



作者 祝威廉 (/users/59d5607f1400) 2015.12.19 21:18*

+ 添加关注 (/users/59d5607f1400/toggle_like)

于 2015 年 12 月 19 日, 被 508 人关注, 获得了 206 个喜欢

Spark Sort Based Shuffle内存分析

字数2444 阅读2293 评论5 喜欢3

分布式系统里的Shuffle 阶段往往是非常复杂的，而且分支条件也多，我只能按着我关注的线去描述。肯定会有不少谬误之处，我会根据自己理解的深入，不断更新这篇文章。

前言

借用和董神的一段对话说下背景：

shuffle共有三种，别人讨论的是hash shuffle，这是最原始的实现，曾经有两个版本，第一版是每个map产生r个文件，一共产生mr个文件，由于产生的中间文件太大影响扩展性，社区提出了第二个优化版本，让一个core上map共用文件，减少文件数目，这样共产生corer个文件，好多了，但中间文件数目仍随任务数线性增加，仍难以应对大作业，但hash shuffle已经优化到头了。为了解决hash shuffle性能差的问题，又引入sort shuffle，完全借鉴mapreduce实现，每个map产生一个文件，彻底解决了扩展性问题

目前Sort Based Shuffle 是作为默认Shuffle类型的。Shuffle 是一个很复杂的过程，任何一个环节都足够写一篇文章。所以这里，我尝试换个方式，从实用的角度出发，让读者有两方面的收获：

1. 剖析哪些环节，哪些代码可能会让内存产生问题
2. 控制相关内存的参数

有时候，我们宁可程序慢点，也不要OOM，至少要先跑步起来，希望这篇文章能够让你达成这个目标。

同时我们会提及一些类名，这些类方便你自己想更深入了解时，可以方便的找到他们，自己去探个究竟。



Shuffle 概览

Spark 的Shuffle 分为 Write,Read 两阶段。我们预先建立三个概念：

- Write 对应的是ShuffleMapTask,具体的写操作ExternalSorter来负责
- Read 阶段由ShuffleRDD里的HashShuffleReader来完成。如果拉来的数据如果过大，需要落地，则也由ExternalSorter来完成的
- 所有Write 写完后，才会执行Read。 他们被分成了两个不同的Stage阶段。

也就是说，Shuffle Write ,Shuffle Read 两阶段都可能需要落磁盘，并且通过Disk Merge 来完成最后的Sort归并排序。

Shuffle Write 内存消耗分析

Shuffle Write 的入口链路为：

```
org.apache.spark.scheduler.ShuffleMapTask
---> org.apache.spark.shuffle.sort.SortShuffleWriter
    ---> org.apache.spark.util.collection.ExternalSorter
```

会产生内存瓶颈的其实就是 `org.apache.spark.util.collection.ExternalSorter`。我们看看这个复杂的ExternalSorter都有哪些地方在占用内存：

第一个地：

```
private var map = new PartitionedAppendOnlyMap[K, C]
```

我们知道，数据都是先写内存，内存不够了，才写磁盘。这里的map就是那个放数据的内存了。

这个 `PartitionedAppendOnlyMap` 内部维持了一个数组，是这样的：

```
private var data = new Array[AnyRef](2 * capacity)
```

也就是他消耗的并不是Storage的内存，所谓Storage内存，指的是由blockManager管理起来的内存。

PartitionedAppendOnlyMap 放不下，要落地，那么不能硬生生的写磁盘，所以需要个buffer,然后把buffer再一次性写入磁盘文件。这个buffer是由参数

```
spark.shuffle.file.buffer=32k
```

控制的。数据获取的过程中，序列化反序列化，也是需要空间的，所以Spark 对数量做了限制，通过如下参数控制：

```
spark.shuffle.spill.batchSize=10000
```

假设一个Executor的可使用的Core为 C个，那么对应需要的内存消耗为：

$$C * 32k + C * 10000 \text{个Record} + C * \text{PartitionedAppendOnlyMap}$$

这么看来，写文件的buffer不是问题，而序列化的batchSize也不是问题，几万或者十几万个Record 而已。那 $C * \text{PartitionedAppendOnlyMap}$ 到底会有多大呢？我先给个结论：

$$C * \text{PartitionedAppendOnlyMap} < \text{ExecutorHeapMemory} * 0.2 * 0.8$$

怎么得到上面的结论呢？核心点就是要判定 PartitionedAppendOnlyMap 需要占用多少内存，而它到底能占用内存，则由触发写磁盘动作决定，因为一旦写磁盘，PartitionedAppendOnlyMap所占有的内存就会被释放。下面是判断是否写磁盘的逻辑代码：

```
estimatedSize = map.estimateSize()
if (maybeSpill(map, estimatedSize)) {
    map = new PartitionedAppendOnlyMap[K, C]
}
```

每放一条记录，就会做一次内存的检查，看 PartitionedAppendOnlyMap 到底占用了多少内存。如果真是这样，假设检查一次内存1ms, 1kw 就不得了的时间了。所以肯定是不行的，所以 estimateSize 其实是使用采样算法来做的。

第二个，我们也不希望 maybeSpill 太耗时,所以 maybeSpill 方法里就搞了很多东西，减少耗时。我们看看都设置了哪些防线

首先会判定要不要执行内部逻辑：

```
elementsRead % 32 == 0 && currentMemory >= myMemoryThreshold
```

每隔 32 次会进行一次检查，并且要当前 `PartitionedAppendOnlyMap` `currentMemory > myMemoryThreshold` 才会进一步判定是不是要spill.

其中 `myMemoryThreshold`可通过如下配置获得初始值

```
spark.shuffle.spill.initialMemoryThreshold = 5 * 1024 * 1024
```

接着会向 `shuffleMemoryManager` 要 $2 * \text{currentMemory} - \text{myMemoryThreshold}$ 的内存，`shuffleMemoryManager` 是被Executor 所有正在运行的Task(Core) 共享的，能够分配出去的内存是：

```
ExecutorHeapMemory * 0.2 * 0.8
```

上面的数字可通过下面两个配置来更改：

```
spark.shuffle.memoryFraction=0.2  
spark.shuffle.safetyFraction=0.8
```

如果无法获取到足够的内存，就会触发真的spill操作了。

看到这里，上面的结论就显而易见了。

然而，这里我们忽略了一个很大的问题，就是

```
estimatedSize = map.estimateSize()
```

为什么说它是大问题，前面我们说了，`estimateSize` 是近似估计，所以有可能估的不准，也就是实际内存会远远超过预期。

具体的大家可以看看 `org.apache.spark.util.collection.SizeTracker`

我这里给出一个结论：

如果你内存开的比较大，其实反倒风险更高，因为estimateSize 并不是每次都去真实的算缓存。它是通过采样来完成的，而采样的周期不是固定的，而是指数增长的，比如第一次采样完后，PartitionedAppendOnlyMap 要经过1.1次的update/insert操作之后才进行第二次采样，然后经过 1.1*1.1 次之后进行第三次采样，以此递推，假设你内存开的大，那PartitionedAppendOnlyMap可能要经过几十万次更新之后之后才会进行一次采样，然后才能计算出新的大小，这个时候几十万次更新带来的新的内存压力，可能已经让你的GC不堪重负了。

当然，这是一种折中，因为确实不能频繁采样。

如果你不想出现这种问题，要么自己替换实现这个类，要么将

```
spark.shuffle.safetyFraction=0.8
```

设置的更小一些。

Shuffle Read 内存消耗分析

Shuffle Read 的入口链路为：

```
org.apache.spark.rdd.ShuffledRDD
---> org.apache.spark.shuffle.sort.HashShuffleReader
    ---> org.apache.spark.util.collection.ExternalAppendOnlyMap
    ---> org.apache.spark.util.collection.ExternalSorter
```

Shuffle Read 会更复杂些，尤其是从各个节点拉取数据。但这块不是不是我们的重点。按流程，主要有：

1. 获取待拉取数据的迭代器
2. 使用AppendOnlyMap/ExternalAppendOnlyMap 做combine
3. 如果需要对key排序，则使用ExternalSorter

其中1后续会单独列出文章。3我们在write阶段已经讨论过。所以这里重点是第二个步骤，combine阶段。

如果你开启了

```
spark.shuffle.spill=true
```

则使用ExternalAppendOnlyMap，否则使用AppendOnlyMap。两者的区别是，前者如果内存不够，则落磁盘，会发生spill操作，后者如果内存不够，直接OOM了。

这里我们会重点分析ExternalAppendOnlyMap。

ExternalAppendOnlyMap 作为内存缓冲数据的对象如下：

```
private var currentMap = new SizeTrackingAppendOnlyMap[K, C]
```

如果currentMap 对象向申请不到内存，就会触发spill动作。判定内存是否充足的逻辑和Shuffle Write 完全一致。

Combine做完之后，ExternalAppendOnlyMap 会返回一个Iterator，叫做 ExternalIterator，这个Iterator背后的数据源是所有spill文件以及当前currentMap里的数据。

我们进去 ExternalIterator 看看，唯一的一个占用内存的对象是这个优先队列：

```
private val mergeHeap = new mutable.PriorityQueue[StreamBuffer]
```

mergeHeap 里元素数量等于所有spill文件个数加一。StreamBuffer 的结构：

```
private class StreamBuffer(  
    val iterator: BufferedIterator[(K, C)],  
    val pairs: ArrayBuffer[(K, C)])
```

其中iterator 只是一个对象引用，pairs 应该保存的是iterator里的第一个元素(如果hash有冲突的话，则为多个)

所以mergeHeap 应该不占用什么内存。到这里我们看看应该占用多少内存。依然假设 CoreNum 为 C,则

$$C * 32k + C * \text{mergeHeap} + C * \text{SizeTrackingAppendOnlyMap}$$

所以这一段占用内存较大的依然是 SizeTrackingAppendOnlyMap，一样的，他的值也符合如下公式

$$C * \text{SizeTrackingAppendOnlyMap} < \text{ExecutorHeapMemeory} * 0.2 * 0.8$$

ExternalAppendOnlyMap 的目的是做Combine,然后如果你还设置了Order,那么接着会启用ExternalSorter 来完成排序。

经过上文对Shuffle Write的使用，相比大家也对ExternalSorter有一定的了解了，此时应该占用内存的地方最大不超过下面的这个值：

$$C * \text{SizeTrackingAppendOnlyMap} + C * \text{PartitionedAppendOnlyMap}$$

不过即使如此，因为他们共享一个shuffleMemoryManager，则理论上只有这么大：

$$C * \text{SizeTrackingAppendOnlyMap} < \text{ExecutorHeapMemory} * 0.2 * 0.8$$

分析到这里，我们可以做个总结：

1. Shuffle Read阶段如果内存不足，有两个阶段会落磁盘，分别是Combine 和 Sort 阶段。对应的都会spill小文件，并且产生读。
2. Shuffle Read 阶段如果开启了spill功能，则基本能保证内存控制在 $\text{ExecutorHeapMemory} * 0.2 * 0.8$ 之内。

后话

如果大家对Sort Shuffle 落磁盘文件这块感兴趣，还可以看看这篇文章 Spark Shuffle Write阶段磁盘文件分析 (<http://www.jianshu.com/p/2d837bf2dab6>)

➤ 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 微信 分享到微信
更多分享 ▼

 jacksu_ (/users/92a1227beb27)
2楼 2015.12.20 13:28 (/p/c83bb237caa8/comments/1066490#comment-1066490)

这边文章如果修改为Sort Shuffle,貌似部分解释有问题吧，有些解释是Hash Shuffle

♡ 喜欢(0)

回复

祝威廉 (/users/59d5607f1400) : @jacksu_ (/users/92a1227beb27) 那个部分是 hash shuffle的部分 ?
2015.12.20 13:55 (/p/c83bb237caa8/comments/1066617#comment-1066617)


回复


jacksu_ (/users/92a1227beb27) : @祝威廉 (/users/59d5607f1400) $C * 32k + C * 10000$ 个Record + $C * PartitionedAppendOnlyMap$, 个人感觉C不是core数而是spark.executor.cores/ spark.task.cpus数即mapTask数。
2016.01.10 13:26 (/p/c83bb237caa8/comments/1199912#comment-1199912)

回复

祝威廉 (/users/59d5607f1400) : @jacksu_ (/users/92a1227beb27) 我说的是某一瞬间，最大的占用量。mapTask可以非常大，但是能够同时并行运行的决定于core数
2016.01.22 10:14 (/p/c83bb237caa8/comments/1292666#comment-1292666)

回复

 添加新回复

 GaryHuang (/users/800293d36cb7)
3楼 2016.01.21 22:27 (/p/c83bb237caa8/comments/1290572#comment-1290572)

挺好~ 期待shuffle read的内存分析 😊

♡ 喜欢(0)

回复

写下你的评论...

发表



Ctrl+Enter 发表



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相关技术进行研究学习

[+ 添加关注 \(/collections/30285/subscribe\)](/collections/30285/subscribe)

(/collection/dff5187d432f)

81篇文章 (/collection/dff5187d432f) · 257人关注

