# Spark Plan

整体继承体系

见visio类图

主要三部分：LeafNode、UnaryNode、BinaryNode

各自的实现类：



提供四个方法

// TODO: Move to `DistributedPlan`

/\*\* Specifies how data is partitioned across different nodes in the cluster. \*/

**def** outputPartitioning: Partitioning = UnknownPartitioning(0) // TODO: WRONG WIDTH!

/\*\* Specifies any partition requirements on the input data for this operator. \*/

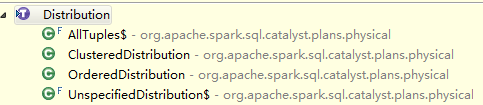
**def** requiredChildDistribution: Seq[Distribution] =

Seq.fill(children.size)(UnspecifiedDistribution)

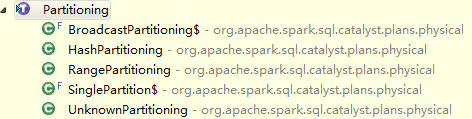
**def** execute(): RDD[Row]

**def** executeCollect(): Array[Row] = execute().collect()

Distribution包括这么几种



Partitioning包括这么几种



# LeafNode

## ExistingRdd

先介绍下Row和GenericRow的概念。

Row是一行output对应的数据，提供getXXX(i: Int)方法

**trait** Row **extends** Seq[Any] **with** Serializable

支持数据类型包括Int, Long, Double, Float, Boolean, Short, Byte, String。支持按序数(ordinal)读取某一个列的值。读取前需要做isNullAt(i: Int)的判断。

对应的有一个MutableRow类，提供setXXX(i: Int, value: Any)方法。可以修改(set)某序数上的值

GenericRow是Row的一种方便实现，存的是一个数组

**class** GenericRow(**protected**[catalyst] **val** values: Array[Any]) **extends** Row

所以对应的取值操作和判断是否为空操作会转化为数组上的定位取值操作。

它也有一个对应的GenericMutableRow类，可以修改(set)值。

ExistingRdd用于把绑定了case class的rdd的数据，转变为RDD[Row]，同时反射提取出case class的属性(output)。转化过程的单例类和伴生对象如下：

**object** ExistingRdd {

**def** convertToCatalyst(a: Any): Any = a **match** {

**case** s: Seq[Any] => s.map(convertToCatalyst)

**case** p: Product => **new** GenericRow(p.productIterator.map(convertToCatalyst).toArray)

**case** other => other

}

// 把RDD[A]映射成为RDD[Row]，map A中每一行数据

**def** productToRowRdd[A <: Product](data: RDD[A]): RDD[Row] = {

// TODO: Reuse the row, don't use map on the product iterator. Maybe code gen?

data.map(r => **new** GenericRow(r.productIterator.map(convertToCatalyst).toArray): Row)

}

**def** fromProductRdd[A <: Product : TypeTag](productRdd: RDD[A]) = {

ExistingRdd(ScalaReflection.attributesFor[A], productToRowRdd(productRdd))

}

}

**case** **class** ExistingRdd(output: Seq[Attribute], rdd: RDD[Row]) **extends** LeafNode {

**def** execute() = rdd

}

# UnaryNode

## Aggregate\*

隐式转换声明，针对本地分区的RDD，扩充了一些操作

/\* Implicit conversions \*/

**import** org.apache.spark.rdd.PartitionLocalRDDFunctions.\_

Groups input data by `groupingExpressions` and computes the `aggregateExpressions` for each group.

**@param** child the input data source.

**case** **class** Aggregate(

partial: Boolean,

groupingExpressions: Seq[Expression],

aggregateExpressions: Seq[NamedExpression],

child: SparkPlan)(@transient sc: SparkContext)

在初始化的时候，partial这个参数用来标志本次Aggregate操作只在本地做，还是要去到符合groupExpression的其他partition上都做。该判断逻辑如下：

**override** **def** requiredChildDistribution =

**if** (partial) { // true, 未知的分布

UnspecifiedDistribution :: Nil

} **else** {

// 如果为空，则分布情况是全部的tuple在一个single partition里

**if** (groupingExpressions == Nil) {

AllTuples :: Nil

// 否则是集群分布的，分布规则来自groupExpressions

} **else** {

ClusteredDistribution(groupingExpressions) :: Nil

}

}

该分布在哪里使用？execute里并没有使用。

最重要的execute()方法：

**def** execute() = attachTree(**this**, "execute") {

// 这里进行了一次隐式转换，生成了PartitionLocalRDDFunctions

**val** grouped = child.execute().mapPartitions { iter =>

**val** buildGrouping = **new** Projection(groupingExpressions)

iter.map(row => (buildGrouping(row), row.copy()))

}.groupByKeyLocally() // 这里生成的结果是RDD[(K, Seq[V])]

**val** result = grouped.map { **case** (group, rows) =>

// 这一步会把aggregateExpressions对应到具体的spark方法都找出来

// 具体做法是遍历aggregateExpressions，各自newInstance

**val** aggImplementations = createAggregateImplementations()

// Pull out all the functions so we can feed each row into them.

**val** aggFunctions = aggImplementations.flatMap(\_ collect { **case** f: AggregateFunction => f })

rows.foreach { row =>

aggFunctions.foreach(\_.update(row))

}

buildRow(aggImplementations.map(\_.apply(group)))

}

// TODO: THIS BREAKS PIPELINING, DOUBLE COMPUTES THE ANSWER, AND USES TOO MUCH MEMORY...

**if** (groupingExpressions.isEmpty && result.count == 0) {

// When there is no output to the Aggregate operator, we still output an empty row.

**val** aggImplementations = createAggregateImplementations()

sc.makeRDD(buildRow(aggImplementations.map(\_.apply(**null**))) :: Nil)

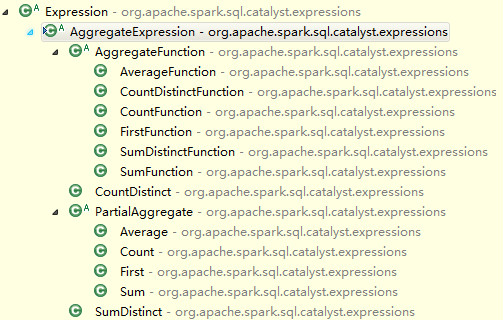
} **else** {

result

}

}

AggregateExpression继承体系如下，这部分代码在Catalyst expressions包的aggregates.scala里：



他的第一类实现AggregateFunction，带一个update(input: Row)操作。子类的update操作是实际对Row执行变化。

## DebugNode

DebugNode是把传进来child SparkPlan调用execute()执行，然后把结果childRdd逐个输出查看

**case** **class** DebugNode(child: SparkPlan) **extends** UnaryNode

## Exchange\*

**case** **class** Exchange(newPartitioning: Partitioning, child: SparkPlan) **extends** UnaryNode

为某个SparkPlan，实施新的分区策略。标黄部分是Spark Core里的类。

execute()方法

**def** execute() = attachTree(**this** , "execute") {

newPartitioning **match** {

**case** HashPartitioning(expressions, numPartitions) =>

// 把expression作用到rdd每个partition的每个row上

**val** rdd = child.execute().mapPartitions { iter =>

**val** hashExpressions = **new** MutableProjection(expressions)

**val** mutablePair = **new** MutablePair[Row, Row]() // 相当于Tuple2

iter.map(r => mutablePair.update(hashExpressions(r), r))

}

**val** part = **new** HashPartitioner(numPartitions)

// 生成ShuffledRDD

**val** shuffled = **new** ShuffledRDD[Row, Row, MutablePair[Row, Row]](rdd, part)

shuffled.setSerializer(**new** SparkSqlSerializer(**new** SparkConf(**false**)))

shuffled.map(\_.\_2) // 输出Tuple2里的第二个值

**case** RangePartitioning(sortingExpressions, numPartitions) =>

// TODO: RangePartitioner should take an Ordering.

**implicit** **val** ordering = **new** RowOrdering(sortingExpressions)

**val** rdd = child.execute().mapPartitions { iter =>

**val** mutablePair = **new** MutablePair[Row, Null](**null**, **null**)

iter.map(row => mutablePair.update(row, **null**))

}

**val** part = **new** RangePartitioner(numPartitions, rdd, ascending = **true**)

**val** shuffled = **new** ShuffledRDD[Row, Null, MutablePair[Row, Null]](rdd, part)

shuffled.setSerializer(**new** SparkSqlSerializer(**new** SparkConf(**false**)))

shuffled.map(\_.\_1)

**case** SinglePartition =>

child.execute().coalesce(1, shuffle = **true**)

**case** \_ => sys.error(s"Exchange not implemented for $newPartitioning")

// TODO: Handle BroadcastPartitioning.

}

}

## Filter

**case** **class** Filter(condition: Expression, child: SparkPlan) **extends** UnaryNode

**def** execute() = child.execute().mapPartitions { iter =>

iter.filter(condition.apply(\_).asInstanceOf[Boolean])

}

## Generate

**case** **class** Generate(

generator: Generator,

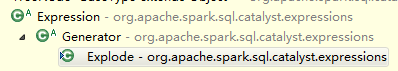
join: Boolean,

outer: Boolean,

child: SparkPlan)

**extends** UnaryNode

首先，Generator是表达式的子类，继承结构如下



Generator的作用是把input的row处理后输出0个或多个rows，makeOutput()的策略由子类实现。

Explode类做法是将输入的input array里的每一个value（可能是ArrayType，可能是MapType），变成一个GenericRow(Array(v))，输出就是一个

回到Generate操作，

join布尔值用于指定最后输出的结果是否要和输入的原tuple显示做join

outer布尔值只有在join为true的时候才生效，且outer为true的时候，每个input的row都至少会被作为一次output

总体上，Generate操作类似FP里的flatMap操作

**def** execute() = {

**if** (join) {

child.execute().mapPartitions { iter =>

**val** nullValues = Seq.fill(generator.output.size)(Literal(**null**))

// Used to produce rows with no matches when outer = true.

**val** outerProjection =

**new** Projection(child.output ++ nullValues, child.output)

**val** joinProjection =

**new** Projection(child.output ++ generator.output, child.output ++ generator.output)

**val** joinedRow = **new** JoinedRow

iter.flatMap {row =>

**val** outputRows = generator(row)

**if** (outer && outputRows.isEmpty) {

outerProjection(row) :: Nil

} **else** {

outputRows.map(or => joinProjection(joinedRow(row, or)))

}

}

}

} **else** {

child.execute().mapPartitions(iter => iter.flatMap(generator))

}

}

## Project

**case** **class** Project(projectList: Seq[NamedExpression], child: SparkPlan) **extends** UnaryNode

project的执行：

**def** execute() = child.execute().mapPartitions { iter =>

@transient **val** reusableProjection = **new** MutableProjection(projectList)

iter.map(reusableProjection)

}

MutableProjection类是Row => Row的继承类，它构造的时候接收一个Seq[Expression]，还允许接收一个inputSchema: Seq[Attribute]。MutableProjection用于根据表达式（和Schema，如果有Schema的话）把Row映射成新的Row，改变内部的column。

## Sample

**case** **class** Sample(fraction: Double, withReplacement: Boolean, seed: Int, child: SparkPlan) **extends** UnaryNode

**def** execute() = child.execute().sample(withReplacement, fraction, seed)

RDD的sample操作：

**def** sample(withReplacement: Boolean, fraction: Double, seed: Int): RDD[T] = {

require(fraction >= 0.0, "Invalid fraction value: " + fraction)

**if** (withReplacement) {

**new** PartitionwiseSampledRDD[T, T](**this**, **new** PoissonSampler[T](fraction), seed)

} **else** {

**new** PartitionwiseSampledRDD[T, T](**this**, **new** BernoulliSampler[T](fraction), seed)

}

}

生成的PartitionwiseSampledRDD会在父RDD的每个partition都选取样本。

PossionSampler和BernoulliSampler是RandomSampler的两种实现。

## Sort

**case** **class** Sort(

sortOrder: Seq[SortOrder],

global: Boolean,

child: SparkPlan)

**extends** UnaryNode

对分布的要求：

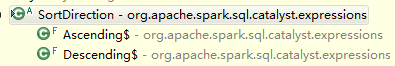
**override** **def** requiredChildDistribution =

**if** (global) OrderedDistribution(sortOrder) :: Nil

**else** UnspecifiedDistribution :: Nil

SortOrder类是UnaryExpression的实现，定义了tuple排序的策略（递增或递减）。该类只是为child expression们声明了排序策略。之所以继承Expression，是为了能影响到子树。

**case** **class** SortOrder(child: Expression, direction: SortDirection) **extends** UnaryExpression



// RowOrdering继承Ordering[Row]

@transient

**lazy** **val** ordering = **new** RowOrdering(sortOrder)

**def** execute() = attachTree(**this**, "sort") {

// TODO: Optimize sorting operation?

child.execute()

.mapPartitions(iterator => iterator.map(\_.copy()).toArray.sorted(ordering).iterator,

preservesPartitioning = **true**)

}

有一次隐式转换过程，.sorted是array自带的一个方法，因为ordering是RowOrdering类，该类继承Ordering[T]，是scala.math.Ordering[T]类。

## StopAfter

**case** **class** StopAfter(limit: Int, child: SparkPlan)(@transient sc: SparkContext) **extends** UnaryNode

StopAfter实质上是一次limit操作

**override** **def** executeCollect() = child.execute().map(\_.copy()).take(limit)

**def** execute() = sc.makeRDD(executeCollect(), 1) // 设置并行度为1

makeRDD实质上调用的是new ParallelCollectionRDD[T]的操作，此处的seq为take()返回的Array[T]，而numSlices为1：

/\*\* Distribute a local Scala collection to form an RDD. \*/

**def** parallelize[T: ClassTag](seq: Seq[T], numSlices: Int = defaultParallelism): RDD[T] = {

**new** ParallelCollectionRDD[T](**this**, seq, numSlices, Map[Int, Seq[String]]())

}

## TopK\*

**case** **class** TopK(limit: Int, sortOrder: Seq[SortOrder], child: SparkPlan)

(@transient sc: SparkContext) **extends** UnaryNode

可以把TopK理解为类似Sort和StopAfter的结合，

@transient

**lazy** **val** ordering = **new** RowOrdering(sortOrder)

**override** **def** executeCollect() = child.execute().map(\_.copy()).takeOrdered(limit)(ordering)

**def** execute() = sc.makeRDD(executeCollect(), 1)

takeOrdered(num)(sorting)实际触发的是RDD的top()()操作

**def** top(num: Int)(**implicit** ord: Ordering[T]): Array[T] = {

mapPartitions { items =>

**val** queue = **new** BoundedPriorityQueue[T](num)

queue ++= items

Iterator.single(queue)

}.reduce { (queue1, queue2) =>

queue1 ++= queue2

queue1

}.toArray.sorted(ord.reverse)

}

BoundedPriorityQueue是Spark util包里的一个数据结构，包装了PriorityQueue，他的优化点在于限制了优先队列的大小，比如在添加元素的时候，如果超出size了，就会进行对堆进行比较和替换。适合TopK的场景。

所以每个partition在排序前，只会产生一个num大小的BPQ(最后只需要选Top num个)，合并之后才做真正的排序，最后选出前num个。

# BinaryNode

## BroadcastNestedLoopJoin\*

**case** **class** BroadcastNestedLoopJoin(

streamed: SparkPlan, broadcast: SparkPlan, joinType: JoinType, condition: Option[Expression])

(@transient sc: SparkContext)

**extends** BinaryNode

操作如下：

**def** execute() = {

// 先将需要广播的SparkPlan执行后进行一次broadcast操作

**val** broadcastedRelation =

sc.broadcast(broadcast.execute().map(\_.copy()).collect().toIndexedSeq)

**val** streamedPlusMatches = streamed.execute().mapPartitions { streamedIter =>

**val** matchedRows = **new** mutable.ArrayBuffer[Row]

**val** includedBroadcastTuples =

**new** mutable.BitSet(broadcastedRelation.value.size)

**val** joinedRow = **new** JoinedRow

streamedIter.foreach { streamedRow =>

**var** i = 0

**var** matched = **false**

**while** (i < broadcastedRelation.value.size) {

// TODO: One bitset per partition instead of per row.

**val** broadcastedRow = broadcastedRelation.value(i)

**if** (boundCondition(joinedRow(streamedRow, broadcastedRow)).asInstanceOf[Boolean]) {

matchedRows += buildRow(streamedRow ++ broadcastedRow)

matched = **true**

includedBroadcastTuples += i

}

i += 1

}

**if** (!matched && (joinType == LeftOuter || joinType == FullOuter)) {

matchedRows += buildRow(streamedRow ++ Array.fill(right.output.size)(**null**))

}

}

Iterator((matchedRows, includedBroadcastTuples))

}

**val** includedBroadcastTuples = streamedPlusMatches.map(\_.\_2)

**val** allIncludedBroadcastTuples =

**if** (includedBroadcastTuples.count == 0) {

**new** scala.collection.mutable.BitSet(broadcastedRelation.value.size)

} **else** {

streamedPlusMatches.map(\_.\_2).reduce(\_ ++ \_)

}

**val** rightOuterMatches: Seq[Row] =

**if** (joinType == RightOuter || joinType == FullOuter) {

broadcastedRelation.value.zipWithIndex.filter {

**case** (row, i) => !allIncludedBroadcastTuples.contains(i)

}.map {

// TODO: Use projection.

**case** (row, \_) => buildRow(Vector.fill(left.output.size)(**null**) ++ row)

}

} **else** {

Vector()

}

// TODO: Breaks lineage.

sc.union(

streamedPlusMatches.flatMap(\_.\_1), sc.makeRDD(rightOuterMatches))

}

## CartesianProduct

**case** **class** CartesianProduct(left: SparkPlan, right: SparkPlan) **extends** BinaryNode

调用的是RDD的笛卡尔积操作，

**def** execute() =

left.execute().map(\_.copy()).cartesian(right.execute().map(\_.copy())).map {

**case** (l: Row, r: Row) => buildRow(l ++ r)

}

## SparkEquiInnerJoin

**case** **class** SparkEquiInnerJoin(

leftKeys: Seq[Expression],

rightKeys: Seq[Expression],

left: SparkPlan,

right: SparkPlan) **extends** BinaryNode

该join操作适用于left和right两部分partition一样大且提供各自keys的情况。

基本上看代码就可以了，没有什么可以说明的，做local join的时候借助的是PartitionLocalRDDFunctions里的方法。

**def** execute() = attachTree(**this**, "execute") {

**val** leftWithKeys = left.execute().mapPartitions { iter =>

**val** generateLeftKeys = **new** Projection(leftKeys, left.output) // 传入了Schema

iter.map(row => (generateLeftKeys(row), row.copy()))

}

**val** rightWithKeys = right.execute().mapPartitions { iter =>

**val** generateRightKeys = **new** Projection(rightKeys, right.output)

iter.map(row => (generateRightKeys(row), row.copy()))

}

// Do the join.

// joinLocally是PartitionLocalRDDFunctions的方法

**val** joined = filterNulls(leftWithKeys).joinLocally(filterNulls(rightWithKeys))

// Drop join keys and merge input tuples.

joined.map { **case** (\_, (leftTuple, rightTuple)) => buildRow(leftTuple ++ rightTuple) }

}

/\*\*

\* Filters any rows where the any of the join keys is null, ensuring three-valued

\* logic for the equi-join conditions.

\*/

**protected** **def** filterNulls(rdd: RDD[(Row, Row)]) =

rdd.filter {

**case** (key: Seq[\_], \_) => !key.exists(\_ == **null**)

}

PartitionLocalRDDFunctions方法如下，该操作并不引入shuffle操作。两个RDD的partition数目需要相等。

**def** joinLocally[W](other: RDD[(K, W)]): RDD[(K, (V, W))] = {

cogroupLocally(other).flatMapValues {

**case** (vs, ws) => **for** (v <- vs.iterator; w <- ws.iterator) **yield** (v, w)

}

}

# Other

## Union

该操作直接继承SparkPlan

**case** **class** Union(children: Seq[SparkPlan])(@transient sc: SparkContext) **extends** SparkPlan

用传入的SparkPlan集合各自的RDD执行结果生成一个UnionRDD

**def** execute() = sc.union(children.map(\_.execute()))

# SparkStrategy: logical to physical

SparkStrategies类继承QueryPlan[SparkPlan]，内部实现的各个Strategy用于将逻辑执行计划的操作对应到SparkPlan的物理执行类上去。

下面展开介绍每个Strategy负责的内容。

## SparkEquiInnerJoin

处理的case

**case** FilteredOperation(predicates, logical.Join(left, right, Inner, condition))

同名字一样，负责SparkEquiInnerJoin，会先做校对和前序工作，满足条件才会new出SparkPlan的SparkEquiInnerJoin子类。

## PartialAggregation

处理的case

**case** logical.Aggregate(groupingExpressions, aggregateExpressions, child)

负责Aggregate

## BroadcastNestedLoopJoin

处理的case

**case** logical.Join(left, right, joinType, condition)

负责BroadcastNestedLoopJoin

## CartesianProduct

处理的case

**case** logical.Join(left, right, \_, None)

logical.Join(left, right, Inner, Some(condition))

分别对应CartesianProduct和Filter(condition, CartesianProduct)

## TopK

处理的case

**case** logical.StopAfter(IntegerLiteral(limit), logical.Sort(order, child))

负责TopK

## BasicOperators

这块就很多了，不例举了