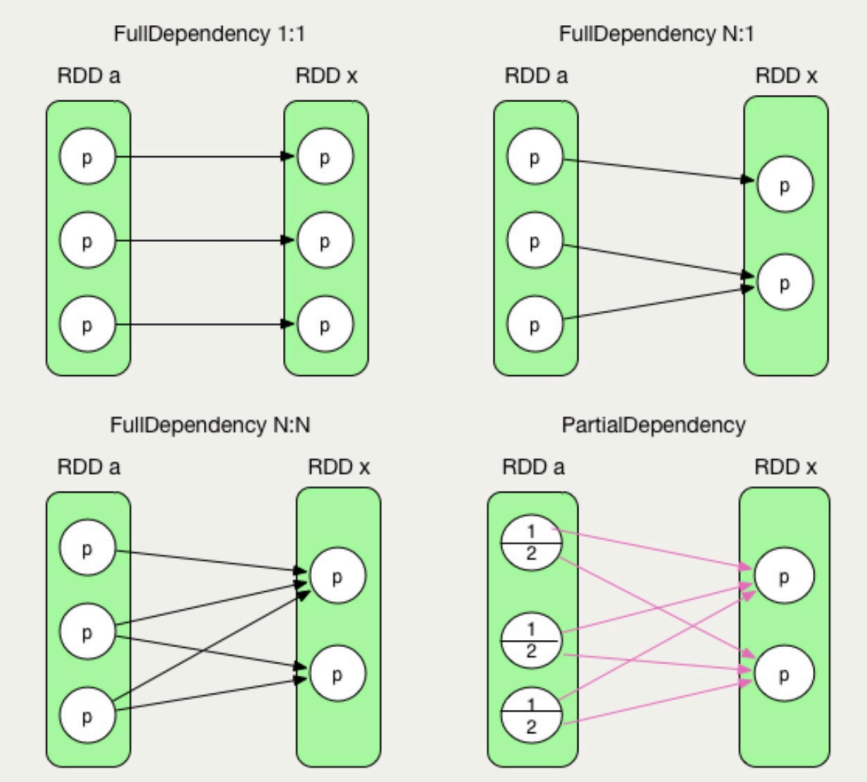
1 Job逻辑执行

1.1 如何计算每个RDD中的数据

如何计算每个 **RDD** 中的数据？逻辑执行图实际上是 computing chain，那么 transformation() 的计算逻辑在哪里被 perform？每个 RDD 里有 compute() 方法，负责接收来自上一个 RDD 或者数据源的 input records，perform transformation() 的计算逻辑，然后输出 records。

1.2 RDD分区依赖关系



前三个是完全依赖，RDD x 中的 partition 与 parent RDD 中的 partition/partitions 完全相关。最后一个是部分依赖，RDD x 中的 partition 只与 parent RDD 中的 partition 一部分数据相关，另一部分数据与 RDD x 中的其他 partition 相关。

在 Spark 中，**完全依赖被称为 NarrowDependency，部分依赖被称为 ShuffleDependency**

1.3 子RDDpartition个数

子RDD中的 partition 个数一般由用户指定，不指定的话一般取`max(numPartitions[parent RDD 1], .., numPartitions[parent RDD n])`

2 Job物理执行

2.1 stage划分算法

划分算法就是：从后往前推算，遇到 **ShuffleDependency** 就断开，遇到 **NarrowDependency** 就将其加入该 **stage**。每个 **stage** 里面 **task** 的数目由该 **stage** 最后一个 **RDD** 中的 **partition** 个数决定。

2.2 生成Job

用户的 driver 程序中一旦出现 action()，就会生成一个 job，比如 `foreach()` 会调用`sc.runJob(this, (iter: Iterator[T]) => iter.foreach(f))`，向 DAGScheduler 提交 job。如果 driver 程序后面还有 action()，那么其他 action() 也会生成 job 提交。所以，driver 有多少个 action()，就会生成多少个 job。这就是 Spark 称 driver 程序为 application（可能包含多个 job）而不是 job 的原因。

2.3 物理图的执行

回想 pipeline 的思想是 **数据用的时候再算，而且数据是流到要计算的位置的**。Result 产生的地方的就是要计算的位置，要确定“需要计算的数据”，我们可以从后往前推，需要哪个 partition 就计算哪个 partition，如果 partition 里面没有数据，就继续向前推，形成 computing chain。这样推下去，结果就是：需要首先计算出每个 stage 最左边的 RDD 中的某些 partition。

对于没有 **parent stage** 的 **stage**，该 **stage** 最左边的 **RDD** 是可以立即计算的，而且每计算出一个 **record** 后便可以流入 **f** 或 **g**（见前面图中的 **patterns**）。如果 f 中的 record 关系是 1:1 的，那么 f(record1) 计算结果可以立即顺着 computing chain 流入 g 中。如果 f 的 record 关系是 N:1，record1 进入 f() 后也可以被回收。总结一下，computing chain 从后到前建立，而实际计算出的数据从前到后流动，而且计算出的第一个 record 流动到不能再流动后，再计算下一个 record。这样，虽然是要计算后续 RDD 的 partition 中的 records，但并不是要求当前 RDD 的 partition 中所有 records 计算得到后再整体向后流动。

对于有 parent stage 的 stage，先等着所有 parent stages 中 final RDD 中数据计算好，然后经过 shuffle 后，问题就又回到了计算“没有 parent stage 的 stage”。

> *代码实现：每个 RDD 包含的 getDependency() 负责确立 RDD 的数据依赖，compute() 方法负责接收 parent RDDs 或者 data block 流入的 records，进行计算，然后输出 record。经常可以在 RDD 中看到这样的代码`firstParent[T].iterator(split, context).map(f)`。firstParent 表示该 RDD 依赖的第一个 parent RDD，iterator() 表示 parentRDD 中的 records 是一个一个流入该 RDD 的，map(f) 表示每流入一个 recod 就对其进行 f(record) 操作，输出 record。为了统一接口，这段 compute() 仍然返回一个 iterator，来迭代 map(f) 输出的 records。*

总结一下：整个 **computing chain** 根据数据依赖关系自后向前建立，遇到 **ShuffleDependency** 后形成 **stage**。在每个 **stage** 中，每个 **RDD** 中的 **compute()** 调用 **parentRDD.iter()** 来将 **parent RDDs** 中的 **records** 一个个 **fetch** 过来。

3 shuffle过程

3.1 对比Hadoop Mapreduce和Spark的shuffle过程

从 **high-level 的角度来看，两者并没有大的差别**。 都是将 mapper（Spark 里是 ShuffleMapTask）的输出进行 partition，不同的 partition 送到不同的 reducer（Spark 里 reducer 可能是下一个 stage 里的 ShuffleMapTask，也可能是 ResultTask）。Reducer 以内存作缓冲区，边 shuffle 边 aggregate 数据，等到数据 aggregate 好以后进行 reduce() （Spark 里可能是后续的一系列操作）。

从 **low-level 的角度来看，两者差别不小**。 Hadoop MapReduce 是 **sort-based**，进入 combine() 和 reduce() 的 records 必须先 sort。这样的好处在于 combine/reduce() 可以处理大规模的数据，因为其输入数据可以通过外排得到（mapper 对每段数据先做排序，reducer 的 shuffle 对排好序的每段数据做归并）。目前的 Spark 默认选择的是 **hash-based**，通常使用 HashMap 来对 shuffle 来的数据进行 aggregate，不会对数据进行提前排序。如果用户需要经过排序的数据，那么需要自己调用类似 sortByKey() 的操作；如果你是Spark 1.1的用户，可以将spark.shuffle.manager设置为sort，则会对数据进行排序。在Spark 1.2中，sort将作为默认的Shuffle实现。

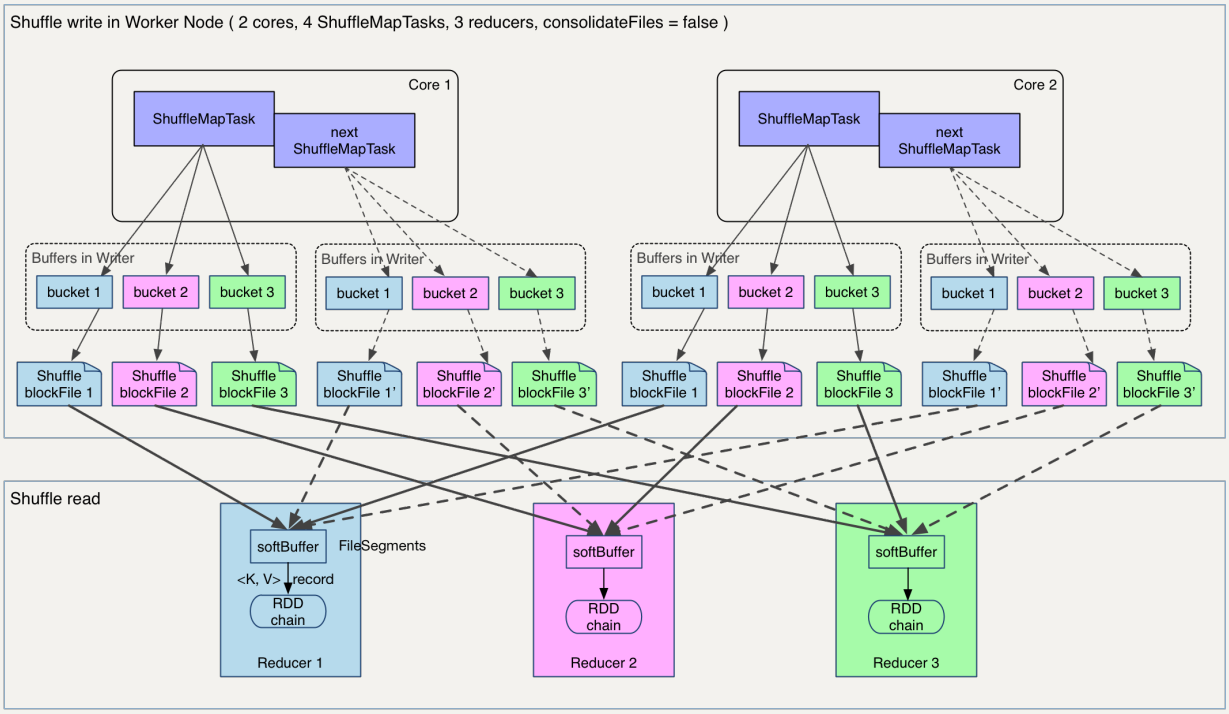
从实现角度来看，两者也有不少差别。 Hadoop MapReduce 将处理流程划分出明显的几个阶段：map(), spill, merge, shuffle, sort, reduce() 等。每个阶段各司其职，可以按照过程式的编程思想来逐一实现每个阶段的功能。在 Spark 中，没有这样功能明确的阶段，只有不同的 stage 和一系列的 transformation()，所以 spill, merge, aggregate 等操作需要蕴含在 transformation() 中。

如果我们将 map 端划分数据、持久化数据的过程称为 shuffle write，而将 reducer 读入数据、aggregate 数据的过程称为 shuffle read。那么在 Spark 中，**问题就变为怎么在 job 的逻辑或者物理执行图中加入 shuffle write 和 shuffle read 的处理逻辑？以及两个处理逻辑应该怎么高效实现？**

3.2 shuffle write

由于不要求数据有序，shuffle write 的任务很简单：将数据 partition 好，并持久化。之所以要持久化，一方面是要减少内存存储空间压力，另一方面也是为了 fault-tolerance。

shuffle write 的任务很简单，那么实现也很简单：将 shuffle write 的处理逻辑加入到 ShuffleMapStage（ShuffleMapTask 所在的 stage） 的最后，该 stage 的 final RDD 每输出一个 record 就将其 partition 并持久化。图示如下：



上图有 4 个 ShuffleMapTask 要在同一个 worker node 上运行，CPU core 数为 2，可以同时运行两个 task。每个 task 的执行结果（该 stage 的 finalRDD 中某个 partition 包含的 records）被逐一写到本地磁盘上。每个 task 包含 R 个缓冲区，R = reducer 个数（也就是下一个 stage 中 task 的个数），缓冲区被称为 bucket，其大小为`spark.shuffle.file.buffer.kb`，默认是 32KB（Spark 1.1 版本以前是 100KB）。

ShuffleMapTask 的执行过程很简单：先利用 pipeline 计算得到 finalRDD 中对应 partition 的 records。每得到一个 record 就将其送到对应的 bucket 里，具体是哪个 bucket 由`partitioner.partition(record.getKey())`决定。每个 bucket 里面的数据会不断被写到本地磁盘上，形成一个 ShuffleBlockFile，或者简称 **FileSegment**。之后的 reducer 会去 fetch 属于自己的 FileSegment，进入 shuffle read 阶段。

为解决文件过多的问题，可以开启FileConsolidation 功能可以通过`spark.shuffle.consolidateFiles=true`来开启。

在一个 core 上连续执行的 ShuffleMapTasks 可以共用一个输出文件 ShuffleFile。先执行完的 ShuffleMapTask 形成 ShuffleBlock i，后执行的 ShuffleMapTask 可以将输出数据直接追加到 ShuffleBlock i 后面

3.3 shuffle读

要计算 ShuffleRDD 中的数据，必须先把 MapPartitionsRDD 中的数据 fetch 过来。那么问题就来了：

- 在什么时候 fetch，parent stage 中的一个 ShuffleMapTask 执行完还是等全部 ShuffleMapTasks 执行完？
- 边 fetch 边处理还是一次性 fetch 完再处理？
- fetch 来的数据存放到哪里？
- 怎么获得要 fetch 的数据的存放位置？

解决问题：

3.3.1 在什么时候 fetch？

当 parent stage 的所有 ShuffleMapTasks 结束后再 fetch。为了迎合 stage 的概念（即一个 stage 如果其 parent stages 没有执行完，自己是不能被提交执行的）

3.3.2 边 fetch 边处理还是一次性 fetch 完再处理？

边 fetch 边处理。因为 Spark 不要求 shuffle 后的数据全局有序，因此没必要等到全部数据 shuffle 完成后再处理

那么如何实现边 shuffle 边处理，而且流入的 records 是无序的？答案是使用可以 aggregate 的数据结构，比如 HashMap。每 shuffle 得到（从缓冲的 FileSegment 中 deserialize 出来）一个 <Key, Value> record，直接将其放进 HashMap 里面。如果该 HashMap 已经存在相应的 Key，那么直接进行 aggregate 也就是 `func(hashMap.get(Key), Value)`，比如上面 WordCount 例子中的 func 就是 `hashMap.get(Key) + Value`，并将 func 的结果重新 put(key) 到 HashMap 中去。这个 func 功能上相当于 reduce()，但实际处理数据的方式与 MapReduce reduce() 有差别

3.3.3 fetch 来的数据存放到哪里？

刚 fetch 来的 FileSegment 存放在 softBuffer 缓冲区，经过处理后的数据放在内存 + 磁盘上。这里我们主要讨论处理后的数据，可以灵活设置这些数据是“只用内存”还是“内存 + 磁盘”。如果 `spark.shuffle.spill = false` 就只用内存。内存使用的是 `AppendOnlyMap` ，类似 Java 的 `HashMap`，内存 + 磁盘使用的是 `ExternalAppendOnlyMap`，如果内存空间不足时，`ExternalAppendOnlyMap` 可以将 <K, V> records 进行 sort 后 spill 到磁盘上，等到需要它们的时候再进行归并，后面会详解。**使用“内存 + 磁盘”的一个主要问题就是如何在两者之间取得平衡？**在 Hadoop MapReduce 中，默认将 reducer 的 70% 的内存空间用于存放 shuffle 来的数据，等到这个空间利用率达到 66% 的时候就开始 merge-combine()-spill。在 Spark 中，也适用同样的策略，一旦 ExternalAppendOnlyMap 达到一个阈值就开始 spill，具体细节下面会讨论。

3.3.4 怎么获得要 fetch 的数据的存放位置？

在上一章讨论物理执行图中的 stage 划分的时候，我们强调 “一个 ShuffleMapStage 形成后，会将该 stage 最后一个 final RDD 注册到 `MapOutputTrackerMaster.registerShuffle(shuffleId, rdd.partitions.size)`，这一步很重要，因为 shuffle 过程需要 MapOutputTrackerMaster 来指示 ShuffleMapTask 输出数据的位置”。因此，reducer 在 shuffle 的时候是要去 driver 里面的 MapOutputTrackerMaster 询问 ShuffleMapTask 输出的数据位置的。每个 ShuffleMapTask 完成时会将 FileSegment 的存储位置信息汇报给 MapOutputTrackerMaster。