测试点

# spark groupby => 操作1 + 操作2 vs groupby 操作1 + groupby 操作2 其中：key相同

## RDD+第一步没有action操作

2000个key，每个key10000条记录。

结果1：12314463282 11564887474 10843711345 12065016101 11021763915 10620947961 9638186801 9547267624 9651596868 8835231508

结果2：11404768219 10776056292 12967828762 11866653063 10026703278 9940132098 9286838297 10048583277 8972938223 8933135620

单位为纳秒

总结：基于RDD的差异不明显。也就是说groupby一次后再次对同一个key groupby差不多。（注意同一个rdd 进行了两次groupby，虽然是根据一个key）

代码如下：

import org.apache.spark.rdd.RDD

val rd = new java.util.Random(1123L)

val rawData = Seq.range(0, 2000).flatMap {

i =>

Array.tabulate(1000)(\_ => (i, rd.nextInt() % 1000 + 100))

}

val rdd = sc.parallelize(rawData)

rdd.cache()

println("rdd总计:" + rdd.count() + "个")

/\*\*

\* 分组排序取前100

\*/

var time1 = 0L

Array.range(0, 10).foreach {

\_ =>

time1 = System.nanoTime()

println("-" \* 80)

val meanByGroup: RDD[(Int, Int)] = rdd.groupByKey().flatMapValues {

sequence =>

sequence.toArray.sorted.take(100)

}

val res = meanByGroup.groupByKey().mapValues {

sequence =>

sequence.sum

}

println(res.values.sum())

println(s"第1-1: ${System.nanoTime() - time1}")

time1 = System.nanoTime()

println(rdd.groupByKey().mapValues {

sequence =>

sequence.toArray.sorted.take(100).sum

}.values.sum())

println(s"第2: ${System.nanoTime() - time1}")

}

## RDD+第一步有action操作

2000个key，每个key10000条记录。

第1-1：15489103275

第1-2： 1319289806

第2： 8784947353

第1-1: 9634575480

第1-2: 1310514344

第2: 9572860280

第1-1: 9979058303

第1-2: 1274875509

第2: 9749251595

第1-1: 9168790590

第1-2: 1260780011

第2: 9925030812

第1-1: 9750042586

第1-2: 1362086943

第2: 9799277278

第1-1: 9753788228

第1-2: 1345591385

第2: 10694585218

第1-1: 9804751960

第1-2: 1238174612

第2: 10309261320

第1-1: 9340374874

第1-2: 1254801564

第2: 10054244663

第1-1: 10 275 195 235

第1-2: 1 359 573 550

第2: 8 962 664 377

第1-1: 12 007 314 604

第1-2: 1 379 667 835

第2: 10 851 016 946

说明：

第一次groupby action后数据备拉到相应的分区上，会极大的降低第二次group by后操作的性能，因为第二次不再用shuffle了。

代码如下：

import org.apache.spark.rdd.RDD

val rd = new java.util.Random(1123L)

val rawData = Seq.range(0, 2000).flatMap {

i =>

Array.tabulate(1000)(\_ => (i, rd.nextInt() % 1000 + 100))

}

val rdd = sc.parallelize(rawData)

rdd.cache()

println("rdd总计:" + rdd.count() + "个")

/\*\*

\* 分组排序取前100

\*/

var time1 = 0L

Array.range(0, 10).foreach {

\_ =>

time1 = System.nanoTime()

println("-" \* 80)

val meanByGroup: RDD[(Int, Int)] = rdd.groupByKey().flatMapValues {

sequence =>

sequence.toArray.sorted.take(100)

}

println(meanByGroup.values.sum())

println(s"第1-1: ${System.nanoTime() - time1}")

time1 = System.nanoTime()

val res = meanByGroup.groupByKey().mapValues {

sequence =>

sequence.sum

}

println(res.values.sum())

println(s"第1-2: ${System.nanoTime() - time1}")

time1 = System.nanoTime()

val res2 = rdd.groupByKey().mapValues {

sequence =>

sequence.toArray.sorted.take(100).sum

}

println(res2.values.sum())

println(s"第2: ${System.nanoTime() - time1}")

}

# 大模型的序列化

有的模型初始化很慢（如随机森林，需要读取后collect再重新排序）

可以序列化的形式存取

测试从HDFS读取50M左右的序列化模型需要8s左右。

原有的模型初始化整个过程要40min

# 大模型序列化后的分区预测方案

## 将模型读取后构造函数 => 序列化为文件 => 打入jar包并构造一个ClassLoader => driver初始化生成函数 => 分区执行

## 将模型读取后构造函数 => 序列化为文件 => 打入jar包并构造一个ClassLoader => 分区返序列化并调用函数

# key+单列join: union+pivot VS outer join

## 两个数据的key集合相近

## 两个数据的key集合相差较远

# 稀疏特征提取方案

# Spark SQL迭代过程优化

## Spark udf中多列利用struct

# 测试点：

1. glom + 普通array统计 vs mappartition + 普通array统计
2. glom + 一些数组运算 vs mappartition + 一些数组运算
3. shuffle多次获得多个模型 vs 一次shuffle同时维护多个模型