四川大学网络空间安全研究院
CYBERSECURITY RESEARCH INSTITUTE, SICHUAN UNIVERSITY

Spark SQL 源码分析

版　本　号：v1.0

项　目　组：大数据平台项目组

文 档 作 者：吴天雄

撰 写 日 期：**2017** 年 **10** 月 **18** 号

评审负责人：

评 审 日 期：

四 川 大 学 网 络 空 间 安 全 研 究 院

目 录

第 1 章　Spark SQL 源码分析--运行框架

1.1　传统关系型数据库运行过程

在介绍 Spark SQL 之前, 我们首先来看看, 传统的关系型数据库是怎么运行的。当我们提交了一个很简单的查询:

```
select a1,b1,c1 from tableA where condition
```

传统关系型数据库执行上述 SQL 时，会按照如图1.1所示的过程执行执行

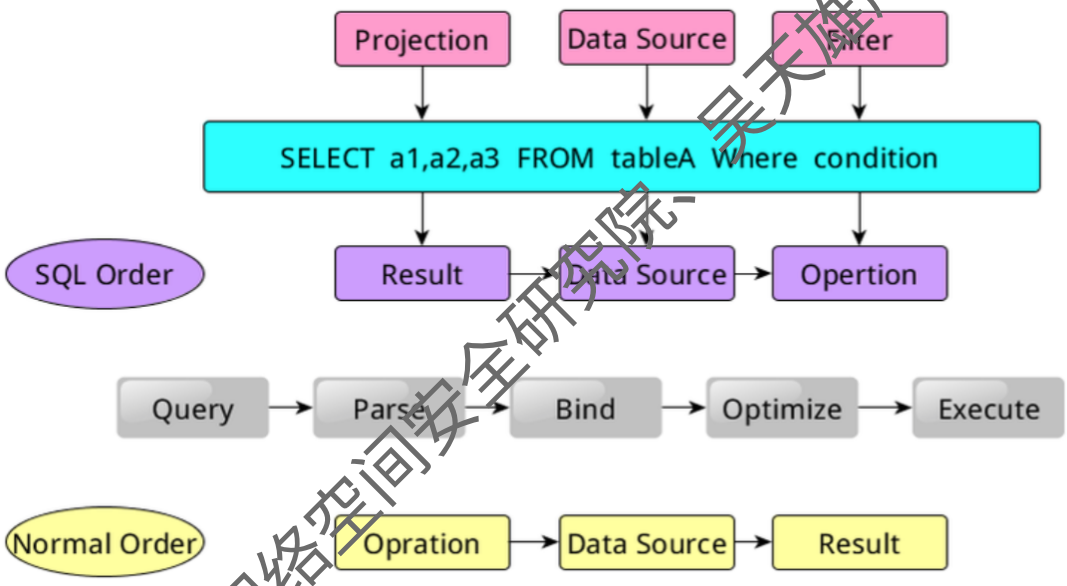


图 1.1　传统关系型数据库 SQL 执行过程

可以看得出来, 该语句是由 Projection(a1,a2,a3)、Data Source(tableA)、Filter(condition) 组成, 分别对应 SQL 查询过程中的 Result、Data Source、Operation, 也就是说 SQL 语句按 Result–>Data Source–>Operation 的次序来描述的。那么,SQL 语句在实际的运行过程中是怎么处理的呢? 一般的数据库系统先将读入的 SQL 语句 (Query) 先进行解析 (Parse), 分辨出 SQL 语句中哪些词是关键词 (如 SELECT、FROM、WHERE), 哪些是表达式、哪些是 Projection、哪些是 Data Source 等等。这一步就可以判断 SQL 语句是否规范, 不规范就报错, 规范就继续下一步过程绑定 (Bind), 这个过程将 SQL 语句和数据库的数据字典 (列、表、视图等等) 进行绑定, 如果相关的 Projection、Data Source 等等都是存在的话, 就表示这个 SQL 语句是可

以执行的; 而在执行前, 一般的数据库会提供几个执行计划, 这些计划一般都有运行统计数据, 数据库会在这些计划中选择一个最优计划 (Optimize), 最终执行该计划 (Execute), 并返回结果。当然在实际的执行过程中, 是按 Operation–>Data Source–>Result 的次序来进行的, 和 SQL 语句的次序刚好相反; 在执行过程有时候甚至不需要读取物理表就可以返回结果, 比如重新运行刚运行过的 SQL 语句, 可能直接从数据库的缓冲池中获取返回结果。

以上过程看上去非常简单, 但实际上会包含很多复杂的操作细节在里面。而这些操作细节都和 Tree 有关, 在数据库解析 (Parse)SQL 语句的时候, 会将 SQL 语句转换成一个树型结构来进行处理, 如下图1.2一个查询, 会形成一个含有多个节点 (TreeNode) 的 Tree, 然后在后续的处理过程中对该 Tree 进行一系列的操作。

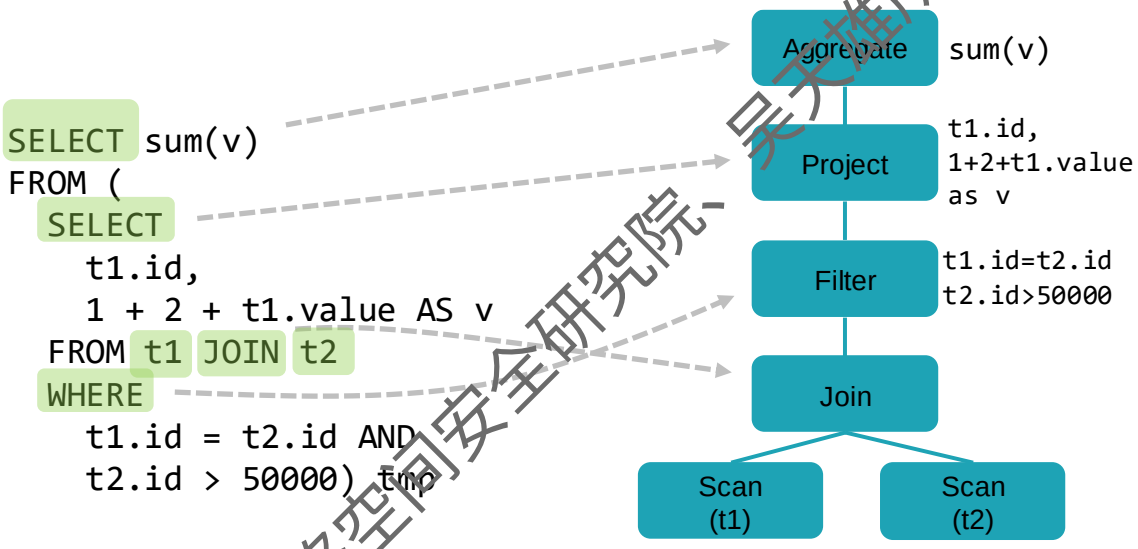


图 1.2　数据库查询与 Tree 对应关系

1.2　Spark SQL 处理数据过程

上一小节介绍了关系型数据库的运行过程, 那么,Spark SQL 是不是也采用类似的方式处理呢? 答案是肯定的。下面我们先来看看 Spark SQL 中的两个重要概念 Tree 和 Rule、然后在看看 sparkSQL 的优化器 Catalyst。

1.2.1　Tree 和 Rule

Spark SQL 对 SQL 语句的处理和关系型数据库对 SQL 语句的处理采用了类似的方法, 首先会将 SQL 语句进行解析 (Parse), 然后形成一个 Tree, 在后续的如绑定、优化等处理过程都是对 Tree 的操作, 而操作的方法是采用 Rule, 通过模式匹配, 对不同类型的节点采用不同的操作。

1.2.1.1　Tree

Spark SQL 中 Tree 是由 TreeNode 实现的，TreeNode 可以通过 scala 集合类操作方法中 foreach、map、flatMap、collect 等进行操作，TreeNode 中还实现了应用 Rule 的方法，比如 TransformDown、transformUp 将 Rule 应用到给定的树段，并用结果替代旧的树段。有了 TreeNode, 通过 Tree 中各个 TreeNode 之间的关系, 可以对 Tree 进行遍历操作，Spark SQL 定义了 catalyst.trees 的日志, 通过这个日志可以形象的表示出树的结构。TreeNode 可以分为以下 3 种：

1. UnaryNode

   一元节点, 即只有一个子节点。如 Limit、Filter 操作;

2. BinaryNode

   二元节点, 即有左右子节点的二叉节点。如 Jion、Union 操作

3. LeafNode

   叶子节点, 没有子节点的节点。主要用户命令类操作, 如 SetCommand.

1.2.1.2　Rule

SQL 语句经过 Parser 阶段的处理，转化为 UnresolvedLogicalPlan 的一棵树，Analyzer 和 Optimizer 将会对 UnresolvedLogicalPlan 的这棵树施加各种分析和优化操作，这些分析和操作实际就是一系列的 Rule。Rule 是一个抽象类, 具体的 Rule 实现是通过 RuleExecutor 完成。Rule 是一个抽象类，其实现如代码1.1所示

</>　程序 **1.1:** *Rule* 的抽象定义　</>

```
abstract class Rule[TreeType <: TreeNode[_]] extends Logging {
    val ruleName: String = {
        val className = getClass.getName
        if (className endsWith "\$") className.dropRight(1) else
        ↪  className
    }
    def apply(plan: TreeType): TreeType
}
```

Rule 通过定义 batch 和 batchs, 可以简便的、模块化地对 Tree 进行 transform 操作，Rule 通过定义 Once 和 FixedPoint, 可以对 Tree 进行一次操作或多次操作 (如对某些 Tree 进行多次迭代操作的时候, 达到 FixedPoint 次数迭代或达到前后两次的树结构没变化才停止操作, 具体参看 RuleExecutor.execute),RuleExecutor 简要代码如

程序1.2所示

程序 **1.2:** *RuleExecutor* 的抽象定义

```
abstract class RuleExecutor[TreeType <: TreeNode[_]] extends Logging {
    abstract class Strategy { def maxIterations: Int }

    case object Once extends Strategy { val maxIterations = 1 }

    case class FixedPoint(maxIterations: Int) extends Strategy

    protected case class Batch(name: String, strategy: Strategy,
      ↪  rules: Rule[TreeType]*)

    protected val batches: Seq[Batch]

    def execute(plan: TreeType): TreeType = {
        ......
    }
}
```

RuleExecutor 用于执行 Rule，RuleExecutor 中的一些概念如下所示

1. Strategy

   执行策略，即执行 Rule 的最大次数；

2. Once

   只执行一次 Rule 的策略；

3. FixedPoint

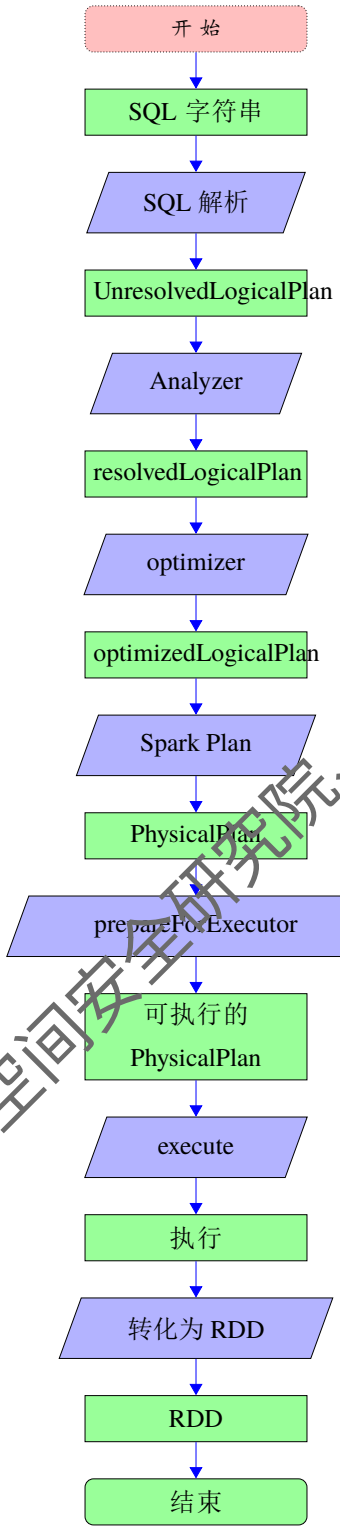
   达到 FixedPointed 指定次数或前后两次的树结构没变化时停止操作的策略；

4. Batch

   一批 Rule 及其相应的执行策略。

1.3　Spark SQL 语句处理流程

Spark Sql 处理 sql 语句的大致流程为：



1. SQL 语句经过 SqlParser 解析成 Unresolved LogicalPlan;
2. 使用 analyzer 结合数据数据字典 (catalog) 进行绑定, 生成 resolved LogicalPlan;
3. 使用 optimizer 对 resolved LogicalPlan 进行优化, 生成 optimized LogicalPlan;
4. 使用 SparkPlan 将 LogicalPlan 转换成 PhysicalPlan;

5. 使用 prepareForExecution() 将 PhysicalPlan 转换成可执行物理计划;
6. 使用 execute() 执行可执行物理计划;
7. 生成 RDD。

第 2 章　Spark SQL 源码分析–SQL 解析

2.1　Spark SQL 解析 SQL 语句的流程

这里我们假设 SQL 语句为

```
val query = sql("SELECT * FROM src")
```

SQLContext 的 SQL 方法是解析所有 SQL 的总入口，但实际上会交给 parseSql 方法，如下代码段所示

```
def sql(sqlText: String): DataFrame = {
    DataFrame(this, parseSql(sqlText))
}
```

parseSql 调用 DDLParser 解析 SQL，代码如下所示

```
protected[sql] def parseSql(sql: String): LogicalPlan =
  ↪  ddlParser.parse(sql, false)
```

上述代码段调用 DDLParser 中的 parse 方法，这个方法会调用抽象类类 AbstractSparkSQLParser 中的 parse 方法，此方法中首先会初始化词法，初始化关键字，根据获取的关键字，对 sql 语句解析，最终获得 logical plan。如代码段2.1所示

</>　　**程序 2.1:** *AbstractSparkSQLParser 解析器*　　</>

```
def parse(input: String): LogicalPlan = synchronized {
    //初始化词法 初始化关键字
    initLexical
    //根据获取的关键字，对 sql 语句解析，最终获得 logical plan
    phrase(start)(new lexical.Scanner(input)) match {
        case Success(plan, _) => plan
        case failureOrError => sys.error(failureOrError.toString)
    }
}
//上面初始化是调用这里，而这里的参数是获取 reservedWords 的函数
```

```
protected lazy val initLexical: Unit =
  ↪  lexical.initialize(reservedWords)
```

下面介绍 reservedWords 中的内容，reservedWords 的代码如下所示，它定义在 AbstractSparkSQLParser 类中

```
protected lazy val reservedWords: Seq[String] =
this
.getClass
.getMethods
.filter(_.getReturnType == classOf[Keyword])
.map(_.invoke(this).asInstanceOf[Keyword].normalize)
```

实际上 Sql 解析最后调用的是 DDLParser，而 DDLParser 又是继承了 AbstractSparkSQLParser 类，所以 reservedWords 里获取的关键字是 DDLParser 中的关键字变量。

AbstractSparkSQLParser 中的 parse 方法，首先会初始化关键字，接着执行 phrase(start)(new lexical.Scanner(input))，而 start 在 AbstractSparkSQLParser 的子类 DDLParser 中重写为如下程序段所示，AbstractSparkSQLParser 实现了三个 parser：createTable 、describeTable 和 refreshTable，并将其重载为 AbstractSparkSQLParser.start 变量

```
protected def start: Parser[LogicalPlan] = ddl
protected lazy val ddl: Parser[LogicalPlan] = createTable |
  ↪  describeTable | refreshTable
```

这里调用的 phrase 方法实际上来自于 AbstractSparkSQLParser 的父类 PackratParsers。PackratParsers 和 StandardTokenParsers 实际上都是 Scala 自带的类。它们的功能较为复杂，而且 SparkSQL 本身的作用原理关系并不是很大。由于我们的 sql 语句为 SELECT * FROM src，在通过 DDLParser.parse 构建的 PackratParser，根本无法匹配到 SELECT，因为 DDLParser 中获取的关键字根本不包含 SELECT。代码段2.1执行时会抛出异常，即 sys.error(failureOrError.toString)，此时 DDLParser.parse 会按照下面方式处理

程序 2.2: *DDLparser* 解析器

```
def parse(input: String, exceptionOnError: Boolean): LogicalPlan = {
    try {
        //上面的异常是通过，下面这句代码产生的
        parse(input)
    } catch {
        case ddlException: DDLException => throw ddlException
        //根据的抛出的异常是 sys.error(failureOrError.toString)，它抛出的是 RuntimeException，所以不会匹配到上面的 ddlexception，
        //因此会执行下面这句话，另外由于 exceptionOnError 刚开始指定的是 false，所以直接执行 parseQuery
        case _ if !exceptionOnError => parseQuery(input)
        case x: Throwable => throw x
    }
}
```

综上，我们的 select 语句在 DDLParser 中最终通过 parseQuery 来进行解析。parseQuery 方法是我们在初始化 DDLParser 时传入的参数，初始化 DDLParser 在 SQLContext 中进行，代码如下所示

```
//这里就是 new 这个 SparkSQLParser 的地方
@transient
protected[sql] val ddlParser = new DDLParser(sqlParser.parse(_))
@transient
protected[sql] val sqlParser = new
  SparkSQLParser(getSQLDialect().parse(_))
```

所以上面那个 parseQuery 就是 SparkSQLParser 的 parse 方法，而 SparkSQLParser 的 parse 方法其实就是 AbstractSparkSQLParser 的 parse 方法，但此时 start 被重写为如下内容

```
override protected lazy val start: Parser[LogicalPlan] = cache |
  uncache | set | show | desc | others
```

虽然这个 SparkSQLParser 中还是没有 SELECT 的关键字。所以它仍然匹配不到，但是万幸的是它的 start 中有一个 others，如下程序段所示

```
private lazy val others: Parser[LogicalPlan] =
wholeInput ^^ {
    case input => fallback(input)
}
```

这里会调用 fallback，而 fallback 是在初始化 SparkSQLParser 传入的参数即 getSQLDialect().parse(_)，getSQLDialect() 通过反射的方法来构建的传入的那个对象，这里反射的是 dialectClassName 对象，dialectClassName 代码如下所示

```
protected[sql] def dialectClassName = if (conf.dialect == "sql") {
    //这里它会执行这个 if 语句，因为 conf.dialect 默认就是"sql"
    classOf[DefaultParserDialect].getCanonicalName
} else {
    conf.dialect
}
```

所以我们最终执行的是 DefaultParserDialect 的 parse 方法。接下来我们看看 DefaultParserDialect 对象的结构，如下所示

```
private[spark] class DefaultParserDialect extends ParserDialect {
    @transient
    protected val sqlParser = SqlParser
    override def parse(sqlText: String): LogicalPlan = {
        //最终 select 语句执行的代码
        sqlParser.parse(sqlText)
    }
}
```

也就是说其实最终的最终，fallback 方法指的是 SqlParser 中的 parse 方法，而这个方法其实还是我们前面提到的 AbstractSparkSQLParser 的这个 parse，但是如今这里它的子类变成了 SqlParser，而 SqlParser 中 start 如下所示

```
protected lazy val start: Parser[LogicalPlan] =start1 | insert | cte
```

而 start1 中包含了对关键字 select 的解析，如下所示

```
protected lazy val start1: Parser[LogicalPlan] =
(select | ("(" ~> select <~ ")")) *
( UNION ~ ALL        ^^^ { (q1: LogicalPlan, q2: LogicalPlan) =>
  ↪  Union(q1, q2) }
| INTERSECT          ^^^ { (q1: LogicalPlan, q2: LogicalPlan) =>
  ↪  Intersect(q1, q2) }
| EXCEPT             ^^^ { (q1: LogicalPlan, q2: LogicalPlan) =>
  ↪  Except(q1, q2)}
| UNION ~ DISTINCT.? ^^^ { (q1: LogicalPlan, q2: LogicalPlan) =>
  ↪  Distinct(Union(q1, q2)) }
)
```

2.2　UnresolvedLogicalPlan 的生成过程

我们的 sql 语句为 SELECT * FROM src，也就是这个 start1 会首先去匹配 SELECT 关键字。select 匹配的程序如2.3所示，这里可以看到复合表达式 ~[①]、~>[②]、<~[③]、|[④]、^^[⑤]及^?[⑥]。

程序 2.3: 匹配 *select* 语句

```
protected lazy val select: Parser[LogicalPlan] =
//到这里他才匹配到这个 SELECT 关键字
SELECT ~> DISTINCT.? ~
repsep(projection, ",") ~
(FROM   ~> relations).? ~
(WHERE  ~> expression).? ~
(GROUP  ~  BY ~> rep1sep(expression, ",")).? ~
(HAVING ~> expression).? ~
sortType.? ~
(LIMIT  ~> expression).? ^^ {
```

① p 成功, 才会 q; 放回 p,q 的结果
② p 成功, 才会 q, 返回 q 的结果
③ p 成功, 才会 q, 返回 p 的结果
④ p 失败则 q, 返回第一个成功的结果
⑤ 如果 p 成功, 将函数 f 应用到 p 的结果上
⑥ 如果 p 成功, 如果函数 f 可以应用到 p 的结果上的话, 就将 p 的结果用 f 进行转换

```
11      case d ~ p ~ r ~ f ~ g ~ h ~ o ~ l =>
12      val base = r.getOrElse(OneRowRelation)
13      val withFilter = f.map(Filter(_, base)).getOrElse(base)
14      val withProjection = g
15      .map(Aggregate(_, p.map(UnresolvedAlias(_)), withFilter))
16      .getOrElse(Project(p.map(UnresolvedAlias(_)), withFilter))
17      val withDistinct = d.map(_ =>
          ↪  Distinct(withProjection)).getOrElse(withProjection)
18      val withHaving = h.map(Filter(_,
          ↪  withDistinct)).getOrElse(withDistinct)
19      val withOrder = o.map(_(withHaving)).getOrElse(withHaving)
20      val withLimit = l.map(Limit(_, withOrder)).getOrElse(withOrder)
21      withLimit
22  }
```

上述程序段第四行匹配到我们的 projection，我们的 sql 语句中是一个“*”

```
1 protected lazy val projection: Parser[Expression] =
2 //先是匹配到这里
3 expression ~ (AS.? ~> ident.?) ^^ {
4     case e ~ a => a.fold(e)(Alias(e, _)())
5 }
```

接下来我只把变量依次写出来，直到匹配到最终的结果：expression->orExpression->andExpression->notExpression->comparisonExpression->termExpression->productExpression->baseExpression，直到这里最后匹配到这个“*”

```
1 protected lazy val baseExpression: Parser[Expression] =
2 ( "*" ^^^ UnresolvedStar(None)
3 | rep1(ident <~ ".") <~ "*" ^^ { case target =>
    ↪  UnresolvedStar(Option(target))}
4 | primary
5 )
```

这里它匹配到的是"*"^^^UnresolvedStar(None)[①] 所以最终的处理结果为 UnresolvedStar(None)。同理匹配 relations 的时候最终得到的是 UnresolvedRelation(tableIdent, alias)

所以说 sql 语句在转义完之后，首先生成的是 Unresolved LogicalPlan。

而程序段2.3的下面经过模式匹配的处理就是在构建 tree。

总结如下：

1. 匹配 SELECT 语句，获取 DISTINCT 语句、投影字段 projecti、From 表 relations、WHERE 后表达式、GROUP BY 后表达式、HAVING 后的表达式、ORDER 以及 LIMIT；
2. case 匹配中将匹配的字符串层层封装为 Filter、Aggregate、Project、Distinct、Sort、Limit，这些即为一个 TreeNode，最终形成一颗 LogicalPlan 的 Tree。

下面针对我们的 Sql 语句解析如下

```
val base = r.getOrElse(OneRowRelation)
```

首先呢，这个 r 匹配的是 (FROM ~> relations).?，这里？就是 opt 的操作。根据上面对 ~> 和 opt 的说明，最终产生的结果为 relations 处理的结果，即为 UnresolvedRelation(tableIdent, alias)

```
val withFilter = f.map(Filter(_, base)).getOrElse(base)
```

f 匹配的是 (WHERE ~> expression).?，而上文中提供的 sql 语句为 SELECT * FROM src，所以它匹配不到 WHERE，所以 f 为 None。这里 withFilter 最终获取的就是 base 的值，即 UnresolvedRelation(tableIdent, alias)

```
val withProjection = g
.map(Aggregate(_, p.map(UnresolvedAlias(_)), withFilter))
.getOrElse(Project(p.map(UnresolvedAlias(_)), withFilter))
```

g 匹配的是 (GROUP~BY~> rep1sep(expression, ",")).?，所以 g 也是 None，那么这段代码最终的结果 withProjection 就是 Project(p.map(UnresolvedAlias(_)), withFilter)。

最后形成的 Tree 如下图2.1所示

① 对于 p^^^v，当 p 成功时，丢弃 p 的结果，返回 v 的结果

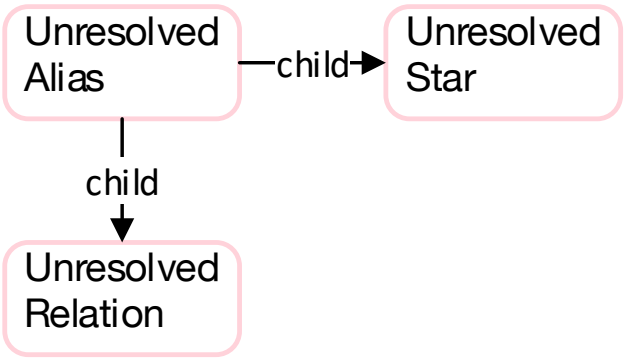


图 2.1　Sql 解析后 UnresolvedLogicalPlan

接下来看 d，h，o，l 的匹配结果，均为 None，所以默认值均是取得 withProjection 的值，所以最终的 tree 的构建结果图2.1。

第 3 章　Spark SQL 源码分析--分析器

3.1　Spark SQL 分析器概述

Analyzer 的功能就是将 SQL Parser 生成的 Unresolve LogiacalPlan 与数据字典 Catalog 进行绑定，生成 Logiacal Plan，在 SQLContext 中，Analyzer 的唯一入口是 executePlan，如下所示

```
protected[sql] def executePlan(plan: LogicalPlan) =
new sparkexecution.QueryExecution(this, plan)
```

Analyzer 中的核心是它里面定义的一个 batches，它是由一系列 Batch 组成的序列，每个 Batch 中定义了规则名、规则所按照的策略以及若干个具体的规则。这里面就包括了 ResolveReferences、ResolveRelations、StarExpansion 等，进行分析器的加工，Unresolved LogicalPlan 已经转化为 Resolved LogicalPlan。分析器中的核心代码如程序3.1所示

</>　程序 **3.1:** 分析器核心部分　</>

```
//下行代码位于 RuleExecutor 中，Batch 的数据结构
protected case class Batch(name: String, strategy: Strategy, rules:
  ↪  Rule[TreeType]*)
//相当于迭代次数的上限，默认为 100
val fixedPoint = FixedPoint(maxIterations)
val extendedResolutionRules: Seq[Rule[LogicalPlan]] = Nil
//Batch: 批次，这个对象是由一系列 Rule 组成的，采用一个策略（策略其
  ↪  实是迭代几次的别名，现包含两种：Once 和 FixedPoint）
//Rule: 理解为一种规则，这种规则会应用到 Logical Plan 从而将
  ↪  UnResolved 转变为 Resolved，在 Batch 中的第三个参数就是 Rule，下
  ↪  面就是一些实现的 Rule。
lazy val batches: Seq[Batch] = Seq(
Batch("Substitution", fixedPoint,
    CTESubstitution,
    WindowsSubstitution),
Batch("Resolution", fixedPoint,
```

```
    ResolveRelations ::
    ResolveReferences ::
    ResolveGroupingAnalytics ::
    ResolvePivot ::
    ResolveUpCast ::
    ResolveSortReferences ::
    ResolveGenerate ::
    ResolveFunctions ::
    ResolveAliases ::
    ExtractWindowExpressions ::
    GlobalAggregates ::
    ResolveAggregateFunctions ::
    HiveTypeCoercion.typeCoercionRules ++
    extendedResolutionRules: _*),
Batch("Nondeterministic", Once,
    PullOutNondeterministic,
    ComputeCurrentTime),
Batch("UDF", Once,
    HandleNullInputsForUDF),
Batch("Cleanup", fixedPoint,
    CleanupAliases)
)
```

Analyzer 解析主要是根据这些 Batch 里面定义的策略[①]和 Rule 来对 Unresolved 的逻辑计划进行解析的。而对于真正的处理操作，其实是在 Analyzer 的继承类 RuleExecutor 的 execute 来实现的。

上述触发分析器的流程如程序3.2所示，这里程序我提取了具体执行过程中的代码示例

</>　程序 3.2: 分析器的触发流程　</>

```
//1
val query = sql("SELECT * FROM src")
```

① 其实策略就两种选项 once 和 fixpoint

```
//2
def sql(sqlText: String): DataFrame = {
    //这里我们已经获得了 Unresolved LogicalPlan
    DataFrame(this, parseSql(sqlText))
}
//3
private[sql] object DataFrame {
    def apply(sqlContext: SQLContext, logicalPlan: LogicalPlan): DataFrame = {
        new DataFrame(sqlContext, logicalPlan)
    }
}
//4
def this(sqlContext: SQLContext, logicalPlan: LogicalPlan) = {
    this(sqlContext, {
        val qe = sqlContext.executePlan(logicalPlan)
            if (sqlContext.conf.dataFrameEagerAnalysis) {
            qe.assertAnalyzed() // This should force analysis and throw errors if there are any
        }
        qe
    })
}
//5
protected[sql] def executePlan(plan: LogicalPlan) = new sparkexecution.QueryExecution(this, plan)
//6 最后在 QueryExecution 这个类的初始化的时候，使用了 execute 这个方法：
class QueryExecution(val sqlContext: SQLContext, val logical: LogicalPlan) {
    ......
    lazy val analyzed: LogicalPlan = sqlContext.analyzer.execute(logical)
```

```
30      ......
31  }
```

最后分析器调用的 execute 这个方法其实是它的父类 RuleExecutor 的方法，下一章要讲解的优化器的执行也是调用该方法。

RuleExecutor 的 execute 方法执行过程描述如下：通过遍历，取出 batches 里缓存的每个 Batch；再遍历每个 Batch 中的 Rule，按照策略指定的次数应用到 TreeNode，如果给 TreeNode 应用 Rule 后的 TreeNode 与之前相同，会退出当前 Batch。

RuleExecutor 的继承体系。如下图3.1所示

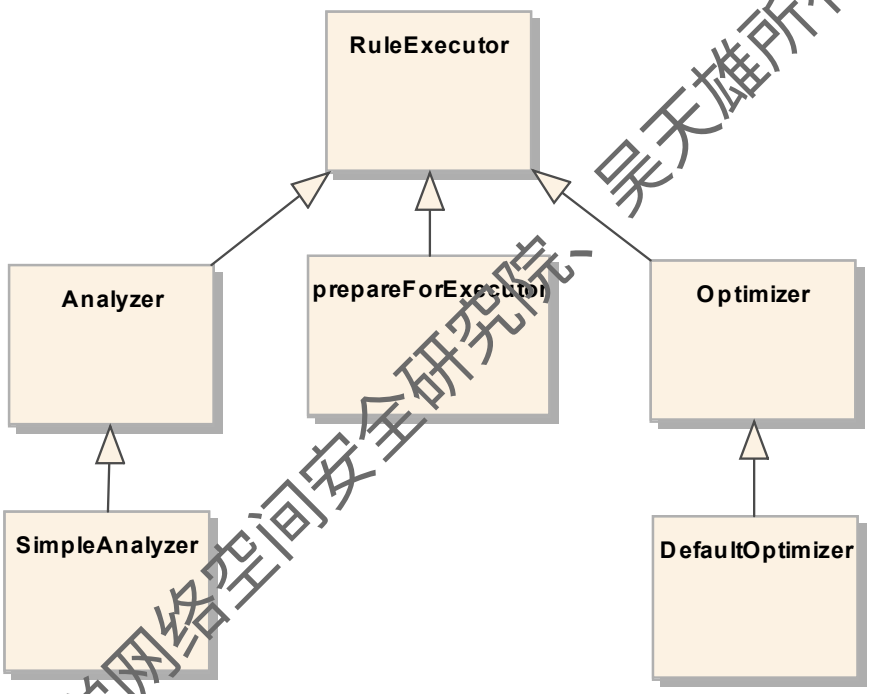


图 3.1　RuleExecutor 的继承体系

RuleExecutor 中 execute 详细介绍如程序3.3所示

程序 **3.3:** *RuleExecutor.execute*

```
1 //执行这个类的子类（Analyzer）定义的 batches（也就 Analyzer.scala 中
  ↪ 提到的那些 batches），batches 中的 batch 会一个接一个的被调用
2 //使用 batch 中定义的策略（Once，FixedPoint），rules 中的 rule 也会一
  ↪ 个接一个的被调用
3 def execute(plan: TreeType): TreeType = {
4     var curPlan = plan
```

```
5     batches.foreach { batch =>
6         val batchStartPlan = curPlan
7         var iteration = 1
8         var lastPlan = curPlan
9         var continue = true
10        while (continue) {
11            //调用 rules 中每一个 rule 作用于计划
12            //flodLeft 是柯里化的一种实现
13            // 这个 result 就是 curPlan, 而这句代码的最终实现就是依
              ↪ 次是用 rules 中的 rule 和 curPlan 来当作输入参数。
14            curPlan = batch.rules.foldLeft(curPlan) {
15                case (plan, rule) =>
16                val startTime = System.nanoTime()
17                //调用 rule 递归对整个 tree 进行处理
18                //这里很显然他是调用了这个 rule 的 apply 方法
19                val result = rule(plan)
20                val runTime = System.nanoTime() - startTime
21                RuleExecutor.timeMap.addAndGet(rule.ruleName, runTime)
22                if (!result.fastEquals(plan)) {
23                    //如果处理后的 plan 有变化, 就打印相关信息
24                    logTrace(
25                    s"""
26                    |=== Applying Rule ${rule.ruleName} ===
27                    |${sideBySide(plan.treeString,
    ↪ result.treeString).mkString("\n")}
28                    """.stripMargin)
29                }
30                result
31            }
32            //batch 迭代次数 +1
33            iteration += 1
34            //如果迭代次数大于最大值, 就停止
35            if (iteration > batch.strategy.maxIterations) {
```

```
                if (iteration != 2) {
                    logInfo(s"Max iterations (${iteration - 1}) reached for batch ${batch.name}")
                }
                continue = false
            }
            //如果处理的 plan 没变化，也停止
            if (curPlan.fastEquals(lastPlan)) {
                logTrace(
                s"Fixed point reached for batch ${batch.name} after ${iteration - 1} iterations.")
                continue = false
            }
            lastPlan = curPlan
        }
        //如果处理的 plan 有变化，打印相关信息
        if (!batchStartPlan.fastEquals(curPlan)) {
            logDebug(
            s"""
            |=== Result of Batch ${batch.name} ===
            |${sideBySide(plan.treeString, curPlan.treeString).mkString("\n")}
            """.stripMargin)
        } else {
            logTrace(s"Batch ${batch.name} has no effect.")
        }
    }
    curPlan
}
```

下面以上节中生成的 Unresolve Logical Plan 为例对规则处理进行讲解。

3.2　UnresolvedRelation 的处理

ResolverResolations 用来把 LogicalPlan 中匹配的 UnresolveRelation 的部分，替换成字典表 Catalog 中注册的 Logical Plan，Analyzer 中的 ResolveRelations 如程序3.4所示

</>　程序 **3.4:** *Analyzer 中的 ResolveRelations*　</>

```
object ResolveRelations extends Rule[LogicalPlan] {
    //下面这个 rule 的作用就是通过 table 从 catalog 中获取相应的
      ↪  relation
    def getTable(u: UnresolvedRelation): LogicalPlan = {
        try {
            //获取 table 中的关系
            catalog.lookupRelation(u.tableIdentifier, u.alias)
        } catch {
            case _: NoSuchTableException =>
            u.failAnalysis(s"Table not found: ${u.tableName}")
        }
    }
    def apply(plan: LogicalPlan): LogicalPlan = plan resolveOperators
      ↪  {
        case i@InsertIntoTable(u: UnresolvedRelation, _, _, _, _) =>
            i.copy(table = EliminateSubQueries(getTable(u)))
        case u: UnresolvedRelation =>
            try {
                getTable(u)
            } catch {
                case _: AnalysisException if
                  ↪  u.tableIdentifier.database.isDefined =>
        }
    }
}
```

程序3.4中的 resolveOperators 方法设定了 Tree 节点的遍历方式，这里遍历的顺序类似于二叉树遍历中的后序遍历，具体的代码介绍如程序3.5所示

</>　程序 **3.5:** *Analyzer* 中的 *resolveOperator*　</>

```
//优先处理它的子节点，然后再处理自己，其实说白了就是树的递归调用。唯一需要注意的是，他会忽略掉已经处理过的子节点.
def resolveOperators(rule: PartialFunction[LogicalPlan, LogicalPlan]): LogicalPlan = {
  if (!analyzed) {
    //处理它的子节点，这个 transformChildren 大家自己看一下，其实很简单，就是一个递归调用
    val afterRuleOnChildren = transformChildren(rule, (t, r) => t.resolveOperators(r))
    //如果处理后的 LogicalPlan 和这一次的相等就说明他没有子节点了，则处理它自己
    if (this fastEquals afterRuleOnChildren) {
      CurrentOrigin.withOrigin(origin) {
        rule.applyOrElse(this, identity[LogicalPlan])
      }
    } else {
      CurrentOrigin.withOrigin(origin) {
        rule.applyOrElse(afterRuleOnChildren, identity[LogicalPlan])
      }
    }
  } else {
    this
  }
}
```

下面我们通过本地模式调试的方式获取了 resolveRelation 规则应用前和应用后的变量 plan 的值，如图3.2和3.3所示

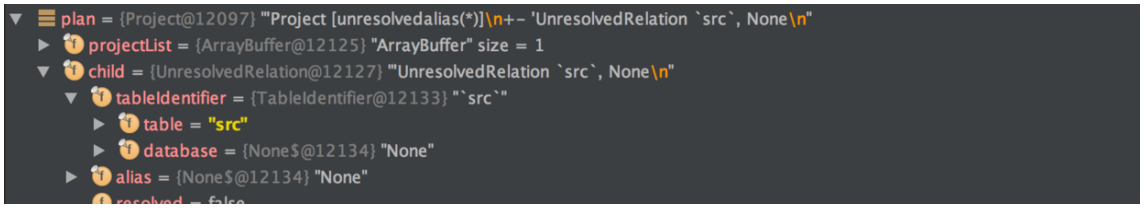


图 3.2　resolveRelation 规则应用前 plan

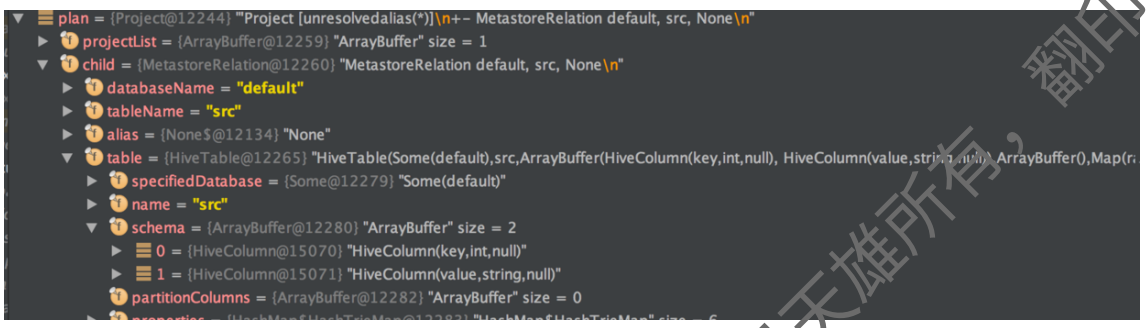


图 3.3　resolveRelation 规则应用后 plan

从图3.3中我们可以看出，经过 resolveRelation 规则处理后的 plan 将 UnresolvedRelation 转换为 MetastoreRelation，这里就完成了对数据库名、表名以及表模式等的绑定。

3.3　UnresolvedStar 的处理

ResolveReferences 规则用来处理 UnresolvedStar。其处理流程和上节对 UnresolvedRelation 的处理类似，关于这个类没有的部分我就不写了，ResolveReferences 关键代码如程序3.6所示。

</>　程序 **3.6:** *ResolveReferences*　</>

```
object ResolveReferences extends Rule[LogicalPlan] {
  .....
  //看到这里你会发现他又使用了这个 resolveOperators 方法，所以你应该明白后续操作了
  def apply(plan: LogicalPlan): LogicalPlan = plan resolveOperators {
    case p: LogicalPlan if !p.childrenResolved => p
    // If the projection list contains Stars, expand it.
```

```
        case p @ Project(projectList, child) if
          ↪  containsStar(projectList) =>
            Project(
                projectList.flatMap {
                case s: Star => s.expand(child, resolver)
                //后面都是 case，因为这里最终只是匹配到了上面这个
                  ↪  case，下面的大家自己看
                ......
        },child)
```

接着直接看这个它匹配到的方法 case s: Star => s.expand(child, resolver)，这里他是调用的 UnresolvedStar 中的 expand 方法，如下程序段所示：

```
override def expand(input: LogicalPlan, resolver: Resolver):
  ↪  Seq[NamedExpression] = {
    // First try to expand assuming it is table.*.
    val expandedAttributes: Seq[Attribute] = target match {
        // If there is no table specified, use all input attributes.
        //通过前面我们得知这里的这个 target 其实就是我们上章中得到的
          ↪  UnresolvedStar(None)
        //所以最终匹配的是这里
        case None => input.output
        // If there is a table, pick out attributes that are part of
          ↪  this table.
        case Some(t) => if (t.size == 1) {
            input.output.filter(_.qualifiers.exists(resolver(_,
              ↪  t.head)))
        } else {
            List()
        }
    }
    if (expandedAttributes.nonEmpty) return expandedAttributes
    //对于后面的代码，他处理的是形如 SELECT record.* from (SELECT
      ↪  struct(a,b,c) as record ...) 这样的 sql 语句
```

```
17 }
```

经过 ResolveReferences 规则处理后的输出如图3.4所示

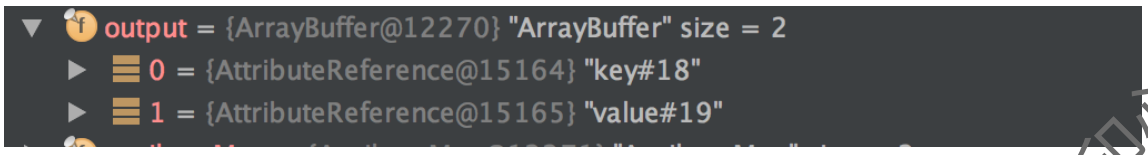


图 3.4　ResolveReferences 规则应用后 output

最后处理完之后的 plan 如图3.5所示

图 3.5　ResolveReferences 规则应用后 plan

到这里整个的分析过程就完成了，其实它这里就完成了一项任务，就是把 sql 语句和数据库进行绑定。

第 4 章　Spark SQL 源码分析--优化器

优化器的入口和分析器的入口都是一样的，都是在 SQLContext 里。优化器是用来将分析器处理得到的 Resolved LogicalPlan 转化为 optimized LogicalPlan 的。QueryExecution 中执行优化的代码如下所示

```
//withCachedData 即为解析完之后的 logicalPlan
lazy val optimizedPlan: LogicalPlan =
  ↪  sqlContext.optimizer.execute(withCachedData)
```

它的执行也是和 Analyzer 一样，也是通过父类 RuleExecutor 的 execute 方法来应用自己的 batches，Optimizer 的默认实现是 DefaultOptimizer，DefaultOptimizer 也内置了很多的 Rule，比如 NullPropagation、ConstantFolding 等。经过 Optimizer 对 Resolved Logical Plan 的优化，就生成了 Optimized LogicalPlan。

优化器的具体代码如程序4.1所示

</> 程序 **4.1:** *Optimizer* 核心代码 </>

```
abstract class Optimizer(conf: CatalystConf) extends
  ↪  RuleExecutor[LogicalPlan] {
    val batches =
    //删除子查询
    Batch("Remove SubQueries", FixedPoint(100),
        EliminateSubQueries) ::
    //合并
    Batch("Aggregate", FixedPoint(100),
        //重写了包含所有 Distinct 的合并
        DistinctAggregationRewriter(conf),
        ReplaceDistinctWithAggregate,
        //去除一些 group 操作中 foldable 的表达式
        RemoveLiteralFromGroupExpressions) ::
    Batch("Operator Optimizations", FixedPoint(100),
        // Operator push down
        SetOperationPushDown,
```

```
        SamplePushDown,
        //谓词下推
        PushPredicateThroughJoin,
        PushPredicateThroughProject,
        PushPredicateThroughGenerate,
        PushPredicateThroughAggregate,
        //列剪枝
        ColumnPruning,
        // Operator combine
        ProjectCollapsing,
        CombineFilters,
        CombineLimits,
        // Constant folding
        //空格处理
        NullPropagation,
        //关键字 In 的优化，替代为 InSet
        OptimizeIn,
        //常量叠加
        ConstantFolding,
        //表达式简化
        LikeSimplification,
        BooleanSimplification,
        RemoveDispensableExpressions,
        SimplifyFilters,
        SimplifyCasts,
        SimplifyCaseConversionExpressions) ::
        //精度优化
    Batch("Decimal Optimizations", FixedPoint(100),
        DecimalAggregates) ::
        //关系转换
    Batch("LocalRelation", FixedPoint(100),
        ConvertToLocalRelation) :: Nil
}
```

这部分的优化理解需要丰富的 SQL 优化经验，这里我们只理解它整个优化流程即可。

无论是 Analyzer 中内置的 Rule，还是 DefaultOptimizer 内置的 Rule，将 Rule 应用到 LogicalPlan 都是通过 TreeNode 里的 transform 系列函数。比方说规则 SimplifyFilter 为例，代码如下所示

```
object SimplifyFilters extends Rule[LogicalPlan] {
    def apply(plan: LogicalPlan): LogicalPlan = plan transform {
        case Filter(Literal(true, BooleanType), child) => child
        case Filter(Literal(null, _), child) =>
          ↪  LocalRelation(child.output, data = Seq.empty)
        case Filter(Literal(false, BooleanType), child) =>
          ↪  LocalRelation(child.output, data = Seq.empty)
    }
}
```

这里所做的优化包括

1. 如果过滤条件总是等于 true，则删除它，即此过滤条件不起作用；
2. 如果过滤条件总是等于 false 或者 null，将输入替换为空的 relation，即将输入全部滤除。

我们也可以看到优化器的 Rule 是通过调用 transform 来作用于 plan 的，其实优化器下所有的 rule 都是通过 transform 实操作 plan。transform 代码就一行调用了 transformDown 的函数，我们直接看 transformDown 方法，如程序4.2所示

</>　程序 **4.2:** *transformDown*　</>

```
def transformDown(rule: PartialFunction[BaseType, BaseType]): BaseType
  ↪  = {
    //这里先优化根节点，是先序遍历
    val afterRule = CurrentOrigin.withOrigin(origin) {
        rule.applyOrElse(this, identity[BaseType])
    }
    // Check if unchanged and then possibly return old copy to avoid
      ↪  gc churn.
    if (this fastEquals afterRule) {
```

```
8         //如果没有改变的话，它就直接返回 rule，而不是每一次都会有产
            ↪ 生一个 afterRule，从而最终有可能导致 gc
9         transformChildren(rule, (t, r) => t.transformDown(r))
10    } else {
11        //如果有改变，就操作它的子节点
12        afterRule.transformChildren(rule, (t, r) =>
            ↪ t.transformDown(r))
13    }
14 }
```

从程序4.2看出这里它又是使用先序便利的方式来实现的。

因为优化器和分析器最终都是调用 RuleExecutor 的 execute 方法，过程都是一样的，所以这里我就没有过多的分析。

第 5 章　Spark SQL 源码分析--物理计划生成

经过了 SqlParser、Analyzer、Optimizer 的处理，SQL 语句就转换为了优化后的 Logical Plan，这时还无法被当做一般的 Job 来处理，为了能够将 Logical Plan 按照其他的 Job 一样对待，需要将 Logical Plan 转换为 Physical Plan，说直接点就是将 sql 语句转化为可以直接操作真实数据的操作及数据和 RDD 的绑定。

5.1　物理执行计划 SparkPlan

首先使用 SparkPlanner，这里直接处理的是从优化器优化得到的 optimizerPlan，将其转化为 PhysicalPlan，程序段如下所示

```
lazy val sparkPlan: SparkPlan = {
    SQLContext.setActive(sqlContext)
    sqlContext.planner.plan(optimizedPlan).next()
}
```

SparkPlanner 继承类关系如图5.1所示

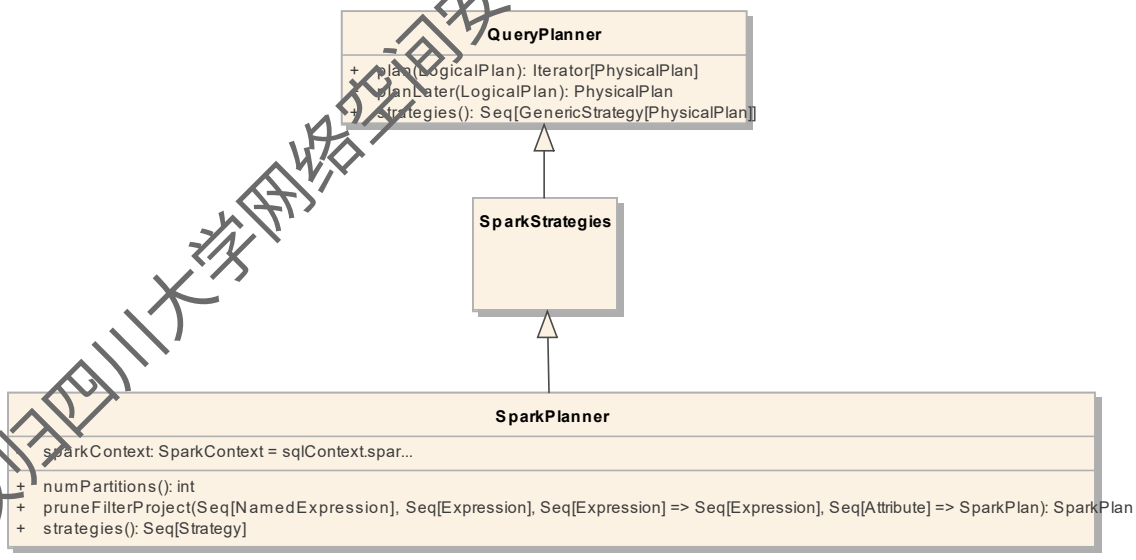


图 5.1　SparkPlannerr 的继承体系

SparkPlanner 的 plan 方法，会返回一个 Iterator[PhysicalPlan]。SparkPlanner 继承了 SparkStrategies，SparkStrategies 继承了 QueryPlanner。SparkStrategies 包含了一系列特定的 Strategies，这些 Strategies 是继承自 QueryPlanner 中定义的 Strategy，

它定义接受一个 Logical Plan，生成一系列的 Physical Plan。SparkPlanne 的核心代码如程序5.1所示

</>　程序 **5.1:** *SparkPlanner*　</>

```
class SparkPlanner(val sqlContext: SQLContext) extends SparkStrategies {
    val sparkContext: SparkContext = sqlContext.sparkContext
    //指定运行时的 partitions 的个数
    def numPartitions: Int = sqlContext.conf.numShufflePartitions
    //需要执行的策略
    def strategies: Seq[Strategy] =
        sqlContext.experimental.extraStrategies ++ (
        DataSourceStrategy ::
        DDLStrategy ::
        TakeOrderedAndProject ::
        Aggregation ::
        LeftSemiJoin ::
        EquiJoinSelection ::
        InMemoryScans ::
        BasicOperators ::
        BroadcastNestedLoop ::
        CartesianProduct ::
        DefaultJoin :: Nil)
    ......
}
```

SparkPlanner 会调用其祖父类 QueryPlanner 中的 plan 方法来对 optimizedPlan 应用策略，QueryPlanner 中的 plan 方法如下程序段所示

```
def plan(plan: LogicalPlan): Iterator[PhysicalPlan] = {
    // Obviously a lot to do here still...
    val iter = strategies.view.flatMap(_(plan)).toIterator
    assert(iter.hasNext, s"No plan for $plan")
    iter
}
```

可以看出它会遍历 SparkPlanner 中的 strategies，将每个策略作用于 optimized-Plan。

5.2　SparkPlanner 实例分析

这里以 Spark-1.6.3 源码自带的 people.json 作为数据源来讲解，其路径位于 examples/src/main/resources/people.json。我们的 spark 程序段如下所示

```
val df =
  ↪  hc.read.json("/home/spark-release-HDP-2.5.0.3-tag/examples/src/main/resources/peop
//下面的操作相当于 select * from table
df.registerTempTable("src")
val query = hc.sql("select * from src where age > 10")
```

其中 people.json 中的内容为

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

它形成的 optimizedLogicalPlan 如下所示

```
Project [age#0L,name#1]
+- Filter (age#0L > 10)
+- Relation[age#0L,name#1] JSONRelation
```

这里第一个应用到策略为 DataSourceStrategy，策略均为 Object 单例类，其主要是调用它的 apply 方法，DataSourceStrategy 中 apply 方法的核心代码如程序5.2所示

程序 **5.2:** *DataSourceStrategy*

```
def apply(plan: LogicalPlan): Seq[execution.SparkPlan] = plan match {
    ......
    // Scanning non-partitioned HadoopFsRelation
    case PhysicalOperation(projects, filters, l @ LogicalRelation(t:
      ↪  HadoopFsRelation, _)) =>
```

```
        // See buildPartitionedTableScan for the reason that we need
          ↪ to create a shard
        // broadcast HadoopConf.
        val sharedHadoopConf = SparkHadoopUtil.get.conf
        val confBroadcast =
        t.sqlContext.sparkContext.broadcast(new
          ↪ SerializableConfiguration(sharedHadoopConf))
        //这里最关键的是这个函数，它的作用就是再修剪
          ↪ OptimizedLogicalPlan，并和 datasource（mysql，json，文件，
          ↪ Par//quet 文件，这里是 json 文件）绑定。这里它修剪的就是
          ↪ project 和 filter。
        pruneFilterProject(l,projects,filters,
            (a, f) => t.buildInternalScan(a.map(_.name).toArray, f,
              ↪ t.paths, confBroadcast)) :: Nil
        ......
}
```

程序5.2中的 pruneFilterProject 直接调用的就是 pruneFilterProjectRaw 方法，pruneFilterProjectRaw 具体的解释一行行进行了注解，如程序5.3所示

</>　程序 5.3: *pruneFilterProjectRaw*　</>

```
protected def pruneFilterProjectRaw(
relation: LogicalRelation,
projects: Seq[NamedExpression],
filterPredicates: Seq[Expression],
scanBuilder: (Seq[Attribute], Seq[Expression], Seq[Filter]) =>
  ↪ RDD[InternalRow]) = {
    //这里里获得的这个 projectSet，就是 Optimized LogicalPlan 中
      ↪ project 的参数，即 age#0L,name#1
    val projectSet = AttributeSet(projects.flatMap(_.references))
    //这里获取的就是过滤条件的参数，在这个例子中是 age#0L
    val filterSet =
      ↪ AttributeSet(filterPredicates.flatMap(_.references))
```

```
10      //过滤出谓词的如 between, like, 在这个例子中是 age#0L>0
11      //所谓谓词，就是取值为 TRUE、FALSE 或 UNKNOWN 的表达式。谓词用于
          ↪ WHERE 子句和 HAVING 子句的搜索条件中，还用于 FROM 子句的联
          ↪ 接条件以及需要布尔值的其他构造中
12      val candidatePredicates = filterPredicates.map { _ transform {
13              case a: AttributeReference => relation.attributeMap(a) //
                  ↪ Match original case of attributes.
14      }}
15      //提取出没有被处理的处理的表达式，及其可转化的 pushdown 谓词。这
          ↪ 里是（(age#0L > 10), GreaterThan(age,10)）
16      val (unhandledPredicates, pushedFilters) =
17      selectFilters(relation.relation, candidatePredicates)
18      //提取出仅用于 pushdown 的参数，这里是 nil，因为它不需要再
          ↪ pushdown
19      val handledSet = {
20          val handledPredicates =
              ↪ filterPredicates.filterNot(unhandledPredicates.contains)
21          val unhandledSet =
              ↪ AttributeSet(unhandledPredicates.flatMap(_.references))
22          AttributeSet(handledPredicates.flatMap(_.references)) --
23          (projectSet ++ unhandledSet).map(relation.attributeMap)
24      }
25      //合并不能转化为数据源中的表达式及没有被处理的 filter，这里是
          ↪ Some((age#0L > 10))
26      val filterCondition =
          ↪ unhandledPredicates.reduceLeftOption(expressions.And)
27      //获取元数据信息
28      val metadata: Map[String, String] = {
29          val pairs = ArrayBuffer.empty[(String, String)]
30          if (pushedFilters.nonEmpty) {
31              pairs += (PUSHED_FILTERS -> pushedFilters.mkString("[", ",
                  ↪ ", "]"))
32          }
```

```
33          relation.relation match {
34              case r: HadoopFsRelation => pairs += INPUT_PATHS ->
                ↪  r.paths.mkString(", ")
35              case _ =>
36          }
37          pairs.toMap
38      }
39      if (projects.map(_.toAttribute) == projects &&
40      projectSet.size == projects.size &&
41      filterSet.subsetOf(projectSet)) {
42          //当这个 project 的参数，足以来处理 filter 的条件，就直接，构
                ↪  建这个 scan
43          //提取这些满足条件的参数，也就是列名
44          val requestedColumns = projects
45          // Safe due to if above.
46          .asInstanceOf[Seq[Attribute]]
47          // Match original case of attributes.
48          .map(relation.attributeMap)
49          // Don't request columns that are only referenced by pushed
                ↪  filters.
50          .filterNot(handledSet.contains)
51          //创建这个 scan，其实就是创建一个 PhysicalRDD，而这个
                ↪  PhysicalRDD 是间接继承自 SparkPlan
52          //另外这里也创建了一个真正的 rdd，可以通过 PhysicalRDD 获取
53          val scan = execution.PhysicalRDD.createFromDataSource(
54          projects.map(_.toAttribute),
55          scanBuilder(requestedColumns, candidatePredicates,
                ↪  pushedFilters),
56          relation.relation, metadata)
57          filterCondition.map(execution.Filter(_, scan)).getOrElse(scan)
58      } else {
59          //这里就是去掉仅仅用于 pushdown 的 fileter，用于计算的还是要
                ↪  保留的
```

```
60          val requestedColumns =
61          (projectSet ++ filterSet --
              ↪  handledSet).map(relation.attributeMap).toSeq
62          val scan = execution.PhysicalRDD.createFromDataSource(
63          requestedColumns,
64          scanBuilder(requestedColumns, candidatePredicates,
              ↪  pushedFilters),
65          relation.relation, metadata)
66          execution.Project(
67              projects, filterCondition.map(execution.Filter(_,
                ↪  scan)).getOrElse(scan))
68      }
69  }
```

它是怎么创建 RDD 的呢？也就是在创建 scan 的时候，传入的那个方法参数 scanBuilder，我们先定位到最初匹配的位置程序5.2，也就是

```
1 (a, f) => t.buildInternalScan(a.map(_.name).toArray, f, t.paths,
    ↪  confBroadcast)) :: Nil
```

这里的方法为 buildInternalScan，具体实现如程序5.4所示

</>　程序 **5.4:** *buildInternalScan*　</>

```
1  final private[sql] def buildInternalScan(
2  requiredColumns: Array[String],
3  filters: Array[Filter],
4  inputPaths: Array[String],
5  broadcastedConf: Broadcast[SerializableConfiguration]):
     ↪  RDD[InternalRow] = {
6      //据上面的处理中得到的 inputPaths，来获取数据的位置
7      //他先认为这个路径是一个目录，读取该目录下的文件进行处理
8      //而后才会处理它是是一个文件
9      val inputStatuses = inputPaths.flatMap { input =>
10         val path = new Path(input)
```

```
        // First assumes `input` is a directory path, and tries to get
          ↪  all files contained in it.
        fileStatusCache.leafDirToChildrenFiles.getOrElse(
        path,
        // Otherwise, `input` might be a file path
        fileStatusCache.leafFiles.get(path).toArray
        ).filter { status =>
            val name = status.getPath.getName
            !name.startsWith("_") && !name.startsWith(".")
        }
    }
    //然后就是通过这个文件来构建 rdd
    buildInternalScan(requiredColumns, filters, inputStatuses,
      ↪  broadcastedConf)
}
```

程序5.4最后一行代码调用的 buildInternalScan 方法是使用的 JSONRelation 中的 buildInternalScan 方法，如下程序5.5所示

</>　程序 **5.5:** *JSONRelation.buildInternalScan*　</>

```
override private[sql] def buildInternalScan(
requiredColumns: Array[String],
filters: Array[Filter],
inputPaths: Array[FileStatus],
broadcastedConf: Broadcast[SerializableConfiguration]):
  ↪  RDD[InternalRow] = {
    // 获取需要处理的列的 Schema 信息，其实就是其对应的类型。
    val requiredDataSchema =
      ↪  StructType(requiredColumns.map(dataSchema(_)))
    //这里就是根据 requiredDataSchema 将 json 格式的数据转化为
      ↪  InternalRow
    val rows = JacksonParser.parse(
    inputRDD.getOrElse(createBaseRdd(inputPaths)),
```

```
11      requiredDataSchema,
12      sqlContext.conf.columnNameOfCorruptRecord,
13      options)
14      //最后将 rdd 中 InternalRow 转化为 UnsafeRow，因为 UnsafeRow 的数
          ↪ 据是直接放在内存上的，而不是 java 对象
15      rows.mapPartitions { iterator =>
16          val unsafeProjection =
              ↪ UnsafeProjection.create(requiredDataSchema)
17          iterator.map(unsafeProjection)
18      }
19  }
```

程序5.5中使用的 createBaseRdd 方法实际上调用了 Spark core 里的构建 hadoopRDD 的方法，这里不在细致介绍。

第 6 章　Spark SQL 源码分析--准备执行

Spark SQL 经过前面的 SparkPlanner 生成了 SparkPlan，然而这个步骤生成的 RDD 其分区策略、分区数等并不能满足生产环境要求，本章即是为 Spark SQL 执行前做本准备工作，准备工作的实例对象为 prepareForExecution，其初始化也是在 SQLContext 中，如下所示

```
protected[sql] val prepareForExecution = new RuleExecutor[SparkPlan] {
    val batches = Seq(
    //用来协调 shuffle 的方法
    Batch("Add exchange", Once, EnsureRequirements(sparkContext)),
    //确保 row 的 formats，因为处理的 row 有可能是 SafeRows 和
     ↪ UnsafeRows，这里需要将处理的记录和相应的格式匹配起来
    Batch("Add row converters", Once, EnsureRowFormats)
    )
}
```

这里它是继承自 RuleExecutor，所以其执行，其实也是使用的 RuleExecutor 的 execute 来实现的，它会遍历 batches，之后再调用每个 Batch 中的每条 rule，调用 rule 的 apply 方法，将规则作用在 plan 上。

第一条规则为 EnsureRequirements，它的 apply 操作会对 plan 的 treeNode 进行遍历，先操作其子节点，再对它自己进行操作。

这里使用这条规则的原因如下

- child 物理计划的输出数据分布不 satisfies 当前物理计划的数据分布要求。比如说 child 物理计划的数据输出分布是 UnspecifiedDistribution，而当前物理计划的数据分布要求是 ClusteredDistribution;
- 对于包含 2 个 Child 物理计划的情况，2 个 Child 物理计划的输出数据有可能不 compatile。因为涉及到两个 Child 物理计划采用相同的 Shuffle 运算方法才能够保证在本物理计划执行的时候一个分区的数据在一个节点，所以 2 个 Child 物理计划的输出数据必须采用 compatile 的分区输出算法。如果不 compatile 需要创建 Exchange 替换掉 Child 物理计划。

6.1　ensureDistributionAndOrdering 函数分析

首先判断 child 物理计划的输出数据分布是否满足当前物理计划的数据分布要求。

```
children = children.zip(requiredChildDistributions).map { case (child,
  ↪  distribution) =>
    if (child.outputPartitioning.satisfies(distribution)) {
        child
    } else {
        Exchange(createPartitioning(distribution,
          ↪  defaultNumPreShufflePartitions), child)
    }
}
```

接着对有两个 children 的情况进行处理，对于包含 2 个 Child 物理计划的情况，2 个 Child 物理计划的输出数据有可能不 compatile，因为涉及到两个 Child 物理计划采用相同的 Shuffle 运算方法才能够保证在本物理计划执行的时候一个分区的数据在一个节点，所以 2 个 Child 物理计划的输出数据必须采用 compatile 的分区输出算法。如果不 compatile 需要创建 Exchange 替换掉 Child 物理计划。

```
if (children.length > 1
&& requiredChildDistributions.toSet != Set(UnspecifiedDistribution)
&& !Partitioning.allCompatible(children.map(_.outputPartitioning))) {
    //获取 Children 下最大的 partition 的数量
    //检查 children 的 partition 数量是否都是这个值，换句话说，也就
      ↪  是各个 child 的 Partitioning 是否是一样的
    val useExistingPartitioning =
      ↪  children.zip(requiredChildDistributions).forall {
        ......
    }
    //如果是一致的则不需要 shuffle child 的输出
    children = if (useExistingPartitioning) {
        children
    } else {
        val numPartitions = {
```

```
14              //检查一下是否需要 shuffle 所有的 child 的输出
15              ......
16              //最终的分区数，如果需要 shuffle 所有的 child，则使用默认
                ↪ 分区数，如果仅需要 shuffle 部分 child，则使用上面得
                ↪ 到的最大分区数。
17              if (shufflesAllChildren) defaultNumPreShufflePartitions
                ↪ else maxChildrenNumPartitions
18          }
19          children.zip(requiredChildDistributions).map {
20              case (child, distribution) => {
21                  val targetPartitioning
                    ↪ =createPartitioning(distribution, numPartitions)
22                  //这里需要主要的是如果 child 的 outputPartitioning 和
                    ↪ 最终的 Partitioning 一致的话，就不需要添加
                    ↪ exchange（这个其实就是对应的部分 shuffle 的情
                    ↪ 况），否则则添加
23                  if
                    ↪ (child.outputPartitioning.guarantees(targetPartitioning))
                    ↪ {
24                      child
25                  } else {
26                      child match {
27                          //关于这个 child 如果本身就是一个 Exchange 的
                            ↪ 话，就直接更改它的 Partitioning，如果不
                            ↪ 是则添加一个 Exchange 操作
28                          ......
29                      }
30                  }
31              }
32          }
33      }
34  }
```

最后处理就是判断是否需要对 shuffle 进行排序，如果需要的话，则再添加一个 Sort 操作。

到这里，整个 sparkplan 的协调工作就全部完成了。在这一步处理完之后还需要进行 EnsureRowFormats 处理，这个就是要确保 row 的 formats 的合理性，因为处理的 row 有可能是 SafeRows 和 UnsafeRows，这里需要将处理的记录和相应的格式匹配起来。

第 7 章　Spark SQL 源码分析--执行部分

QueryExecution 类中处理完 executedPlan 后会立即调用 SparkPlan 中的 execute 方法，这个方法的处理逻辑是先进行一些准备工作，接着执行具体的业务逻辑。

```
RDDOperationScope.withScope(sparkContext, nodeName, false, true) {
    prepare()
    doExecute()
}
```

不同的 SparkPlan 的这两个步骤的执行逻辑不一样，接下来以 Exchange 为例进行说明。

Exchange 的 doPrepare 很简单，直接像 exchangeCoordinator 注册了 Exchange。

接下来我们看 Exchange 的 doExecute()，这里的操作逻辑不负责，就通过 exchangeCoordinator 返回了 shuffleRDD。

```
coordinator match {
    //为上面有使用 exchangeCoordinator
    case Some(exchangeCoordinator) =>
        val shuffleRDD = exchangeCoordinator.postShuffleRDD(this)
        assert(shuffleRDD.partitions.length ==
          ↪  newPartitioning.numPartitions)
        shuffleRDD
    case None =>
        val shuffleDependency = prepareShuffleDependency()
        preparePostShuffleRDD(shuffleDependency)
}
```

postShuffleRDD 会处理协调器如何来 shuffle 数据。这里的处理函数为 doEstimationIfNecessary()。此函数有两个重要的函数需要介绍

7.1　prepareShuffleDependency

此部分将返回 ShuffleDependency，这个 ShuffleDependency 用来对它的子 RDD 进行分区。

首先获取它依赖的 RDD，接着为它构建分区器，在接着构建一个用来根据分区获得 key 的提取器，下面展示 HashPartitioner 的分区提取器

```
//这里就是获取一个 Partition 的 key 的提取器
def getPartitionKeyExtractor(): InternalRow => Any = newPartitioning match {
    ......
    case HashPartitioning(expressions, _) =>
      newMutableProjection(expressions, child.output)()
    ......
}
```

接着将此父 RDD 的结构进行转换 RDD[Product2[Int, InternalRow]]，将 partition 中数据的类型改为（partitionID，row），程序如下所示

```
val rddWithPartitionIds: RDD[Product2[Int, InternalRow]] = {
    // needToCopyObjectsBeforeShuffle 是很重要的一个方法，下面就是根据下面的生成新的 RDD
    if (needToCopyObjectsBeforeShuffle(part, serializer)) {
        rdd.mapPartitionsInternal { iter =>
            val getPartitionKey = getPartitionKeyExtractor()
            //改变 partition 中数据的类型为（partitionID, row）
            iter.map { row =>
              (part.getPartition(getPartitionKey(row)), row.copy())
            }
        }
    } else {
        ......
    }
}
```

最终就构建了 dependency，此依赖的父 RDD 为上面得到的 rddWithPartitionIds，这里 partitioner 可不是 HashPartioner，不过他的数据是通过 HashPartioner 来构建的。程序如下

```
val dependency =
//最终就构建了这个 dependency，这里注意这个 partitioner 可不是 HashPartioner，不过他的数据是通过 HashPartioner 来构建的。
new ShuffleDependency[Int, InternalRow, InternalRow](
rddWithPartitionIds,
//这里的这个 PartitionIdPassthrough 只是一个最简单的 Partioner。
new PartitionIdPassthrough(part.numPartitions),
Some(serializer))
```

7.2　estimatePartitionStartIndices

这个方法就是根据前面 rdd 计算得到的数据，来根据 shuffle 后的分区数来来决定这个 rdd 处理的数据属于哪个分区，这里其实就是指定 shuffle 后的各个分区获取哪些数据。

首先是获取 shuffle 后的目标分区数，接着计算前面 RDD shuffle 的分区数，最后通过下面来计算要 shuffle 的数据是否满足在 targetPostShuffleInputSize 这个值以内，如果不满足，则记录下来，在构建 RDD 的时候重新进行分区。

```
val partitionStartIndices = ArrayBuffer[Int]()
// The first element of partitionStartIndices is always 0.
partitionStartIndices += 0
var postShuffleInputSize = 0L
var i = 0
//通过下面来计算要 shuffle 的数据是否满足在 targetPostShuffleInputSize 这个值以内，
// 如果不满足，则记录下来，在构建 RDD 的时候重新进行分区。
while (i < numPreShufflePartitions) {
    var j = 0
    while (j < mapOutputStatistics.length) {
        postShuffleInputSize +=
          mapOutputStatistics(j).bytesByPartitionId(i)
        j += 1
    }
```

```
    if (postShuffleInputSize >= targetPostShuffleInputSize) {
        if (i < numPreShufflePartitions - 1) {
            partitionStartIndices += i + 1
        } else {
        }
        postShuffleInputSize = 0L
    }
    i += 1
}
partitionStartIndices.toArray
```

7.3　doEstimationIfNecessary

接下来我们进入主程序 doEstimationIfNecessary，此方法中我们需要首先提交前面的 map stage，等前面提交的 map stage 完成之后，接着估计产生 shuffle 数据的索引，即上面介绍的 estimatePartitionStartIndices 方法，最终根据 estimatePartitionStartIndices 处理的结果来构建一个 rdd。

```
val rdd = exchange.preparePostShuffleRDD(shuffleDependencies(k),
  ↪  partitionStartIndices)
```

然后最终把获得的这些 rdd 放到 postShuffleRDDs 变量中，其实真正的 shuffle 操作是发生在这些 newPostShuffleRDDs 中的。

```
newPostShuffleRDDs.put(exchange, rdd)

assert(postShuffleRDDs.isEmpty)
assert(newPostShuffleRDDs.size() == numExchanges)
postShuffleRDDs.putAll(newPostShuffleRDDs)
```