> 思考何为抽象?

Spark之常规性能调优

> Spark性能调优的第一步，就是为**任务分配更多的资源【核心】**，在一定范围内，增加资源的分配与性能的提升是成正比的，实现了最优的资源配置后，在此基础上再考虑进行后面论述的性能调优策略。资源的分配在使用脚本提交Spark任务时进行指定，标准的Spark任务提交脚本如代码清单2-1所示：

调优1：最优资源配置

标准Spark提交脚本

```
/usr/opt/modules/spark/bin/spark-submit \
--class com.atguigu.spark.Analysis \
--num-executors 80 \
--driver-memory 6g \
--executor-memory 6g \
--executor-cores 3 \
/usr/opt/modules/spark/jar/spark.jar \
```

名称	***说明***
--num-executors	配置Executor的数量
--driver-memory	配置Driver内存（影响不大）
--executor-memory	配置每个Executor的内存大小
--executor-cores	配置每个Executor的CPU core数量

调节原则：尽量将任务分配的资源调节到可以使用的资源的最大限度。

对于具体资源的分配，我们分别讨论Spark的两种Cluster运行模式：

- 第一种是Spark Standalone模式，你在提交任务前，一定知道或者可以从运维部门获取到你可以使用的资源情况，在编写submit脚本的时候，就根据可用的资源情况进行资源的分配，比如说集群有15台机器，每台机器为8G内存，2个CPU core，那么就指定15个Executor，每个Executor分配8G内存，2个CPU core。
- 第二种是Spark Yarn模式，由于Yarn使用资源队列进行资源的分配和调度，在表写submit脚本的时候，就根据Spark作业要提交到的资源队列，进行资源的分配，比如资源队列有400G内存，100个CPU core，那么指定50个Executor，每个Executor分配8G内存，2个CPU core。

对表2-1中的各项资源进行了调节后，得到的性能提升如表2-2所示：

名称	*解析*
增加Executor·个数	在资源允许的情况下，增加Executor的个数可以提高执行task的并行度。比如有4个Executor，每个Executor有2个CPU core，那么可以并行执行8个task，如果将Executor的个数增加到8个（资源允许的情况下），那么可以并行执行16个task，此时的并行能力提升了一倍。
增加每个Executor的CPU core个数	在资源允许的情况下，增加每个Executor的Cpu core个数，可以提高执行task的并行度。比如有4个Executor，每个Executor有2个CPU core，那么可以并行执行8个task，如果将每个Executor的CPU core个数增加到4个（资源允许的情况下），那么可以并行执行16个task，此时的并行能力提升了一倍。
增加每个Executor的内存量	在资源允许的情况下，增加每个Executor的内存量以后，对性能的提升有三点：1. 可以缓存更多的数据（即对RDD进行cache），写入磁盘的数据相应减少，甚至可以不写入磁盘，减少了可能的磁盘IO；2. 可以为shuffle操作提供更多内存，即有更多空间来存放reduce端拉取的数据，写入磁盘的数据相应减少，甚至可以不写入磁盘，减少了可能的磁盘IO；3. 可以为task的执行提供更多内存，在task的执行过程中可能创建很多对象，内存较小时会引发频繁的GC，增加内存后，可以避免频繁的GC，提升整体性能。

生产环境Spark submit脚本配置

```
/usr/local/spark/bin/spark-submit \
--class com.atguigu.spark.WordCount \
--num-executors 80 \
--driver-memory 6g \
--executor-memory 6g \
--executor-cores 3 \
--master yarn-cluster \
--queue root.default \
--conf spark.yarn.executor.memoryOverhead=2048 \
--conf spark.core.connection.ack.wait.timeout=300 \
/usr/local/spark/spark.jar
```

参数配置参考值：

--num-executors：50~100

--driver-memory：1G~5G

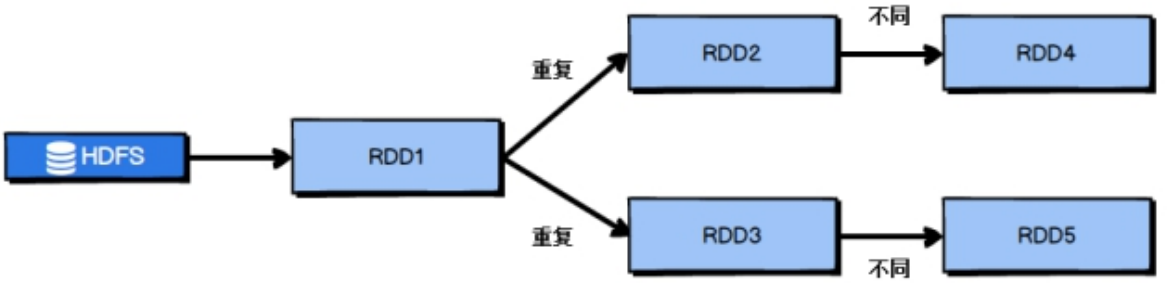
--executor-memory：6G~10G

--executor-cores：3

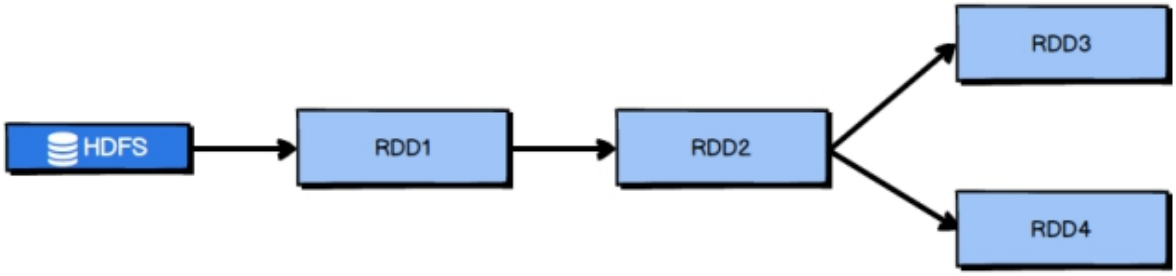
--master：实际生产环境一定使用yarn-cluster

调优2：RDD优化

在对RDD进行算子时，要避免相同的算子和计算逻辑之下对RDD进行重复的计算，如图所示：



对上图中的RDD计算架构进行修改，得到如图所示的优化结果：



RDD持久化

在Spark中，当多次对同一个RDD执行算子操作时，每一次都会对这个RDD以之前的父RDD重新计算一次，这种情况是必须要避免的，对同一个RDD的重复计算是对资源的极大浪费，因此，必须对**多次使用的RDD进行持久化**，通过持久化将公共RDD的数据缓存到内存/磁盘中，之后对于公共RDD的计算都会从内存/磁盘中直接获取RDD数据。

对于RDD的持久化，有两点需要说明：

第一，RDD的持久化是可以进行序列化的，当内存无法将RDD的数据完整的进行存放的时候，可以考虑使用序列化的方式减小数据体积，将数据完整存储在内存中。

第二，如果对于数据的可靠性要求很高，**并且内存充足，可以使用副本机制**，对RDD数据进行持久化。当持久化启用了复本机制时，对于持久化的每个数据单元都存储一个副本，放在其他节点上面，由此实现数据的容错，一旦一个副本数据丢失，不需要重新计算，还可以使用另外一个副本。【有问题都可以私聊我WX：focusbigdata，或者关注我的公众号：FocusBigData，注意大小写】

RDD尽可能早的filter操作

获取到初始RDD后，应该考虑尽早地过滤掉不需要的数据，进而减少对内存的占用，从而提升Spark作业的运行效率。

调优三：并行度调节

> **Spark作业中的并行度指各个stage的task的数量。**

如果并行度设置不合理而导致并行度过低，会导致资源的极大浪费，例如，20个Executor，每个Executor分配3个CPU core，而Spark作业有40个task，这样每个Executor分配到的task个数是2个，这就使得每个Executor有一个CPU core空闲，导致资源的浪费。

理想的并行度设置，应该是让并行度与资源相匹配，简单来说就是在资源允许的前提下，并行度要设置的尽可能大，达到可以充分利用集群资源。合理的设置并行度，可以提升整个Spark作业的性能和运行速度。

Spark官方推荐，**task数量应该设置为Spark作业总CPU core数量的2~3倍。**之所以没有推荐task数量与CPU core总数相等，是因为task的执行时间不同，有的task执行速度快而有的task执行速度慢，如果task数量与CPU core总数相等，那么执行快的task执行完成后，会出现CPU core空闲的情况。如果task数量设置为CPU core总数的2~3倍，那么一个task执行完毕后，CPU core会立刻执行下一个task，降低了资源的浪费，同时提升了Spark作业运行的效率。

代码实现

```
val conf = new SparkConf()
  .set("spark.default.parallelism", "500")
```

调优四：广播大变量

问题场景

默认情况下，task中的算子中如果使用了外部的变量，每个task都会获取一份变量的复本，这就造成了内存的极大消耗。一方面，如果后续对RDD进行持久化，可能就无法将RDD数据存入内存，只能写入磁盘，磁盘IO将会严重消耗性能；另一方面，task在创建对象的时候，也许会发现堆内存无法存放新创建的对象，这就会导致频繁的GC，GC会导致工作线程停止，进而导致Spark暂停工作一段时间，严重影响Spark性能。

假设当前任务配置了20个Executor，指定500个task，有一个20M的变量被所有task共用，此时会在500个task中产生500个副本，耗费集群10G的内存，如果使用了广播变量， 那么每个Executor保存一个副本，一共消耗400M内存，内存消耗减少了5倍。

解决方案

广播变量**在每个Executor保存一个副本，此Executor的所有task共用此广播变量，这让变量产生的副本数量大大减少。**在初始阶段，广播变量只在Driver中有一份副本。task在运行的时候，想要使用广播变量中的数据，此时首先会在自己本地的Executor对应的BlockManager中尝试获取变量，如果本地没有，BlockManager就会从Driver或者其他节点的BlockManager上远程拉取变量的复本，并由本地的BlockManager进行管理；之后此Executor的所有task都会直接从本地的BlockManager中获取变量。

调优五：Kryo序列化

默认情况下，Spark使用Java的序列化机制。Java的序列化机制使用方便，不需要额外的配置，在算子中使用的变量实现Serializable接口即可，但是，Java序列化机制的效率不高，序列化速度慢并且序列化后的数据所占用的空间依然较大。

Kryo序列化机制比Java序列化机制性能提高10倍左右，Spark之所以没有默认使用Kryo作为序列化类库，是因为它***不支持所有对象的序列化***，同时Kryo需要用户在使用前注册需要序列化的类型，不够方便，但从Spark 2.0.0版本开始，简单类型、简单类型数组、字符串类型的Shuffling RDDs 已经默认使用Kryo序列化方式了。

Kryo序列化注册方式的实例代码

```
public class MyKryoRegistrator implements KryoRegistrator
{
 @Override
 public void registerClasses(Kryo kryo)
 {
  kryo.register(StartupReportLogs.class);
 }
}
```

配置Kryo序列化方式的实例代码

```
//创建SparkConf对象
val conf = new SparkConf().setMaster(…).setAppName(…)

//使用Kryo序列化库，如果要使用Java序列化库，需要把该行屏蔽掉
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");

//在Kryo序列化库中注册自定义的类集合，如果要使用Java序列化库，需要把该行屏蔽掉
conf.set("spark.kryo.registrator", "zhutian.com.MyKryoRegistrator");
```

调优六：调节本地化等待时长

Spark作业运行过程中，Driver会对每一个stage的task进行分配。根据Spark的task分配算法，Spark希望task能够运行在它要计算的数据算在的节点（数据本地化思想），这样就可以**避免数据的网络传输。**通常来说，task可能不会被分配到它处理的数据所在的节点，因为这些节点可用的资源可能已经用尽，此时，Spark会等待一段时间，默认3s，如果等待指定时间后仍然无法在指定节点运行，那么会**自动降级**，尝试将task分配到比较差的本地化级别所对应的节点上，比如将task分配到离它要计算的数据比较近的一个节点，然后进行计算，如果当前级别仍然不行，那么继续降级。

当task要处理的数据不在task所在节点上时，会发生数据的传输。task会通过所在节点的BlockManager获取数据，BlockManager发现数据不在本地时，户通过网络传输组件从数据所在节点的BlockManager处获取数据。

网络传输数据的情况是我们不愿意看到的，**大量的网络传输会严重影响性能**，因此，我们希望通过调节本地化等待时长，如果在等待时长这段时间内，目标节点处理完成了一部分task，那么当前的task将有机会得到执行，这样就能够改善Spark作业的整体性能。

Spark的本地化等级如表所示：

名称	***解析***
PROCESS_LOCAL	进程本地化，task和数据在同一个Executor中，性能最好。
NODE_LOCAL	节点本地化，task和数据在同一个节点中，但是task和数据不在同一个Executor中，数据需要在进程间进行传输。
RACK_LOCAL	机架本地化，task和数据在同一个机架的两个节点上，数据需要通过网络在节点之间进行传输。
NO_PREF	对于task来说，从哪里获取都一样，没有好坏之分。
ANY	task和数据可以在集群的任何地方，而且不在一个机架中，性能最差。

在Spark项目开发阶段，可以使用client模式对程序进行测试，此时，可以在本地看到比较全的日志信息，日志信息中有明确的task数据本地化的级别，如果大部分都是**PROCESS_LOCAL，那么就无需进行调节，但是如果发现很多的级别都是NODE_LOCAL、ANY，那么需要对本地化的等待时长进行调节**，通过延长本地化等待时长，看看task的本地化级别有没有提升，并观察Spark作业的运行时间有没有缩短。

注意，过犹不及，不要将本地化等待时长延长地过长，导致因为大量的等待时长，使得Spark作业的运行时间反而增加了。

Spark本地化等待时长的设置如代码

```
val conf = new SparkConf()
  .set("spark.locality.wait", "6")
```