近期优化了一个spark流量统计的程序，此程序跑5分钟小数据量日志不到5分钟，但相同的程序跑一天大数据量日志各种失败。经优化，使用160 vcores + 480G memory，一天的日志可在2.5小时内跑完，下面对一些优化的思路方法进行梳理。

优化的目标

1. 保证大数据量下任务运行成功
2. 降低资源消耗
3. 提高计算性能

三个目标优先级依次递减，首要解决的是程序能够跑通大数据量，资源性能尽量进行优化。

基础优化

这部分主要对程序进行优化，主要考虑stage、cache、partition等方面。

Stage

在进行shuffle操作时，如reduceByKey、groupByKey，会划分新的stage。同一个stage内部使用pipe line进行执行，效率较高；stage之间进行shuffle，效率较低。故大数据量下，应进行代码结构优化，尽量减少shuffle操作。

Cache

本例中，首先计算出一个baseRDD，然后对其进行cache，后续启动三个子任务基于cache进行后续计算。

对于5分钟小数据量，采用StorageLevel.MEMORY_ONLY，而对于大数据下我们直接采用了StorageLevel.DISK_ONLY。DISK_ONLY_2相较DISK_ONLY具有2备份，cache的稳定性更高，但同时开销更大，cache除了在executor本地进行存储外，还需走网络传输至其他节点。后续我们的优化，会保证executor的稳定性，故没有必要采用DISK_ONLY_2。实时上，如果优化的不好，我们发现executor也会大面积挂掉，这时候即便DISK_ONLY_2，也是然并卵，所以保证executor的稳定性才是保证cache稳定性的关键。

cache是lazy执行的，这点很容易犯错，例如：

```
val raw = sc.textFile(file)
val baseRDD = raw.map(...).filter(...)
baseRDD.cache()
val threadList = new Array(
  new Thread(new SubTaskThead1(baseRDD)),
  new Thread(new SubTaskThead2(baseRDD)),
  new Thread(new SubTaskThead3(baseRDD))
)
threadList.map(_.start())
threadList.map(_.join())
```

这个例子在三个子线程开始并行执行的时候，baseRDD由于lazy执行，还没被cache，这时候三个线程会同时进行baseRDD的计算，cache的功能形同虚设。可以在baseRDD.cache()后增加baseRDD.count()，显式的触发cache，当然count()是一个action，本身会触发一个job。

再举一个错误的例子：

```
val raw = sc.textFile(file)
val pvLog = raw.filter(isPV(_))
val clLog = raw.filter(isCL(_))
val baseRDD = pvLog.union(clLog)
val baseRDD.count()
```

由于textFile()也是lazy执行的，故本例会进行两次相同的hdfs文件的读取，效率较差。解决办法，是对pvLog和clLog共同的父RDD进行cache。

Partition

一个stage由若干partition并行执行，partition数是一个很重要的优化点。

本例中，一天的日志由6000个小文件组成，加上后续复杂的统计操作，某个stage的parition数达到了100w。parition过多会有很多问题，比如所有task返回给driver的MapStatus都已经很大了，超过spark.driver.maxResultSize（默认1G），导致driver挂掉。虽然spark启动task的速度很快，但是每个task执行的计算量太少，有一半多的时

间都在进行task序列化，造成了浪费，另外shuffle过程的网络消耗也会增加。

对于reduceByKey()，如果不加参数，生成的rdd与父rdd的parition数相同，否则与参数相同。还可以使用coalesce()和repartition()降低parition数。例如，本例中由于有6000个小文件，导致baseRDD有6000个parition，可以使用coalesce()降低parition数，这样parition数会减少，每个task会读取多个小文件。

```
1. val raw = sc.textFile(file).coalesce(300)
2. val baseRDD = raw.map(...).filter(...)
3. baseRDD.cache()
```

那么对于每个stage设置多大的partition数合适那？当然不同的程度的复杂度不同，这个数值需要不断进行调试，本例中经测试保证每个parition的输入数据量在1G以内即可，如果parition数过少，每个parition读入的数据量变大，会增加内存的压力。例如，我们的某一个stage的ShuffleRead达到了3T，我设置parition数为6000，平均每个parition读取500M数据。

```
1. val bigRDD = ...
2. bigRDD.coalesce(6000).reduceBy(...)
```

最后，一般我们的原始日志很大，但是计算结果很小，在saveAsTextFile前，可以减少结果rdd的parition数目，这样会计算hdfs上的结果文件数，降低小文件数会降低hdfs namenode的压力，也会减少最后我们收集结果文件的时间。

```
1. val resultRDD = ...
2. resultRDD.repartition(1).saveAsTextFile(output)
```

这里使用repartition()不使用coalesce()，是为了不降低resultRDD计算的并发量，通过再做一次shuffle将结果进行汇总。

资源优化

在搜狗我们的spark程序跑在yarn集群上，我们应保证我们的程序有一个稳定高效的集群环境。

设置合适的资源参数

一些常用的参数设置如下：

```
1. --queue：集群队列
```

2. --num-executors：executor数量，默认2
3. --executor-memory：executor内存，默认512M
4. --executor-cores：每个executor的并发数，默认1

executor的数量可以根据任务的并发量进行估算，例如我有1000个任务，每个任务耗时1分钟，若10个并发则耗时100分钟，100个并发耗时10分钟，根据自己对并发需求进行调整即可。默认每个executor内有一个并发执行任务，一般够用，也可适当增加，当然内存的使用也会有所增加。

对于yarn-client模式，整个application所申请的资源为：

1. total vores = executor-cores * num-executors + spark.yarn.am.cores
2. total memory= (executor-memory + spark.yarn.executor.memoryOverhead) * num-executors + (spark.yarn.am.memory + spark.yarn.am.memoryOverhead)

当申请的资源超出所指定的队列的min cores和min memory时，executor就有被yarn kill掉的风险。而spark的每个stage是有状态的，如果被kill掉，对性能影响比较大。例如，本例中的baseRDD被cache，如果某个executor被kill掉，会导致其上的cache的parition失效，需要重新计算，对性能影响极大。

这里还有一点需要注意，executor-memory设置的是executor jvm启动的最大堆内存，java内存除了堆内存外，还有栈内存、堆外内存等，所以spark使用spark.yarn.executor.memoryOverhead对非堆内存进行限制，也就是说executor-memory + spark.yarn.executor.memoryOverhead是所能使用的内存的上线，如果超过此上线，就会被yarn kill掉。本次优化，堆外内存的优化起到了至关重要的作用，我们后续会看到。

spark.yarn.executor.memoryOverhead默认是executor-memory * 0.1，最小是384M。比如，我们的executor-memory设置为1G，spark.yarn.executor.memoryOverhead是默认的384M，则我们向yarn申请使用的最大内存为1408M，但由于yarn的限制为倍数（不知道是不是只是我们的集群是这样），实际上yarn运行我们运行的最大内存为2G。这样感觉浪费申请的内存，申请的堆内存为1G，实际上却给我们分配了2G，如果对spark.yarn.executor.memoryOverhead要求不高

的话，可以对executor-memory再精细化，比如申请executor-memory为640M，加上最小384M的spark.yarn.executor.memoryOverhead，正好一共是1G。
除了启动executor外，spark还会启动一个am，可以使用spark.yarn.am.memory设置am的内存大小，默认是512M，spark.yarn.am.memoryOverhead默认也是最小384M。有时am会出现OOM的情况，可以适当调大spark.yarn.am.memory。
executor默认的永久代内存是64K，可以看到永久代使用率长时间为99%，通过设置spark.executor.extraJavaOptions适当增大永久代内存，例如：–conf spark.executor.extraJavaOptions=”-XX:MaxPermSize=64m”
driver端在yarn-client模式下运行在本地，也可以对相关参数进行配置，如–driver-memory等。

查看日志

executor的stdout、stderr日志在集群本地，当出问题时，可以到相应的节点查询，当然从web ui上也可以直接看到。
executor除了stdout、stderr日志，我们可以把gc日志打印出来，便于我们对jvm的内存和gc进行调试。

```
--conf spark.executor.extraJavaOptions="-XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -XX:+PrintGCApplicationStoppedTime -XX:+PrintHeapAtGC -XX:+PrintGCApplicationConcurrentTime -Xloggc:gc.log"
```

除了executor的日志，nodemanager的日志也会给我们一些帮助，比如因为超出内存上限被kill、资源抢占被kill等原因都能看到。
除此之外，spark am的日志也会给我们一些帮助，从yarn的application页面可以直接看到am所在节点和log链接。

复杂的集群环境

我们的yarn集群节点上上跑着mapreduce、hive、pig、tez、spark等各类任务，除了内存有所限制外，CPU、带宽、磁盘IO等都没有限制（当然，这么做也是为了提高集群的硬件利用率），加上集群整体业务较多负载较高，使得spark的执行环境十分恶劣。常见的一些由于集群环境，导致spark程序失败或者性能下降的情况有：

- 节点挂掉，导致此节点上的spark executor挂掉
- 节点OOM，把节点上的spark executor kill掉
- CPU使用过高，导致spark程序执行过慢
- 磁盘目录满，导致spark写本地磁盘失败
- 磁盘IO过高，导致spark写本地磁盘失败
- HDFS挂掉，hdfs相关操作失败

内存/GC优化

经过上述优化，我们的程序的稳定性有所提升，但是让我们完全跑通的最后一根救命稻草是内存、GC相关的优化。

Direct Memory

我们使用的spark版本是1.5.2（更准确的说是1.5.3-shapshot），shuffle过程中block的传输使用
netty（spark.shuffle.blockTransferService）。基于netty的shuffle，使用direct memory存进行
buffer（spark.shuffle.io.preferDirectBufs），所以在大数据量shuffle时，堆外内存使用较多。当然，也可以使用传统的nio方式处理shuffle，但是此方式在spark 1.5版本设置为deprecated，并将会在1.6版本彻底移除，所以我最终还是采用了netty的shuffle。
jvm关于堆外内存的配置相对较少，通过-XX:MaxDirectMemorySize可以指定最大的direct memory。默认如果不设置，则与最大堆内存相同。
Direct Memory是受GC控制的，例如ByteBuffer bb = ByteBuffer.allocateDirect(1024)，这段代码的执行会在堆外占用1k的内存，Java堆内只会占用一个对象的指针引用的大小，堆外的这1k的空间只有当bb对象被回收时，才会被回收，这里会发现一个明显的不对称现象，就是堆外可能占用了很多，而堆内没占用多少，导致还没触发GC。加上-XX:MaxDirectMemorySize这个大小限制后，那么只要Direct Memory使用到达了这个大小，就会强制触发GC，这个大小如果设置的不够用，那么在日志中会看到java.lang.OutOfMemoryError: Direct buffer memory。
例如，在我们的例子中，发现堆外内存飙升的比较快，很容易被yarn

kill掉，所以应适当调小-XX:MaxDirectMemorySize（也不能过小，否则会报Direct buffer memory异常）。当然你也可以调大spark.yarn.executor.memoryOverhead，加大yarn对我们使用内存的宽容度，但是这样比较浪费资源了。

GC优化

GC优化前，最好是把gc日志打出来，便于我们进行调试。

```
--conf spark.executor.extraJavaOptions="-XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -XX:+PrintGCApplicationStoppedTime -XX:+PrintHeapAtGC -XX:+PrintGCApplicationConcurrentTime -Xloggc:gc.log"
```

通过看gc日志，我们发现一个case，特定时间段内，堆内存其实很闲，堆内存使用率也就5%左右，长时间不进行父gc，导致Direct Memory一直不进行回收，一直在飙升。所以，我们的目标是让父gc更频繁些，多触发一些Direct Memory回收。

第一，可以减少整个堆内存的大小，当然也不能太小，否则堆内存也会报OOM。这里，我配置了1G的最大堆内存。

第二，可以让年轻代的对象尽快进入年老代，增加年老代的内存。这里我使用了-Xmn100m，将年轻代大小设置为100M。另外，年轻代的对象默认会在young gc 15次后进入年老代，这会造成年轻代使用率比较大，young gc比较多，但是年老代使用率低，父gc比较少，通过配置-XX:MaxTenuringThreshold=1，年轻代的对象经过一次young gc后就进入年老代，加快年老代父gc的频率。

第三，可以让年老代更频繁的进行父gc。一般年老代gc策略我们主要有-XX:+UseParallelOldGC和-XX:+UseConcMarkSweepGC这两种，ParallelOldGC吞吐率较大，ConcMarkSweepGC延迟较低。我们希望父gc频繁些，对吞吐率要求较低，而且ConcMarkSweepGC可以设置-XX:CMSInitiatingOccupancyFraction，即年老代内存使用率达到什么比例时触发CMS。我们决定使用CMS，并设置-XX:CMSInitiatingOccupancyFraction=10，即年老代使用率10%时触发父gc。

通过对GC策略的配置，我们发现父gc进行的频率加快了，带来好处就是Direct Memory能够尽快进行回收，当然也有坏处，就是gc时间增加

了，cpu使用率也有所增加。

最终我们对executor的配置如下：

```
--executor-memory 1G --num-executors 160 --executor-cores 1 --conf
spark.yarn.executor.memoryOverhead=2048 --conf
spark.executor.extraJavaOptions="-XX:MaxPermSize=64m -
XX:+CMSClassUnloadingEnabled -XX:MaxDirectMemorySize=1536m -
Xmn100m -XX:MaxTenuringThreshold=1 -XX:+UseConcMarkSweepGC -
XX:+CMSParallelRemarkEnabled -XX:+UseCMSCompactAtFullCollection -
XX:+UseCMSInitiatingOccupancyOnly -
XX:CMSInitiatingOccupancyFraction=10 -XX:+UseCompressedOops -
XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -
XX:+PrintGCDateStamps -XX:+PrintGCApplicationStoppedTime -
XX:+PrintHeapAtGC -XX:+PrintGCApplicationConcurrentTime -Xloggc:gc.log
-XX:+HeapDumpOnOutOfMemoryError"
```

总结

通过对Stage/Cache/Partition、资源、内存/GC的优化，我们的spark程序最终能够在160 vcores + 480G memory资源下，使用2.5小时跑通一天的日志。

对于程序优化，我认为应本着如下几点进行：

1. 通过监控CPU、内存、网络、IO、GC、应用指标等数据，切实找到系统的瓶颈点。
2. 统筹全局，制定相应的解决方案，解决问题的思路是否清晰准确很重要，另外切勿『头疼医头，脚疼医脚』，应总体考虑把握。
3. 了解一些技术的背景知识，对于每次优化尽量做得彻底些，多进行总结。