Performance Tunning

8mins, 44sec TO 3mins, 45sec

硬件级别

提高磁盘IO的性能

noatime 我为两台slaves server设置了noatime. vi /etc/fstab.map task的平均执行时间减少两秒,这影响硬盘IO的性能,shuffle的时间也相应地减少了1分钟,不影响reduce的执行时间

client端设置

map与reduce task数量

map task的数量由split的数量决定,split的数据越小,每个map task执行的时间就越短,但相应地,job的执行时间就拉长了, 因为内部调度的时间更长了
benchmark:
之前是67个map task,平均执行时间是16秒, job完成时间接近7分钟
后来map task变成265个, 平均每个map task执行8秒,但job完成时间差不多12分钟

reduce task的数量由client来设置
我测试的结果client设置result task略大于或等于集群reduce slot, 当然这是整个集群只有一个job在执行的情况下,当有多个job执行时, 网上的建议是少于集群reduce slots总量
集群reduce slots数量是4,我设置reduce数量成8的时候,每个reduce执行的很快,shuffle过程也短,最终job完成时间差不多是7分钟,而设置为2时,shuffle时间很长,job完成时间为12分钟.当我再设置为4个reduce task时, 执行时间差不多8分钟

后来我又做了三个长时间job并发运行的测试,结果显示纵使有很多个map slot在运行, 两台slaves的CPU与内存利用率不是很离谱, 但不同的场景应该有不同的设置,主要还是根据slave的负载来决定. 查看slave机器的负载可使用top命令

使用压缩

使用压缩,可有效减少传输数据量与存储数据量.
conf.setBoolean("mapreduce.output.fileoutputformat.compress", true);
conf.setClass("mapreduce.output.fileoutputformat.compress.codec", GzipCodec.class, CompressionCodec.class);
像这样使用的话,只会在reduce完成后对输出数据做压缩,我测试的结果是原本200MB的数据在使用Gzip压缩后就变成60多MB,压缩率差不多是60%多.可能因为是要对输出数据做压缩,reduce执

行的时间有少量增加,在我的测试中增加了差不多半分钟

想要压缩map的中间结果,就这样设置
conf.setBoolean("mapreduce.map.output.compress", true);
conf.setClass("mapreduce.map.output.compress.codec", GzipCodec.class, CompressionCodec.class);
从最终执行的Counter上来看, Reduce shuffle bytes项由最初的2,299,992,204变成495,498,004, 压缩率差不多是80%,相应地, map的平均执行时间由21秒增加到25秒. 很奇怪的是shuffle的过程增加了两分钟,难道是因为shuffle过程在等待map task压缩结果. reduce的执行时间减少了,这是正常的

具体使用哪种压缩算法也是需要考虑的

但压缩与job执行时间还是有冲突,如果在乎存储或网络流量等资源，就有必要压缩，如果在乎job完成时间, 就不要设置压缩

Code级设置

重用writable类型

我想着重用Writable类型应该会让程序执行加快,但每次测试的结果都是相反的. 比如我在wordcount时重用mapper中的加1,map task的平均执行时长从30秒增加到35秒,且map task的总GC时间由40s增加到120s,不可理喻

使用合适的Writable类型

这点太重要了,但容易被忽略. 在大部分情况下,这些数值都是挺小的.我们一般使用IntWritable类型来计数,但IntWritable类型固定占四个字节,用它来存储小数值有些浪费. 在我的WordCount例子中,统计的叠加都是一些简单的数值,所以就使用VIntWritable类型.它可能根据数值的大小来适应不同的存储字节, 这样节省资源.

Mapper与reducer中value使用了VIntWritable类型。尽管map 与reduce task的平均执行时间都有增长,但总的执行时间倒下降了一分钟. 最有效的是, map的中间结果小了600多MB, 基本上节省了30%的map 端存储. 很给力

jobtracker设置

使用合适的Task Scheduler

MapReduce默认的task scheduler很不给力,基本不能满足

增加Jobtracker处理RPC线程的数量

在集群有些大的时候, jobtracker的处理RPC能力就有些不足了.之前看到的评估是, jobtracker为每个tasktracker服务的时间差不多是100ms,当前默认有10个handler, 所以jobtracker每秒可处理差不多100个tasktracker请求, 如果tasktracker数量很大,那么很有必要增加这个设置来提高jobtracker的响应. mapreduce.jobtracker.handler.count

增加分解job到task的线程数

当client提交的job很多时, jobtracker需要将每个job分解为task,然后交由task scheduler来管理. 这个负责分解job的线程池默认有4条线程，我们可以由业务需求,适当扩大这个线程池的处理能力 mapreduce.jobtracker.jobinit.threads

避免推测式task

从感觉上看推测式task是有意义的, 可以让job tracker有机会来弥补效率低task的过错.但在无视造成task效率低的原因时,启动推测式task,反倒增加了系统的负载. 在正常情况下可以拒绝推测式task的运行.
要么设置
job.setMapSpeculativeExecution(false);
job.setReduceSpeculativeExecution(false);
要么设置
conf.setBoolean("mapreduce.map.speculative", false);
conf.setBoolean("mapreduce.reduce.speculative", false);

测试结果很理想, 整个job的运行时间由最初的8分半到现在的6分半. 每个task的运行时间基本没变，但减少了很多莫名其妙的task执行, 甚好

tasktracker及task执行期设置

考虑使用Partitioner

这不是一条性能参考点, 但还是与业务相关. 默认是使用HashPartitioner, 只简单的将key hash, 然后转到不同的reducer, 可能不太适合于某些场景. 如果自己想实现Partitioner, 那么就像这样
public class OnlyOnePartitioner<KEY, VALUE> extends Partitioner<KEY, VALUE>

自己实现了Partitioner, 测试中发现map task的平均执行时间从30s下降到27s, 不确定是由于引入Partitioner导致的

适当添加Combiner

因为我的测试例子是wordcount, 添加Combiner后reduce执行时间降低为之前的一半. 重要的是map端有个counter是 FILE_BYTES_WRITTEN, 它的记录从2.3G下降到1.5G,数据量少了很多. job总的完成时间减少一分钟.很多例子是不能添加combiner的,慎重

Combiner不像书上说的只在merge阶段才运行, 它在0.21.0的中有两个地方运行. 一是只要有spill发生，都会使用combiner. 二是在map 执行完成时的merge阶段, 这里才是当spill文件数大于3时才使用, 哎天呐, 让我好生迷惑

调整Reducer拉取map结果的线程数

MapReduce默认使用HashPartitioner, 如果map task有100个, reducer有5个, 平均每个reducer需要拉取20个map task的输出结果, 但默认情况下reducer 会初始化5个拉取数据的线程, 逐次从map端copy. 适当地增加reduce 端拉取map 数据的线程数, 让shuffle过程执行的更快些. 可能这样设置mapreduce.reduce.shuffle.parallelcopies

测试是基于当前有67个maptask, 8个reduce task, 平均每个reduce task需要拉取近9个map task的结果，在我设置为并发copy线程数为10的时候，程序的shuffle过程比平时快了10s，并不是很突出，但能说明些问题. 在测试267个map task时 shuffle的过程平均快了1分多钟

在设置它的同步, 也需要设置另外一个参数. mapreduce.tasktracker.http.threads, 它表示每个tasktracker为http服务的线程池是多大, 默认是40, 因为shuffle截断copy数据是通过http方式, 所以这个值如果不能支持reducer 的copy请求, 那么上个参数也没有多大意义

在最近的一次测试中发现, 我为每个reduce task启动了70个拉取数据的线程请求, 为每个tasktracker设置150个线程的连接池, 运行速度快了3分钟, 尤其给力的是reduce 执行从三分钟下降到不到一分钟，shuffle阶段也快了一分多, 很有意思

重用JVM

默认情况下, tasktracker会为每个需要执行的task新开JVM.这会引入创建与销毁JVM的开销.可以设置JVM重用，在执行完task后,tasktracker不必销毁该JVM,而执行新的task. 可通过mapreduce.job.jvm.numtasks来设置,默认为1,也就是执行过一个task后就销毁,可以设置为-1, 一直保持重用.

我的测试结果是: 当使用JVM重用后, map task和shuffle的执行时间有小幅下降, 而reduce task的执行时间好像没什么变化.总的执行时间降低有一分钟

尽量让reduce merge发生在内存中

每个reduce task把map 结果copy过去时都要对从各个map 端来的数据做merge动作, 我们最希望merge就发生在内存中，然后直接作为reduce的输入. 当然这也只是我们希望，reduce端的内存没有那么大的时候, 只能把拉过来的数据先保存到本地磁盘中然后才做merge, 如果我们把reduce端的内存设置很大, 完全就可以做memtomem的merge动作, 哈

怎么调大reduce端的堆内存呢?mapreduce.reduce.java.opts, 当然这只是描述分配给执行reduce task的JVM的最大可用内存, 具体堆内存能分配到多少还得看JVM的比例设置

假如我们设置得到的堆内存是500M, 在merge发生时reduce可使用的最大内存是总的堆内存*mapreduce.reduce.shuffle.input.buffer.percent, 默认是0.9, 最后得到的这个内存才是执行merge动作的内存

如果当前拉过来的数据大于merge总内存就得把内存中的数据全部spill到磁盘中去. 如果没有这样的内存顾及, 当内存中的数据片大于10时就可以做merge了,

首先想要让所有merge都在内存中执行, 就要开启这种memtomem的模式 mapreduce.reduce.merge.memtomem.enabled, 不然所有merge结果最后都得到内存呆着

HDFS级设置

读写文件的buffer设置

io.file.buffer.size这个属性只要是读写文件, 就都得使用. 做为大数据量的buffer, 4K是有些小了 , 在我的测试中, 设置为64K, map的平均执行时间由20s下降到14s, 整个job的执行时间下降了一分钟. 当设置为128K的时候, 执行的效率下降了很多, job执行时间拉长

Tuning的结果检查

监控 Ganglia
Counter