Table of Contents

目录

本篇主要介绍一下SparkSQL的使用方法，架构设计和代码分析(基于Spark1.2.0)。

第一章主要介绍SparkSQL的发展历程和性能的优化。

第二章主要介绍SparkSQL的使用方法。

第三章主要介绍SparkSQL的调优。

第四章将深入SparkSQL，分析其架构。

第五章将介绍SparkSQL的优化器Catalyst。

第六章将解析SparkSQL的各个模块。

第七章将推荐一些学习Spark的参考资料。

相关链接

- Gitbook 网页版
- 源代码

SparkSQL介绍

2014年9月SparkSQL发布了1.1.0版本，紧接着在12月又发布了1.2.0版本。SparkSQL发展速度异常迅猛，原因是将SQL类型的查询语言整合到Spark的核心RDD概念里，这样流处理，批处理，包括机器学习里都可以和SQL混合使用。

SparkSQL1.1.0

(release note)

版本主要的变动有：

1. 增加了JDBC/ODBC Server (ThriftServer)，用户可以在应用程序中连接到SparkSQL并使用其中的表和缓存表。
2. 增加了对JSON文件的支持
3. 增加了对parquet文件的本地优化
4. 增加了支持将python、scala、java的lambda函数注册成UDF，并能在SQL中直接使用。
5. 引入了动态字节码生成技术（bytecode generation，即CG），明显地提升了复杂表达式求值查询的速率。
6. 统一API接口，如sql()、SchemaRDD生成等。

SparkSQL1.2.0

(release note)

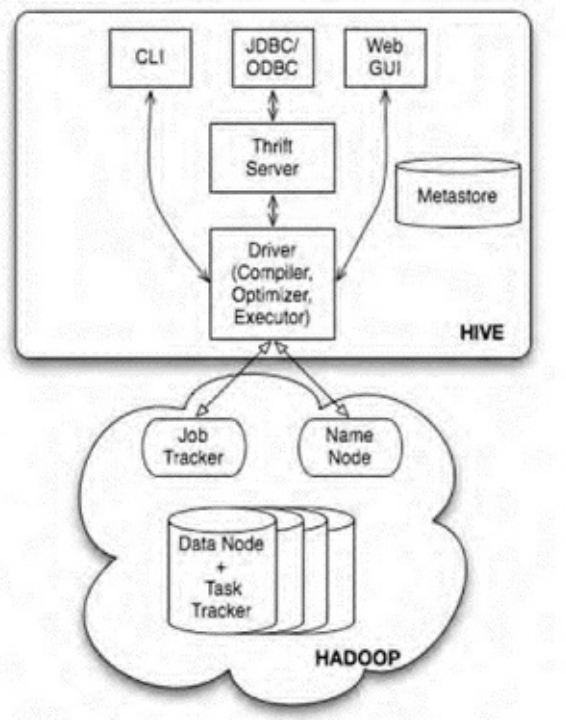
版本主要的变动有：

1. 增加访问外部数据的新API，通过该API可以在运行时将外部数据注册为临时表格，并且支持predicte pushdown方式的优化，Spark原生支持Parquet和JSON格式，用户可以自己编写读取其他格数据的代码。
2. 增加支持Hive 0.13，增加对fixed-precision decimal数据类型的支持。
3. 增加dynamically partitioned inserts
4. 优化了cache SchemaRDD，并重新定义了其语义，并增加了statistics-based partition pruning。

SparkSQL的发展历程

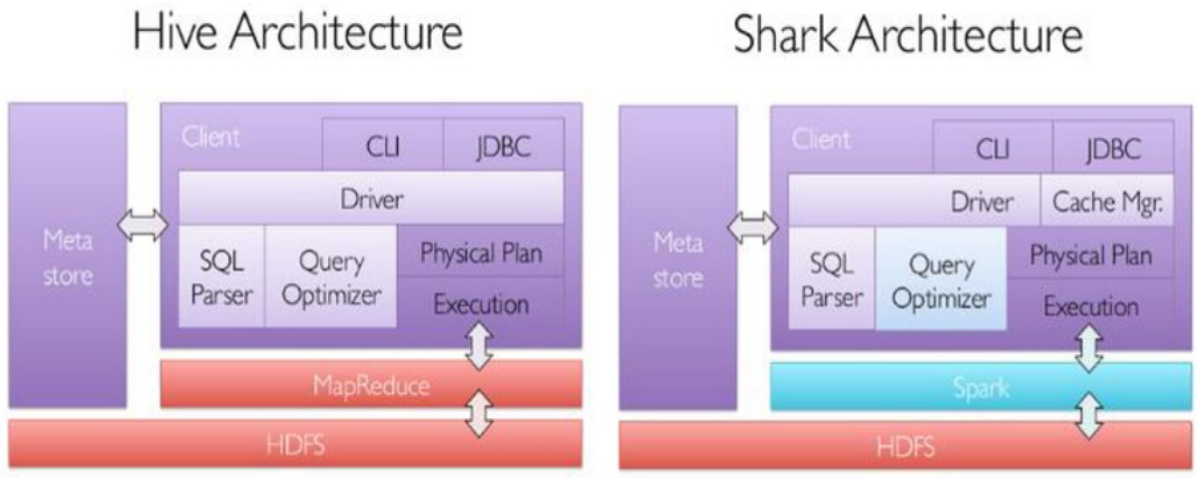
HDFS -> Hive

由于Hadoop在企业生产中的大量使用，HDFS上积累了大量数据，为了给熟悉RDBMS但又不理解MMapReduce的技术人员提供快速上手的工具，Hive应运而生。Hive的原理是将SQL语句翻译成MapReduce计算。

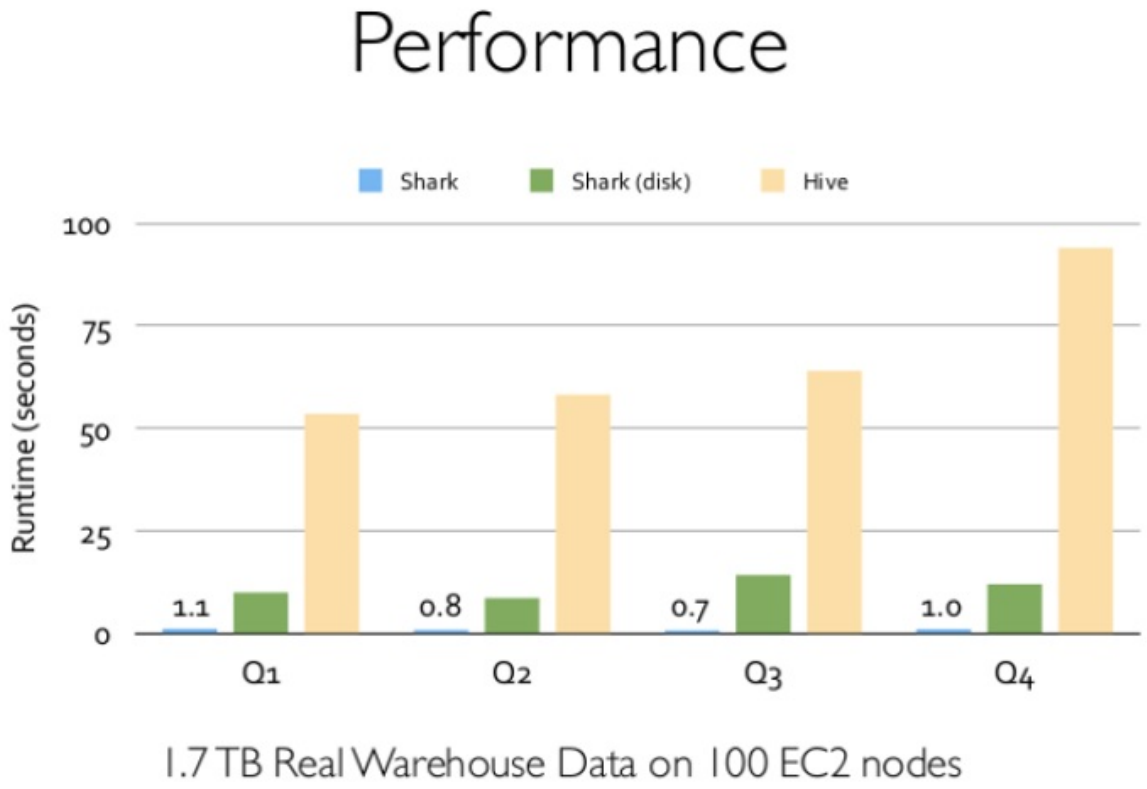


Hive -> Shark

但是，MapReduce计算过程中大量的中间磁盘落地过程消耗了大量的I/O，降低了运行效率，为了提供SQL-on-Hadoop的效率，Shark出现了。



Shark是伯克利AMPLab实验室Spark生态环境的组件之一，它修改了Hive中的内存管理、物理计划和执行三个模块，使得SQL语句直接运行在Spark上，从而使得SQL查询的速度得到10-100倍的提升。



Shark之死

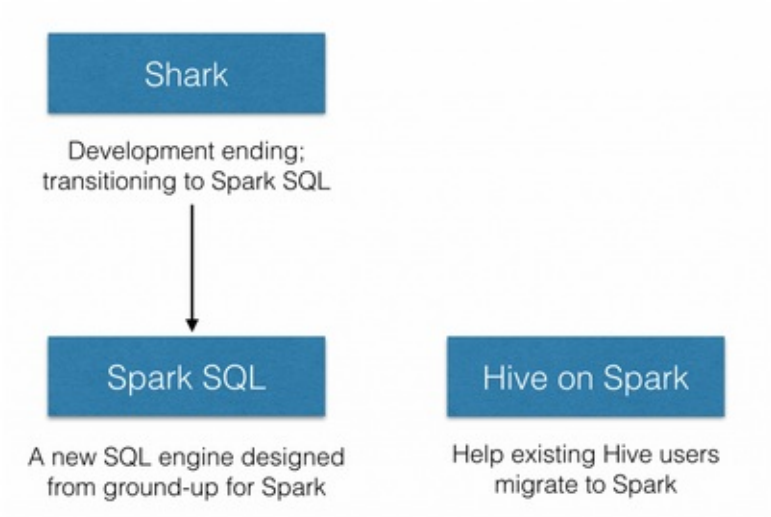
2014年6月1日，Shark项目和SparkSQL项目的主持人Reynold Xin宣布：停止对Shark的开发，团队将所有资源放sparkSQL项目上，至此，Shark的发展画上了句话。Reynold在其微博上发出了下面这段话：

为什么Shark会死呢？Databricks在其官网上给出了答案

Shark built on the Hive codebase and achieved performance improvements by swapping out the physical execution engine part of Hive. While this approach enabled Shark users to speed up their Hive queries, Shark inherited a large, complicated code base from Hive that made it hard to optimize

and maintain. As we moved to push the boundary of performance optimizations and integrating sophisticated analytics with SQL, we were constrained by the legacy that was designed for MapReduce.

随着Spark的发展，Shark对于Hive的太多依赖制约了Spark的One Stack rule them all的方针，制约了Spark各个组件的相互集成，同时Shark也无法利用Spark的特性进行深度优化，所以决定放弃Shark，提出了SparkSQL项目。



随着Shark的结束，两个新的项目应运而生：SparkSQL和Hive on Spark。其中SparkSQL作为Spark生态的一员继续发展，而不再受限于Hive，只是兼容Hive；而Hive on Spark是一个Hive的发展计划，该计划将Spark作为Hive的底层引擎之一，也就是说，Hive将不再受限于一个引擎，可以采用Map-Reduce、Tez、Spark等引擎。

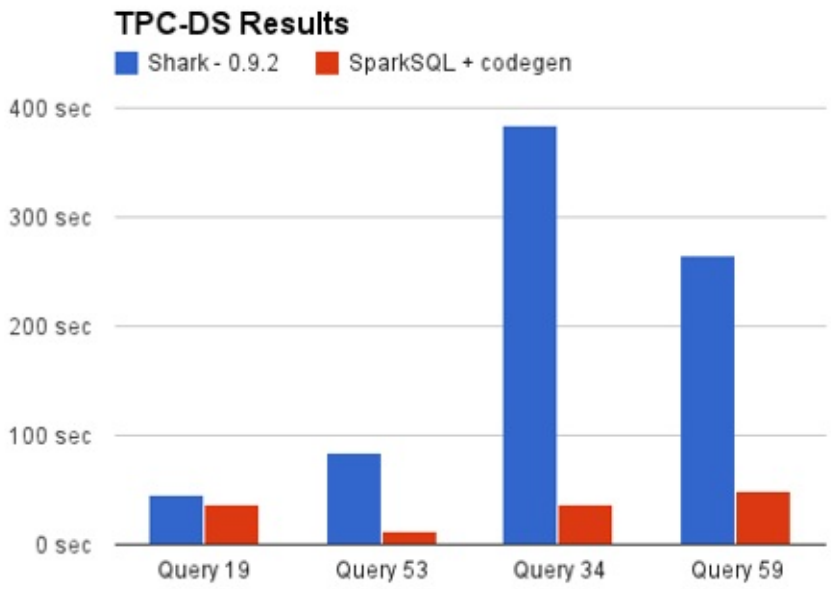
Shark -> SparkSQL

SparkSQL无论在数据兼容、性能优化、组件扩展方面都得到了极大的方便，真可谓“退一步， 海阔天空”。

1. 数据兼容方面 不但兼容hive，还可以从RDD、parquet文件、JSON文件中获取数据，未来版本甚至支持获取RDBMS数据以及cassandra等NOSQL数据
2. 性能优化方面 除了采取In-Memory Columnar Storage、byte-code generation等优化技术外、将会引进Cost Model对查询进行动态评估、获取最佳物理计划等等
3. 组件扩展方面 无论是SQL的语法解析器、分析器还是优化器都可以重新定义，进行扩展

SparkSQL的性能

摆脱了Hive限制的SparkSQL的性能在Shark的基础上又有了长足的进步。



While this is only a few queries from the TPC-DS benchmark, we plan to release more comprehensive results in the future.

SparkSQL主要在下面几点做了性能上的优化：

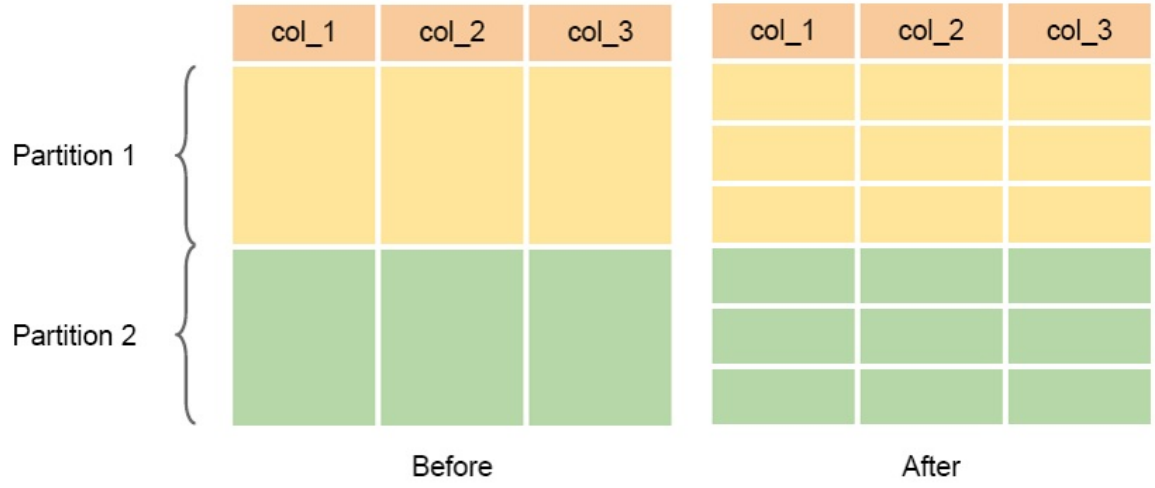
1. 内存列存储(In-Memory Columnar Storage)

SparkSQL继承了Shark的内存列存储，内存列存储有诸多好处：

1. GC友好：行存储的情况下每一行会产出一个java对象，而列存储每一列才会产生一个对象，在大数据的情况下，行存储会对GC产生巨大的压力。
2. 压缩友好：相同数据类型的数据在内存中存放在一起，有利于压缩。

3. Cache友好：分析查询中频繁使用的聚合特定列，性能会得到很大的提高，原因就是这些列的数据放在一起，更容易读入内存进行计算。

SparkSQL1.2中把每个列又分成多个batch，这样就可以避免在加载large table的时候出现OOM。



2. Code Generation

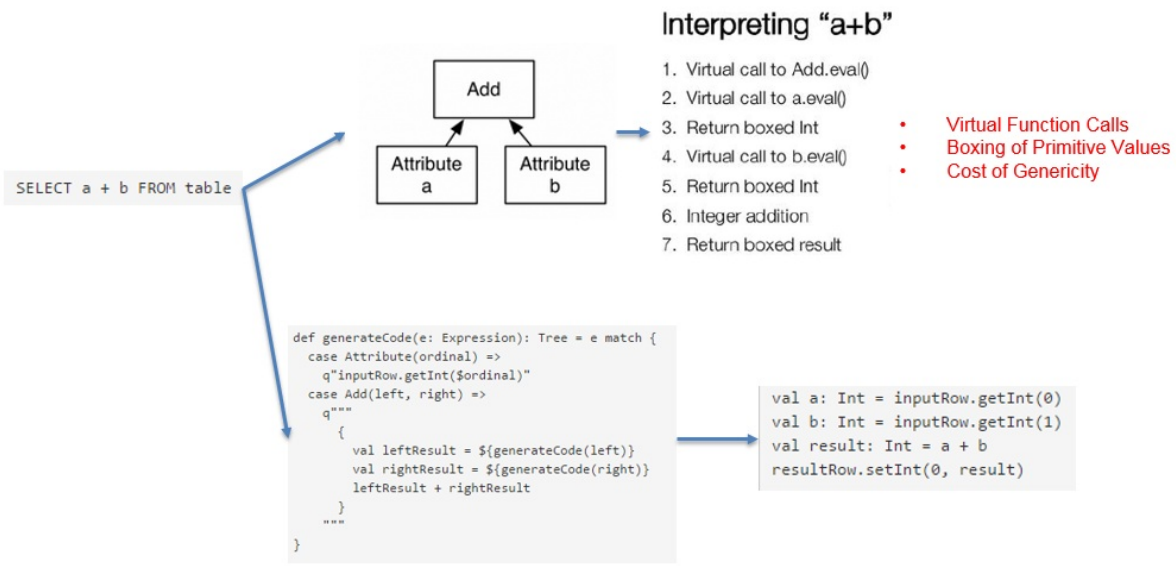
在数据库查询中有一个昂贵的操作是查询语句中的表达式，主要是由于JVM的内存模型引起的。比如如下一个查询：

```
SELECT a + b FROM table
```

在这个查询里，如果采用通用的SQL语法途径去处理，会先生成一个表达式树（有两个节点的Add树，参考后面章节），在物理处理这个表达式树的时候，将会如图所示的7个步骤：

1. 调用虚函数Add.eval()，需要确认Add两边的数据类型
2. 调用虚函数a.eval()，需要确认a的数据类型
3. 确定a的数据类型是Int，装箱
4. 调用虚函数b.eval()，需要确认b的数据类型
5. 确定b的数据类型是Int，装箱
6. 调用Int类型的Add
7. 返回装箱后的计算结果 其中多次涉及到虚函数的调用，虚函数的调用会打断CPU的正常流水线处理，减缓执行。

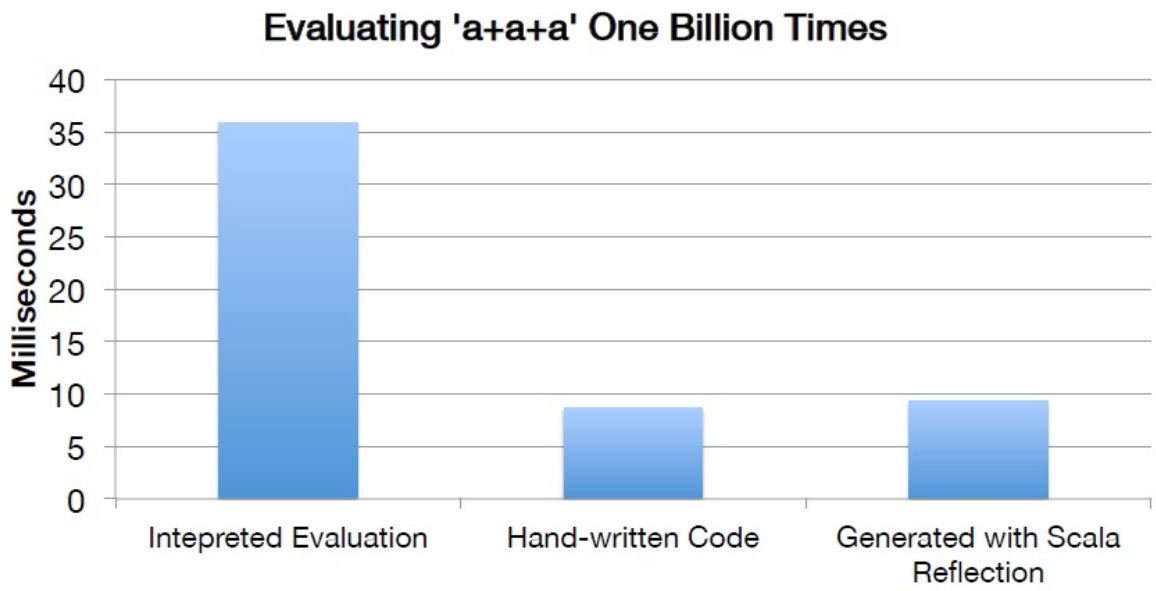
Spark1.1.0在catalyst模块的expressions增加了codegen模块，如果使用动态字节码生成技术（配置spark.sql.codegen参数），sparkSQL在执行物理计划的时候，对匹配的表达式采用特定的代码，动态编译，然后运行。



如上图中的例子，开启CG后，SparkSQL最终实现效果类似如下伪代码：

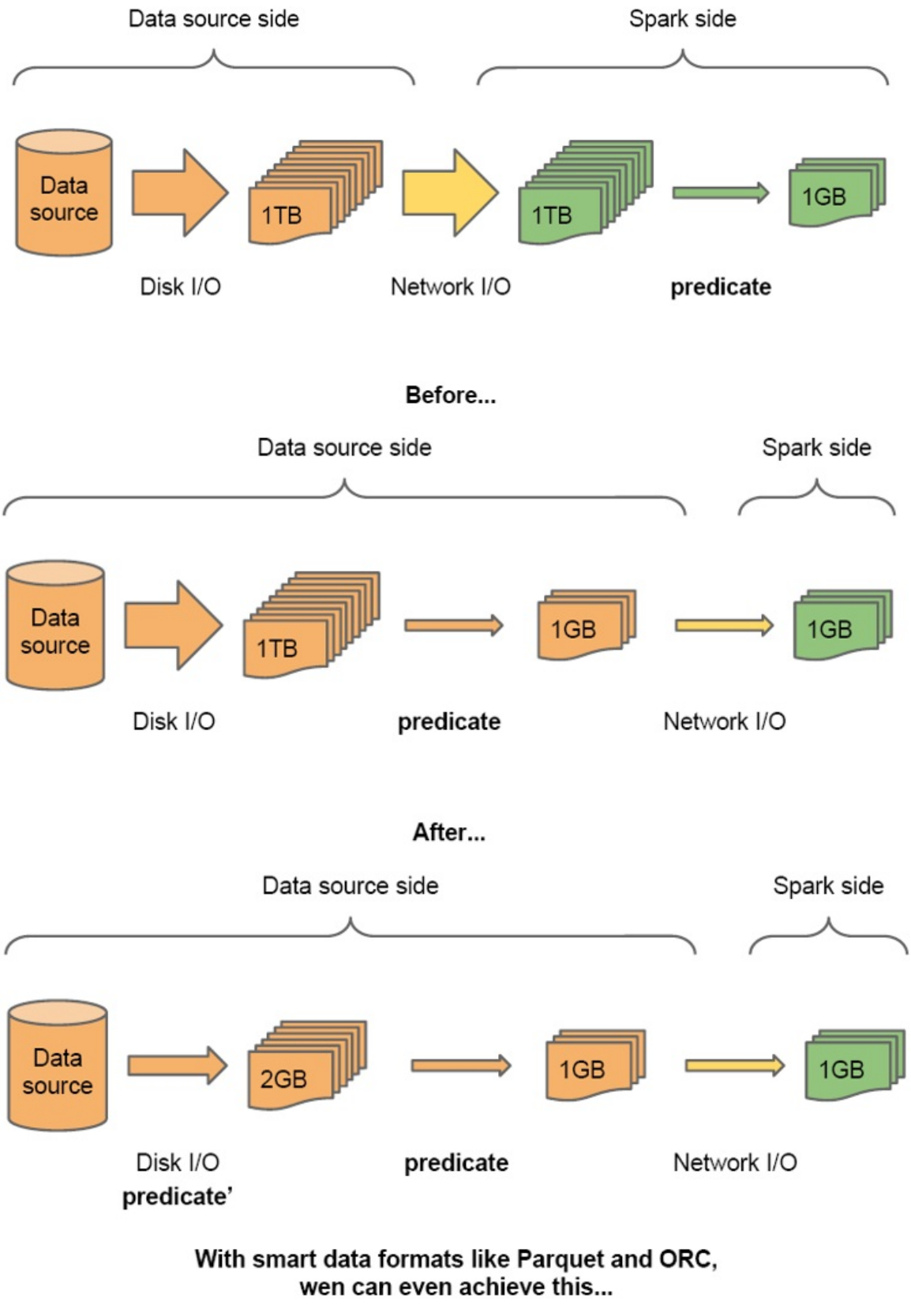
```
val a: Int = inputRow.getInt(0)
val b: Int = inputRow.getInt(1)
val result: Int = a + b
resultRow.setInt(0, result)
```

CG优化的实现主要还是依靠scala2.10的reflection和Quasiquotes。CG的性能对比如下图：

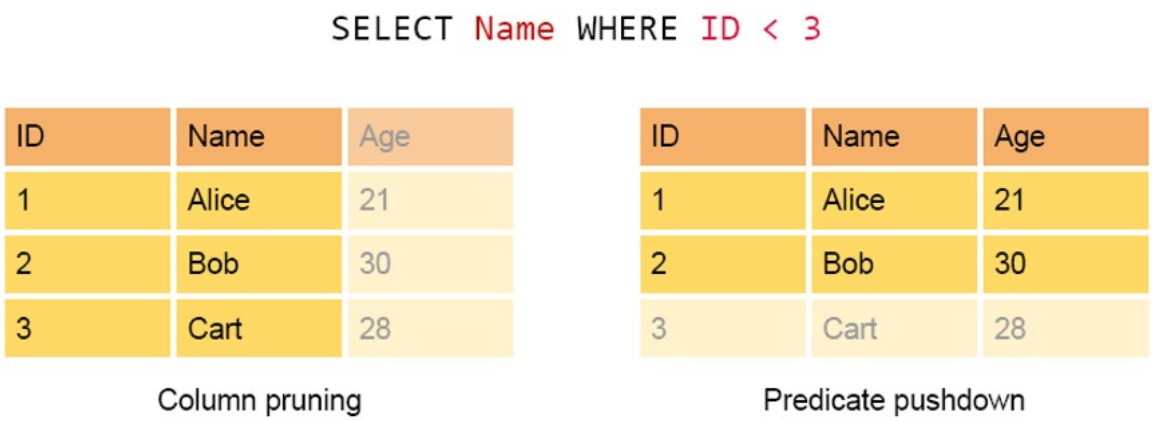


3. 外部数据源Predicate pushdown

SparkSQL1.2.0可以在读取外部数据以后马上进行filter操作，以减少网络传输的数据量；对于Parquet和ORC类型的数据，SparkSQL甚至可以在读取数据的时候就进行某些filter操作，以减少磁盘IO。

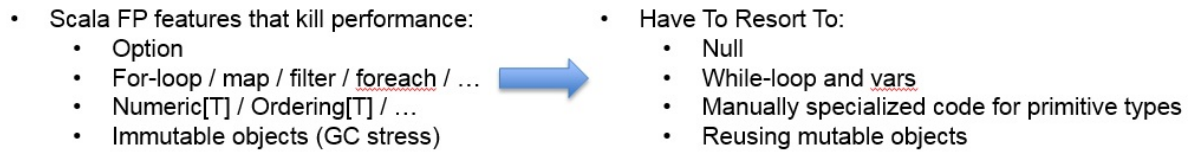


主要有两种过滤数据的方式，分别为Column Pruning和Predicte pushdown，分别是针对列和行的过滤。



4. Scala代码优化

SparkSQL在使用Scala编写代码的时候，尽量避免低效的、容易GC的代码；尽管增加了编写代码的难度，但对于用户来说，还是使用统一的接口，没受到使用上的困难。下图是一个scala代码优化的示意图：



SparkSQL的使用

SparkSQL引入了一种新的RDD——SchemaRDD，SchemaRDD由行对象（row）以及描述行对象中每列数据类型的schema组成；SchemaRDD很象传统数据库中的表。SchemaRDD可以通过RDD、Parquet文件、JSON文件、或者通过使用hiveql查询hive数据来建立。SchemaRDD除了可以和RDD一样操作外，还可以通过registerTempTable注册成临时表，然后通过SQL语句进行操作。

通过函数registerTempTable注册的表是一个临时表，生命周期只在所定义的sqlContext或hiveContext实例之中。换而言之，在一个sqlContext（或hiveContext）中registerTempTable的表不能在另一个sqlContext（或hiveContext）中使用。

另外，spark1.1开始提供了语法解析器选项spark.sql.dialect，就目前而言，spark提供了两种语法解析器：sql语法解析器和hiveql语法解析器。

1. sqlContext现在只支持sql语法解析器（SQL-92语法）
2. hiveContext现在支持sql语法解析器和hivesql语法解析器，默认为hivesql语法解析器，用户可以通过配置切换成sql语法解析器，来运行hiveql不支持的语法，如select 1。

hiveContext继承了sqlContext，所以拥有sqlContext的特性之外，还拥有自身的特性（最大的特性就是支持hive）。

编译Spark

maven编译

```
mvn -Pyarn -Dhadoop.version=2.5.0-cdh5.2.0 -Pkinesis-asl -Phive -DskipTests
```

maven编译并打包

```
./make-distribution.sh --tgz --name name -Pyarn \
-Dhadoop.version=2.5.0-cdh5.2.0 -Pkinesis-asl -Phive -DskipTests
```

本篇所用到的数据

people.json

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

people.txt

```
Michael, 29
Andy, 30
Justin, 19
```

Date.txt

Stock.txt

StockDetail.txt

创建hive表格代码

SqlContext的使用

SparkSQL提供了两种方式生成SchemaRDD:

1. 通过定义class class，使用类的反射机制生成Schema
2. 自定义Schema，应用到RDD上，生成SchemaRDD

此外SparkSQL还支持Parquet和Json等数据格式的读写以及DSL调用。

通过Case Class生成SchemaRDD

SparkSQL中可以通过定义Case Class来将普通的RDD隐式转换成SchemaRDD，从而注册为SQL临时表。

例子代码

```
case class Person(name: String, age: Int) //定义case class

val sqlContext = new SQLContext(sc)
import sqlContext._

val rddpeople = sc.textFile("./sparksql/people.txt").
    map(_.split(",")).map(p => Person(p(0), p(1).trim.toInt))

rddpeople.registerTempTable("rddTable")

rddpeople.printSchema()
sql("SELECT name FROM rddTable WHERE age >= 13 AND age <= 19").
    collect().foreach(println)
```

输出

```
root
  |-- name: string (nullable = true)
  |-- age: integer (nullable = false)
[Justin]
```

RDD隐式转为SchemaRDD的定义在SQLContext中：

```
implicit def createSchemaRDD[A <: Product](rdd: RDD[A])(
  implicit arg0: scala.reflect.api.JavaUniverse.TypeTag[A]
 ): SchemaRDD
```

同时SchemaRDDLike里面定义了registerTempTable方法，用于将RDD注册为临时表格

```
def registerTempTable(tableName: String): Unit
```

而SchemaRDD继承SchemaRDDLike

```
class SchemaRDD extends RDD[Row] with SchemaRDDLike
```

通过Apply Schema生成SchemaRDD

由于Scala中对Case Class有22列的限制，而且使用Case Class的方式创建SchemaRDD需要提前知道Schema的格式，所以SparkSQL另外提供了一种更加动态的创建SchemaRDD的方式。SQLContext定义了applySchema方法，可以把RDD+Schema转化为SchemaRDD

```
def applySchema(rowRDD: RDD[Row], schema: StructType): SchemaRDD
```

例子代码

```
val sqlContext = new SQLContext(sc)
import sqlContext._

//创建于数据结构匹配的schema
val schemaString = "name age"
val schema = StructType(schemaString.split(" ").
   map(fieldName => StructField(fieldName, StringType, true)))

//创建rowRDD
val rowRDD = sc.textFile("./sparksql/people.txt").
    map(_.split(",")).
    map(p => Row(p(0), p(1).trim))

//用applySchema将schema应用到rowRDD
val rddpeople2 = applySchema(rowRDD, schema)
rddpeople2.printSchema()
sql("SELECT name FROM rddTable2 WHERE age >= 13 AND age <= 19").
    map(t => "Name: " + t(0)).collect().foreach(println)
```

输出

```
root
 |-- name: string (nullable = true)
 |-- age: string (nullable = true)
Name: Justin
```

通过Parquet文件生成SchemaRDD

SQLContext提供了parquetFile和saveAsParquetFile方法用来读写parquet格式的数据

parquet是一种基于列式存储的数据存储格式

例子代码

```
val sqlContext = new SQLContext(sc)
import sqlContext._

rddpeople2.saveAsParquetFile("./sparksql/people.parquet")
val parquetpeople = sqlContext.parquetFile("./sparksql/people.parquet")
parquetpeople.registerTempTable("parquetTable")

parquetpeople.printSchema()
sqlContext.sql("SELECT name FROM parquetTable WHERE age >= 25").
   map(t => "Name: " + t(0)).collect().foreach(println)
```

输出

```
root
  |-- name: string (nullable = true)
  |-- age: string (nullable = true)
Name: Michael
Name: Andy
```

通过Json文件生成SchemaRDD

SQLContext定义了jsonFile方法，用来读取json格式的数据

```
def jsonFile(path: String): SchemaRDD
```

例子代码

```
val sqlContext = new SQLContext(sc)
import sqlContext._

val jsonpeople = jsonFile("./sparksql/people.json")
jsonpeople.registerTempTable("jsonTable")

jsonpeople.printSchema()
sql("SELECT name FROM jsonTable WHERE age >= 25").
   map(t => "Name: " + t(0)).collect().foreach(println)
```

输出

```
root
  |-- age: integer (nullable = true)
  |-- name: string (nullable = true)
    Name: Andy
```

SQL Queries & Language Integrated Queries

有两种方式可以调用sql查询

方法一： 通过调用SQLContext提供的sql函数

```
def sql(sqlText: String): SchemaRDD
```

例子代码

```
val sqlContext = new SQLContext(sc)
import sqlContext._
sql("SELECT name FROM jsonTable WHERE age >= 25").
   map(t => "Name: " + t(0)).collect().foreach(println)
```

方法二： 通过SchemaRDD提供的Language Integrated Queries

```
def where(dynamicUdf: (DynamicRow) -> Boolean): SchemaRDD
def select(exprs: Expression*): SchemaRDD
def orderBy(sortExprs: SortOrder*): SchemaRDD
//etc
```

例子代码

```
rddpeople.
      where('age >= 13).
      where('age <= 19).
      select('name).
      map(t => "Name: " + t(0)).
      collect().foreach(println)
```

HiveContext的使用

使用hiveContext之前首先确认两点：

1. 使用的Spark是支持hive的，可以通过查看lib目录下有没有datanucleus-api-jdo-*.jar,datanucleus-core-*.jar,datanucleus-rdbms-*.jar这三个文件
2. hive的配置文件hive-site.xml已经复制到spark的conf目录中

使用HiveContext

使用HiveContext首先要构建HiveContext

```
import org.apache.spark.sql.hive.HiveContext
val hiveContext = new HiveContext(sc)
```

然后就可以对hive数据进行操作了，下面我们将使用hive数据仓库里面的数据，首先切换数据库到spark_test，并查看里面的表格

```
hiveContext.sql("use spark_test")
hiveContext.sql("show tables").collect().foreach(println)
```

结果是

```
[tbldate]
[tblstock]
[tblstockdetail]
```

然后查询一下所有订单中每年的销售单数、销售总额：

```
/* 所有订单中每年的销售单数、销售总额
三个表连接后以count(distinct a.ordernumber)
计销售单数，sum(b.amount)计销售总额 */

hiveContext.sql("""select c.theyear,count(distinct a.ordernumber), sum(b.amount)
    from tblStock a
    join tblStockDetail b on a.ordernumber=b.ordernumber
    join tbldate c on a.dateid=c.dateid
    group by c.theyear order by c.theyear""").
    collect().foreach(println)
```

结果是

```
[2004,1094,3265696]
[2005,3828,13247234]
```

```
[2006,3772,13670416]
[2007,4885,16711974]
[2008,4861,14670698]
[2009,2619,6322137]
[2010,94,210924]
```

SparkSQL的三种使用方式

本篇介绍SparkSQL的三种使用方式

1. Spark Shell
2. CLI
3. Thrift Server

通过spark-shell调用SparkSQL

在启动spark shell时添加mysql-jdbc-driver.jar的依赖，然后import HiveContext，就可以正常使用SparkSQL了

运行spark shell

```
export SPARK_CLASSPATH=/usr/lib/hadoop/lib/hadoop-lzo.jar:\
/usr/lib/hive/lib/mysql-jdbc-driver.jar
./bin/spark-shell --master local
```

```
import org.apache.spark.sql.hive.HiveContext
val hiveContext = new HiveContext(sc)
hiveContext.sql("use spark_test")
hiveContext.sql("show tables").collect().foreach(println)
```

输出

```
[tbldate]
[tblstock]
[tblstockdetail]
```

通过CLI调用SparkSQL

SparkSQL提供了类似于sql命令行的CLI接口，用户可以直接使用sql语言。

运行spark sql

```
export SPARK_CLASSPATH=/usr/lib/hadoop/lib/hadoop-lzo.jar:\
/usr/lib/hive/lib/mysql-jdbc-driver.jar
./bin/spark-sql --master local
```

```
use spark_test;
show tables;
```

输出

```
[tbldate]
[tblstock]
[tblstockdetail]
```

通过ThriftServer调用SparkSQL

开启thrift server

```
export SPARK_CLASSPATH=/usr/lib/hadoop/lib/hadoop-lzo.jar:\
/usr/lib/hive/lib/mysql-jdbc-driver.jar
./sbin/start-thriftserver.sh \
--hiveconf hive.server2.thrift.port=10000 \
--hiveconf hive.server2.thrift.bind.host=localhost \
--master local
```

开启beeline client

```
bash /data/git/spark/com.iqiyi/spark-1.1.0-2.5.0-cdh5.2.0-qiyi/bin/beeline

!connect jdbc:hive2://localhost:10000

use spark_test;
show tables;
```

输出

```
+-----------------+
|     result      |
+-----------------+
| tbldate         |
| tblstock        |
| tblstockdetail  |
+-----------------+
```

常用操作

下面介绍一下hive/console的常用操作，主要是和运行计划相关的常用操作。在操作前，首先定义一个表people和查询query。在控制台逐行运行

```
import org.apache.spark.sql.hive.HiveContext

case class Person(name:String, age:Int, state:String)

val hiveContext = new HiveContext(sc)
import hiveContext._

sc.parallelize(
Person("Michael",29,"CA")::
Person("Andy",30,"NY")::
Person("Justin",19,"CA")::
Person("Justin",25,"CA")::Nil).
registerTempTable("people")

val query= sql("select * from people")
```

查看查询的schema

```
query.printSchema
```

输出

```
root
 |-- name: string (nullable = true)
 |-- age: integer (nullable = false)
 |-- state: string (nullable = true)
```

查看查询的整个运行计划

```
query.queryExecution
```

输出

```
== Parsed Logical Plan ==
'Project [*]
 'UnresolvedRelation None, people, None

== Analyzed Logical Plan ==
```

```
Project [name#0,age#1,state#2]
 LogicalRDD [name#0,age#1,state#2], MapPartitionsRDD[1] at mapPartitions at Existin
gRDD.scala:36

== Optimized Logical Plan ==
LogicalRDD [name#0,age#1,state#2], MapPartitionsRDD[1] at mapPartitions at Existing
RDD.scala:36

== Physical Plan ==
PhysicalRDD [name#0,age#1,state#2], MapPartitionsRDD[1] at mapPartitions at Existin
gRDD.scala:36

Code Generation: false
== RDD ==
```

查看查询的Unresolved LogicalPlan

```
query.queryExecution.logical
```

输出

```
'Project [*]
 'UnresolvedRelation None, people, None
```

查看查询的analyzed LogicalPlan

```
query.queryExecution.analyzed
```

输出

```
Project [name#0,age#1,state#2]
 LogicalRDD [name#0,age#1,state#2], MapPartitionsRDD[1] at mapPartitions at Existin
gRDD.scala:36
```

查看优化后的LogicalPlan

```
query.queryExecution.optimizedPlan
```

输出

```
LogicalRDD [name#0,age#1,state#2], MapPartitionsRDD[1] at mapPartitions at Existing
RDD.scala:36
```

查看物理计划

```
query.queryExecution.sparkPlan
```

输出

```
PhysicalRDD [name#0,age#1,state#2], MapPartitionsRDD[1] at mapPartitions at Existin
gRDD.scala:36
```

查看RDD的转换过程

```
query.toDebugString
```

输出

```
== Query Plan ==
== Physical Plan ==
PhysicalRDD [name#0,age#1,state#2], MapPartitionsRDD[1] at mapPartitions at Existin
gRDD.scala:36 []
 |  MapPartitionsRDD[1] at mapPartitions at ExistingRDD.scala:36 []
 |  ParallelCollectionRDD[0] at parallelize at <console>:21 []
```

Cache Table

SparkSQL的cache可以使用两种方法来实现：

1. cacheTable()方法
2. CACHE TABLE命令

SparkSQL的cache与RDD的cache有下面几点不同：

1. SparkSQL的cache采用列存储
2. SparkSQL的cache有两种模式可以选择lazy和Eager，而RDD的cache是lazy的

值得注意的是，SparkSQL 1.1.0中，SQL Cache是lazy模式的，而在1.2.0中，SQL Cache默认是eager模式。

Cache Table的几种用法

1. API调用

通过SqlContext的cacheTable函数调用Cache，该函数是eager类型调用

```
cacheTable("rddtable")
```

2. sql调用

通过SqlContext的sql函数调用Cache，如果没有加lazy关键字，默认是eager类型调用

```
sql("cache table rddtable")
```

3. lazy调用

通过SqlContext的sql函数调用Cache，如果添加lazy关键字，只有在触发action的时候才会去cache

```
sql("cache lazy table rddtable")
```

4. SchemaRDD.cache

调用SchemaRDD的cache同样会触发Cache操作，该函数也是eager类型调用

```
createSchemaRDD(rddpeople).cache()
```

错误的方法

如果调用RDD的cache方法，不会触发SparkSQL的cache，而只会触发普通RDD的cache操作，此外该函数是lazy类型的

```
rddpeople.cache()
```

外部数据源

随着Spark1.2的发布，Spark SQL开始正式支持外部数据源。Spark SQL开放了一系列接入外部数据源的接口，来让开发者可以实现。这使得Spark SQL支持了更多的类型数据源，如json, parquet, avro, csv格式。只要我们愿意，我们可以开发出任意的外部数据源来连接到Spark SQL。HBASE, Cassandra都可以用外部数据源的方式来实现无缝集成。

API方式创建外部数据源表

在Spark1.2之前就已经支持了api方式穿件外部数据源表，支持

1. json
2. parquet
3. hive meta store

```
val sqlContext = new SQLContext(sc)
import sqlContext._

val peopleParquet = parquetFile("./sparksql/people.parquet")

peopleParquet.registerTempTable("parquetTable")

sql("SELECT name FROM parquetTable WHERE age >= 25").
  map(t => "Name: " + t(0)).collect().foreach(println)
```

DDL创建外部数据源表

在Spark1.2之后，支持了一种CREATE TEMPORARY TABLE USING OPTIONS的DDL语法来创建外部数据源的表。Spark自带实现了

1. org.apache.spark.sql.parquet
2. org.apache.spark.sql.json

未来准备把hive meta store也整合到该框架中。

```
val sqlContext = new SQLContext(sc)
import sqlContext._

val peopleParquet = sql(
  s"""
    |CREATE TEMPORARY TABLE parquetTable
    |USING org.apache.spark.sql.parquet
    |OPTIONS (
    |path './sparksql/people.parquet'
    |)""".stripMargin)
```

```
println(peopleParquet.toDebugString)

sql("SELECT name FROM parquetTable WHERE age >= 25").
  map(t => "Name: " + t(0)).collect().foreach(println)
```

自定义外部数据源格式

使用DDL方式创建外部数据源表，使得自定义数据格式成为可能，用户可以自己实现读取外部表的代码，动态插入SparkSQL中，就可以实现自定义外部数据源格式。

可以参考Databricks在github上提交了读取avro格式的自定义数据源，spark-avro。

SparkSQL调优

并行性

SparkSQL在集群中运行，将一个查询任务分解成大量的Task分配给集群中的各个节点来运行。通常情况下，Task的数量是大于集群的并行度。shuffle的时候使用了缺省的spark.sql.shuffle.partitions，即200个partition，也就是200个Task。

而如果实验的集群环境却只能并行3个Task，也就是说同时只能有3个Task保持Running。

这时大家就应该明白了，要跑完这200个Task就要跑200/3=67批次。如何减少运行的批次呢？那就要尽量提高查询任务的并行度。查询任务的并行度由两方面决定：集群的处理能力和集群的有效处理能力。

对于Spark Standalone集群来说，集群的处理能力是由conf/spark-env中的SPARK_WORKER_INSTANCES参数、SPARK_WORKER_CORES参数决定的；而SPARK_WORKER_INSTANCES*SPARK_WORKER_CORES不能超过物理机器的实际CPU core；

集群的有效处理能力是指集群中空闲的集群资源，一般是指使用spark-submit或spark-shell时指定的--total-executor-cores，一般情况下，我们不需要指定，这时候，Spark Standalone集群会将所有空闲的core分配给查询，并且在Task轮询运行过程中，Standalone集群会将其他spark应用程序运行完后空闲出来的core也分配给正在运行中的查询。

综上所述，sparkSQL的查询并行度主要和集群的core数量相关，合理配置每个节点的core可以提高集群的并行度，提高查询的效率。

高效的数据格式

高效的数据格式，一方面是加快了数据的读入速度，另一方面可以减少内存的消耗。高效的数据格式包括多个方面。

数据本地性

分布式计算系统的精粹在于移动计算而非移动数据，但是在实际的计算过程中，总存在着移动数据的情况，除非是在集群的所有节点上都保存数据的副本。移动数据，将数据从一个节点移动到另一个节点进行计算，不但消耗了网络IO，也消耗了磁盘IO，降低了整个计算的效率。为了提高数据的本地性，除了优化算法（也就是修改spark内存，难度有点高），就是合理设置数据的副本。设置数据的副本，这需要通过配置参数并长期观察运行状态才能获取的一个经验值。

下面是spark webUI监控Stage的一个图：

26	60	0	SUCCESS	PROCESS_LOCAL	hadoop3	2014/09/21 20:52:44	0.2 s			164.0 MB (memory)	52.0 B	
30	61	0	SUCCESS	PROCESS_LOCAL	hadoop1	2014/09/21 20:52:44	0.2 s			164.5 MB (memory)	52.0 B	
27	62	0	SUCCESS	PROCESS_LOCAL	hadoop3	2014/09/21 20:52:44	0.2 s			164.3 MB (memory)	52.0 B	
33	63	0	SUCCESS	PROCESS_LOCAL	hadoop1	2014/09/21 20:52:45	0.2 s			164.6 MB (memory)	52.0 B	
29	64	0	SUCCESS	NODE_LOCAL	hadoop3	2014/09/21 20:52:48	13 s	36 ms		128.0 MB (hadoop)	52.0 B	
23	65	0	SUCCESS	NODE_LOCAL	hadoop2	2014/09/21 20:52:48	10 s	45 ms		128.0 MB (hadoop)	52.0 B	
24	66	0	SUCCESS	ANY	hadoop1	2014/09/21 20:52:52	14 s	44 ms		128.0 MB (hadoop)	52.0 B	
25	67	0	SUCCESS	NODE_LOCAL	hadoop2	2014/09/21 20:52:58	8 s	41 ms		128.0 MB (hadoop)	52.0 B	
32	68	0	SUCCESS	NODE_LOCAL	hadoop3	2014/09/21 20:53:01	13 s	56 ms		128.0 MB (hadoop)	52.0 B	
28	69	0	SUCCESS	NODE_LOCAL	hadoop2	2014/09/21 20:53:06	13 s	40 ms		128.0 MB (hadoop)	52.0 B	
31	70	0	SUCCESS	ANY	hadoop1	2014/09/21 20:53:10	15 s	37 ms		128.0 MB (hadoop)	52.0 B	
34	71	0	SUCCESS	ANY	hadoop3	2014/09/21 20:53:14	13 s	53 ms		137.1 MB (hadoop)	52.0 B	

- PROCESS_LOCAL是指读取缓存在本地节点的数据
- NODE_LOCAL是指读取本地节点硬盘数据
- ANY是指读取非本地节点数据
- 通常读取数据PROCESS_LOCAL>NODE_LOCAL>ANY，尽量使数据以PROCESS_LOCAL或NODE_LOCAL方式读取。其中PROCESS_LOCAL还和cache有关。

合适的数据类型

对于要查询的数据，定义合适的数据类型也是非常有必要。对于一个tinyint可以使用的数据列，不需要为了方便定义成int类型，一个tinyint的数据占用了1个byte，而int占用了4个byte。也就是说，一旦将这数据进行缓存的话，内存的消耗将增加数倍。在SparkSQL里，定义合适的数据类型可以节省有限的内存资源。

合适的数据列

对于要查询的数据，在写SQL语句的时候，尽量写出要查询的列名，如Select a,b from tbl，而不是使用Select * from tbl；这样不但可以减少磁盘IO，也减少缓存时消耗的内存。

更优的数据存储格式

在查询的时候，最终还是要读取存储在文件系统中的文件。采用更优的数据存储格式，将有利于数据的读取速度。查看sparkSQL的stage，可以发现，很多时候，数据读取消耗占有很大的比重。对于sqlContext来说，支持 textFiile、SequenceFile、ParquetFile、jsonFile；对于hiveContext来说，支持AvroFile、ORCFile、Parquet File，以及各种压缩。根据自己的业务需求，测试并选择合适的数据存储格式将有利于提高sparkSQL的查询效率。

内存的使用

spark应用程序最纠结的地方就是内存的使用了，也是最能体现“细节是魔鬼”的地方。Spark的内存配置项有不少，其中比较重要的几个是：

- SPARK_WORKER_MEMORY，在conf/spark-env.sh中配置SPARK_WORKER_MEMORY 和SPARK_WORKER_INSTANCES，可以充分的利用节点的内存资源，SPARK_WORKER_INSTANCES*SPARK_WORKER_MEMORY不要超过节点本身具备的内存容量；
- executor-memory，在spark-shell或spark-submit提交spark应用程序时申请使用的内存数量；不要超过节点的SPARK_WORKER_MEMORY；
- spark.storage.memoryFraction spark应用程序在所申请的内存资源中可用于cache的比例

- spark.shuffle.memoryFraction spark应用程序在所申请的内存资源中可用于shuffle的比例 在实际使用上，对于后两个参数，可以根据常用查询的内存消耗情况做适当的变更。另外，在SparkSQL使用上，有几点建议：
- 对于频繁使用的表或查询才进行缓存，对于只使用一次的表不需要缓存；
- 对于join操作，优先缓存较小的表；
- 要多注意Stage的监控，多思考如何才能更多的Task使用PROCESS_LOCAL；
- 要多注意Storage的监控，多思考如何才能Fraction cached的比例更多

合适的Task

对于SparkSQL，还有一个比较重要的参数，就是shuffle时候的Task数量，通过spark.sql.shuffle.partitions来调节。调节的基础是spark集群的处理能力和要处理的数据量，spark的默认值是200。Task过多，会产生很多的任务启动开销，Task多少，每个Task的处理时间过长，容易straggle。

其他的一些建议

优化的方面的内容很多，但大部分都是细节性的内容，下面就简单地提提：

- 想要获取更好的表达式查询速度，可以将spark.sql.codegen设置为Ture；
- 对于大数据集的计算结果，不要使用collect() ,collect()就结果返回给driver，很容易撑爆driver的内存；一般直接输出到分布式文件系统中；
- 对于Worker倾斜，设置spark.speculation=true 将持续不给力的节点去掉；
- 对于数据倾斜，采用加入部分中间步骤，如聚合后cache，具体情况具体分析；
- 适当的使用序化方案以及压缩方案；
- 善于利用集群监控系统，将集群的运行状况维持在一个合理的、平稳的状态；
- 善于解决重点矛盾，多观察Stage中的Task，查看最耗时的Task，查找原因并改善；

SparkSQL的运行过程

SqlContext的运行过程

SparkSQL有两个分支SqlContext和Hivecontext，SqlContext现在只支持sql语法解析器（SQL-92语法），而HiveContext现在既支持sql语法解析器又支持hivesql语法解析器，默认为hivesql语法解析器，用户可以通过配置切换成sql语法解析器，来运行hiveql不支持的语法。

SqlContext使用sqlContext.sql(sqlText)来提交用户sql语句，SqlContext首先会调用parserSql对sqlText进行语法分析，然后返回给用户SchemaRDD。SchemaRDD继承自SchemaRDDLike。

```
  /**
   * Executes a SQL query using Spark, returning the result as a SchemaRDD.  The di
alect that is
   * used for SQL parsing can be configured with 'spark.sql.dialect'.
   *
   * @group userf
   */
  def sql(sqlText: String): SchemaRDD = {
    if (dialect == "sql") {
      new SchemaRDD(this, parseSql(sqlText))
    } else {
      sys.error(s"Unsupported SQL dialect: $dialect")
    }
  }

protected[sql] val sqlParser = {
    val fallback = new catalyst.SqlParser
    new catalyst.SparkSQLParser(fallback(_))
  }

class SchemaRDD(
    @transient val sqlContext: SQLContext,
    @transient val baseLogicalPlan: LogicalPlan)
  extends RDD[Row](sqlContext.sparkContext, Nil) with SchemaRDDLike
```

parseSql首先会尝试dll语法解析，如果失败则进行sql语法解析。

```
   protected[sql] def parseSql(sql: String): LogicalPlan = {
    ddlParser(sql).getOrElse(sqlParser(sql))
  }
```

然后调用SchemaRDDLike中的sqlContext.executePlan(baseLogicalPlan)来执行catalyst.SqlParser解析后生成的Unresolved LogicalPlan。

```
private[sql] trait SchemaRDDLike {
  @transient def sqlContext: SQLContext
  @transient val baseLogicalPlan: LogicalPlan
```

```
  private[sql] def baseSchemaRDD: SchemaRDD
  ...
  lazy val queryExecution = sqlContext.executePlan(baseLogicalPlan)
  ...
  }
```

接着executePlan会调用QueryExecution

```
protected[sql] def executePlan(plan: LogicalPlan): this.QueryExecution =
    new this.QueryExecution { val logical = plan }
```

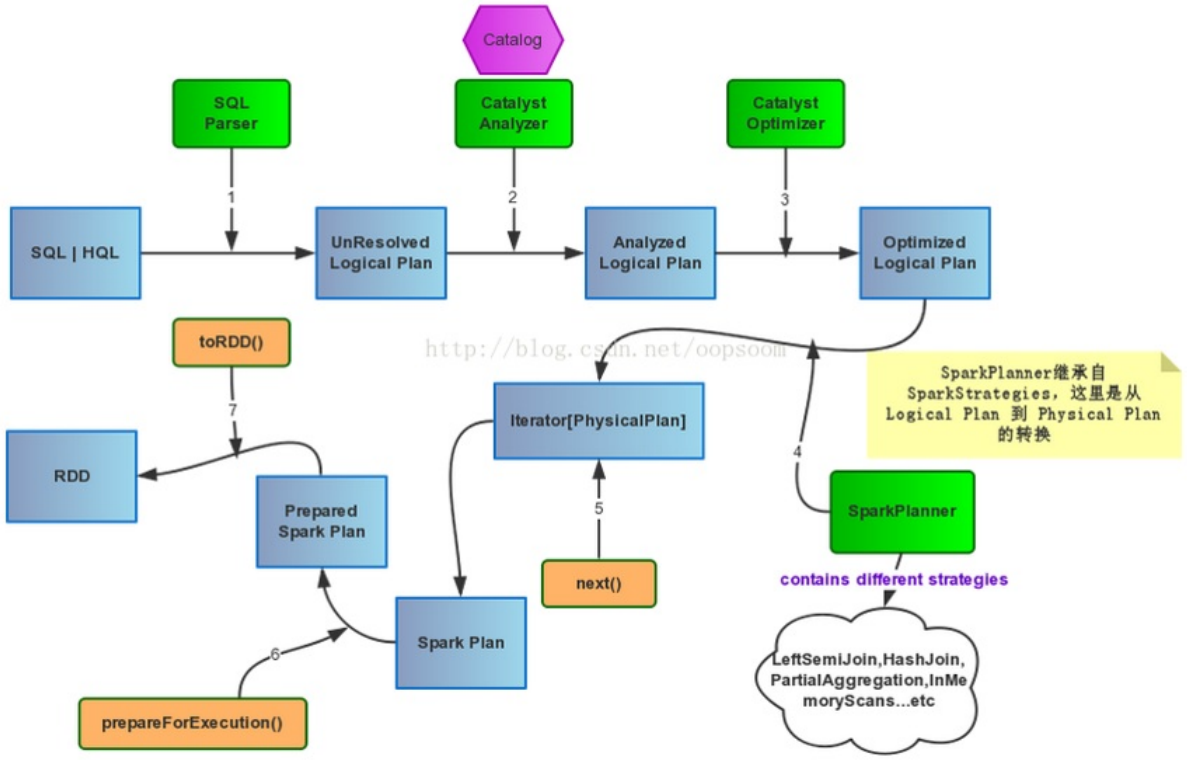
先看看QueryExecution的代码

```
/**
   * :: DeveloperApi ::
   * The primary workflow for executing relational queries using Spark.  Designed t
o allow easy
   * access to the intermediate phases of query execution for developers.
   */
  @DeveloperApi
  protected abstract class QueryExecution {
    def logical: LogicalPlan

    lazy val analyzed = ExtractPythonUdfs(analyzer(logical))
    lazy val withCachedData = useCachedData(analyzed)
    lazy val optimizedPlan = optimizer(withCachedData)

    // TODO: Don't just pick the first one...
    lazy val sparkPlan = {
      SparkPlan.currentContext.set(self)
      planner(optimizedPlan).next()
    }
    // executedPlan should not be used to initialize any SparkPlan. It should be
    // only used for execution.
    lazy val executedPlan: SparkPlan = prepareForExecution(sparkPlan)

    /** Internal version of the RDD. Avoids copies and has no schema */
    lazy val toRdd: RDD[Row] = executedPlan.execute()
    ...
    }
```

QueryExecution的执行如下

1. 使用analyzer结合数据数据字典（catalog）进行绑定，生成resolved LogicalPlan
2. 处理UDF
3. 处理Cache
4. 使用optimizer对resolved LogicalPlan进行优化，生成optimized LogicalPlan

5. 使用SparkPlan将LogicalPlan转换成PhysicalPlan
6. 使用prepareForExecution()将PhysicalPlan转换成可执行物理计划
7. 使用execute()执行可执行物理计划
8. 生成SchemaRDD



HiveContext的运行过程

由于历史原因，实际应用中很多数据已经定义了hive meta data，使用SparkSQL的HiveContext可以无缝访问这些数据。使用HiveContext前只需要把hive-site.xml复制到spark/conf/中。

HiveContext继承自SQLContext，在hiveContext的的运行过程中除了override的函数和变量，可以使用和sqlContext一样的函数和变量。

```
/**
 * An instance of the Spark SQL execution engine that integrates with data stored i
n Hive.
 * Configuration for Hive is read from hive-site.xml on the classpath.
 */
class HiveContext(sc: SparkContext) extends SQLContext(sc) {
...
override def sql(sqlText: String): SchemaRDD = {
    // TODO: Create a framework for registering parsers instead of just hardcoding
if statements.
    if (dialect == "sql") {
      super.sql(sqlText)
    } else if (dialect == "hiveql") {
      new SchemaRDD(this, ddlParser(sqlText).getOrElse(HiveQl.parseSql(sqlText)))
    }  else {
      sys.error(s"Unsupported SQL dialect: $dialect.  Try 'sql' or 'hiveql'")
    }
  }
...
}
```

HiveContext使用HiveContext.sql(sqlText)来提交用户查询。hiveContext.sql首先根据用户的语法设置（spark.sql.dialect）决定具体的执行过程，如果dialect == "sql"则采用sqlContext的sql语法执行过程；如果是dialect == "hiveql"，则采用hiveql语法执行过程。在这里我们主要看看hiveql语法执行过程。

首先会尝试利用DDLParser进行ddl语法解析，如果失败的话，则进行HiveQl进行sql语法解析，并返回SchemaRDD。

```
 /** Returns a LogicalPlan for a given HiveQL string. */
  def parseSql(sql: String): LogicalPlan = hqlParser(sql)

  protected val hqlParser = {
    val fallback = new ExtendedHiveQlParser
    new SparkSQLParser(fallback(_))
  }

private[sql] class SparkSQLParser(fallback: String => LogicalPlan) extends Abstract
SparkSQLParser {
...
```

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

因为sparkSQL所支持的hiveql除了兼容hive语句外，还兼容一些sparkSQL本身的语句，所以在hiveql语句语法解析的时候：

1. 首先调用专门解析SparkSQL语法的解析器SparkSQLParser
2. 当SparksQLParser无法解析的时候，会调用ExtendedHiveQlParser进行解析

hiveContext的运行过程基本和sqlContext一致，除了override的catalog、functionRegistry、analyzer、planner、optimizedPlan、toRdd。 hiveContext的catalog，是指向 Hive Metastore。hiveContext的analyzer，使用了新的catalog和functionRegistry。hiveContext的planner，使用新定义的hivePlanner。

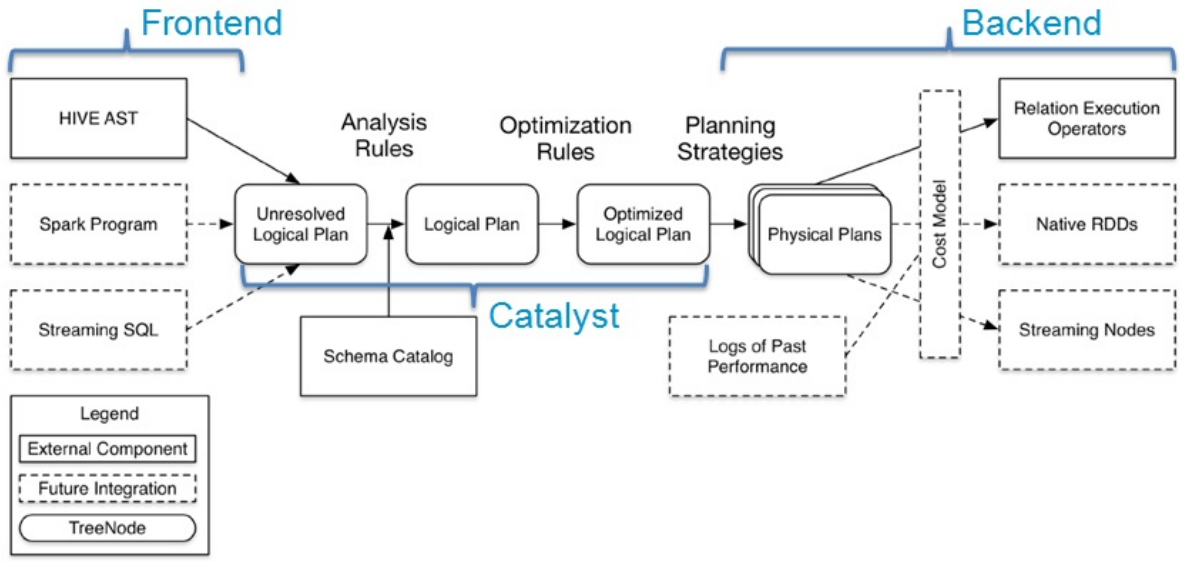
```
/* A catalyst metadata catalog that points to the Hive Metastore. */
  @transient
  override protected[sql] lazy val catalog = new HiveMetastoreCatalog(this) with Ov
errideCatalog

  // Note that HiveUDFs will be overridden by functions registered in this context.
  @transient
  override protected[sql] lazy val functionRegistry =
    new HiveFunctionRegistry with OverrideFunctionRegistry

  /* An analyzer that uses the Hive metastore. */
  @transient
  override protected[sql] lazy val analyzer =
    new Analyzer(catalog, functionRegistry, caseSensitive = false) {
      override val extendedRules =
        catalog.CreateTables ::
        catalog.PreInsertionCasts ::
        ExtractPythonUdfs ::
        Nil
    }

    @transient
  override protected[sql] val planner = hivePlanner
```

Spark1.2.0中HiveContext的执行基本上和SqlContext保持了统一。

Catalyst优化器

在传统关系型数据库当中，一个简单的sql语句将会依次经过Parser生成Logical Pla，Optimizer生成Optimized Logical Plan，最后生成Physical Plan，交给执行器去执行。SparkSQL也采用了类似的方式进行处理，下面介绍一下SparkSQL的优化器：**Catalyst**。

先来看一下Catalyst在整个sql执行流程中所处的位置：



图中虚线部分是以后版本要实现的功能，实线部分是已经实现的功能。从上图看，整个SQL的执行框架主要的实现组件有：

1. SqlParse：sql语句的语法解析功能
2. Analyzer：将Unresolved Logical Plan和Schema Catalog进行绑定，生成Resolved Logical Plan
3. Optimizer：对Resolved Logical Plan进行优化，生成Optimized Logical Plan
4. Planner：将Logical Plan转换成Physical Plan
5. CostModel：根据过去的性能统计数据，选择最佳的物理执行计划

Catalyst在整个流程中处于中段位置，它主要有两个任务

1. 将Unresolved Logical Plan和Schema Catalog进行绑定，生成Resolved Logical Plan
2. 对Resolved Logical Plan进行优化，生成Optimized Logical Plan

这两个任务正好说明了Catalyst名字的由来，即Catalog + Analyst。

（本文基于spark1.2.0）

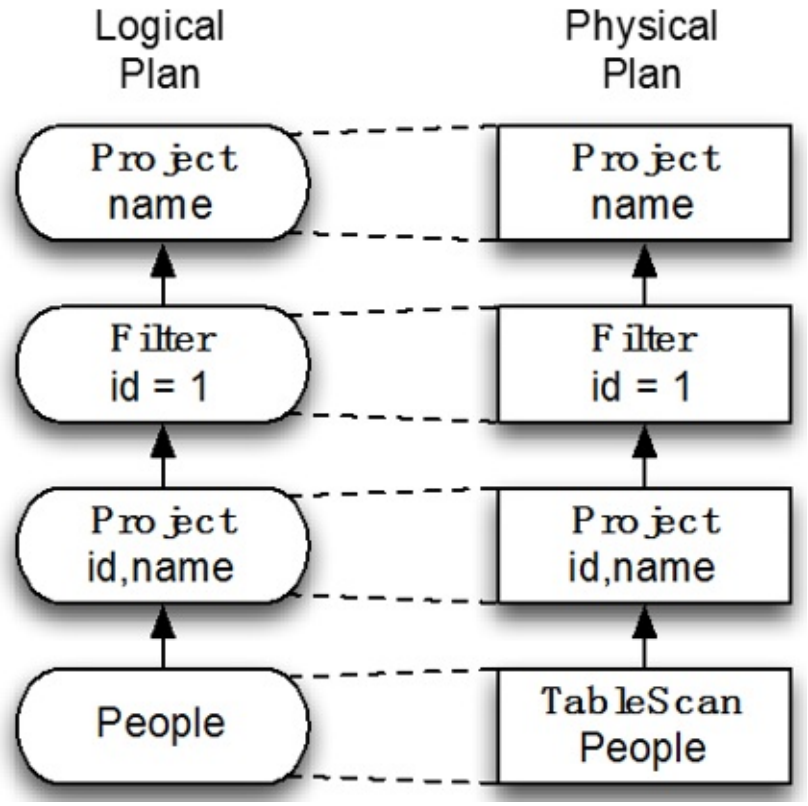
Catalyst介绍

Catalyst是一款基于规则的优化器，Analyzer和Optimizer定义了一系列优化规则，Catalyst根据这些规则对执Logical Plan进行优化。

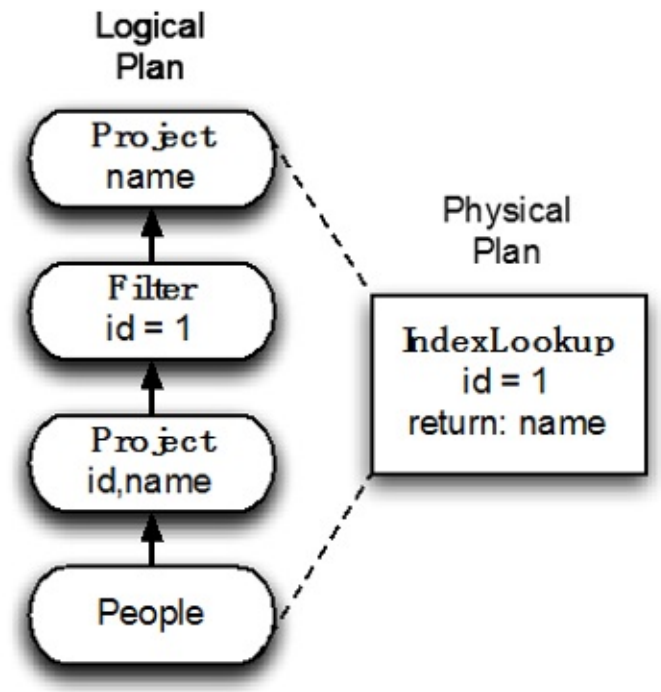
对于下面这个sql查询

```
select name from(
    select id, name from people
    ) p
    where p.id = 1
```

最简单就是不做优化，直接一对一将Logical Plan映射为Physical Plan。

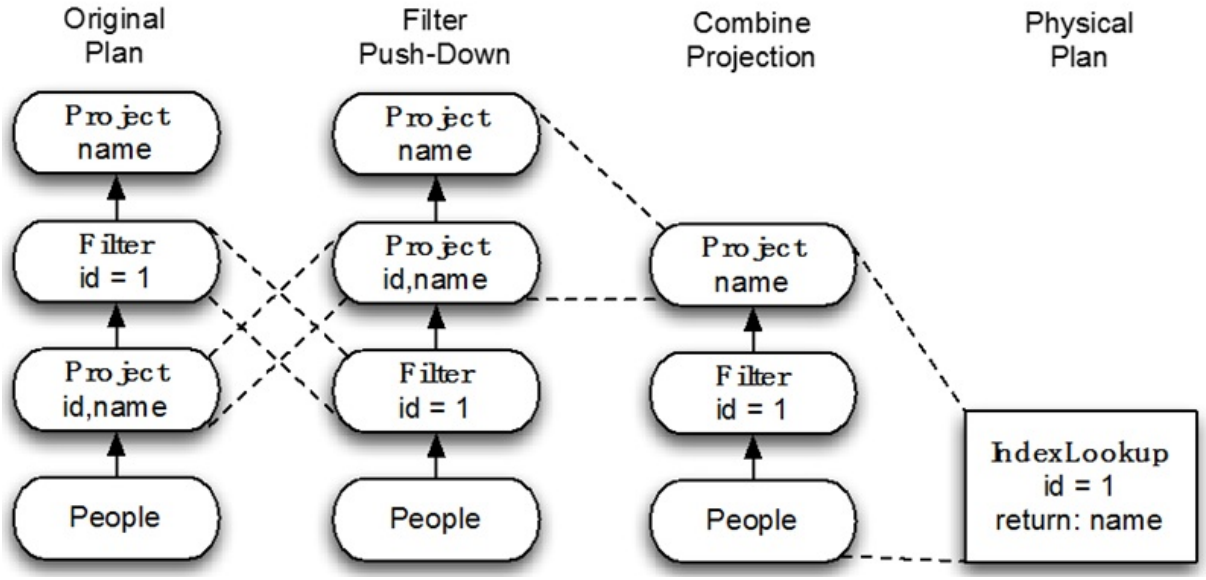


而最直接的方法就是就按照下图的方式进行优化，这种方法的难点在于：很难写一个通用的框架支持不同情况的优化方法。



Catalyst的做法是，**每一条优化规则只做最简单的修改，不同的规则互相协作，依次循环地对Logical Plan进行优化，最后形成一个稳定的Optimized Logical Plan**。如下图所示，对于最原始的的Logical Plan

1. 先使用Filter Push Down规则，将Filter和Project进行交换
2. 然后使用Combine Projection规则，将两个Project合并
3. 最后生成物理计划



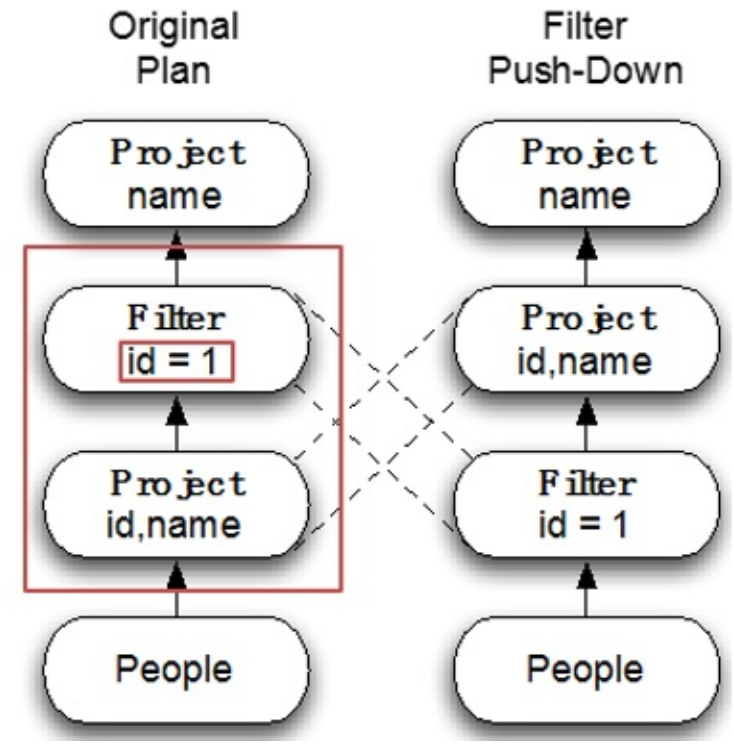
规则

一个规则一般包括三个部分：

1. 匹配

2. 过滤
3. 修改

让我们以Filter Push Down规则为例，看看Catalyst是如果执行规则的。



1. 首先在Logical Plan上寻找Filter Push Down的匹配条件，即Filter下面是Project
2. 然后判断过滤条件，即Filter是否可以不需要通过Project就可以计算
3. 最后做修改，即交换Filter和Project

利用Scala的 `Case Class` 和 `match` 语法来实现Catalyst的优化规则是非常简单的，例如Filter Push Down规则可以用下面几句代码实现：

```
val newPlan = queryPlan transform {
    case f @ Filter(_, p @ Project(_, grandChild))
      if(f.references subsetOf grandChild.output) =>
    p.copy(child = f.copy(child = grandChild)
}
```

1. `queryPlan` 是原始的未经优化的Logical Plan，是一个Tree
2. `case f @ Filter(_, p @ Project(_, grandChild))` 用来寻找Filter下面是Project的模式
3. `if(f.references subsetOf grandChild.output)` 用来判断Filter是否可以不需要通过Project就可以计算
4. `p.copy(child = f.copy(child = grandChild)` 用来交换Filter和Project

例子

让我们以下面这个sql查询为例子，看看SparkSQL是怎么从Logical Plan进行优化，最后生成Physical Plan的。

```
println(
      sql( s"""
            |SELECT name
            |FROM (SELECT name, age FROM rddTable) p
            |WHERE p.age >= 13 AND p.age <= 19
            |""".stripMargin).queryExecution
    )
```

Parsed Logical Plan

首先SqlParser会对输入的sql语句进行parser，生成Parsed Logical Plan。该Parsed Logcial Plan没有经过任何优化，是sql语句的直接翻译。

```
== Parsed Logical Plan ==
'Project ['name]
 'Filter (('p.age. >= 13) && ('p.age. <= 19))
  'Subquery p
   'Project ['name,'age]
    'UnresolvedRelation None, rddTable, None
```

我们依次从下往上看

1. UnresolvedRelation表示一个表格，目前只有表名rddTable，对应于sql中的 `FROM rddTable`
2. Project表示投影，对应于sql中的 `SELECT name, age`
3. Subquery是子查询，对应于sql中的 `(SELECT name, age FROM rddTable) p`
4. Filter是过滤条件，对应于sql中的 `WHERE p.age >= 13 AND p.age <= 19`
5. Project表示投影，对应于sql中的 `SELECT name`

值得注意的是，Project、Filter、Subquery和UnresolvedRelation上面的单引号表示该Logical Plan是Unresolved。

Analyzed Logical Plan

接下来的任务正式交接给Catalyst，首先Catalyst会根据内部定义的规则，将Parsed Logical Plan变成Analyzed Logical Plan。

```
== Analyzed Logical Plan ==
Project [name#0]
 Filter ((age#1 >= 13) && (age#1 <= 19))
  Project [name#0,age#1]
   LogicalRDD [name#0,age#1], MapPartitionsRDD[4] at mapPartitions at ExistingRDD.s
cala:36
```

其中

1. UnresolvedRelation被映射成了具体的LogicalRDD，而table name到Logical Plan的映射被保持在Catalog

2. UnresolvedAttribute被映射成AttributeReference

Optimized Logical Plan

然后Catalyst会继续对Analyzed Logical Plan进行优化，生成Optimized Logical Plan。

```
== Optimized Logical Plan ==
Project [name#0]
 Filter ((age#1 >= 13) && (age#1 <= 19))
  LogicalRDD [name#0,age#1], MapPartitionsRDD[4] at mapPartitions at ExistingRDD.sc
ala:36
```

其中进行了两次规则应用：

1. Filter Push Down将Filter和Project进行交换
2. Combine Projection将两个Project合并

Physical Plan

最后把Optimized Logical一对一的映射为Physical Plan。

```
== Physical Plan ==
Project [name#0]
 Filter ((age#1 >= 13) && (age#1 <= 19))
  PhysicalRDD [name#0,age#1], MapPartitionsRDD[4] at mapPartitions at ExistingRDD.s
cala:36
```

总结

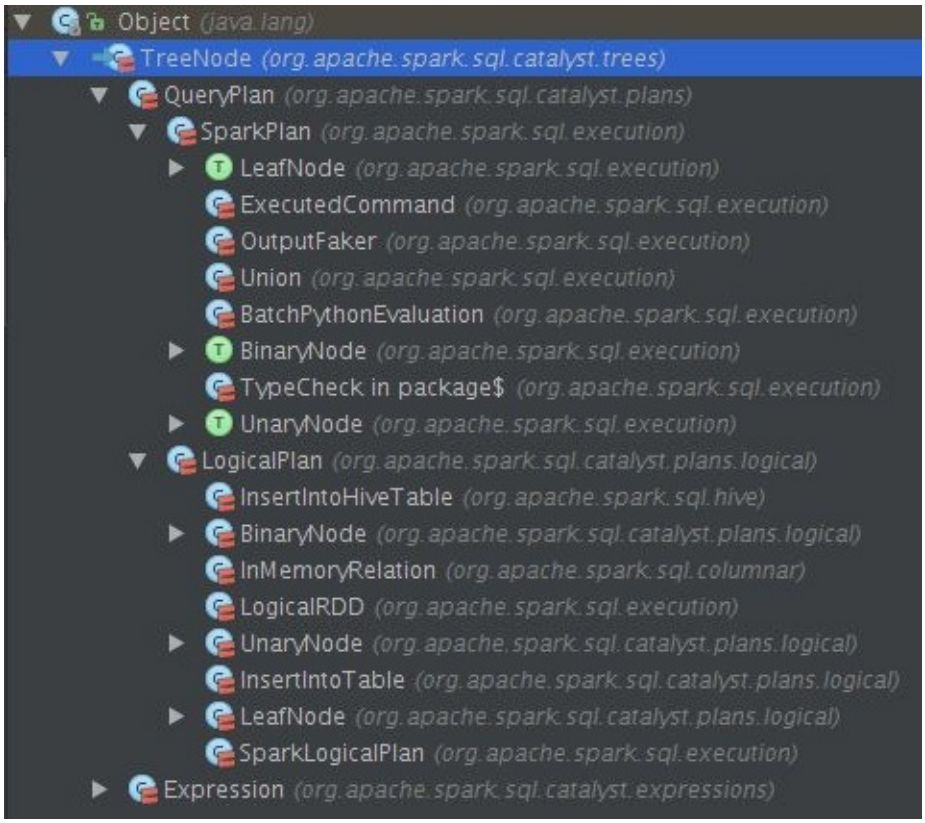
Catalyst的输入是Unresolved Logical Plan，在优化的过程当中，Catalyst会根据Analyzer和Optimizer中定义的规则，检测输入的执行计划中有没有符合规则条件的子树，如果有的话就会触发某个特定的优化规则，这些规则将依次循环地运行，直到达到最大迭代次数或者达到稳定的输出。

TreeNode

Catalyst中规则的匹配和Logical Plan的转换操作，其实都是基于树的操作，LogicalPlan继承自QueryPlan，而QueryPlan继承自TreeNode。

```
abstract class LogicalPlan extends QueryPlan[LogicalPlan] with Logging

abstract class QueryPlan[PlanType <: TreeNode[PlanType]] extends TreeNode[PlanType]

abstract class TreeNode[BaseType <: TreeNode[BaseType]]
```

TreeNode Library是Catalyst的核心类库，语法树的构建都是由一个个TreeNode组成。在Catalyst里，这些Node都是继承自Logical Plan，可以说每一个TreeNode节点就是一个Logical Plan。主要继承关系类图如下：



下面介绍一下TreeNode上的一些主要操作。

transform

该方法接受一个PartialFunction，例如Analyzer的Batch里面的Rule。将Rule迭代应用到该节点的所有子节点，最后返回这个节点的副本。如果rule没有对一个节点进行PartialFunction的操作，就返回这个节点本身。

```
/**
```

```
   * Returns a copy of this node where `rule` has been recursively applied to the t
ree.
   * When `rule` does not apply to a given node it is left unchanged.
   * Users should not expect a specific directionality. If a specific directionalit
y is needed,
   * transformDown or transformUp should be used.
   * @param rule the function use to transform this nodes children
   */
  def transform(rule: PartialFunction[BaseType, BaseType]): BaseType = {
    transformDown(rule)
  }
```

transformDown & transformUp

transform方法真正的调用是transformDown，这里用到了用先序遍历来对子节点进行递归的Rule应用。如果在对当前节点应用rule成功，修改后的节点afterRule，来对其children节点进行rule的应用。

transfromUp用的是后序遍历。

```
 /**
   * Returns a copy of this node where `rule` has been recursively applied to it an
d all of its
   * children (pre-order). When `rule` does not apply to a given node it is left un
changed.
   * @param rule the function used to transform this nodes children
   */
  def transformDown(rule: PartialFunction[BaseType, BaseType]): BaseType = {
    val afterRule = rule.applyOrElse(this, identity[BaseType])
    // Check if unchanged and then possibly return old copy to avoid gc churn.
    if (this fastEquals afterRule) {
      transformChildrenDown(rule)
    } else {
      afterRule.transformChildrenDown(rule)
    }
  }
```

transformChildrenDown & transformChildrenUp

transformChildrenDown是最重要的方法，对children节点进行递归的调用PartialFunction，利用最终返回的newArgs来生成一个新的节点，这里调用了makeCopy()来生成节点。

transformChildrenUp类似，只是最后调用transformUp。

```
  /**
   * Returns a copy of this node where `rule` has been recursively applied to all t
he children of
   * this node.  When `rule` does not apply to a given node it is left unchanged.
   * @param rule the function used to transform this nodes children
```

```
   */
  def transformChildrenDown(rule: PartialFunction[BaseType, BaseType]): this.type =
 {
    var changed = false
    val newArgs = productIterator.map {
      case arg: TreeNode[_] if children contains arg =>
        val newChild = arg.asInstanceOf[BaseType].transformDown(rule)
        if (!(newChild fastEquals arg)) {
          changed = true
          newChild
        } else {
          arg
        }
      case Some(arg: TreeNode[_]) if children contains arg =>
        val newChild = arg.asInstanceOf[BaseType].transformDown(rule)
        if (!(newChild fastEquals arg)) {
          changed = true
          Some(newChild)
        } else {
          Some(arg)
        }
      case m: Map[_,_] => m
      case args: Traversable[_] => args.map {
        case arg: TreeNode[_] if children contains arg =>
          val newChild = arg.asInstanceOf[BaseType].transformDown(rule)
          if (!(newChild fastEquals arg)) {
            changed = true
            newChild
          } else {
            arg
          }
        case other => other
      }
      case nonChild: AnyRef => nonChild
      case null => null
    }.toArray
    if (changed) makeCopy(newArgs) else this
  }
```

makeCopy

通过反射生成节点副本。

```
/**
   * Creates a copy of this type of tree node after a transformation.
   * Must be overridden by child classes that have constructor arguments
   * that are not present in the productIterator.
   * @param newArgs the new product arguments.
   */
```

```
  def makeCopy(newArgs: Array[AnyRef]): this.type = attachTree(this, "makeCopy") {
    try {
      // Skip no-arg constructors that are just there for kryo.
      val defaultCtor = getClass.getConstructors.find(_.getParameterTypes.size != 0
).head
      if (otherCopyArgs.isEmpty) {
        defaultCtor.newInstance(newArgs: _*).asInstanceOf[this.type]
      } else {
        defaultCtor.newInstance((newArgs ++ otherCopyArgs).toArray: _*).asInstanceO
f[this.type]
      }
    } catch {
      case e: java.lang.IllegalArgumentException =>
        throw new TreeNodeException(
          this, s"Failed to copy node.  Is otherCopyArgs specified correctly for $n
odeName? "
            + s"Exception message: ${e.getMessage}.")
    }
  }
```

其他函数

除此以外，TreeNode还支持一些集合操作函数

map 将函数作用到每一个结点，返回转换后的树

```
/**
   * Returns a Seq containing the result of applying the given function to each
   * node in this tree in a preorder traversal.
   * @param f the function to be applied.
   */
  def map[A](f: BaseType => A): Seq[A] = {
    val ret = new collection.mutable.ArrayBuffer[A]()
    foreach(ret += f(_))
    ret
  }
```

flatMap 将函数作用到每个结点，返回一个Seq

```
/**
   * Returns a Seq by applying a function to all nodes in this tree and using the e
lements of the
   * resulting collections.
   */
  def flatMap[A](f: BaseType => TraversableOnce[A]): Seq[A] = {
    val ret = new collection.mutable.ArrayBuffer[A]()
    foreach(ret ++= f(_))
    ret
  }
```

collect 将函数作用到每个结点，去除没有该函数定义的结点，返回剩下的结果Seq

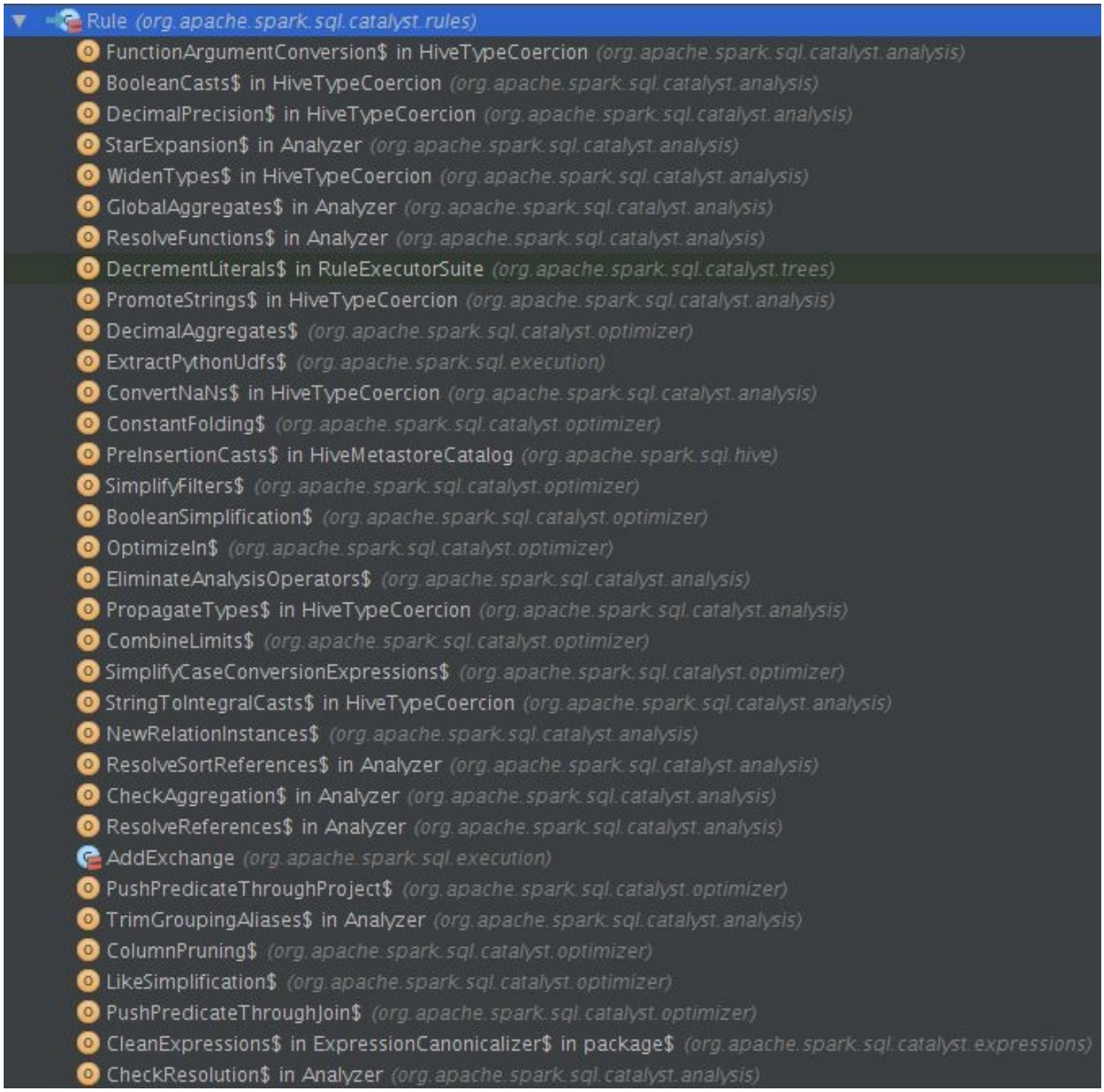
```
/**
   * Returns a Seq containing the result of applying a partial function to all elem
ents in this
   * tree on which the function is defined.
   */
  def collect[B](pf: PartialFunction[BaseType, B]): Seq[B] = {
    val ret = new collection.mutable.ArrayBuffer[B]()
    val lifted = pf.lift
    foreach(node => lifted(node).foreach(ret.+=))
    ret
  }
```

Rule

Rule类是抽象类，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。调用apply方法可以进行Tree的transformation。

```
abstract class Rule[TreeType <: TreeNode[_]] extends Logging {

  /** Name for this rule, automatically inferred based on class name. */
  val ruleName: String = {
    val className = getClass.getName
    if (className endsWith "$") className.dropRight(1) else className
  }

  def apply(plan: TreeType): TreeType
}
```

SparkSQL中为Catalyst定义了各种各样的Rule，如下图（不全），随着SparkSQL的不断优化，里面的Rule会越来越多。



Strategy

最大的执行次数，如果执行次数在最大迭代次数之前就达到了fix point，策略就会停止，不再应用了。

```
/**
   * An execution strategy for rules that indicates the maximum number of execution
s. If the
   * execution reaches fix point (i.e. converge) before maxIterations, it will stop
.
   */
  abstract class Strategy { def maxIterations: Int }
```

Once

执行且仅执行一次

```
 /** A strategy that only runs once. */
  case object Once extends Strategy { val maxIterations = 1 }
```

FixedPoint

相当于迭代次数的上限。

```
 /** A strategy that runs until fix point or maxIterations times, whichever comes f
irst. */
  case class FixedPoint(maxIterations: Int) extends Strategy
```

Batch

批次，这个对象是由一系列Rule组成的，采用一个策略，目前有两种策略Once和FixedPoint

```
/** A batch of rules. */
  protected case class Batch(name: String, strategy: Strategy, rules: Rule[TreeType
]*)
```

RuleExecutor

Rule具体的实现在RuleExecutor中。

通过定义batchs，可以模块化地对Tree进行transform操作。Once和FixedPoint分别可以对Tree进行一次操作或多次操作，例如对某些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。

```
abstract class RuleExecutor[TreeType <: TreeNode[_]] extends Logging {
  /**
   * An execution strategy for rules that indicates the maximum number of execution
s. If the
   * execution reaches fix point (i.e. converge) before maxIterations, it will stop
.
   */
  abstract class Strategy { def maxIterations: Int }

  /** A strategy that only runs once. */
  case object Once extends Strategy { val maxIterations = 1 }

  /** A strategy that runs until fix point or maxIterations times, whichever comes
first. */
  case class FixedPoint(maxIterations: Int) extends Strategy

  /** A batch of rules. */
  protected case class Batch(name: String, strategy: Strategy, rules: Rule[TreeType
]*)
```

```
  /** Defines a sequence of rule batches, to be overridden by the implementation. *
/
  protected val batches: Seq[Batch]
}
```

触发RuleExecutor启动的是apply函数：

1. 首先遍历batches数组
2. 依次运行单个batch里面的所有规则
3. 直到达到Strtegy里面定义的次数，或者优化前后Logical Plan不再变化

```
/**
   * Executes the batches of rules defined by the subclass. The batches are execute
d serially
   * using the defined execution strategy. Within each batch, rules are also execut
ed serially.
   */
  def apply(plan: TreeType): TreeType = {
    var curPlan = plan

    batches.foreach { batch =>
      val batchStartPlan = curPlan
      var iteration = 1
      var lastPlan = curPlan
      var continue = true

      // Run until fix point (or the max number of iterations as specified in the s
trategy.
      while (continue) {
        curPlan = batch.rules.foldLeft(curPlan) {
          case (plan, rule) =>
            val result = rule(plan)
            if (!result.fastEquals(plan)) {
              logTrace(
                s"""
                  |=== Applying Rule ${rule.ruleName} ===
                  |${sideBySide(plan.treeString, result.treeString).mkString("\n")}
                """.stripMargin)
            }

            result
        }
        iteration += 1
        if (iteration > batch.strategy.maxIterations) {
          // Only log if this is a rule that is supposed to run more than once.
          if (iteration != 2) {
            logInfo(s"Max iterations (${iteration - 1}) reached for batch ${batch.n
ame}")
          }
```

```
          continue = false
        }

        if (curPlan.fastEquals(lastPlan)) {
          logTrace(
            s"Fixed point reached for batch ${batch.name} after ${iteration - 1} it
erations.")
          continue = false
        }
        lastPlan = curPlan
      }

      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
  }
```

Analyser

Analyzer的主要职责是将Sql Parser生成的Unresolved Logical Plan转化成Resolved Logical Plan。Analyzer会利用Catalog和FunctionRegistry里面注册的表格和用户定义的函数，将UnresolvedAttribute和UnresolvedRelation转换为Catalyst里全类型的对象。

在介绍Analyzer之前先介绍一下Catalog和FunctionRegistry这两个模块。

Catalog

Catalog里面记录了Table Name到Logical Plan的映射，提供了注册表格，查找表格等接口。

```
/**
 * An interface for looking up relations by name.  Used by an [[Analyzer]].
 */
trait Catalog {

  def caseSensitive: Boolean

  def tableExists(db: Option[String], tableName: String): Boolean

  def lookupRelation(
    databaseName: Option[String],
    tableName: String,
    alias: Option[String] = None): LogicalPlan

  def registerTable(databaseName: Option[String], tableName: String, plan: LogicalP
lan): Unit

  def unregisterTable(databaseName: Option[String], tableName: String): Unit

  def unregisterAllTables(): Unit
  ...
}
```

Catalog具体的实现是SimpleCatalog，里面是用HashMap来记录Table Name到Logical Plan的映射。

```
class SimpleCatalog(val caseSensitive: Boolean) extends Catalog {
  val tables = new mutable.HashMap[String, LogicalPlan]()

  override def registerTable(
      databaseName: Option[String],
      tableName: String,
      plan: LogicalPlan): Unit = {
    val (dbName, tblName) = processDatabaseAndTableName(databaseName, tableName)
    tables += ((tblName, plan))
  }
```

```
  override def unregisterTable(
      databaseName: Option[String],
      tableName: String) = {
    val (dbName, tblName) = processDatabaseAndTableName(databaseName, tableName)
    tables -= tblName
  }

  override def unregisterAllTables() = {
    tables.clear()
  }

  override def tableExists(db: Option[String], tableName: String): Boolean = {
    val (dbName, tblName) = processDatabaseAndTableName(db, tableName)
    tables.get(tblName) match {
      case Some(_) => true
      case None => false
    }
  }

  override def lookupRelation(
      databaseName: Option[String],
      tableName: String,
      alias: Option[String] = None): LogicalPlan = {
    val (dbName, tblName) = processDatabaseAndTableName(databaseName, tableName)
    val table = tables.getOrElse(tblName, sys.error(s"Table Not Found: $tableName")
)
    val tableWithQualifiers = Subquery(tblName, table)

    // If an alias was specified by the lookup, wrap the plan in a subquery so that
 attributes are
    // properly qualified with this alias.
    alias.map(a => Subquery(a, tableWithQualifiers)).getOrElse(tableWithQualifiers)
  }
}
```

FunctionRegistry

FunctionRegistry记录了用户定义的函数名到该函数的表达式的映射，并提供注册函数，查找函数等接口。 FunctionBuilder被定义成为 `Seq[Expression] => Expression` ，可以理解为输入多个Expression作为参数，输出一个Expression作为结果。

```
/** A catalog for looking up user defined functions, used by an [[Analyzer]]. */
trait FunctionRegistry {
  type FunctionBuilder = Seq[Expression] => Expression

  def registerFunction(name: String, builder: FunctionBuilder): Unit

  def lookupFunction(name: String, children: Seq[Expression]): Expression
```

```
}
```

FunctionRegistry具体的实现是SimpleFunctionRegistry，里面用HashMap来记录用户定义的函数名到该函数的表达式的映射。

```
class SimpleFunctionRegistry extends FunctionRegistry {
  val functionBuilders = new mutable.HashMap[String, FunctionBuilder]()

  def registerFunction(name: String, builder: FunctionBuilder) = {
    functionBuilders.put(name, builder)
  }

  override def lookupFunction(name: String, children: Seq[Expression]): Expression
= {
    functionBuilders(name)(children)
  }
}
```

Analyzer

Analyzer里面有一个fixedPoint对象，一个Seq[Batch]对象。

```
/**
 * Provides a logical query plan analyzer, which translates [[UnresolvedAttribute]]
s and
 * [[UnresolvedRelation]]s into fully typed objects using information in a schema [
[Catalog]] and
 * a [[FunctionRegistry]].
 */
class Analyzer(catalog: Catalog, registry: FunctionRegistry, caseSensitive: Boolean
)
  extends RuleExecutor[LogicalPlan] with HiveTypeCoercion {

  val resolver = if (caseSensitive) caseSensitiveResolution else caseInsensitiveRes
olution

  // TODO: pass this in as a parameter.
  val fixedPoint = FixedPoint(100)

  /**
   * Override to provide additional rules for the "Resolution" batch.
   */
  val extendedRules: Seq[Rule[LogicalPlan]] = Nil

  lazy val batches: Seq[Batch] = Seq(
    Batch("MultiInstanceRelations", Once,
      NewRelationInstances),
    Batch("Resolution", fixedPoint,
```

```
      ResolveReferences ::
      ResolveRelations ::
      ResolveSortReferences ::
      NewRelationInstances ::
      ImplicitGenerate ::
      StarExpansion ::
      ResolveFunctions ::
      GlobalAggregates ::
      UnresolvedHavingClauseAttributes ::
      TrimGroupingAliases ::
      typeCoercionRules ++
      extendedRules : _*),
    Batch("Check Analysis", Once,
      CheckResolution,
      CheckAggregation),
    Batch("AnalysisOperators", fixedPoint,
      EliminateAnalysisOperators)
  )
  ...
}
```

Analyzer解析主要是根据这个batches里面的各种Rule来对Unresolved Logical Plan进行解析的。这里Analyzer类本身并没有定义执行的方法，而实现在它的父类RuleExecutor[LogicalPlan]中。

Rules介绍

在batches里面定义了4个Batch:

1. MultiInstanceRelations (Once)
2. Resolution (fixedPoint)
3. Check Analysis (Once)
4. AnalysisOperators (fixedPoint)

不同的batch是顺序执行的，也就是说MultiInstanceRelations执行完了，才会执行Resolution。

Once表示MultiInstanceRelations的Rule只会执行一次，fixedPoint表示Resolution里面的Rule会反复执行多次，具体几次定义在FixedPoint里面（当然如果运行Rule前后的LogicalPlan没有变化，也会提前停止执行）。

Resolution这个Batch里面定义了十几个Rules，例如ResolveReferences，ResolveRelations，ResolveSortReferences,etc。这些不同的Rule会循环执行fixedPoint次，执行的顺序是依次执行，也就是说，ResolveReferences -> ResolveRelations -> ResolveSortReferences -> ResolveReferences -> ResolveRelations -> ResolveSortReferences -> ...，类似这个顺序。

下面介绍一下每一个Rule具体做什么事情。

MultiInstanceRelation

如果一个实例在Logical Plan里出现了多次，则会应用NewRelationInstances这条Rule。

```
/**
 * If any MultiInstanceRelation appears more than once in the query plan then the p
lan is updated so
 * that each instance has unique expression ids for the attributes produced.
 */
object NewRelationInstances extends Rule[LogicalPlan] {
  def apply(plan: LogicalPlan): LogicalPlan = {
    //将logical plan应用partial function得到所有MultiInstanceRelation的plan的集合
    val localRelations = plan collect { case l: MultiInstanceRelation => l}

    val multiAppearance = localRelations
      .groupBy(identity[MultiInstanceRelation]) //group by操作
      .filter { case (_, ls) => ls.size > 1 } //如果只取size大于1的进行后续操作
      .map(_._1)
      .toSet

    //更新plan，使得每个实例的expId是唯一的。
    plan transform {
      case l: MultiInstanceRelation if multiAppearance contains l => l.newInstance
    }
  }
}
```

ResolveReferences

将Sql parser解析出来的UnresolvedAttribute全部都转为对应的实际的catalyst.expressions.AttributeReference。这里调用了Logical Plan的resolveChildren方法，将属性转为NamedExepression。

```
/**
   * Replaces [[UnresolvedAttribute]]s with concrete
   * [[catalyst.expressions.AttributeReference AttributeReferences]] from a logical
 plan node's
   * children.
   */
  object ResolveReferences extends Rule[LogicalPlan] {
    def apply(plan: LogicalPlan): LogicalPlan = plan transformUp {
      case q: LogicalPlan if q.childrenResolved =>
        logTrace(s"Attempting to resolve ${q.simpleString}")
        q transformExpressions {
          case u @ UnresolvedAttribute(name) =>
            // Leave unchanged if resolution fails.  Hopefully will be resolved nex
t round.
            val result = q.resolveChildren(name, resolver).getOrElse(u)
            logDebug(s"Resolving $u to $result")
            result
        }
    }
  }
```

ResolveRelations

在 `select * from src` 中，src表parse后就是一个UnresolvedRelation节点。ResolveRelations就是把src替换成具体的LogicalPlan。而这个table name到LogicalPlan的映射可以从Catalog里获得。

```
/**
   * Replaces [[UnresolvedRelation]]s with concrete relations from the catalog.
   */
  object ResolveRelations extends Rule[LogicalPlan] {
    def apply(plan: LogicalPlan): LogicalPlan = plan transform {
      case i @ InsertIntoTable(UnresolvedRelation(databaseName, name, alias), _, _,
 _) =>
        i.copy(
          table = EliminateAnalysisOperators(catalog.lookupRelation(databaseName, n
ame, alias)))
      case UnresolvedRelation(databaseName, name, alias) =>
        catalog.lookupRelation(databaseName, name, alias)
    }
  }
```

```
def lookupRelation(
    databaseName: Option[String],
    tableName: String,
    alias: Option[String] = None): LogicalPlan
```

ResolveSortReferences

在某些SQL的定义里面，可以允许按照没有出现在select里面的attribute进行sort。这个规则是用来检测这些语法，并且自动把sort的attribute加入到select里面，并且在上次加入去到这个attribute的projection。

```
/**
   * In many dialects of SQL is it valid to sort by attributes that are not present
 in the SELECT
   * clause.  This rule detects such queries and adds the required attributes to th
e original
   * projection, so that they will be available during sorting. Another projection
is added to
   * remove these attributes after sorting.
   */
  object ResolveSortReferences extends Rule[LogicalPlan] {
    def apply(plan: LogicalPlan): LogicalPlan = plan transformUp {
      case s @ Sort(ordering, p @ Project(projectList, child)) if !s.resolved && p.
resolved =>
        val unresolved = ordering.flatMap(_.collect { case UnresolvedAttribute(name
) => name })
        val resolved = unresolved.flatMap(child.resolve(_, resolver))
        val requiredAttributes = AttributeSet(resolved.collect { case a: Attribute
```

```
=> a })

        val missingInProject = requiredAttributes -- p.output
        if (missingInProject.nonEmpty) {
          // Add missing attributes and then project them away after the sort.
          Project(projectList.map(_.toAttribute),
            Sort(ordering,
              Project(projectList ++ missingInProject, child)))
        } else {
          logDebug(s"Failed to find $missingInProject in ${p.output.mkString(", ")}
")
          s // Nothing we can do here. Return original plan.
        }
      case s @ Sort(ordering, a @ Aggregate(grouping, aggs, child)) if !s.resolved
&& a.resolved =>
        val unresolved = ordering.flatMap(_.collect { case UnresolvedAttribute(name
) => name })
        // A small hack to create an object that will allow us to resolve any refer
ences that
        // refer to named expressions that are present in the grouping expressions.
        val groupingRelation = LocalRelation(
          grouping.collect { case ne: NamedExpression => ne.toAttribute }
        )

        logDebug(s"Grouping expressions: $groupingRelation")
        val resolved = unresolved.flatMap(groupingRelation.resolve(_, resolver))
        val missingInAggs = resolved.filterNot(a.outputSet.contains)
        logDebug(s"Resolved: $resolved Missing in aggs: $missingInAggs")
        if (missingInAggs.nonEmpty) {
          // Add missing grouping exprs and then project them away after the sort.
          Project(a.output,
            Sort(ordering,
              Aggregate(grouping, aggs ++ missingInAggs, child)))
        } else {
          s // Nothing we can do here. Return original plan.
        }
    }
  }
```

ImplicitGenerate

如果在select语句里只有一个表达式，而且这个表达式是一个Generator（Generator是一个1条记录生成到N条记录的映射）。当在解析逻辑计划时，遇到Project节点的时候，就可以将它转换为Generate类（Generate类是将输入流应用一个函数，从而生成一个新的流）。

```
/**
   * When a SELECT clause has only a single expression and that expression is a
   * [[catalyst.expressions.Generator Generator]] we convert the
   * [[catalyst.plans.logical.Project Project]] to a [[catalyst.plans.logical.Gener
ate Generate]].
```

```
   */
  object ImplicitGenerate extends Rule[LogicalPlan] {
    def apply(plan: LogicalPlan): LogicalPlan = plan transform {
      case Project(Seq(Alias(g: Generator, _)), child) =>
        Generate(g, join = false, outer = false, None, child)
    }
  }
```

StarExpansion

在Project操作符里，如果是*符号，可以将所有的references都展开成实际的字段。

```
/**
   * Expands any references to [[Star]] (*) in project operators.
   */
  object StarExpansion extends Rule[LogicalPlan] {
    def apply(plan: LogicalPlan): LogicalPlan = plan transform {
      // Wait until children are resolved
      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.output, resolver)
            case o => o :: Nil
          },
          child)
      case t: ScriptTransformation if containsStar(t.input) =>
        t.copy(
          input = t.input.flatMap {
            case s: Star => s.expand(t.child.output, resolver)
            case o => o :: Nil
          }
        )
      // If the aggregate function argument contains Stars, expand it.
      case a: Aggregate if containsStar(a.aggregateExpressions) =>
        a.copy(
          aggregateExpressions = a.aggregateExpressions.flatMap {
            case s: Star => s.expand(a.child.output, resolver)
            case o => o :: Nil
          }
        )
    }

    /**
     * Returns true if `exprs` contains a [[Star]].
     */
    protected def containsStar(exprs: Seq[Expression]): Boolean =
      exprs.collect { case _: Star => true }.nonEmpty
  }
```

```
}
```

ResolveFunctions

这里主要是对UDF进行resolve，将这些UDF可以从FunctionRegistry里找到。

```
/**
   * Replaces [[UnresolvedFunction]]s with concrete [[catalyst.expressions.Expressi
on Expressions]].
   */
  object ResolveFunctions extends Rule[LogicalPlan] {
    def apply(plan: LogicalPlan): LogicalPlan = plan transform {
      case q: LogicalPlan =>
        q transformExpressions {
          case u @ UnresolvedFunction(name, children) if u.childrenResolved =>
            registry.lookupFunction(name, children)
        }
    }
  }
```

GlobalAggregates

如果遇到包含Aggregate的Project，就返回一个Aggregate。

```
/**
   * Turns projections that contain aggregate expressions into aggregations.
   */
  object GlobalAggregates extends Rule[LogicalPlan] {
    def apply(plan: LogicalPlan): LogicalPlan = plan transform {
      case Project(projectList, child) if containsAggregates(projectList) =>
        Aggregate(Nil, projectList, child)
    }

    def containsAggregates(exprs: Seq[Expression]): Boolean = {
      exprs.foreach(_.foreach {
        case agg: AggregateExpression => return true
        case _ =>
      })
      false
    }
  }
```

UnresolvedHavingClauseAttributes

这条规则会寻找Having子句中Unresolved Attributes，将这些Attributes下降到下面的Aggregates里面，最后在最上面添加Project。

```
/**
   * This rule finds expressions in HAVING clause filters that depend on
```

```
   * unresolved attributes.  It pushes these expressions down to the underlying
   * aggregates and then projects them away above the filter.
   */
  object UnresolvedHavingClauseAttributes extends Rule[LogicalPlan] {
    def apply(plan: LogicalPlan): LogicalPlan = plan transformUp {
      case filter @ Filter(havingCondition, aggregate @ Aggregate(_, originalAggExp
rs, _))
          if aggregate.resolved && containsAggregate(havingCondition) => {
        val evaluatedCondition = Alias(havingCondition,  "havingCondition")()
        val aggExprsWithHaving = evaluatedCondition +: originalAggExprs

        Project(aggregate.output,
          Filter(evaluatedCondition.toAttribute,
            aggregate.copy(aggregateExpressions = aggExprsWithHaving)))
      }
    }

    protected def containsAggregate(condition: Expression): Boolean =
      condition
        .collect { case ae: AggregateExpression => ae }
        .nonEmpty
  }
```

TrimGroupingAliases

去除Aggreate中没有操作的alias。

```
/**
   * Removes no-op Alias expressions from the plan.
   */
  object TrimGroupingAliases extends Rule[LogicalPlan] {
    def apply(plan: LogicalPlan): LogicalPlan = plan transform {
      case Aggregate(groups, aggs, child) =>
        Aggregate(groups.map(_.transform { case Alias(c, _) => c }), aggs, child)
    }
  }
```

CheckResolution

在上述主要的优化规则都运行完后，CheckResolution会运行一次，用来检查是不是所有的节点都已经resolved了，如果不是就抛异常。

```
/**
   * Makes sure all attributes and logical plans have been resolved.
   */
  object CheckResolution extends Rule[LogicalPlan] {
    def apply(plan: LogicalPlan): LogicalPlan = {
      plan.transform {
        case p if p.expressions.exists(!_.resolved) =>
```

```
          throw new TreeNodeException(p,
            s"Unresolved attributes: ${p.expressions.filterNot(_.resolved).mkString
(",")}")
        case p if !p.resolved && p.childrenResolved =>
          throw new TreeNodeException(p, "Unresolved plan found")
      } match {
        // As a backstop, use the root node to check that the entire plan tree is r
esolved.
        case p if !p.resolved =>
          throw new TreeNodeException(p, "Unresolved plan in tree")
        case p => p
      }
    }
  }
```

CheckAggregation

在上述主要的优化规则都运行完后，CheckAggregation也会运行一次，用于检查是否存在non-aggregated attributes，如果不是就抛异常。

```
/**
   * Checks for non-aggregated attributes with aggregation
   */
  object CheckAggregation extends Rule[LogicalPlan] {
    def apply(plan: LogicalPlan): LogicalPlan = {
      plan.transform {
        case aggregatePlan @ Aggregate(groupingExprs, aggregateExprs, child) =>
          def isValidAggregateExpression(expr: Expression): Boolean = expr match {
            case _: AggregateExpression => true
            case e: Attribute => groupingExprs.contains(e)
            case e if groupingExprs.contains(e) => true
            case e if e.references.isEmpty => true
            case e => e.children.forall(isValidAggregateExpression)
          }

          aggregateExprs.find { e =>
            !isValidAggregateExpression(e.transform {
              // Should trim aliases around `GetField`s. These aliases are introduc
ed while
              // resolving struct field accesses, because `GetField` is not a `Name
dExpression`.
              // (Should we just turn `GetField` into a `NamedExpression`?)
              case Alias(g: GetField, _) => g
            })
          }.foreach { e =>
            throw new TreeNodeException(plan, s"Expression not in GROUP BY: $e")
          }

          aggregatePlan
      }
```

```
    }
  }
```

EliminateAnalysisOperators

将Subquery移除。

```
/**
 * Removes [[catalyst.plans.logical.Subquery Subquery]] operators from the plan.  S
ubqueries are
 * only required to provide scoping information for attributes and can be removed o
nce analysis is
 * complete.
 */
object EliminateAnalysisOperators extends Rule[LogicalPlan] {
  def apply(plan: LogicalPlan): LogicalPlan = plan transform {
    case Subquery(_, child) => child
  }
}
```

Optimizer

Optimizer的主要职责是将Analyzer输出的Resolved Logical Plan根据不同的优化策略Batch，来对语法树进行优化。Optimizer的工作方式其实类似Analyzer，因为它们都继承自RuleExecutor[LogicalPlan]，都是执行一系列的Batch操作。

Rules介绍

Optimizer里的batches包含了4大类优化策略：

1. Combine Limits： 合并Limits
2. ConstantFolding： 常量合并
3. Decimal Optimizations： Decimal类型数据计算优化
4. Filter Pushdown： 过滤器下推

```
object DefaultOptimizer extends Optimizer {
  val batches =
    Batch("Combine Limits", FixedPoint(100),
      CombineLimits) ::
    Batch("ConstantFolding", FixedPoint(100),
      NullPropagation,
      ConstantFolding,
      LikeSimplification,
      BooleanSimplification,
      SimplifyFilters,
      SimplifyCasts,
      SimplifyCaseConversionExpressions,
      OptimizeIn) ::
    Batch("Decimal Optimizations", FixedPoint(100),
      DecimalAggregates) ::
    Batch("Filter Pushdown", FixedPoint(100),
      UnionPushdown,
      CombineFilters,
      PushPredicateThroughProject,
      PushPredicateThroughJoin,
      ColumnPruning) :: Nil
}
```

CombineLimits

目前该规则无法优化 `val query = sql("select * from (select * from table limit 100) limit 10 ")` 这样的语句，因为CombineLimits在第一个batch里面，而且是唯一一个，也就是说它不能在其他规则的基础上再运行自己，而只能优化连续出现的两个Limit。

```
/**
 * Combines two adjacent [[Limit]] operators into one, merging the
```

```
 * expressions into one single expression.
 */
object CombineLimits extends Rule[LogicalPlan] {
  def apply(plan: LogicalPlan): LogicalPlan = plan transform {
    case ll @ Limit(le, nl @ Limit(ne, grandChild)) =>
      Limit(If(LessThan(ne, le), ne, le), grandChild)
  }
}
```

NullPropagation

Literal字面量是一个能匹配任意基本类型的类。

```
object Literal {
  def apply(v: Any): Literal = v match {
    case i: Int => Literal(i, IntegerType)
    case l: Long => Literal(l, LongType)
    case d: Double => Literal(d, DoubleType)
    case f: Float => Literal(f, FloatType)
    case b: Byte => Literal(b, ByteType)
    case s: Short => Literal(s, ShortType)
    case s: String => Literal(s, StringType)
    case b: Boolean => Literal(b, BooleanType)
    case d: BigDecimal => Literal(Decimal(d), DecimalType.Unlimited)
    case d: Decimal => Literal(d, DecimalType.Unlimited)
    case t: Timestamp => Literal(t, TimestampType)
    case d: Date => Literal(d, DateType)
    case a: Array[Byte] => Literal(a, BinaryType)
    case null => Literal(null, NullType)
  }
}
```

注意Literal是一个LeafExpression，核心方法是eval，给定Row，计算表达式返回值。

```
case class Literal(value: Any, dataType: DataType) extends LeafExpression {

  override def foldable = true
  def nullable = value == null


  override def toString = if (value != null) value.toString else "null"

  type EvaluatedType = Any
  override def eval(input: Row):Any = value
}
```

NullPropagation是一个能将Expression Expressions替换为等价的Literal值的优化，并且能够避免NULL值在SQL语法树的传播。

```
/**
 * Replaces [[Expression Expressions]] that can be statically evaluated with
 * equivalent [[Literal]] values. This rule is more specific with
 * Null value propagation from bottom to top of the expression tree.
 */
object NullPropagation extends Rule[LogicalPlan] {
  def apply(plan: LogicalPlan): LogicalPlan = plan transform {
    case q: LogicalPlan => q transformExpressionsUp {
      case e @ Count(Literal(null, _)) => Cast(Literal(0L), e.dataType)
      case e @ Sum(Literal(c, _)) if c == 0 => Cast(Literal(0L), e.dataType)
      case e @ Average(Literal(c, _)) if c == 0 => Literal(0.0, e.dataType)
      case e @ IsNull(c) if !c.nullable => Literal(false, BooleanType)
      case e @ IsNotNull(c) if !c.nullable => Literal(true, BooleanType)
      case e @ GetItem(Literal(null, _), _) => Literal(null, e.dataType)
      case e @ GetItem(_, Literal(null, _)) => Literal(null, e.dataType)
      case e @ GetField(Literal(null, _), _) => Literal(null, e.dataType)
      case e @ EqualNullSafe(Literal(null, _), r) => IsNull(r)
      case e @ EqualNullSafe(l, Literal(null, _)) => IsNull(l)

      // For Coalesce, remove null literals.
      case e @ Coalesce(children) =>
        val newChildren = children.filter {
          case Literal(null, _) => false
          case _ => true
        }
        if (newChildren.length == 0) {
          Literal(null, e.dataType)
        } else if (newChildren.length == 1) {
          newChildren(0)
        } else {
          Coalesce(newChildren)
        }

      case e @ Substring(Literal(null, _), _, _) => Literal(null, e.dataType)
      case e @ Substring(_, Literal(null, _), _) => Literal(null, e.dataType)
      case e @ Substring(_, _, Literal(null, _)) => Literal(null, e.dataType)

      // Put exceptional cases above if any
      case e: BinaryArithmetic => e.children match {
        case Literal(null, _) :: right :: Nil => Literal(null, e.dataType)
        case left :: Literal(null, _) :: Nil => Literal(null, e.dataType)
        case _ => e
      }
      case e: BinaryComparison => e.children match {
        case Literal(null, _) :: right :: Nil => Literal(null, e.dataType)
        case left :: Literal(null, _) :: Nil => Literal(null, e.dataType)
        case _ => e
      }
      case e: StringRegexExpression => e.children match {
        case Literal(null, _) :: right :: Nil => Literal(null, e.dataType)
```

```
        case left :: Literal(null, _) :: Nil => Literal(null, e.dataType)
        case _ => e
      }
      case e: StringComparison => e.children match {
        case Literal(null, _) :: right :: Nil => Literal(null, e.dataType)
        case left :: Literal(null, _) :: Nil => Literal(null, e.dataType)
        case _ => e
      }
    }
  }
}
```

ConstantFolding

常量合并是属于Expression优化的一种，对于可以直接计算的常量，不用放到物理执行里去生成对象来计算了，直接可以在计划里就计算出来。

```
/**
 * Replaces [[Expression Expressions]] that can be statically evaluated with
 * equivalent [[Literal]] values.
 */
object ConstantFolding extends Rule[LogicalPlan] {
  def apply(plan: LogicalPlan): LogicalPlan = plan transform {
    case q: LogicalPlan => q transformExpressionsDown {
      // Skip redundant folding of literals. This rule is technically not necessary
. Placing this
      // here avoids running the next rule for Literal values, which would create a
 new Literal
      // object and running eval unnecessarily.
      case l: Literal => l

      // Fold expressions that are foldable.
      case e if e.foldable => Literal(e.eval(null), e.dataType)

      // Fold "literal in (item1, item2, ..., literal, ...)" into true directly.
      case In(Literal(v, _), list) if list.exists {
          case Literal(candidate, _) if candidate == v => true
          case _ => false
        } => Literal(true, BooleanType)
    }
  }
}
```

LikeSimplification

简化Like字句，如果符合下面四种情况之一：

1. startsWith
2. endsWith
3. contains

4. equalTo

```
/**
 * Simplifies LIKE expressions that do not need full regular expressions to evaluat
e the condition.
 * For example, when the expression is just checking to see if a string starts with
 a given
 * pattern.
 */
object LikeSimplification extends Rule[LogicalPlan] {
  // if guards below protect from escapes on trailing %.
  // Cases like "something\%" are not optimized, but this does not affect correctne
ss.
  val startsWith = "([^_%]+)%".r
  val endsWith = "%([^_%]+)".r
  val contains = "%([^_%]+)%".r
  val equalTo = "([^_%]*)".r

  def apply(plan: LogicalPlan): LogicalPlan = plan transformAllExpressions {
    case Like(l, Literal(startsWith(pattern), StringType)) if !pattern.endsWith("\\
") =>
      StartsWith(l, Literal(pattern))
    case Like(l, Literal(endsWith(pattern), StringType)) =>
      EndsWith(l, Literal(pattern))
    case Like(l, Literal(contains(pattern), StringType)) if !pattern.endsWith("\\")
 =>
      Contains(l, Literal(pattern))
    case Like(l, Literal(equalTo(pattern), StringType)) =>
      EqualTo(l, Literal(pattern))
  }
}
```

BooleanSimplification

这个是对布尔表达式的优化，有点像java布尔表达式中的短路判断。看看布尔表达式2边能不能通过只计算1边，而省去计算另一边而提高效率，称为简化布尔表达式。

```
/**
 * Simplifies boolean expressions where the answer can be determined without evalua
ting both sides.
 * Note that this rule can eliminate expressions that might otherwise have been eva
luated and thus
 * is only safe when evaluations of expressions does not result in side effects.
 */
object BooleanSimplification extends Rule[LogicalPlan] {
  def apply(plan: LogicalPlan): LogicalPlan = plan transform {
    case q: LogicalPlan => q transformExpressionsUp {
      case and @ And(left, right) =>
        (left, right) match {
          case (Literal(true, BooleanType), r) => r
```

```
          case (l, Literal(true, BooleanType)) => l
          case (Literal(false, BooleanType), _) => Literal(false)
          case (_, Literal(false, BooleanType)) => Literal(false)
          case (_, _) => and
        }

      case or @ Or(left, right) =>
        (left, right) match {
          case (Literal(true, BooleanType), _) => Literal(true)
          case (_, Literal(true, BooleanType)) => Literal(true)
          case (Literal(false, BooleanType), r) => r
          case (l, Literal(false, BooleanType)) => l
          case (_, _) => or
        }

      case not @ Not(exp) =>
        exp match {
          case Literal(true, BooleanType) => Literal(false)
          case Literal(false, BooleanType) => Literal(true)
          case GreaterThan(l, r) => LessThanOrEqual(l, r)
          case GreaterThanOrEqual(l, r) => LessThan(l, r)
          case LessThan(l, r) => GreaterThanOrEqual(l, r)
          case LessThanOrEqual(l, r) => GreaterThan(l, r)
          case Not(e) => e
          case _ => not
        }

      // Turn "if (true) a else b" into "a", and if (false) a else b" into "b".
      case e @ If(Literal(v, _), trueValue, falseValue) => if (v == true) trueValue
 else falseValue
    }
  }
}
```

SimplifyFilters

如果filter总是返回true，则删除filter返回child，如果filter总是返回false，则返回empty的relation。

```
/**
 * Removes filters that can be evaluated trivially.  This is done either by eliding
 the filter for
 * cases where it will always evaluate to `true`, or substituting a dummy empty rel
ation when the
 * filter will always evaluate to `false`.
 */
object SimplifyFilters extends Rule[LogicalPlan] {
  def apply(plan: LogicalPlan): LogicalPlan = plan transform {
    // If the filter condition always evaluate to true, remove the filter.
    case Filter(Literal(true, BooleanType), child) => child
    // If the filter condition always evaluate to null or false,
```

```
    // replace the input with an empty relation.
    case Filter(Literal(null, _), child) => LocalRelation(child.output, data = Seq.
empty)
    case Filter(Literal(false, BooleanType), child) => LocalRelation(child.output,
data = Seq.empty)
  }
}
```

SimplifyCasts

如果Cast的类型和实际类型一致，则去除没必要的cast。

```
/**
 * Removes [[Cast Casts]] that are unnecessary because the input is already the cor
rect type.
 */
object SimplifyCasts extends Rule[LogicalPlan] {
  def apply(plan: LogicalPlan): LogicalPlan = plan transformAllExpressions {
    case Cast(e, dataType) if e.dataType == dataType => e
  }
}
```

SimplifyCaseConversionExpressions

去除内层没必要的大小写转换，直接返回外层的转换。

```
/**
 * Removes the inner [[CaseConversionExpression]] that are unnecessary because
 * the inner conversion is overwritten by the outer one.
 */
object SimplifyCaseConversionExpressions extends Rule[LogicalPlan] {
  def apply(plan: LogicalPlan): LogicalPlan = plan transform {
    case q: LogicalPlan => q transformExpressionsUp {
      case Upper(Upper(child)) => Upper(child)
      case Upper(Lower(child)) => Upper(child)
      case Lower(Upper(child)) => Lower(child)
      case Lower(Lower(child)) => Lower(child)
    }
  }
}
```

OptimizeIn

将In(value, Seq[Literal])的节点转换为等价的InSet(value, HashSet[Literal])，会快很多。

```
/**
 * Replaces [[In (value, seq[Literal])]] with optimized version[[InSet (value, Hash
Set[Literal])]]
 * which is much faster
```

```
 */
object OptimizeIn extends Rule[LogicalPlan] {
  def apply(plan: LogicalPlan): LogicalPlan = plan transform {
    case q: LogicalPlan => q transformExpressionsDown {
      case In(v, list) if !list.exists(!_.isInstanceOf[Literal]) =>
          val hSet = list.map(e => e.eval(null))
          InSet(v, HashSet() ++ hSet)
    }
  }
}
```

DecimalAggregates

计算fixed-precision Decimal类型的sum和avg的时候，先把它转换为Long类型，做计算，最后在转化为Decimal，会比较快。

```
/**
 * Speeds up aggregates on fixed-precision decimals by executing them on unscaled L
ong values.
 *
 * This uses the same rules for increasing the precision and scale of the output as
 * [[org.apache.spark.sql.catalyst.analysis.HiveTypeCoercion.DecimalPrecision]].
 */
object DecimalAggregates extends Rule[LogicalPlan] {
  import Decimal.MAX_LONG_DIGITS

  /** Maximum number of decimal digits representable precisely in a Double */
  val MAX_DOUBLE_DIGITS = 15

  def apply(plan: LogicalPlan): LogicalPlan = plan transformAllExpressions {
    case Sum(e @ DecimalType.Expression(prec, scale)) if prec + 10 <= MAX_LONG_DIGI
TS =>
      MakeDecimal(Sum(UnscaledValue(e)), prec + 10, scale)

    case Average(e @ DecimalType.Expression(prec, scale)) if prec + 4 <= MAX_DOUBLE
_DIGITS =>
      Cast(
        Divide(Average(UnscaledValue(e)), Literal(math.pow(10.0, scale), DoubleType
)),
        DecimalType(prec + 4, scale + 4))
  }
}
```

UnionPushdown

把filter和project pushdown到union下面。

```
/**
  *  Pushes operations to either side of a Union.
```

```
  */
object UnionPushdown extends Rule[LogicalPlan] {

  /**
    *  Maps Attributes from the left side to the corresponding Attribute on the rig
ht side.
    */
  def buildRewrites(union: Union): AttributeMap[Attribute] = {
    assert(union.left.output.size == union.right.output.size)

    AttributeMap(union.left.output.zip(union.right.output))
  }

  /**
    *  Rewrites an expression so that it can be pushed to the right side of a Union
 operator.
    *  This method relies on the fact that the output attributes of a union are alw
ays equal
    *  to the left child's output.
    */
  def pushToRight[A <: Expression](e: A, rewrites: AttributeMap[Attribute]): A = {
    val result = e transform {
      case a: Attribute => rewrites(a)
    }

    // We must promise the compiler that we did not discard the names in the case o
f project
    // expressions.  This is safe since the only transformation is from Attribute =
> Attribute.
    result.asInstanceOf[A]
  }

  def apply(plan: LogicalPlan): LogicalPlan = plan transform {
    // Push down filter into union
    case Filter(condition, u @ Union(left, right)) =>
      val rewrites = buildRewrites(u)
      Union(
        Filter(condition, left),
        Filter(pushToRight(condition, rewrites), right))

    // Push down projection into union
    case Project(projectList, u @ Union(left, right)) =>
      val rewrites = buildRewrites(u)
      Union(
        Project(projectList, left),
        Project(projectList.map(pushToRight(_, rewrites)), right))
  }
}
```

CombineFilters

合并两个相邻的Filter,这个和上述Combine Limit差不多。合并2个节点，就可以减少树的深度从而减少重复执行过滤的代价。

```
/**
 * Combines two adjacent [[Filter]] operators into one, merging the
 * conditions into one conjunctive predicate.
 */
object CombineFilters extends Rule[LogicalPlan] {
  def apply(plan: LogicalPlan): LogicalPlan = plan transform {
    case ff @ Filter(fc, nf @ Filter(nc, grandChild)) => Filter(And(nc, fc), grandC
hild)
  }
}
```

PushPredicateThroughProject

Predict push到project下面，如果predict不依赖于project。

```
/**
 * Pushes [[Filter]] operators through [[Project]] operators, in-lining any [[Alias
 Aliases]]
 * that were defined in the projection.
 *
 * This heuristic is valid assuming the expression evaluation cost is minimal.
 */
object PushPredicateThroughProject extends Rule[LogicalPlan] {
  def apply(plan: LogicalPlan): LogicalPlan = plan transform {
    case filter @ Filter(condition, project @ Project(fields, grandChild)) =>
      val sourceAliases = fields.collect { case a @ Alias(c, _) =>
        (a.toAttribute: Attribute) -> c
      }.toMap
      project.copy(child = filter.copy(
        replaceAlias(condition, sourceAliases),
        grandChild))
  }

  def replaceAlias(condition: Expression, sourceAliases: Map[Attribute, Expression]
): Expression = {
    condition transform {
      case a: AttributeReference => sourceAliases.getOrElse(a, a)
    }
  }
}
```

PushPredicateThroughJoin

Predict push到project下面，如果predict不依赖于join。

```
/**
```

```
 * Pushes down [[Filter]] operators where the `condition` can be
 * evaluated using only the attributes of the left or right side of a join.  Other
 * [[Filter]] conditions are moved into the `condition` of the [[Join]].
 *
 * And also Pushes down the join filter, where the `condition` can be evaluated usi
ng only the
 * attributes of the left or right side of sub query when applicable.
 *
 * Check https://cwiki.apache.org/confluence/display/Hive/OuterJoinBehavior for mor
e details
 */
object PushPredicateThroughJoin extends Rule[LogicalPlan] with PredicateHelper {
  /**
   * Splits join condition expressions into three categories based on the attribute
s required
   * to evaluate them.
   * @return (canEvaluateInLeft, canEvaluateInRight, haveToEvaluateInBoth)
   */
  private def split(condition: Seq[Expression], left: LogicalPlan, right: LogicalPl
an) = {
    val (leftEvaluateCondition, rest) =
        condition.partition(_.references subsetOf left.outputSet)
    val (rightEvaluateCondition, commonCondition) =
        rest.partition(_.references subsetOf right.outputSet)

    (leftEvaluateCondition, rightEvaluateCondition, commonCondition)
  }

  def apply(plan: LogicalPlan): LogicalPlan = plan transform {
    // push the where condition down into join filter
    case f @ Filter(filterCondition, Join(left, right, joinType, joinCondition)) =>
      val (leftFilterConditions, rightFilterConditions, commonFilterCondition) =
        split(splitConjunctivePredicates(filterCondition), left, right)

      joinType match {
        case Inner =>
          // push down the single side `where` condition into respective sides
          val newLeft = leftFilterConditions.
            reduceLeftOption(And).map(Filter(_, left)).getOrElse(left)
          val newRight = rightFilterConditions.
            reduceLeftOption(And).map(Filter(_, right)).getOrElse(right)
          val newJoinCond = (commonFilterCondition ++ joinCondition).reduceLeftOpti
on(And)

          Join(newLeft, newRight, Inner, newJoinCond)
        case RightOuter =>
          // push down the right side only `where` condition
          val newLeft = left
          val newRight = rightFilterConditions.
            reduceLeftOption(And).map(Filter(_, right)).getOrElse(right)
          val newJoinCond = joinCondition
```

```
          val newJoin = Join(newLeft, newRight, RightOuter, newJoinCond)

          (leftFilterConditions ++ commonFilterCondition).
            reduceLeftOption(And).map(Filter(_, newJoin)).getOrElse(newJoin)
        case _ @ (LeftOuter | LeftSemi) =>
          // push down the left side only `where` condition
          val newLeft = leftFilterConditions.
            reduceLeftOption(And).map(Filter(_, left)).getOrElse(left)
          val newRight = right
          val newJoinCond = joinCondition
          val newJoin = Join(newLeft, newRight, joinType, newJoinCond)

          (rightFilterConditions ++ commonFilterCondition).
            reduceLeftOption(And).map(Filter(_, newJoin)).getOrElse(newJoin)
        case FullOuter => f // DO Nothing for Full Outer Join
      }

    // push down the join filter into sub query scanning if applicable
    case f @ Join(left, right, joinType, joinCondition) =>
      val (leftJoinConditions, rightJoinConditions, commonJoinCondition) =
        split(joinCondition.map(splitConjunctivePredicates).getOrElse(Nil), left, r
ight)

      joinType match {
        case Inner =>
          // push down the single side only join filter for both sides sub queries
          val newLeft = leftJoinConditions.
            reduceLeftOption(And).map(Filter(_, left)).getOrElse(left)
          val newRight = rightJoinConditions.
            reduceLeftOption(And).map(Filter(_, right)).getOrElse(right)
          val newJoinCond = commonJoinCondition.reduceLeftOption(And)

          Join(newLeft, newRight, Inner, newJoinCond)
        case RightOuter =>
          // push down the left side only join filter for left side sub query
          val newLeft = leftJoinConditions.
            reduceLeftOption(And).map(Filter(_, left)).getOrElse(left)
          val newRight = right
          val newJoinCond = (rightJoinConditions ++ commonJoinCondition).reduceLeft
Option(And)

          Join(newLeft, newRight, RightOuter, newJoinCond)
        case _ @ (LeftOuter | LeftSemi) =>
          // push down the right side only join filter for right sub query
          val newLeft = left
          val newRight = rightJoinConditions.
            reduceLeftOption(And).map(Filter(_, right)).getOrElse(right)
          val newJoinCond = (leftJoinConditions ++ commonJoinCondition).reduceLeftO
ption(And)

          Join(newLeft, newRight, joinType, newJoinCond)
```

```
        case FullOuter => f
      }
  }
}
```

ColumnPruning

列裁剪用的比较多，就是减少不必要select的某些列。列裁剪在3种地方可以用：

1. 在聚合操作中，可以做列裁剪
2. 在join操作中，左右孩子可以做列裁剪
3. 合并相邻的Project的列

```
/**
 * Attempts to eliminate the reading of unneeded columns from the query plan using
the following
 * transformations:
 *
 *  - Inserting Projections beneath the following operators:
 *   - Aggregate
 *   - Project <- Join
 *   - LeftSemiJoin
 *  - Collapse adjacent projections, performing alias substitution.
 */
object ColumnPruning extends Rule[LogicalPlan] {
  def apply(plan: LogicalPlan): LogicalPlan = plan transform {
    // Eliminate attributes that are not needed to calculate the specified aggregat
es.
    case a @ Aggregate(_, _, child) if (child.outputSet -- a.references).nonEmpty =
>
      a.copy(child = Project(a.references.toSeq, child))

    // Eliminate unneeded attributes from either side of a Join.
    case Project(projectList, Join(left, right, joinType, condition)) =>
      // Collect the list of all references required either above or to evaluate th
e condition.
      val allReferences: AttributeSet =
        AttributeSet(
          projectList.flatMap(_.references.iterator)) ++
          condition.map(_.references).getOrElse(AttributeSet(Seq.empty))

      /** Applies a projection only when the child is producing unnecessary attribu
tes */
      def pruneJoinChild(c: LogicalPlan) = prunedChild(c, allReferences)

      Project(projectList, Join(pruneJoinChild(left), pruneJoinChild(right), joinTy
pe, condition))

    // Eliminate unneeded attributes from right side of a LeftSemiJoin.
    case Join(left, right, LeftSemi, condition) =>
```

```
      // Collect the list of all references required to evaluate the condition.
      val allReferences: AttributeSet =
        condition.map(_.references).getOrElse(AttributeSet(Seq.empty))

      Join(left, prunedChild(right, allReferences), LeftSemi, condition)

    // Combine adjacent Projects.
    case Project(projectList1, Project(projectList2, child)) =>
      // Create a map of Aliases to their values from the child projection.
      // e.g., 'SELECT ... FROM (SELECT a + b AS c, d ...)' produces Map(c -> Alias
(a + b, c)).
      val aliasMap = projectList2.collect {
        case a @ Alias(e, _) => (a.toAttribute: Expression, a)
      }.toMap

      // Substitute any attributes that are produced by the child projection, so th
at we safely
      // eliminate it.
      // e.g., 'SELECT c + 1 FROM (SELECT a + b AS C ...' produces 'SELECT a + b +
1 ...'
      // TODO: Fix TransformBase to avoid the cast below.
      val substitutedProjection = projectList1.map(_.transform {
        case a if aliasMap.contains(a) => aliasMap(a)
      }).asInstanceOf[Seq[NamedExpression]]

      Project(substitutedProjection, child)

    // Eliminate no-op Projects
    case Project(projectList, child) if child.output == projectList => child
  }

  /** Applies a projection only when the child is producing unnecessary attributes
*/
  private def prunedChild(c: LogicalPlan, allReferences: AttributeSet) =
    if ((c.outputSet -- allReferences.filter(c.outputSet.contains)).nonEmpty) {
      Project(allReferences.filter(c.outputSet.contains).toSeq, c)
    } else {
      c
    }
}
```

总结

在log4j.properies里面设置 `log4j.rootCategory=TRACE, console` 可以将Catalyst详细的优化过程打印到Console中。

```
sql( s"""
        |SELECT name
        |FROM (SELECT name, age FROM rddTable) p
        |WHERE p.age >= 13 AND p.age <= 19
        |""".stripMargin).queryExecution
```

拿上面这句sql查询为例，它会经过以下几个优化过程。

1: Analyzer阶段 (Batch Resolution)

```
=== Applying Rule org.apache.spark.sql.catalyst.analysis.Analyzer$ResolveRelations
===
 'Project ['name]                                    'Project ['name]
  'Filter (('p.age. >= 13) && ('p.age. <= 19))    'Filter (('p.age. >= 13) && ('p.a
ge. <= 19))
   'Subquery p                                        'Subquery p
    'Project ['name,'age]                              'Project ['name,'age]
!    'UnresolvedRelation None, rddTable, None        Subquery rddTable
!                                                      LogicalRDD [name#0,age#1], Ma
pPartitionsRDD[4] at mapPartitions at ExistingRDD.scala:36

=== Applying Rule org.apache.spark.sql.catalyst.analysis.Analyzer$ResolveReferences
 ===
!'Project ['name]
            Project [name#0]
! 'Filter (('p.age. >= 13) && ('p.age. <= 19))
             Filter ((age#1 >= 13) && (age#1 <= 19))
!  'Subquery p
              Subquery p
!   'Project ['name,'age]
               Project [name#0,age#1]
     Subquery rddTable
                Subquery rddTable
      LogicalRDD [name#0,age#1], MapPartitionsRDD[4] at mapPartitions at ExistingRD
D.scala:36        LogicalRDD [name#0,age#1], MapPartitionsRDD[4] at mapPartitions a
t ExistingRDD.scala:36

=== Result of Batch Resolution ===
!'Project ['name]                                   Project [name#0]
! 'Filter (('p.age. >= 13) && ('p.age. <= 19))    Filter ((age#1 >= 13) && (age#1 <
= 19))
!  'Subquery p                                       Subquery p
!   'Project ['name,'age]                             Project [name#0,age#1]
```

```
!    'UnresolvedRelation None, rddTable, None        Subquery rddTable
!                                                     LogicalRDD [name#0,age#1], Ma
pPartitionsRDD[4] at mapPartitions at ExistingRDD.scala:36
```

2: Analyzer阶段 (Batch AnalysisOperators)

```
=== Applying Rule org.apache.spark.sql.catalyst.analysis.EliminateAnalysisOperators
 ===
 Project [name#0]
             Project [name#0]
  Filter ((age#1 >= 13) && (age#1 <= 19))
              Filter ((age#1 >= 13) && (age#1 <= 19))
!  Subquery p
               Project [name#0,age#1]
!   Project [name#0,age#1]
                LogicalRDD [name#0,age#1], MapPartitionsRDD[4] at mapPartitions at
ExistingRDD.scala:36
!    Subquery rddTable
!     LogicalRDD [name#0,age#1], MapPartitionsRDD[4]at mapPartitions at ExistingRDD
.scala:36

=== Result of Batch AnalysisOperators ===
!'Project ['name]                                  Project [name#0]
! 'Filter (('p.age. >= 13) && ('p.age. <= 19))    Filter ((age#1 >= 13) && (age#1 <
= 19))
!  'Subquery p                                      Project [name#0,age#1]
!   'Project ['name,'age]                            LogicalRDD [name#0,age#1], MapP
artitionsRDD[4] at mapPartitions at ExistingRDD.scala:36
!    'UnresolvedRelation None, rddTable, None
```

3: Optimizer阶段 (Batch Filter Pushdown)

```
=== Applying Rule org.apache.spark.sql.catalyst.optimizer.PushPredicateThroughProje
ct ===
 Project [name#0]
           Project [name#0]
! Filter ((age#1 >= 13) && (age#1 <= 19))
            Project [name#0,age#1]
!  Project [name#0,age#1]
             Filter ((age#1 >= 13) && (age#1 <= 19))
    LogicalRDD [name#0,age#1], MapPartitionsRDD[4] at mapPartitions at ExistingRDD.
scala:36      LogicalRDD [name#0,age#1], MapPartitionsRDD[4] at mapPartitions at Ex
istingRDD.scala:36

=== Applying Rule org.apache.spark.sql.catalyst.optimizer.ColumnPruning ===
 Project [name#0]
           Project [name#0]
! Project [name#0,age#1]
            Filter ((age#1 >= 13) && (age#1 <= 19))
```

```
!  Filter ((age#1 >= 13) && (age#1 <= 19))
             LogicalRDD [name#0,age#1], MapPartitionsRDD[4] at mapPartitions at Exi
stingRDD.scala:36
!   LogicalRDD [name#0,age#1], MapPartitionsRDD[4] at mapPartitions at ExistingRDD.
scala:36

=== Result of Batch Filter Pushdown ===
 Project [name#0]
           Project [name#0]
  Filter ((age#1 >= 13) && (age#1 <= 19))
            Filter ((age#1 >= 13) && (age#1 <= 19))
!  Project [name#0,age#1]
             LogicalRDD [name#0,age#1], MapPartitionsRDD[4] at mapPartitions at Exi
stingRDD.scala:36
!   LogicalRDD [name#0,age#1], MapPartitionsRDD[4] at mapPartitions at ExistingRDD.
scala:36
```

Catalyst优化器里面的各种规则互相配合，最后把一个Unresolved Logical Plan优化为一个Optimized Logical Plan。

SparkSQL组件解析

本篇将详细地介绍一下SparkSQL的一些关键组件。

SqlParser

SqlParser是一个SQL语言的解析器，功能是将SQL语句解析成Unresolved Logical Plan。

Scala的词法和语法解析器

SqlParser使用的是Scala提供的StandardTokenParsers和PackratParsers，分别用于词法解析和语法解析，首先为了理解对SQL语句解析过程的理解，先来看看下面这个简单数字表达式的解析过程。

```
import scala.util.parsing.combinator.PackratParsers
import scala.util.parsing.combinator.syntactical._

object MyLexical extends StandardTokenParsers with PackratParsers{

  //定义分割符
  lexical.delimiters ++= List(".", ";", "+", "-", "*")
  //定义表达式，支持加，减，乘
  lazy val expr: PackratParser[Int] = plus | minus | multi
  //加法表示式的实现
  lazy val plus: PackratParser[Int] = num ~ "+" ~ num ^^ { case n1 ~ "+" ~ n2 => n1
.toInt + n2.toInt}
  //减法表达式的实现
  lazy val minus: PackratParser[Int] = num ~ "-" ~ num ^^ { case n1 ~ "-" ~ n2 => n
1.toInt - n2.toInt}
  //乘法表达式的实现
  lazy val multi: PackratParser[Int] = num ~ "*" ~ num ^^ { case n1 ~ "*" ~ n2 => n
1.toInt * n2.toInt}
  lazy val num = numericLit

  def parse(input: String) = {
    //定义词法读入器myread，并将扫描头放置在input的首位
    val myread = new PackratReader(new lexical.Scanner(input))
    print("处理表达式 " + input)
    phrase(expr)(myread) match {
      case Success(result, _) => println(" Success!"); println(result); Some(result
)
      case n => println(n); println("Err!"); None
    }
  }

  def main(args: Array[String]) {
    val prg = "6 * 3" :: "24-/*aaa*/4" :: "a+5" :: "21/3" :: Nil
    prg.map(parse)
  }
}
```

运行结果：

```
处理表达式 6 * 3 Success!     //lexical对空格进行了处理，得到6*3
18     //6*3符合乘法表达式，调用n1.toInt * n2.toInt，得到结果并返回
处理表达式 24-/*aaa*/4 Success!  //lexical对注释进行了处理，得到20-4
20    //20-4符合减法表达式，调用n1.toInt - n2.toInt，得到结果并返回
处理表达式 a+5[1.1] failure: number expected
      //lexical在解析到a，发现不是整数型，故报错误位置和内容
a+5
^
Err!
处理表达式 21/3[1.3] failure: ``*'' expected but ErrorToken(illegal character) found
      //lexical在解析到/，发现不是分割符，故报错误位置和内容
21/3
  ^
Err!
```

在运行的时候，首先对表达式 6 * 3 进行解析，词法读入器myread将扫描头置于6的位置；当phrase()函数使用定义好的数字表达式expr处理6 * 3的时候，每读入一个词就和expr进行匹配，如读入6*3和expr进行匹配，先匹配表达式plus，*和+匹配不上；就继续匹配表达式minus，*和-匹配不上；就继续匹配表达式multi，这次匹配上了，等读入3的时候，因为3是num类型，就调用调用n1.toInt * n2.toInt进行计算。

注意，这里的expr、plus、minus、multi、num都是表达式，|、~、^^是复合因子，表达式和复合因子可以组成一个新的表达式，如plus（num ~ "+" ~ num ^^ { case n1 ~ "+" ~ n2 => n1.toInt + n2.toInt}）就是一个由num、+、num、函数构成的复合表达式；而expr（plus | minus | multi）是由plus、minus、multi构成的复合表达式；复合因子的含义定义在类scala/util/parsing/combinator/Parsers.scala，下面是几个常用的复合因子：

表达式	含义
p ~ q	p成功，才会q，放回p,q的结果
p ~> q	p成功，才会q，返回q的结果
p <~ q	p成功，才会q，返回p的结果
p 或 q	p失败则q，返回第一个成功的结果
p ^^ f	如果p成功，将函数f应用到p的结果上
p ^? f	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

针对上面的6 * 3使用的是multi表达式

```
(num ~ "\*" ~ num ^^ { case n1 ~ "\*" ~ n2 => n1.toInt \* n2.toInt})
```

其含义就是：num后跟*再跟num，如果满足就将使用函数n1.toInt * n2.toInt。

SqlParser入口

SqlParser的入口在SqlContext的sql()函数，该函数会调用parserSql并返回SchemaRDD。

```
/**
   * Executes a SQL query using Spark, returning the result as a SchemaRDD.  The di
alect that is
   * used for SQL parsing can be configured with 'spark.sql.dialect'.
   *
   * @group userf
   */
  def sql(sqlText: String): SchemaRDD = {
    if (dialect == "sql") {
      new SchemaRDD(this, parseSql(sqlText))
    } else {
      sys.error(s"Unsupported SQL dialect: $dialect")
    }
  }
```

parseSql会调用ddlParser，如果不成功就调用sqlParser。接着sqlParser会调用SparkSQLParser，并且把catalyst.SqlParser传递进去。

```
 protected[sql] def parseSql(sql: String): LogicalPlan = {
    ddlParser(sql).getOrElse(sqlParser(sql))
  }

    protected[sql] val sqlParser = {
    val fallback = new catalyst.SqlParser
    new catalyst.SparkSQLParser(fallback(_))
  }
```

SparkSQLParser的功能是解析SparkSQL特有的语法，例如cache，lazy等。SparkSQLParser会首先按照自己定义的词法和语法进行解析，当遇到以下两种情况的时候，会调用传递进来的catalyst.SqlParser:

1. Cache关键字后面的语法解析
2. 其他SparkSQLParser未定义的语法

```
private[sql] class SparkSQLParser(fallback: String => LogicalPlan) extends Abstract
SparkSQLParser {
...
private lazy val cache: Parser[LogicalPlan] =
    CACHE ~> LAZY.? ~ (TABLE ~> ident) ~ (AS ~> restInput).? ^^ {
      case isLazy ~ tableName ~ plan =>
        CacheTableCommand(tableName, plan.map(fallback), isLazy.isDefined)
    }
...
  private lazy val others: Parser[LogicalPlan] =
    wholeInput ^^ {
      case input => fallback(input)
    }
}
```

SqlParser

SqlParser继承自AbstractSparkSQLParser，而AbstractSparkSQLParser继承自StandardTokenParsers和 PackratParsers。SqlParser中定义了SQL语言的词法和语法规则：

词法： SqlParser首先定义了一堆Keyword，然后通过反射机制把这些Keyword全部加到一个reservedWords的集合当中，最后把这些关键字加到SqlLexical中。SqlLexical中除了定义关键字以外，还定义了分隔符。

```
class SqlParser extends AbstractSparkSQLParser {
...
  protected val ABS = Keyword("ABS")
  protected val ALL = Keyword("ALL")
  protected val AND = Keyword("AND")
  protected val APPROXIMATE = Keyword("APPROXIMATE")
  protected val AS = Keyword("AS")
  protected val ASC = Keyword("ASC")
  protected val AVG = Keyword("AVG")
  protected val BETWEEN = Keyword("BETWEEN")
  ...
  protected val reservedWords =
    this
      .getClass
      .getMethods
      .filter(_.getReturnType == classOf[Keyword])
      .map(_.invoke(this).asInstanceOf[Keyword].str)
  ...
  override val lexical = new SqlLexical(reservedWords)
  ...
}
```

```
class SqlLexical(val keywords: Seq[String]) extends StdLexical {
  ...
  reserved ++= keywords.flatMap(w => allCaseVersions(w))

  delimiters += (
    "@", "*", "+", "-", "<", "=", "<>", "!=", "<=", ">=", ">", "/", "(", ")",
    ",", ";", "%", "{", "}", ":", "[", "]", ".", "&", "|", "^", "~", "<=>"
  )
  ...
}
```

语法： sql语法的根节点是 `val start: Parser[LogicalPlan]` ，语法树的返回类型是 `LogicalPlan` 。

```
class SqlParser extends AbstractSparkSQLParser {
  ...
```

```
  protected lazy val start: Parser[LogicalPlan] =
    ( 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, q
2)}
      | UNION ~ DISTINCT.? ^^^ { (q1: LogicalPlan, q2: LogicalPlan) => Distinct(Uni
on(q1, q2)) }
      )
    | insert
    )

  protected lazy val select: Parser[LogicalPlan] =
    SELECT ~> DISTINCT.? ~
      repsep(projection, ",") ~
      (FROM   ~> relations).? ~
      (WHERE  ~> expression).? ~
      (GROUP  ~  BY ~> rep1sep(expression, ",")).? ~
      (HAVING ~> expression).? ~
      (ORDER  ~  BY ~> ordering).? ~
      (LIMIT  ~> expression).? ^^ {
        case d ~ p ~ r ~ f ~ g ~ h ~ o ~ l  =>
          val base = r.getOrElse(NoRelation)
          val withFilter = f.map(Filter(_, base)).getOrElse(base)
          val withProjection = g
            .map(Aggregate(_, assignAliases(p), withFilter))
            .getOrElse(Project(assignAliases(p), withFilter))
          val withDistinct = d.map(_ => Distinct(withProjection)).getOrElse(withPro
jection)
          val withHaving = h.map(Filter(_, withDistinct)).getOrElse(withDistinct)
          val withOrder = o.map(Sort(_, withHaving)).getOrElse(withHaving)
          val withLimit = l.map(Limit(_, withOrder)).getOrElse(withOrder)
          withLimit
      }

  protected lazy val insert: Parser[LogicalPlan] =
    INSERT ~> OVERWRITE.? ~ (INTO ~> relation) ~ select ^^ {
      case o ~ r ~ s => InsertIntoTable(r, Map.empty[String, Option[String]], s, o.
isDefined)
    }
    ...
}
```

AbstractSparkSQLParser

sql真正的解析是在AbstractSparkSQLParser中进行的，AbstractSparkSQLParser继承自StandardTokenParsers和PackratParsers。解析功能的核心代码就是：`phrase(start)(new lexical.Scanner(input))`。可以看得出来，该语句就是调用phrase()函数，使用SQL语法表达式start，对词法读入器lexical读入的SQL语句进行解析，其中

1. 词法分析器lexical定义在SqlParser中 `override val lexical = new SqlLexical(reservedWords)`
2. 语法分析器start定义在SqlParser中 `protected lazy val start: Parser[LogicalPlan] =...`

```
abstract class AbstractSparkSQLParser
  extends StandardTokenParsers with PackratParsers {

  def apply(input: String): LogicalPlan = phrase(start)(new lexical.Scanner(input))
 match {
    case Success(plan, _) => plan
    case failureOrError => sys.error(failureOrError.toString)
  }
 ...
}
```

Physical Plan

物理计划是Spark SQL执行Spark job的前置，也是最后一道计划。

SparkPlanner

Optimizer接受输入的Analyzed Logical Plan后，会由SparkPlanner来对Optimized Logical Plan进行转换，生成Physical Plan。

```
protected abstract class QueryExecution {
    lazy val analyzed = ExtractPythonUdfs(analyzer(logical))
    lazy val withCachedData = useCachedData(analyzed)
    lazy val optimizedPlan = optimizer(withCachedData)
    lazy val sparkPlan = {
      SparkPlan.currentContext.set(self)
      planner(optimizedPlan).next()
    }
    ...
}
```

SparkPlanner的apply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。

```
protected[sql] class SparkPlanner extends SparkStrategies {
    ...
    val strategies: Seq[Strategy] =
      extraStrategies ++ (
      CommandStrategy(self) ::
      DataSourceStrategy ::
      TakeOrdered ::
      HashAggregation ::
      LeftSemiJoin ::
      HashJoin ::
      InMemoryScans ::
      ParquetOperations ::
      BasicOperators ::
      CartesianProduct ::
      BroadcastNestedLoopJoin :: Nil)
    ...
}
```

QueryPlanner 是SparkPlanner的基类，定义了一系列的关键点，如Strategy，planLater和apply。

```
abstract class QueryPlanner[PhysicalPlan <: TreeNode[PhysicalPlan]] {
```

```
  /** A list of execution strategies that can be used by the planner */
  def strategies: Seq[GenericStrategy[PhysicalPlan]]

  /**
   * Returns a placeholder for a physical plan that executes `plan`. This placehold
er will be
   * filled in automatically by the QueryPlanner using the other execution strategi
es that are
   * available.
   */
  protected def planLater(plan: LogicalPlan) = apply(plan).next()

  def apply(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
  }
}
```

prepareForExecution

Spark Plan是Catalyst里经过所有Strategies apply 的最终的物理执行计划的抽象类，它只是用来执行spark job的。

```
  lazy val executedPlan: SparkPlan = prepareForExecution(sparkPlan)
```

prepareForExecution其实是一个RuleExecutor[SparkPlan]，当然这里的Rule就是SparkPlan了。

```
  protected[sql] val prepareForExecution = new RuleExecutor[SparkPlan] {
    val batches =
      Batch("Add exchange", Once, AddExchange(self)) :: Nil
  }
```

prepareForExecution里面只有一条规则：AddExchange。主要工作是检查是否有不匹配的partition类型，如果不兼容就增加一个Exchange节点，用来重新分区。

```
private[sql] case class AddExchange(sqlContext: SQLContext) extends Rule[SparkPlan]
 {
  ...
  def apply(plan: SparkPlan): SparkPlan = plan.transformUp {
    case operator: SparkPlan =>
      // Check if every child's outputPartitioning satisfies the corresponding
      // required data distribution.
      def meetsRequirements =
        !operator.requiredChildDistribution.zip(operator.children).map {
          case (required, child) =>
```

```
            val valid = child.outputPartitioning.satisfies(required)
            logDebug(
              s"${if (valid) "Valid" else "Invalid"} distribution," +
                s"required: $required current: ${child.outputPartitioning}")
            valid
        }.exists(!_)

      // Check if outputPartitionings of children are compatible with each other.
      // It is possible that every child satisfies its required data distribution
      // but two children have incompatible outputPartitionings. For example,
      // A dataset is range partitioned by "a.asc" (RangePartitioning) and another
      // dataset is hash partitioned by "a" (HashPartitioning). Tuples in these two
      // datasets are both clustered by "a", but these two outputPartitionings are
not
      // compatible.
      // TODO: ASSUMES TRANSITIVITY?
      def compatible =
        !operator.children
          .map(_.outputPartitioning)
          .sliding(2)
          .map {
            case Seq(a) => true
            case Seq(a,b) => a compatibleWith b
          }.exists(!_)

      // Check if the partitioning we want to ensure is the same as the child's out
put
      // partitioning. If so, we do not need to add the Exchange operator.
      def addExchangeIfNecessary(partitioning: Partitioning, child: SparkPlan) =
        if (child.outputPartitioning != partitioning) Exchange(partitioning, child)
 else child

      if (meetsRequirements && compatible) {
        operator
      } else {
        // At least one child does not satisfies its required data distribution or
        // at least one child's outputPartitioning is not compatible with another c
hild's
        // outputPartitioning. In this case, we need to add Exchange operators.
        val repartitionedChildren = operator.requiredChildDistribution.zip(operator
.children).map {
          case (AllTuples, child) =>
            addExchangeIfNecessary(SinglePartition, child)
          case (ClusteredDistribution(clustering), child) =>
            addExchangeIfNecessary(HashPartitioning(clustering, numPartitions), chi
ld)
          case (OrderedDistribution(ordering), child) =>
            addExchangeIfNecessary(RangePartitioning(ordering, numPartitions), chil
d)
          case (UnspecifiedDistribution, child) => child
          case (dist, _) => sys.error(s"Don't know how to ensure $dist")
```

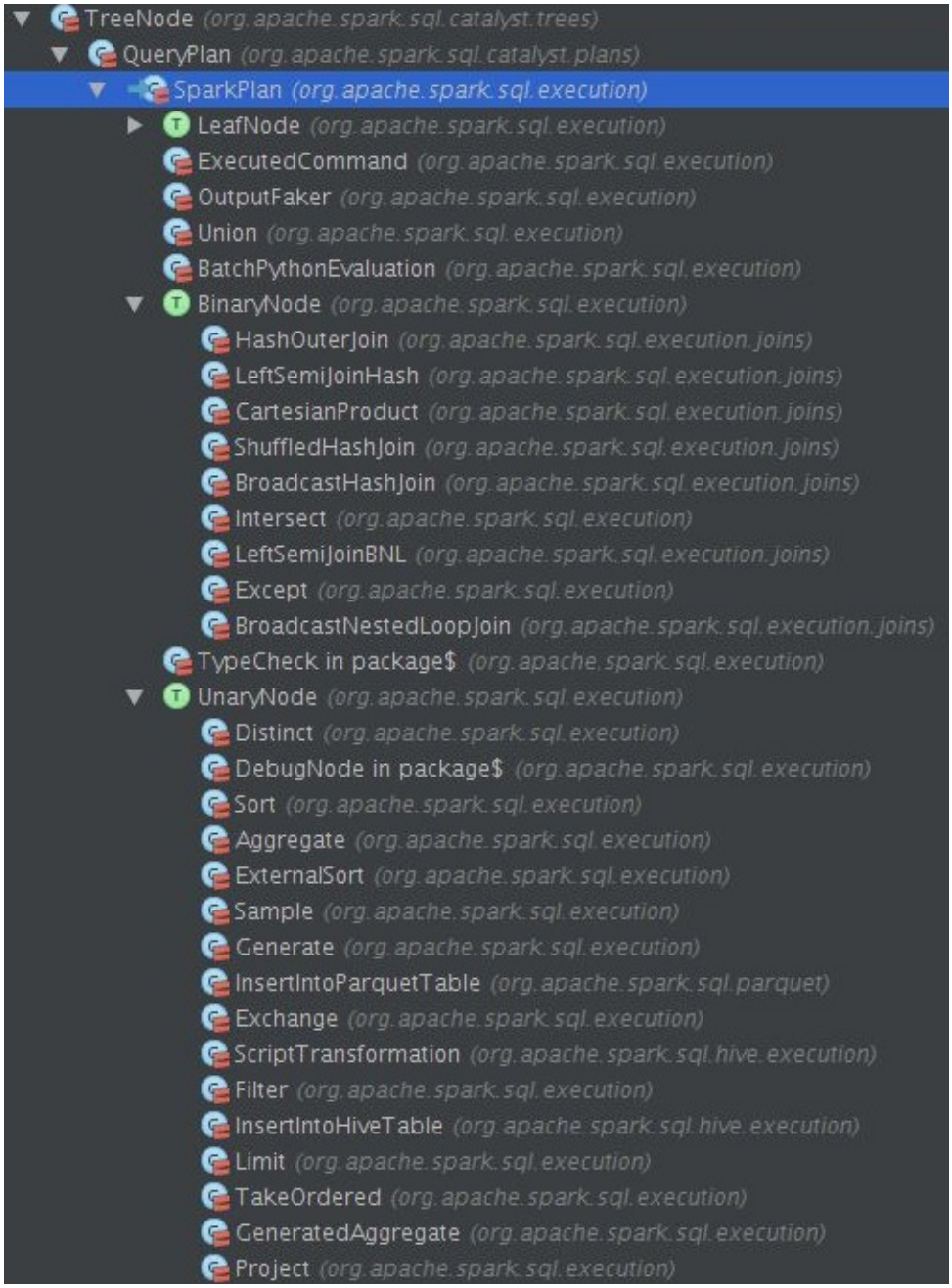
```
        }
        operator.withNewChildren(repartitionedChildren)
      }
  }
}
```

Spark Plan

Spark Plan是SparkSQL中的Physical Plan。它继承自Query Plan[Spark Plan]，里面定义了partition，requiredChildDistribution以及spark sql启动执行的execute方法。

```
abstract class SparkPlan extends QueryPlan[SparkPlan] with Logging with Serializabl
e {
  self: Product =>

  /** Specifies how data is partitioned across different nodes in the cluster. */
  def outputPartitioning: Partitioning = UnknownPartitioning(0) // TODO: WRONG WIDT
H!

  /** Specifies any partition requirements on the input data for this operator. */
  def requiredChildDistribution: Seq[Distribution] =
    Seq.fill(children.size)(UnspecifiedDistribution)

  /**
   * Runs this query returning the result as an RDD.
   */
  def execute(): RDD[Row]
  ...
}
```

目前SparkSQL中实现了一下十几种不同的Spark Plan，下面介绍几个比较重要的Spark Plan。



PhysicalRDD

当向SqlContext注册一个SchemaRDD时，就会生成一个PhysicalRDD，表示已经存在的RDD，不需要额外再去计算。

```
case class PhysicalRDD(output: Seq[Attribute], rdd: RDD[Row]) extends LeafNode {
  override def execute() = rdd
}
```

InMemoryColumnarTableScan

当使用cache时，就用生成InMemoryColumnarTableScan，内存列存储就是在这个类里面实现的。

```
private[sql] case class InMemoryColumnarTableScan(
    attributes: Seq[Attribute],
    predicates: Seq[Expression],
    relation: InMemoryRelation)
  extends LeafNode {
  ...
}
```

ParquetTableScan

读取Parquet类型数据的实现。

```
case class ParquetTableScan(
    attributes: Seq[Attribute],
    relation: ParquetRelation,
    columnPruningPred: Seq[Expression])
  extends LeafNode {
  ...
}
```

CacheTableCommand

Cache的物理执行实现。

```
case class CacheTableCommand(
    tableName: String,
    plan: Option[LogicalPlan],
    isLazy: Boolean)
  extends LeafNode with Command {

  override protected lazy val sideEffectResult = {
    import sqlContext._

    plan.foreach(_.registerTempTable(tableName))
    cacheTable(tableName)

    if (!isLazy) {
      // Performs eager caching
      table(tableName).count()
    }

    Seq.empty[Row]
  }

  override def output: Seq[Attribute] = Seq.empty
}
```

ExecutedCommand

执行传递进来的RunnableCommand，返回执行的结果。

```
case class ExecutedCommand(cmd: RunnableCommand) extends SparkPlan {
  /**
   * A concrete command should override this lazy field to wrap up any side effects
 caused by the
   * command or any other computation that should be evaluated exactly once. The va
lue of this field
   * can be used as the contents of the corresponding RDD generated from the physic
al plan of this
   * command.
   *
   * The `execute()` method of all the physical command classes should reference `s
ideEffectResult`
   * so that the command can be executed eagerly right after the command query is c
reated.
   */
  protected[sql] lazy val sideEffectResult: Seq[Row] = cmd.run(sqlContext)

  override def output = cmd.output

  override def children = Nil

  override def executeCollect(): Array[Row] = sideEffectResult.toArray

  override def execute(): RDD[Row] = sqlContext.sparkContext.parallelize(sideEffect
Result, 1)
}
```

HashJoin

Join操作主要包含BroadcastHashJoin、LeftSemiJoinHash、ShuffledHashJoin均实现了HashJoin这个trait。

HashJoin这个trait的主要成员有：

- buildSide是左连接还是右连接，有一种基准的意思。
- leftKeys是左孩子的expressions, rightKeys是右孩子的expressions。
- left是左孩子物理计划，right是右孩子物理计划。
- buildSideKeyGenerator是一个Projection是根据传入的Row对象来计算buildSide的Expression的。
- streamSideKeyGenerator是一个MutableProjection是根据传入的Row对象来计算streamSide的Expression的。
- 这里buildSide如果是left的话，可以理解为buildSide是左表，那么去连接这个左表的右表就是streamSide。

HashJoin关键的操作是joinIterators，简单来说就是join两个表，把每个表看着Iterators[Row]. 方式：

1. 首先遍历buildSide，计算buildKeys然后利用一个HashMap，形成 (buildKeys, Iterators[Row])的格式。

2. 遍历StreamedSide，计算streamedKey，去HashMap里面去匹配key，来进行join
3. 最后生成一个joinRow，这个将两个row对接。

```
trait HashJoin {
  self: SparkPlan =>

  val leftKeys: Seq[Expression]
  val rightKeys: Seq[Expression]
  val buildSide: BuildSide
  val left: SparkPlan
  val right: SparkPlan

  protected lazy val (buildPlan, streamedPlan) = buildSide match {
    case BuildLeft => (left, right)
    case BuildRight => (right, left)
  }

  protected lazy val (buildKeys, streamedKeys) = buildSide match {
    case BuildLeft => (leftKeys, rightKeys)
    case BuildRight => (rightKeys, leftKeys)
  }

  override def output = left.output ++ right.output

  @transient protected lazy val buildSideKeyGenerator: Projection =
    newProjection(buildKeys, buildPlan.output)

  @transient protected lazy val streamSideKeyGenerator: () => MutableProjection =
    newMutableProjection(streamedKeys, streamedPlan.output)

  protected def hashJoin(streamIter: Iterator[Row], hashedRelation: HashedRelation)
: Iterator[Row] =
  {
    new Iterator[Row] {
      //left数据的当前行
      private[this] var currentStreamedRow: Row = _
      //right中key等于当前left数据的所有行
      private[this] var currentHashMatches: CompactBuffer[Row] = _
      //currentHashMatches访问到的当前Index
      private[this] var currentMatchPosition: Int = -1

      // Mutable per row objects.
      //使用mutable的row对象，减少GC压力
      private[this] val joinRow = new JoinedRow2

      private[this] val joinKeys = streamSideKeyGenerator()

      //先访问currentHashMatches，如果currentHashMatches没有数据了，从stream中取下一个
      override final def hasNext: Boolean =
        (currentMatchPosition != -1 && currentMatchPosition < currentHashMatches.si
```

```
ze) ||
          (streamIter.hasNext && fetchNext())

      //从currentHashMatches中取下一个
      override final def next() = {
        val ret = buildSide match {
          case BuildRight => joinRow(currentStreamedRow, currentHashMatches(current
MatchPosition))
          case BuildLeft => joinRow(currentHashMatches(currentMatchPosition), curre
ntStreamedRow)
        }
        currentMatchPosition += 1
        ret
      }

      private final def fetchNext(): Boolean = {
        currentHashMatches = null
        currentMatchPosition = -1

        while (currentHashMatches == null && streamIter.hasNext) {
          currentStreamedRow = streamIter.next() //从stream中取下一个
          if (!joinKeys(currentStreamedRow).anyNull) {
            //从hash map中取出所有key=joinKeys.currentValue的行
            currentHashMatches = hashedRelation.get(joinKeys.currentValue)
          }
        }

        if (currentHashMatches == null) {
          false
        } else {
          currentMatchPosition = 0
          true
        }
      }
    }
  }
}
```

HashOuterJoin

使用Shuffle+HashMap的方式进行Outer Join。具体步骤如下

1. 调用zipPartitions将两个rdd对应的partition数据放到一起
2. 在每个partition中，对两个数据分别建立两个HashMap
3. 根据Outer Join的类型(left, right, full)，生成对应的iterator

```
case class HashOuterJoin(
    leftKeys: Seq[Expression],
    rightKeys: Seq[Expression],
    joinType: JoinType,
```

```
    condition: Option[Expression],
    left: SparkPlan,
    right: SparkPlan) extends BinaryNode {
    ...
    override def execute() = {
    left.execute().zipPartitions(right.execute()) { (leftIter, rightIter) =>
      // TODO this probably can be replaced by external sort (sort merged join?)
      // Build HashMap for current partition in left relation
      val leftHashTable = buildHashTable(leftIter, newProjection(leftKeys, left.out
put))
      // Build HashMap for current partition in right relation
      val rightHashTable = buildHashTable(rightIter, newProjection(rightKeys, right
.output))
      val boundCondition =
        condition.map(newPredicate(_, left.output ++ right.output)).getOrElse((row:
 Row) => true)
      joinType match {
        case LeftOuter => leftHashTable.keysIterator.flatMap { key =>
          leftOuterIterator(key, leftHashTable.getOrElse(key, EMPTY_LIST),
            rightHashTable.getOrElse(key, EMPTY_LIST))
        }
        case RightOuter => rightHashTable.keysIterator.flatMap { key =>
          rightOuterIterator(key, leftHashTable.getOrElse(key, EMPTY_LIST),
            rightHashTable.getOrElse(key, EMPTY_LIST))
        }
        case FullOuter => (leftHashTable.keySet ++ rightHashTable.keySet).iterator.
flatMap { key =>
          fullOuterIterator(key,
            leftHashTable.getOrElse(key, EMPTY_LIST),
            rightHashTable.getOrElse(key, EMPTY_LIST))
        }
        case x => throw new Exception(s"HashOuterJoin should not take $x as the Joi
nType")
      }
    }
  }
}
```

LeftSemiJoinHash

将第二个表的join keys放到hash set中，遍历第一个表，从hash set中查找join key。

```
case class LeftSemiJoinHash(
    leftKeys: Seq[Expression],
    rightKeys: Seq[Expression],
    left: SparkPlan,
    right: SparkPlan) extends BinaryNode with HashJoin {

  override val buildSide = BuildRight
```

```
  override def requiredChildDistribution =
    ClusteredDistribution(leftKeys) :: ClusteredDistribution(rightKeys) :: Nil

  override def output = left.output

  override def execute() = {
    buildPlan.execute().zipPartitions(streamedPlan.execute()) { (buildIter, streamI
ter) =>
      val hashSet = new java.util.HashSet[Row]()
      var currentRow: Row = null

      // Create a Hash set of buildKeys
      while (buildIter.hasNext) {
        currentRow = buildIter.next()
        val rowKey = buildSideKeyGenerator(currentRow)
        if (!rowKey.anyNull) {
          val keyExists = hashSet.contains(rowKey)
          if (!keyExists) {
            hashSet.add(rowKey)
          }
        }
      }

      val joinKeys = streamSideKeyGenerator()
      streamIter.filter(current => {
        !joinKeys(current).anyNull && hashSet.contains(joinKeys.currentValue)
      })
    }
  }
}
```

ShuffledHashJoin

先Shuffle数据，再通过hash join的方式实现inner join。

```
case class ShuffledHashJoin(
    leftKeys: Seq[Expression],
    rightKeys: Seq[Expression],
    buildSide: BuildSide,
    left: SparkPlan,
    right: SparkPlan)
  extends BinaryNode with HashJoin {

  override def outputPartitioning: Partitioning = left.outputPartitioning

  override def requiredChildDistribution =
    ClusteredDistribution(leftKeys) :: ClusteredDistribution(rightKeys) :: Nil

  override def execute() = {
    buildPlan.execute().zipPartitions(streamedPlan.execute()) { (buildIter, streamI
```

```
ter) =>
      val hashed = HashedRelation(buildIter, buildSideKeyGenerator)
      hashJoin(streamIter, hashed)
    }
  }
}
```

BroadcastHashJoin

将其中一个数据broadcast出去，然后在另一个数据的每个partition进行hash join。

```
case class BroadcastHashJoin(
    leftKeys: Seq[Expression],
    rightKeys: Seq[Expression],
    buildSide: BuildSide,
    left: SparkPlan,
    right: SparkPlan)
  extends BinaryNode with HashJoin {

  override def outputPartitioning: Partitioning = streamedPlan.outputPartitioning

  override def requiredChildDistribution =
    UnspecifiedDistribution :: UnspecifiedDistribution :: Nil

  @transient
  private val broadcastFuture = future {
    // Note that we use .execute().collect() because we don't want to convert data
to Scala types
    val input: Array[Row] = buildPlan.execute().map(_.copy()).collect()
    val hashed = HashedRelation(input.iterator, buildSideKeyGenerator, input.length
)
    sparkContext.broadcast(hashed)
  }

  override def execute() = {
    val broadcastRelation = Await.result(broadcastFuture, 5.minute)

    streamedPlan.execute().mapPartitions { streamedIter =>
      hashJoin(streamedIter, broadcastRelation.value)
    }
  }
}
```

直接调用rdd函数

- Intersect：rdd.intersection
- Except：rdd.subtract
- Sample：rdd.sample
- TakeOrdered：rdd.takeOrdered

直接调用SparkContext的函数

- Union：sparkContext.union

在rdd.mapPartitions中进行简单计算

- Distinct：使用HashSet类
- Sort: 调用SeqLike的sort函数
- Filter：调用iterator的filter函数
- ExternalSorter：使用ExternalSorter类
- Project：调用iterator的filter函数

Strategies

下面来看一下在生成物理计划中使用到的十几种strategy。

CommandStrategy

CommandStrategy是专门针对Command类型的Logical Plan，即set key = value 、 explain sql、cache table 这类操作

1. RunnableCommand：执行继承自RunnableCommand的命令，并将Seq[Row]转化为RDD。
2. SetCommand：设置SparkContext的参数
3. ExplainCommand：利用executed Plan打印出tree string
4. CacheTableCommand：将RDD以列式方式缓存到内存中
5. UncacheTableCommand：将缓存的RDD清除

```
case class CommandStrategy(context: SQLContext) extends Strategy {
    def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
      case r: RunnableCommand => ExecutedCommand(r) :: Nil
      case logical.SetCommand(kv) =>
        Seq(execution.SetCommand(kv, plan.output)(context))
      case logical.ExplainCommand(logicalPlan, extended) =>
        Seq(execution.ExplainCommand(logicalPlan, plan.output, extended)(context))
      case logical.CacheTableCommand(tableName, optPlan, isLazy) =>
        Seq(execution.CacheTableCommand(tableName, optPlan, isLazy))
      case logical.UncacheTableCommand(tableName) =>
        Seq(execution.UncacheTableCommand(tableName))
      case _ => Nil
    }
  }
```

DataSourceStrategy

根据不同的BaseRelation生产不同的PhysicalRDD。支持4种BaseRelation:

1. TableScan：默认的Scan策略
2. PrunedScan：列裁剪，不需要的列不会从外部数据源加载
3. PrunedFilterScan：在列裁剪的基础上加入Filter，在加载数据也的时候就进行过滤，而不是在客

户端请求返回时做Filter

4. CatalystScan：Catalyst的支持传入expressions来进行Scan，支持列裁剪和Filter。

```
private[sql] object DataSourceStrategy extends Strategy {
  def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
    case PhysicalOperation(projectList, filters, l @ LogicalRelation(t: CatalystSca
n)) =>
      pruneFilterProjectRaw(
        l,
        projectList,
        filters,
        (a, f) => t.buildScan(a, f)) :: Nil

    case PhysicalOperation(projectList, filters, l @ LogicalRelation(t: PrunedFilte
redScan)) =>
      pruneFilterProject(
        l,
        projectList,
        filters,
        (a, f) => t.buildScan(a, f)) :: Nil

    case PhysicalOperation(projectList, filters, l @ LogicalRelation(t: PrunedScan)
) =>
      pruneFilterProject(
        l,
        projectList,
        filters,
        (a, _) => t.buildScan(a)) :: Nil

    case l @ LogicalRelation(t: TableScan) =>
      execution.PhysicalRDD(l.output, t.buildScan()) :: Nil

    case _ => Nil
  }
...
}
```

TakeOrdered

如果有Limit和Sort操作将会使用TakeOrdered策略，返回一个TakeOrdered的Spark Plan。

```
 object TakeOrdered extends Strategy {
    def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
      case logical.Limit(IntegerLiteral(limit), logical.Sort(order, child)) =>
        execution.TakeOrdered(limit, order, planLater(child)) :: Nil
      case _ => Nil
    }
  }
```

HashAggregation

聚合操作可以映射为RDD的shuffle操作。

```
object HashAggregation extends Strategy {
    def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
      // Aggregations that can be performed in two phases, before and after the shu
ffle.

      // Cases where all aggregates can be codegened.
      case PartialAggregation(
             namedGroupingAttributes,
             rewrittenAggregateExpressions,
             groupingExpressions,
             partialComputation,
             child)
             if canBeCodeGened(
                  allAggregates(partialComputation) ++
                  allAggregates(rewrittenAggregateExpressions)) &&
               codegenEnabled =>
          execution.GeneratedAggregate(
            partial = false,
            namedGroupingAttributes,
            rewrittenAggregateExpressions,
            execution.GeneratedAggregate(
              partial = true,
              groupingExpressions,
              partialComputation,
              planLater(child))) :: Nil

      // Cases where some aggregate can not be codegened
      case PartialAggregation(
             namedGroupingAttributes,
             rewrittenAggregateExpressions,
             groupingExpressions,
             partialComputation,
             child) =>
        execution.Aggregate(
          partial = false,
          namedGroupingAttributes,
          rewrittenAggregateExpressions,
          execution.Aggregate(
            partial = true,
            groupingExpressions,
            partialComputation,
            planLater(child))) :: Nil

      case _ => Nil
    }
    ...
  }
```

LeftSemiJoin

如果Logical Plan里的Join是joinType为LeftSemi的话，就会执行这种策略，这里ExtractEquiJoinKeys是一个pattern定义在patterns.scala里，主要是做模式匹配用的。这里匹配只要是等值的join操作，都会封装为ExtractEquiJoinKeys对象，它会解析当前join，最后返回(joinType, rightKeys, leftKeys, condition, leftChild, rightChild)的格式。最后返回一个execution.LeftSemiJoinHash这个Spark Plan。

```
object LeftSemiJoin extends Strategy with PredicateHelper {
    def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
      // Find left semi joins where at least some predicates can be evaluated by ma
tching join keys
      case ExtractEquiJoinKeys(LeftSemi, leftKeys, rightKeys, condition, left, righ
t) =>
        val semiJoin = joins.LeftSemiJoinHash(
          leftKeys, rightKeys, planLater(left), planLater(right))
        condition.map(Filter(_, semiJoin)).getOrElse(semiJoin) :: Nil
      // no predicate can be evaluated by matching hash keys
      case logical.Join(left, right, LeftSemi, condition) =>
        joins.LeftSemiJoinBNL(planLater(left), planLater(right), condition) :: Nil
      case _ => Nil
    }
  }
```

HashJoin

HashJoin是我们最见的操作，innerJoin类型，里面提供了2种Spark Plan：BroadcastHashJoin 和 ShuffledHashJoin。 BroadcastHashJoin的实现是一种广播变量的实现方法，如果设置了spark.sql.join.broadcastTables这个参数的表就会用spark的Broadcast Variables方式先将一张表给查询出来，然后广播到各个机器中。ShuffledHashJoin是一种最传统的默认的join方式，会根据shuffle key进行shuffle的hash join。

```
object HashJoin extends Strategy with PredicateHelper {

    private[this] def makeBroadcastHashJoin(
        leftKeys: Seq[Expression],
        rightKeys: Seq[Expression],
        left: LogicalPlan,
        right: LogicalPlan,
        condition: Option[Expression],
        side: joins.BuildSide) = {
      val broadcastHashJoin = execution.joins.BroadcastHashJoin(
        leftKeys, rightKeys, side, planLater(left), planLater(right))
      condition.map(Filter(_, broadcastHashJoin)).getOrElse(broadcastHashJoin) :: N
il
    }

    def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
      case ExtractEquiJoinKeys(Inner, leftKeys, rightKeys, condition, left, right)
```

```
        if sqlContext.autoBroadcastJoinThreshold > 0 &&
           right.statistics.sizeInBytes <= sqlContext.autoBroadcastJoinThreshold =>
        makeBroadcastHashJoin(leftKeys, rightKeys, left, right, condition, joins.Bu
ildRight)

      case ExtractEquiJoinKeys(Inner, leftKeys, rightKeys, condition, left, right)
        if sqlContext.autoBroadcastJoinThreshold > 0 &&
           left.statistics.sizeInBytes <= sqlContext.autoBroadcastJoinThreshold =>
          makeBroadcastHashJoin(leftKeys, rightKeys, left, right, condition, joins.
BuildLeft)

      case ExtractEquiJoinKeys(Inner, leftKeys, rightKeys, condition, left, right)
=>
        val buildSide =
          if (right.statistics.sizeInBytes <= left.statistics.sizeInBytes) {
            joins.BuildRight
          } else {
            joins.BuildLeft
          }
        val hashJoin = joins.ShuffledHashJoin(
          leftKeys, rightKeys, buildSide, planLater(left), planLater(right))
        condition.map(Filter(_, hashJoin)).getOrElse(hashJoin) :: Nil

      case ExtractEquiJoinKeys(joinType, leftKeys, rightKeys, condition, left, righ
t) =>
        joins.HashOuterJoin(
          leftKeys, rightKeys, joinType, condition, planLater(left), planLater(righ
t)) :: Nil

      case _ => Nil
    }
  }
```

InMemoryScans

InMemoryScans主要是对InMemoryRelation这个Logical Plan操作。调用的其实是Spark Planner里的pruneFilterProject这个方法。

```
object InMemoryScans extends Strategy {
    def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
      case PhysicalOperation(projectList, filters, mem: InMemoryRelation) =>
        pruneFilterProject(
          projectList,
          filters,
          identity[Seq[Expression]], // All filters still need to be evaluated.
          InMemoryColumnarTableScan(_,  filters, mem)) :: Nil
      case _ => Nil
    }
  }
```

ParquetOperations

支持ParquetOperations的读写，插入Table等。

```
object ParquetOperations extends Strategy {
    def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
      // TODO: need to support writing to other types of files.  Unify the below co
de paths.
      case logical.WriteToFile(path, child) =>
        val relation =
          ParquetRelation.create(path, child, sparkContext.hadoopConfiguration, sql
Context)
        // Note: overwrite=false because otherwise the metadata we just created wil
l be deleted
        InsertIntoParquetTable(relation, planLater(child), overwrite = false) :: Ni
l
      case logical.InsertIntoTable(table: ParquetRelation, partition, child, overwr
ite) =>
        InsertIntoParquetTable(table, planLater(child), overwrite) :: Nil
      case PhysicalOperation(projectList, filters: Seq[Expression], relation: Parqu
etRelation) =>
        val prunePushedDownFilters =
          if (sqlContext.parquetFilterPushDown) {
            (predicates: Seq[Expression]) => {
              // Note: filters cannot be pushed down to Parquet if they contain mor
e complex
              // expressions than simple "Attribute cmp Literal" comparisons. Here
we remove all
              // filters that have been pushed down. Note that a predicate such as
"(A AND B) OR C"
              // can result in "A OR C" being pushed down. Here we are conservative
 in the sense
              // that even if "A" was pushed and we check for "A AND B" we still wa
nt to keep
              // "A AND B" in the higher-level filter, not just "B".
              predicates.map(p => p -> ParquetFilters.createFilter(p)).collect {
                case (predicate, None) => predicate
              }
            }
          } else {
            identity[Seq[Expression]] _
          }
        pruneFilterProject(
          projectList,
          filters,
          prunePushedDownFilters,
          ParquetTableScan(
            _,
            relation,
            if (sqlContext.parquetFilterPushDown) filters else Nil)) :: Nil
```

```
      case _ => Nil
    }
  }
```

BasicOperators

所有定义在org.apache.spark.sql.execution里的基本的Spark Plan，它们都在org.apache.spark.sql.execution包下basicOperators.scala内。有Project、Filter、Sample、Union、Limit、TakeOrdered、Sort、ExistingRdd。这些是基本元素，实现都相对简单，基本上都是RDD里的方法来实现的。

```
// Can we automate these 'pass through' operations?
  object BasicOperators extends Strategy {
    def numPartitions = self.numPartitions

    def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
      case logical.Distinct(child) =>
        execution.Distinct(partial = false,
          execution.Distinct(partial = true, planLater(child))) :: Nil

      case logical.Sort(sortExprs, child) if sqlContext.externalSortEnabled =>
        execution.ExternalSort(sortExprs, global = true, planLater(child)):: Nil
      case logical.Sort(sortExprs, child) =>
        execution.Sort(sortExprs, global = true, planLater(child)):: Nil

      case logical.SortPartitions(sortExprs, child) =>
        // This sort only sorts tuples within a partition. Its requiredDistribution
 will be
        // an UnspecifiedDistribution.
        execution.Sort(sortExprs, global = false, planLater(child)) :: Nil
      case logical.Project(projectList, child) =>
        execution.Project(projectList, planLater(child)) :: Nil
      case logical.Filter(condition, child) =>
        execution.Filter(condition, planLater(child)) :: Nil
      case logical.Aggregate(group, agg, child) =>
        execution.Aggregate(partial = false, group, agg, planLater(child)) :: Nil
      case logical.Sample(fraction, withReplacement, seed, child) =>
        execution.Sample(fraction, withReplacement, seed, planLater(child)) :: Nil
      case SparkLogicalPlan(alreadyPlanned) => alreadyPlanned :: Nil
      case logical.LocalRelation(output, data) =>
        val nPartitions = if (data.isEmpty) 1 else numPartitions
        PhysicalRDD(
          output,
          RDDConversions.productToRowRdd(sparkContext.parallelize(data, nPartitions
),
            StructType.fromAttributes(output))) :: Nil
      case logical.Limit(IntegerLiteral(limit), child) =>
        execution.Limit(limit, planLater(child)) :: Nil
      case Unions(unionChildren) =>
        execution.Union(unionChildren.map(planLater)) :: Nil
```

```
      case logical.Except(left, right) =>
        execution.Except(planLater(left), planLater(right)) :: Nil
      case logical.Intersect(left, right) =>
        execution.Intersect(planLater(left), planLater(right)) :: Nil
      case logical.Generate(generator, join, outer, _, child) =>
        execution.Generate(generator, join = join, outer = outer, planLater(child))
 :: Nil
      case logical.NoRelation =>
        execution.PhysicalRDD(Nil, singleRowRdd) :: Nil
      case logical.Repartition(expressions, child) =>
        execution.Exchange(HashPartitioning(expressions, numPartitions), planLater(
child)) :: Nil
      case e @ EvaluatePython(udf, child, _) =>
        BatchPythonEvaluation(udf, e.output, planLater(child)) :: Nil
      case LogicalRDD(output, rdd) => PhysicalRDD(output, rdd) :: Nil
      case _ => Nil
    }
  }
```

CartesianProduct

笛卡尔积的Join，有待过滤条件的Join。主要是利用RDD的cartesian实现的。

```
object CartesianProduct extends Strategy {
    def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
      case logical.Join(left, right, _, None) =>
        execution.joins.CartesianProduct(planLater(left), planLater(right)) :: Nil
      case logical.Join(left, right, Inner, Some(condition)) =>
        execution.Filter(condition,
          execution.joins.CartesianProduct(planLater(left), planLater(right))) :: N
il
      case _ => Nil
    }
  }
```

BroadcastNestedLoopJoin

BroadcastNestedLoopJoin可用于Left Outer， Right Outer， Full Outer这三种类型的join，Hash Join仅仅用于InnerJoin。

```
object BroadcastNestedLoopJoin extends Strategy {
    def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
      case logical.Join(left, right, joinType, condition) =>
        val buildSide =
          if (right.statistics.sizeInBytes <= left.statistics.sizeInBytes) {
            joins.BuildRight
          } else {
            joins.BuildLeft
          }
```

```
          joins.BroadcastNestedLoopJoin(
            planLater(left), planLater(right), buildSide, joinType, condition) :: Nil
        case _ => Nil
      }
    }
```

UDF

在sql语句中，除了可以使用+ - * /等表达式外，还可以使用用户定义的函数UDF。下面是SqlParser中对UDF的语法定义：

```
protected lazy val function: Parser[Expression] =
    ( SUM   ~> "(" ~> expression             <~ ")" ^^ { case exp => Sum(exp) }
    ...
    | ident ~ ("(" ~> repsep(expression, ",")) <~ ")" ^^
      { case udfName ~ exprs => UnresolvedFunction(udfName, exprs) }
    )
```

将SqlParser传入的udfName和exprs封装成一个叫 UnresolvedFunction的类，该类继承自Expression。只是这个Expression的dataType等一系列属性和eval计算方法均无法访问，强制访问会抛出异常，因为它没有被Resolved，只是一个载体。

```
case class UnresolvedFunction(name: String, children: Seq[Expression]) extends Expr
ession {
  override def dataType = throw new UnresolvedException(this, "dataType")
  override def foldable = throw new UnresolvedException(this, "foldable")
  override def nullable = throw new UnresolvedException(this, "nullable")
  override lazy val resolved = false

  // Unresolved functions are transient at compile time and don't get evaluated dur
ing execution.
  override def eval(input: Row = null): EvaluatedType =
    throw new TreeNodeException(this, s"No function to evaluate expression. type: $
{this.nodeName}")

  override def toString = s"'$name(${children.mkString(",")})"
}
```

在SqlContext中有一个functionRegistry对象，使用的是SimpleFunctionRegistry，用来存储用户定义的函数。

```
protected[sql] lazy val functionRegistry: FunctionRegistry = new SimpleFunctionRegi
stry
```

UDF注册

UDFRegistration

registerFunction是UDFRegistration下的方法，SQLContext现在实现了UDFRegistration这个trait，只要导入SQLContext，即可以使用udf功能。

```
class SQLContext(@transient val sparkContext: SparkContext)
  extends org.apache.spark.Logging
  with SQLConf
  with CacheManager
  with ExpressionConversions
  with UDFRegistration
  with Serializable
```

registerFunction接受一个name和 一个func，可以是Function1到Function22，即这个udf的参数只支持1-22个。

```
private[sql] trait UDFRegistration {
...
def registerFunction[T: TypeTag](name: String, func: Function1[_, T]): Unit = {
    def builder(e: Seq[Expression]) = ScalaUdf(func, ScalaReflection.schemaFor[T].d
ataType, e)
    functionRegistry.registerFunction(name, builder)
  }
  ...
  def registerFunction[T: TypeTag](name: String, func: Function22[_, _, _, _, _, _,
 _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, T]): Unit = {
    def builder(e: Seq[Expression]) = ScalaUdf(func, ScalaReflection.schemaFor[T].d
ataType, e)
    functionRegistry.registerFunction(name, builder)
  }
}
```

内部builder通过ScalaUdf来构造一个Expression，这里ScalaUdf继承自Expression，传入scala的function作为UDF的实现。

```
case class ScalaUdf(function: AnyRef, dataType: DataType, children: Seq[Expression]
)
  extends Expression {
  ...
}
```

SimpleFunctionRegistry

SimpleFunctionRegistry中使用HashMap存储用户定义的函数。

```
type FunctionBuilder = Seq[Expression] => Expression
```

```
class SimpleFunctionRegistry extends FunctionRegistry {
  val functionBuilders = new mutable.HashMap[String, FunctionBuilder]()

  def registerFunction(name: String, builder: FunctionBuilder) = {
    functionBuilders.put(name, builder)
```

```
  }

  override def lookupFunction(name: String, children: Seq[Expression]): Expression
= {
    functionBuilders(name)(children)
  }
}
```

UDF计算

UDF既然已经被封装为catalyst树里的一个Expression节点，那么计算的时候也就是计算ScalaUdf的eval方法。先通过Row和表达式计算function所需要的参数，最后通过反射调用function，来达到计算udf的目的。

```
case class ScalaUdf(function: AnyRef, dataType: DataType, children: Seq[Expression]
)
  extends Expression {

  override def eval(input: Row): Any = {
    val result = children.size match {
      case 0 => function.asInstanceOf[() => Any]()
      case 1 =>
        function.asInstanceOf[(Any) => Any](
          ScalaReflection.convertToScala(children(0).eval(input), children(0).dataT
ype))
          ...
}
```

In-Memory Columnar Storage

Spark SQL可以将数据缓存到内存中，我们可以见到的通过调用cache table tableName即可将一张表缓存到内存中，来极大的提高查询效率。这就涉及到内存中的数据的存储形式，我们知道基于关系型的数据可以存储为基于行存储结构或者基于列存储结构，或者基于行和列的混合存储，即Row Based Storage、Column Based Storage、 PAX Storage。

Spark SQL 将数据加载到内存是以列的存储结构。称为In-Memory Columnar Storage。若直接存储Java Object会产生很大的内存开销，并且这样是基于Row的存储结构。查询某些列速度略慢，虽然数据以及载入内存，查询效率还是低于面向列的存储结构。

Spark SQL的In-Memory Columnar Storage是位于spark列下面org.apache.spark.sql.columnar包内。核心的类有 ColumnBuilder, InMemoryColumnarTableScan, ColumnAccessor, ColumnType。如果列有压缩的情况：compression包下面有具体的build列和access列的类。

当我们调用 `sql("cache table src")` 时，会产生一个catalyst.plans.logical.CacheTableCommand，是一个LogicalPlan。

```
case class CacheTableCommand(tableName: String, plan: Option[LogicalPlan], isLazy:
Boolean)
  extends Command

abstract class Command extends LeafNode {
  self: Product =>
  def output: Seq[Attribute] = Seq.empty
}

abstract class LeafNode extends LogicalPlan with trees.LeafNode[LogicalPlan] {
  self: Product =>
}
```

在生成物理计划的时候，会转换成execution.CacheTableComman的Physical Plan。

```
case logical.CacheTableCommand(tableName, optPlan, isLazy) =>
        Seq(execution.CacheTableCommand(tableName, optPlan, isLazy))
```

```
case class CacheTableCommand(
    tableName: String,
    plan: Option[LogicalPlan],
    isLazy: Boolean)
  extends LeafNode with Command {

  override protected lazy val sideEffectResult = {
    import sqlContext._

    plan.foreach(_.registerTempTable(tableName))
```

```
    cacheTable(tableName)

    if (!isLazy) {
      // Performs eager caching
      table(tableName).count()
    }

    Seq.empty[Row]
  }

  override def output: Seq[Attribute] = Seq.empty
}
```

接着调用CacheManager的cacheTable函数，然后调cacheQuery函数。在cacheQuery里面生成了InMemoryRelation对象，就是列式存储的数据结构。

```
private[sql] trait CacheManager {
    ...
    def cacheTable(tableName: String): Unit = cacheQuery(table(tableName), Some(tab
leName))
    ...
     private[sql] def cacheQuery(
      query: SchemaRDD,
      tableName: Option[String] = None,
      storageLevel: StorageLevel = MEMORY_AND_DISK): Unit = writeLock {
    val planToCache = query.queryExecution.analyzed
    if (lookupCachedData(planToCache).nonEmpty) {
      logWarning("Asked to cache already cached data.")
    } else {
      cachedData +=
        CachedData(
          planToCache,
          InMemoryRelation(
            useCompression,
            columnBatchSize,
            storageLevel,
            query.queryExecution.executedPlan,
            tableName))
    }
  }
  ...
}
```

用法

SparkSQL中有三种方法触发cache：

1. sqlContext.sql("cache table tableName")
2. sqlContext.cacheTable("tableName")

3. schemaRDD.cache()

以上三种用法都会使用到列存储的方式对rdd进行缓存。如果调用了普通rdd的cache方法，是不会触发列式存储的cache。

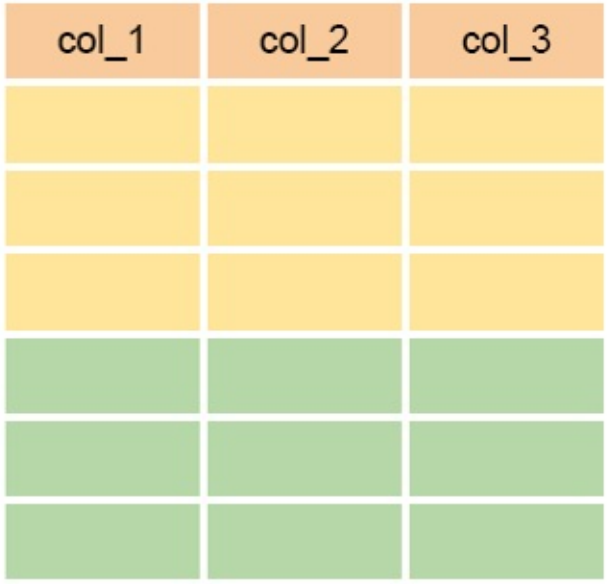
在Spark1.2.0中，cache table的执行时eager模式的，如果想触发lazy模式，可以主动添加lazy关键字，例如 `cache lazy table tabelName` 。

而在Spark1.2.0之前，cache table的默认语义是lazy的，所以需要主动触发action才会真正执行cache操作。

InMemoryRelation

InMemoryRelation继承自LogicalPlan。_cachedColumnBuffers这个类型为RDD[CachedBatch]的私有字段。CachedBatch是Array[Array[Byte]]的封装。

```
case class CachedBatch(buffers: Array[Array[Byte]], stats: Row)
```



构造一个InMemoryRelation需要该Relation

1. output Attribute
2. 是否需要useCoompression来压缩，默认为false
3. 一次处理的多少行数据batchSize
4. storageLevel 缓存到什么地方
5. child 即SparkPlan
6. table名
7. _cachedColumnBuffers最终将table放入内存的存储句柄，是一个RDD[CachedBatch]
8. _statistics是统计信息

```
private[sql] case class InMemoryRelation(
    output: Seq[Attribute],
```

```
    useCompression: Boolean,
    batchSize: Int,
    storageLevel: StorageLevel,
    child: SparkPlan,
    tableName: Option[String])(
    private var _cachedColumnBuffers: RDD[CachedBatch] = null,
    private var _statistics: Statistics = null)
  extends LogicalPlan with MultiInstanceRelation
```

可以通过设置： spark.sql.inMemoryColumnarStorage.compressed为true来设置内存中的列存储是否需要压缩。 spark.sql.inMemoryColumnarStorage.batchSize来设置一次处理多少row spark.sql.defaultSizeInBytes来设置初始化的column的bufferbytes的默认大小，这里只是其中一个参数。

缓存主流程：

1. 判断_cachedColumnBuffers是否为null，如果不是null，则已经Cache了当前table，重复cache不会触发cache操作，如果是null，则调用buildBuffers。
2. child是物理执行计划SparkPlan
3. 执行mapPartitions操作，对当前RDD的每个分区的数据进行操作。
4. 对于每一个分区，迭代里面的数据生成新的Iterator。每个Iterator里面是CachedBatch
5. 对于child.output的每一列，都会生成一个ColumnBuilder，最后组合为一个columnBuilders是一个数组。
6. 数组内每个CommandBuilder持有一个ByteBuffer
7. 遍历原始分区的记录，将对于的行转为列，并将数据存到ByteBuffer内。
8. 最后将此RDD调用persist方法，将RDD缓存。
9. 将cached赋给_cachedColumnBuffers。

```
if (_cachedColumnBuffers == null) {
    buildBuffers()
  }

private def buildBuffers(): Unit = {
    val output = child.output
    val cached = child.execute().mapPartitions { rowIterator =>
      new Iterator[CachedBatch] {
        def next() = {
          val columnBuilders = output.map { attribute =>
            val columnType = ColumnType(attribute.dataType)
            val initialBufferSize = columnType.defaultSize * batchSize
            ColumnBuilder(columnType.typeId, initialBufferSize, attribute.name, use
Compression)
          }.toArray

          var rowCount = 0
          while (rowIterator.hasNext && rowCount < batchSize) {
            val row = rowIterator.next()
            var i = 0
```

```
            while (i < row.length) {
              columnBuilders(i).appendFrom(row, i)
              i += 1
            }
            rowCount += 1
          }

          val stats = Row.fromSeq(
            columnBuilders.map(_.columnStats.collectedStatistics).foldLeft(Seq.empt
y[Any])(_ ++ _))

          batchStats += stats
          CachedBatch(columnBuilders.map(_.build().array()), stats)
        }

        def hasNext = rowIterator.hasNext
      }
    }.persist(storageLevel)

    cached.setName(tableName.map(n => s"In-memory table $n").getOrElse(child.toStri
ng))
    _cachedColumnBuffers = cached
  }
```

ColumnBuilder

columnBuilders是一个存储ColumnBuilder的数组。

```
 val columnBuilders = output.map { attribute =>
            val columnType = ColumnType(attribute.dataType)
            val initialBufferSize = columnType.defaultSize * batchSize
            ColumnBuilder(columnType.typeId, initialBufferSize, attribute.name, use
Compression)
          }.toArray
```

初始化ColumnBuilder的时候会传入的参数：

1. columnType.typeId 表示列的数据类型
2. initialBufferSize ByteBuffer的初始化大小，列类型默认长度 * batchSize ， 默认batchSize是1000。拿Int类型举例，initialBufferSize of IntegerType = 4 * 1000
3. attribute.name 即字段名age,name，etc
4. useCompression 是否开启压缩

ColumnType封装了该类型的typeId和该类型的defaultSize。并且提供了extract、append、getField方法，来向buffer里追加和获取数据。

```
private[sql] sealed abstract class ColumnType[T <: DataType, JvmType](
    val typeId: Int,
```

```
    val defaultSize: Int) {

  def extract(buffer: ByteBuffer): JvmType

  def append(v: JvmType, buffer: ByteBuffer): Unit

  def actualSize(row: Row, ordinal: Int): Int = defaultSize
  ...
}
```

ColumnBuilder的主要职责是：管理ByteBuffer，包括初始化buffer，添加数据到buffer内，检查剩余空间，和申请新的空间这几项主要职责。

initialize负责初始化buffer。

```
override def initialize(
      initialSize: Int,
      columnName: String = "",
      useCompression: Boolean = false) = {

    val size = if (initialSize == 0) DEFAULT_INITIAL_BUFFER_SIZE else initialSize
    this.columnName = columnName

    // Reserves 4 bytes for column type ID
    buffer = ByteBuffer.allocate(4 + size * columnType.defaultSize)
    buffer.order(ByteOrder.nativeOrder()).putInt(columnType.typeId)
  }
```

appendFrom是负责添加数据。

```
override def appendFrom(row: Row, ordinal: Int): Unit = {
    buffer = ensureFreeSpace(buffer, columnType.actualSize(row, ordinal))
    columnType.append(row, ordinal, buffer)
  }
```

ensureFreeSpace主要是操作buffer，如果要追加的数据大于剩余空间，就扩大buffer。

```
private[columnar] def ensureFreeSpace(orig: ByteBuffer, size: Int) = {
    if (orig.remaining >= size) {
      orig
    } else {
      // grow in steps of initial size
      val capacity = orig.capacity()
      val newSize = capacity + size.max(capacity / 8 + 1)
      val pos = orig.position()

      ByteBuffer
        .allocate(newSize)
```

```
          .order(ByteOrder.nativeOrder())
          .put(orig.array(), 0, pos)
      }
    }
```

External Data Source

外部数据源的注册流程

DDLParser

`Create Temmporary Table Using` 语法的解析由DDLParser负责，最后生成一个CreateTableUsing的类。

```
protected lazy val createTable: Parser[LogicalPlan] =
    CREATE ~ TEMPORARY ~ TABLE ~> ident ~ (USING ~> className) ~ (OPTIONS ~> option
s) ^^ {
      case tableName ~ provider ~ opts =>
        CreateTableUsing(tableName, provider, opts)
    }
```

CreateTableUsing (Logical Plan)

CreateTableUsing继承自RunnableCommand，是一个Logical Plan。

CreateTableUsing类接受三个参数：

1. tableName：表名
2. provider：解析具体文件数据的类或包
3. options：解析时需要的其他参数

```
private[sql] case class CreateTableUsing(
    tableName: String,
    provider: String,
    options: Map[String, String]) extends RunnableCommand {

  def run(sqlContext: SQLContext) = {
    val loader = Utils.getContextOrSparkClassLoader
    val clazz: Class[_] = try loader.loadClass(provider) catch {
      case cnf: java.lang.ClassNotFoundException =>
        try loader.loadClass(provider + ".DefaultSource") catch {
          case cnf: java.lang.ClassNotFoundException =>
            sys.error(s"Failed to load class for data source: $provider")
        }
    }
    val dataSource = clazz.newInstance().asInstanceOf[org.apache.spark.sql.sources.
RelationProvider]
    val relation = dataSource.createRelation(sqlContext, options)

    sqlContext.baseRelationToSchemaRDD(relation).registerTempTable(tableName)
    Seq.empty
  }
}
```

CreateTableUsing的run方法定义了这个Logical Plan需要做的事情：

1. 加载provider类
2. 如果1失败，加载provider包中的DefaultSource类
3. 新建加载进来的类的对象，类型转换为org.apache.spark.sql.sources.RelationProvider
4. 通过RelationProvider的createRelation方法创建BaseRelation对象
5. 通过SqlContext的baseRelationToSchemaRDD方法创建并注册SchemaRDD

baseRelationToSchemaRDD方法

1. 首先把baseRelation包装成LogicalRelation(baseRelation)
2. 然后调用logicalPlanToSparkQuery
3. 最后被包装成 SchemaRDD(SqlContext, LogicalPlan(baseRelation))

```
implicit def baseRelationToSchemaRDD(baseRelation: BaseRelation): SchemaRDD = {
    logicalPlanToSparkQuery(LogicalRelation(baseRelation))
  }

implicit def logicalPlanToSparkQuery(plan: LogicalPlan): SchemaRDD = new SchemaRDD(
this, plan)
```

ExecutedCommand (Physical Plan)

CreateTableUsing这个Logical Plan最后会被转换为ExecutedCommand的Physcial Plan。执行ExecutedCommand的时候，会调用CreateTableUsing的run函数，从而触发resolver类的加载以及SchemaRDD的注册。

```
case class ExecutedCommand(cmd: RunnableCommand) extends SparkPlan {
 protected[sql] lazy val sideEffectResult: Seq[Row] = cmd.run(sqlContext)

  override def output = cmd.output

  override def children = Nil

  override def executeCollect(): Array[Row] = sideEffectResult.toArray

  override def execute(): RDD[Row] = sqlContext.sparkContext.parallelize(sideEffect
Result, 1)
}
```

外部数据源的核心类

下面介绍一下外部数据源的两个核心类RelationProvider和BaseRelation。

RelationProvider

RelationProvider是外部数据源的入口，如果想让SparkSQL读取某种格式的外部数据，必须继承该类。RelationProvider只有一个方法createRelation，接收SqlContext和一堆参数，返回BaseRelation。

```
trait RelationProvider {
  /** Returns a new base relation with the given parameters. */
  def createRelation(sqlContext: SQLContext, parameters: Map[String, String]): Base
Relation
}
```

来看一下SparkSQL自带的读取JSON格式数据的DefaultSource：

1. 该类继承自RelationProvider
2. createRelation方法首先读取参数path（数据路径）和samplingRate（取样比例）
3. 然后创建JSONRelation

```
private[sql] class DefaultSource extends RelationProvider {
  /** Returns a new base relation with the given parameters. */
  override def createRelation(
      sqlContext: SQLContext,
      parameters: Map[String, String]): BaseRelation = {
    val fileName = parameters.getOrElse("path", sys.error("Option 'path' not specif
ied"))
    val samplingRatio = parameters.get("samplingRatio").map(_.toDouble).getOrElse(1
.0)

    JSONRelation(fileName, samplingRatio)(sqlContext)
  }
}
```

BaseRelation

SparkSQL用BaseRelation类来把一个数据文件抽象成一张表格，在BaseRelation中定义了Schema。

```
abstract class BaseRelation {
  def sqlContext: SQLContext
  def schema: StructType
  def sizeInBytes = sqlContext.defaultSizeInBytes
}
```

BaseRelation有四个子类，如果想让SparkSQL读取某种格式的外部数据，必须继承下面四个类中的其中一个。 四个子类都提供了一个buildScan的方法，返回值都一样是RDD[Row]，而参数各不一样。

- TableScan是最基本的BaseRelation，不接收任何参数
- PrunedScan只读取某些特定列，接收Stirng数组作为参数
- PrunedFilteredScan在PrunedScan的基础上还增加行过滤，额外接收Filter数组
- CatalystScan类似于PrunedFilteredScan，不同点是filters的类型为Expression

```
abstract class TableScan extends BaseRelation {
```

```
  def buildScan(): RDD[Row]
}

abstract class PrunedScan extends BaseRelation {
  def buildScan(requiredColumns: Array[String]): RDD[Row]
}

abstract class PrunedFilteredScan extends BaseRelation {
  def buildScan(requiredColumns: Array[String], filters: Array[Filter]): RDD[Row]
}

abstract class CatalystScan extends BaseRelation {
  def buildScan(requiredColumns: Seq[Attribute], filters: Seq[Expression]): RDD[Row
]
}
```

看一下SparkSQL为读取json格式的数据实现的JSONRelation:

1. JSONRelation继承自TableScan，说明不做任何过滤
2. 首先通过SparkContext的textFile生成baseRDD
3. 然后调用JsonRDD的inferSchema，从原始数据中分析出Schema信息
4. 最后调用JsonRDD的jsonStringToRow，把Json格式的数据转成SchemaRDD格式

```
private[sql] case class JSONRelation(fileName: String, samplingRatio: Double)(
    @transient val sqlContext: SQLContext)
  extends TableScan {

  private def baseRDD = sqlContext.sparkContext.textFile(fileName)

  override val schema =
    JsonRDD.inferSchema(
      baseRDD,
      samplingRatio,
      sqlContext.columnNameOfCorruptRecord)

  override def buildScan() =
    JsonRDD.jsonStringToRow(baseRDD, schema, sqlContext.columnNameOfCorruptRecord)
}
```

外部数据源Table查询的计划解析流程

DataSourcesStrategy (Logical Plan -> Physical Plan)

DataSourcesStrategy是Logical Plan转换成Physical Plan中

```
private[sql] object DataSourceStrategy extends Strategy {
  def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
    case PhysicalOperation(projectList, filters, l @ LogicalRelation(t: CatalystSca
n)) =>
```

```
      pruneFilterProjectRaw(
        l,
        projectList,
        filters,
        (a, f) => t.buildScan(a, f)) :: Nil

    case PhysicalOperation(projectList, filters, l @ LogicalRelation(t: PrunedFilte
redScan)) =>
      pruneFilterProject(
        l,
        projectList,
        filters,
        (a, f) => t.buildScan(a, f)) :: Nil

    case PhysicalOperation(projectList, filters, l @ LogicalRelation(t: PrunedScan)
) =>
      pruneFilterProject(
        l,
        projectList,
        filters,
        (a, _) => t.buildScan(a)) :: Nil

    case l @ LogicalRelation(t: TableScan) =>
      execution.PhysicalRDD(l.output, t.buildScan()) :: Nil

    case _ => Nil
  }
  ...
}
```

如何自定义一个外部数据源

参考

Spark SQL之External DataSource外部数据源（一）示例

Spark SQL之External DataSource外部数据源（二）源码分析

Code Generation

SparkSQL利用Scala提供的Reflection和Quasiquotes机制来实现Code Generation。

Scala Reflection

Java本身就已经提供了Reflection功能，在Scala2.10之前，Scala并不提供额外的Reflection机制，Scala2.10在Java Reflection基础上又实现了一套更加powerful的Reflection机制。Scala中的Reflection分为两大类：Runtime Reflection和Compile-time Reflection。

Runtime Reflection

Runtime Reflection主要包括

1. 动态获取对象的类型
2. 动态新建对象
3. 动态调用对象的方法

这部分和Java提供的Reflection比较类似。

Compile-time Reflection

Scala的Compile-time Reflection允许程序在编译期间修改自己的代码，从而实现meta-programming。Complie-time Reflection是通过macros实现的，macros允许程序可以在编译期间修改自己的语法树。

Abstract Syntax Tree

Scala使用Abstract Syntax Tree来表示Scala程序，Scala的Reflection提供了多种方法来操作AST:

1. reify方法可以把一个表达式变成一个AST
2. 在Compile-time，通过macros可以操作AST
3. 在Runtime，通过toolboxes也可以操作AST

下面的例子演示了如果在Runtime利用toolboxes操作AST

1. 在Runtime利用ToolBox的parse函数，将一段代码变了AST
2. 然后通过ToolBox的eval函数，将生成的AST进行求值，返回该段代码的返回值f，f是一个实现+1操作的函数
3. 最后调用返回的函数f

```
import scala.tools.reflect.ToolBox

object HelloToolBox {
  def main(args: Array[String]): Unit = {
    val toolBox = scala.reflect.runtime.universe.
      runtimeMirror(getClass.getClassLoader).mkToolBox()

    val code =
```

```
      """
        val f = {x: Int => x + 1}
        f
      """

    val tree = toolBox.parse(code)
    println(tree)

    val func = toolBox.eval(tree)
    val result = func.asInstanceOf[Int => Int](1)
    println(result)

  }
}
```

程序的输出为：

```
{
  val f = ((x: Int) => x.$plus(1));
  f
}
2
```

Scala Quasiquotes

Quasiquotes是Scala提供的方便操作AST的库，当把代码放到 `q"..."` 里面时，将会返回一个AST。

```
scala> val tree = q"i am { a quasiquote }"
tree: universe.Tree = i.am(a.quasiquote)
```

下面的例子演示了如何利用Quasiquotes来实现Code Generation:

```
import scala.reflect.runtime.currentMirror
import scala.tools.reflect.ToolBox

object HelloQuasiquotes {

  def main(args: Array[String]): Unit = {
    val universe: scala.reflect.runtime.universe.type = scala.reflect.runtime.unive
rse
    import universe._

    val toolbox = currentMirror.mkToolBox()
    val tree = q"""
              val f = {x: Int => x + 1}
              f
             """
```

```
    println(showCode(tree))

    val func = toolbox.eval(tree)
    val result = func.asInstanceOf[Int => Int](1)
    println(result)
  }
}
```

程序输出为：

```
{
  val f = ((x: Int) => x.+(1));
  f
}
2
```

SparkSQL Code Generation

SparkSQL使用Quasiquotes来实现Code Genration。Quasiquotes是Scala 2.11的新功能，当使用Scala 2.10来编译Spark时，只能通过macro paradise compiler plugin的方式使用quasiquotes。

CodeGenerator

CodeGenerator是代码生成的基类，里面定义了

1. expressionEvaluator：把一个Expression变成EvaluatedExpression，即AST
2. toolBox: 可以在返回的AST上求值，返回动态生成的代码的调用入口

```
abstract class CodeGenerator[InType <: AnyRef, OutType <: AnyRef] extends Logging {
  import scala.reflect.runtime.{universe => ru}
  import scala.reflect.runtime.universe._
  import scala.tools.reflect.ToolBox

  protected val toolBox = runtimeMirror(getClass.getClassLoader).mkToolBox()

  def expressionEvaluator(e: Expression): EvaluatedExpression = {
    ...
  }
  ...
}
```

EvaluatedExpression可以理解为已经生成好的代码，这段代码可以在Row上求值，EvaluatedEpression包括

1. code：AST的List，用来求值的代码
2. nullTerm：代码中的变量名，表示最后的值是否是null
3. primitiveTerm：代码中的变量名，表示最后的值的原始类型，无效如果nullTerm=true
4. objectTerm：代码中的变量名，表示最后的值的包装类型

```
  protected case class EvaluatedExpression(
      code: Seq[Tree],
      nullTerm: TermName,
      primitiveTerm: TermName,
      objectTerm: TermName)
```

下面来看一下Code Generation的关键函数expressionEvaluator。

1. 调用freshName为primitiveTerm、nullTerm、objectTerm生成变量名
2. 调用primitiveEvaluation，把Expression变成Seq[Tree]，可能会失败
3. 如果primitiveEvaluation失败，则生成表达式树求值的代码
4. 返回EvaluatedExpression

```
def expressionEvaluator(e: Expression): EvaluatedExpression = {
    val primitiveTerm = freshName("primitiveTerm")
    val nullTerm = freshName("nullTerm")
    val objectTerm = freshName("objectTerm")
    val primitiveEvaluation: PartialFunction[Expression, Seq[Tree]] = {...}
    val code: Seq[Tree] =
      primitiveEvaluation.lift.apply(e).getOrElse {
        log.debug(s"No rules to generate $e")
        val tree = reify { e }
        q"""
          val $objectTerm = $tree.eval(i)
          val $nullTerm = $objectTerm == null
          val $primitiveTerm = $objectTerm.asInstanceOf[${termForType(e.dataType)}]
         """.children
      }
      ...
    EvaluatedExpression(code ++ debugCode, nullTerm, primitiveTerm, objectTerm)
}
```

primitiveEvaluation是一个PartialFunction，把Expression变成Seq[Tree]。

下面分析一下And是如何生成代码的：

1. 分别生成e1和e2的AST
2. 分几种情况判断And的结果是否为null以及是否为primitive
3. 返回e1、e2和add的代码

```
val primitiveEvaluation: PartialFunction[Expression, Seq[Tree]] = {
    ...
    case And(e1, e2) =>
        val eval1 = expressionEvaluator(e1)
        val eval2 = expressionEvaluator(e2)

        eval1.code ++ eval2.code ++
        q"""
```

```
          var $nullTerm = false
          var $primitiveTerm: ${termForType(BooleanType)} = false

          if ((!${eval1.nullTerm} && !${eval1.primitiveTerm}) ||
              (!${eval2.nullTerm} && !${eval2.primitiveTerm})) {
            $nullTerm = false
            $primitiveTerm = false
          } else if (${eval1.nullTerm} || ${eval2.nullTerm} ) {
            $nullTerm = true
          } else {
            $nullTerm = false
            $primitiveTerm = true
          }
         """.children
  ...
}
```

Add的代码看上去更加简洁，只有一句话，其实是因为把(e1, e2)隐式转换成了class Evaluate2，并调用其evaluate方法。

```
case Add(e1, e2) =>      (e1, e2) evaluate { case (eval1, eval2) => q"$eval1 + $eva
l2" }
```

evaluate方法接受一个function，返回一个Seq[Tree]。function接收两个scala reflaction中的TermName，返回一个AST。

1. evaluate方首先调用expressionEvaluator，分别计算两个表达式的AST
2. 然后生成传递进来的function的AST，参数为1中得到的primitiveTerm
3. 生成function代码的nullTerm
4. 生成function代码的primitiveTerm，如果是null的话返回该类型的默认值

```
implicit class Evaluate2(expressions: (Expression, Expression)) {
    def evaluate(f: (TermName, TermName) => Tree): Seq[Tree] =
        evaluateAs(expressions._1.dataType)(f)

      def evaluateAs(resultType: DataType)(f: (TermName, TermName) => Tree): Seq[Tr
ee] = {
        // TODO: Right now some timestamp tests fail if we enforce this...
        if (expressions._1.dataType != expressions._2.dataType) {
          log.warn(s"${expressions._1.dataType} != ${expressions._2.dataType}")
        }

        val eval1 = expressionEvaluator(expressions._1)
        val eval2 = expressionEvaluator(expressions._2)
        val resultCode = f(eval1.primitiveTerm, eval2.primitiveTerm)

        eval1.code ++ eval2.code ++
        q"""
          val $nullTerm = ${eval1.nullTerm} || ${eval2.nullTerm}
```

```
          val $primitiveTerm: ${termForType(resultType)} =
            if($nullTerm) {
              ${defaultPrimitive(resultType)}
            } else {
              $resultCode.asInstanceOf[${termForType(resultType)}]
            }
        """.children : Seq[Tree]
      }
    }
    ...
}
```

来看一下Code Generation生成出来的Add的示例代码：

```
(() => {
  final class $anon extends org.apache.spark.sql.catalyst.expressions.MutableProjec
tion {
    def <init>() = {
      super.<init>();
      ()
    };
    private[this] var mutableRow: org.apache.spark.sql.catalyst.expressions.Mutable
Row = new org.apache.spark.sql.catalyst.expressions.GenericMutableRow(1);
    def target(row: org.apache.spark.sql.catalyst.expressions.MutableRow): org.apac
he.spark.sql.catalyst.expressions.MutableProjection = {
      mutableRow = row;
      this
    };
    def currentValue: org.apache.spark.sql.catalyst.expressions.Row = mutableRow;
    def apply(i: org.apache.spark.sql.catalyst.expressions.Row): org.apache.spark.s
ql.catalyst.expressions.Row = {
      val nullTerm$4: Boolean = i.isNullAt(0);
      val primitiveTerm$3: org.apache.spark.sql.catalyst.types.IntegerType.JvmType
= if (nullTerm$4)
        -1
      else
        i.getInt(0);
      ();
      val nullTerm$7: Boolean = i.isNullAt(1);
      val primitiveTerm$6: org.apache.spark.sql.catalyst.types.IntegerType.JvmType
= if (nullTerm$7)
        -1
      else
        i.getInt(1);
      ();
      val nullTerm$1 = nullTerm$4.$bar$bar(nullTerm$7);
      val primitiveTerm$0: org.apache.spark.sql.catalyst.types.IntegerType.JvmType
= if (nullTerm$1)
        -1
      else
```

```
        primitiveTerm$3.$plus(primitiveTerm$6).asInstanceOf[org.apache.spark.sql.ca
talyst.types.IntegerType.JvmType];
      ();
      if (nullTerm$1)
        mutableRow.setNullAt(0)
      else
        mutableRow.setInt(0, primitiveTerm$0);
      mutableRow
    }
  };
  new $anon()
})
```

有四个类继承了CodeGenrator

1. GenerateProjection
2. GenerateMutableProjection
3. GenerateOrdering
4. GeneratePredicate

GeneratePredicate

下面来分析一下GeneratePredicate这个类。create函数根据一个表达式生成了一段函数代码，该函数的输入是Row（即一行数据），输出是Boolean（即该行是否被过滤）。

1. 首先调用expressionEvaluator方法，计算出Expression的AST
2. 然后生成返回的函数的代码，函数首先调用1中所得到的AST，如果1中AST的nullTerm=true，则返回false，否则返回1中AST的实际返回值primitiveTerm
3. 最后调用toolBox的eval函数，获得AST中function的实例入口地址

```
object GeneratePredicate extends CodeGenerator[Expression, (Row) => Boolean] {
  import scala.reflect.runtime.{universe => ru}
  import scala.reflect.runtime.universe._

  protected def canonicalize(in: Expression): Expression = ExpressionCanonicalizer(
in)

  protected def bind(in: Expression, inputSchema: Seq[Attribute]): Expression =
    BindReferences.bindReference(in, inputSchema)

  protected def create(predicate: Expression): ((Row) => Boolean) = {
    val cEval = expressionEvaluator(predicate)

    val code =
      q"""
        (i: $rowType) => {
          ..${cEval.code}
          if (${cEval.nullTerm}) false else ${cEval.primitiveTerm}
        }
      """
```

```
    toolBox.eval(code).asInstanceOf[Row => Boolean]
  }
}
```

参考

http://docs.scala-lang.org/overviews/reflection/symbols-trees-types.html

http://docs.scala-lang.org/overviews/quasiquotes/intro.html

http://docs.scala-lang.org/overviews/quasiquotes/setup.html

http://docs.scala-lang.org/overviews/macros/paradise.html

http://www.infoq.com/cn/news/2013/01/scala-macros

https://databricks.com/blog/2014/06/02/exciting-performance-improvements-on-the-horizon-for-spark-sql.html

https://gist.github.com/marmbrus/9efb31d2b5154aea6652

推荐资料

官方资料

Databricks官网

Spark官网

Spark Github

Spark GIRA

Spark Submit

Spark博客

OopsOutOfMemory

mmicky的hadoop、Spark世界

徽沪一郎

baishuo491

赛赛的网络日志 Jerry Shao

JerryLead博客园

张包峰的博客

Spark深入研究

Spark Internals by JerryLead

Spark论文

Spark: Cluster Computing with Working Sets Matei Zaharia

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing Matei Zaharia

Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters Matei Zaharia

Shark: SQL and Rich Analytics at Scale Reynold Shi Xin, Matei Zaharia