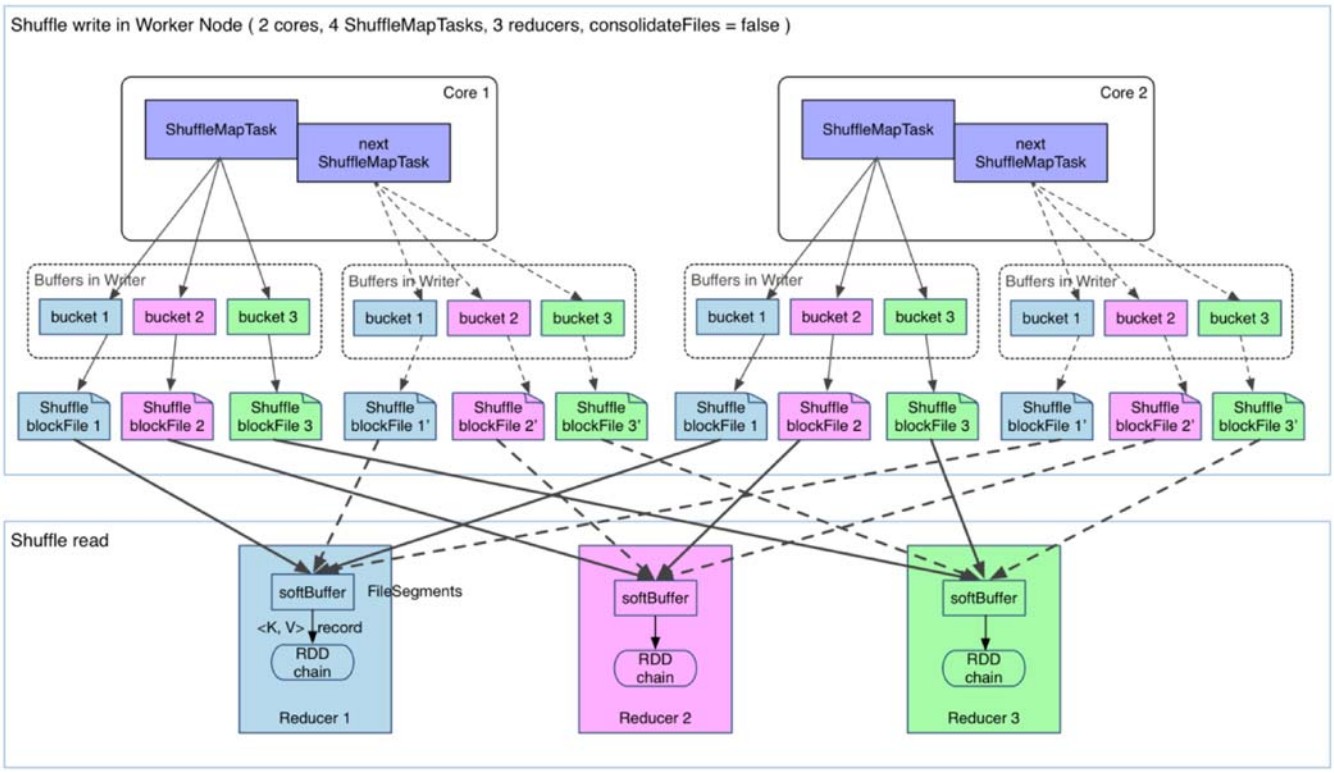
1 Hash based shuffle

Spark Shuffle机制演进——Hash Shuffle



该图描述了最简单的Spark 0.X版本的Spark Shuffle过程。 与Hadoop Map Reduce的区别在于输出文件个数的变化。

每个ShuffleMapTask产生与Ruducer个数相同的Shuffle blockFile文件，图中有3个reducer，那么每个ShuffleMapTask就产生3个Shuffle blockFile文件，4个ShuffleMapTask，那么一共产生12个Shuffle blockFile文件。

在内存中每个Shuffle blockFile文件都会存在一个句柄从而消耗一定内存，又因为物理内存的限制，就不能有很多并发，这样就限制了Spark集群的规模。

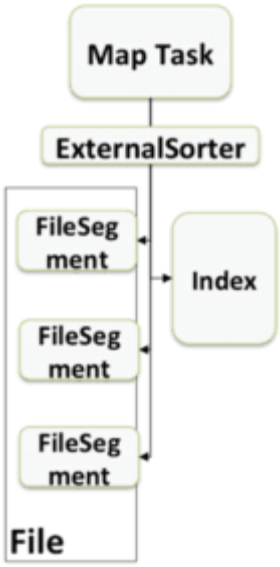
该图描绘的只是Spark 0.X版本而已，让人误以为Spark不支持大规模的集群计算，当时这只是Hash Based Shuffle。Spark后来做了改进，引入了Sort Based Shuffle之后，就再也没有人说Spark只支持小规模的集群运算了。

Hash based shuffle的每个mapper都需要为每个reducer写一个文件，供reducer读取，即需要产生M*R个数量的文件，如果mapper和reducer的数量比较大，产生的文件数会非常多。 Hadoop Map Reduce被人诟病的地方，很多不需要sort的地方的sort导致了不必要的开销，于是Spark的Hash based shuffle设计的目标之一就是避免不需要的排序， 但是它在处理超大规模数据集的时候，产生了大量的磁盘IO和内存的消耗，很影响性能。Hash based shuffle不断优化，Spark 0.8.1引入的file consolidation在一定程度上解决了这个问题。

2 Sort based shuffle

为了解决hash based shuffle性能差的问题，Spark 1.1 引入了Sort based shuffle，完全借鉴map reduce实现，每个Shuffle Map Task只产生一个文件，不再为每个Reducer生成一个单独的文件，将所有的结果只写到一个Data文件里，同时生成一个index文件，index文件存储了Data中的数据是如何进行分类的。Reducer可以通过

这个index文件取得它需要处理的数据。 下一个Stage中的Task就是根据这个Index文件来获取自己所要抓取的上一个Stage中的Shuffle Map Task的输出数据。 Shuffle Map Task产生的结果只写到一个Data文件里, 避免产生大量的文件，从而节省了内存的使用和顺序Disk IO带来的低延时。节省内存的使用可以减少GC的风险和频率。 而减少文件的数量可以避免同时写多个文件对系统带来的压力。 Sort based shuffle在速度和内存使用方面也优于Hash based shuffle。 以上逻辑可以使用下图来描述：



Sort based Shuffle包含两阶段的任务： 1）产生Shuffle数据的阶段(Map阶段) 需要实现ShuffleManager中的getWriter来写数据，数据可以通过BlockManager写在内存、磁盘以及Tachyon等，例如想非常快的Shuffle，此时考虑可以把数据写在内存中，但是内存不稳定，建议采用内存+磁盘。 2）使用Shuffle数据的阶段(Reduce阶段) 需要实现ShuffleManager的getReader，Reader会向Driver去获取上一个Stage产生的Shuffle数据)

3 Tungsten-sort Based Shuffle

Tungsten-sort 在特定场景下基于现有的Sort Based Shuffle处理流程，对内存/CPU/Cache使用做了非常大的优化。带来高效的同时，也就限定了自己的使用场景，所以Spark 默认开启的还是Sort Based Shuffle。

Tungsten 是钨丝的意思。 Tungsten Project 是 Databricks 公司提出的对Spark优化内存和CPU使用的计划，该计划初期对Spark SQL优化的最多，不过部分RDD API 还有Shuffle也因此受益。 Tungsten-sort是对普通sort的一种优化，排序的不是内容本身，而是内容序列化后字节数组的指针(元数据)，把数据的排序转变为了指针数组的排序，实现了直接对序列化后的二进制数据进行排序。由于直接基于二进制数据进行操作，所以在这里面没有序列化和反序列化的过程。内存的消耗降低，相应的也会减少gc的开销。

Tungsten-sort优化点主要在三个方面:

1）直接在serialized binary data上进行sort而不是java objects，减少了memory的开销和GC的overhead。 2）提供cache-efficient sorter，使用一个8 bytes的指针，把排序转化成了一个指针数组的排序。 3）spill的merge过程也无需反序列化即可完成。

这些优化的实现导致引入了一个新的内存管理模型，类似OS的Page，Page是由MemoryBlock组成的, 支持off-heap(用NIO或者Tachyon管理) 以及 on-heap 两种模式。为了能够对Record 在这些MemoryBlock进行定位，又引入了Pointer的概念。