

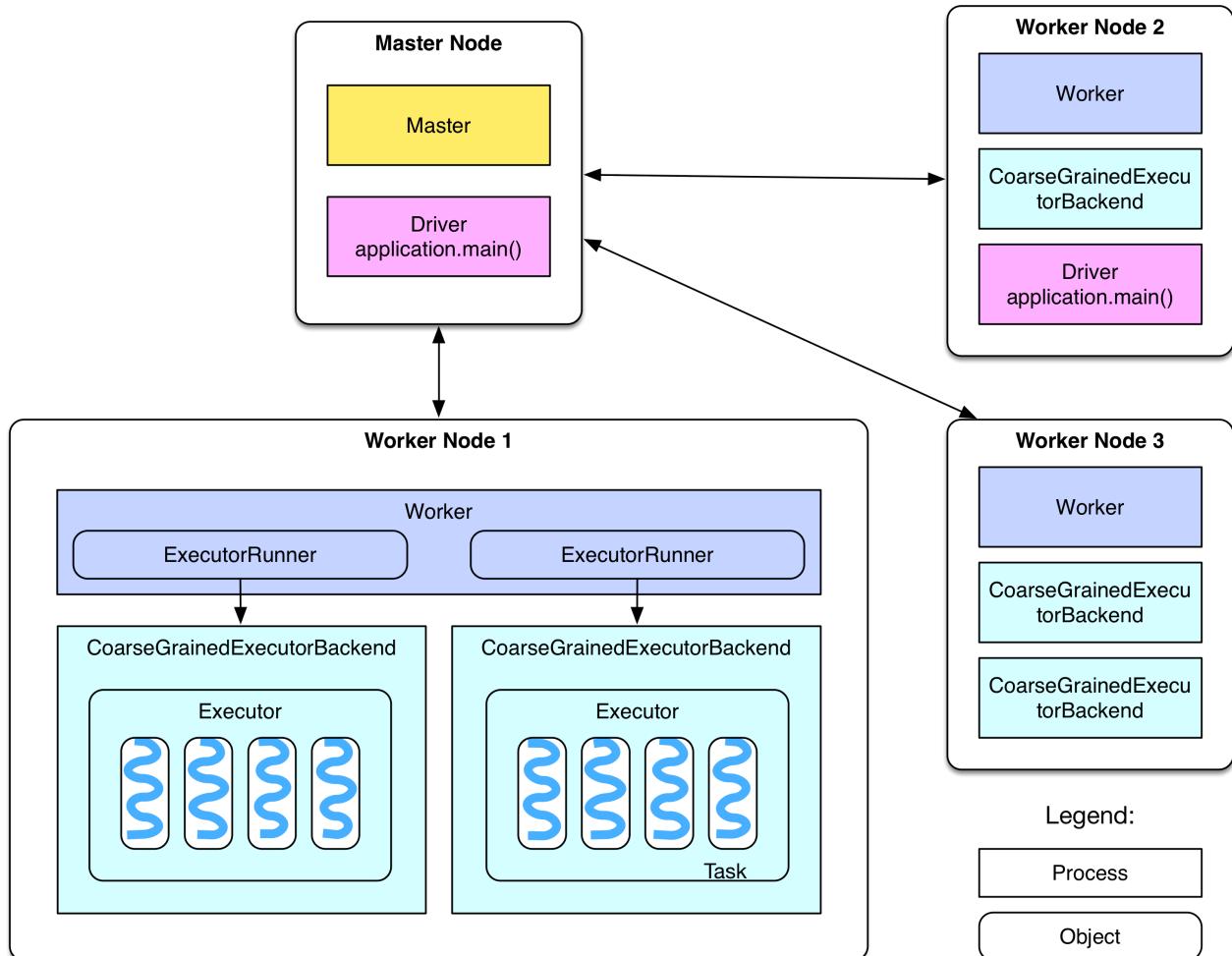
2017-02-03 Spark 學習筆記

=====

Spark設計與實現

參考網站：<https://spark-internals.books.yourtion.com/markdown/1-Overview.html>

=====



```
package org.apache.spark.examples

import java.util.Random

import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.SparkContext._

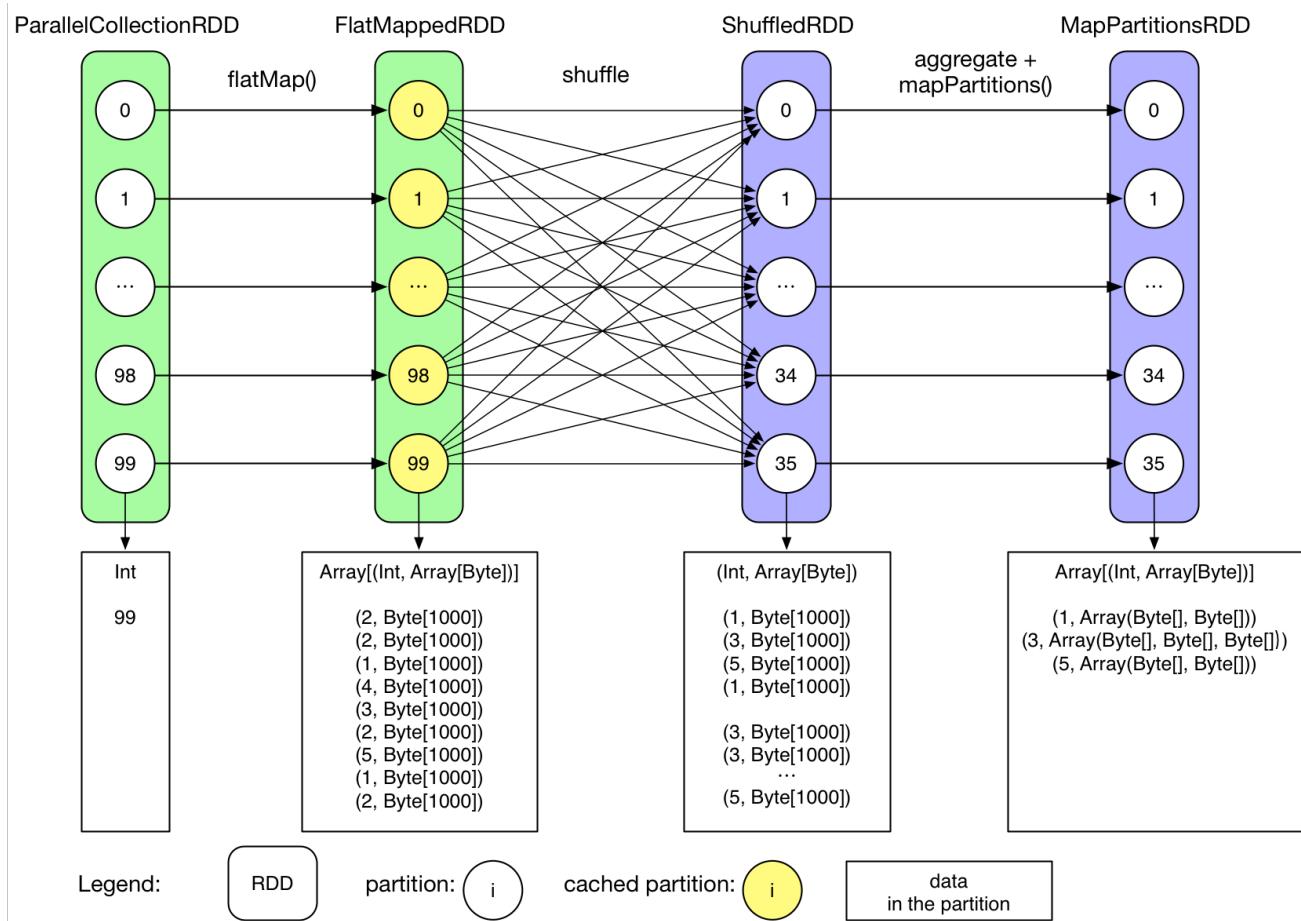
/** 
 * Usage: GroupByTest [numMappers] [numKVPairs] [valSize] [numReducers]
 */
object GroupByTest {
  def main(args: Array[String]) {
    val sparkConf = new SparkConf().setAppName("GroupBy Test")
    var numMappers = 100
    var numKVPairs = 10000
    var valSize = 1000
    var numReducers = 36

    val sc = new SparkContext(sparkConf)

    val pairs1 = sc.parallelize(0 until numMappers, numMappers).flatMap { p =>
      val ranGen = new Random
      var arr1 = new Array[(Int, Array[Byte])](numKVPairs)
      for (i <- 0 until numKVPairs) {
        val byteArr = new Array[Byte](valSize)
        ranGen.nextBytes(byteArr)
        arr1(i) = (ranGen.nextInt(Int.MaxValue), byteArr)
      }
      arr1
    }.cache
    // Enforce that everything has been calculated and in cache
    pairs1.count

    println(pairs1.groupByKey(numReducers).count)

    sc.stop()
  }
}
```



computing chain?????????????

=====

`Union()`

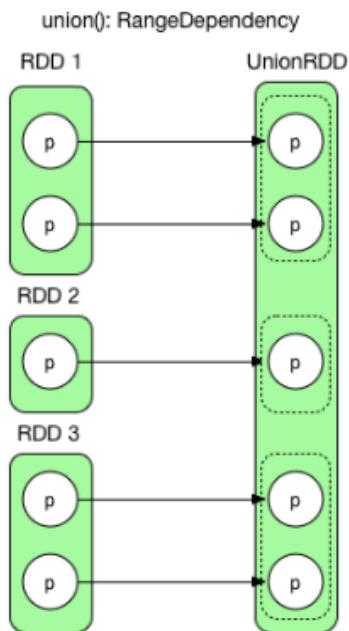
概念：將兩個RDD進行合併，不去除重複的部分

參考網站：<http://lxw1234.com/archives/2015/07/345.htm>

=====

3. 给出一些典型的 transformation() 的计算过程及数据依赖图

1) union(otherRDD)



union() 将两个 RDD 简单合并在一起，不改变 partition 里面的数据。RangeDependency 实际上也是 1:1，只是为了访问 union() 后的 RDD 中的 partition 方便，保留了原始 RDD 的 range 边界。

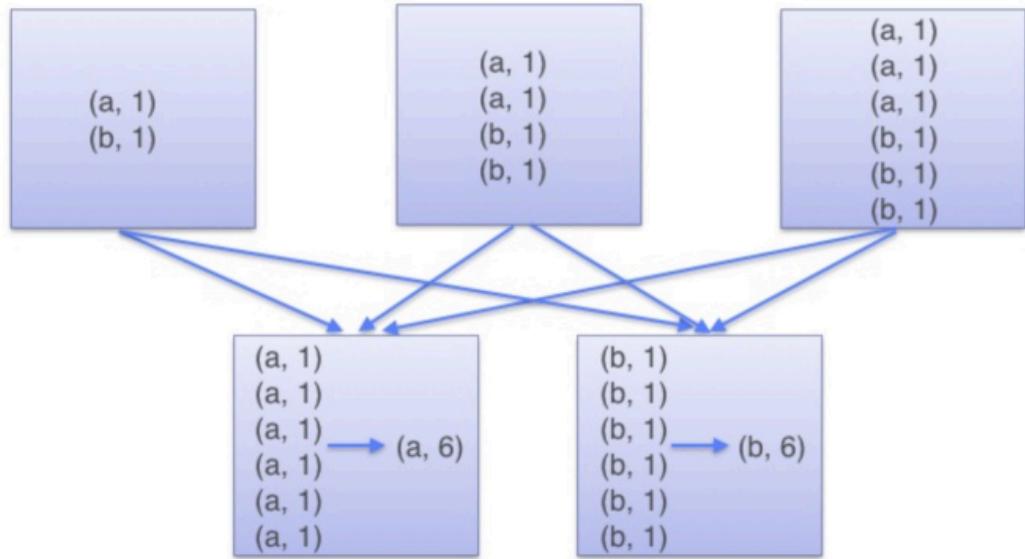
```
-- - - - -  
scala> var rdd1 = sc.makeRDD(1 to 2,1)  
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[2] at makeRDD at <console>:28  
  
scala> var rdd2 = sc.makeRDD(2 to 3,1)  
rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[3] at makeRDD at <console>:28  
  
scala> rdd1.collect()  
res3: Array[Int] = Array(1, 2)  
  
scala> rdd2.collect()  
res4: Array[Int] = Array(2, 3)  
  
scala> rdd1.union(rdd2).collect  
res5: Array[Int] = Array(1, 2, 2, 3)
```

groupByKey()

參考網站：<http://lxw1234.com/archives/2015/07/345.htm>

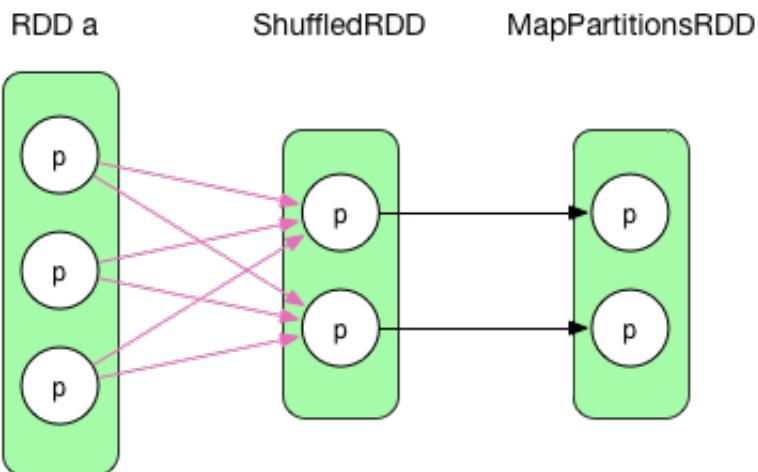
=====

GroupByKey

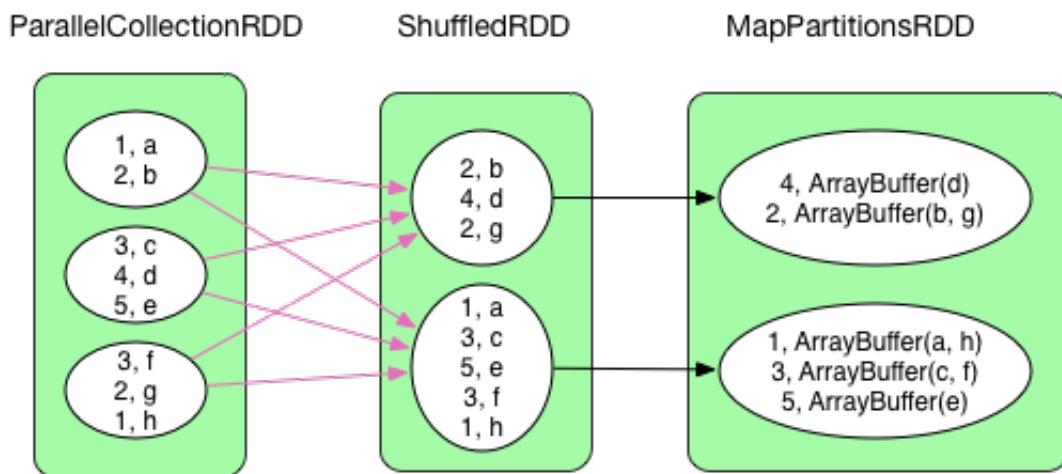


image

groupByKey(numPartitions)



Example: groupByKey(2)



groupByKey() 只需要将 Key 相同的 records 聚合在一起，一个简单的 shuffle 过程就可以完成。

ShuffledRDD 中的 compute() 只负责将属于每个 partition 的数据 fetch 过来，之后使用 mapPartitions() 操作（前面的 OneToOneDependency 展示过）进行 aggregate，生成 MapPartitionsRDD，到这里 groupByKey() 已经结束。最后为了统一返回值接口，将 value 中的 ArrayBuffer[] 数据结构抽象化成 Iterable[]。

groupByKey() 没有在 map 端进行 combine，因为 map 端 combine 只会省掉 partition 里面重复 key 占用的空间，当重复 key 特别多时，可以考虑开启 combine。

这里的 ArrayBuffer 实际上应该是 CompactBuffer，An append-only buffer similar to ArrayBuffer, but more memory-efficient for small buffers.

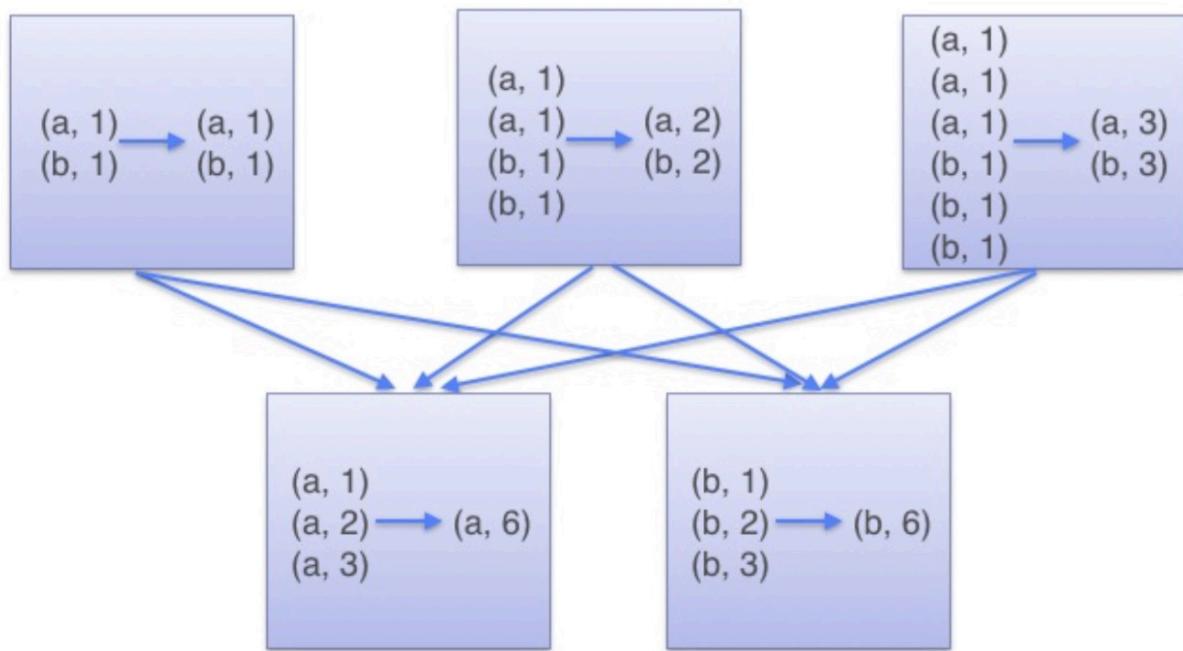
=====

reduceByKey()

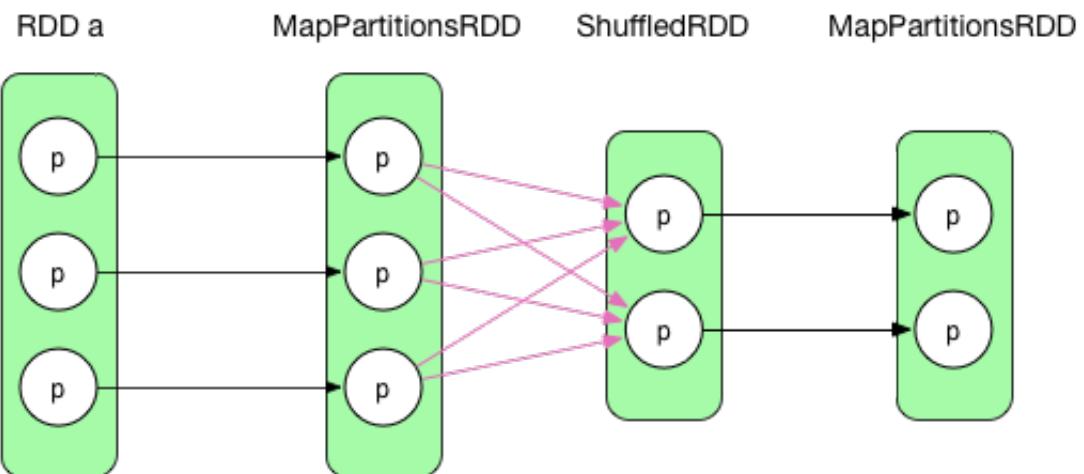
參考網站：<http://lxw1234.com/archives/2015/07/345.htm>

=====

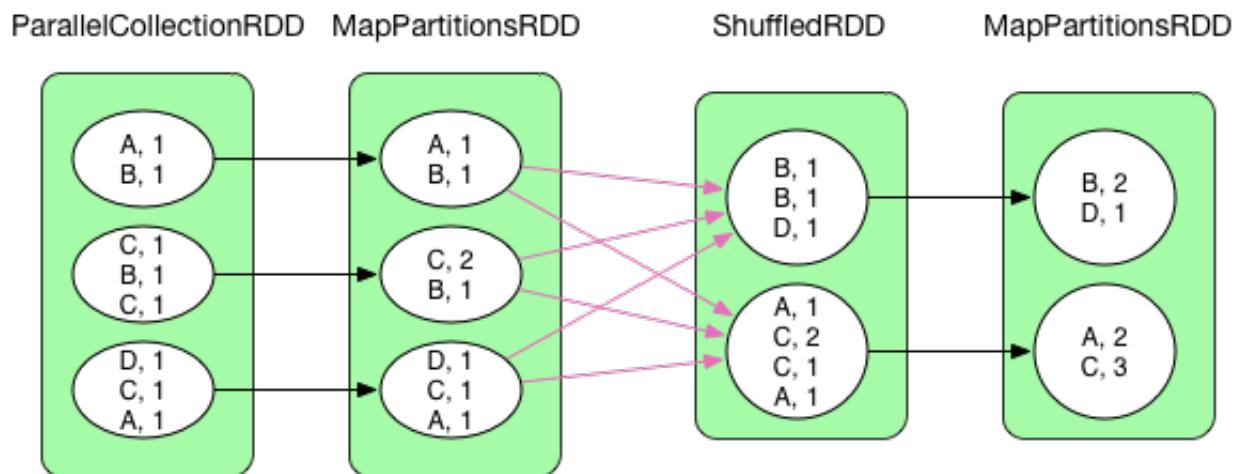
ReduceByKey



reduceByKey(f, numPartitions)



Example (WordCount): reduceByKey(_ + _, 2)



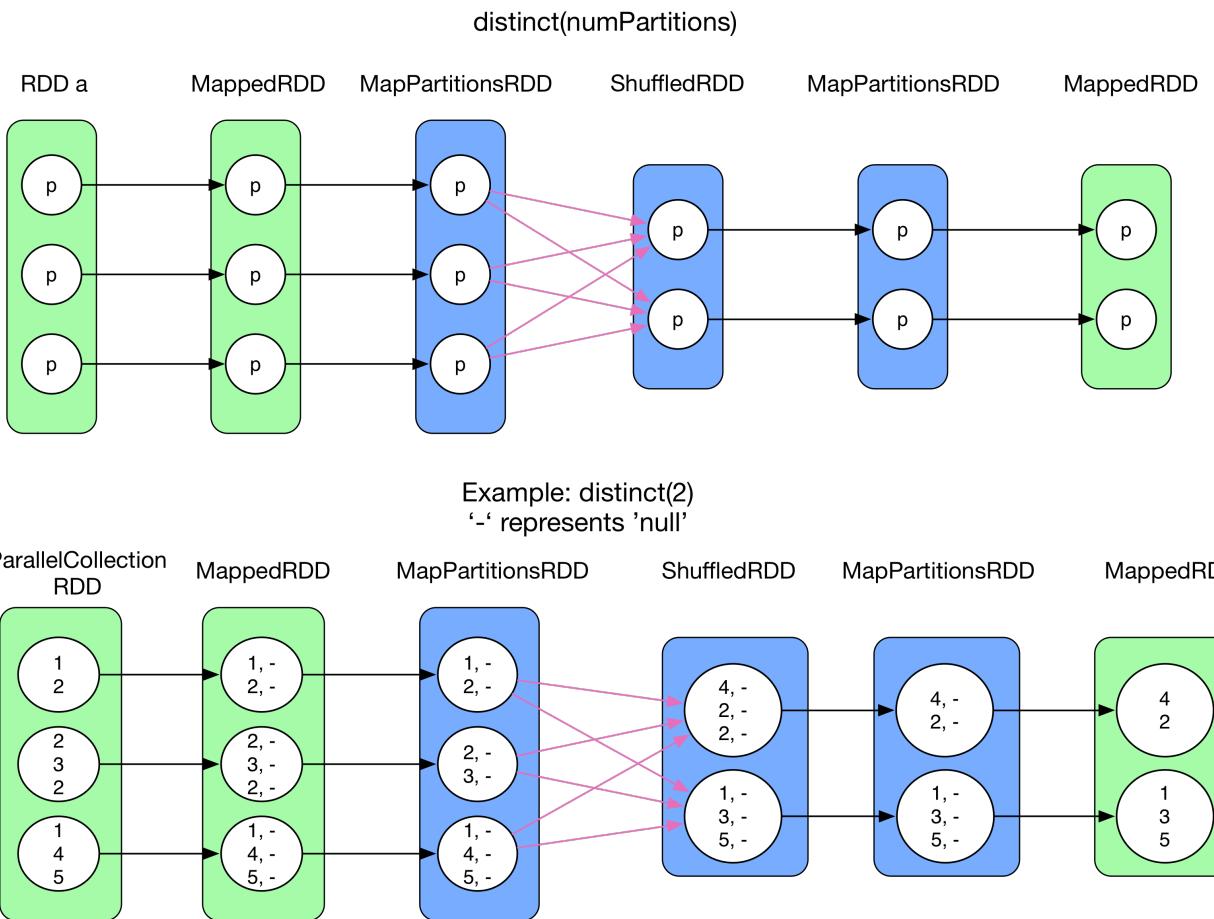
reduceByKey() 相当于传统的 MapReduce，整个数据流也与 Hadoop 中的数据流基本一样。

reduceByKey() 默认在 map 端开启 combine()，因此在 shuffle 之前先通过 mapPartitions 操作进行 combine，得到 MapPartitionsRDD，然后 shuffle 得到 ShuffledRDD，然后再进行 reduce（通过 aggregate + mapPartitions() 操作来实现）得到 MapPartitionsRDD。

=====

distinct()

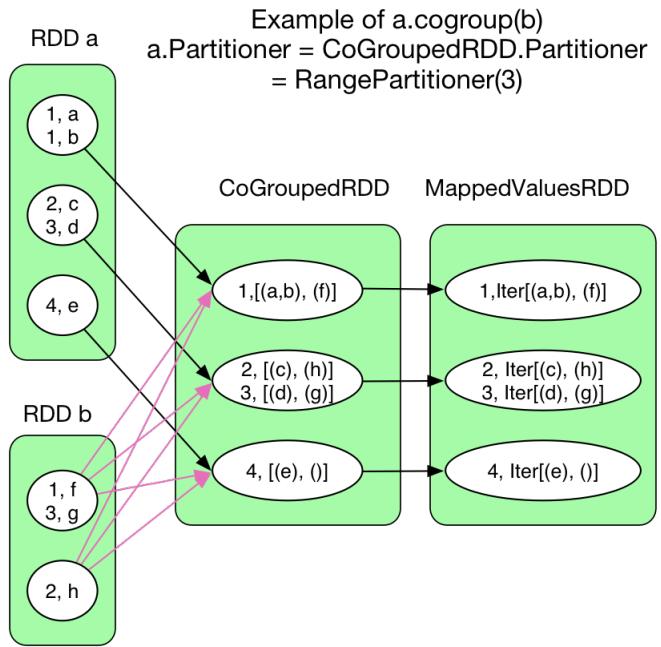
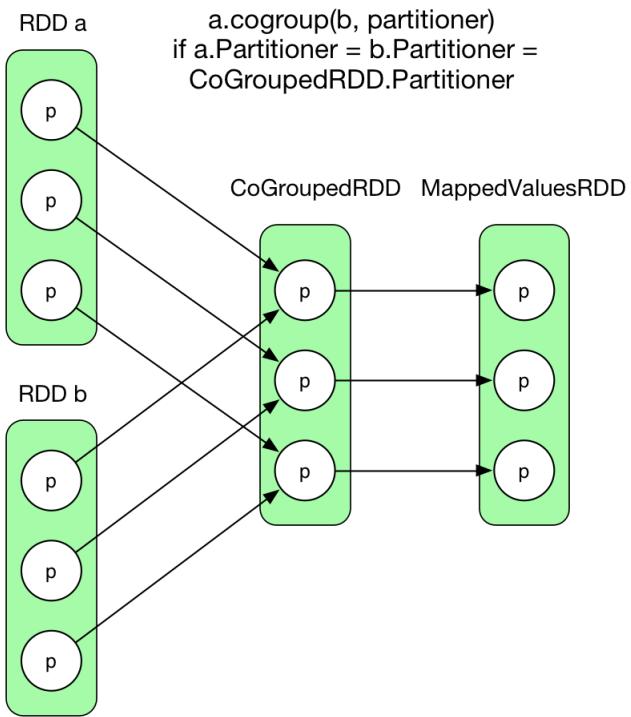
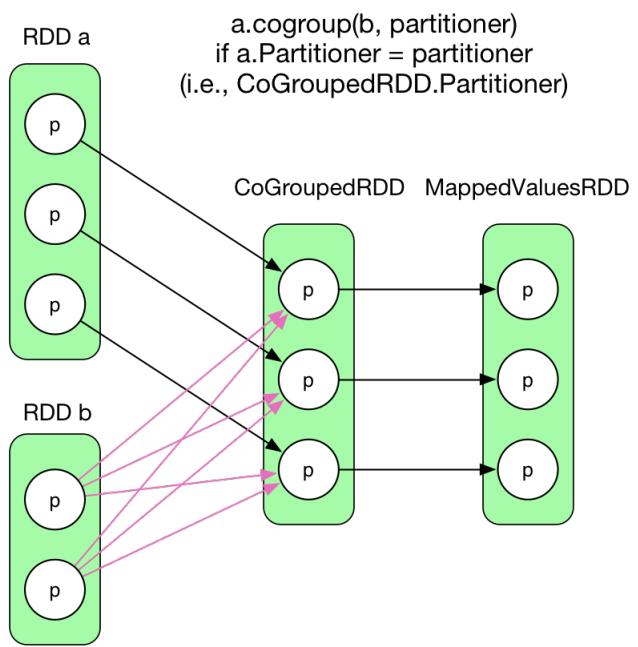
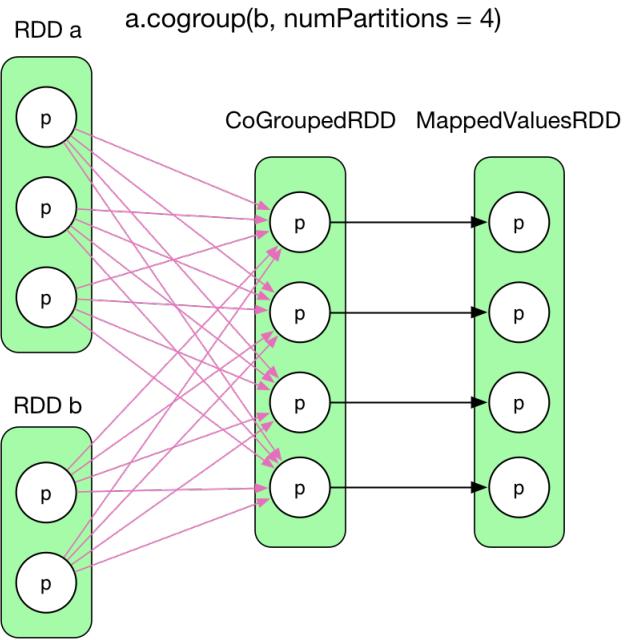
=====



重要觀念：shuffle要求的數據類型是<K,V>

`distinct()` 功能是 deduplicate `RDD` 中的所有重复数据。由于重复数据可能分散在不同的 `partition` 里面，因此需要 `shuffle` 来进行 `aggregate` 后再去重。然而，`shuffle` 要求数据类型是 `<K, V>`。如果原始数据只有 `Key`（比如例子中 `record` 只有一个整数），那么需要补充成 `<K, null>`。这个补充过程由 `map()` 操作完成，生成 `MappedRDD`。然后调用上面的 `reduceByKey()` 来进行 `shuffle`，在 `map` 端进行 `combine`，然后 `reduce` 进一步去重，生成 `MapPartitionsRDD`。最后，将 `<K, null>` 还原成 `K`，仍然由 `map()` 完成，生成 `MappedRDD`。蓝色的部分就是调用的 `reduceByKey()`。

```
=====
cogroup()
=====
```



与 groupByKey() 不同，cogroup() 要 aggregate 两个或两个以上的 RDD。那么 **CoGroupedRDD** 与 **RDD a** 和 **RDD b** 的关系都必须是 **ShuffleDependency** 吗？是否存在 **OneToOneDependency**？

首先要明确的是 **CoGroupedRDD** 存在几个 partition 可以由用户直接设定，与 **RDD a** 和 **RDD b** 无关。然而，如果 **CoGroupedRDD** 中 partition 个数与 **RDD a/b** 中的 partition 个数不一样，那么不可能存在 1:1 的关系。

再次，**cogroup()** 的计算结果放在 **CoGroupedRDD** 中哪个 partition 是由用户设置的 **partitioner** 确定的（默认是 **HashPartitioner**）。那么可以推出：即使 **RDD a/b** 中的 partition 个数与 **CoGroupedRDD** 中的一样，如果 **RDD a/b** 中的 **partitioner** 与 **CoGroupedRDD** 中的不一样，也不可能存在 1:1 的关系。比如，在上图的 example 里面，**RDD a** 是 **RangePartitioner**，**b** 是 **HashPartitioner**，**CoGroupedRDD** 也是 **RangePartitioner** 且 partition 个数与 **a** 的相同。那么很自然地，**a** 中的每个 partition 中 records 可以直接送到 **CoGroupedRDD** 中对应的 partition。**RDD b** 中的 records 必须再次进行划分与 shuffle 后才能进入对应的 partition。

最后，经过上面分析，对于两个或两个以上的 **RDD** 聚合，当且仅当聚合后的 **RDD** 中 **partitioner** 类别及 **partition** 个数与前面的 **RDD** 都相同，才会与前面的 **RDD** 构成 1:1 的关系。否则，只能是 **ShuffleDependency**。这个算法对应的代码可以在 **CoGroupedRDD.getDependencies()** 中找到，虽然比较难理解。

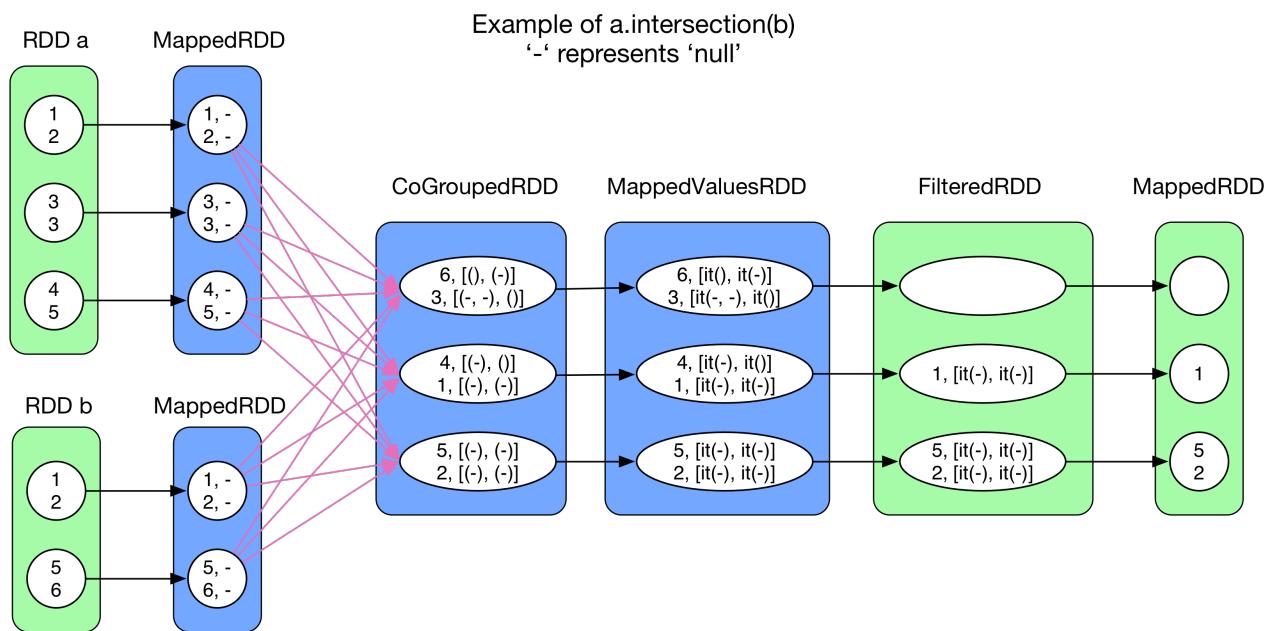
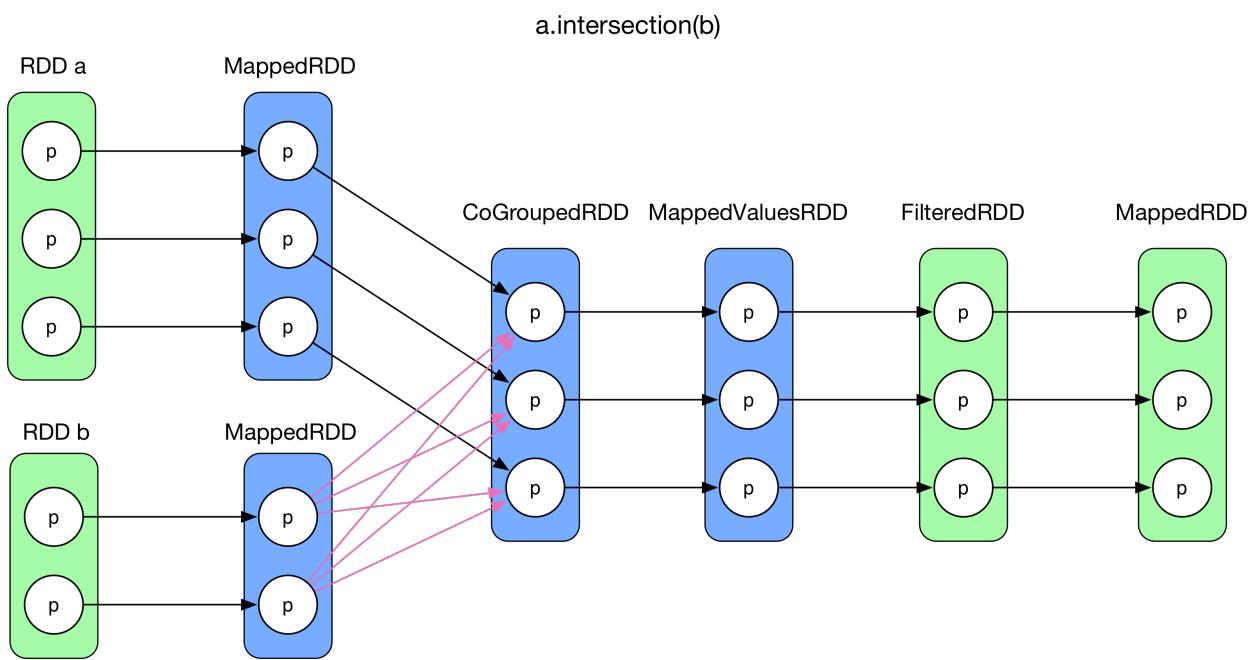
Spark 代码中如何表示 **CoGroupedRDD** 中的 partition 依赖于多个 parent RDDs 中的 partitions？

首先，将 **CoGroupedRDD** 依赖的所有 **RDD** 放进数组 **rdds[RDD]** 中。再次，**foreach i**，如果 **CoGroupedRDD** 和 **rdds(i)** 对应的 **RDD** 是 **OneToOneDependency** 关系，那么 **Dependecy[i] = new OneToOneDependency(rdd)**，否则 = **new ShuffleDependency(rdd)**。最后，返回与每个 parent **RDD** 的依赖关系数组 **deps[Dependency]**。

Dependency 类中的 **getParents(partition id)** 负责给出某个 partition 按照该 dependency 所依赖的 parent **RDD** 中的 partitions: **List[Int]**。

getPartitions() 负责给出 **RDD** 中有多少个 partition，以及每个 partition 如何序列化。

=====
intersection(otherRDD)
=====



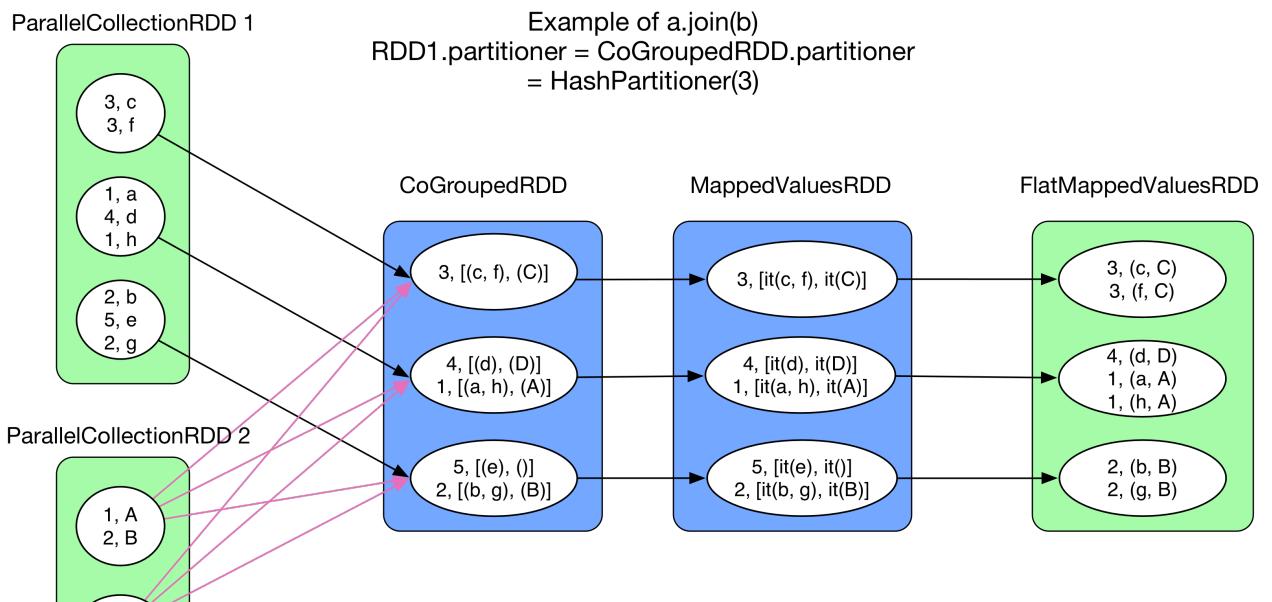
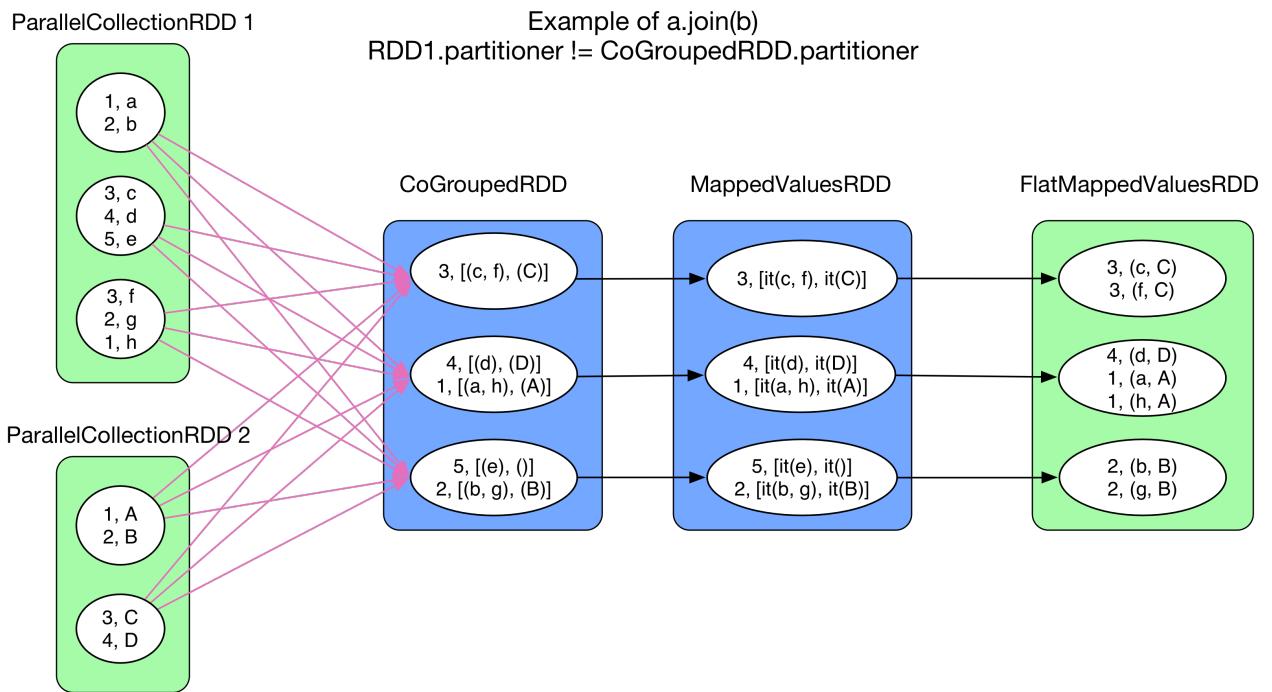
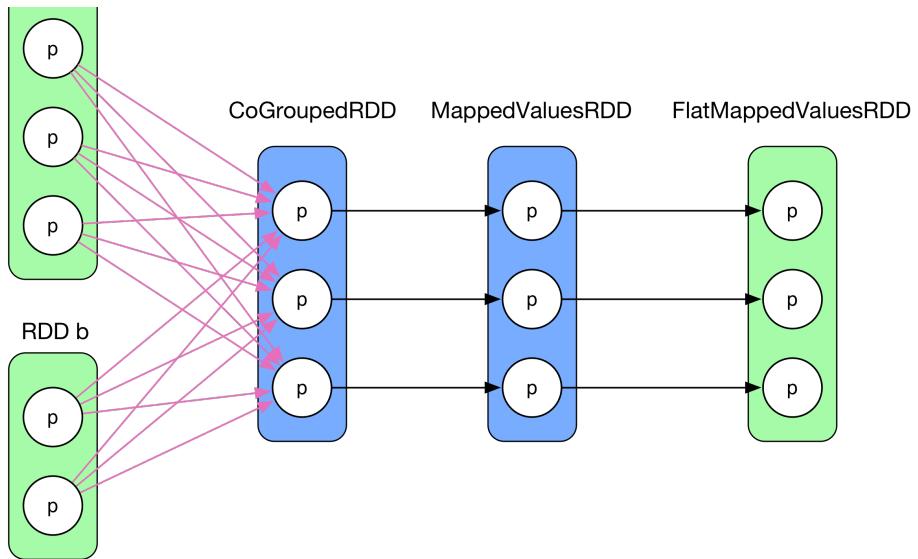
intersection() 功能是抽取出 RDD a 和 RDD b 中的公共数据。先使用 map() 将 RDD[T] 转变成 RDD[(T, null)]，这里的 T 只要不是 Array 等集合类型即可。接着，进行 a.cogroup(b)，蓝色部分与前面的 cogroup() 一样。之后再使用 filter() 过滤掉 [iter(groupA()), iter(groupB())] 中 groupA 或 groupB 为空的 records，得到 FilteredRDD。最后，使用 keys() 只保留 key 即可，得到 MappedRDD。

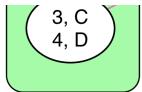
=====

join(otherRDD,Numpartitions)

=====







join() 将两个 $\text{RDD}[(\text{K}, \text{V})]$ 按照 SQL 中的 join 方式聚合在一起。与 intersection() 类似，首先进行 cogroup()，得到 $\langle \text{K}, (\text{Iterable}[\text{V1}], \text{Iterable}[\text{V2}]) \rangle$ 类型的 MappedValuesRDD，然后对 Iterable[V1] 和 Iterable[V2] 做笛卡尔集，并将集合 flat() 化。

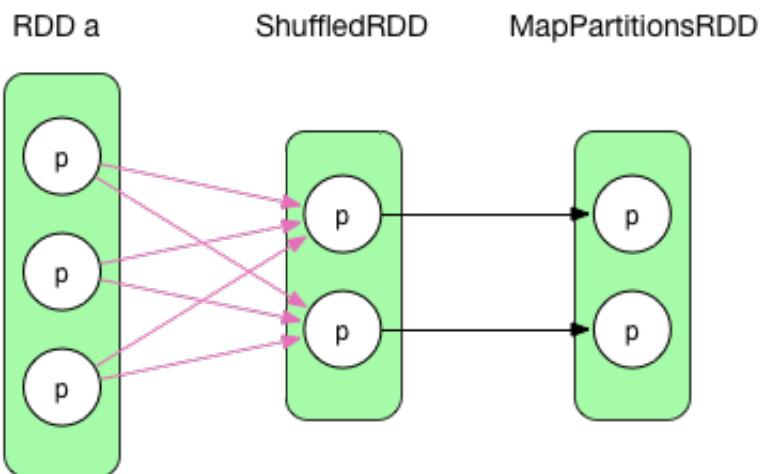
这里给出了两个 example，第一个 example 的 RDD 1 和 RDD 2 使用 RangePartitioner 划分，而 CoGroupedRDD 使用 HashPartitioner，与 RDD 1/2 都不一样，因此是 ShuffleDependency。第二个 example 中，RDD 1 事先使用 HashPartitioner 对其 key 进行划分，得到三个 partition，与 CoGroupedRDD 使用的 HashPartitioner(3) 一致，因此数据依赖是 1:1。如果 RDD 2 事先也使用 HashPartitioner 对其 key 进行划分，得到三个 partition，那么 join() 就不存在 ShuffleDependency 了，这个 join() 也就变成了 hashjoin()。

=====

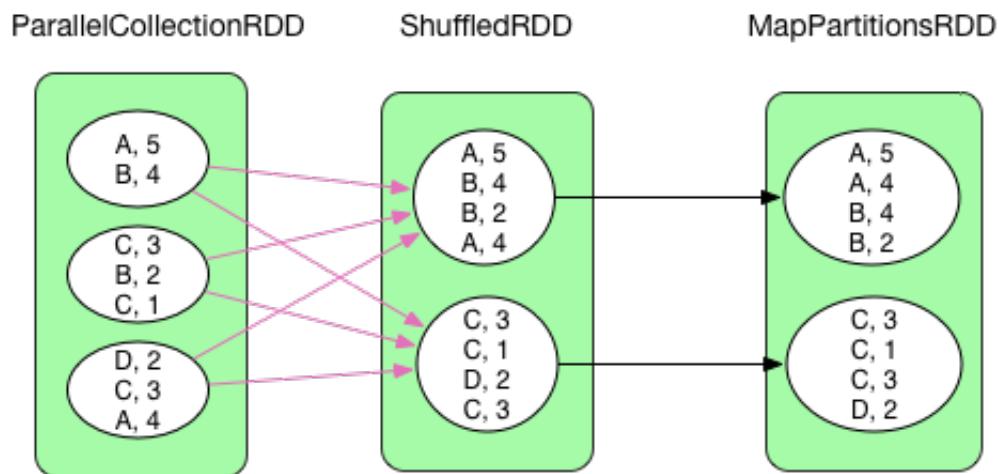
`sortByKey(ascending, numPartitions)`

=====

sortByKey(ascending, numPartitions)



Example of sortByKey(true, 2)



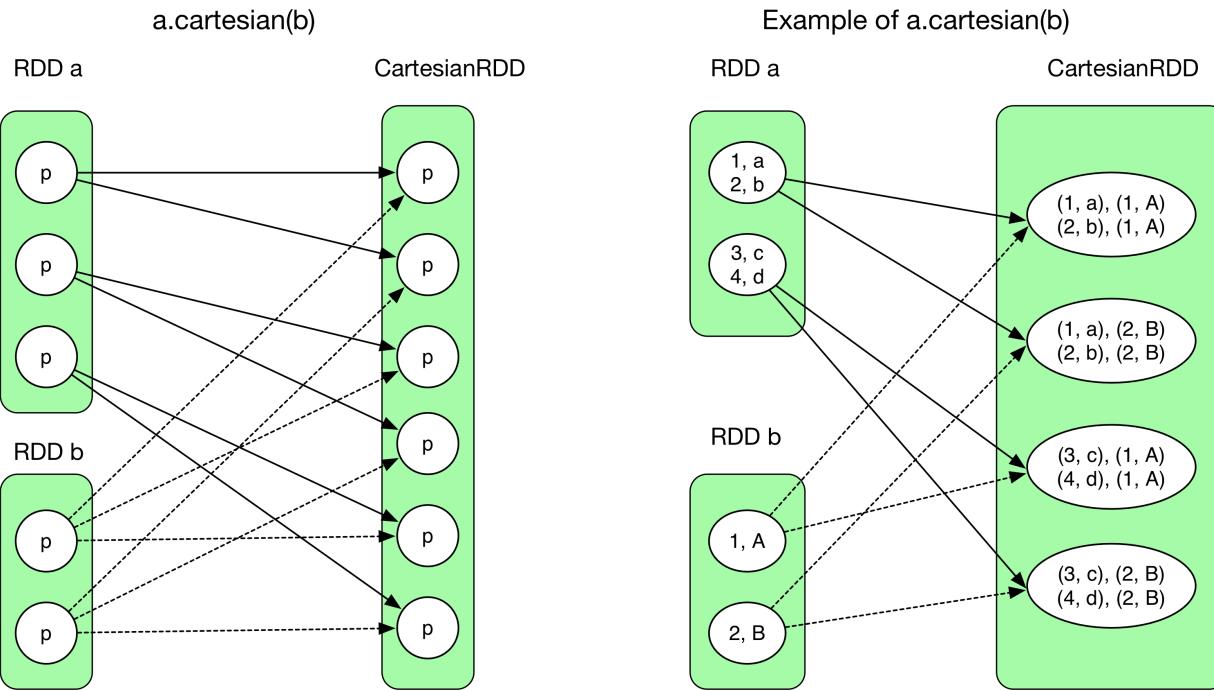
sortByKey() 将 RDD[(K, V)] 中的 records 按 key 排序，ascending = true 表示升序，false 表示降序。目前 sortByKey() 的数据依赖很简单，先使用 shuffle 将 records 聚集在一起（放到对应的 partition 里面），然后将 partition 内的所有 records 按 key 排序，最后得到的 MapPartitionsRDD 中的 records 就有序了。

目前 sortByKey() 先使用 Array 来保存 partition 中所有的 records，再排序。

=====

cartesian(otherRDD)

=====



Cartesian 对两个 RDD 做笛卡尔集，生成的 CartesianRDD 中 partition 个数 = `partitionNum(RDD a) * partitionNum(RDD b)`。

这里的依赖关系与前面的不太一样，CartesianRDD 中每个 partition 依赖两个 parent RDD，而且其中每个 partition 完全依赖 RDD a 中一个 partition，同时又完全依赖 RDD b 中另一个 partition。这里没有红色箭头，因为所有依赖都是 NarrowDependency。

`CartesianRDD.getDependencies()` 返回 `rdds[RDD a, RDD b]`。CartesianRDD 中的 partiton i 依赖于 `(RDD a).List(i / numPartitionsInRDDb)` 和 `(RDD b).List(i % numPartitionsInRDDb)`。

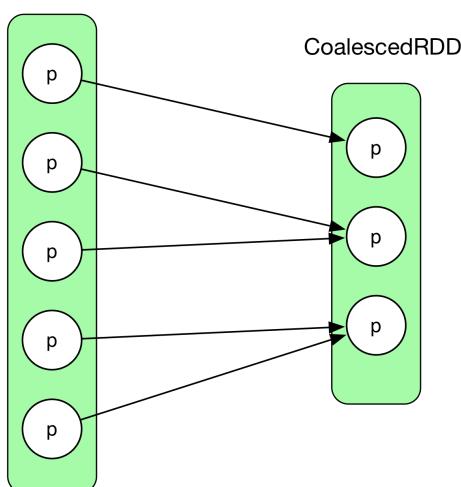
=====

`coalesce(numPartitions,shuffle=false)`

=====

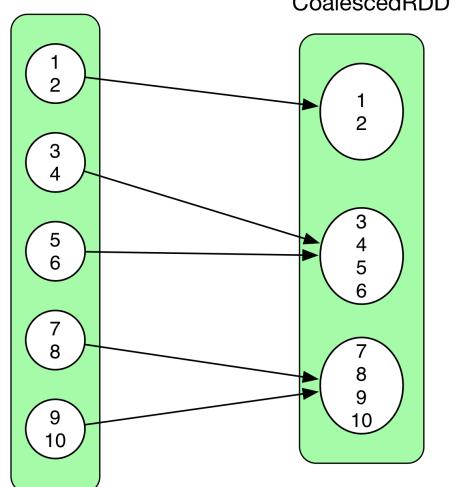
a.coalesce(numPartitions, shuffle = false)

RDD a



Example: a.coalesce(3, shuffle = false)

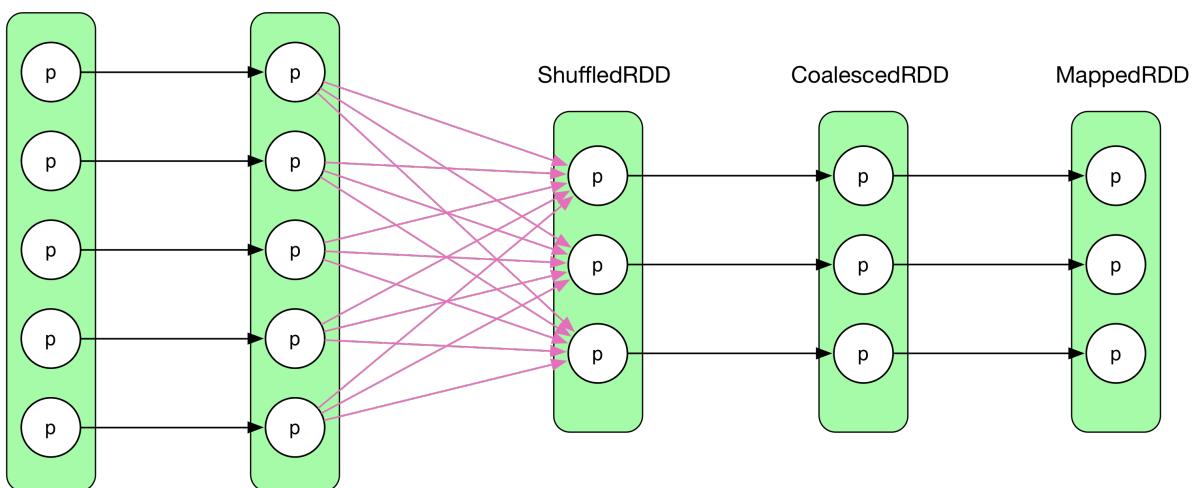
RDD a



a.coalesce(numPartitions, shuffle = true)

RDD a

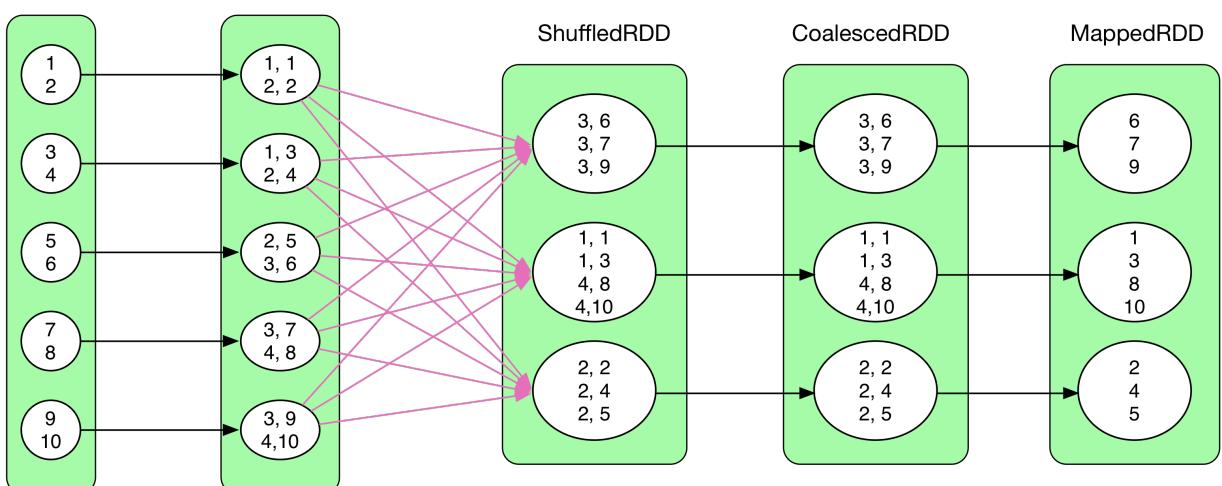
MapPartitionsRDD



Example: a.coalesce(3, shuffle = true)

RDD a

MapPartitionsRDD



不太懂~~~~~

coalesce() 可以将 parent RDD 的 partition 个数进行调整，比如从 5 个减少到 3 个，或者从 5 个增加到 10 个。需要注意的是当 shuffle = false 的时候，是不能增加 partition 个数的（不能从 5 个变为 10 个）。

coalesce() 的核心问题是如何确立 **CoalescedRDD** 中 **partition** 和其 **parent RDD** 中 **partition** 的关系。

- coalesce(shuffle = false) 时，由于不能进行 shuffle，问题变为 **parent RDD** 中哪些 **partition** 可以合并在一起。合并因素除了要考虑 partition 中元素个数外，还要考虑 locality 及 balance 的问题。因此，Spark 设计了一个非常复杂的算法来解决该问题（算法部分我还没有深究）。注意 Example:
`a.coalesce(3, shuffle = false)` 展示了 N:1 的 NarrowDependency。
- coalesce(shuffle = true) 时，由于可以进行 **shuffle**，问题变为如何将 **RDD** 中所有 **records** 平均划分到 **N 个 partition** 中。很简单，在每个 partition 中，给每个 record 附加一个 key，key 递增，这样经过 hash(key) 后，key 可以被平均分配到不同的 partition 中，类似 Round-robin 算法。在第二个例子中，RDD a 中的每个元素，先被加上了递增的 key（如 MapPartitionsRDD 第二个 partition 中 (1, 3) 中的 1）。在每个 partition 中，第一个元素 (Key, Value) 中的 key 由 `(new Random(index)).nextInt(numPartitions)` 计算得到，index 是该 partition 的索引，numPartitions 是 CoalescedRDD 中的 partition 个数。接下来元素的 key 是递增的，然后 shuffle 后的 ShuffledRDD 可以得到均分的 records，然后经过复杂算法来建立 ShuffledRDD 和 CoalescedRDD 之间的数据联系，最后过滤掉 key，得到 coalesce 后的结果 MappedRDD。