## 8.Spark SQL 原理剖析☆☆☆

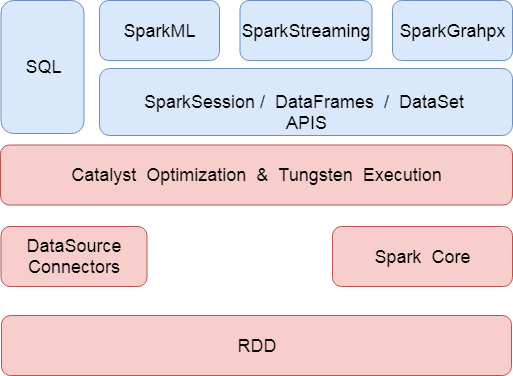
为什么我们需要学习Spark SQL 的原理呢？

* 面试会问，如果回答的不错，肯定是加分项
* 工作会用，懂Spark SQL原理的话，在我们日常工作中遇到Spark报错或者Spark调优时，更有利用我们解决这些问题。

### 8.1 SparkSQL的架构设计☆☆☆

**8.1.1 SparkSQL架构设计概述**

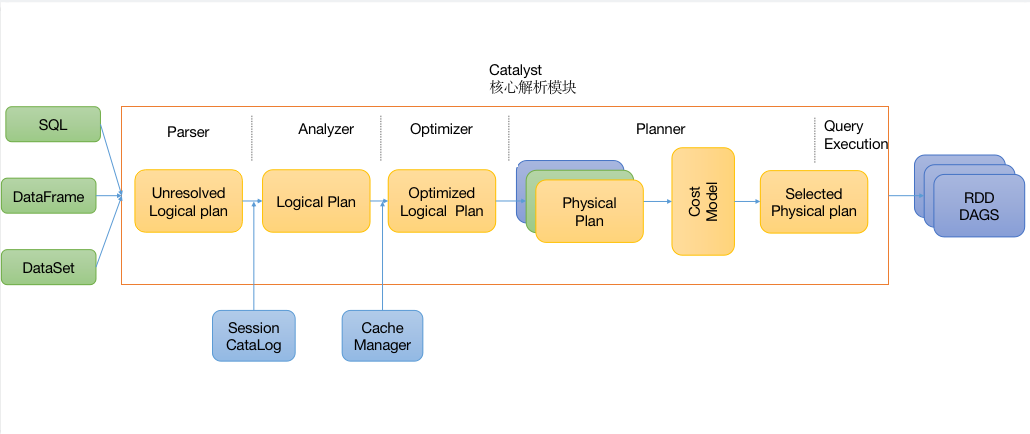
SparkSQL是spark技术栈当中又一非常出彩的模块，通过引入SQL的支持，大大降低了开发人员和学习人员的使用成本，让我们开发人员直接使用SQL的方式就能够实现大数据的开发，它同时支持DSL以及SQL的语法风格，目前在spark的整个架构设计当中，所有的spark模块，例如SQL，SparkML，sparkGrahpx以及Structed Streaming等都是基于 Catalyst Optimization & Tungsten Execution模块之上运行，如下图所示就显示了spark的整体架构模块设计



**8.2.2 Spark SQL 架构设计实现**

通常的 SQL 执行先会经过 SQL Parser 解析 SQL，然后经过 Catalyst 优化器处理，最后到 Spark 执行。而 Catalyst 的过程又分为很多个过程，其中包括：

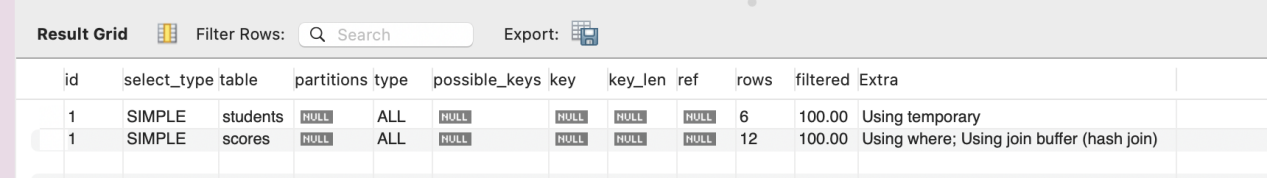
* **Analysis：**主要利用 Catalog 信息将 Unresolved Logical Plan 解析成 Analyzed logical plan；
* **Logical Optimizations：**利用一些 Rule （规则）将 Analyzed logical plan 解析成 Optimized Logical Plan；
* **Physical Planning：**前面的 logical plan 不能被 Spark 执行，而这个过程是把 logical plan 转换成多个 physical plans，然后利用代价模型（cost model）选择最佳的 physical plan；
* **Code Generation：**这个过程会把 SQL· 查询生成 Java 字节码。



例如执行以下SQL语句

|  |
| --- |
| SELECT temp1.class,  sum(temp1.degree),  avg(temp1.degree)  FROM  (SELECT students.sno AS ssno,  students.sname,  students.ssex,  students.sbirthday,  students.class,  scores.sno,  scores.degree,  scores.cno  FROM students  LEFT JOIN scores  ON students.sno = scores.sno ) temp1  GROUP BY temp1.class |

Mysql 执行计划



代码实现过程如下：

package com.ds.spark.SparkSQL.chapter08

import java.util.Properties

import org.apache.spark.SparkConf

import org.apache.spark.sql.{DataFrame, SparkSession}

object SparkSQLParseDemo extends App {

// 1.创建SparkConf对象

val sparkConf: SparkConf = new SparkConf().setAppName("DataFromMysql").setMaster("local[2]")

//sparkConf.set("spark.sql.codegen.wholeStage","true")

//2、创建SparkSession对象

val spark: SparkSession = SparkSession.builder().config(sparkConf).getOrCreate()

spark.sparkContext.setLogLevel("WARN")

//3、读取mysql表的数据

//3.1 指定mysql连接地址

val url = "jdbc:mysql://localhost:3306/ds\_mysql?characterEncoding=UTF-8"

//3.2 指定要加载的表名

val student = "students"

val score = "scores"

// 3.3 配置连接数据库的相关属性

val properties = new Properties()

//用户名

properties.setProperty("user", "root")

//密码

properties.setProperty("password", "12345678")

val studentFrame: DataFrame = spark.read.jdbc(url, student, properties)

val scoreFrame: DataFrame = spark.read.jdbc(url, score, properties)

//把dataFrame注册成表

studentFrame.createTempView("students")

scoreFrame.createOrReplaceTempView("scores")

spark.sql(

"""

|SELECT temp1.class,

| SUM(temp1.degree),

| AVG(temp1.degree)

|FROM

| (SELECT students.sno AS ssno,

| students.sname,

| students.ssex,

| students.sbirthday,

| students.class,

| scores.sno,

| scores.degree,

| scores.cno

| FROM students

| LEFT JOIN scores

| ON students.sno = scores.sno ) temp1

|GROUP BY temp1.class

|""".stripMargin).show()

val resultFrame: DataFrame = spark.sql(

"""

|SELECT temp1.class,

| SUM(temp1.degree),

| AVG(temp1.degree)

|FROM

| (SELECT students.sno AS ssno,

| students.sname,

| students.ssex,

| students.sbirthday,

| students.class,

| scores.sno,

| scores.degree,

| scores.cno

| FROM students

| LEFT JOIN scores

| ON students.sno = scores.sno

| WHERE degree > 60

| AND sbirthday > '1973-01-01 00:00:00' ) temp1

|GROUP BY temp1.class

|""".stripMargin)

resultFrame.explain(true)

resultFrame.show()

spark.stop()

}

通过explain方法来查看sql的执行计划，得到以下信息

|  |
| --- |
| == Parsed Logical Plan ==  'Aggregate ['temp1.class], ['temp1.class, unresolvedalias('SUM('temp1.degree), None), unresolvedalias('AVG('temp1.degree), None)]  +- 'SubqueryAlias temp1  +- 'Project ['students.sno AS ssno#57, 'students.sname, 'students.ssex, 'students.sbirthday, 'students.class, 'scores.sno, 'scores.degree, 'scores.cno]  +- 'Filter (('degree > 60) AND ('sbirthday > 1973-01-01 00:00:00))  +- 'Join LeftOuter, ('students.sno = 'scores.sno)  :- 'UnresolvedRelation [students], [], false  +- 'UnresolvedRelation [scores], [], false  == Analyzed Logical Plan ==  class: string, sum(degree): decimal(20,1), avg(degree): decimal(14,5)  Aggregate [class#4], [class#4, sum(degree#12) AS sum(degree)#60, avg(degree#12) AS avg(degree)#61]  +- SubqueryAlias temp1  +- Project [sno#0 AS ssno#57, sname#1, ssex#2, sbirthday#3, class#4, sno#10, degree#12, cno#11]  +- Filter ((cast(degree#12 as decimal(10,1)) > cast(cast(60 as decimal(2,0)) as decimal(10,1))) AND (sbirthday#3 > cast(1973-01-01 00:00:00 as timestamp)))  +- Join LeftOuter, (sno#0 = sno#10)  :- SubqueryAlias students  : +- View (`students`, [sno#0,sname#1,ssex#2,sbirthday#3,class#4])  : +- Relation [sno#0,sname#1,ssex#2,sbirthday#3,class#4] JDBCRelation(students) [numPartitions=1]  +- SubqueryAlias scores  +- View (`scores`, [sno#10,cno#11,degree#12])  +- Relation [sno#10,cno#11,degree#12] JDBCRelation(scores) [numPartitions=1]  == Optimized Logical Plan ==  Aggregate [class#4], [class#4, sum(degree#12) AS sum(degree)#60, cast((avg(UnscaledValue(degree#12)) / 10.0) as decimal(14,5)) AS avg(degree)#61]  +- Project [class#4, degree#12]  +- Join Inner, (sno#0 = sno#10)  :- Project [sno#0, class#4]  : +- Filter ((isnotnull(sbirthday#3) AND (sbirthday#3 > 1973-01-01 00:00:00)) AND isnotnull(sno#0))  : +- Relation [sno#0,sname#1,ssex#2,sbirthday#3,class#4] JDBCRelation(students) [numPartitions=1]  +- Project [sno#10, degree#12]  +- Filter ((isnotnull(degree#12) AND (degree#12 > 60.0)) AND isnotnull(sno#10))  +- Relation [sno#10,cno#11,degree#12] JDBCRelation(scores) [numPartitions=1]  == Physical Plan ==  AdaptiveSparkPlan isFinalPlan=false  +- HashAggregate(keys=[class#4], functions=[sum(degree#12), avg(UnscaledValue(degree#12))], output=[class#4, sum(degree)#60, avg(degree)#61])  +- Exchange hashpartitioning(class#4, 200), ENSURE\_REQUIREMENTS, [id=#239]  +- HashAggregate(keys=[class#4], functions=[partial\_sum(degree#12), partial\_avg(UnscaledValue(degree#12))], output=[class#4, sum#69, isEmpty#70, sum#71, count#72L])  +- Project [class#4, degree#12]  +- SortMergeJoin [sno#0], [sno#10], Inner  :- Sort [sno#0 ASC NULLS FIRST], false, 0  : +- Exchange hashpartitioning(sno#0, 200), ENSURE\_REQUIREMENTS, [id=#231]  : +- Scan JDBCRelation(students) [numPartitions=1] [sno#0,class#4] PushedAggregates: [], PushedFilters: [\*IsNotNull(sbirthday), \*GreaterThan(sbirthday,1973-01-01 00:00:00.0), \*IsNotNull(sno)], PushedGroupby: [], ReadSchema: struct<sno:string,class:string>  +- Sort [sno#10 ASC NULLS FIRST], false, 0  +- Exchange hashpartitioning(sno#10, 200), ENSURE\_REQUIREMENTS, [id=#232]  +- Scan JDBCRelation(scores) [numPartitions=1] [sno#10,degree#12] PushedAggregates: [], PushedFilters: [\*IsNotNull(degree), \*GreaterThan(degree,60.0), \*IsNotNull(sno)], PushedGroupby: [], ReadSchema: struct<sno:string,degree:decimal(10,1)> |

### 8.2 Catalyst执行过程☆☆☆

从上面的查询计划我们可以看得出来，我们编写的sql语句，经过多次转换，最终进行编译成为字节码文件进行执行，这一整个过程经过了好多个步骤，其中包括以下几个重要步骤：

* sql解析阶段 parse
* 生成逻辑计划 Analyzer
* sql语句调优阶段 Optimizer
* 生成物理查询计划 planner

**8.2.1 SQL解析阶段**

在spark2.x的版本当中，为了解析sparkSQL的sql语句，引入了**==Antlr==**。Antlr 是一款强大的语法生成器工具，可用于读取、处理、执行和翻译结构化的文本或二进制文件，是当前 Java 语言中使用最为广泛的语法生成器工具，我们常见的大数据 SQL 解析都用到了这个工具，包括 Hive、Cassandra、Phoenix、Pig 以及 presto 等。目前最新版本的 Spark 使用的是==ANTLR4==，通过这个对 SQL 进行词法分析并构建语法树。

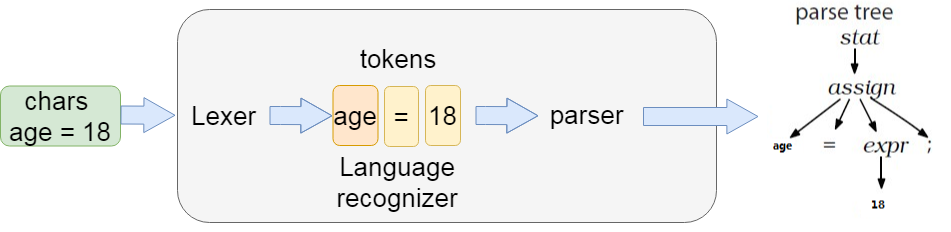
我们可以通过github去查看spark的源码，具体路径如下：

<https://github.com/apache/spark/tree/master/sql/catalyst/src/main/antlr4/org/apache/spark/sql/catalyst/parser>



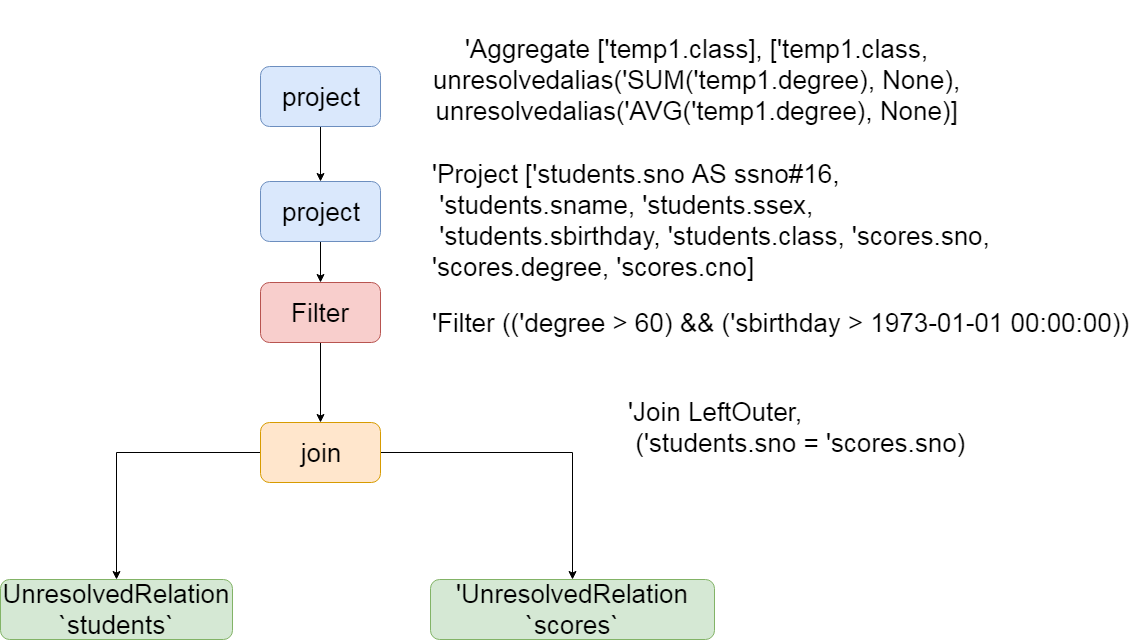


查看得到sparkSQL支持的SQL语法，所有sparkSQL支持的语法都定义在了这个文件当中。如果我们需要重构sparkSQL的语法，那么我们只需要重新定义好相关语法，然后使用Antlr4对SqlBase.g4进行语法解析，生成相关的java类，其中就包含重要的词法解析器SqlBaseLexer.java和语法解析器SqlBaseParser.java。在我们运行上面的java的时候，第一步就是通过SqlBaseLexer来解析关键词以及各种标识符，然后使用SqlBaseParser来构建语法树。



最终通过Lexer以及parse解析之后，生成语法树，生成语法树之后，使用AstBuilder将语法树转换成为LogicalPlan，这个LogicalPlan也被称为Unresolved LogicalPlan。解析之后的逻辑计划如下：

|  |
| --- |
| == Parsed Logical Plan ==  'Aggregate ['temp1.class], ['temp1.class, unresolvedalias('SUM('temp1.degree), None), unresolvedalias('AVG('temp1.degree), None)]  +- 'SubqueryAlias temp1  +- 'Project ['students.sno AS ssno#16, 'students.sname, 'students.ssex, 'students.sbirthday, 'students.class, 'scores.sno, 'scores.degree, 'scores.cno]  +- 'Filter (('degree > 60) && ('sbirthday > 1973-01-01 00:00:00))  +- 'Join LeftOuter, ('students.sno = 'scores.sno)  :- 'UnresolvedRelation `students`  +- 'UnresolvedRelation `scores` |



从上图可以看得到，两个表被join之后生成了UnresolvedRelation，选择的列以及聚合的字段都有了，sql解析的第一个阶段就已经完成，接着准备进入到第二个阶段

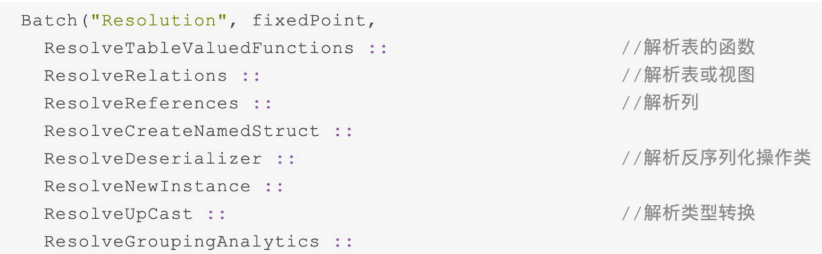
### 8.2.2绑定逻辑计划Analyzer

在sql解析parse阶段，生成了很多的unresolvedalias ， UnresolvedRelation等很多未解析出来的有些关键字，这些都是属于 Unresolved LogicalPlan解析的部分。 Unresolved LogicalPlan仅仅是一种数据结构，不包含任何数据信息，例如不知道数据源，数据类型，不同的列来自哪张表等等。Analyzer 阶段会使用事先定义好的 Rule 以及 SessionCatalog 等信息对 Unresolved LogicalPlan 进行 transform。SessionCatalog 主要用于各种函数资源信息和元数据信息（数据库、数据表、数据视图、数据分区与函数等）的统一管理。而Rule 是定义在 Analyzer 里面的，具体的类的路径如下：

org.apache.spark.sql.catalyst.analysis.Analyzer

具体的rule规则定义如下：

Rule的种类有非常多，之前所说的和SessionCatalog结合一起对数据库、表、列的解析就是其中的几种Rule（ResolveRelations，ResolveReference等）。





override def batches: Seq[Batch] = Seq(

Batch("Substitution", fixedPoint,

// This rule optimizes `UpdateFields` expression chains so looks more like optimization rule.

// However, when manipulating deeply nested schema, `UpdateFields` expression tree could be

// very complex and make analysis impossible. Thus we need to optimize `UpdateFields` early

// at the beginning of analysis.

OptimizeUpdateFields,

CTESubstitution,

WindowsSubstitution,

EliminateUnions,

SubstituteUnresolvedOrdinals),

Batch("Disable Hints", Once,

new ResolveHints.DisableHints),

Batch("Hints", fixedPoint,

ResolveHints.ResolveJoinStrategyHints,

ResolveHints.ResolveCoalesceHints),

Batch("Simple Sanity Check", Once,

LookupFunctions),

Batch("Resolution", fixedPoint,

ResolveTableValuedFunctions(v1SessionCatalog) ::

ResolveNamespace(catalogManager) ::

new ResolveCatalogs(catalogManager) ::

ResolveUserSpecifiedColumns ::

ResolveInsertInto ::

ResolveRelations ::

ResolveTables ::

ResolvePartitionSpec ::

ResolveAlterTableCommands ::

AddMetadataColumns ::

DeduplicateRelations ::

ResolveReferences ::

ResolveExpressionsWithNamePlaceholders ::

ResolveDeserializer ::

ResolveNewInstance ::

ResolveUpCast ::

ResolveGroupingAnalytics ::

ResolvePivot ::

ResolveOrdinalInOrderByAndGroupBy ::

ResolveAggAliasInGroupBy ::

ResolveMissingReferences ::

ExtractGenerator ::

ResolveGenerate ::

ResolveFunctions ::

ResolveAliases ::

ResolveSubquery ::

ResolveSubqueryColumnAliases ::

ResolveWindowOrder ::

ResolveWindowFrame ::

ResolveNaturalAndUsingJoin ::

ResolveOutputRelation ::

ExtractWindowExpressions ::

GlobalAggregates ::

ResolveAggregateFunctions ::

TimeWindowing ::

SessionWindowing ::

ResolveInlineTables ::

ResolveHigherOrderFunctions(catalogManager) ::

ResolveLambdaVariables ::

ResolveTimeZone ::

ResolveRandomSeed ::

ResolveBinaryArithmetic ::

ResolveUnion ::

typeCoercionRules ++

Seq(ResolveWithCTE) ++

extendedResolutionRules : \_\*),

Batch("Remove TempResolvedColumn", Once, RemoveTempResolvedColumn),

Batch("Apply Char Padding", Once,

ApplyCharTypePadding),

Batch("Post-Hoc Resolution", Once,

Seq(ResolveCommandsWithIfExists) ++

postHocResolutionRules: \_\*),

Batch("Remove Unresolved Hints", Once,

new ResolveHints.RemoveAllHints),

Batch("Nondeterministic", Once,

PullOutNondeterministic),

Batch("UDF", Once,

HandleNullInputsForUDF,

ResolveEncodersInUDF),

Batch("UpdateNullability", Once,

UpdateAttributeNullability),

Batch("Subquery", Once,

UpdateOuterReferences),

Batch("Cleanup", fixedPoint,

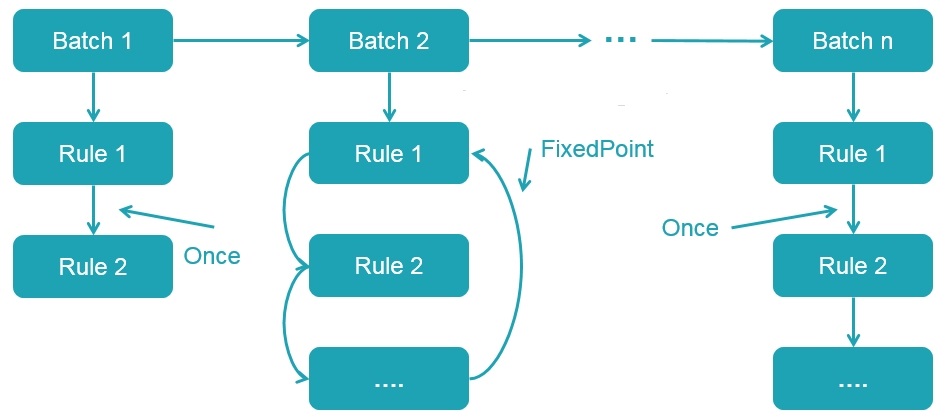
CleanupAliases),

Batch("HandleAnalysisOnlyCommand", Once,

HandleAnalysisOnlyCommand)

)

从上面代码可以看出，多个性质类似的 Rule 组成一个 Batch，比如上面名为 Hints 的 Batch就是由很多个 Hints Rule 组成；而多个 Batch 构成一个 batches。这些 batches 会由 RuleExecutor 执行，先按一个一个 Batch 顺序执行，然后对 Batch 里面的每个 Rule 顺序执行。每个 Batch 会执行一次（Once）或多次（FixedPoint，由 **spark.sql.optimizer.maxIterations** 参数决定），执行过程如下：



所以上面的 SQL 经过这个阶段生成的 Analyzed Logical Plan 如下：

|  |
| --- |
| == Analyzed Logical Plan ==  class: string, sum(degree): decimal(20,1), avg(degree): decimal(14,5)  Aggregate [class#4], [class#4, sum(degree#12) AS sum(degree)#27, avg(degree#12) AS avg(degree)#28]  +- SubqueryAlias temp1  +- Project [sno#0 AS ssno#16, sname#1, ssex#2, sbirthday#3, class#4, sno#10, degree#12, cno#11]  +- Filter ((cast(degree#12 as decimal(10,1)) > cast(cast(60 as decimal(2,0)) as decimal(10,1))) && (cast(sbirthday#3 as string) > 1973-01-01 00:00:00))  +- Join LeftOuter, (sno#0 = sno#10)  :- SubqueryAlias students  : +- Relation[sno#0,sname#1,ssex#2,sbirthday#3,class#4] JDBCRelation(students) [numPartitions=1]  +- SubqueryAlias scores  +- Relation[sno#10,cno#11,degree#12] JDBCRelation(scores) [numPartitions=1] |

从上面的解析过程来看，students和scores表已经被解析成为了带有sno#0 AS ssno#16, sname#1, ssex#2, sbirthday#3, class#4, sno#10, degree#12, cno#11这么具体的字段，其中还有聚合函数

Aggregate [class#4], [class#4, sum(degree#12) AS sum(degree)#27, avg(degree#12) AS avg(degree)#28]，并且最终返回的四个字段的类型也已经确定了class: string, sum(degree): decimal(20,1), avg(degree): decimal(14,5)，而且也已经知道了数据来源是JDBCRelation(students)表和 JDBCRelation(scores)表。

**总结来看Analyzed Logical Plan主要就是干了一些这些事情**

1、确定最终返回字段名称以及返回类型

|  |
| --- |
| class: string, sum(degree): decimal(20,1), avg(degree): decimal(14,5) |

2、确定聚合函数

|  |
| --- |
| Aggregate [class#4], [class#4, sum(degree#12) AS sum(degree)#27, avg(degree#12) AS avg(degree)#28] |

3、确定表当中获取的查询字段

|  |
| --- |
| Project [sno#0 AS ssno#16, sname#1, ssex#2, sbirthday#3, class#4, sno#10, degree#12, cno#11] |

4、确定过滤条件

|  |
| --- |
| Filter ((cast(degree#12 as decimal(10,1)) > cast(cast(60 as decimal(2,0)) as decimal(10,1))) && (cast(sbirthday#3 as string) > 1973-01-01 00:00:00)) |

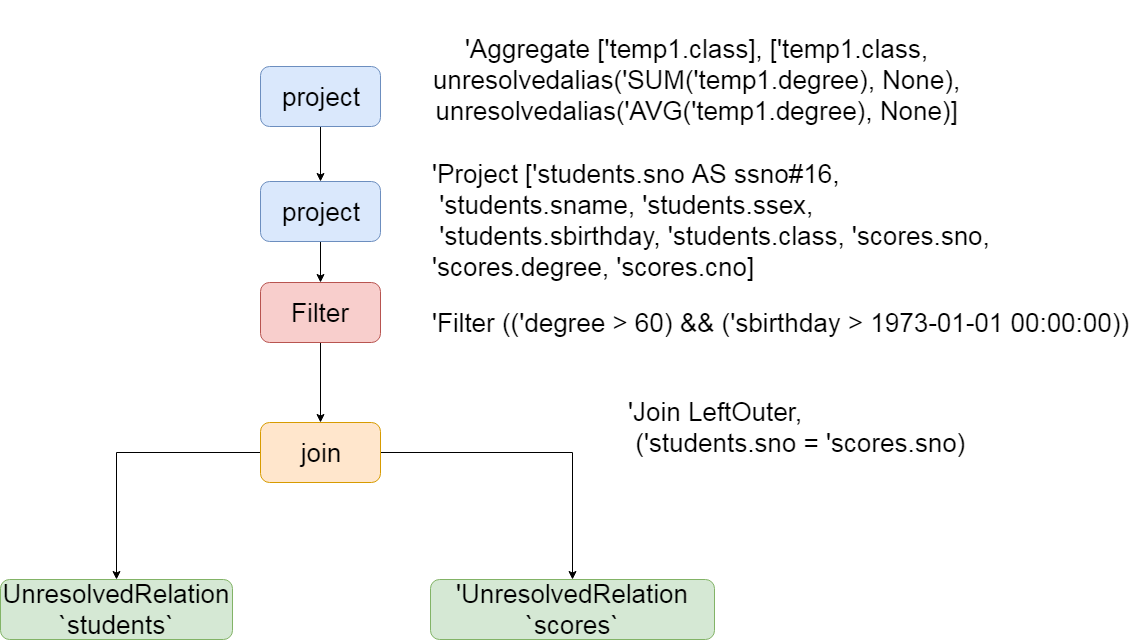
5、确定join方式

|  |
| --- |
| Join LeftOuter, (sno#0 = sno#10) |

6、确定表当中的数据来源以及分区个数

|  |
| --- |
| JDBCRelation(students) [numPartitions=1]  JDBCRelation(scores) [numPartitions=1] |

至此Analyzed Logical Plan已经完成。对比Unresolved Logical Plan到Analyzed Logical Plan 过程如下图





到这里， Analyzed LogicalPlan 就完全生成了

思考题：Spark Catalog 包含那些内容？

### 8.2.3 逻辑优化阶段Optimizer

在前文的绑定逻辑计划阶段对 Unresolved LogicalPlan 进行相关 transform 操作得到了 Analyzed Logical Plan，这个 Analyzed Logical Plan 是可以直接转换成 Physical Plan 然后在 Spark 中执行。但是如果直接这么弄的话，得到的 Physical Plan 很可能不是最优的，因为在实际应用中，很多低效的写法会带来执行效率的问题，需要进一步对Analyzed Logical Plan 进行处理，得到更优的逻辑算子树。于是， 针对 SQL 逻辑算子树的优化器 Optimizer 应运而生。

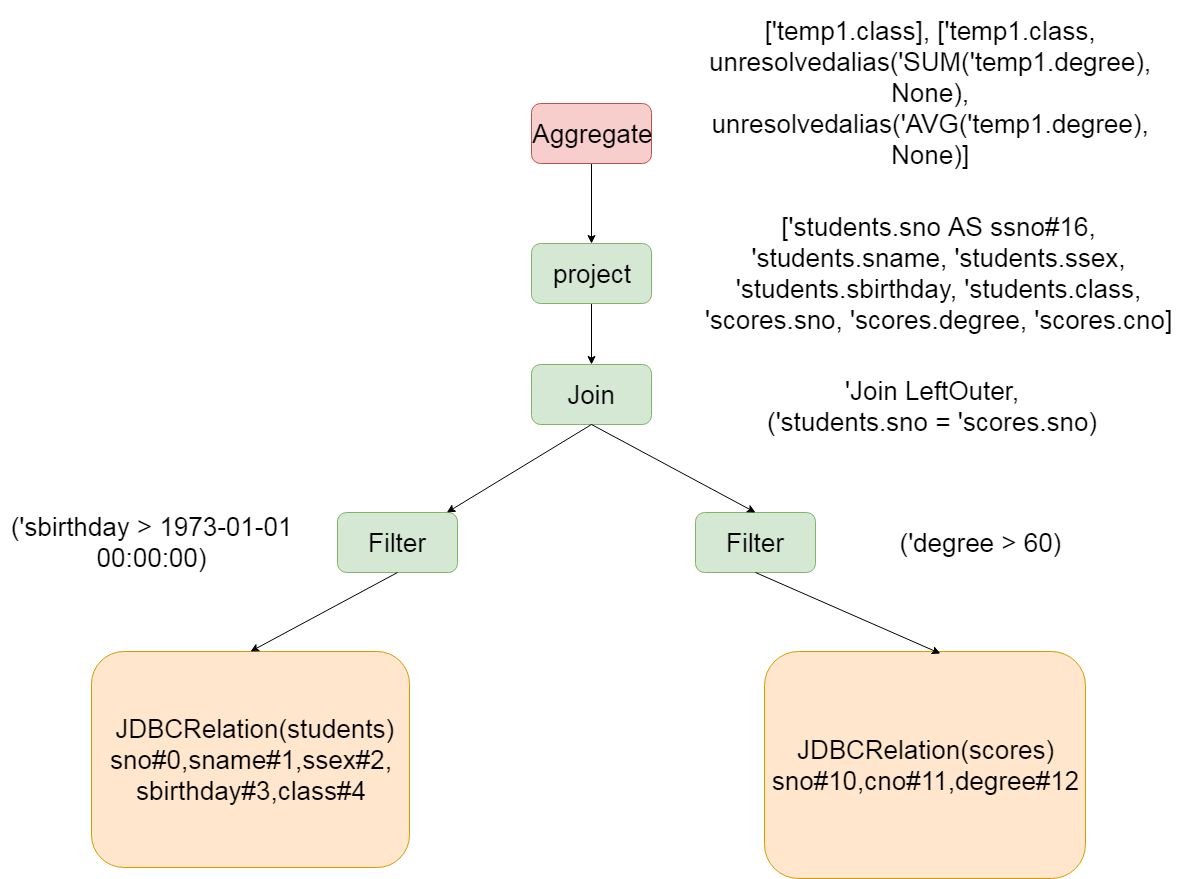
这个阶段的优化器主要是基于规则的（Rule-based Optimizer，简称 RBO），而绝大部分的规则都是启发式规则，也就是基于直观或经验而得出的规则，比如 列裁剪（过滤掉查询不需要使用到的列）、谓词下推（将过滤尽可能地下沉到数据源端）、常量累加（比如 1 + 2 这种事先计算好） 以及常量替换（比如 SELECT \* FROM table WHERE i = 5 AND j = i + 3 可以转换成 SELECT \* FROM table WHERE i = 5 AND j = 8）等等。

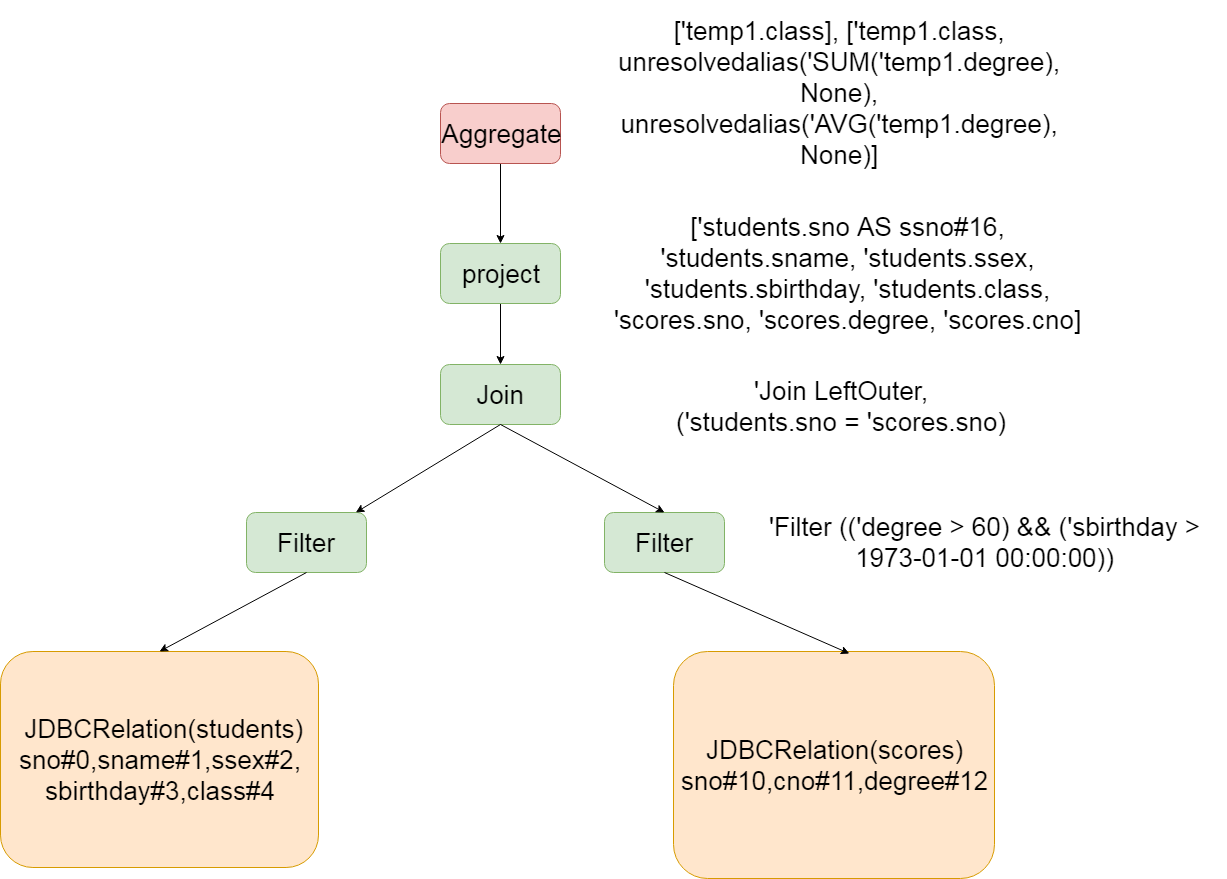
与上面我们介绍绑定逻辑计划阶段类似，这个阶段所有的规则也是实现 Rule 抽象类，多个规则组成一个 Batch，多个 Batch 组成一个 batches，同样也是在 RuleExecutor 中进行执行

**这里按照 Rule 执行顺序一一进行说明。**

**谓词下推**

谓词下推在 SparkQL 是由 PushDownPredicate 实现的，这个过程主要将过滤条件尽可能地下推到底层，最好是数据源。所以针对我们上面介绍的 SQL，使用谓词下推优化得到的逻辑计划如下

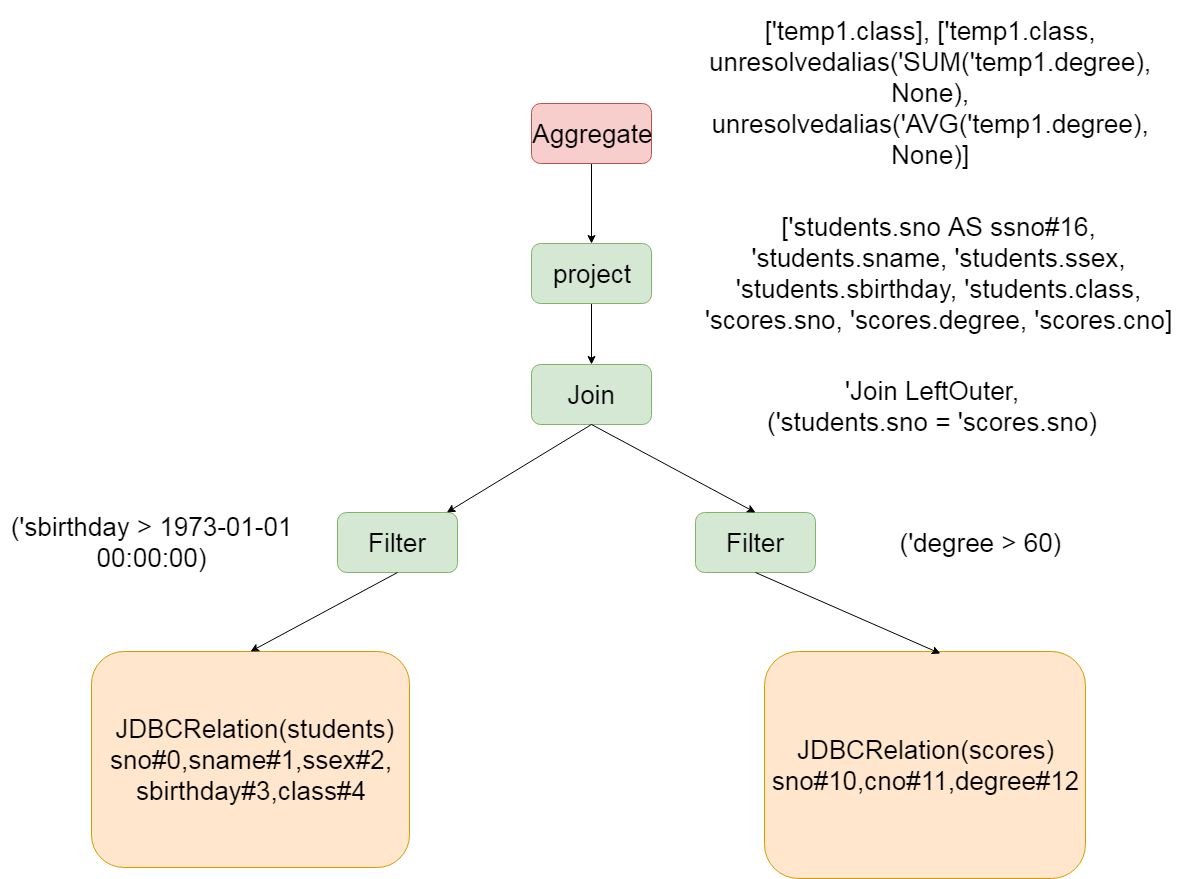


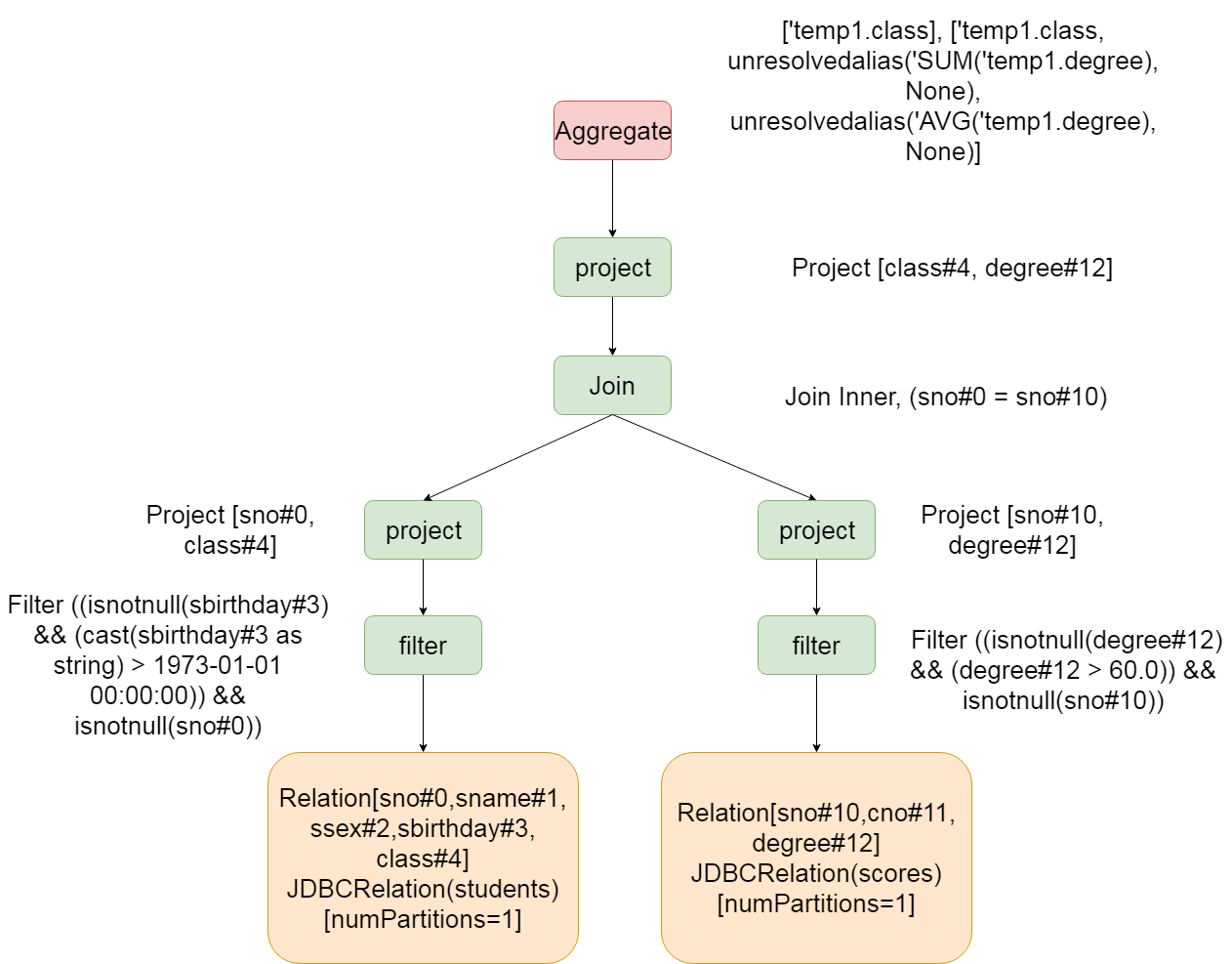


从上图可以看出，谓词下推将 Filter 算子直接下推到 Join 之前了（注意，上图是从下往上看的）。也就是在扫描 student表的时候使用条件过滤条件过滤出满足条件的数据；同时在扫描 scores 表的时候会先使用 isnotnull(degree#12) && (degree#12 > 60) 过滤条件过滤出满足条件的数据。经过这样的操作，可以大大减少 Join 算子处理的数据量，从而加快计算速度。

**列裁剪**

列裁剪在 Spark SQL 是由 ColumnPruning 实现的。因为我们查询的表可能有很多个字段，但是每次查询我们很大可能不需要扫描出所有的字段，这个时候利用列裁剪可以把那些查询不需要的字段过滤掉，使得扫描的数据量减少。所以针对我们上面介绍的 SQL，使用列裁剪优化得到的逻辑计划如下：

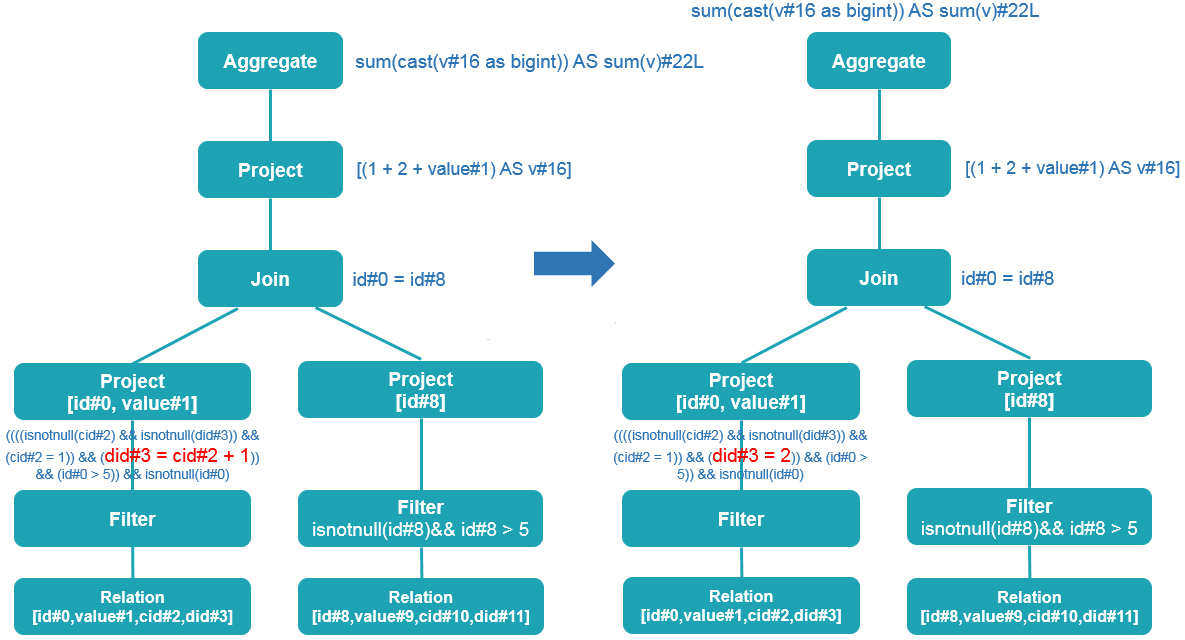




从上图可以看出，经过列裁剪后，students 表只需要查询 sno和 class 两个字段；scores 表只需要查询 sno,degree 字段。这样减少了数据的传输，而且如果底层的文件格式为列存（比如 Parquet），可以大大提高数据的扫描速度的。

**常量替换**

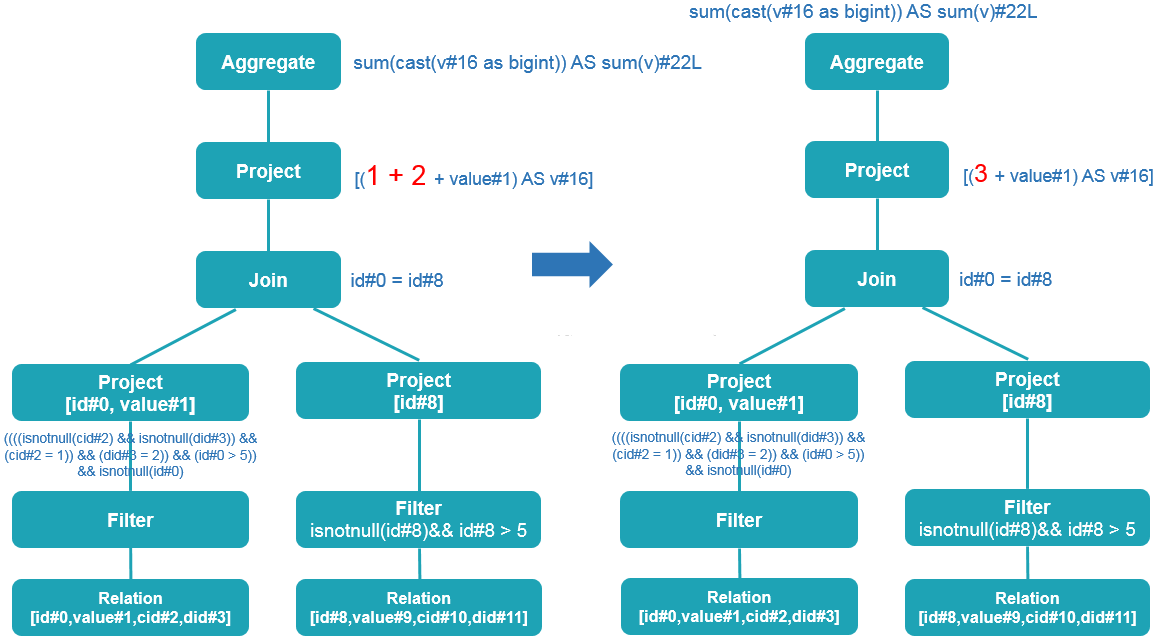
常量替换在 Spark SQL 是由 ConstantPropagation 实现的。也就是将变量替换成常量，比如 SELECT \* FROM table WHERE i = 5 AND j = i + 3 可以转换成 SELECT \* FROM table WHERE i = 5 AND j = 8。这个看起来好像没什么的，但是如果扫描的行数非常多可以减少很多的计算时间的开销的。经过这个优化，得到的逻辑计划如下



我们的查询中有 t1.cid = 1 AND t1.did = t1.cid + 1 查询语句，从里面可以看出 t1.cid 其实已经是确定的值了，所以我们完全可以使用它计算出 t1.did。

**常量累加**

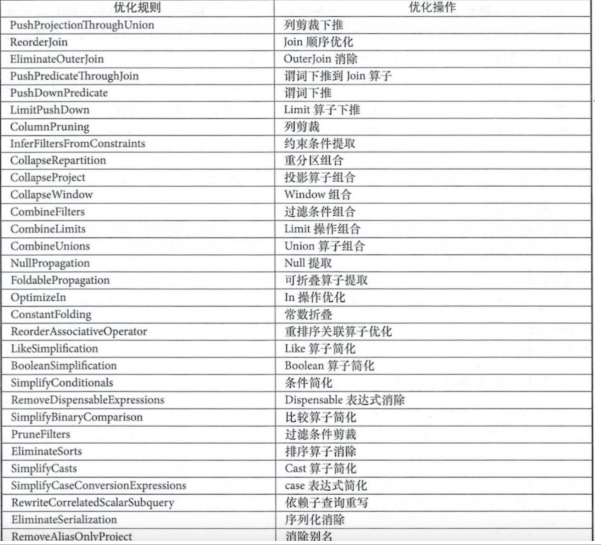
常量累加在 Spark SQL 是由 ConstantFolding 实现的。这个和常量替换类似，也是在这个阶段把一些常量表达式事先计算好。这个看起来改动的不大，但是在数据量非常大的时候可以减少大量的计算，减少 CPU 等资源的使用。经过这个优化，得到的逻辑计划如下：



所以经过上面四个步骤的优化之后，得到的优化之后的逻辑计划为：

|  |
| --- |
| == Optimized Logical Plan ==  Aggregate [class#4], [class#4, sum(degree#12) AS sum(degree)#27, cast((avg(UnscaledValue(degree#12)) / 10.0) as decimal(14,5)) AS avg(degree)#28]  +- Project [class#4, degree#12]  +- Join Inner, (sno#0 = sno#10)  :- Project [sno#0, class#4]  : +- Filter ((isnotnull(sbirthday#3) && (cast(sbirthday#3 as string) > 1973-01-01 00:00:00)) && isnotnull(sno#0))  : +- Relation[sno#0,sname#1,ssex#2,sbirthday#3,class#4] JDBCRelation(students) [numPartitions=1]  +- Project [sno#10, degree#12]  +- Filter ((isnotnull(degree#12) && (degree#12 > 60.0)) && isnotnull(sno#10))  +- Relation[sno#10,cno#11,degree#12] JDBCRelation(scores) [numPartitions=1] |

**Rule 举例：**



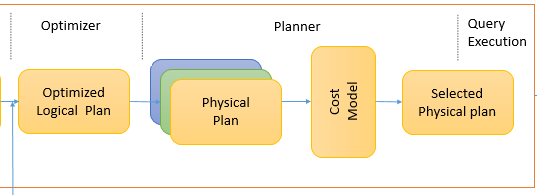
到此为止，优化逻辑阶段基本完成，另外更多的其他优化，参见spark源码：

### <https://github.com/apache/spark/blob/master/sql/catalyst/src/main/scala/org/apache/spark/sql/catalyst/optimizer/Optimizer.scala>

**8.2.4生成可执行的物理计划阶段Physical Plan**

经过前面多个步骤，包括parse，analyzer以及Optimizer等多个阶段，得到经过优化之后的sql语句，但是这个sql语句仍然不能执行，为了能够执行这个sql，最终必须得要翻译成为可以被执行的物理计划，到这个阶段spark就知道该如何执行这个sql了，和前面逻辑计划绑定和优化不一样，这个阶段使用的是策略strategy，而且经过前面介绍的逻辑计划绑定和 Transformations 动作之后，树的类型并没有改变，也就是说：Expression 经过 Transformations 之后得到的还是 Transformations ；Logical Plan 经过 Transformations 之后得到的还是 Logical Plan。而到了这个阶段，经过 Transformations 动作之后，树的类型改变了，由 Logical Plan 转换成 Physical Plan 了。

一个逻辑计划（Logical Plan）经过一系列的策略处理之后，得到多个物理计划（Physical Plans），物理计划在 Spark 是由 SparkPlan 实现的。多个物理计划再经过代价模型（Cost Model）得到选择后的物理计划（Selected Physical Plan），整个过程如下所示：



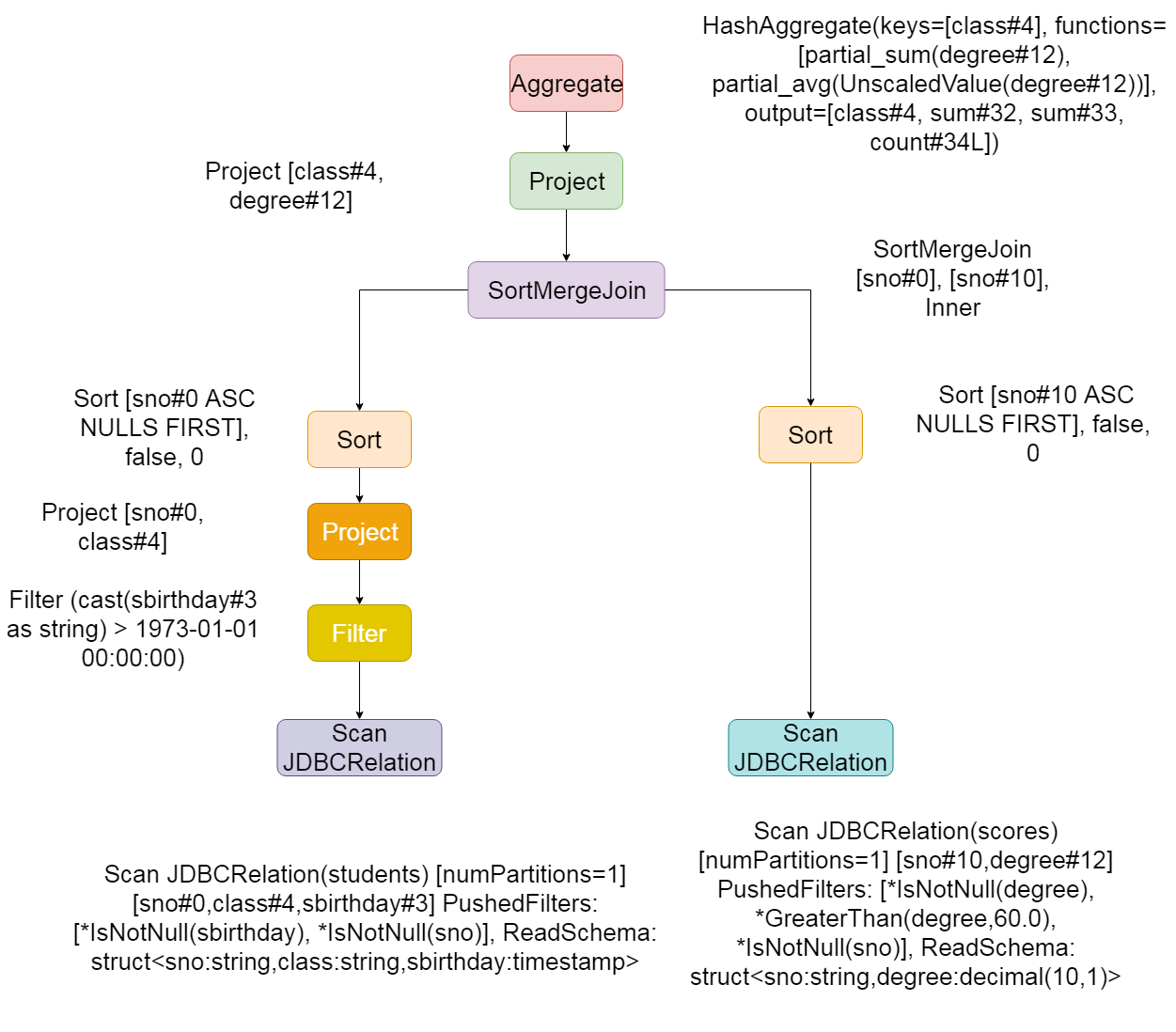
https://issues.apache.org/jira/browse/SPARK-16026

Cost Model 对应的就是基于代价的优化（Cost-based Optimizations，CBO，主要由华为的大佬们实现的，详见 [SPARK-16026](https://www.iteblog.com/redirect.php?url=aHR0cHM6Ly9pc3N1ZXMuYXBhY2hlLm9yZy9qaXJhL2Jyb3dzZS9TUEFSSy0xNjAyNg==&article=true) ），核心思想是计算每个物理计划的代价，然后得到最优的物理计划。

[SPARK-16026](https://www.iteblog.com/redirect.php?url=aHR0cHM6Ly9pc3N1ZXMuYXBhY2hlLm9yZy9qaXJhL2Jyb3dzZS9TUEFSSy0xNjAyNg==&article=true) 引入的 CBO 优化主要是在前面介绍的优化逻辑计划阶段 - Optimizer 阶段进行的，对应的 Rule 为 CostBasedJoinReorder，并且默认是关闭的spark.sql.cbo.joinReorder.enabled ，需要通过 spark.sql.cbo.enabled或 参数开启。 所以到了这个节点，最后得到的物理计划如下：

|  |
| --- |
| == Physical Plan ==  \*(6) HashAggregate(keys=[class#4], functions=[sum(degree#12), avg(UnscaledValue(degree#12))], output=[class#4, sum(degree)#27, avg(degree)#28])  +- Exchange hashpartitioning(class#4, 200)  +- \*(5) HashAggregate(keys=[class#4], functions=[partial\_sum(degree#12), partial\_avg(UnscaledValue(degree#12))], output=[class#4, sum#32, sum#33, count#34L])  +- \*(5) Project [class#4, degree#12]  +- \*(5) SortMergeJoin [sno#0], [sno#10], Inner  :- \*(2) Sort [sno#0 ASC NULLS FIRST], false, 0  : +- Exchange hashpartitioning(sno#0, 200)  : +- \*(1) Project [sno#0, class#4]  : +- \*(1) Filter (cast(sbirthday#3 as string) > 1973-01-01 00:00:00)  : +- \*(1) Scan JDBCRelation(students) [numPartitions=1] [sno#0,class#4,sbirthday#3] PushedFilters: [\*IsNotNull(sbirthday), \*IsNotNull(sno)], ReadSchema: struct<sno:string,class:string,sbirthday:timestamp>  +- \*(4) Sort [sno#10 ASC NULLS FIRST], false, 0  +- Exchange hashpartitioning(sno#10, 200)  +- \*(3) Scan JDBCRelation(scores) [numPartitions=1] [sno#10,degree#12] PushedFilters: [\*IsNotNull(degree), \*GreaterThan(degree,60.0), \*IsNotNull(sno)], ReadSchema: struct<sno:string,degree:decimal(10,1)> |

从上面的结果可以看出，物理计划阶段已经知道数据源是从 JDBC里面读取了，也知道文件的路径，数据类型等。而且在读取文件的时候，直接将过滤条件（PushedFilters）加进去了同时，这个 Join 变成了 SortMergeJoin，



到这里， Physical Plan 就完全生成了

SparkPlan = Physical Plan

SparkPlan =>SparkPlan tree =>RDD的DAG

**8.2.5 代码生成阶段**

从以上多个过程执行完成之后，例如parser，analyzer，Optimizer，physicalPlan等，最终我们得到的物理执行计划，这个物理执行计划标明了整个的代码执行过程当中我们代码层面的执行过程，以及最终要得到的数据字段以及字段类型，也包含了我们对应的数据源的位置，虽然得到了物理执行计划，但是这个物理执行计划想要被执行，最终还是得要生成完整的代码，底层还是基于sparkRDD去进行处理的，spark最后也还会有一些Rule对生成的物理执行计划进行处理，这个处理过程就是prepareForExecution，这些rule规则定义在

org.apache.spark.sql.execution.QueryExecution 这个类当中的这个方法里面

*/*\*\*

\* Prepares a planned [[SparkPlan]] for execution by inserting shuffle operations and internal

\* row format conversions as needed.

\*/

private[execution] def prepareForExecution(

preparations: Seq[Rule[SparkPlan]],

plan: SparkPlan): SparkPlan = {

val planChangeLogger = new PlanChangeLogger[SparkPlan]()

val preparedPlan = preparations.foldLeft(plan) { case (sp, rule) =>

val result = rule.apply(sp)

planChangeLogger.logRule(rule.ruleName, sp, result)

result

}

planChangeLogger.logBatch("Preparations", plan, preparedPlan)

preparedPlan

}

/\*\*

\* Construct a sequence of rules that are used to prepare a planned [[SparkPlan]] for execution.

\* These rules will make sure subqueries are planned, make use the data partitioning and ordering

\* are correct, insert whole stage code gen, and try to reduce the work done by reusing exchanges

\* and subqueries.

\*/

private[execution] def preparations(

sparkSession: SparkSession,

adaptiveExecutionRule: Option[InsertAdaptiveSparkPlan] = None,

subquery: Boolean): Seq[Rule[SparkPlan]] = {

// `AdaptiveSparkPlanExec` is a leaf node. If inserted, all the following rules will be no-op

// as the original plan is hidden behind `AdaptiveSparkPlanExec`.

adaptiveExecutionRule.toSeq ++

Seq(

CoalesceBucketsInJoin, // 在桶表关联的时候做合并

PlanDynamicPruningFilters(sparkSession)// 动态分区列裁剪,

PlanSubqueries(sparkSession), // 子查询物理计划处理

RemoveRedundantProjects, // 从spark计划中删除冗余的ProjectExec节点

EnsureRequirements(), //确保执行计划分区排序正确

// `RemoveRedundantSorts` needs to be added after `EnsureRequirements` to guarantee the same

// number of partitions when instantiating PartitioningCollection.

RemoveRedundantSorts,

DisableUnnecessaryBucketedScan,

ApplyColumnarRulesAndInsertTransitions(

sparkSession.sessionState.columnarRules, outputsColumnar = false),

// 收集生成代码

CollapseCodegenStages()) ++

(if (subquery) {

Nil

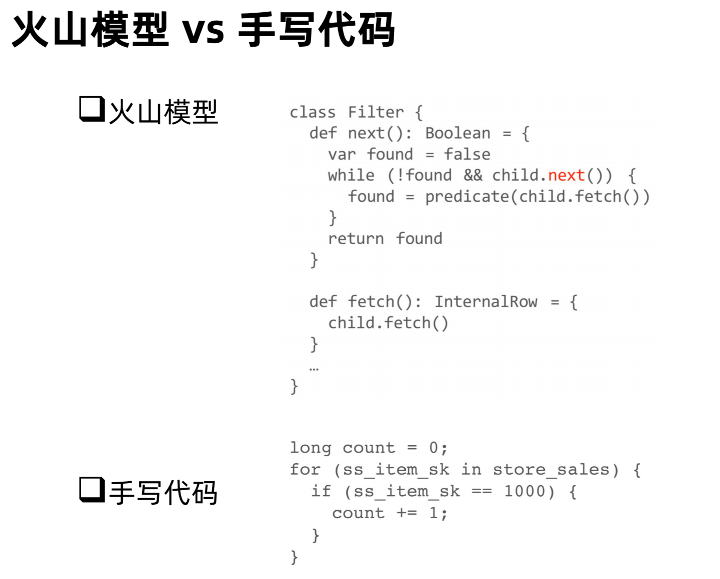
} else {

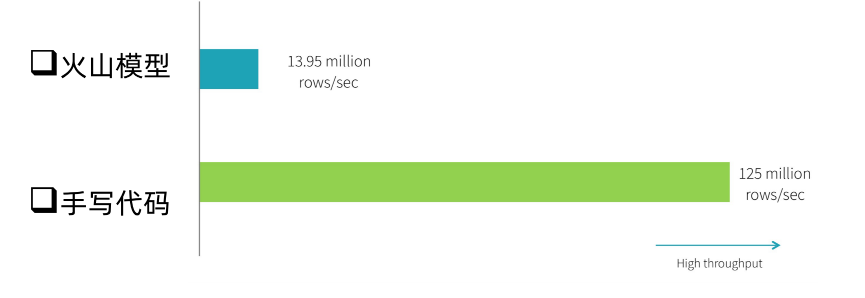
Seq(ReuseExchangeAndSubquery) //节点重用和子查询重用

})

}

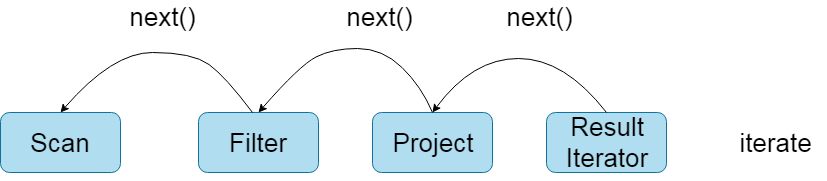
上面的 Rule 中 CollapseCodegenStages 是重头戏，这就是大家熟知的全代码阶段生成，Catalyst 全阶段代码生成的入口就是这个规则。当然，如果需要 Spark 进行全阶段代码生成，需要将 spark.sql.codegen.wholeStage 设置为 true（默认）。





**生成代码与sql解析引擎的区别**

在sparkSQL当中，通过生成代码，来实现sql语句的最终生成，说白了最后底层执行的还是代码，那么为什么要这么麻烦，使用代码的方式来执行我们的sql语句，难道没有sql的解析引擎直接执行sql语句嘛？当然是有的，在spark2.0版本之前使用的都是基于Volcano Iterator Model（参见 [《Volcano-An Extensible and Parallel Query Evaluation System》](http://paperhub.s3.amazonaws.com/dace52a42c07f7f8348b08dc2b186061.pdf)） 来实现sql的解析的，这个是由 Goetz Graefe 在 1993 年提出的，当今绝大多数数据库系统处理 SQL 在底层都是基于这个模型的。这个模型的执行可以概括为：首先数据库引擎会将 SQL 翻译成一系列的关系代数算子或表达式，然后依赖这些关系代数算子逐条处理输入数据并产生结果。每个算子在底层都实现同样的接口，比如都实现了 next() 方法，然后最顶层的算子 next() 调用子算子的 next()，子算子的 next() 在调用孙算子的 next()，直到最底层的 next()，具体过程如下图表示：



Volcano Iterator Model 的优点是抽象起来很简单，很容易实现，而且可以通过任意组合算子来表达复杂的查询。但是缺点也很明显，存在大量的虚函数调用，会引起 CPU 的中断，最终影响了执行效率。[databricks的官方博客](https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html)对比过使用 Volcano Iterator Model 和手写代码的执行效率，结果发现手写的代码执行效率要高出十倍！

所以总结起来就是将sql解析成为代码，比sql引擎直接解析sql语句效率要快，所以spark2.0最终选择使用代码生成的方式来执行sql语句

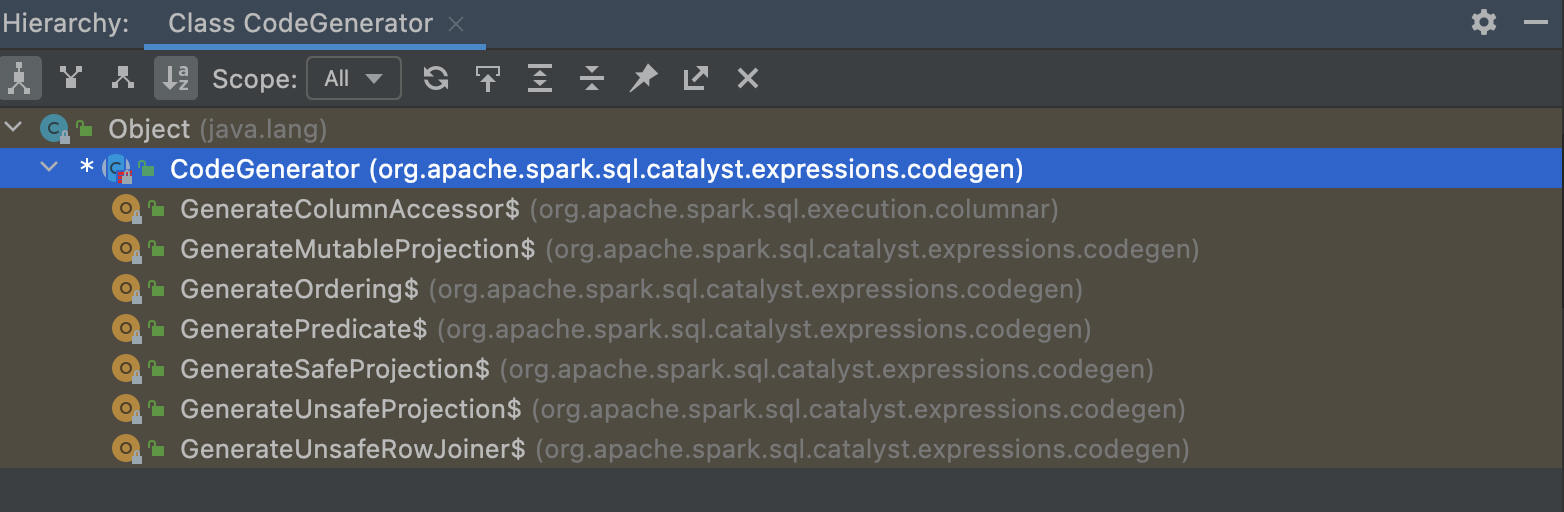
基于上面的发现，从 Apache Spark 2.0 开始，社区开始引入了 Whole-stage Code Generation，参见 [SPARK-12795](https://issues.apache.org/jira/browse/SPARK-12795)，主要就是想通过这个来模拟手写代码，从而提升 Spark SQL 的执行效率。Whole-stage Code Generation 来自于2011年 Thomas Neumann 发表的 [Efficiently Compiling Efficient Query Plans for Modern Hardware](http://www.vldb.org/pvldb/vol4/p539-neumann.pdf)论文，这个也是 Tungsten 计划的一部分。

**Tungsten 代码生成分为三部分：**

* 表达式代码生成（expression codegen）
* 全阶段代码生成（Whole-stage Code Generation）
* 加速序列化和反序列化（speed up serialization/deserialization）

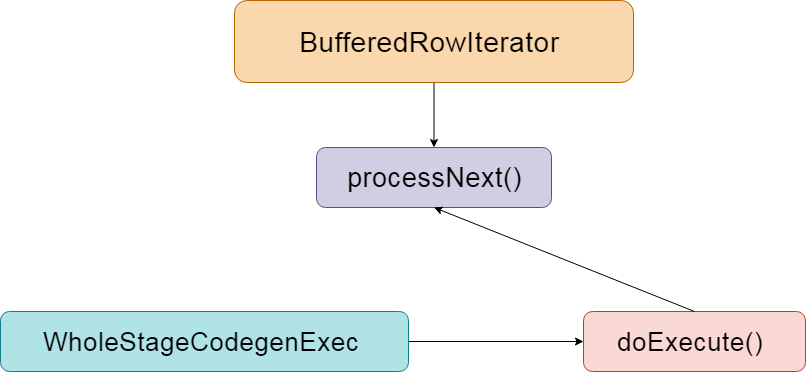
**表达式代码生成（expression codegen）**

这个其实在 Spark 1.x 就有了。表达式代码生成的基类是 org.apache.spark.sql.catalyst.expressions.codegen.CodeGenerator，其下有七个子类：



我们前文的 SQL 生成的逻辑计划中的 (isnotnull(sbirthday#3) && (cast(sbirthday#3 as string) > 1973-01-01 00:00:00) 就是最基本的表达式。它也是一种 Predicate，所以会调用 org.apache.spark.sql.catalyst.expressions.codegen.GeneratePredicate 来生成表达式的代码。

**全阶段代码生成（Whole-stage Code Generation）**



全阶段代码生成（Whole-stage Code Generation），用来将多个处理逻辑整合到单个代码模块中，其中也会用到上面的表达式代码生成。和前面介绍的表达式代码生成不一样，这个是对整个 SQL 过程进行代码生成，前面的表达式代码生成仅对于表达式的。全阶段代码生成都是继承自 org.apache.spark.sql.execution.BufferedRowIterator 的，生成的代码需要实现 processNext() 方法，这个方法会在 org.apache.spark.sql.execution.WholeStageCodegenExec 里面的 doExecute 方法里面被 调用。而这个方法里面的 rdd 会将数据传进生成的代码里面 ，比如我们上文 SQL 这个例子的数据源是 JDBC文件，底层使用 org.apache.spark.sql.execution.RowDataSourceScanExec这个类读取文件，然后生成 inputRDD，这个 rdd 在 WholeStageCodegenExec 类中的

doExecute 方法里面调用生成的代码，然后执行我们各种判断得到最后的结果。WholeStageCodegenExec 类中的 doExecute 方法部分代码如下：

override def doExecute(): RDD[InternalRow] = {

val (ctx, cleanedSource) = doCodeGen()

// try to compile and fallback if it failed

val (\_, compiledCodeStats) = try {

CodeGenerator.compile(cleanedSource)

} catch {

case NonFatal(\_) if !Utils.isTesting && conf.codegenFallback =>

// We should already saw the error message

logWarning(s"Whole-stage codegen disabled for plan (id=$codegenStageId):\n $treeString")

return child.execute()

}

// Check if compiled code has a too large function

if (compiledCodeStats.maxMethodCodeSize > conf.hugeMethodLimit) {

logInfo(s"Found too long generated codes and JIT optimization might not work: " +

s"the bytecode size (${compiledCodeStats.maxMethodCodeSize}) is above the limit " +

s"${conf.hugeMethodLimit}, and the whole-stage codegen was disabled " +

s"for this plan (id=$codegenStageId). To avoid this, you can raise the limit " +

s"`${SQLConf.WHOLESTAGE\_HUGE\_METHOD\_LIMIT.key}`:\n$treeString")

return child.execute()

}

val references = ctx.references.toArray

val durationMs = longMetric("pipelineTime")

// Even though rdds is an RDD[InternalRow] it may actually be an RDD[ColumnarBatch] with

// type erasure hiding that. This allows for the input to a code gen stage to be columnar,

// but the output must be rows.

val rdds = child.asInstanceOf[CodegenSupport].inputRDDs()

assert(rdds.size <= 2, "Up to two input RDDs can be supported")

if (rdds.length == 1) {

rdds.head.mapPartitionsWithIndex { (index, iter) =>

val (clazz, \_) = CodeGenerator.compile(cleanedSource)

val buffer = clazz.generate(references).asInstanceOf[BufferedRowIterator]

buffer.init(index, Array(iter))

new Iterator[InternalRow] {

override def hasNext: Boolean = {

val v = buffer.hasNext

if (!v) durationMs += buffer.durationMs()

v

}

override def next: InternalRow = buffer.next()

}

}

} else {

// Right now, we support up to two input RDDs.

rdds.head.zipPartitions(rdds(1)) { (leftIter, rightIter) =>

Iterator((leftIter, rightIter))

// a small hack to obtain the correct partition index

}.mapPartitionsWithIndex { (index, zippedIter) =>

val (leftIter, rightIter) = zippedIter.next()

val (clazz, \_) = CodeGenerator.compile(cleanedSource)

val buffer = clazz.generate(references).asInstanceOf[BufferedRowIterator]

buffer.init(index, Array(leftIter, rightIter))

new Iterator[InternalRow] {

override def hasNext: Boolean = {

val v = buffer.hasNext

if (!v) durationMs += buffer.durationMs()

v

}

override def next: InternalRow = buffer.next()

}

}

}

}

在WholeStageCodegenExec 这个类的注释当中也说明了，最终生成的代码过程如下

/\*\*

\* WholeStageCodegen compiles a subtree of plans that support codegen together into single Java

\* function.

\*

\* Here is the call graph of to generate Java source (plan A supports codegen, but plan B does not):

\*

\* WholeStageCodegen Plan A FakeInput Plan B

\* =========================================================================

\*

\* -> execute()

\* |

\* doExecute() ---------> inputRDDs() -------> inputRDDs() ------> execute()

\* |

\* +-----------------> produce()

\* |

\* doProduce() -------> produce()

\* |

\* doProduce()

\* |

\* doConsume() <--------- consume()

\* |

\* doConsume() <-------- consume()

\*

\* SparkPlan A should override `doProduce()` and `doConsume()`.

\*

\* `doCodeGen()` will create a `CodeGenContext`, which will hold a list of variables for input,

\* used to generated code for [[BoundReference]].

\*/

相比 Volcano Iterator Model，全阶段代码生成的执行过程如下：



通过引入全阶段代码生成，大大减少了虚函数的调用，减少了 CPU 的调用，使得 SQL 的执行速度有很大提升。

**代码编译**

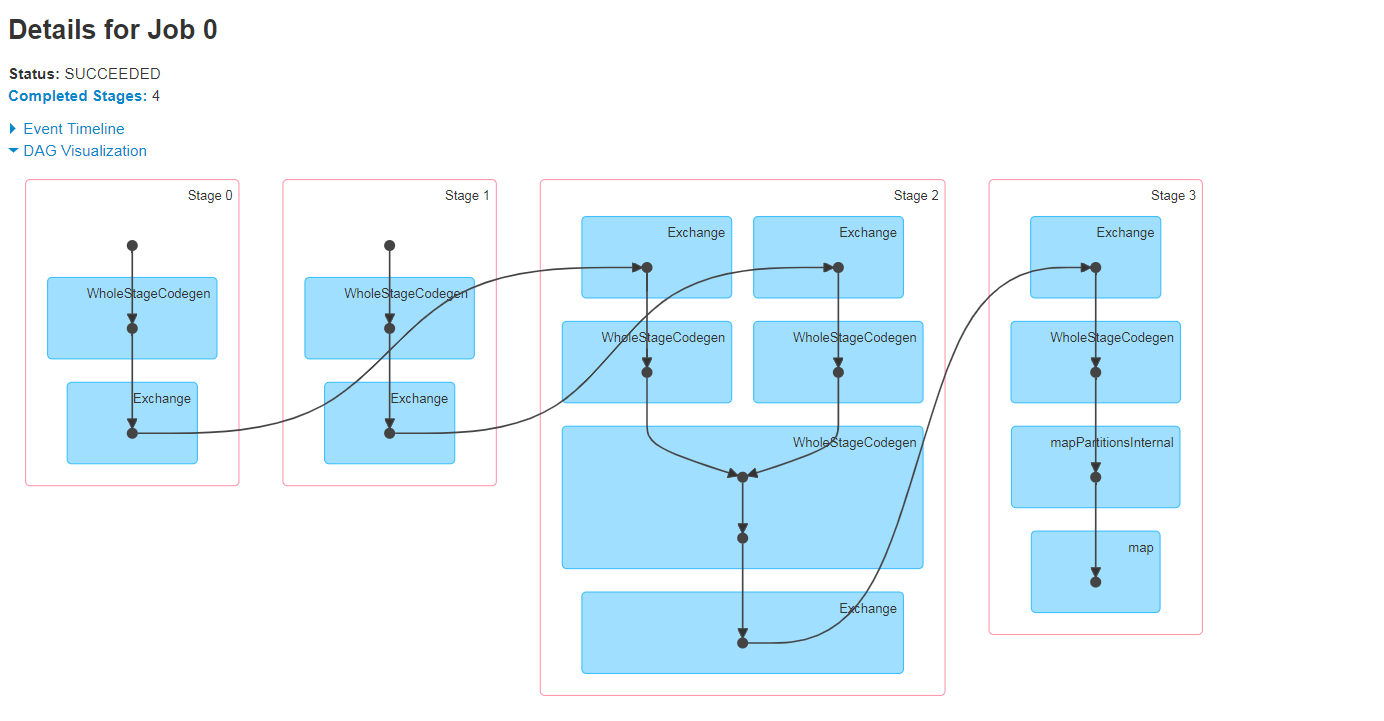
生成代码之后需要解决的另一个问题是如何将生成的代码进行编译然后加载到同一个 JVM 中去。在早期 Spark 版本是使用 Scala 的 Reflection 和 Quasiquotes 机制来实现代码生成的。Quasiquotes 是一个简洁的符号，可以让我们轻松操作 Scala 语法树，具体参见 [这里](https://docs.scala-lang.org/overviews/quasiquotes/intro.html)。虽然 Quasiquotes 可以很好的为我们解决代码生成等相关的问题，但是带来的新问题是编译代码时间比较长（大约 50ms - 500ms）！所以社区不得不默认关闭表达式代码生成。

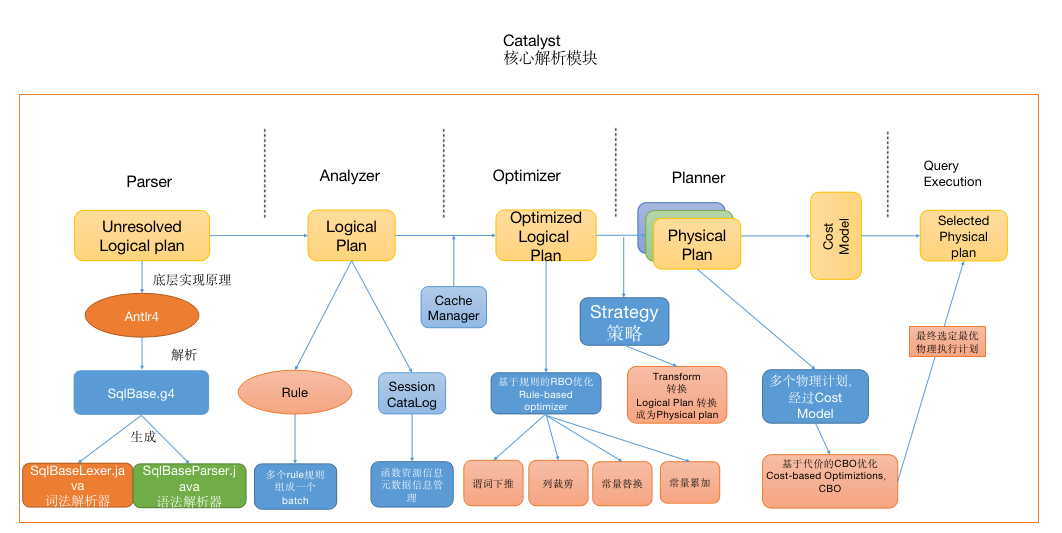
为了解决这个问题，Spark 引入了 Janino 项目，参见 [SPARK-7956](https://issues.apache.org/jira/browse/SPARK-7956)。Janino 是一个超级小但又超级快的 Java™ 编译器. 它不仅能像 javac 工具那样将一组源文件编译成字节码文件，还可以对一些 Java 表达式，代码块，类中的文本(class body)或者内存中源文件进行编译，并把编译后的字节码直接加载到同一个 JVM 中运行。Janino 不是一个开发工具, 而是作为运行时的嵌入式编译器，比如作为表达式求值的翻译器或类似于 JSP 的服务端页面引擎，关于 Janino 的更多知识请参见[这里](https://janino-compiler.github.io/janino/)。通过引入了 Janino 来编译生成的代码，结果显示 SQL 表达式的编译时间减少到 5ms。在 Spark 中使用了 ClassBodyEvaluator 来编译生成之后的代码，参见 org.apache.spark.sql.catalyst.expressions.codegen.CodeGenerator。

需要注意的是，代码生成是在 Driver 端进行的，而代码编译是在 Executor 端进行的。

**SQL执行**

终于到了 SQL 真正执行的地方了。这个时候 Spark 会执行上阶段生成的代码，然后得到最终的结果，DAG 执行图如下：

****



从上面可以看得出来，sparkSQL的执行主要经过了这么几个大的步骤

1）、输入sql，dataFrame或者dataSet

2）、经过Catalyst过程，生成最终我们得到的最优的物理执行计划

**a）、parser阶段**

主要是通过Antlr4解析SqlBase.g4 ，所有Spark支持的语法方式都是定义在sqlBase.g4里面了，如果需要扩展Spark SQL的语法，我们只需要扩展sqlBase.g4即可，通过antlr4解析sqlBase.g4文件，生成了我们的语法解析器SqlBaseLexer.java和词法解析器SqlBaseParser.java

parse阶段 => antlr4 => 解析 => SqlBase.g4 =>得到 => 语法解析器SqlBaseLexer.java + 词法解析器SqlBaseParser.java

**b）、analyzer阶段**

使用基于Rule的规则解析以及Session Catalog来实现函数资源信息和元数据管理信息

Analyzer 阶段 => 使用 Rule + Session Catalog =>多个rule => 组成一个batch session CataLog => 保存函数资源信息以及元数据信息等

**c）、optimizer阶段**

optimizer调优阶段 => 基于规则的RBO优化rule-based optimizer => 谓词下推 + 列剪枝 + 常量替换 + 常量累加

**d）、planner阶段**

通过analyzer生成多个物理计划 ==》 经过Cost Model进行最优选择 ==》基于代价的CBO优化 ==》 最终选定得到的最优物理执行计划

**e）、选定最终的物理计划，准备执行**

最终选定的最优物理执行计划 ==》 准备生成代码去开始执行

3）、将最终得到的物理执行计划进行代码生成，提交代码去执行我们的最终任务

### 8.4 Spark SQL Join原理分析☆☆☆

**8.4.1 Spark SQL Join 背景**

数据分析中将两个数据集进行 Join 操作是很常见的场景。在 Spark 的物理计划阶段，Spark 的 Join Selection 类会根据 Join hints 策略、Join 表的大小、 Join 是等值 Join 还是不等值以及参与 Join 的 key 是否可以排序等条件来选择最终的 Join 策略，最后 Spark 会利用选择好的 Join 策略执行最终的计算。当前 Spark 一共支持五种Join 策略：

* **Broadcast hash join (BHJ)**
* Shuffle hash join（SHJ）
* **Shuffle sort merge join (SMJ)**
* Shuffle-and-replicate nested loop join，又称笛卡尔积（Cartesian product join)
* Broadcast nested loop join (BNLJ)

其中 BHJ 和 SMJ 这两种 Join 策略是我们运行 Spark 作业最常见的。JoinSelection 会先根据 Join 的 Key 为等值 Join 来选择 Broadcast hash join、Shuffle hash join 以及 Shuffle sort merge join 中的一个；如果 Join 的 Key 为不等值 Join 或者没有指定Join 条件，则会选择 Broadcast nested loop join 或 Shuffle-and-replicate nested loop join。

不同的 Join 策略在执行上效率差别很大，了解每种 Join 策略的执行过程和适用条件是很有必要的。

**8.4.2 Hash Join原理**

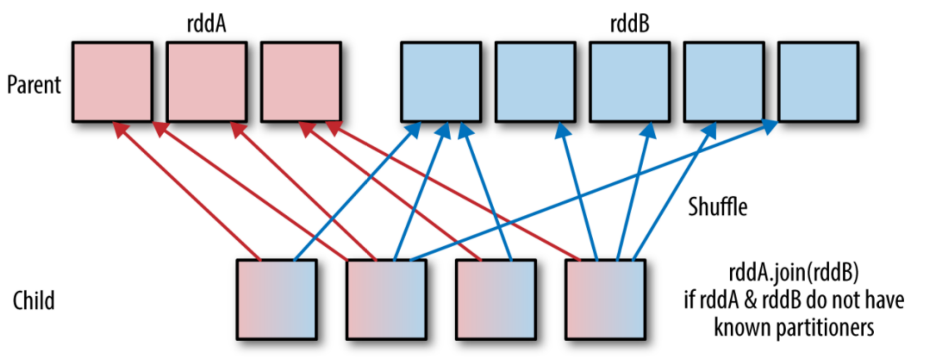
先来看看这样一条SQL语句：select \* from order,item where item.id = order.i\_id，参与join的两张表是order和item，join key分别是item.id以及order.i\_id。现在假设Join采用的是hash join算法，整个过程会经历三步：

* 确定Build Table以及Probe Table：这个概念比较重要，Build Table会被构建成以join key为key的hash table，而Probe Table使用join key在这张hash table表中寻找符合条件的行，然后进行join链接。Build表和Probe表是Spark决定的。通常情况下，小表会被作为Build Table，较大的表会被作为Probe Table。
* 构建Hash Table：依次读取Build Table(item)的数据，对于每一条数据根据Join Key(item.id)进行hash，hash到对应的bucket中(类似于HashMap的原理)，最后会生成一张HashTable，HashTable会缓存在内存中，如果内存放不下会dump到磁盘中。
* 匹配：生成Hash Table后，在依次扫描Probe Table(order)的数据，使用相同的hash函数(在spark中，实际上就是要使用相同的partitioner)在Hash Table中寻找hash(join key)相同的值，如果匹配成功就将两者join在一起。

**8.4.3 Shuffle Hash Join**

当表中的数据比较大，又不适合使用广播，这个时候就可以考虑使用 Shuffle Hash Join。

Shuffle Hash Join 同样是在大表和小表进行 Join 的时候选择的一种策略。它的计算思想是：把大表和小表按照相同的分区算法和分区数进行分区（根据参与 Join 的keys 进行分区），这样就保证了 hash 值一样的数据都分发到同一个分区中，然后在同一个 Executor 中两张表 hash 值一样的分区就可以在本地进行 hash Join 了。 在进行 Join 之前，还会对小表的分区构建 Hash Map。Shuffle hash join 利用了分治思想，把大问题拆解成小问题去解决。



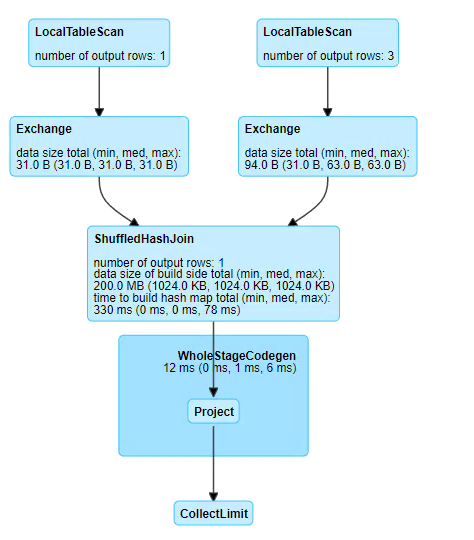
要启用 Shuffle Hash Join 必须满足以下条件：

* 仅支持等值 Join，不要求参与 Join 的 Keys 可排序
* spark.sql.join.preferSortMergeJoin 参数必须设置为 false，参数是从 Spark 2.0.0 版本引入的，默认值为 true，也就是默认情况下选择 Sort Merge Join
* 小表的大小（plan.stats.sizeInBytes）必须小于

spark.sql.autoBroadcastJoinThreshold \* spark.sql.shuffle.partitions（默认值 200）而且小表大小（stats.sizeInBytes）的三倍必须小于等于大表的大小

（stats.sizeInBytes），也就是 a.stats.sizeInBytes \* 3 < = b.stats.sizeInBytes

使用示例：



// 因为我们下面测试数据都很小，所以我们先把 BroadcastJoin 关闭

spark.conf.set("spark.sql.autoBroadcastJoinThreshold", 1)

// 为了启用 Shuffle Hash Join 必须将 spark.sql.join.preferSortMergeJoin 设置为 false

// spark.conf.set("spark.sql.join.preferSortMergeJoin", false)

val df1 = Seq(

(2, "wangwu")

).toDF("id", "info")

val df2 = Seq(

(0, "zhangsan"),

(1, "lisi"),

(2, "wangwu")

).toDF("id", "info")

val r2 = df1.join(df2, Seq("id"), "inner")

r2.explain()

/\* == Physical Plan ==

AdaptiveSparkPlan isFinalPlan=false

+- Project [id#38, info#39, info#8]

+- ShuffledHashJoin [id#38], [id#7], Inner, BuildLeft

:- Exchange hashpartitioning(id#38, 200), ENSURE\_REQUIREMENTS, [id=#74]

: +- LocalTableScan [id#38, info#39]

+- Exchange hashpartitioning(id#7, 200), ENSURE\_REQUIREMENTS, [id=#75]

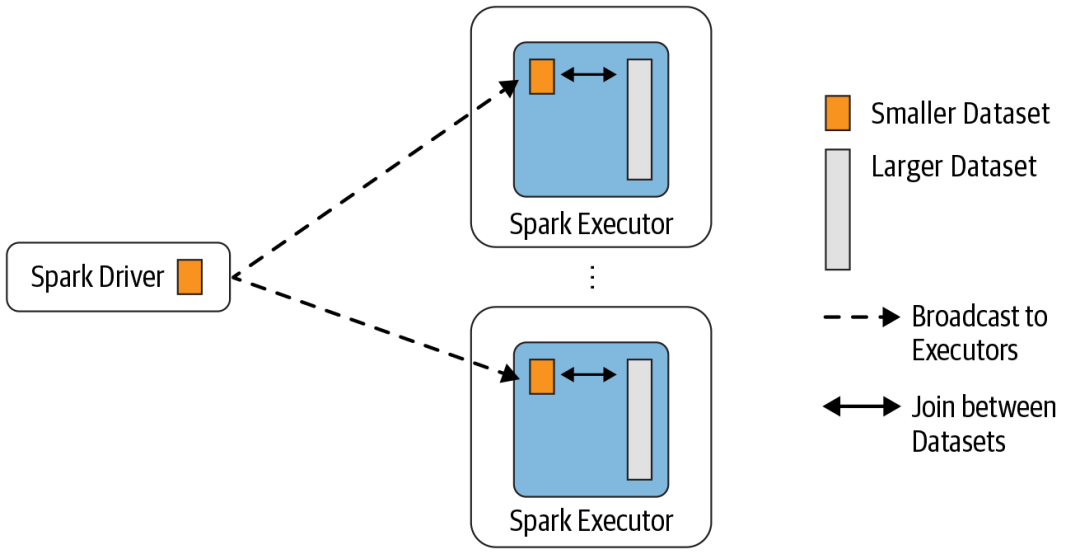
+- LocalTableScan [id#7, info#8]\*/

**8.4.4 Broadcast Hash Join**

Broadcast Hash Join 的实现是将小表的数据广播到 Spark 所有的 Executor 端，这个广播过程和我们自己去广播数据没什么区别：

* 利用 collect 算子将小表的数据从 Executor 端拉到 Driver 端
* 在 Driver 端调用 sparkContext.broadcast 广播到所有 Executor 端
* 在 Executor 端使用广播的数据与大表进行 Join 操作（实际上是执行map操作）

这种 Join 策略避免了 Shuffle 操作。一般而言，Broadcast Hash Join 会比其他 Join策略执行的要快。



使用这种 Join 策略必须满足以下条件：

* 小表的数据必须很小，可以通过 spark.sql.autoBroadcastJoinThreshold 参数来配置，默认是 10MB
* 如果内存比较大，可以将阈值适当加大
* 将 spark.sql.autoBroadcastJoinThreshold 参数设置为 -1，可以关闭这种连接方式
* 只能用于等值 Join，不要求参与 Join 的 keys 可排序

使用示例：

import spark.implicits.\_

val df = Seq(

(0, "zhangsan"),

(1, "lisi"),

(2, "wangwu")

).toDF("id", "info")

val r = df.join(df, Seq("id"), "inner")

r.explain()

r.show()

/\*

== Physical Plan ==

AdaptiveSparkPlan isFinalPlan=false

+- Project [id#7, info#8, info#14]

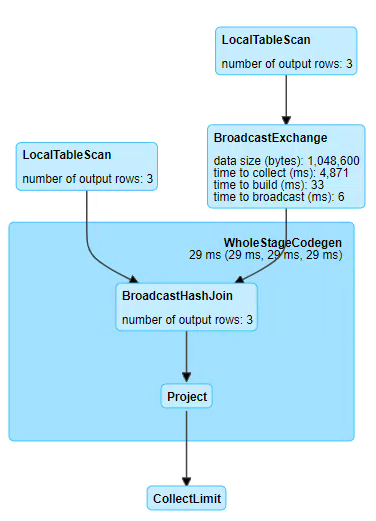
+- BroadcastHashJoin [id#7], [id#13], Inner, BuildRight, false

:- LocalTableScan [id#7, info#8]

+- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0, int, false] as bigint)),false), [id=#16]

+- LocalTableScan [id#13, info#14]

\*/



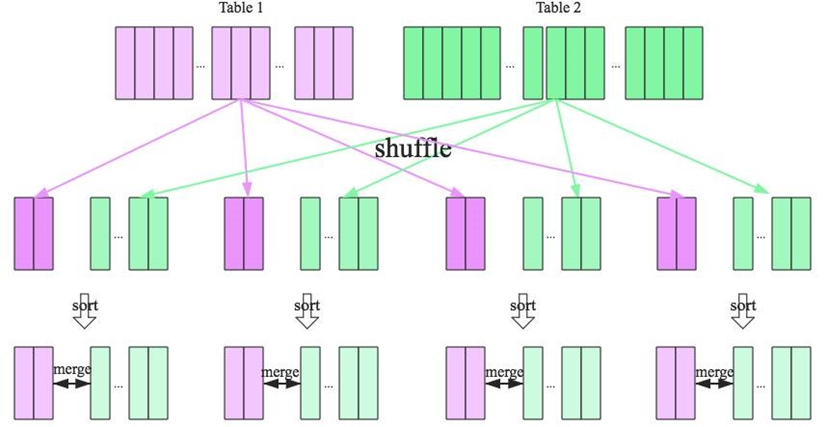
**8.4.5 Shuffle Sort Merge Join**

前面两种 Join 策略对表的大小都有条件的，如果参与 Join 的表都很大，这时候就得考虑用 Shuffle Sort Merge Join 了。

Shuffle Sort Merge Join 的实现思想：

* 将两张表按照 join key 进行shuffle，保证join key值相同的记录会被分在相应的分区
* 对每个分区内的数据进行排序
* 排序后再对相应的分区内的记录进行连接

无论分区有多大，Sort Merge Join都不用把一侧的数据全部加载到内存中，而是即用即丢；因为两个序列都有序。从头遍历，碰到key相同的就输出，如果不同，左边小就继续取左边，反之取右边。从而大大提高了大数据量下sql join的稳定性。



要启用 Shuffle Sort Merge Join 必须满足以下条件：

* 仅支持等值 Join，并且要求参与 Join 的 Keys 可排序

使用示例：

// 因为我们下面测试数据都很小，所以我们先把 BroadcastJoin 关闭

spark.conf.set("spark.sql.autoBroadcastJoinThreshold", 1)

// 为了启用 Shuffle Hash Join 必须将 spark.sql.join.preferSortMergeJoin 设置为 false

val df3 = Seq(

(2, "wangwu")

).toDF("id", "info")

val df4 = Seq(

(0, "zhangsan"),

(1, "lisi"),

(2, "wangwu")

).toDF("id", "info")

val r3 = df1.join(df2, Seq("id"), "inner")

r3.explain()

/\* == Physical Plan ==

AdaptiveSparkPlan isFinalPlan=false

+- Project [id#38, info#39, info#50]

+- SortMergeJoin [id#38], [id#49], Inner

:- Sort [id#38 ASC NULLS FIRST], false, 0

: +- Exchange hashpartitioning(id#38, 200), ENSURE\_REQUIREMENTS, [id=#97]

: +- LocalTableScan [id#38, info#39]

+- Sort [id#49 ASC NULLS FIRST], false, 0

+- Exchange hashpartitioning(id#49, 200), ENSURE\_REQUIREMENTS, [id=#98]

+- LocalTableScan [id#49, info#50]\*/



**8.4.6 Cartesian product join**

如果 Spark 中两张参与 Join 的表没指定连接条件，那么会产生 Cartesian product join，这个 Join 得到的结果其实就是两张表行数的乘积。

**使用示例：**

spark.sql("set spark.sql.crossJoin.enabled=true")

val df5 = Seq(

(0, "1111"),

(1, "2222"),

(2, "3333")

).toDF("id", "info")

val df6 = Seq(

(0, "zhangsan"),

(1, "lisi"),

(2, "wangwu")

).toDF("id", "info")

val r4 = df5.join(df6, Nil, "inner")

// df5.crossJoin(df6)

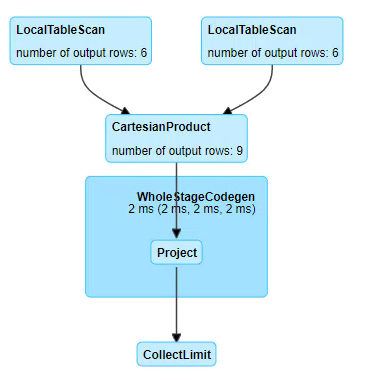
/\* == Physical Plan ==

CartesianProduct

:- LocalTableScan [id#38, info#39]

+- LocalTableScan [id#49, info#50]\*/

r4.explain()



**8.4.7 Broadcast nested loop join**

可以把 Broadcast nested loop join 的执行看做下面的计算：

|  |
| --- |
| for record\_1 in relation\_1:  for record\_2 in relation\_2:  # join condition is executed |

可以看出 Broadcast nested loop join 在某些情况会对某张表重复扫描多次，效率非常低下。从名字可以看出，这种 join 会根据相关条件对小表进行广播，以减少表的扫描次数。

**触发广播的需要满足以下三个条件之一：**

* right outer join 是会广播左表；
* left outer, left semi, left anti 或者 existence join 时会广播右表
* inner join 的时候两张表都会广播。

适用条件：

Broadcast nested loop join 支持等值和不等值 Join，支持所有的 Join 类型。

**使用示例：**

val df7 = Seq(

(0, "1111"),

(1, "2222"),

(2, "3333")

).toDF("id", "info")

val df8 = Seq(

(0, "zhangsan"),

(1, "lisi"),

(2, "wangwu")

).toDF("id", "info")

val r5 = df7.join(df8, Nil, "left\_outer")

r5.explain()

/\* == Physical Plan ==

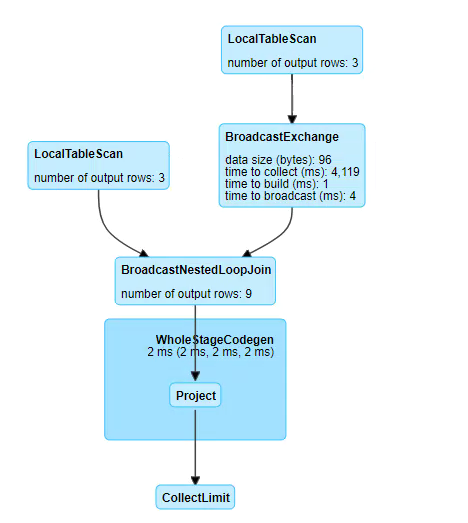
AdaptiveSparkPlan isFinalPlan=false

+- BroadcastNestedLoopJoin BuildRight, LeftOuter

:- LocalTableScan [id#124, info#125]

+- BroadcastExchange IdentityBroadcastMode, [id=#129]

+- LocalTableScan [id#135, info#136]\*/



**8.4.8 Spark 如何选择Join策略**

Spark 有五种 Join 策略，那么 Spark 是按照什么顺序来选择呢？前面我们也说了，Spark 的 Join 策略是在 JoinSelection 类里面实现的，关键代码如下

org.apache.spark.sql.execution.JoinSelection

def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {

// If it is an equi-join, we first look at the join hints w.r.t. the following order:

// 1. broadcast hint: pick broadcast hash join if the join type is supported. If both sides

// have the broadcast hints, choose the smaller side (based on stats) to broadcast.

// 2. sort merge hint: pick sort merge join if join keys are sortable.

// 3. shuffle hash hint: We pick shuffle hash join if the join type is supported. If both

// sides have the shuffle hash hints, choose the smaller side (based on stats) as the

// build side.

// 4. shuffle replicate NL hint: pick cartesian product if join type is inner like.

//

// If there is no hint or the hints are not applicable, we follow these rules one by one:

// 1. Pick broadcast hash join if one side is small enough to broadcast, and the join type

// is supported. If both sides are small, choose the smaller side (based on stats)

// to broadcast.

// 2. Pick shuffle hash join if one side is small enough to build local hash map, and is

// much smaller than the other side, and `spark.sql.join.preferSortMergeJoin` is false.

// 3. Pick sort merge join if the join keys are sortable.

// 4. Pick cartesian product if join type is inner like.

// 5. Pick broadcast nested loop join as the final solution. It may OOM but we don't have

// other choice.

case j @ ExtractEquiJoinKeys(joinType, leftKeys, rightKeys, nonEquiCond, left, right, hint) =>

def createBroadcastHashJoin(onlyLookingAtHint: Boolean) = {

getBroadcastBuildSide(left, right, joinType, hint, onlyLookingAtHint, conf).map {

buildSide =>

Seq(joins.BroadcastHashJoinExec(

leftKeys,

rightKeys,

joinType,

buildSide,

nonEquiCond,

planLater(left),

planLater(right)))

}

}

def createShuffleHashJoin(onlyLookingAtHint: Boolean) = {

getShuffleHashJoinBuildSide(left, right, joinType, hint, onlyLookingAtHint, conf).map {

buildSide =>

Seq(joins.ShuffledHashJoinExec(

leftKeys,

rightKeys,

joinType,

buildSide,

nonEquiCond,

planLater(left),

planLater(right)))

}

}

def createSortMergeJoin() = {

if (RowOrdering.isOrderable(leftKeys)) {

Some(Seq(joins.SortMergeJoinExec(

leftKeys, rightKeys, joinType, nonEquiCond, planLater(left), planLater(right))))

} else {

None

}

}

def createCartesianProduct() = {

if (joinType.isInstanceOf[InnerLike]) {

// `CartesianProductExec` can't implicitly evaluate equal join condition, here we should

// pass the original condition which includes both equal and non-equal conditions.

Some(Seq(joins.CartesianProductExec(planLater(left), planLater(right), j.condition)))

} else {

None

}

}

def createJoinWithoutHint() = {

createBroadcastHashJoin(false)

.orElse(createShuffleHashJoin(false))

.orElse(createSortMergeJoin())

.orElse(createCartesianProduct())

.getOrElse {

// This join could be very slow or OOM

val buildSide = getSmallerSide(left, right)

Seq(joins.BroadcastNestedLoopJoinExec(

planLater(left), planLater(right), buildSide, joinType, nonEquiCond))

}

}

createBroadcastHashJoin(true)

.orElse { if (hintToSortMergeJoin(hint)) createSortMergeJoin() else None }

.orElse(createShuffleHashJoin(true))

.orElse { if (hintToShuffleReplicateNL(hint)) createCartesianProduct() else None }

.getOrElse(createJoinWithoutHint())

case j @ ExtractSingleColumnNullAwareAntiJoin(leftKeys, rightKeys) =>

Seq(joins.BroadcastHashJoinExec(leftKeys, rightKeys, LeftAnti, BuildRight,

None, planLater(j.left), planLater(j.right), isNullAwareAntiJoin = true))

// If it is not an equi-join, we first look at the join hints w.r.t. the following order:

// 1. broadcast hint: pick broadcast nested loop join. If both sides have the broadcast

// hints, choose the smaller side (based on stats) to broadcast for inner and full joins,

// choose the left side for right join, and choose right side for left join.

// 2. shuffle replicate NL hint: pick cartesian product if join type is inner like.

//

// If there is no hint or the hints are not applicable, we follow these rules one by one:

// 1. Pick broadcast nested loop join if one side is small enough to broadcast. If only left

// side is broadcast-able and it's left join, or only right side is broadcast-able and

// it's right join, we skip this rule. If both sides are small, broadcasts the smaller

// side for inner and full joins, broadcasts the left side for right join, and broadcasts

// right side for left join.

// 2. Pick cartesian product if join type is inner like.

// 3. Pick broadcast nested loop join as the final solution. It may OOM but we don't have

// other choice. It broadcasts the smaller side for inner and full joins, broadcasts the

// left side for right join, and broadcasts right side for left join.

case logical.Join(left, right, joinType, condition, hint) =>

val desiredBuildSide = if (joinType.isInstanceOf[InnerLike] || joinType == FullOuter) {

getSmallerSide(left, right)

} else {

// For perf reasons, `BroadcastNestedLoopJoinExec` prefers to broadcast left side if

// it's a right join, and broadcast right side if it's a left join.

// TODO: revisit it. If left side is much smaller than the right side, it may be better

// to broadcast the left side even if it's a left join.

if (canBuildBroadcastLeft(joinType)) BuildLeft else BuildRight

}

def createBroadcastNLJoin(buildLeft: Boolean, buildRight: Boolean) = {

val maybeBuildSide = if (buildLeft && buildRight) {

Some(desiredBuildSide)

} else if (buildLeft) {

Some(BuildLeft)

} else if (buildRight) {

Some(BuildRight)

} else {

None

}

maybeBuildSide.map { buildSide =>

Seq(joins.BroadcastNestedLoopJoinExec(

planLater(left), planLater(right), buildSide, joinType, condition))

}

}

def createCartesianProduct() = {

if (joinType.isInstanceOf[InnerLike]) {

Some(Seq(joins.CartesianProductExec(planLater(left), planLater(right), condition)))

} else {

None

}

}

def createJoinWithoutHint() = {

createBroadcastNLJoin(canBroadcastBySize(left, conf), canBroadcastBySize(right, conf))

.orElse(createCartesianProduct())

.getOrElse {

// This join could be very slow or OOM

Seq(joins.BroadcastNestedLoopJoinExec(

planLater(left), planLater(right), desiredBuildSide, joinType, condition))

}

}

createBroadcastNLJoin(hintToBroadcastLeft(hint), hintToBroadcastRight(hint))

.orElse { if (hintToShuffleReplicateNL(hint)) createCartesianProduct() else None }

.getOrElse(createJoinWithoutHint())

// --- Cases where this strategy does not apply ---------------------------------------------

case \_ => Nil

}

}

由于 Spark 的计算引擎优化器不是万能的，有些场景下会选择错误的 Join 策略，所以 Spark 2.4 & Spark 3.0 引入了 Join hint，也就是用户可以自己选择 Join 策略。上面的代码可以看出，用户指定的 Join hint 优先级最高。从代码中可以看出 Spark 3.x 是按照下面下面顺序来选择 Join 策略的：

**先判断是不是等值 Join，那么是按照下面顺序选择 Join 策略：**

* 用户是不是指定了 BROADCAST hint （BROADCAST、BROADCASTJOIN 以及 MAPJOIN 中的一个），如果指定了，那就用 Broadcast Hash Join；
* 用户是不是指定了 SHUFFLE MERGE hint （SHUFFLE\_MERGE、MERGE 以及 MERGEJOIN 中的一个），如果指定了，那就用 Shuffle sort merge join；
* 用户是不是指定了 Shuffle Hash Join hint （SHUFFLE\_HASH），如果指定了，那就用 Shuffle Hash Join；
* 用户是不是指定了 shuffle-and-replicate nested loop join hint （SHUFFLE\_REPLICATE\_NL），如果指定了，那就用 Cartesian product join；
* •如果用户没有指定任何 Join hint，那根据 Join 的适用条件按照 Broadcast Hash Join -> Shuffle Hash Join -> Sort Merge Join ->Cartesian Product Join -> Broadcast Nested Loop Join 顺序选择 Join 策略

**如果是不等值 Join，那么是按照下面顺序选择 Join 策略：**

* 用户是不是指定了 BROADCAST hint （BROADCAST、BROADCASTJOIN 以及 MAPJOIN 中的一个），如果指定了，那就广播对应的表，并选择 Broadcast Nested Loop Join；
* 用户是不是指定了 shuffle-and-replicate nested loop join hint （SHUFFLE\_REPLICATE\_NL），如果指定了，那就用 Cartesian product join；
* 如果用户没有指定任何 Join hint，那根据 Join 的适用条件按照 Broadcast Nested Loop Join ->Cartesian Product Join -> Broadcast Nested Loop Join 顺序选择 Join 策略

### 8.5 Spark 三大误解☆

**8.5.1 Spark是一种内存技术**

大家对Spark最大的误解就是其是一种内存技术（in-memory technology）。其实不是这样的！没有一个Spark开发者正式说明这个，这是对Spark计算过程的误解。

我们从头开始说明。什么样的技术才能称得上是内存技术？在我看来，就是允许你将数据持久化（persist）在RAM中并有效处理的技术。然而Spark并不具备将数据数据存储在RAM的选项，虽然我们都知道可以将数据存储在HDFS, Tachyon, HBase, Cassandra等系统中，但是不管是将数据存储在磁盘还是内存，都没有内置的持久化代码（ native persistence code）。它所能做的事就是缓存（cache）数据，而这个并不是数据持久化（persist）。已经缓存的数据可以很容易地被删除，并且在后期需要时重新计算。

但是即使有这些信息，仍然有些人还是会认为Spark就是一种基于内存的技术，因为Spark是在内存中处理数据的。这当然是对的，因为我们无法使用其他方式来处理数据。操作系统中的API都只能让你把数据从块设备加载到内存，然后计算完的结果再存储到块设备中。我们无法直接在HDD设备上计算；所以现代系统中的所有处理基本上都是在内存中进行的。

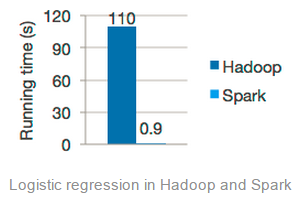
虽然Spark允许我们使用内存缓存以及LRU替换规则，但是你想想现在的RDBMS系统，比如Oracle 和 PostgreSQL，你认为它们是如何处理数据的？它们使用共享内存段（shared memory segment）作为table pages的存储池，所有的数据读取以及写入都是通过这个池的，这个存储池同样支持LRU替换规则；所有现代的数据库同样可以通过LRU策略来满足大多数需求。但是为什么我们并没有把Oracle 和 PostgreSQL称作是基于内存的解决方案呢？你再想想Linux IO，你知道吗？所有的IO操作也是会用到LRU缓存技术的。

你现在还认为Spark在内存中处理所有的操作吗？你可能要失望了。比如Spark的核心：shuffle，其就是将数据写入到磁盘的。如果你再SparkSQL中使用到group by语句，或者你将RDD转换成PairRDD并且在其之上进行一些聚合操作，这时候你强制让Spark根据key的哈希值将数据分发到所有的分区中。shuffle的处理包括两个阶段：map 和 reduce。Map操作仅仅根据key计算其哈希值，并将数据存放到本地文件系统的不同文件中，文件的个数通常是reduce端分区的个数；Reduce端会从 Map端拉取数据，并将这些数据合并到新的分区中。所有如果你的RDD有M个分区，然后你将其转换成N个分区的PairRDD，那么在shuffle阶段将会创建 M\*N 个文件！虽然目前有些优化策略可以减少创建文件的个数，但这仍然无法改变每次进行shuffle操作的时候你需要将数据先写入到磁盘的事实！

所以结论是：Spark并不是基于内存的技术！它其实是一种可以有效地使用内存LRU策略的技术。

**8.5.2 Spark要比Hadoop快 10x-100x**

相信大家在Spark的官网肯定看到了如下所示的图片



这个图片是分别使用 Spark 和 Hadoop 运行逻辑回归（Logistic Regression）机器学习算法的运行时间比较，从上图可以看出Spark的运行速度明显比Hadoop快上百倍！但是实际上是这样的吗？大多数机器学习算法的核心部分是什么？其实就是对同一份数据集进行相同的迭代计算，而这个地方正是Spark的LRU算法所骄傲的地方。当你多次扫描相同的数据集时，你只需要在首次访问时加载它到内存，后面的访问直接从内存中获取即可。这个功能非常的棒！但是很遗憾的是，官方在使用Hadoop运行逻辑回归的时候很大可能没有使用到HDFS的缓存功能，而是采用极端的情况。如果在Hadoop中运行逻辑回归的时候采用到HDFS缓存功能，其表现很可能只会比Spark差3x-4x，而不是上图所展示的一样

根据经验，企业所做出的基准测试报告一般都是不可信的！一般独立的第三方基准测试报告是比较可信的，比如：TPC-H。他们的基准测试报告一般会覆盖绝大部分场景，以便真实地展示结果。

一般来说，Spark比MapReduce运行速度快的原因主要有以下几点：

* task启动时间比较快，Spark是fork出线程；而MR是启动一个新的进程；
* 更快的shuffles，Spark只有在shuffle的时候才会将数据放在磁盘，而MR却不是。
* 更快的工作流：典型的MR工作流是由很多MR作业组成的，他们之间的数据交互需要把数据持久化到磁盘才可以；而Spark支持DAG以及pipelining，在没有遇到shuffle完全可以不把数据缓存到磁盘。
* 缓存：虽然目前HDFS也支持缓存，但是一般来说，Spark的缓存功能更加高效，特别是在SparkSQL中，我们可以将数据以列式的形式储存在内存中。

所有的这些原因才使得Spark相比Hadoop拥有更好的性能表现；在比较短的作业确实能快上100倍，但是在真实的生产环境下，一般只会快 2.5x ~ 3x！

**8.5.3 Spark在数据处理方面引入了全新的技术**

事实上，Spark并没有引入任何革命性的新技术！其擅长的LRU缓存策略和数据的pipelining处理其实在MPP数据库中早就存在！Spark做出重要的一步是使用开源的方式来实现它！并且企业可以免费地使用它。大部分企业势必会选择开源的Spark技术，而不是付费的MPP技术。