * 过往记忆博客，专注于hadoop、hive、spark、shark、flume的技术博客，大量的干货
 * 过往记忆博客微信公共帐号：iteblog_hadoop
 */

scala> val df = spark.read.option("header","true").csv("file:///user/iteblog/sales.csv")
df: org.apache.spark.sql.DataFrame = [transactionId: string, customerId: string ... 2 more fields
]

scala> val multipliedDF = df.selectExpr("amountPaid * 1")
multipliedDF: org.apache.spark.sql.DataFrame = [(amountPaid * 1): double]

scala> println(multipliedDF.queryExecution.optimizedPlan.numberedTreeString)
00 Project [(cast(amountPaid#89 as double) * 1.0) AS (amountPaid * 1)#91]
01 +- Relation[transactionId#86,customerId#87,itemId#88,amountPaid#89] csv
```

上面代码中我们加载了一个csv文件，并对每一行的amountPaid自动乘以1。我们可以使用queryExecution方法上的optimizedPlan对象来查看这个DataFrame的优化计划。queryExecution允许我们访问运行这个查询的所有信息。优化计划就是其中一个。

Spark中的所有计划都是使用tree代表的。所以numberedTreeString方法以树形的方式打印出优化计划。正如上面的输出一样。

所有的计划都是从下往上读的。下面是树中的两个节点：
1、01 Relation：表示从csv文件创建的DataFrame；
2、00 Project：表示投影（也就是需要查询的列）。

从上面的输出可以看到，为了得到正确的结果，Spark通过cast将amountPaid转换成double类型。

自定义优化计划

从上面的计划可以看出，Spark自动对每一行的amountPaid乘上1.0。但是这不是最优计划！因为如果是乘以1，最终的结果是一样的。所有我们可以利用这个知识来编写自定义的优化规则，并将这个规则加入到Catalyst中。

下面代码片段展示了如何自定义优化规则：

```
/**
 * User: 过往记忆
 * Date: 2016年07月14日
 * Time: 下午22:49
 * bolg: https://www.iteblog.com
 * 本文地址：https://www.iteblog.com/archives/1706
 * 过往记忆博客，专注于hadoop、hive、spark、shark、flume的技术博客，大量的干货
 * 过往记忆博客微信公共帐号：iteblog_hadoop
 */

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

scala> import org.apache.spark.sql.catalyst.expressions.{Literal, Multiply}
import org.apache.spark.sql.catalyst.expressions.{Literal, Multiply}

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

scala> import org.apache.spark.sql.catalyst.rules.Rule
import org.apache.spark.sql.catalyst.rules.Rule

scala> object MultiplyOptimizationRule extends Rule[LogicalPlan] {
     |   def apply(plan: LogicalPlan): LogicalPlan = plan transformAllExpressions {
     |     case Multiply(left,right) if right.isInstanceOf[Literal] &&
     |       right.asInstanceOf[Literal].value.asInstanceOf[Double] == 1.0 =>
     |       println("optimization of one applied")
     |       left
     |   }
     | }
defined object MultiplyOptimizationRule
```

这里我们扩展了Rule，Rule是直接操作逻辑计划的。绝大多数的规则都是使用Scala中的模式匹配。在上面的代码中，我们首先判断优化的操作数(operand)是否是文字(literal)，然后判断其值是否是1.0。为了简便起见，我们限定了1出现的位置，如果1出现在左边，这个优化规则将不起作用。但是我们可以仿照上面的示例轻松地实现。

通过上面的规则，如果右边的值是1，我们将直接返回左边的值。

将自定义的优化规则加入到Catalyst中

上面我们已经定义好自定义的规则，接下来我们需要将这个规则添加到Catalyst中，如下代码片段：

```
scala> spark.experimental.extraOptimizations = Seq(MultiplyOptimizationRule)
spark.experimental.extraOptimizations: Seq[org.apache.spark.sql.catalyst.rules.
Rule[org.apache.spark.sql.catalyst.plans.logical.LogicalPlan]] =
List(MultiplyOptimizationRule$@3aaaf13b)
```

SparkSession中提供了experimental对象，其包含了所有的实验室API。我们可以使用extraOptimizations来添加一系列的自定义规则到Catalyst中。

使用自定义优化规则

添加好自定义规则之后，我们需要验证这个规则是否启用。如下代码所示：

```
/**
 * User: 过往记忆
 * Date: 2016年07月14日
 * Time: 下午22:49
 * bolg: https://www.iteblog.com
 * 本文地址：https://www.iteblog.com/archives/1706
 * 过往记忆博客，专注于hadoop、hive、spark、shark、flume的技术博客，大量的干货
 * 过往记忆博客微信公共帐号：iteblog_hadoop
```

```
 */

scala> val multipliedDFWithOptimization = df.selectExpr("amountPaid * 1")
multipliedDFWithOptimization: org.apache.spark.sql.DataFrame = [(amountPaid * 1): double]

scala> println("after optimization")
after optimization

scala> println(multipliedDFWithOptimization.queryExecution.
     | optimizedPlan.numberedTreeString)
optimization of one applied
00 Project [cast(amountPaid#89 as double) AS (amountPaid * 1)#93]
01 +- Relation[transactionId#86,customerId#87,itemId#88,amountPaid#89] csv
```

从上面的输出结果可以看出，amountPaid上的乘法已经没了，这证明了我们的优化规则已经起作用了。有了这个强大的可插拔优化规则，将会为开发者提供极大的便利。