Spark的性能调优

2016-01-16 四火 hadoop123

下面这些关于Spark的性能调优项，有的是来自官方的，有的是来自别的的工程师，有的则是我自己总结的。

基本概念和原则

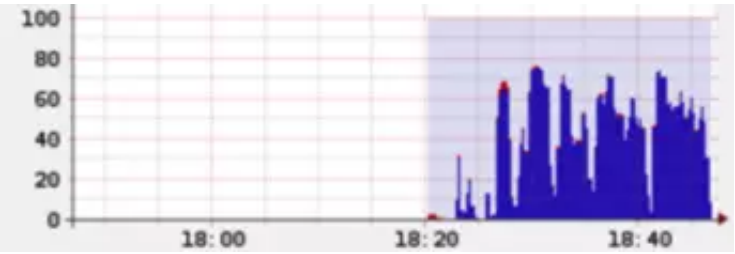
首先，要搞清楚Spark的几个基本概念和原则，否则系统的性能调优无从谈起：

- 每一台host上面可以并行N个worker，每一个worker下面可以并行M个executor，task们会被分配到executor上面去执行。Stage指的是一组并行运行的task，stage内部是不能出现shuffle的，因为shuffle的就像篱笆一样阻止了并行task的运行，遇到shuffle就意味着到了stage的边界。
- CPU的core数量，每个executor可以占用一个或多个core，可以通过观察CPU的使用率变化来了解计算资源的使用情况，例如，很常见的一种浪费是一个executor占用了多个core，但是总的CPU使用率却不高（因为一个executor并不总能充分利用多核的能力），这个时候可以考虑让么个executor占用更少的core，同时worker下面增加更多的executor，或者一台host上面增加更多的worker来增加并行执行的executor的数量，从而增加CPU利用率。但是增加executor的时候需要考虑好内存消耗的控制，以免出现Out of Memory的情况。
- partition和parallelism，partition指的就是数据分片的数量，每一次task只能处理一个partition的数据，这个值太小了会导致每片数据量太大，导致内存压力，或者诸多executor的计算能力无法利用充分；但是如果太大了则会导致分片太多，执行效率降低。在执行action类型操作的时候（比如各种reduce操作），partition的数量会选择parent RDD中最大的那一个。而parallelism则指的是在RDD进行reduce类操作的时候，默认返回数据的paritition数量（而在进行map类操作的时候，partition数量通常取自parent RDD中较大的一个，而且也不会涉及shuffle，因此这个parallelism的参数没有影响）。所以说，这两个概念密切相关，都是涉及到数据分片的，作用方式其实是统一的。通过spark.default.parallelism可以设置默认的分片数量，而很多RDD的操作都可以指定一个partition参数来显式控制具体的分片数量。
- 上面这两条原理上看起来很简单，但是却非常重要，根据硬件和任务的情况选择不同的取值。想要取一个放之四海而皆准的配置是不现实的。看这样几个例子：（1）实践中

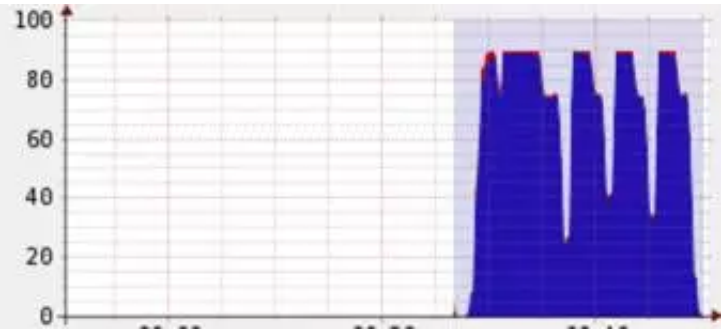
跑的EMR Spark job，有的特别慢，查看CPU利用率很低，我们就尝试减少每个executor占用CPU core的数量，增加并行的executor数量，同时配合增加分片，整体上增加了CPU的利用率，加快数据处理速度。（2）发现某job很容易发生内存溢出，我们就增大分片数量，从而减少了每片数据的规模，同时还减少并行的executor数量，这样相同的内存资源分配给数量更少的executor，相当于增加了每个task的内存分配，这样运行速度可能慢了些，但是总比OOM强。（3）数据量特别少，有大量的小文件生成，就减少文件分片，没必要创建那么多task，这种情况，如果只是最原始的input比较小，一般都能被注意到；但是，如果是在运算过程中，比如应用某个reduceBy或者某个filter以后，数据大量减少，这种低效情况就很少被留意到。

- 最后再补充一点，随着参数和配置的变化，性能的瓶颈是变化的，在分析问题的时候不要忘记。例如在每台机器上部署的executor数量增加的时候，性能一开始是增加的，同时也观察到CPU的平均使用率在增加；但是随着单台机器上的executor越来越多，性能下降了，因为随着executor的数量增加，被分配到每个executor的内存数量减小，在内存里直接操作的越来越少，spill over到磁盘上的数据越来越多，自然性能就变差了。

下面给这样一个直观的例子，当前总的cpu利用率并不高：



但是经过根据上述原则的的调整之后，可以显著发现cpu总利用率增加了：



其次，涉及性能调优我们经常要改配置，在Spark里面有三种常见的配置方式，虽然有些参数的配置是可以互相替代，但是作为最佳实践，还是需要遵循不同的情形下使用不同的配置：

1. 设置环境变量，这种方式主要用于和环境、硬件相关的配置；
2. 命令行参数，这种方式主要用于不同次的运行会发生变化的参数，用双横线开头；
3. 代码里面（比如Scala）显式设置（SparkConf对象），这种配置通常是application级别的配置，一般不改变。

举一个配置的具体例子。Node、worker和executor之间的比例调整。我们经常需要调整并行的executor的数量，那么简单说有两种方式：

- 一个是调整并行的worker的数量，比如，SPARK_WORKER_INSTANCES可以设置每个node的worker的数量，但是在改变这个参数的时候，比如改成2，一定要相应设置

SPARK_WORKER_CORES的值，让每个worker使用原有一半的core，这样才能让两个worker一同工作；

- 另一个是调整worker内executor的数量，我们是在YARN框架下采用这个调整来实现executor数量改变的，一种典型办法是，一个host只跑一个worker，然后配置spark.executor.cores为host上CPU core的N分之一，同时也设置spark.executor.memory为host上分配给Spark计算内存的N分之一，这样这个host上就能够启动N个executor。

有的配置在不同的MR框架/工具下是不一样的，比如YARN下有的参数的默认取值就不同，这点需要注意。

明确这些基础的事情以后，再来一项一项看性能调优的要点。

内存

Memory Tuning，Java对象会占用原始数据2~5倍甚至更多的空间。最好的检测对象内存消耗的办法就是创建RDD，然后放到cache里面去，然后在UI上面看storage的变化；当然也可以使用SizeEstimator来估算。使用-XX:+UseCompressedOops选项可以压缩指针（8字节变成4字节）。在调用collect等等API的时候也要小心——大块数据往内存拷贝的时候心里要清楚。内存要留一些给操作系统，比如20%，这里面也包括了OS的buffer cache，如果预留得太少了，会见到这样的错误：

> *Required executor memory (235520+23552 MB) is above the max threshold (241664 MB) of this cluster! Please increase the value of 'yarn.scheduler.maximum-allocation-mb'.*

或者干脆就没有这样的错误，但是依然有因为内存不足导致的问题，有的会有警告，比如这个：

> *16/01/13 23:54:48 WARN scheduler.TaskSchedulerImpl: Initial job has not accepted any resources; check your cluster UI to ensure that workers are registered and have sufficient memory*

Reduce Task的内存使用。在某些情况下reduce task特别消耗内存，比如当shuffle出现的时候，比如sortByKey、groupByKey、reduceByKey和join等，要在内存里面建立一个巨大的hash table。其中一个解决办法是增大level of parallelism，这样每个task的输入规模就相应减小。

注意原始input的大小，有很多操作始终都是需要某类全集数据在内存里面完成的，那么并非拼命增加parallelism和partition的值就可以把内存占用减得非常小的。我们遇到过某些性能低下甚至OOM的问题，是改变这两个参数所难以缓解的。但是可以通过增加每台机器的内存，或者增加机器的数量都可以直接或间接增加内存总量来解决。

在选择EC2机器类型的时候，要明确瓶颈（可以借由测试来明确），比如我们遇到的情况就是使用r3.8 xlarge和c3.8 xlarge选择的问题，运算能力相当，前者比后者贵50%，但是内存是后者的5倍。

CPU

Level of Parallelism。指定它以后，在进行reduce类型操作的时候，默认partition的数量就被指定

了。这个参数在实际工程中通常是必不可少的，一般都要根据input和每个executor内存的大小来确定。设置level of parallelism或者属性spark.default.parallelism来改变并行级别，通常来说，每一个CPU核可以分配2~3个task。

CPU core的访问模式是共享还是独占。即CPU核是被同一host上的executor共享还是瓜分并独占。比如YARN环境，一台机器上共有32个CPU core的资源，同时部署了两个executor，总内存是50G，那么一种方式是配置spark.executor.cores为16，spark.executor.memory为20G，这样由于内存的限制，这台机器上会部署两个executor，每个都使用20G内存，并且各使用独占的16个CPU core资源；而如果把spark.executor.cores配置为32，那么依然会部署两个executor，但是二者会共享这32个core。根据我的测试，独占模式的性能要略好与共享模式。同时，独占模式也是Spark官方文档上推荐的方式。

GC调优。打印GC信息：-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps。默认60%的executor内存可以被用来作为RDD的缓存，因此只有40%的内存可以被用来作为对象创建的空间，这一点可以通过设置spark.storage.memoryFraction改变。如果有很多小对象创建，但是这些对象在不完全GC的过程中就可以回收，那么增大Eden区会有一定帮助。如果有任务从HDFS拷贝数据，内存消耗有一个简单的估算公式——比如HDFS的block size是64MB，工作区内有4个task拷贝数据，而解压缩一个block要增大3倍大小，那么内存消耗就是：4*3*64MB。另外，工作中遇到过这样的一个问题：GC默认情况下有一个限制，默认是GC时间不能超过2%的CPU时间，但是如果大量对象创建（在Spark里很容易出现，代码模式就是一个RDD转下一个RDD），就会导致大量的GC时间，从而出现OutOfMemoryError: GC overhead limit exceeded，可以通过设置-XX:-UseGCOverheadLimit关掉它。

序列化和传输

Data Serialization，默认使用的是Java Serialization，这个程序员最熟悉，但是性能、空间表现都比较差。还有一个选项是Kryo Serialization，更快，压缩率也更高，但是并非支持任意类的序列化。在Spark UI上能够看到序列化占用总时间开销的比例，如果这个比例高的话可以考虑优化内存使用和序列化。

Broadcasting Large Variables。在task使用静态大对象的时候，可以把它broadcast出去。Spark会打印序列化后的大小，通常来说如果它超过20KB就值得这么做。有一种常见情形是，一个大表join一个小表，把小表broadcast后，大表的数据就不需要在各个node之间疯跑，安安静静地呆在本地等小表broadcast过来就好了。

Data Locality。数据和代码要放到一起才能处理，通常代码总比数据要小一些，因此把代码送到各处会更快。Data Locality是数据和处理的代码在屋里空间上接近的程度：PROCESS_LOCAL（同一个JVM）、NODE_LOCAL（同一个node，比如数据在HDFS上，但是和代码在同一个node）、NO_PREF、RACK_LOCAL（不在同一个server，但在同一个机架）、ANY。当然优先级从高到低，但是如果在空闲的executor上面没有未处理数据了，那么就有两个选择：（1）要么等如今繁忙的CPU闲下来处理尽可能"本地"的数据，（1）要么就不等直接启动task去处理相对远程的数据。默认当这种情况发生Spark会等一会儿（spark.locality），即策略（1），如果繁忙的CPU停不下来，就会执行策略（2）。

代码里对大对象的引用。在task里面引用大对象的时候要小心，因为它会随着task序列化到每个节点上去，引发性能问题。只要序列化的过程不抛出异常，引用对象序列化的问题事实上很少被人重视。如果，这个大对象确实是需要的，那么就不如干脆把它变成RDD好了。绝大多数时候，对于大对象的序列化行为，是不知不觉发生的，或者说是预期之外的，比如在我们的项目中有这样一段代码：

```
rdd.map(r => {
  println(BackfillTypeIndex)
})
```

其实呢，它等价于这样：

```
rdd.map(r => {
  println(this.BackfillTypeIndex)
})
```

对于这样的问题，一种最直接的解决方法就是：

```
val dereferencedVariable = this.BackfillTypeIndex
rdd.map(r => println(dereferencedVariable)) // "this" is not serialized
```

相关地，注解@transient用来标识某变量不要被序列化，这对于将大对象从序列化的陷阱中排除掉是很有用的。另外，注意class之间的继承层级关系，有时候一个小的case class可能来自一棵大树。

文件读写

文件存储和读取的优化。比如对于一些case而言，如果只需要某几列，使用rcfile和parquet这样的格式会大大减少文件读取成本。再有就是存储文件到S3上或者HDFS上，可以根据情况选择更合适的格式，比如压缩率更高的格式。另外，特别是对于shuffle特别多的情况，考虑留下一定量的额外内存给操作系统作为操作系统的buffer cache，比如总共50G的内存，JVM最多分配到40G多一点。

文件分片。比如在S3上面就支持文件以分片形式存放，后缀是partXX。使用coalesce方法来设置分成多少片，这个调整成并行级别或者其整数倍可以提高读写性能。但是太高太低都不好，太低了没法充分利用S3并行读写的能力，太高了则是小文件太多，预处理、合并、连接建立等等都是时间开销啊，读写还容易超过throttle。

任务

Spark的Speculation。通过设置spark.speculation等几个相关选项，可以让Spark在发现某些task执行特别慢的时候，可以在不等待完成的情况下被重新执行，最后相同的task只要有一个执行完了，那么最快执行完的那个结果就会被采纳。

减少Shuffle。其实Spark的计算往往很快，但是大量开销都花在网络和IO上面，而shuffle就是一个典型。举个例子，如果(k, v1) join (k, v2) => (k, v3)，那么，这种情况其实Spark是优化得非常好的，因为需要join的都在一个node的一个partition里面，join很快完成，结果也是在同一个node（这一系列操作可以被放在同一个stage里面）。但是如果数据结构被设计为(obj1) join (obj2) => (obj3)，而其中的join条件为obj1.column1 == obj2.column1，这个时候往往就被迫shuffle了，因为不再有同一个key使得数据在同一个node上的强保证。在一定要shuffle的情况下，尽可能减少shuffle前的数据规模，比如这个避免groupByKey的例子。下面这个比较的图片来自Spark Summit 2013的一个演讲，讲的是同一件事情：

Repartition。运算过程中数据量时大时小，选择合适的partition数量关系重大，如果太多partition就导致有很多小任务和空任务产生；如果太少则导致运算资源没法充分利用，必要时候可以使用repartition来调整，不过它也不是没有代价的，其中一个最主要代价就是shuffle。再有一个常见问题是数据大小差异太大，这种情况主要是数据的partition的key其实取值并不均匀造成的（默认使用HashPartitioner），需要改进这一点，比如重写hash算法。测试的时候想知道partition的数量可以调用rdd.partitions().size()获知。

Task时间分布。关注Spark UI，在Stage的详情页面上，可以看得到shuffle写的总开销，GC时间，当前方法栈，还有task的时间花费。如果你发现task的时间花费分布太散，就是说有的花费时间很长，有的很短，这就说明计算分布不均，需要重新审视数据分片、key的hash、task内部的计算逻辑等等，瓶颈出现在耗时长的task上面。

重用资源。有的资源申请开销巨大，而且往往相当有限，比如建立连接，可以考虑在partition建立的时候就创建好（比如使用mapPartition方法），这样对于每个partition内的每个元素的操作，就只要重用这个连接就好了，不需要重新建立连接。

可供参考的文档：官方调优文档Tuning Spark，Spark配置的官方文档，Spark Programming Guide，JVMGC调优文档，JVM性能调优文档，How-to: Tune Your Apache Spark Jobs part-1 & part-2。

阅读原文

微信扫一扫
关注该公众号