Spark企业级大数据项目实战 第9课

DATAGURU专业数据分析社区

法律声明

【声明】本视频和幻灯片为炼数成金网络课程的教学资料，所有资料只能在课程内使用，不得在课程以外范围散播，违者将可能被追究法律和经济责任。

课程详情访问炼数成金培训网站

http://edu.dataguru.cn

炼数成金逆向收费式网络课程

- Dataguru（炼数成金）是专业数据分析网站，提供教育，媒体，内容，社区，出版，数据分析业务等服务。我们的课程采用新兴的互联网教育形式，独创地发展了逆向收费式网络培训课程模式。既继承传统教育重学习氛围，重竞争压力的特点，同时又发挥互联网的威力打破时空限制，把天南地北志同道合的朋友组织在一起交流学习，使到原先孤立的学习个体组合成有组织的探索力量。并且把原先动辄成千上万的学习成本，直线下降至百元范围，造福大众。我们的目标是：低成本传播高价值知识，构架中国第一的网上知识流转阵地。
- 关于逆向收费式网络的详情，请看我们的培训网站 http://edu.dataguru.cn

本课内容

- coalesce与repartition
- 解决小文件问题
- 通过源码理解Spark Shuffle的优化
- Spark on YARN任务提交
- Spark on YARN占用资源分析与调优

1 coalesce与repartition

- **repartition(numPartitions: Int)**
  - 返回numPartitions分区个数的新RDD(或DataFrame)。
  - 可以增加或减少此RDD中的并行性级别，内部使用shuffle来重新分配数据。
  - 如果要减少partition数量，可考虑使用`coalesce`，这可以避免执行shuffle。

- **coalesce(numPartitions: Int, shuffle: Boolean = false)**
  - 返回一个新的RDD，该RDD被缩减为`numPartitions`分区。
  - 这导致窄依赖，例如， 如果从1000个分区转到100个分区，则不会有shuffle，而是100个新分区中的每一个都将声明当前分区的10个分区。
  - 如果您正在进行剧烈的合并，例如将numPartitions从1000减少为1，这将会导致计算发生在非常少的节点上（例如numPartitions = 1的情况下为一个节点）。

    为了避免这种情况，可以传递shuffle = true，或者直接使用repartition。 这将添加一个shuffle步骤，意味着当前的上游分区将并行执行（无论当前的分区是什么）。
  - 注意：随着shuffle = true，实际上可以合并到更大数量的分区。 如果你有少量的分区，比如100，那么这很有用，可能有几个分区异常大。 调用coalesce（1000，shuffle = true）将导致使用hash partitioner分发数据到1000个分区。

1 coalesce与repartition

```
val list = List(1, 2, 3)
val rdd = sc.parallelize(list)

println("rdd默认分区数: " + rdd.getNumPartitions)

println("rdd.coalesce(4): " + rdd.coalesce(4).getNumPartitions)

println("rdd.repartition(4): " + rdd.repartition(4).getNumPartitions)

println("rdd.coalesce(1): " + rdd.coalesce(1).getNumPartitions)

println("rdd.repartition(1): " + rdd.repartition(1).getNumPartitions)
```

```
scala> val list = List(1, 2, 3)
list: List[Int] = List(1, 2, 3)

scala> val rdd = sc.parallelize(list)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[11] at parallelize at <console>:26

scala>

scala> println("rdd默认分区数: " + rdd.getNumPartitions)
rdd默认分区数: 2

scala> println("rdd.coalesce(4): " + rdd.coalesce(4).getNumPartitions)
rdd.coalesce(4): 2

scala> println("rdd.repartition(4): " + rdd.repartition(4).getNumPartitions)
rdd.repartition(4): 4

scala> println("rdd.coalesce(1): " + rdd.coalesce(1).getNumPartitions)
rdd.coalesce(1): 1

scala> println("rdd.repartition(1): " + rdd.repartition(1).getNumPartitions)
rdd.repartition(1): 1
```

1. repartition用于增大分区
2. partition负载均衡。
   比如1000个分区， 有个分区数据量很小， 有的很大。， repartition
3. 分区数剧烈的合并，使用repartition增加并行度
4. 如果要增大分区， 只能使用repartition。 coalesce只能减少分区， 不能增大分区

2 解决小文件问题

解决小文件问题

- 数据采集阶段： 配置合理的flume参数等
- 数据清洗： 使用coalesce或repatition设置合理的分区数
- 使用hbase保存数据
- 合并小文件程序

2 解决小文件问题-使用coalesce或repatition

解决小文件问题

```
val df = sqlContext.createDataFrame(rowRdd, struct)
val newDF = df.coalesce(1)
```

将newDF导入到hive表或者使用DataFrame的数据源方式写数据。

2 解决小文件问题-合并小文件

1. 将小文件目录(srcDataPath)下的文件移动到临时目录/mergePath/${mergeTime}/src

2. 计算临时目录(/mergePath/${mergeTime}/src)的大小。 根据大小确定分区的数。1024M /128M = 8

3. 使用coalesce或者repartition， 传入分区数。 将临时目录数据写入临时的数据目录(/mergePath/${mergeTime}/data)

4. 将临时数据目录文件move到文件目录(srcDataPath)

5. 删除临时目录(/mergePath/${mergeTime}/src)

${mergeTime} 是变量， 用于标识每次合并的唯一标识

见代码案例实现

2 shuffle阶段产生小文件的原因

spark.sql.shuffle.partitions

调整Shuffle的并行度， 即task的数量。

shuffle的每个partition对应一个task， task数越多， 效率越高。

Spark默认在执行聚合（即shuffle）时，默认有200个分区。 这是由conf变量“spark.sql.shuffle.partitions”定义的。 这就是使用DataFrame或者spark整合Hive，产生shuffle后会产生大量小文件的原因。

3 通过源码理解Spark Shuffle的优化

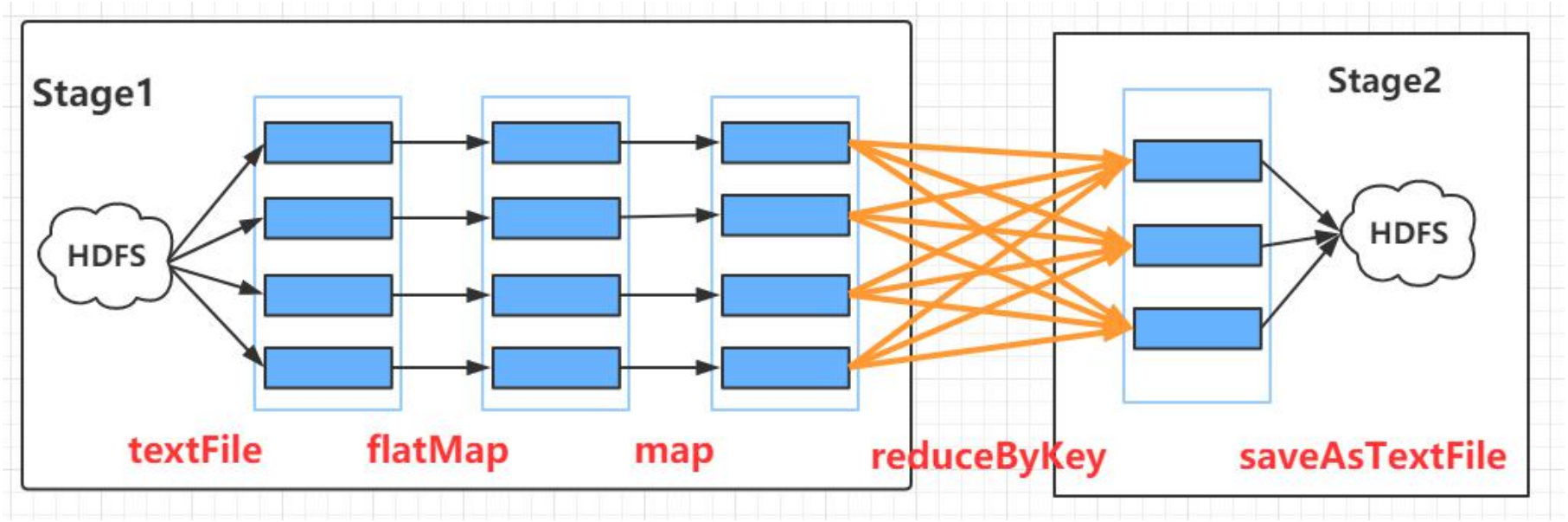
报错信息：

WARN scheduler.TaskSetManager: Lost task 19.0 in stage 6.0 (TID 120, 10.111.32.47):
java.lang.IllegalArgumentException: Size exceeds Integer.MAX_VALUE
at sun.nio.ch.FileChannelImpl.map(FileChannelImpl.java:828)
at org.apache.spark.storage.DiskStore.getBytes(DiskStore.scala:123)
at org.apache.spark.storage.DiskStore.getBytes(DiskStore.scala:132)
at org.apache.spark.storage.BlockManager.doGetLocal(BlockManager.scala:517)
at org.apache.spark.storage.BlockManager.getLocal(BlockManager.scala:432)
at org.apache.spark.storage.BlockManager.get(BlockManager.scala:618)
at org.apache.spark.CacheManager.putInBlockManager(CacheManager.scala:146)
at org.apache.spark.CacheManager.getOrCompute(CacheManager.scala:70)

Spark Shuffle Block的大小不能超过2Ｇ！！

什么是Shuffle Block

```
val result = sc.textFile("/data/word.txt").flatMap(line=>line.split("\\s+")
                .map(word=>(word,1)).reduceByKey(_+_)
result.saveAsTextFile("/data/result/")
```



上图的每个橙色箭头代表一个Shuffle Block

ByteBuffer的2G大小限制

Spark中Block的底层抽象是ByteBuffer， 不幸的是ByteBuffer有大小限制Integer.MAX_VALUE (~2GB).。
适用于所使用的管理block和shuffle block。（内存映射块也被键值为2g）

org.apache.spark.storage.DiskStore

```
if (length < minMemoryMapBytes) {
    val buf = ByteBuffer.allocate(length.toInt)
    channel.position(offset)
    while (buf.remaining() != 0) {
      if (channel.read(buf) == -1) {
        throw new IOException("Reached EOF before filling buffer\n" +
          s"offset=$offset\nfile=${file.getAbsolutePath}\nbuf.remaining=${buf.remaining}")
      }
    }
    buf.flip()
    Some(buf)
  } else {
    Some(channel.map(MapMode.READ_ONLY, offset, length))
  }
```

Spark SQL

- Spark SQ在执行聚合（即shuffle）时，默认有200个分区。
  通过参数spark.sql.shuffle.partitions控制
- 分区数越小， Shuffle Block的大小越大
- 非常大的数据量， 默认的200分区数可能不够用
- 数据倾斜， 导致少数分区的Block大小过大

解决方案

- 在Spark SQL中， 增加分区数， 从而减少Spark SQL在shuffle时的Block 大小
  在Spark SQL中增加spark.sql.shuffle.partitions值

- 避免数据倾斜

- 在Spark RDD， 设置repartiton、coalesce
  rdd.repartiton() 或 rdd.coalesce()

如何确定分区数

经验法则：每个分区大小为128M左右

3 通过源码理解Spark Shuffle的优化

分区数>2000 VS. 分区数<=2000

在shuffle时， 当分区数大于2000和小于2000两种场景， Spark使用不同的数据结构保存数据。

org.apache.spark.scheduler.MapStatus

```
def apply(loc: BlockManagerId, uncompressedSizes: Array[Long]): MapStatus = {
  if (uncompressedSizes.length > 2000) {
    HighlyCompressedMapStatus(loc, uncompressedSizes)
  } else {
    new CompressedMapStatus(loc, uncompressedSizes)
  }
}
```

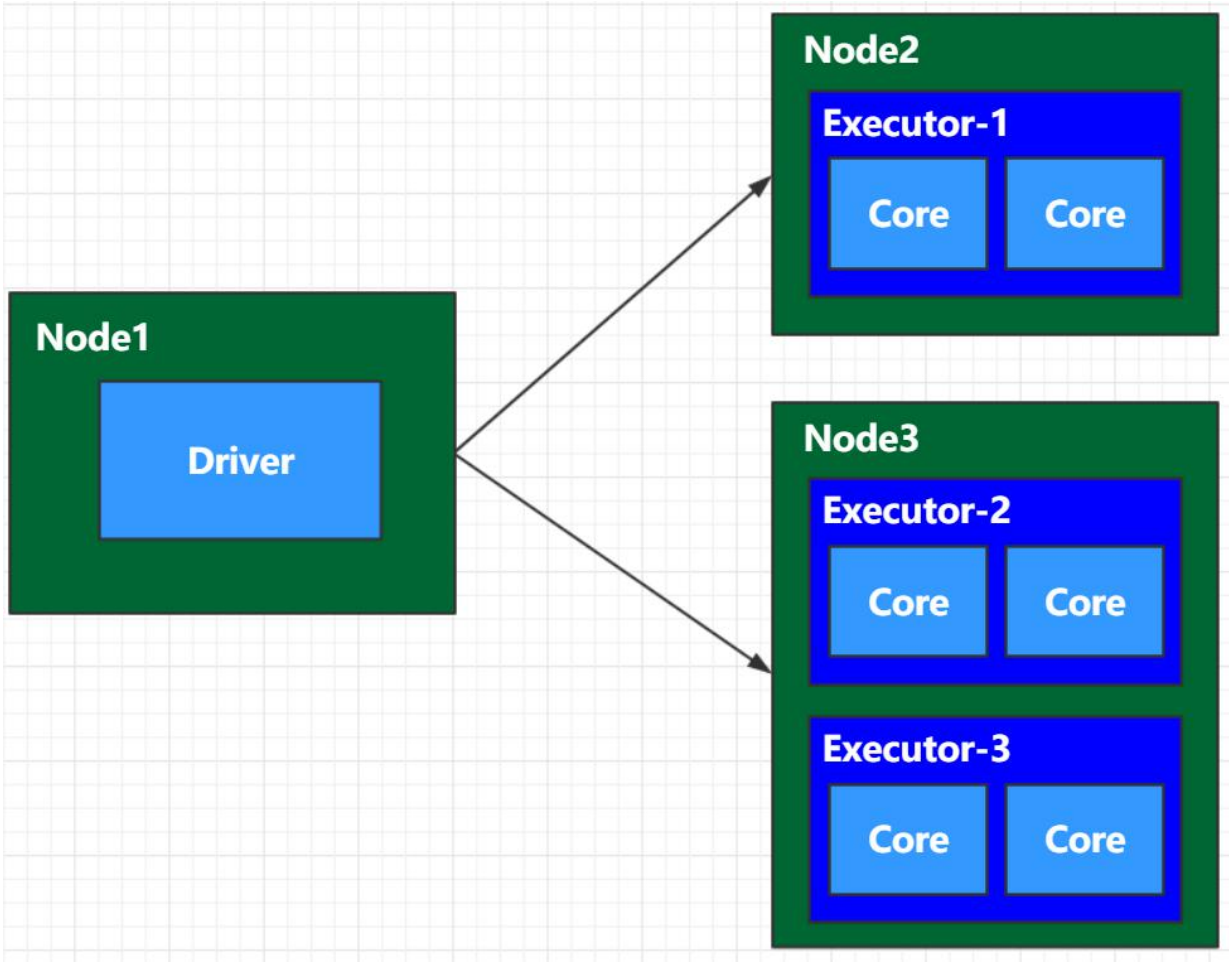
分区数>2000 VS. 分区数<=2000

建议：当Spark 应用的分区数小于2000，但是很接近2000，将分区数调整到比2000稍微大一点

4 Spark on YARN任务提交

```
spark-shell  \
--master yarn-client  \
--num-executors 3 \
--driver-memory 10g \
--executor-memory 3g \
--executor-cores 2 \
--conf spark.yarn.executor.memoryOverhead=1024m
```

上面的spark任务， 申请了3个executor， 每个executor有2个core 。 每个executor内存大小为3g。 executor的堆外内存大小为1024M。
Driver内存大小为10g。

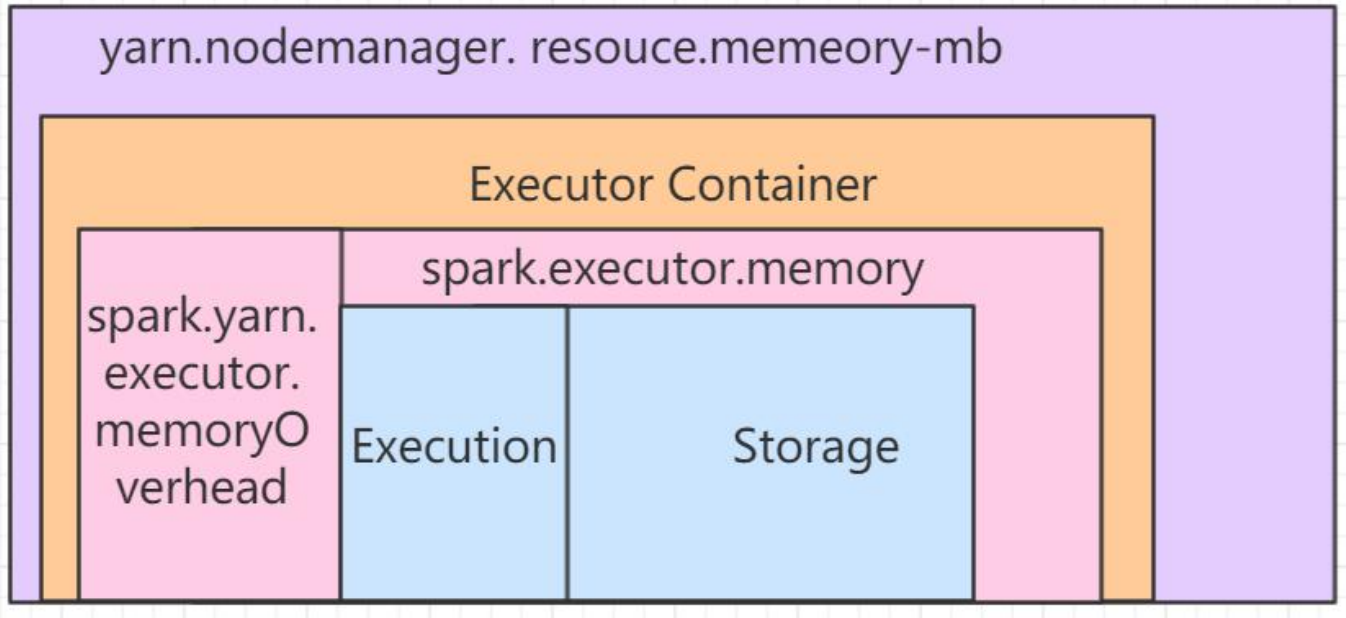


4 Spark on YARN任务提交

参数说明：

--num-executors： 启动的executor数量，即执行spakr计算任务启动的java jvm的数量， 默认是2个。
--executor-memory：executor内存大小，默认1G。
--executor-cores： 每个executor使用的内核数，默认为1
--driver-memory： Driver程序使用内存大小
--jars： Driver依赖的第三方jar包
--name: Application名称
--class: 主类名称，含包名
--queue: 提交应用程序给哪个YARN的队列，默认是default队列
--conf PROP=VALUE: 任意的Spark属性

5 Spark on YARN占用资源分析 - Spark 内存模型



Spark的Excutor的Container内存有两大部分组成：堆外内存和Excutor内存

- 堆外内存(spark.yarn.executor.memoryOverhead)
  主要用于JVM自身的开销。 默认： MAX(executorMemory * 0.10, 384m)

- Excutor内存(spark.executor.memory)
  - Execution: shuffle、排序、聚合等用于计算的内存
  - Storage： 用于集群中缓冲和传播内部数据的内存（cache、广播变量）

5 Spark on YARN占用资源分析 - Spark 内存模型

两个重要参数：

- spark.memory.fraction

用于设置Execution和Storage内存在内存（这个内存是JVM的堆内存 - 300M， 这300M是预留内存）中的占比，默认是60%。即Execution和Storage内存大小之和占堆内存比例。
剩下的40%用于用户的数据结构、Spark的元数据和预留防止OOM的内存。

- spark.memory.storageFraction

表示Storage内存在Execution和Storage内存之和的占比。 设置这个参数可避免缓冲的数据块被清理出内存。

5 Spark on YARN占用资源分析 - Spark 内存模型

堆外内存： 除了前面介绍的Executor的堆外内存， Driver、Application Master进程也有堆外内存。

- Driver的堆外内存设置
  spark.driver.memoryOverhead
  默认： MAX(Driver memory * 0.10, 384m)

- Application Master的堆外内存设置
  spark.yarn.am.memoryOverhead
  默认： MAX(AM memory * 0.10, 384m)

Application Master的内存也是--conf参数设置。

```
spark2-shell --master yarn \
--master yarn-client  \
--num-executors 4 \
--driver-memory 10g \
--executor-memory 3g \
--conf spark.yarn.am.memory=1000m \
--conf  spark.yarn.am.memoryOverhead=1000m \
--conf  spark.driver.memoryOverhead=1g
```

5. Spark on YARN占用资源分析 - 案例

Spark应用程序的Executor的具有相同的固定CPU core数和固定的堆内存大小。

从命令行调用spark-submit或者spark-shell 时， 可以指定资源的属性。

使用 --executor-cores或者spark.executor.cores指定core是数量。

堆内存使用--executor-memory或者spark.executor.memory属性指定。

core属性控制Executor可以运行的并发Task数。 --executor-cores 5 意味着每个Executor可以同时运行五个任务。

内存属性会影响Spark可以缓存的数据量以及用于分组、聚合、关联等shuffle的最大大小。

yarn.nodemanager.resource.memory-mb： 控制每个节点上容器使用的最大内存总和。

yarn.nodemanager.resource.cpu-vcores： 控制每个节点上容器使用的最大内核总数。

5. Spark on YARN占用资源分析 - 案例

申请5个executor cores将从YARN申请5个虚拟的cores， 同Yarn申请内存稍微复杂一点， 主要由以下两个方面的原因：

1. --executor-memory / spark.executor.memory 控制executor的堆内存， 但是JVM也需要一些堆外内存， 即用于inerned字符串和直接字符缓冲区。
   spark.yarn.executor.memoryOverhead属性会被添加到executor的内存中， 以确定每个executor同yarn申请的完整的内存。

2. YARN可能会将申请的内存稍微增加一点。 YARN的yarn.scheduler.minimum-allocation-mb和yarn.scheduler.increment-allocation-mb属性分别控制最小和增量请求值。

5. Spark on YARN占用资源分析 - 案例

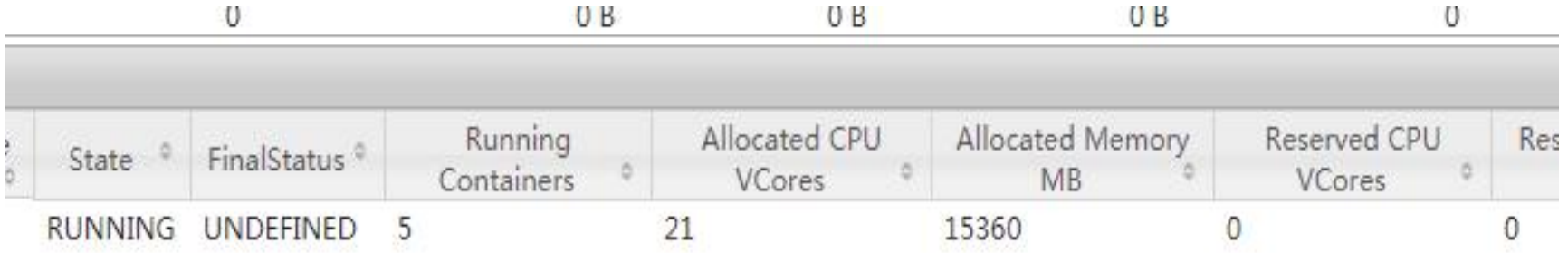
前置条件：

- 调度器：Fair Scheduler
- 规整化因子：
  - yarn.scheduler.minimum-allocation-mb：最小可申请内存量，默认是1024m
  - yarn.scheduler.minimum-allocation-vcores：最小可申请CPU数，默认是1
  - yarn.scheduler.maximum-allocation-mb：最大可申请内存量，默认是36g
  - yarn.scheduler.maximum-allocation-vcores：最大可申请CPU数，默认是4
  - yarn.scheduler.increment-allocation-mb，用于内存最小的增量, 默认512m， 不足512取512
  - yarn.scheduler.increment-allocation-vcores， 用于CPU最小的增量，默认1

5 Spark on YARN占用资源分析-场景1

场景1：任务提交参数

```
spark-shell --master yarn \
--master yarn-client  \
--num-executors 4 \
--driver-memory 10g \
--executor-memory 3g \
--executor-cores 5
```



0		0 B	0 B	0 B		0
State	FinalStatus	Running Containers	Allocated CPU VCores	Allocated Memory MB	Reserved CPU VCores	Res
RUNNING	UNDEFINED	5	21	15360	0	0

从Yarn的管理页面， 分配了5个Container， 21个Cpu Core， 15360M内存

5 Spark on YARN占用资源分析-场景1

- 占用的CPU Core数
  Excutor的Core数 = 4 * 5个core = 20个
  AM的Core数默认为1
  占用的CPU Core数： Excutor的Core数 + AM的Core数 = 4 * 5 + 1 = 21个。

- 占用的Container数
  4个Excutor 进程 + 1个Applicatoin Master进程 = 5个Container

- 占用的资源情况：
  1. Excutor的Container内存： 堆外内存 + Excutor内存。
     (1) 堆外内存 = 4 * MAX(executor-memory*0.10, 384) = 4 * 384。 由于yarn.scheduler.increment-allocation-mb设置为512M，因此384向上取增量大小512M,因此堆外内存： 4*512M = 2048M
     (2) Excutor内存： 4 * 3 * 1024 = 12288M
  Excutor的Container内存： 2048M + 12288M = 14336M
  2. AM(Application Master)的Container内存： 堆外内存 + AM内存
     AM内存默认是512M。
     堆外内存 = MAX(AM内存*0.10, 384) = 384M， 根据规整因子512M, 堆外内存取512M。
     AM内存的Container内存： 512M + 512M = 1024
  总的内存大小：Excutor的Container内存 + AM的Container内存 = 14336M + 1024M = 15360M

注意： Driver的内存分配在Yarn的管理页面没有体现。

5 Spark on YARN占用资源分析-场景2

场景2：任务提交参数

```
spark-shell --master yarn \
--master yarn-client  \
--num-executors 4 \
--driver-memory 10g \
--executor-memory 3g \
--executor-cores 6 \
--conf spark.yarn.executor.memoryOverhead=1024m \
--conf spark.yarn.am.memory=1000m \
--conf spark.yarn.am.memoryOverhead=600m
```

Running Containers	Allocated CPU VCores	Allocated Memory MB	Reserved CPU VCores
5	25	18432	0

从Yarn的管理页面， 分配了5个Container， 25个Cpu Core， 18432M内存

5 Spark on YARN占用资源分析--场景2

- 占用的CPU Core数
  Excutor的Core数 = 4 * 6个core = 24个
  AM的Core数默认为1
  占用的CPU Core数： Excutor的Core数 + AM的Core数 = 4 * 5 + 1 = 25个。

- 占用的Container数
  4个Excutor 进程 + 1个Applicatoin Master进程 = 5个Container

- 占用的资源情况：
  1. Excutor的Container内存： 堆外内存 + Excutor内存。
     (1) 堆外内存 = 4 * 1024 = 4048M
     (2) Excutor内存： 4 * 3 * 1024 = 12288M
  Excutor的Container内存： 4096M+ 12288M = 16336M
  2. AM(Application Master)的Container内存： 堆外内存 + AM内存
     AM内存是1000M， 根据参数参数分配1024M。
     堆外内存 = 1024M。
     AM内存的Container内存： 1024M + 1024M = 2048M。

  总的内存大小：Excutor的Container内存 + AM的Container内存 = 16336M+ 2048M= 18432M

注意： Driver的内存分配在Yarn的管理页面没有体现。

5 Spark 资源分配调优

http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/

1. Application Master， 是一个非Executor的容器， 它会从YARN申请Container运行。 因此， 在考虑spark的资源分配的时候， 需要将Application Master所需要的内存和cpu core考虑在内。
   在yarn client模式， spark版本为1.6， 默认是512M 和 一个vcore。
   在yarn cluster模式， driver是运行在application上面， 因此可以设置--driver-memory 和--driver-cores属性强化AM的资源。

2. 运行的一个executor内存过大通常会导致过多的垃圾回收延迟。对于单个executor来说，64GB是一个很好的上限。

3. 最好将executor的core数量保持在5个以下， 即 每个executor 最多可以指定五个任务， 以实现完全的写入吞吐量。

4. 每个executor的core数量太小， 比如一个executor只分配一个core和足够的内存，这会导致申请非常多的executor。这个将抛弃使用单个JVM运行多个任务带来的好处。 首先是JVM自身需要开销， 此外广播变量是在每个executor上复制一次， 如果很多小的executor会导致更多的数据副本。

5 Spark 资源分配调优-案例

http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/

5 Spark 资源分配调优-案例

5 Spark 资源分配调优-案例

Thanks

FAQ时间