# Hadoop

## Hadoop优化

### （★★★）对 Hadoop 有没有调优经验，没有什么使用心得？（调优从参数调优讲起）

**MR优化总结**

**优化前**

熟知业务要求

熟知数据分布状态（是否倾斜、是否是多个小文件等），可以使用采样来了解数据

**通用型优化策略**

文件存储格式

使用更加优化的格式的文件，例如Parquet、ORC，综合来说，ORC最优。

文件压缩

可以在mr各阶段启用压缩，例如：数据块可以被压缩（reduce输出可以被压缩）、map输出数据可以被压缩（减少shuffle过程中传输的数据量）

shuffle优化

核心思想：减少磁盘IO。可以通过调节一下几个方面来达到效果：

增加环形缓冲区大小

增大merge过程一次可以合并的溢写文件的数量（注意：量力（内存）而行）

增大溢写的第一级门槛

在能使用Combiner的业务场景下，尽可能的使用Combiner

合理设置Combiner的阈值

map输出的k、v尽量小

**专用型优化策略**

**减少不必要的消耗**

使用更加优化的数据格式文件，例如ORC

map端join（省去shuffle过程）

countall 或 topn场景下合理使用cleanup方法（减少shuffle过程中的数据传输量）

排序操作尽量使用shuffle内置的排序机制而不要自己手动排序

**合理进行任务分配**

**数据本身不均衡**

**数据倾斜**

通过自定义业务k，均分分散倾斜的数据，要实现同等效果还可以自定义Partitioner

大量小文件

1.当MR程序的输入是一堆小文件时，可以先将小文件合并，然后在再进行处理

2.使用Uber模式

**数据本身均衡**

合理设置ReduceTask的数量，假定map总输出的键值对的大小是10GB（并且去重后的key的数量远大于ReduceTask的数量），而一个ReduceTask所用的Continer的内存为1GB，那么合理的ReduceTask的数量为10÷（1\*0.7）≈15（这里需要采用进1法）；

为具体的任务设置合适的Continer大小，无论是MapTask还是ReduceTask，只要是cpu密集型的，其所使用的Continer就应该由较多的cpu和合适的内存所组成。同样的，如果任务是IO密集型的，其所使用的Continer就应该改由合适的cpu和较大的内存所组成。这个一般不全局调节

### （★★★）Hadoop框架中怎么来优化

（1） 从应用程序角度进行优化。由于MapReduce是迭代逐行解析数据文件的，怎样在迭代的情况下，编写高效率的应用程序，是一种优化思路。

（2） 对Hadoop参数进行调优。当前Hadoop系统有190多个配置参数，怎样调整这些参数，使Hadoop作业运行尽可能的快，也是一种优化思路。

（3） 从系统实现角度进行优化。这种优化难度是最大的，它是从Hadoop实现机制角度，发现当前Hadoop设计和实现上的缺点，然后进行源码级地修改。该方法虽难度大，但往往效果明显。

（4）Linux内核参数调整

### （★★★）从应用程序角度进行优化

（1） 避免不必要的reduce任务

如果MapReduce程序中reduce是不必要的，那么我们可以在map中处理数据, Reducer设置为0。这样避免了多余的reduce任务。

（2） 为job添加一个Combiner

为job添加一个combiner可以大大减少shuffle阶段从map task拷贝给远程reduce task的数据量。一般而言，combiner与reducer相同。

（3） 根据处理数据特征使用最适合和简洁的Writable类型

Text对象使用起来很方便，但它在由数值转换到文本或是由UTF8字符串转换到文本时都是低效的，且会消耗大量的CPU时间。当处理那些非文本的数据时，可以使用二进制的Writable类型，如IntWritable， FloatWritable等。二进制writable好处：避免文件转换的消耗；使map task中间结果占用更少的空间。

（4） 重用Writable类型

很多MapReduce用户常犯的一个错误是，在一个map/reduce方法中为每个输出都创建Writable对象。例如，你的Wordcout mapper方法可能这样写：

public void map(...) {

…

for (String word : words) {

output.collect(new Text(word), new IntWritable(1));

}

}

这样会导致程序分配出成千上万个短周期的对象。Java垃圾收集器就要为此做很多的工作。更有效的写法是：

class MyMapper … {

Text wordText = new Text();

IntWritable one = new IntWritable(1);

public void map(...) {

for (String word: words) {

wordText.set(word);

output.collect(wordText, new IntWritable(1));

}

}

}

（5） 使用StringBuffer而不是String

当需要对字符串进行操作时，使用StringBuffer而不是String，String是read-only的，如果对它进行修改，会产生临时对象，而StringBuffer是可修改的，不会产生临时对象。

### （★）怎么提升多个 JOB 同时执行带来的压力,如何优化,说说思路

增加运算能力

### （★★★）MapReduce 性能优化（压缩算法、更换调度器、设置 InputSplit 大小减少 map 任务数量、 map 和 reduce 的 slot 如何设置）？

**1.压缩算法:**

**Compression and Input Splits**

当我们使用压缩数据作为MapReduce的输入时，需要确认数据的压缩格式是否支持切片？

Bzip2的各个数据块之间存放有专门的“Synchronization Marker”，因此它是可以支持切片的。

Hadoop通常处理的都是大规模的数据集，因此我们必须尽可能的利用压缩优化性能。具体使用哪一个压缩格式依赖于文件大小、文件格式以及我们使用的分析工具。以下是一些使用建议：

（1）使用一些容器文件格式，如Sequence File、Avro DataFile、ORCFile、Parquet File，这些文件格式全部支持压缩和切片，配合一个快速的压缩算法（如：LZO、LZ4、Snappy）使用通常是一个好的选择；

（2）使用一个支持切片的压缩算法，如bzip2、lzo（通过索引处理之后可以支持切片）；

（3）将一个文件人为地切分为Chunk（即一个文件被切分为多个文件），然后将这些Chunks逐个的进行压缩，可以使用任意支持的压缩算法，且不需要考虑压缩算法是否支持切片，但是Chunk压缩后的大小需要接近于HDFS Block的大小；

（4）文件不作压缩处理。

对于大型的文件，我们不能选择不支持切片的压缩算法，因为这会导致MapReduce Job丧失数据本地性且运行效率低下。

**Using Compression in MapReduce**

MapReduce读取输入路径中的压缩文件时会自动完成数据解压（可参考CompressionCodecFactory）。

如果MapReduce Job的结果输出需要使用压缩，可以通过设置Job的相关配置属性来实现：

mapreduce.output.fileoutputformat.compress：true

mapreduce.output.fileoutputformat.compress.codec：CompressionCodec全限定类名

也可以通过FileOutputFormat提供的静态方法设置，如：

FileOutputFormat.setCompressOutput(job, true);

FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);

不同的输出文件格式可能相应的设置属性会有不同。

**Compressing map output**

Map Task的输出被写出到本地磁盘，而且需要通过网络传输至Reduce Task的节点，只要简单地使用一个快速的压缩算法（如LZO、LZ4、Snappy）就可以带来性能的提升，因为压缩机制的使用避免了Map Tasks与Reduce Tasks之间大量中间结果数据被传输。可以通过设置相应的Job配置属性开启：

mapreduce.map.output.compress：true

mapreduce.map.output.compress.codec：CompressionCodec全限定类名

也可以通过Configuration API进行设置：

new API：

Configuration conf = new Configuration();

conf.setBoolean(Job.MAP\_OUTPUT\_COMPRESS, true);

conf.setClass(Job.MAP\_OUTPUT\_COMPRESS\_CODEC, GzipCodec.class, CompressionCodec.class);

Job job = new Job(conf);

old API：

conf.setCompressMapOutput(true);

conf.setMapOutputCompressorClass(GzipCodec.class);

**2.更换YARN 资源调度器**

1. YARN-FIFO Scheduler

将所有应用程序放入到一个队列中

先进入队里排在前面的程序先获得资源

局限性

资源利用率低，无法交叉运行作业

不够灵活，比如紧急的作业无法插队，耗时长作业拖慢耗时短作业

2. YARN-多队列分开调度器

所有资源按照比例划分到不同的队列

每个队列可以实现单独的调度策略

优点

按照不同的资源使用情况将资源划分到不同队列

能够让更多的应用程序获得资源

使用灵活，资源利用率高

**3.适量增大InputSplit的大小**

**4.减少map的任务数量**

**5.设置map 和reduce 的slot**

因为hadoop 的集群所有机器不可能完全硬件配置一样，那么不同节点机器负载也不一样，不同节点上并发的最大map

和reduce数量也不一样。

可以先试验下将每台tasktracker的map槽数调整到合适的数量，使得其能执行最大map数量。

在对应节点机器上修改mapred-site.xml文件，添加如下参数：默认参数为2：

在对应的节点机器上修改mapreduce-sit.xml文件，添加参数如下：

<property>

<name>mapred.tasktracker.map.tasks.maximum</name>

<value>2</value>

<description>The maximum number of map tasks that will be run

simultaneously by a task tracker.

</description>

</property>

<property>

<name>mapred.tasktracker.reduce.tasks.maximum</name>

<value>2</value>

<description>The maximum number of reduce tasks that will be run

simultaneously by a task tracker.

</description>

</property>