## Spark Shuffle源码分析

刘凯

**免责声明：**

该文产生基于网络和对spark2.2版本的源代码的阅读整理而成。甚至图片均来自于网络。因为该文章不会独立发表，因此就对引用的图片以及内容不做注解。请原作者谅解。

**前言：**

对于分布式的离线计算不可避免的要涉及到shuffle，从微观层面看，对于shuffle细节的理解决定了你对计算引擎的掌握程度。本文试图从源码角度来描述spark shuffle过程的全部细节。

本文以spark2.0的代码为基础，主要讲述sort based shuffle机制的整个shuffle代码细节。

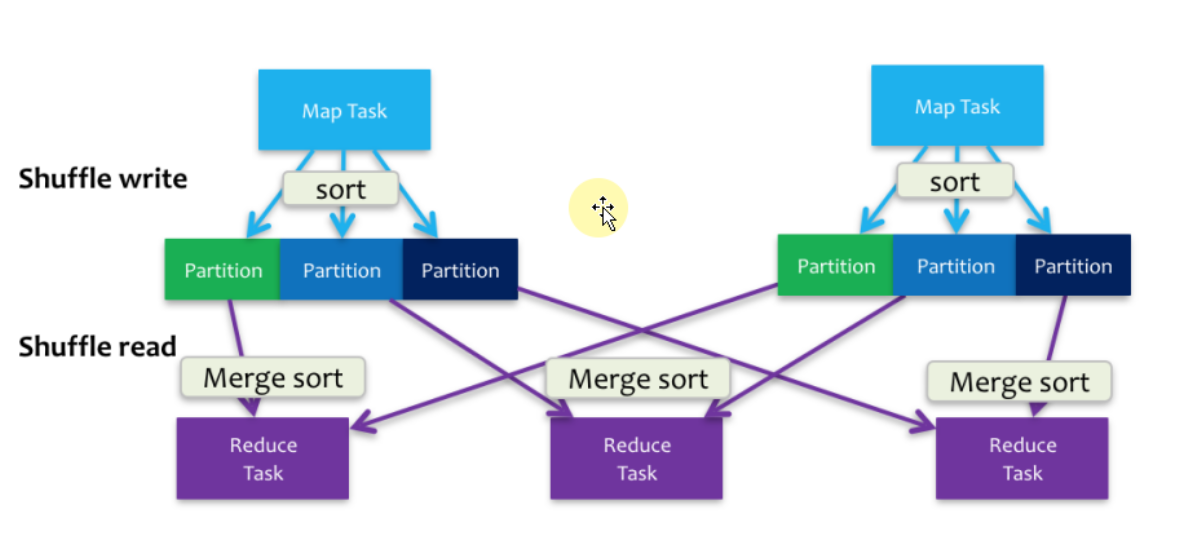
在spark早期版本spark会用到Hash Based Shuffle的机制，这个机制在数据量小的情况下，有优势，但是当数据量比较大的时候，在shuffle write阶段会产生很多的小文件（M\*R个小文件）这样对文件系统的压力比较大（产生大量的文件句柄）同时会对IO吞吐量有比较大的影响。当然引入Consolidation机制可以降低生产的文件数量。但是依然不是很理想的解决方案。因此在后面版本就引入了sort based shuffle（与MapReduce的shuffle机制类似）。包括spark1.6，spark2.0这些spark的主流版本，主要采用的都是sort based shuffle机制。

sort based shuffle机制与Hash Based Shuffle机制相比，sort based shuffle在shuffle write阶段会按照Partition id以及key对记录进行排序。同时将全部结果写到一个数据文件中，同时生成一个索引文件。因此对文件系统的压力比较小。同时大大的提高io的吞吐量。

目前最新版本的spark2.0以及以上的版本，主要存在两种shuffle机制，sort based shuffle和Tungsten-Sort Based Shuffle。Hash Based Shuffle机制在spark2.0之后就已经被spark团队抛弃了。因此，我们在讲述代码的时候，会以sort based shuffle机制为主要脉络开展此文。同时后面会讲述Tungsten-Sort Based Shuffle的引入对shuffle性能提升的帮助。

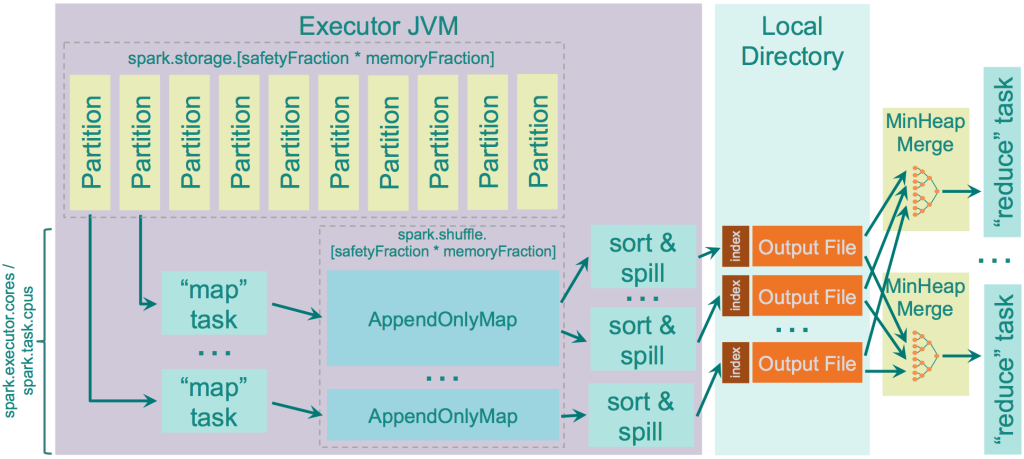
### Spark Sorted-Based Shuffle介绍

Sorted-Based Shuffle 不会为每个Reducer 中的Task 生产一个单独的文件，相反Sorted-Based Shuffle 会把Mapper 中每个ShuffleMapTask 所有的输出数据Data 只写到一个文件中，因为每个ShuffleMapTask 中的数据会被会按照Partition id以及key对记录进行排序，所以Sort-based Shuffle 使用了index 文件存储具体ShuffleMapTask 输出数据在同一个Data 文件中是如何分类的信息。所以说基于 Sort-based Shuffle 会在 Mapper 中的每一个 ShuffleMapTask 中产生两个文件 (并发度的个数 x 2)。并行度=executor-num\*executor-cores。



**文件数计算举例说明**：

假设现在 Mapper 端有 1000 个数据分片，Reducer 端也有 1000 个数据分片，它的并发度是100，使用 Sorted-Based Shuffle 会产生多少个 Mapper端的小文件，答案是 100 x 2 = 200 个。 MapTask 会独自运行，每个 MapTask 在运行的时候写2个文件，运行成功后就不需要这个 MapTask 的文件句柄，无论是文件本身的句柄还是索引的句柄都不需要。



Sorted-Based Shuffle 主要是在Mapper阶段，这个跟Reducer端没有任何关系，在Mapper阶段它要进行排序，你可以认为是二次排序，它的原理是有2个Key进行排序，第一个是 PartitionId进行排序，第二个是就是本身数据的Key进行排序。看下图，它会把 PartitionId 分成3个，分别是索引为 0、1、2，这个在Mapper端进行排序的过程其实是让Reducer去抓取数据的时候变得更高效，比如说第一个Reducer，它会到Mappr端的索引为 0 的数据分片中抓取数据。

具体而言，Reducer 首先找 Driver 去获取父 Stage 中每个 ShuffleMapTask 输出的位置信息，跟据位置信息获取 Index 文件，解析 Index 文件，从解析的 Index 文件中获取 Data 文件中属于自己的那部份内容。

一个Mapper任务除了有一个数据文件以外，它也会有一个索引文件，Map Task 把数据写到文件磁盘是顺序根据自身的Key写进去的，也同时按照 Partition写进去的，因为它是顺序写数据，记录每个 Partition 的大小。

**Sort-Based Shuffle 的弱点：**

* 如果 Mapper 中 Task 的数量过大，依旧会产生很多小文件，此时在 Shuffle 传数据的过程中到 Reducer 端，Reducer 会需要同时大量的记录来进行反序例化，导致大量内存消耗和GC 的巨大负担，造成系统缓慢甚至崩溃。
* 强制了在 Mapper 端必顺要排序，这里的前提是本身数据根本不需要排序的话；
* 如果需要在分片内也进行排序的话，此时需要进行 Mapper 端和 Reducer 端的两次排序！
* 它要基于记录本身进行排序，这就是 Sort-Based Shuffle 最致命的性能消耗；

**Shuffle 中六大令人费解的问题**

**第一大问题：**什么时候进行 Shuffle 的 fetch 操作？Shuffle 是在一边进行 Mapper 端 map 操作的同时，一边进行 Reducer 端的 shuffle 和 reduce 操作吗？

错误的观点是：Spark 是一边 Mapper 一边 Shuffle 的，而 Hadoop MapReduce 是先完成 Mapper 然后才进行 Reducer 的 Shuffle。正确的观点是 **Spark一定是先完成 Mapper 端所有的 Tasks，才会进行 Reducer 端的 Shuffle 过程**。这是因为 Spark Job 是按照 Stage 线性执行的，前面的 Stage 必须执行完毕，才能够执行后面 Reducer 端的 Shuffle 过程。

更准确来说 **Spark Shuffle 的过程是边拉取数据边进行 Aggregrate 操作的**，其实与 Hadoop MapReduce 相比其优势确实是在速度上，但是也会导致一些算法不好实现，例如求平均值等，为什么呢？因为边拉取数据边进行 Aggregrate 这个过程中，后面的Stage依赖于前面的Stage，Spark 是以 Stage 为单位进行计算的，如果里面的任务没有计算完，后面你怎么计算呢。**但如果你是求和的话，它就会计算的特别快**；

Hadoop MapReduce 是把数据拉过来之后，然后进行计算，如果用 MapReduce 求平均值的话，它的算法就会很好实现。

**第二大问题：**Shuffle fetch 过来的数据到底放在了那里？

Spark 这种很灵活地使用并行度以及倾向于优先使用内存的计算模型，如果不正常地使用这些特征的话会很容易导致 Spark 的应用程序出现 OOM 的情况，Spark 在不同的版本 fetch 过来的数据放在哪里是有不同的答案。抓过来的数据首先会放在 Reducer 端的内存缓存区中，Spark曾经有版本要求只能放在内存缓存中，其数据结构类似于 HashMap (AppendOnlyMap)，显然这个设计特别消耗内存和极易出现OOM，同时这也极大的限制了 Spark 集群的规模，现在的实现都是内存 + 磁盘的方式 (数据结构类使用了 ExternalAppendOnlyMap)，当然也可以通过调以下参数来设置只能使用内存。

1

spark.shuffle.spill=false

如果设置了这个运行模式，在生产环境下建义对内存的数据作2份备份，因为在默认情况下内存数据只有1份，它不像HDFS那样，天然有3份备份。使用 ExternalAppendOnlyMap 的方式时，如果内存占用率达到一定的临界值后会首先尝试在内存中扩大 ExternalAppendOnlyMap (内部有实现算法)，如果不能扩容的话才会 spill 到磁盘。

第三大问题：Shuffle 的数据在 Mapper 端如何存储，在 Reducer 端如何知道数据具体在那里的？在Spark的实现上每一个Stage (里面是 ShuffleMapTask) 中的 Task 在 Stage 的最后一个 RDD 上一定会注册给 Driver 上的 MapOutputTrackerMaster，Mapper 通过和 MapOutputTrackerMaster 来汇报 ShuffleMapTask 具体输出数据的位置 (具体的输出文件及内容是和 Reducer 有关的)，Reducer 是向 Driver 中的 MapOutputTrackerMaster 请求数据的元数据信息，然后和 Mapper 所在的 Executor 进行通信。

第四大问题：竟竟从 HashShuffle 的角度来讲，我们在 Shuffle 的时候到底可以产生多少 Mapper 端的中间文件？

这里有一个很重要的调优参数 (可以在 TaskSchedulerImpl.scala 中找到此参数)，该参数决定了 Spark 在运行时每个 Task 所需要的 Core 的个数，默认情况是1个，现在假设 spark.task.cpus=T。

问题一、例如说有M个Mapper、R个Reducer 和 C个Core，那么 HashShuffle 可以产生多少个 Mapper 的中间文件？

HashShuffle 会产生 C x R 的小文件。

Consolidated HashShuffle 有可能产生 C x R 个小文件。因为设置了 spark.task.cpus 的参数，那么真实的答案是 (C / T) x R 个小文件。

问题二、例如在生产环境下有 E 个 Executors (例如100个)，每个 Executor上有 C 个Cores (例如10个)，同时也有 R 个Reducer，那么 HashShuffle 可以产生多少个 Mapper 的中间文件？

HashShuffle 会产生实际 Task 的个数 x R 个的小文件。

Consolidated HashShuffle 会产生 (E x (C / T)) x R 个的小文件。

第五大问题：Spark中Sorted-Based Shuffle 数据结果默认是排序的吗？Sorted-Based Shuffle 采用了什么的排序算法？这个排序算法的好处是什么？

Spark Sorted-Based Shuffle 在 Mapper 端是排序的，包括 partition 的排序和每个 partition 内部元素的排序！但在 Reducer 端是没有进行排序，所以 Job 的结果默认不是排序的。Sorted-Based Shuffle 采用了 Tim-Sort 排序算法，好处是可以极为高效的使用 Mapper 端的排序成果全局排序。

第六大问题：Spark Tungsten-Sorted Shuffle 在 Mapper 中会对内部元素进行排序吗？Tungsten-Sorted Shuffle不适用于什么情况？说出具体的原因。

Tungsten-Sorted Shuffle 在 Mapper 中不会对内部元素进行排序 (它只会对Partition进行排序)，原因是它自己管理的二进制序列化后的数据，问题来啦：数据是进入 Buffer 时或者是进入磁盘的时才进行排序呢？答案是数据的排序是发生在 Buffer 要满了 spill 到磁盘时才进行排序的。所以 Tungsten-Sorted Shuffle 它对内部不会进行排序

Tungsten-Sorted Shuffle 什么时候会退化成为 Sorted-Based Shuffle？它是在程序有 Aggregrate 操作的时候；或者是 Mapper 端输出的 partition 大于 16777216；或者是一条 Record 大于128M的时候，原因也是因为它自己管理的二进制序列化后的数据以及数组指针管理范围。

Shuffle源码结构图：

