Spark性能优化：shuffle调优

Spark　2016-05-15　6640　2评论　下载为PDF　为什么不允许复制

《Spark性能优化：开发调优篇》
《Spark性能优化：资源调优篇》
《Spark性能优化：数据倾斜调优》
《Spark性能优化：shuffle调优》

文章目录

1 shuffle调优
- 1.1 调优概述
- 1.2 ShuffleManager发展概述
- 1.3 HashShuffleManager运行原理
  - 1.3.1 未经优化的HashShuffleManager
  - 1.3.2 优化后的HashShuffleManager
- 1.4 SortShuffleManager运行原理
  - 1.4.1 普通运行机制
  - 1.4.2 bypass运行机制
- 1.5 shuffle相关参数调优
  - 1.5.1 spark.shuffle.file.buffer
  - 1.5.2 spark.reducer.maxSizeInFlight

1.5.3 spark.shuffle.io.maxRetries
1.5.4 spark.shuffle.io.retryWait
1.5.5 spark.shuffle.memoryFraction
1.5.6 spark.shuffle.manager
1.5.7 spark.shuffle.sort.bypassMergeThreshold
1.5.8 spark.shuffle.consolidateFiles
2 写在最后的话

shuffle调优

调优概述

大多数Spark作业的性能主要就是消耗在了shuffle环节，因为该环节包含了大量的磁盘IO、序列化、网络数据传输等操作。因此，如果要让作业的性能更上一层楼，就有必要对shuffle过程进行调优。但是也必须提醒大家的是，影响一个Spark作业性能的因素，主要还是代码开发、资源参数以及数据倾斜，shuffle调优只能在整个Spark的性能调优中占到一小部分而已。因此大家务必把握住调优的基本原则，千万不要舍本逐末。下面我们就给大家详细讲解shuffle的原理，以及相关参数的说明，同时给出各个参数的调优建议。

ShuffleManager发展概述

在Spark的源码中，负责shuffle过程的执行、计算和处理的组件主要就是ShuffleManager，也即shuffle管理器。而随着Spark的版本的发展，ShuffleManager也在不断迭代，变得越来越先进。

在Spark 1.2以前，默认的shuffle计算引擎是HashShuffleManager。该ShuffleManager而HashShuffleManager有着一个非常严重的弊端，就是会产生大量的中间磁盘文件，进而由大量的磁盘IO操作影响了性能。

因此在Spark 1.2以后的版本中，默认的ShuffleManager改成了SortShuffleManager。SortShuffleManager相较于HashShuffleManager来说，有了一定的改进。主要就在于，每个Task在进行shuffle操作时，虽然也会产生较多的临时磁盘文件，但是最后会将所有的临时文件合并（merge）成一个磁盘文件，因此每个Task就只有一个磁盘文件。在下一个stage的shuffle read task拉取自己的数据时，只要根据索引读取每个磁盘文件中的部分数据即可。

下面我们详细分析一下HashShuffleManager和SortShuffleManager的原理。

HashShuffleManager运行原理

未经优化的HashShuffleManager

下图说明了未经优化的HashShuffleManager的原理。这里我们先明确一个假设前提：每个Executor只有1个CPU core，也就是说，无论这个Executor上分配多少个task线程，同一时间都只能执行一个task线程。

我们先从shuffle write开始说起。shuffle write阶段，主要就是在一个stage结束计算之后，为了下一个stage可以执行shuffle类的算子（比如reduceByKey），而将每个task处理的数据按key进行“分类”。所谓“分类”，就是对相同的key执行hash算法，从而将相同key都写入同一个磁盘文件中，而每一个磁盘文件都只属于下游stage的一个task。在将数据写入磁盘之前，会先将数据写入内存缓冲中，当内存缓冲填满之后，才会溢写到磁盘文件中去。

那么每个执行shuffle write的task，要为下一个stage创建多少个磁盘文件呢？很简单，下一个stage的task有多少个，当前stage的每个task就要创建多少份磁盘文件。比如下一个stage总共有100个task，那么当前stage的每个task都要创建100份磁盘文件。如果当前stage有50个task，总共有10个Executor，每个Executor执行5个Task，那么每个Executor上总共就要创建500个磁盘文件，所有Executor上会创建5000个磁盘文件。由此可见，未经优化的shuffle write操作所产生的磁盘文件的数量是极其惊人的。

接着我们来说说shuffle read。shuffle read，通常就是一个stage刚开始时要做的事情。此时该stage的每一个task就需要将上一个stage的计算结果中的所有相同key，从各个节点上通过网络都拉取到自己所在的节点上，然后进行key的聚合或连接等操作。由于shuffle write的过程中，task给下游stage的每个task都创建了一个磁盘文件，因此shuffle read的过程中，每个task只要从上游stage的所有task所在节点上，拉取属于自己的那一个磁盘文件即可。

shuffle read的拉取过程是一边拉取一边进行聚合的。每个shuffle read task都会有一个自己的buffer缓冲，每次都只能拉取与buffer缓冲相同大小的数据，然后通过内存中的一个Map进行聚合等操作。聚合完一批数据后，再拉取下一批数据，并放到buffer缓冲中进行聚合操作。以此类推，直到最后将所有数据到拉取完，并得到最终的结果。

优化后的HashShuffleManager

下图说明了优化后的HashShuffleManager的原理。这里说的优化，是指我们可以设置一个参数，spark.shuffle.consolidateFiles。该参数默认值为false，将其设置为true即可开启优化机制。通常来说，如果我们使用HashShuffleManager，那么都建议开启这个选项。

开启consolidate机制之后，在shuffle write过程中，task就不是为下游stage的每个task创建一个磁盘文件了。此时会出现shuffleFileGroup的概念，每个shuffleFileGroup会对应一批磁盘文件，磁盘文件的数量与下游stage的task数量是相同的。一个Executor上有多少个CPU core，就可以并行执行多少个task。而第一批并行执行的每个task都会创建一个shuffleFileGroup，并将数据写入对应的磁盘文件内。

当Executor的CPU core执行完一批task，接着执行下一批task时，下一批task就会复用之前已有的shuffleFileGroup，包括其中的磁盘文件。也就是说，此时task会将数据写入已有的磁盘文件中，而不写入新的磁盘文件中。因此，consolidate机制允许不同的task复用同一批磁盘文件，这样就可以有效

多个task的磁盘文件进行一定程度上的合并，从而大幅度减少磁盘文件的数量，进而提升shuffle write的性能。

假设第二个stage有100个task，第一个stage有50个task，总共还是有10个Executor，每个Executor执行5个task。那么原本使用未经优化的HashShuffleManager时，每个Executor会产生500个磁盘文件，所有Executor会产生5000个磁盘文件的。但是此时经过优化之后，每个Executor创建的磁盘文件的数量的计算公式为：CPU core的数量 * 下一个stage的task数量。也就是说，每个Executor此时只会创建100个磁盘文件，所有Executor只会创建1000个磁盘文件。

SortShuffleManager运行原理

SortShuffleManager的运行机制主要分成两种，一种是普通运行机制，另一种是bypass运行机制。当shuffle read task的数量小于等于spark.shuffle.sort.bypassMergeThreshold参数的值时（默认为200），就会启用bypass机制。

普通运行机制

下图说明了普通的SortShuffleManager的原理。在该模式下，数据会先写入一个内存数据结构中，此时根据不同的shuffle算子，可能选用不同的数据结构。如果是reduceByKey这种聚合类的shuffle算子，那么会选用Map数据结构，一边通过Map进行聚合，一边写入内存；如果是join这种普通的shuffle算子，那么会选用Array数据结构，直接写入内存。接着，每写一条数据进入内存数据结构之后，就会判断一下，是否达到了某个临界阈值。如果达到临界阈值的话，那么就会尝试将内存数据结构中的数据溢写到磁盘，然后清空内存数据结构。

在溢写到磁盘文件之前，会先根据key对内存数据结构中已有的数据进行排序。排序过后，会分批将数据写入磁盘文件。默认的batch数量是10000条，也就是说，排序好的数据，会以每批1万条数据的形式分批写入磁盘文件。写入磁盘文件是通过Java的BufferedOutputStream实现的。BufferedOutputStream是Java的缓冲输出流，首先会将数据缓冲在内存中，当内存缓冲满溢之后再一次写入磁盘文件中，这样可以减少磁盘IO次数，提升性能。

一个task将所有数据写入内存数据结构的过程中，会发生多次磁盘溢写操作，也就会产生多个临时文件。最后会将之前所有的临时磁盘文件都进行合并，这就是merge过程，此时会将之前所有临时磁盘文件中的数据读取出来，然后依次写入最终的磁盘文件之中。此外，由于一个task就只对应一个磁盘文件，也就意味着该task为下游stage的task准备的数据都在这一个文件中，因此还会单独写一份索引文件，其中标识了下游各个task的数据在文件中的start offset与end offset。

SortShuffleManager由于有一个磁盘文件merge的过程，因此大大减少了文件数量。比如第一个stage有50个task，总共有10个Executor，每个Executor执行5个task，而第二个stage有100个task。由于每个task最终只有一个磁盘文件，因此此时每个Executor上只有5个磁盘文件，所有Executor只有50个磁盘文件。

bypass运行机制

下图说明了bypass SortShuffleManager的原理。bypass运行机制的触发条件如下：

1、shuffle map task数量小于spark.shuffle.sort.bypassMergeThreshold参数的值。
2、不是聚合类的shuffle算子（比如reduceByKey）。

此时task会为每个下游task都创建一个临时磁盘文件，并将数据按key进行hash然后根据key的hash值，将key写入对应的磁盘文件之中。当然，写入磁盘文件时也是先写入内存缓冲，缓冲写满之后再溢写到磁盘文件的。最后，同样会将所有临时磁盘文件都合并成一个磁盘文件，并创建一个单独的索引文件。

该过程的磁盘写机制其实跟未经优化的HashShuffleManager是一模一样的，因为都要创建数量惊人的磁盘文件，只是在最后会做一个磁盘文件的合并而已。因此少量的最终磁盘文件，也让该机制相对未经优化的HashShuffleManager来说，shuffle read的性能会更好。

而该机制与普通SortShuffleManager运行机制的不同在于：第一，磁盘写机制不同；第二，不会进行排序。也就是说，启用该机制的最大好处在于，shuffle write过程中，不需要进行数据的排序操作，也就节省掉了这部分的性能开销。

shuffle相关参数调优

以下是Shffule过程中的一些主要参数，这里详细讲解了各个参数的功能、默认值以及基于实践经验给出的调优建议。

spark.shuffle.file.buffer

1、默认值：32k
参数说明：该参数用于设置shuffle write task的BufferedOutputStream的buffer缓冲大小。将数据写到磁盘文件之前，会先写入buffer缓冲中，待缓冲写满之后，才会溢写到磁盘。
调优建议：如果作业可用的内存资源较为充足的话，可以适当增加这个参数的大小（比如64k），从而减少shuffle write过程中溢写磁盘文件的次数，也就可以减少磁盘IO次数，进而提升性能。在实践中发现，合理调节该参数，性能会有1%~5%的提升。

spark.reducer.maxSizeInFlight

默认值：48m
参数说明：该参数用于设置shuffle read task的buffer缓冲大小，而这个buffer缓冲决定了每次能够拉取多少数据。
调优建议：如果作业可用的内存资源较为充足的话，可以适当增加这个参数的大小（比如96m），从而减少拉取数据的次数，也就可以减少网络传输的次数，进而提升性能。在实践中发现，合理调节该参数，性能会有1%~5%的提升。

spark.shuffle.io.maxRetries

默认值：3

参数说明：shuffle read task从shuffle write task所在节点拉取属于自己的数据时，如果因为网络异常导致拉取失败，是会自动进行重试的。该参数就代表了可以重试的最大次数。如果在指定次数之内拉取还是没有成功，就可能会导致作业执行失败。

调优建议：对于那些包含了特别耗时的shuffle操作的作业，建议增加重试最大次数（比如60次），以避免由于JVM的full gc或者网络不稳定等因素导致的数据拉取失败。在实践中发现，对于针对超大数据量（数十亿~上百亿）的shuffle过程，调节该参数可以大幅度提升稳定性。

spark.shuffle.io.retryWait

默认值：5s

参数说明：具体解释同上，该参数代表了每次重试拉取数据的等待间隔，默认是5s。

调优建议：建议加大间隔时长（比如60s），以增加shuffle操作的稳定性。

spark.shuffle.memoryFraction

默认值：0.2

参数说明：该参数代表了Executor内存中，分配给shuffle read task进行聚合操作的内存比例，默认是20%。

调优建议：在资源参数调优中讲解过这个参数。如果内存充足，而且很少使用持久化操作，建议调高这个比例，给shuffle read的聚合操作更多内存，以避免由于内存不足导致聚合过程中频繁读写磁盘。在实践中发现，合理调节该参数可以将性能提升10%左右。

spark.shuffle.manager

默认值：sort

参数说明：该参数用于设置ShuffleManager的类型。Spark 1.5以后，有三个可选项：hash、sort和tungsten-sort。HashShuffleManager是Spark 1.2以前的默认选项，但是Spark 1.2以及之后的版本默认都是SortShuffleManager了。tungsten-sort与sort类似，但是使用了tungsten计划中的堆外内存管理机制，内存使用效率更高。

调优建议：由于SortShuffleManager默认会对数据进行排序，因此如果你的业务逻辑中需要该排序机制的话，则使用默认的SortShuffleManager就可以；而如果你的业务逻辑不需要对数据进行排序，那么建议参考后面的几个参数调优，通过bypass机制或优化的HashShuffleManager来避免排序操作，同时提供较好的磁盘读写性能。这里要注意的是，tungsten-sort要慎用，因为之前发现了一些相应的bug。

spark.shuffle.sort.bypassMergeThreshold

默认值：200

参数说明：当ShuffleManager为SortShuffleManager时，如果shuffle read task的数量小于这个阈值（默认是200），则shuffle write过程中不会进行排序操作，而是直接按照未经优化的HashShuffleManager的方式去写数据，但是最后会将每个task产生的所有临时磁盘文件都合并成一个

件，并会创建单独的索引文件。

调优建议：当你使用SortShuffleManager时，如果的确不需要排序操作，那么建议将这个参数调大一些，大于shuffle read task的数量。那么此时就会自动启用bypass机制，map-side就不会进行排序了，减少了排序的性能开销。但是这种方式下，依然会产生大量的磁盘文件，因此shuffle write性能有待提高。

spark.shuffle.consolidateFiles

默认值：false

参数说明：如果使用HashShuffleManager，该参数有效。如果设置为true，那么就会开启consolidate机制，会大幅度合并shuffle write的输出文件，对于shuffle read task数量特别多的情况下，这种方法可以极大地减少磁盘IO开销，提升性能。

调优建议：如果的确不需要SortShuffleManager的排序机制，那么除了使用bypass机制，还可以尝试将spark.shffle.manager参数手动指定为hash，使用HashShuffleManager，同时开启consolidate机制。在实践中尝试过，发现其性能比开启了bypass机制的SortShuffleManager要高出10%~30%。

写在最后的话

本文分别讲解了开发过程中的优化原则、运行前的资源参数设置调优、运行中的数据倾斜的解决方案、为了精益求精的shuffle调优。希望大家能够在阅读本文之后，记住这些性能调优的原则以及方案，在Spark作业开发、测试以及运行的过程中多尝试，只有这样，我们才能开发出更优的Spark作业，不断提升其性能。

本文转载自：http://tech.meituan.com/spark-tuning-pro.html

Spark

« Spark性能优化：数据倾斜调优

Zookeeper四字命令 »

[电子书]Machine Learning with Spark Second Edition PDF下载	Spark sql解析异常 java.lang.StackOverflowErr 处理	解决Spark shell模式下初始化Job出现的异常	[电子书]Mastering Spark for Data Science PDF下载
[电子书]Machine Learning with Spark	Spark sql解析异常 java.lang.StackOverflowEr	解决Spark shell模式下初始化Job出现的异常	[电子书]Mastering Spark for Data Science PDF下

Apache Spark常见的三大误解	Apache Hivemall:可运行在Hive, Spark 和 Pig 上的可扩展机器学习库	Spark Structured Streaming入门编程指南	Spark 2.1.0与CarbonData 1.0.0集群模式部署及使用入门指南
Apache Spark常见的三大误解	Apache Hivemall:可运行在Hive, Spark 和 Pig 上	Spark Structured Streaming入门编程指南	Spark 2.1.0与CarbonData 1.0.0集群模

下面文章您可能感兴趣

- Apache Flink 1.1.0和1.1.1发布，支持SQL
- Spark shuffle：hash和sort性能对比
- Newspaper: 新闻文章元数据抽取的开源Python库
- MapReduce作业的map task和reduce task调度参数
- Spark将计算结果写入到Mysql中
- 常用Hadoop生态圈软件分布式安装文章汇集
- Spark 1.1.0正式发布
- 四种解决Spark数据倾斜（Data Skew）的方法
- Hadoop 2.2.0安装和配置lzo
- 2013年各大IT公司研发类笔试题
- 使用Apache Spark将数据写入ElasticSearch
- 寻找n个整数中前最小的k个元素
- Apache Spark 1.5新特性介绍
- Web数据挖掘
- Kafka管理工具介绍
- 为WordPress的suffusion主题添加文章浏览次数
- Apache Arrow：内存列式的数据结构标准
- Elasticsearch配置参数介绍
- WordPress的使用小技巧

发表我的评论

写点什么吧...

表情 本博客评论系统带有自动识别垃圾评论功能，请写一些有意义的评论，谢谢！ 提交评论

(2)个小伙伴在吐槽

"2、不是聚合类的算子(比如reduceByKey)"这里是笔误吧，应该是"比如join"?

Aaron 2016-07-27 17:33 回复

优化后的hashshuffleManager，每个Executor创建的磁盘文件的数量的计算公式为：CPU core的数量 * 下一个stage的task数量。这话感觉不对劲啊，那CPU core越多，磁盘文件岂不是越多吗

zm 2016-05-20 10:23 回复