Spark调优

内存溢出两点：

- Driver内存不够
- Executor内存不够

Driver 内存不够两点：

- 读取数据太大
- 数据回传

*Executor*内存不够两点：

- map 类操作产生大量数据，包括 *map*、*flatMap*、*filter*、*mapPartitions* 等
- shuffle后产生数据倾斜

*Driver heap OOM*的三大原因*:*

(1).用户在Driver端口生成大对象,比如创建了一个大的集合数据结构

解决思路:

- 考虑将该大对象转化成Executor端加载. 例如调用sc.textFile/sc.hadoopFile等
- 如若无法避免, 自我评估该大对象占用的内存, 相应增加driver-memory的值

(2).从Executor端收集数据回Driver端

比如***Collect***. 某个Stage中Executor端发回的所有数据量不能超过spark.driver.maxResultSize，默认1g. 如果用户增加该值, 请对应增加2delta increase到Driver Memory, resultSize该值只是数据序列化之后的Size, 如果是Collect的操作会将这些数据反序列化收集, 此时真正所需内存需要膨胀2-5倍, 甚至10倍. 解决思路:

- 本身不建议将大的数据从Executor端, collect回来. 建议将Driver端对collect回来的数据所做的操作, 转化成Executor端RDD操作.

- 如若无法避免, 自我评collect需要的内存, 相应增加driver-memory的值

(3)Spark本身框架的数据消耗.

现在在Spark1.6版本之后主要由Spark UI数据消耗, 取决于作业的累计Task个数.

解决思路:

- 考虑缩小大numPartitions的Stage的partition个数, 例如从HDFS load的partitions一般自动计算, 但是后续用户的操作中做了过滤等操作已经大大减少数据量, 此时可以缩小Parititions。

- 通过参数spark.ui.retainedStages(默认1000)/spark.ui.retainedJobs(默认1000)控制.

- 实在没法避免, 相应增加内存.***--driver-memory MEM***

Spark 内存模型：

Spark在一个Executor中的内存分为三块，一块是execution内存，一块是storage内存，一块是other内存。

- execution内存是执行内存，文档中说join，aggregate都在这部分内存中执行，shuffle的数据也会先缓存在这个内存中，满了再写入磁盘，能够减少IO。其实map过程也是在这个内存中执行的。

- storage内存是存储broadcast，cache，persist数据的地方。

- other内存是程序执行时预留给自己的内存。

Executor heap

Executor内存不够有个通用的解决办法就是增加Executor内--executor-memory MEM

map过程产生大量对象导致内存溢出：

在不增加内存的情况下，可以通过减少每个Task的大小，以便达到每个Task即使产生大量的对象Executor的内存也能够装得下。

具体做法可以在会产生大量对象的map操作之前调用repartition方法，分区成更小的块传入map。

例如：rdd.repartition(10000).map(x=>for(i <- 1 to 10000) yield i.toString)。

面对这种问题注意，不能使用rdd.coalesce方法，这个方法只能减少分区，不能增加分区，不会有shuffle的过程

coalesce调用导致内存溢出

解决这个问题的方法是令程序按照想的先执行100个Task再将结果合并成10个文件，这个问题同样可以通过repartition解决，调用repartition(10)，因为这就有一个shuffle的过程，shuffle前后是两个Stage，一个100个分区，一个是10个分区，就能按照想法执行。

Shuffle后内存溢出

shuffle内存溢出的情况可以说都是shuffle后，单个文件过大导致的。Shufflel类算子发生shuffle时，需要传入一个partitioner，大部分Spark中的shuffle操作，默认的partitioner都是HashPatitioner，默认值是父RDD中最大的分区数,这个参数通过spark.default.parallelism控制(在spark-sql中用spark.sql.shuffle.partitions)，

spark.default.parallelism参数只对HashPartitioner有效，所以如果是别的Partitioner或者自己实现的Partitioner就不能使用spark.default.parallelism这个参数来控制shuffle的并发量了。

如果是别的partitioner导致的shuffle内存溢出，就需要从partitioner的代码增加partitions的数量。

解决方法：

- **提高任务并行度。**
- **或者代码层面增加partitions的数量**

Standalone模式下资源分配不均匀导致内存溢出

在standalone的模式下如果配置了--total-executor-cores 和 --executor-memory 这两个参数，但是没有配置--executor-cores这个参数的话，就有可能导致，每个Executor的memory是一样的，但是cores的数量不同，那么在cores数量多的Executor中，由于能够同时执行多个Task，就容易导致内存溢出的情况。

这种情况的解决方法就是同时配置--executor-cores或者spark.executor.cores参数，确保Executor资源分配均匀。

在RDD中，共用对象能够减少OOM的情况：

这个比较特殊，这里说记录一下，遇到过一种情况，类似这样rdd.flatMap(x=>for(i <- 1 to 1000) yield ("key","value"))导致OOM，但是在同样的情况下，使用rdd.flatMap(x=>for(i <- 1 to 1000) yield "key"+"value")就不会有OOM的问题，这是因为每次("key","value")都产生一个Tuple对象，而"key"+"value"，不管多少个，都只有一个对象，指向常量池。

如果RDD中有大量的重复数据,或者Array中需要存大量重复数据的时候我们都可以将重复数据转化为String,能够有效的减少内存使用.

优化：

- 使用mapPartitions代替大部分map操作，或者连续使用的map操作
- 整个分区的操作，减少了中间结果的输出，避免了频繁的创建了对象。

- DataFrame代替 RDD任务被划分成多个 stage，在每个 stage 内部，RDD 是无法自动优化的，而 DataFrame使用 sql 查询，自带 sql 优化器，可自动找到最优方案

*broadcast join*和普通*join*

在大数据分布式系统中，大量数据的移动对性能的影响也是巨大的。基于这个思想，在两个RDD进行join操作的时候，如果其中一个RDD相对小很多，可以将小的RDD进行collect操作然后设置为broadcast变量，这样做之后，另一个RDD就可以使用map操作进行join，这样**能够有效的减少相对大很多的那个*RDD***的数据移动**。

先*filter*再*join*

这个就是谓词下推，这个很显然，filter之后再join，shuffle的数据量会减少，这里提一点是spark-sql的优化器已经对这部分有优化了，不需要用户显示的操作，个人实现rdd的计算的时候需要注意这个。

*partitonBy*优化

如果一个RDD需要多次在join(特别是迭代)中使用,那么事先使用partitionBy对RDD进行分区,可以减少大量的shuffle.

*combineByKey*的使用

因为combineByKey是Spark中一个比较核心的高级函数，其他一些高阶键值对函数底层都是用它实现的。诸如groupByKey,reduceByKey等等

在内存不足的使用使用

rdd.persist(StorageLevel.MEMORY_AND_DISK_SER)代替rdd.cache()

rdd.cache()和rdd.persist(Storage.MEMORY_ONLY)是等价的，在内存不足的时候rdd.cache()的数据会丢失，

再次使用的时候会重算，而rdd.persist(StorageLevel.MEMORY_AND_DISK_SER)在内存不足的时候会存储在磁盘，避免重算，只是消耗点IO时间。

在*spark*使用*hbase*的时候，*spark*和*hbase*搭建在同一个集群

在spark结合hbase的使用中，spark和hbase最好搭建在同一个集群上上，或者spark的集群节点能够覆盖hbase的所有节点。hbase中的数据存储在HFile中，通常单个HFile都会比较大。

另外Spark在读取Hbase的数据的时候，不是按照一个HFile对应一个RDD的分区，而是一个region对应一个RDD分区。所以在Spark读取Hbase的数据时，通常单个RDD都会比较大，如果不是搭建在同一个集群，数据移动会耗费很多的时间。

参数优化部分：

spark.driver.memory (default 1g)

这个参数用来设置Driver的内存。在Spark程序中，SparkContext，DAGScheduler都是运行在Driver端的。对应rdd的Stage切分也是在Driver端运行，如果用户自己写的程序有过多的步骤，切分出过多的Stage，这部分信息消耗的是Driver的内存，这个时候就需要调大Driver的内存。

spark.rdd.compress (default false)

这个参数在内存吃紧的时候，又需要persist数据有良好的性能，就可以设置这个参数为true，这样在使用persist(StorageLevel.MEMORY_ONLY_SER)的时候，就能够压缩内存中的rdd数据。

减少内存消耗，就是在使用的时候会占用CPU的解压时间。

spark.serializer (default org.apache.spark.serializer.JavaSerializer)

建议设置为 org.apache.spark.serializer.KryoSerializer，因为KryoSerializer比JavaSerializer快，但是有可能会有些Object会序列化失败，这个时候就需要显示的对序列化失败的类进行KryoSerializer的注册，这个时候要配置spark.kryo.registrator参数或者使用参照如下代码：

```
val conf=newSparkConf().setMaster(...).setAppName(...)

conf.registerKryoClasses(Array(classOf[MyClass1],classOf[MyClass2]))

valsc =newSparkContext(conf)
```

spark.memory.storageFraction (default 0.5)

这个参数设置内存表示 Executor内存中 storage/(storage+execution)，虽然spark-1.6.0+的版本内存storage和execution的内存已经是可以互相借用的了，但是借用和赎回也是需要消耗性能的，所以如果明知道程序中storage是多是少就可以调节一下这个参数。

spark.locality.wait (default 3s)

spark中有4中本地化执行level，

PROCESS_LOCAL->NODE_LOCAL->RACK_LOCAL->ANY,

一个task执行完，等待spark.locality.wait时间如果，第一次等待PROCESS的Task到达，如果没有，等待任务的等级下调到NODE再等待spark.locality.wait时间，依次类推，直到ANY。

分布式系统是否能够很好的执行本地文件对性能的影响也是很大的。如果RDD的每个分区数据比较多，每个分区处理时间过长，就应该把 spark.locality.wait 适当调大一点，让Task能够有更多的时间等待本地数据。

特别是在使用persist或者cache后，这两个操作过后，在本地机器调用内存中保存的数据效率会很高，但是如果需要跨机器传输内存中的数据，效率就会很低。

spark.speculation (default false):

一个大的集群中，每个节点的性能会有差异，spark.speculation这个参数表示空闲的资源节点会不会尝试执行还在运行，并且运行时间过长的Task，避免单个节点运行速度过慢导致整个任务卡在一个节点上。

这个参数最好设置为true。与之相配合可以一起设置的参数有spark.speculation.×开头的参数。参考中有文章详细说明这个参数。

Spark shuffle优化

spark.shuffle.file.buffer

默认值：32k

参数说明：该参数用于设置shuffle write task的BufferedOutputStream的buffer缓冲大小。将数据写到磁盘文件之前，会先写入buffer缓冲中，待缓冲写满之后，才会溢写到磁盘。

调优建议：如果作业可用的内存资源较为充足的话，可以适当增加这个参数的大小（比如64k），从而减少shuffle write过程中溢写磁盘文件的次数，也就可以减少磁盘IO次数，进而提升性能。在实践中发现，合理调节该参数，性能会有1%~5%的提升。

spark.reducer.maxSizeInFlight

默认值：48m

参数说明：该参数用于设置shuffle read task的buffer缓冲大小，而这个buffer缓冲决定了每次能够拉取多少数据。

调优建议：如果作业可用的内存资源较为充足的话，可以适当增加这个参数的大小（比如96m），从而减少拉取数据的次数，也就可以减少网络传输的次数，进而提升性能。在实践中发现，合理调节该参数，性能会有1%~5%的提升。

spark.shuffle.io.maxRetries

默认值：3

参数说明：shuffle read task从shuffle write task所在节点拉取属于自己的数据时，如果因为网络异常导致拉取失败，是会自动进行重试的。该参数就代表了可以重试的最大次数。如果在指定次数之内拉取还是没有成功，就可能会导致作业执行失败。

调优建议：对于那些包含了特别耗时的shuffle操作的作业，**建议增加重试最大次数**（比如60次），以避免由于JVM的full gc或者网络不稳定等因素导致的数据拉取失败。在实践中发现，对于针对超大数据量（数十亿~上百亿）的shuffle过程，调节该参数可以大幅度提升稳定性。

spark.shuffle.io.retryWait

默认值：5s

参数说明：具体解释同上，该参数代表了**每次重试拉取数据的等待间隔**，默认是5s。

调优建议：建议加大间隔时长（比如60s），以增加shuffle操作的稳定性。

spark.shuffle.memoryFraction

默认值：0.2

参数说明：该参数代表了Executor内存中，分配给shuffle read task进行聚合操作的内存比例，默认是20%。

调优建议：在资源参数调优中讲解过这个参数。如果内存充足，而且很少使用持久化操作，建议调高这个比例，给shuffle read的聚合操作更多内存，以避免由于内存不足导致聚合过程中频繁读写磁盘。在实践中发现，合理调节该参数可以将性能提升10%左右。

spark.shuffle.manager

默认值：sort

参数说明：该参数用于设置ShuffleManager的类型。Spark 1.5以后，有三个可选项：hash、sort和tungsten-sort。HashShuffleManager是Spark 1.2以前的默认选项，但是Spark 1.2以及之后的版本默认都是SortShuffleManager了。tungsten-sort与sort类似，但是使用了tungsten计划中的堆外内存管理机制，内存使用效率更高。

调优建议：由于SortShuffleManager默认会对数据进行排序，因此如果你的业务逻辑中需要该排序机制的话，则使用默认的SortShuffleManager就可以；而如果业务逻辑不需要对数据进行排序，那么建议参考后面的几个参数调优，通过bypass机制或优化的HashShuffleManager来避免排序操作，同时提供较好的磁盘读写性能。这里要注意的是，tungsten-sort要慎用，因为之前发现了一些相应的bug。

spark.shuffle.sort.bypassMergeThreshold

默认值：200

参数说明：当ShuffleManager为SortShuffleManager时，如果shuffle read task的数量小于这个阈值（默认是200），则shuffle write过程中不会进行排序操作，而是直接按照未经优化的HashShuffleManager的方式去写数据，但是最后会将每个task产生的所有临时磁盘文件都合并成一个文件，并会创建单独的索引文件。

调优建议：当你使用SortShuffleManager时，如果的确不需要排序操作，那么建议将这个参数调大一些，大于shuffle read task的数量。那么此时就会自动启用bypass机制，map-side就不会进行排序了，减少了排序的性能开销。但是这种方式下，依然会产生大量的磁盘文件，因此shuffle write性能有待提高。

spark.shuffle.consolidateFiles

默认值：false

参数说明：如果使用HashShuffleManager，该参数有效。如果设置为true，那么就会开启consolidate机制，会大幅度合并shuffle write的输出文件，对于shuffle read task数量特别多的情况下，这种方法可以极大地减少磁盘IO开销，提升性能。

调优建议：如果的确不需要SortShuffleManager的排序机制，那么除了使用bypass机制，还可以尝试将spark.shffle.manager参数手动指定为hash，使用HashShuffleManager，同时开启consolidate机制。在实践中尝试过，发现其性能比开启了bypass机制的SortShuffleManager要高出10%~30%。