1. 介绍

Shuffle描述着数据从map task输出到reduce task输入的这段过程。shuffle是连接Map和Reduce之间的桥梁，Map的输出要用到Reduce中必须经过shuffle这个环节，shuffle的性能高低直接影响了整个程序的性能和吞吐量。因为在分布式情况下，reduce task需要跨节点去拉取其它节点上的map task结果。这一过程将会产生网络资源消耗和内存，磁盘IO的消耗。通常shuffle分为两部分：Map阶段的数据准备和Reduce阶段的数据拷贝处理。一般将在map端的Shuffle称之为Shuffle Write，在Reduce端的Shuffle称之为Shuffle Read.

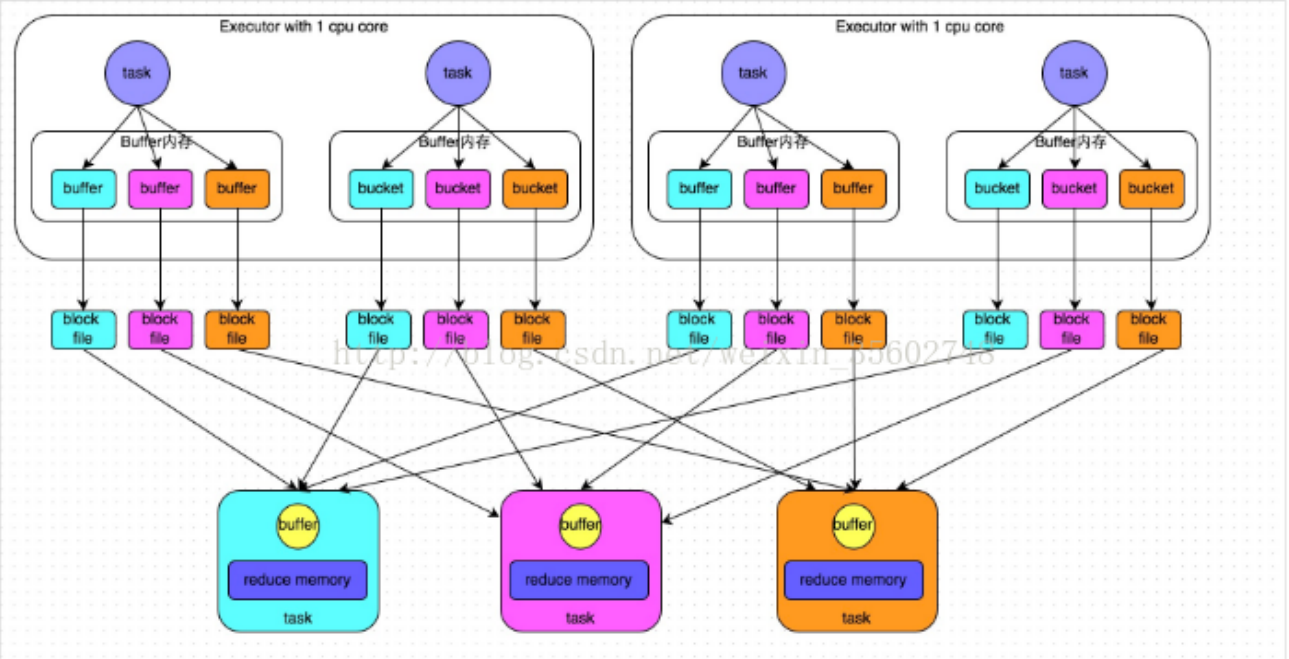
在Spark的中，负责shuffle过程的执行、计算和处理的组件主要就是ShuffleManager，也即shuffle管理器。ShuffleManager随着Spark的发展有两种实现的方式，分别为HashShuffleManager和SortShuffleManager，因此spark的Shuffle有Hash Shuffle和Sort Shuffle两种(前者在2.0版本后已遗弃)

spark shuffle 演进的历史

- Spark 0.8及以前 Hash Based Shuffle
- Spark 0.8.1 为Hash Based Shuffle引入File Consolidation机制
- Spark 0.9 引入ExternalAppendOnlyMap
- Spark 1.1 引入Sort Based Shuffle，但默认仍为Hash Based Shuffle
- Spark 1.2 默认的Shuffle方式改为Sort Based Shuffle
- Spark 1.4 引入Tungsten-Sort Based Shuffle
- Spark 1.6 Tungsten-sort并入Sort Based Shuffle
- Spark 2.0 Hash Based Shuffle退出历史舞台

所以现在就只有 sortshuffle

2. HashShuffleManager



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。

图中有3个 Reducer，从Task 开始那边各自把自己进行 Hash 计算(分区器：hash/numreduce取模)，分类出3个不同的类别，每个 Task 都分成3种类别的数据，想把不同的数据汇聚然后计算出最终的结果，所以Reducer 会在每个 Task 中把属于自己类别的数据收集过来，汇聚成一个同类别的大集合，每1个 Task 输出3份本地文件，这里有4个 Mapper Tasks，所以总共输出了4个 Tasks x 3个分类文件 = 12个本地小文件。

1. 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，groupByKey)，而将每个task处理的数据按key进行“分区”。所谓“分区”，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reduce端的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。

2.shuffle read阶段

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Reduce端的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。

注意：

1).buffer起到的是缓存作用，缓存能够加速写磁盘，提高计算的效率,buffer的默认大小32k。

分区器：根据hash/numRedcue取模决定数据由几个Reduce处理，也决定了写入几个buffer中

block file：磁盘小文件，从图中我们可以知道磁盘小文件的个数计算公式：

block file=M*R

2).M为map task的数量，R为Reduce的数量，一般Reduce的数量等于buffer的数量，都是由分区器决定的

Hash shuffle普通机制的问题

1).Shuffle前在磁盘上会产生海量的小文件，建立通信和拉取数据的次数变多,此时会产生大量耗时低效的 IO 操作(因为产生过多的小文件)

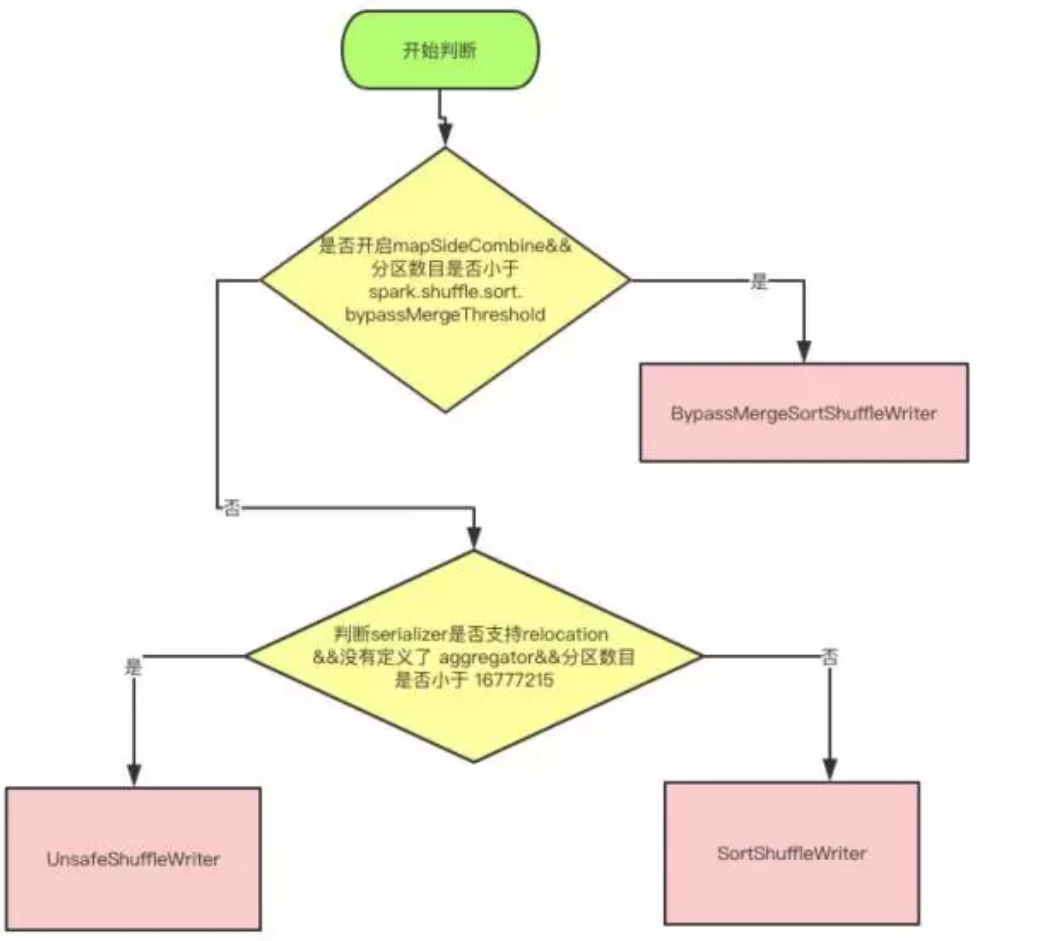
2).可能导致OOM，大量耗时低效的 IO 操作 ，导致写磁盘时的对象过多，读磁盘时候的对象也过多，这些对象存储在堆内存中，会导致堆内存不足，相应会导致频繁的GC，GC会导致OOM。由于内存中需要保存海量文件操作句柄和临时信息，如果数据处理的规模比较庞大的话，内存不可承受，会出现 OOM 等问题。

所以有了 合并机制, 就是复用buffer，开启合并机制的配置是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。

> 更多见 https://www.cnblogs.com/itboys/p/9226479.html

3. SortShuffleManager

现在2.1 分为三种writer， 分为 BypassMergeSortShuffleWriter， SortShuffleWriter 和 UnsafeShuffleWriter



上面是使用哪种 writer 的判断依据， 是否开启 mapSideCombine 这个判断，是因为有些算子会在 map 端先进行一次 combine， 减少传输数据。 因为 BypassMergeSortShuffleWriter 会临时输出Reducer个（分区数目）小文件，所以分区数必须要小于一个阀值，默认是小于200。

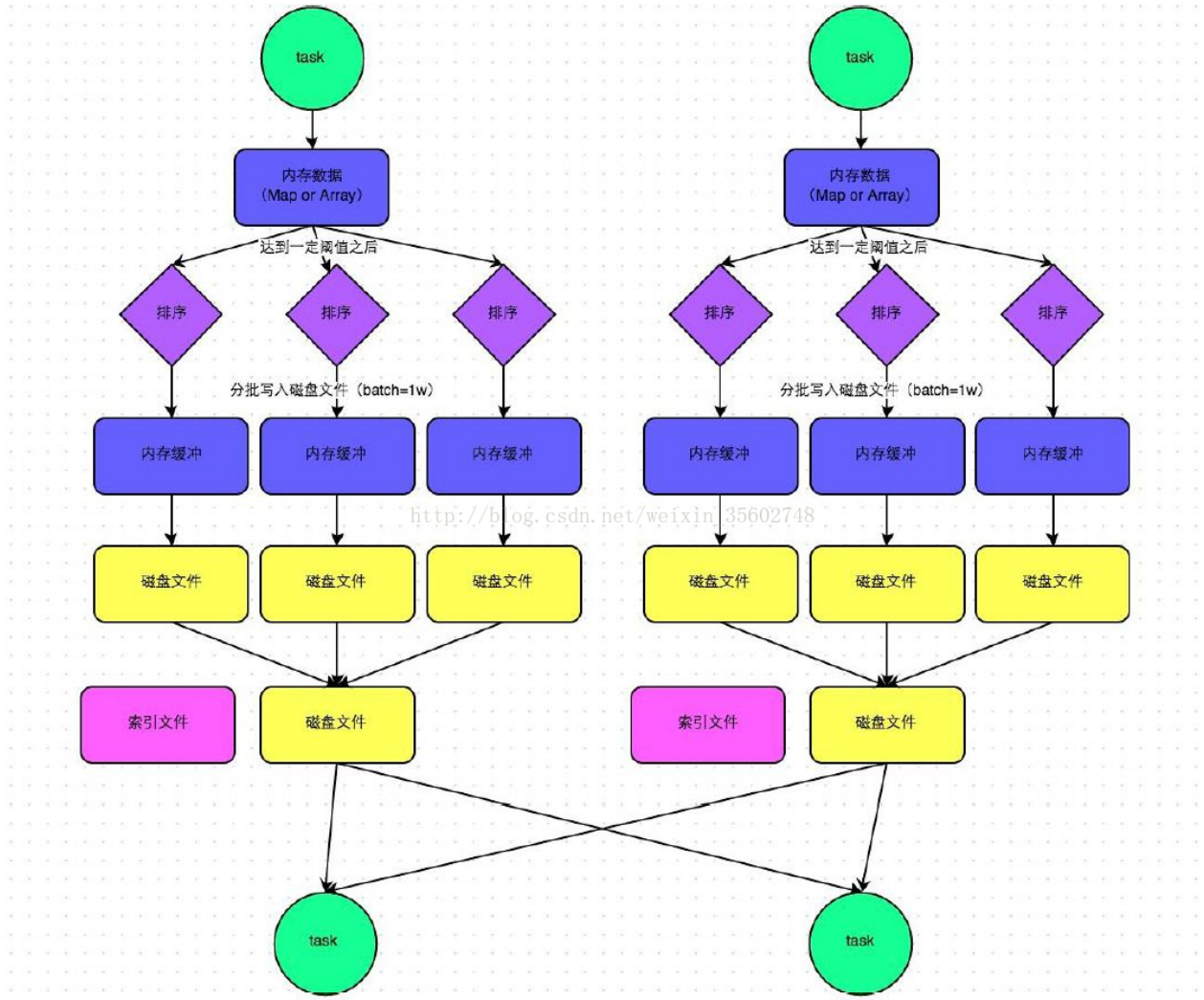
UnsafeShuffleWriter需要Serializer支持relocation，Serializer支持relocation：原始数据首先被序列化处理，并且再也不需要反序列，在其对应的元数据被排序后，需要Serializer支持relocation，在指定位置读取对应数据。

3.1 BypassMergeSortShuffleWriter

BypassMergeSortShuffleWriter和Hash Shuffle中的HashShuffleWriter实现基本一致， 唯一的区别在于，map端的多个输出文件会被汇总为一个文件。 所有分区的数据会合并为同一个文件，会生成一个索引文件，是为了索引到每个分区的起始地址，可以随机 access 某个partition的所有数据。

但是需要注意的是，这种方式不宜有太多分区，因为过程中会并发打开所有分区对应的临时文件，会对文件系统造成很大的压力。

3.2 SortShuffleWriter



写入内存数据结构

该图说明了普通的SortShuffleManager的原理。在该模式下，数据会先写入一个内存数据结构中(默认5M)，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。

注意：

shuffle中的定时器：定时器会检查内存数据结构的大小，如果内存数据结构空间不够，那么会申请额外的内存，申请的大小满足如下公式：

applyMemory=nowMenory*2-oldMemory

申请的内存=当前的内存情况*2-上一次的内嵌情况

意思就是说内存数据结构的大小的动态变化，如果存储的数据超出内存数据结构的大小，将申请内存数据结构存储的数据*2-内存数据结构的设定值的内存大小空间。申请到了，内存数据结构的大小变大，内存不够，申请不到，则发生溢写

- 排序

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。(此时排序都是内存小,数据多的情况,出现了外部空间排序,这中间会出现临时文件的出现)

- 溢写

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。

- merge

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Reduce端的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。

注意:

1)block file= 2M

一个map task会产生一个索引文件和一个数据大文件

2) m*r>2m(r>2)：SortShuffle会使得磁盘小文件的个数再次的减少

3.3 UnsafeShuffleWriter

UnsafeShuffleWriter 里面维护着一个 ShuffleExternalSorter， 用来做外部排序， 外部排序就是要先部分排序数据并把数据输出到磁盘，然后最后再进行merge 全局排序， 既然这里也是外部排序，跟 SortShuffleWriter 有什么区别呢， 这里只根据 record 的 partition id 先在内存 ShuffleInMemorySorter 中进行排序， 排好序的数据经过序列化压缩输出到换一个临时文件的一段，并且记录每个分区段的seek位置，方便后续可以单独读取每个分区的数据，读取流经过解压反序列化，就可以正常读取了。

整个过程就是不断地在 ShuffleInMemorySorter 插入数据，如果没有内存就申请内存，如果申请不到内存就 spill 到文件中，最终合并成一个 依据 partition id 全局有序 的大文件。

SortShuffleWriter 和 UnsafeShuffleWriter 对比

区别	UnsafeShuffleWriter	SortShuffleWriter
排序方式	最终只是 partition 级别的排序	先 partition 排序，相同分区 key有序
aggregation	没有反序列化，没有aggregation	支持 aggregation

使用 UnsafeShuffleWriter 的条件

- 没有指定 aggregation 或者key排序， 因为 key 没有编码到排序指针中，所以只有 partition 级别的排序
- 原始数据首先被序列化处理，并且再也不需要反序列，在其对应的元数据被排序后，需要Serializer支持relocation，在指定位置读取对应数据。 KryoSerializer 和 spark sql 自定义的序列化器 支持这个特性。

- 分区数目必须小于 16777216，因为 partition number 使用24bit 表示的。
- 因为每个分区使用 27 位来表示 record offset， 所以一个 record 不能大于这个值。

> https://www.cnblogs.com/itboys/p/9201750.html
> https://www.cnblogs.com/itboys/p/9226479.html