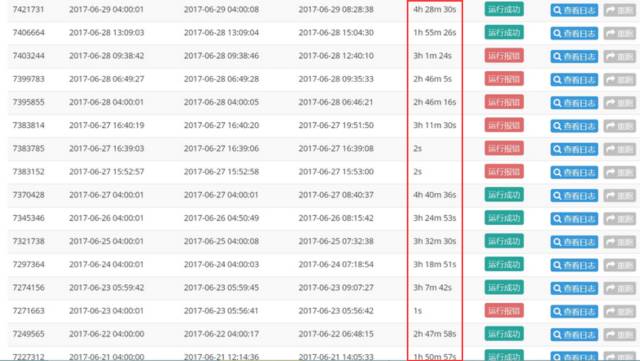
手把手教你 Spark 性能调优

**0、背景**

上周四接到反馈，集群部分 spark 任务执行很慢，且经常出错，参数改来改去怎么都无法优化其性能和解决频繁随机报错的问题。

看了下任务的历史运行情况，平均时间 3h 左右，而且极其不稳定，偶尔还会报错：



**1、优化思路**

任务的运行时间跟什么有关？

**（1）数据源大小差异**

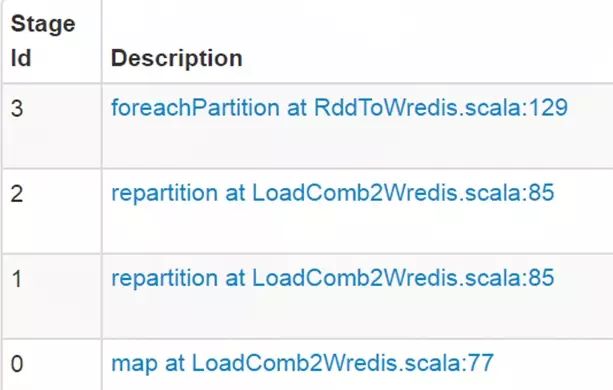
在有限的计算下，job的运行时长和数据量大小正相关，在本例中，数据量大小基本稳定，可以排除是日志量级波动导致的问题：



**（2）代码本身逻辑缺陷**

比如代码里重复创建、初始化变量、环境、RDD资源等，随意持久化数据等，大量使用 shuffle 算子等，比如reduceByKey、join等算子。

在这份100行的代码里，一共有 3 次 shuffle 操作，任务被 spark driver 切分成了 4 个 stage 串行执行，代码位置如下：



咱们需要做的就是从算法和业务角度尽可能减少 shuffle 和 stage，提升并行计算性能，这块是个大的话题，本次不展开详述。

**（3）参数设置不合理**

这块技巧相对通用，咱们来看看之前的核心参数设置：

num-executors=10 || 20 ，executor-cores=1 || 2， executor-memory= 10 || 20，driver-memory=20，spark.default.parallelism=64

假设咱们的 spark 队列资源情况如下：

memory=1T，cores=400

参数怎么设置在这里就有些技巧了，首先得明白 spark 资源的分配和使用原理：

在默认的非动态资源分配场景下， spark 是预申请资源，任务还没起跑就独占资源，一直到整个 job 所有 task 结束，比如你跳板机起了一个 spark-shell 一直没退出，也没执行任务，那也会一直占有所有申请的资源。（如果设置了 num-executors，动态资源分配会失效）

注意上面这句话，spark 的资源使用分配方式和 mapreduce/hive 是有很大差别的，如果不理解这个问题就会在参数设置上引发其它问题。

比如 executor-cores 设多少合适？少了任务并行度不行，多了会把整个队列资源独占耗光，其他同学的任务都无法执行，比如上面那个任务，在 num-executors=20 executor-cores=1 executor-memory= 10 的情况下，会独占20个cores，200G内存，一直持续3个小时。

那针对本case中的任务，结合咱们现有的资源，如何设置这 5 个核心参数呢？

1） executor\_cores\*num\_executors 不宜太小或太大！一般不超过总队列 cores 的 25%，比如队列总 cores 400，最大不要超过100，最小不建议低于 40，除非日志量很小。

2） executor\_cores 不宜为1！否则 work 进程中线程数过少，一般 2~4 为宜。

3） executor\_memory 一般 6~10g 为宜，最大不超过 20G，否则会导致 GC 代价过高，或资源浪费严重。

4） spark\_parallelism 一般为 executor\_cores\*num\_executors 的 1~4 倍，系统默认值 64，不设置的话会导致 task 很多的时候被分批串行执行，或大量 cores 空闲，资源浪费严重。

5） driver-memory 早前有同学设置 20G，其实 driver 不做任何计算和存储，只是下发任务与yarn资源管理器和task交互，除非你是 spark-shell，否则一般 1-2g 就够了。

Spark Memory Manager：

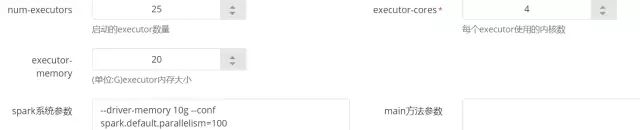
6）spark.shuffle.memoryFraction(默认 0.2) ，也叫 ExecutionMemory。这片内存区域是为了解决 shuffles,joins, sorts and aggregations 过程中为了避免频繁IO需要的buffer。如果你的程序有大量这类操作可以适当调高。

7）spark.storage.memoryFraction(默认0.6)，也叫 StorageMemory。这片内存区域是为了解决 block cache(就是你显示调用dd.cache, rdd.persist等方法), 还有就是broadcasts,以及task results的存储。可以通过参数，如果你大量调用了持久化操作或广播变量，那可以适当调高它。

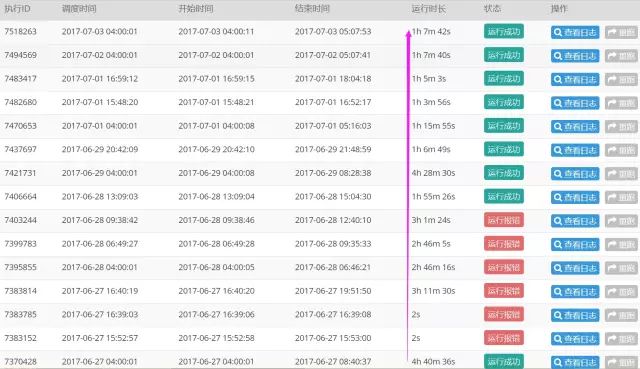
8）OtherMemory，给系统预留的，因为程序本身运行也是需要内存的， (默认为0.2)。Other memory在1.6也做了调整，保证至少有300m可用。你也可以手动设置 spark.testing.reservedMemory . 然后把实际可用内存减去这个reservedMemory得到 usableMemory。 ExecutionMemory 和 StorageMemory 会共享usableMemory \* 0.75的内存。0.75可以通过 新参数 spark.memory.fraction 设置。目前spark.memory.storageFraction 默认值是0.5,所以ExecutionMemory，StorageMemory默认情况是均分上面提到的可用内存的。

例如，如果需要加载大的字典文件，可以增大executor中 StorageMemory 的大小，这样就可以避免全局字典换入换出，减少GC，在这种情况下，我们相当于用内存资源来换取了执行效率。

最终优化后的参数如下：



效果如下：



**（4）通过执行日志分析性能瓶颈**

最后的任务还需要一个小时，那这一个小时究竟耗在哪了？按我的经验和理解，一般单天的数据如果不是太大，不涉及复杂迭代计算，不应该超过半小时才对。

由于集群的 Spark History Server 还没安装调试好，没法通过 spark web UI 查看历史任务的可视化执行细节，所以我写了个小脚本分析了下前后具体的计算耗时信息，可以一目了然的看到是哪个 stage 的问题，有针对性的优化。



可以看到优化后的瓶颈主要在最后写 redis 的阶段，要把 60G 的数据，25亿条结果写入 redis，这对 redis 来说是个挑战，这个就只能从写入数据量和 kv 数据库选型两个角度来优化了。



**（5）其它优化角度**

当然，优化和高性能是个很泛、很有挑战的话题，除了前面提到的代码、参数层面，还有怎样防止或减少数据倾斜等，这都需要针对具体的场景和日志来分析，此处也不展开。

**2、spark 初学者的一些误区**

对于初学者来说 spark 貌似无所不能而且高性能，甚至在某些博客、技术人眼里 spark 取代 mapreduce、hive、storm 分分钟的事情，是大数据批处理、机器学习、实时处理等领域的银弹。但事实确实如此吗？

从上面这个 case 可以看到，会用 spark、会调 API 和能用好 spark，用的恰到好处是两码事，这要求咱们不仅了解其原理，还要了解业务场景，将合适的技术方案、工具和合适的业务场景结合——这世上本就不存在什么银弹。。。

说道 spark 的性能，想要它快，就得充分利用好系统资源，尤其是内存和CPU：核心思想就是能用内存 cache 就别 spill 落磁盘，CPU 能并行就别串行，数据能 local 就别 shuffle。

**Refer：**

[1] spark 内存管理

https://zhangyi.gitbooks.io/spark-in-action/content/chapter2/memory\_management.html

[2] Spark Memory解析

https://github.com/ColZer/DigAndBuried/blob/master/spark/spark-memory-manager.md

[3] Spark1.6内存管理模型设计稿-翻译

http://ju.outofmemory.cn/entry/240714

[4] Spark内存管理

http://blog.csdn.net/vegetable\_bird\_001/article/details/51862422

[5] Apache Spark 内存管理详解

https://www.ibm.com/developerworks/cn/analytics/library/ba-cn-apache-spark-memory-management/index.html