



作者 祝威廉 (/users/59d5607f1400) 2015.12.20 10:15\*

+ 添加关注 (/users/59d5607f1400/toggle\_like)

写了960个字,被508人关注,获得了206个喜欢

# Spark Shuffle Write阶段磁盘文件分析

字数827 阅读1197 评论0 喜欢0

## 前言

上篇写了 Spark Shuffle 内存分析 (<http://www.jianshu.com/p/c83bb237caa8>) 后,有不少人提出了疑问,大家也对如何落文件挺感兴趣的,所以这篇文章会详细介绍,Sort Based Shuffle Write 阶段是如何进行落磁盘的

## 流程分析

入口处:

```
org.apache.spark.scheduler.ShuffleMapTask.runTask
```

runTask对应的代码为:

```
val manager = SparkEnv.get.shuffleManager
writer = manager.getWriter[Any, Any](
    dep.shuffleHandle,
    partitionId,
    context)
writer.write(rdd.iterator(partition, context).asInstanceOf[Iterator[_ <: Product2[Any, Any]])
writer.stop(success = true).get
```

这里manager 拿到的是

```
org.apache.spark.shuffle.sort.SortShuffleWriter
```

我们看他是如何拿到可以写磁盘的那个sorter的。我们分析的线路假设需要做mapSideCombine

```
sorter = if (dep.mapSideCombine) {
  require(dep.aggregator.isDefined, "Map-side combine without Aggregator specified!")
  new ExternalSorter[K, V, C](
    dep.aggregator,
    Some(dep.partitioner),
    dep.keyOrdering, de.serializer)
```

接着将map的输出放到sorter当中：

```
sorter.insertAll(records)
```

其中insertAll 的流程是这样的：

```
while (records.hasNext) {
  addElementsRead() kv = records.next()
  map.changeValue((getPartition(kv._1), kv._1), update)
  maybeSpillCollection(usingMap = true)}
```

里面的map 其实就是PartitionedAppendOnlyMap，这个是全内存的一个结构。当把这个写满了，才会触发spill操作。你可以看到maybeSpillCollection在PartitionedAppendOnlyMap每次更新后都会被调用。

一旦发生呢个spill后，产生的文件名称是：

```
"temp_shuffle_" + id
```

逻辑在这：

```
val (blockId, file) = diskBlockManager.createTempShuffleBlock()

def createTempShuffleBlock(): (TempShuffleBlockId, File) = {
  var blockId = new TempShuffleBlockId(UUID.randomUUID())
  while (getFile(blockId).exists()) {
    blockId = new TempShuffleBlockId(UUID.randomUUID())
  }
  (blockId, getFile(blockId))
}
```

产生的所有 spill文件被记录在一个数组里：

```
private val spills = new ArrayBuffer[SpilledFile]
```

迭代完一个task对应的partition数据后，会做merge操作，把磁盘上的spill文件和内存的，迭代处理，得到一个新的iterator，这个iterator的元素会是这个样子的：

```
(p, mergeWithAggregation(  
    iterators,  
    aggregator.get.mergeCombiners, keyComparator,  
    ordering.isDefined))
```

其中p 是reduce 对应的partitionId, p对应的所有数据都会在其对应的iterator中。

接着会获得最后的输出文件名：

```
val outputFile = shuffleBlockResolver.getDataFile(dep.shuffleId, mapId)
```

文件名格式会是这样的：

```
"shuffle_" + shuffleId + "_" + mapId + "_" + reduceId + ".data"
```

其中reduceId 是一个固定值NOOP\_REDUCE\_ID，默认为0。

然后开始真实写入文件

```
val partitionLengths = sorter.writePartitionedFile(  
    blockId,  
    context,  
    outputFile)
```

写入文件的过程过程是这样的：

```

for ((id, elements) <- this.partitionedIterator) {
    if (elements.hasNext) {

val writer = blockManager.getDiskWriter(blockId,
    outputFile,
    serInstance,
    fileBufferSize,
    context.taskMetrics.shuffleWriteMetrics.get)

    for (elem <- elements) {
        writer.write(elem._1, elem._2)
    }

    writer.commitAndClose()
    val segment = writer.fileSegment()
    lengths(id) = segment.length
    }
}

```

刚刚我们说了，这个 `this.partitionedIterator` 其实内部元素是 `reduce partitionID` -> 实际 `record` 的 `iterator`，所以它其实是顺序写每个分区的记录，写完形成一个 `fileSegment`，并且记录偏移量。这样后续每个的 `reduce` 就可以根据偏移量拿到自己需要的数据。对应的文件名，前面也提到了，是：

```
"shuffle_" + shuffleId + "_" + mapId + "_" + NOOP_REDUCE_ID + ".data"
```

刚刚我们说偏移量，其实是存在内存里的，所以接着要持久化，通过下面的 `writeIndexFile` 来完成：


```

shuffleBlockResolver.writeIndexFile(
    dep.shuffleId,
    mapId,
    partitionLengths)

```

具体的文件名是：

```

简 ☰ ☷ ☶ "shuffle_" + shuffleId + "_" + mapId + "_" + NOOP_REDUCE_ID + ".index" 🌙 

```

至此，一个 `task` 的写入操作完成，对应一个文件。

## 最终结论

所以最后的结论是，一个Executor 最终对应的文件数应该是：

MapNum（注：不包含index文件）

同时持有并且会进行写入的文件数最多为：

CoreNum

🔗 推荐拓展阅读

🚩 举报文章    © 著作权归作者所有

如果觉得我的文章对您有用，请随意打赏。您的支持将鼓励我继续创作！

¥ 打赏支持

♡ 喜欢

🐦 分享到微博    🗨️ 分享到微信  
更多分享 ▼

0条评论（ 按时间正序 · 按时间倒序 · 按喜欢排序 ）

✎ 添加新评论

写下你的评论...


发表    😊    Ctrl+Enter 发表

↑

QR

✎

被以下专题收入，发现更多相似内容：

 **Spark深度学习** (/collection/dff5187d432f)  
Spark深度学习专题旨在通过高质量的文章对Spark相关技术进行研究和总结  
(/collection/dff5187d432f) · 81篇文章 · 258人关注

🔔 添加关注 (/collections/30285/subscribe)

