# 概述

本文档补充Spark SQL一些细节实现的部分，并且跟进了Spark SQL最新的PR融入的new feature。

# SchemaRDD

组成：上下文和一个具体的逻辑执行计划

class SchemaRDD(

@transient val sqlContext: SQLContext,

@transient protected[spark] val logicalPlan: LogicalPlan)

extends RDD[Row](sqlContext.sparkContext, Nil) with SchemaRDDLike {

增加了SchemaRDDLike这个trait，抽象出了几个SchemaRDD都会使用的方法（Scala、Java SchemaRDD）。

SchemaRDD方法依然是三部分：

1. RDD继承方法
2. DSL方法
3. 其他和SqlContext相关的操作

## How to create

## How to bind to Schema

# SqlContext

分析SqlContext内目前新增的feature和细节。

## Cache table

/\*\* Caches the specified table in-memory. \*/

def cacheTable(tableName: String): Unit = {

// 得到的是一个logicalPlan

val currentTable = catalog.lookupRelation(None, tableName)

// 物理执行计划生成之后交给InMemoryColumnarTableScan

val asInMemoryRelation =

InMemoryColumnarTableScan(currentTable.output, executePlan(currentTable).executedPlan)

// SparkLogicalPlan接受的Plan必须是已经确定plan好的SparkPlan

catalog.registerTable(None, tableName, SparkLogicalPlan(asInMemoryRelation))

}

从上面那段代码可以看到，cache之前，需要先把本次cache的table的物理执行计划生成出来。上述的currentTable其实是一个logicalPlan，来自catalog的lookupRelation。

最后注册表的时候，涉及到的SparkLogicalPlan类是LogicalPlan的实现类（但是本身其实是一个SparkPlan），它接受的是SparkPlan，并且是已经确定Plan好了的逻辑执行计划，目前接受两类：ExistingRdd和InMemoryColumnarTableScan。

在cache这个过程里，InMemoryColumnarTableScan并没有执行，但是生成了以InMemoryColumnarTableScan为物理执行计划的SparkLogicalPlan，并存成table的plan。

## Uncache table

在这一步，除了删除catalog里的table信息之外，还调用了InMemoryColumnarTableScan的cacheColumnBuffers方法，得到RDD集合，并进行了unpersist()操作。cacheColumnBuffers方法具体见Columner内，主要做了把RDD每个partition里的ROW的每个Field存到了ColumnBuilder内。

## SparkPlanner 列裁剪

SparkPlanner是SparkStrategies的实现类，借助SparkStrategies内部实现的几套Rule，定义了strategies：

val strategies: Seq[Strategy] =

TakeOrdered ::

PartialAggregation ::

HashJoin ::

ParquetOperations ::

BasicOperators ::

CartesianProduct ::

BroadcastNestedLoopJoin :: Nil

此外，定义了一个pruneFilterProject方法：

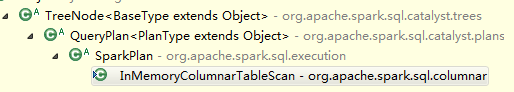
## prepareForExecution细节

# Columnar

## InMemoryColumnarTableScan

### 实现

InMemoryColumnarTableScan类是SparkPlan LeafNode的实现，即是一个物理执行计划。



private[sql] case class InMemoryColumnarTableScan(attributes: Seq[Attribute], child: SparkPlan)

extends LeafNode {

传入的child是一个SparkPlan（确认了的物理执行计划）和一个属性序列。

行转列并cache的过程如下：

lazy val cachedColumnBuffers = {

val output = child.output

// 遍历每个RDD的partiti on

val cached = child.execute().mapPartitions { iterator =>

// 把属性Seq转换成为ColumnBuilder数组

val columnBuilders = output.map { attribute =>

// 都是基本ColumnBuilder，默认ByteBuffer大小

ColumnBuilder(ColumnType(attribute.dataType).typeId, 0, attribute.name)

}.toArray

var row: Row = null

// RDD每个Partition的Rows，每个Row的所有field信息存到ColumnBuilder里

while (iterator.hasNext) {

row = iterator.next()

var i = 0

while (i < row.length) {

columnBuilders(i).appendFrom(row, i)

i += 1

}

}

Iterator.single(columnBuilders.map(\_.build()))

}.cache()

cached.setName(child.toString)

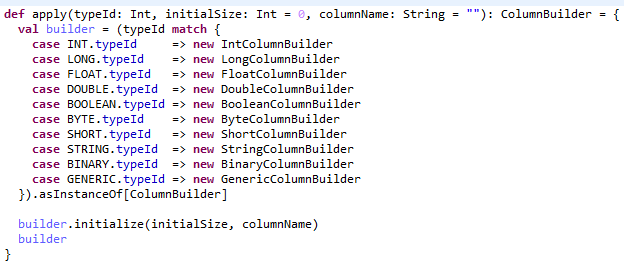
// Force the materialization of the cached RDD.

cached.count()

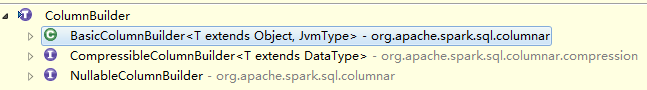
cached

}

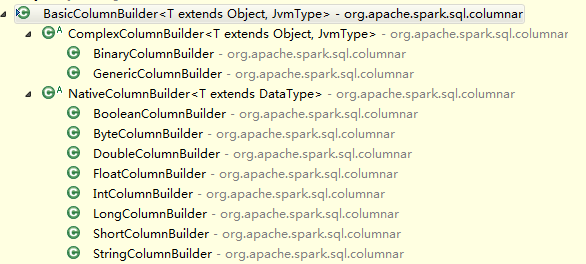
ColumnType类用于表示Column的类型，他的typeId变量用来区分数据类型，生成对应的ColumnBuilder(typeId, initialSize =0, columnName)。ColumnBuilder的生成如下：



他的继承结构如下，主要有三大体系：



这里涉及到的是Basic这个体系，继承结构如下：



BasicColumnBuilder里，initialSize = 0，指使用ByteBuffer的默认大小，即10\*1024\*104。然后在initialize()方法，会初始化ByteBuffer。

接下来，针对RDD每个partition，

var row: Row = null

while (iterator.hasNext) {

row = iterator.next()

var i = 0

while (i < row.length) {

columnBuilders(i).appendFrom(row, i)

i += 1

}

}

进行了appendFrom操作：

override def appendFrom(row: Row, ordinal: Int) {

val field = columnType.getField(row, ordinal)

buffer = ensureFreeSpace(buffer, columnType.actualSize(field))

columnType.append(field, buffer)

}

用于把一个Row的每一个field，都存到一个ColumnBuilder里。在这里指BasicColumnBuilder这个类，维护了一个自己的ByteBuffer，把row里的各个field信息都存在了buffer里。

最后ColumnBuilders里的每个ColumnBuilder进行build()，即BasicColumnBuilder.build()方法，进行了一次ByteBuffer的rewind()方法。

这个方法的结果是一个RDD集合。由于在结束前调用了.count()方法，所以RDD的计算是被执行了的，返回的是新的RDD。

在Spark SQL里，外部调用cachedColumnBuffers方法只有在uncache table的时候，进行了unpersisit()操作。

下面看execute()方法：

override def execute() = {

cachedColumnBuffers.mapPartitions { iterator =>

// 在RDD partition里，iterator.next()返回的是一个ByteBuffer

// 也就是说，cachedColumnBuffers返回的结果RDD，类型是ByteBuffer

val columnBuffers = iterator.next()

assert(!iterator.hasNext)

new Iterator[Row] {

// 访问每一个ByteBuffer里的列信息

val columnAccessors = columnBuffers.map(ColumnAccessor(\_))

val nextRow = new GenericMutableRow(columnAccessors.length)

override def next() = {

var i = 0

// 把column里的信息再转到Row里

while (i < nextRow.length) {

columnAccessors(i).extractTo(nextRow, i)

i += 1

}

nextRow

}

override def hasNext = columnAccessors.head.hasNext

}

}

}

### 使用

在SqlContext里选择cache table的时候，会使用该类。

其实在cache的时候，首先去catalog里寻找这个table的信息和table的执行计划，然后会进行执行（执行到物理执行计划生成），然后把这个table再放回catalog里维护起来，这个时候的执行计划已经是最终要执行的物理执行计划了。但是此时Columner模块相关的转换等操作都是没有触发的。

真正的触发还是在execute()的时候，同其他SparkPlan的execute()方法触发场景是一样的。

## ColumnBuilder 与 ColumnAccessor

一个包装Row的每个field成Column；一个访问column，然后可以转回Row

## 关于压缩

private[sql] abstract class NativeColumnBuilder[T <: NativeType](

override val columnStats: NativeColumnStats[T],

override val columnType: NativeColumnType[T])

extends BasicColumnBuilder[T, T#JvmType](columnStats, columnType)

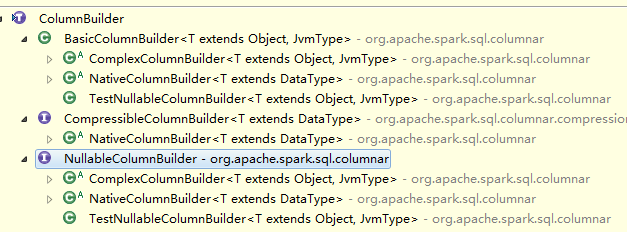
with NullableColumnBuilder

with AllCompressionSchemes

with CompressibleColumnBuilder[T]

private[sql] class BooleanColumnBuilder extends NativeColumnBuilder(new BooleanColumnStats, BOOLEAN)

private[sql] class IntColumnBuilder extends NativeColumnBuilder(new IntColumnStats, INT)



从继承结构看，压缩的builder和Accessor都以trait的方式继承了ColumnBuilder，而子类比如IntColumnBuilder，不但继承了BaseColumnBuilder，同时也具备压缩处理能力。

具体压缩处理可以参考CompressibleColumnBuilder类里的实现。

是否压缩会做一次判断，压缩比在0.8以下才执行压缩。

在build()的时候实施压缩，并且按照以下结构存在bytebuffer内。

*\* .--------------------------- Column type ID (4 bytes)*

*\* | .----------------------- Null count N (4 bytes)*

*\* | | .------------------- Null positions (4 x N bytes, empty if null count is zero)*

*\* | | | .------------- Compression scheme ID (4 bytes)*

*\* | | | | .--------- Compressed non-null elements*

*\* V V V V V*

*\* +---+---+-----+---+---------+*

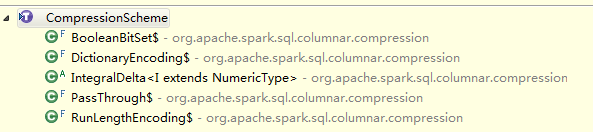
*\* | | | ... | | ... ... |*

*\* +---+---+-----+---+---------+*

*\* \-----------/ \-----------/*

*\* header body*

CompressionScheme子类是不同的压缩实现



都是scala实现的，未借助第三方库。不同的实现，指定了支持的column data类型。在build()的时候，会比较每种压缩，选择压缩率最小的（若仍大于0.8就不压缩了）。

这里的估算能力，在子类实现里，好像是由gatherCompressibilityStats方法实现的。

## Why Columnar

首先，压缩可以节省内存空间。

其次，转列是为了压缩比。