第 3 章

键值对与分区

本章介绍如何使用键值对 RDD，这是 Spark 中许多操作所需的常见数据类型。键值 RDD 用于执行聚合，通常做一些数据初始的提取、转换和加载，以将数据转换为键值对格式。在键值对 RDD 上可以应用新的操作，例如计数每个产品的评论；将数据与相同的键分组在一起，并将两个不同的 RDD 分组在一起。

另外，本章还将讨论一个高级功能，即分区功能，可让用户跨节点地控制配对 RDD 的布局。使用可以控制的分区，应用程序有时可以通过确保数据在同一个节点上一起访问，大大降低数据分布在不同节点上的通信成本，这样可以显著减少 RDD 计算时间。

3.1 键值对 RDD

到目前为止，我们已经使用了 RDD，其中每行代表一个值，例如整数或字符串。在许多用例中，需要按某个键进行分组或聚合、联结两个 RDD。现在看一下另一个 RDD 类型：键值对 RDD。键值对的数据格式可以在多种编程语言中找到。它是一组数据类型，由带有一组关联值的键标识符组成。使用分布式数据时，将数据组织成键值对是有用的，因为它允许在网络上聚合数据或重新组合数据。与 MapReduce 类似，Spark 以 RDD 的形式支持键值对数据格式。

在 Scala 语言中，Spark 键值对 RDD 的表示是二维元组。键值对 RDD 在许多 Spark 程序中使用。当想在分布式系统中进行值聚合或重新组合时，需要通过其中的键进行索引，例如有一个包含城市级别人口的数据集，并且想要在省级别汇总，那么就需要按省对这些行进行分组，并对每个省所有城市的人口求和；另一个例子是提取客户标识作为键，以查看所有客户的订单。要想满足键值对 RDD 的要求，每一行必须包含一个元组，其中第一个元素代表键，第二个元素代表值。键和值的类型可以是简单的类型，例如整数或字符串，也可以是复杂的类型，例如对象或值的集合或另一个元组。键值对 RDD 带有一组 API，可以围绕键执行常规操作，例如分组、聚合和连接。

```
scala>val rdd=sc.parallelize(List("Spark","is","an", "amazing", "piece",
"of","technology"))
rdd: org. apache. spark. rdd. RDD [String] = ParallelCollectionRDD [0] at
parallelize at <console>:24

scala>val pairRDD=rdd.map(w =>(w.length,w))
```

```
pairRDD: org.apache.spark.rdd.RDD[(Int, String)]=MapPartitionsRDD[1] at map
at <console>:25

scala>pairRDD.collect().foreach(println)
(5,Spark)
(2,is)
(2,an)
(7,amazing)
(5,piece)
(2,of)
(10,technology)
```

代码 3-1

上面的代码创建了键值对 RDD,每一行为一个元组,其中键是长度,值是单词。它们被包裹在一对括号内。一旦以这种方式排列了每一行,就可以通过按键分组轻松发现长度相同的单词。以下各节将介绍如何创建键值对 RDD,以及如何使用关联的转换和操作。

3.1.1 创建

创建键值对最常用的方法有:使用已经存在的非键值对;加载特定数据创建键值对;通过内存中的集合创建键值对。

虽然大多数 Spark 操作适用于包含任何类型对象的 RDD,但是几个特殊操作只能在键值对的 RDD 上使用,例如按键分组或聚合元素,这些操作都需要进行分布式洗牌。键值对操作在 PairRDDFunctions 类中自动封装在元组 RDD 上。键值对 RDD 中的键和值可以是标量值或复杂值,可以是对象、对象集合或另一个元组。当使用自定义对象作为键值对 RDD 中的键时,该对象的类必须同时定义自定义的 equals()和 hashCode()方法。

语法解释:Scala 元组结合多个固定数量的元素在一起,使它们可以被作为一个整体进行数据传递。不像一个数组或列表,元组可以容纳不同类型的对象,但它们也是不可改变的。下面是一个包括整数、字符串和 Console 的元组:

```
val t=(1, "hello", Console)
```

代码 3-2

这是语法方糖,是下面代码的简写方式:

```
val t=new Tuple3(1, "hello", Console)
```

代码 3-3

一个元组的实际类型取决于它包含的元素和这些元素的类型和数目。因此,该类型(99, "Luftballons")是 Tuple2[Int, String];而('u', 'r', "the", 1, 4, "me")的类型是

Tuple6[Char, Char, String, Int, Int, String]。元组类型包括 Tuple1、Tuple2、Tuple3等,至少目前的上限为22,如果需要更多,可以使用一个集合,而不是一个元组。对于每个 TupleN 类型,其中 1<= N <= 22,Scala 定义了许多元素的访问方法。假定定义一个元组 t 为

```
val t = (4,3,2,1)
```

代码 3-4

要访问元组 t 中的元素,可以使用方法 t._1 访问第一个元素,使用 t._2 访问第二个元素,以此类推。例如,下面的表达式计算 t 的所有元素的总和:

```
val sum=t._1 + t._2 + t._3 + t._4
```

代码 3-5

存在许多格式的数据可以直接加载为键值对,例如 sequenceFile 文件是 Hadoop 用来存储二进制形式的键值对[Key,Value]而设计的一种平面文件。在此示例中,SequenceFile 由键值对(Category,1)组成,当加载到 Spark 中时,会产生键值对 RDD,代码如下。

```
scala>val data=sc.parallelize(List(("key1", 1), ("Key2", 2), ("Key3", 2)))
data: org.apache.spark.rdd.RDD[(String, Int)]=ParallelCollectionRDD[16] at
parallelize at <console>:24
scala>data.saveAsSequenceFile("/data/seq-output")
```

代码 3-6

SequenceFile 可用于解决大量小文件问题,SequenceFile 是 Hadoop API 提供的一种二进制文件支持,直接将键值对序列化到文件中,一般对小文件可以使用这种文件合并,即将文件名作为键,文件内容作为值序列化到大文件中,读取 SequenceFile 的示例如下。

```
scala>import org.apache.hadoop.io.{Text, IntWritable}
import org.apache.hadoop.io.{Text, IntWritable}

scala>val result=sc.sequenceFile("/data/seq-output", classOf[Text],
classOf[IntWritable]).map{case (x, y) =>(x.toString, y.get())}
result: org.apache.spark.rdd.RDD[(String, Int)]=MapPartitionsRDD[19] at map
at <console>:26

scala>result.collect
res11: Array[(String, Int)]=Array((key1,1), (Kay2,2), (Key3,2))
```

代码 3-7

➢ def sequenceFile[K, V](path: String, keyClass: Class[K], valueClass: Class[V]): RDD[(K, V)]

使用给定的键和值类型获取 Hadoop SequenceFile 的 RDD。

- path 为输入数据文件的目录，可以是逗号分隔的路径作为输入列表。
- keyClass 为与 SequenceFileInputFormat 关联的键类。
- valueClass 为与 SequenceFileInputFormat 关联的值类。

可以说，键值对 RDD 在许多程序中起着非常有用的构建块的作用。基本上，一些操作允许我们并行操作每个键，通过这一点可以在整个网络上重新组合数据。reduceByKey()方法分别为每个键聚合数据，而 join()方法通过将具有相同键的元素分组将两个 RDD 合并在一起。可以从 RDD 中提取字段，例如客户 ID、事件时间或其他标识符，然后将这些字段用作键值对 RDD 中的键。

- Scala 模式匹配

Scala 提供了强大的模式匹配机制，应用也非常广泛。一个模式匹配包含一系列备选项，每个都开始于关键字 case。每个备选项都包含一个模式及一到多个表达式。箭头符号 => 隔开了模式和表达式。上面的代码中使用了元组匹配模式，可以使用下面的例子学习其语法。

```
val langs=Seq(
  ("Scala", "Martin", "Odersky"),
  ("Clojure", "Rich", "Hickey"),
  ("Lisp", "John", "McCarthy"))
```

代码 3-8

定义 langs 序列(Seq)变量，其中包含三个三维元组。

```
for (tuple <-langs) {
  tuple match {
    case ("Scala", _, _) =>println("Found Scala")
    case (lang, first, last) =>
      println(s"Found other language: $lang ($first, $last)")
  }
}
```

代码 3-9

在 for 循环中定义了 case 模式匹配。第一个 case 匹配一个三元素元组，其中第一个元素是字符串“Scala”，忽略第二个和第三个参数；第二个 case 匹配任何三元素元组，元素可以是任何类型，但是由于输入的是 langs，因此它们被推断为字符串。将元素提取为变量 lang、first 和 last，输出结果为

```
Found Scala
Found other language: Clojure(Rich, Hickey)
Found other language: Lisp(John, McCarthy)
```

代码 3-10

在上面的代码中，一个元组可以分解成其组成元素。可以匹配元组中的字面值，在任

何想要的位置，可以忽略不关心的元素。

使用 Scala 和 Python 语言，可以使用 SparkContext.parallelize()方法从内存中的数据集合创建一键值对，代码如下。

```
scala>val dist1=Array(("INGLESIDE",1), ("SOUTHERN",1), ("PARK",1),
("NORTHERN",1))
dist1: Array[(String, Int)]=Array((INGLESIDE,1), (SOUTHERN,1), (PARK,1),
(NORTHERN,1))

scala>val dist1RDD=sc.parallelize(dist1)
dist1RDD: org.apache.spark.rdd.RDD[(String, Int)]=ParallelCollectionRDD[44]
at parallelize at <console>:30

scala>dist1RDD.collect
res29: Array[(String, Int)] = Array((INGLESIDE, 1), (SOUTHERN, 1), (PARK, 1),
(NORTHERN,1))
```

代码 3-11

在这个例子中，首先这是在内存中创建键值对集合 dist1，然后通过 SparkContext.parallelize()方法应用于 dist1 创建键值对 dist1RDD。另外，在一组小文本文件上运行 sc.wholetextFiles 将创建键值对，其中键是文件的名称，而值为文件中的内容。

3.1.2　转换

键值对 RDD 允许使用标准 RDD 可用的所有转换，由于键值对包含元组，因此需要在转换方法中传递可以在元组上操作的函数。下面总结了键值对常用的转换。

■ 基于一个键值对 RDD 的转换

创建一个键值对 RDD。

```
scala>val rdd=sc.parallelize(List((1, 2), (3, 4), (3, 6)))
rdd: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[15] at
parallelize at <console>:24
```

代码 3-12

➢ reduceByKey(func：(V，V)⇒ V，numPartitions：Int)：RDD[(K，V)]

调用包含(K，V)的数据集，返回的结果也为(K，V)。数据集中的每个键对应的所有值被聚集，使用给定的汇总功能 func，其类型必须为(V，V) ＝＞ V。像 groupByKey，汇总任务的数量通过第二个可选的参数 numPartitions 配置，这个参数用于设置 RDD 的分区数。

```
scala>rdd.reduceByKey((x, y) =>x +y).collect
res5: Array[(Int, Int)]=Array((1,2), (3,10))
```

代码 3-13

➢ groupByKey(numPartitions：Int)：RDD[(K，Iterable[V])]

调用包含(K，V)的数据集，返回(K，Iterable<V>)。如果分组的目的是为了对每个键执行聚集，如总和或平均值，使用 reduceByKey 或 aggregateByKey 将产生更好的性能。默认情况下，输出的并行任务数取决于 RDD 谱系中父 RDD 的分区数，可以通过一个可选的参数 numPartitions 设置不同数量的任务。

```
scala>rdd.groupByKey().collect
res6: Array [( Int,  Iterable [ Int ])] = Array (( 1, CompactBuffer ( 2)),  ( 3,
CompactBuffer(4, 6)))
```

代码 3-14

➢ combineByKey[C](createCombiner：(V)⇒ C，mergeValue：(C，V) ⇒ C，mergeCombiners：(C，C) ⇒ C)：RDD[(K，C)]

使用相同的键组合值，产生与输入不同的结果类型，例子和详细说明见后面的部分。

➢ mapValues[U](f：(V)⇒ U)：RDD[(K，U)]

对键值对 RDD 的每个值应用一个方法，而不用改变键。

```
scala>rdd.mapValues(x =>x+1).collect
res11: Array[(Int, Int)]=Array((1,3), (3,5), (3,7))
```

代码 3-15

➢ flatMapValues[U](f：(V)⇒ TraversableOnce[U])：RDD[(K，U)]

与 mapValues 相似，将键值对中的每个值传递给函数 f 而不改变键，不同的是将数据的内在结构扁平化。

```
scala>rdd.flatMapValues(x =>(x to 5)).collect
res13: Array[(Int, Int)]=Array((1,2), (1,3), (1,4), (1,5), (3,4), (3,5))
```

代码 3-16

➢ keys：RDD[K]

将键值对 RDD 中每个元组的键返回，产生一个 RDD。

```
scala>rdd.keys.collect
res15: Array[Int]=Array(1, 3, 3)
```

代码 3-17

➢ values：RDD[V]

将键值对 RDD 中每个元组的值返回，产生一个 RDD。

```
scala>rdd.values.collect
res20: Array[Int]=Array(2, 4, 6)
```

代码 3-18

➢ sortByKey(ascending: Boolean = true, numPartitions: Int = self.partitions.length): RDD[(K, V)]

当在数据集(K, V)上被调用时,K 实现了有序化,返回按照键的顺序排列的数据集(K,V),在布尔参数 ascending 中指定升序或降序。

```
scala>rdd.sortByKey().collect
res25: Array[(Int, Int)]=Array((1,2), (3,4), (3,6))
```

代码 3-19

➢ aggregateByKey[U](zeroValue: U)(seqOp: (U, V)⇒ U, combOp: (U, U) ⇒ U)(implicit arg0: ClassTag[U]): RDD[(K, U)]

使用给定的组合函数和中性 zeroValue 聚合每个键的值。该函数可以返回与输入键值对 RDD 中的 V 值类型不同的结果类型 U。因此,需要一个用于将 V 合并到 U 中的操作和一个用于合并两个 U 的操作,如在 scala.TraversableOnce 中,前一个函数 seqOp 用于合并分区中的值,后者 combOp 用于在分区之间合并值。为了避免内存分配,这两个函数都允许修改并返回其第一个参数,而不是创建一个新的 U。

```
scala>val pairRDD=sc.parallelize(List( ("cat",2), ("cat", 5), ("mouse", 4),
("cat", 12), ("dog", 12), ("mouse", 2)), 2)
pairRDD: org.apache.spark.rdd.RDD[(String, Int)]=ParallelCollectionRDD[1] at
parallelize at <console>:24

scala>def myfunc(index: Int, iter: Iterator[(String, Int)]) : Iterator[String]={
    |  iter.map(x =>"[partID:" +  index +", val: " +x +"]")
    | }
myfunc: (index: Int, iter: Iterator[(String, Int)])Iterator[String]

scala>pairRDD.mapPartitionsWithIndex(myfunc).collect
res0: Array[String]=Array([partID:0, val: (cat,2)], [partID:0, val: (cat,5)],
[partID:0, val: (mouse,4)], [partID:1, val: (cat,12)], [partID:1, val: (dog,
12)], [partID:1, val: (mouse,2)])

scala>pairRDD.aggregateByKey(0)(math.max(_, _), _+_).collect
res1: Array[(String, Int)]=Array((dog,12), (cat,17), (mouse,6))

scala>pairRDD.aggregateByKey(100)(math.max(_, _), _+_).collect
res2: Array[(String, Int)]=Array((dog,100), (cat,200), (mouse,200))
```

代码 3-20

上面的代码中,通过定义 myfunc 函数,分别打印出 RDD 分区中的内容。

■ 基于两个键值对 RDD 的转换

创建两个键值对 RDD,分别为

```
scala>val rdd=sc.parallelize(List((1, 2), (3, 4), (3, 6)))
rdd: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[42] at
parallelize at <console>:24

scala>val other=sc.parallelize(List((3,9)))
other: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[43] at
parallelize at <console>:24
```

代码 3-21

➢ subtractByKey

从 RDD 中删除 other 中存在的键元素。

```
scala>rdd.subtractByKey(other).collect
res27: Array[(Int, Int)]=Array((1,2))
```

代码 3-22

➢ join(otherDataset，[numTasks])

在两个 RDD 之间执行内部连接。

```
scala>rdd.join(other).collect
res28: Array[(Int, (Int, Int))]=Array((3,(4,9)), (3,(6,9)))
```

代码 3-23

➢ rightOuterJoin

在两个 RDD 之间执行连接，其中键必须存在于 other 中。

```
scala>rdd.rightOuterJoin(other).collect
res30: Array[(Int, (Option[Int], Int))]=Array((3,(Some(4),9)), (3,(Some(6),
9)))
```

代码 3-24

➢ leftOuterJoin

在两个 RDD 之间执行连接，其中键必须存在于 rdd 中。

```
scala>rdd.leftOuterJoin(other).collect
res31: Array[(Int, (Int, Option[Int]))]=Array((1,(2,None)), (3,(4,Some(9))),
(3,(6,Some(9))))
```

代码 3-25

➢ cogroup(otherDataset，[numTasks])

将两个 RDD 具有相同键的值组合在一起。

```
scala>rdd.cogroup(other).collect
res32: Array[(Int, (Iterable[Int], Iterable[Int]))]=Array((1,(CompactBuffer(2),
CompactBuffer())), (3,(CompactBuffer(4, 6),CompactBuffer(9))))
```

代码 3-26

3.1.2.1 聚合

当使用键值对描述数据集时，通常需要在具有相同键的所有元素上统计数据。对于基本的 RDD 的 fold、combine 和 reduce 操作，在键值对 RDD 上也有基于键的类似操作，这些操作基于相同的键进行汇集。这些操作是转换，而不是动作。

1. reduceByKey

基本上，reduceByKey 函数仅适用于包含键值对元素类型的 RDD，即 Tuple 或 Map 作为数据元素。这是一个转型操作，意味着被惰性评估。我们需要传递一个关联函数作为参数，该函数将应用于键值对 RDD，创建带有结果值的 RDD，即新的键值对。由于分区间可能发生数据 Shuffle，因此此操作是一项涉及全数据集的广泛操作。

在数学中，关联属性是一些二元运算的属性。在命题逻辑中，关联性是在逻辑证明中替换表达式的有效规则。在包含同一个关联运算符的一行中出现两次或更多次的表达式中，只要操作数序列未更改，操作的执行次序就无关紧要。也就是说，重新排列这种表达式中的括号不会改变其值。考虑下面的等式：

$$\left.\begin{aligned}(2+3)+4=2+(3+4)=9\\2\times(3\times4)=(2\times3)\times4=24\end{aligned}\right\} \tag{3-1}$$

关联性让我们可以按顺序并行使用相同的函数。reduceByKey 使用该属性计算 RDD 的结果，RDD 是由分区组成的分布式集合。直观地说，这个函数在重复应用于具有多个分区的同一组 RDD 数据时会产生相同的结果，而不管元素的顺序如何。此外，它首先使用 Reduce 函数在本地执行合并，然后在分区之间发送记录，以准备最终结果。通过下面的代码看一看 reduceByKey 的执行过程。

```
scala>val x =sc.parallelize(Array(("a", 1), ("b", 1), ("a", 1),("a", 1), ("b",
1), ("a", 1),("b", 1), ("b", 1), ("a", 1), ("b", 1), ("a", 1),("b", 1)), 3)
x: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[5] at
parallelize at <console>:24

scala>x.reduceByKey(_ +_).collect()
res3: Array[(String, Int)]=Array((a,6), (b,6))
```

代码 3-27

在图 3-1 中，可以看到 RDD 具有多个键值对元素，如(a,1)和(b,1)，以及 3 个分区。在对整个分区之间的数据洗牌之前，先在每个本地分区中进行相同的聚合。可以使用 reduceByKey 与 mapValues 一起计算每个键的平均值，代码和图示(见图 3-2)如下。

```
scala>val rdd=sc.parallelize(List(("panda",0),("pink",3),
("pirate",3),("panda",1),("pink",4)))
rdd: org.apache.spark.rdd.RDD[(String, Int)]=ParallelCollectionRDD[29] at
parallelize at <console>:24
scala>rdd.mapValues(x =>(x, 1)).reduceByKey((x, y) =>(x._1 +y._1, x._2 +
y._2)).collect
res38: Array[(String, (Int, Int))]=Array((panda, (1, 2)), (pink, (7, 2)),
(pirate,(3,1)))
```

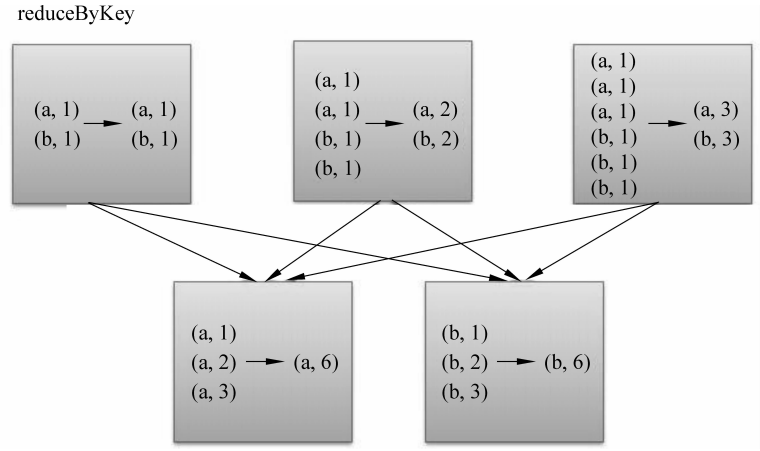

图 3-1　reduceByKey 运行示意图

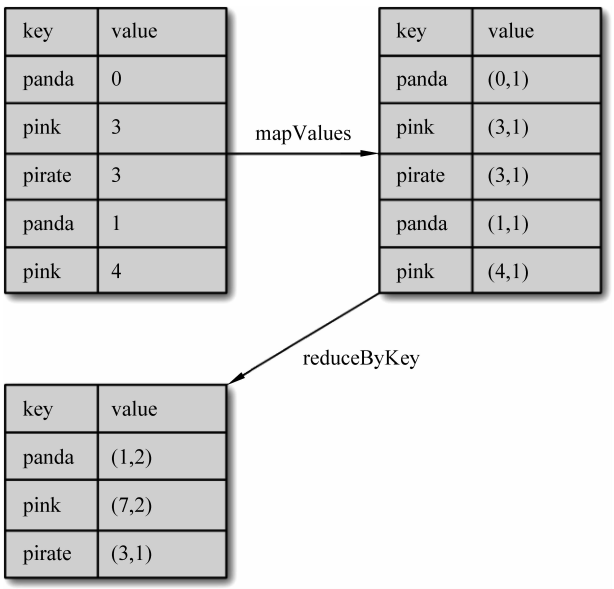

图 3-2　每键平均值计算的数据流

实际上，reduceByKey 是 aggregateByKey 的一个特例。aggregateByKey 有两个参数：一个应用于每个分区的聚合；另一个应用于分区之间的聚合。reduceByKey 在上述两种情况下都使用相同的关联函数，在每个分区上都执行一遍，然后在分区间执行一遍，将第一遍的结果合并为最终结果。

2. combineByKey

combineByKey 调用是一种聚合的优化。使用 combineByKey 值时，每个分区合并为一个值，然后将每个分区值合并为一个值。值得注意的是，组合值的类型不必与原始值的类型相匹配，而且通常不会。combineByKey 函数将 3 个函数作为参数，第一个函数为创建组合器的函数，在 aggregateByKey 函数中，第一个参数只是 zeroValue，在 combineByKey 中提供了一个函数，它将接受当前的值作为参数，并返回将与合成值合并的新值；第二个函数是一个合并函数，它接受一个值并将它合并或组合到先前收集的值中；第三个函数将合并的值组合在一起，基本上这个函数采用在分区级别上产生的新值，并将它们结合起来，直到得到一个最后的结果。下面是一段执行 combineByKey 的代码。

```
scala>val data=sc.parallelize(List(("A", 3), ("A", 9), ("A", 12),("B", 4),
("B", 10), ("B", 11)))
data: org.apache.spark.rdd.RDD[(String, Int)]=ParallelCollectionRDD[0] at
parallelize at <console>:24

scala>val sumCount=data.combineByKey((v)=>(v,1),(acc:(Int,Int),v) =>(acc._1
+v,acc._2+1),(acc1:(Int,Int),acc2:(Int,Int))=>(acc1._1+acc2._1,acc1._2+
acc2._2))
sumCount: org.apache.spark.rdd.RDD[(String, (Int, Int))]=ShuffledRDD[1] at
combineByKey at <console>:26

scala>sumCount.foreach(println)
(B,(25,3))
(A,(24,3))

scala>val averageByKey=sumCount.map{case(key, value)=>(key, value._1 /
value._2.toFloat)}
averageByKey: org.apache.spark.rdd.RDD[(String, Float)]=MapPartitionsRDD[2]
at map at <console>:28

scala>averageByKey.foreach(println)
(A,8.0)
(B,8.333333)
```

代码 3-28

参考上面的代码，combineByKey 需要三个函数，分别为 createCombiner、mergeValue 和 mergeCombiner。

■ **createCombiner**

```
(v)=>(v,1)
```

combineByKey()方法中的第一个函数是必选参数,用作每个键的第一个聚合步骤。当在每个分区中,如果找不到每个键的组合器,createCombiner 会为分区上每个遇到的第一个键创建初始组合器。上面的代码是用在分区中遇到的第一个值和为 1 的键计数器初始化一个 tuple,其值为(v, 1),v 代表第一个遇到的值,表示存储组合器的存储内容为(sum, count)。

■ **mergeValue**

```
(acc:(Int,Int),v) =>(acc._1+v,acc._2+1)
```

这是下一个必需的函数,告诉 combineByKey 当组合器被赋予一个新值时该怎么做。该函数的参数是组合器 acc 和新值 v。组合器的结构在上面被定义为(sum, count)形式的元组,acc._1 执行累加代表组合器中的 sum,acc._2 执行计数代表组合器中的 count。所以,通过将新值 v 添加到组合器元组的第一个元素,同时加 1 到组合器元组的第二个元素合并新值。mergeValue 只有在这个分区上已经创建了初始组合器(在我们的例子中为元组)时才被触发。

■ **mergeCombiner**

```
(acc1:(Int,Int),acc2:(Int,Int))=>(acc1._1+acc2._1,acc1._2+acc2._2)
```

最终一个必需的函数告诉 combineByKey 如何合并分区之间的两个组合器。在这个例子中,每个分区组合器元组的形式为(sum, count),需要做的是将第一个分区依次到最后一个分区中的组合器加在一起。

最终的目标是逐个计算平均值 averageByKey()。combineByKey()的结果是 RDD,其格式为(label, (sum, count)),因此可以通过使用 map()方法,映射(sum, count)到 sum/count 轻松获取平均值。接下来将数据的子集分解到多个分区,并在实际中看数据的计算方式。

```
分区一
A=3 -->createCombiner(3) ==>accum[A]=(3, 1)
A=9 -->mergeValue(accum[A], 9) ==>accum[A]=(3 +9, 1 +1)
B=11 -->createCombiner(11) ==>accum[B]=(11, 1)
分区二
A=12 -->createCombiner(12) ==>accum[A]=(12, 1)
B=4 -->createCombiner(4) ==>accum[B]=(4, 1)
B=10 -->mergeValue(accum[B], 10) ==>accum[B]=(4 +10, 1 +1)
合并分区
A ==>mergeCombiner((12, 2), (12, 1)) ==>(12 +12, 2 +1)
B ==>mergeCombiner((11, 1), (14, 2)) ==>(11 +14, 1 +2)
sumCount 输出为
Array((A, (24, 3)), (B, (25, 3)))
```

3.1.2.2　分组

使用键值对数据，一个常见的用例是按键分组的数据，例如一起查看客户的所有订单。如果数据已经按照想要的方式组成键值对元组，groupByKey 将使用 RDD 中的键对数据进行分组。在由 K 型键和 V 型值构成的 RDD 上，分组后得到[K, Iterable[V]]类型的 RDD。现在使用 groupByKey 实现上面 reduceByKey 代码的功能。

```
scala>val x=sc.parallelize(Array(("a", 1), ("b", 1), ("a", 1),("a", 1), ("b",
1), ("a", 1),("b", 1), ("b", 1), ("a", 1), ("b", 1), ("a", 1),("b", 1)), 3)
x: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[4] at
parallelize at <console>:24

scala>x.groupByKey().map(t =>(t._1, t._2.sum)).collect
res4: Array[(String, Int)]=Array((a,6), (b,6))
```

代码 3-29

得到的结果与上面的代码一致，但是数据的计算过程不一样。另一方面，当调用 groupByKey 时所有的键值对都在洗牌，在网络中传输了大量不必要的数据。当在一个执行器上有更多的数据在内存中进行洗牌时，Spark 将内存数据溢出到磁盘中。但是，一次只会将一个键数据刷新到磁盘上，因此如果单个键的值超过内存容量，则会发生内存不足的异常。这种情况应该避免。当 Spark 需要溢出到磁盘时，性能会受到严重影响。groupByKey 运行示意图如图 3-3 所示。

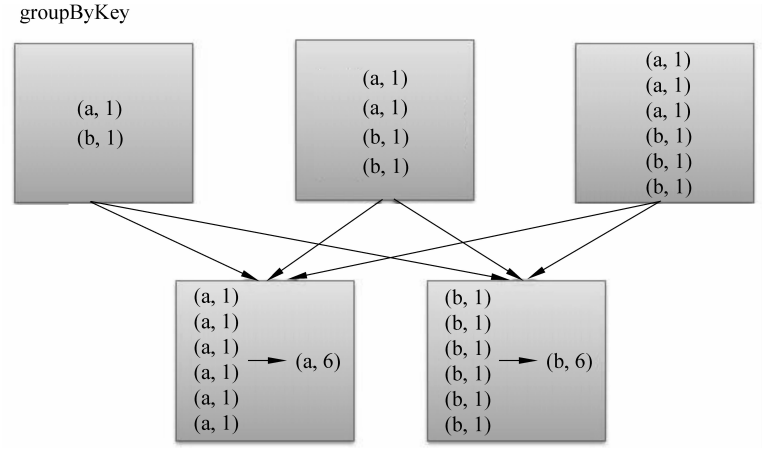

图 3-3　groupByKey 运行示意图

可以尝试的一种优化方法是合并或组合值，因此最终只发送较少的键值对。另外，较少的键值对意味着 Reduce 不会有太多的工作要做，从而带来额外的性能提升。groupByKey()调用不会尝试进行合并或组合值，因此这是一项昂贵的操作。对于一个更大的数据集，洗牌数据量的差异在 reduceByKey()和 groupByKey()之间会变得更加夸张

和不同。以下是比 groupByKey 更优化的方法。

- combineByKey()：可用于组合元素，但返回类型与输入值类型不同。
- foldByKey()：使用关联函数和中性 zeroValue 合并每个键的值。

3.1.2.3 连接

将键值对 RDD 与其他键值对 RDD 进行连接，将数据连接在一起可能是键值对中最常见的操作之一，并且有一系列选项，包括左右外连接、交叉连接和内连接。由于数据框功能的增强，这部分功能也可能通过数据框的 join 操作实现。

简单的 join 运算符是内连接，只输出两键值对 RDD 中共同拥有的键。当在其中一个输入 RDD 中具有相同键和多个值的键值对时，结果键值对 RDD 将具有来自两个输入键值对 RDD 的该键的每个可能的值对，下面的代码可以帮助理解这种操作结果。

```
scala>val employees=sc.parallelize(List((31,"Rafferty"), (33,"Jones"), (33,"
Heisenberg"),(34,"Robinson"), (34,"Smith"), (30,"Williams")))
employees: org.apache.spark.rdd.RDD[(Int, String)]=ParallelCollectionRDD[60] at
parallelize at <console>:24

scala>val departments=sc.parallelize(Array((31, "Sales"), (33,
"Engineering"), (34, "Clerical"),(35, "Marketing")))
departments: org.apache.spark.rdd.RDD[(Int, String)]=ParallelCollectionRDD[61]
at parallelize at <console>:24
scala>departments.join(employees).collect
res66: Array[(Int, (String, String))]=Array((34,(Clerical,Robinson)), (34,
(Clerical,Smith)), (33,(Engineering,Jones)), (33,(Engineering,Heisenberg)),
(31,(Sales,Rafferty)))
```

代码 3-30

有时并不需要结果键值对 RDD 中的键同时出现在两个输入键值对 RDD 中。例如，通过建议加入客户信息，如果没有任何建议，可能不想删除客户信息。leftOuterJoin(other)和 rightOuterJoin(other)都通过键将两个输入键值对 RDD 连接在一起，其中一个 RDD 可能丢掉无法匹配的键，而另一个 RDD 保存了所有的键。

使用 leftOuterJoin，结果 RDD 将保留所有源 RDD 中的每个键。在结果 RDD 中，与每个键关联的值是一个元组，由输入键值对源 RDD 的值以及来自另一输入键值对 RDD 的值 Option 组成。与 join 类似，每个键可以有多个条目；当这种情况发生时，得到两个值列表之间的笛卡儿乘积。rightOuterJoin 与 rightOuterJoin 几乎相同，除了键必须存在于另一个 RDD 中，并且生成的元组中具有 Option 的为源输入键值对 RDD，而不是另一个。下面使用代码 3-30 中的两个输入键值对 departments 和 employees 演示 leftOuterJoin 和 rightOuterJoin 的用法。

```
scala>departments.leftOuterJoin(employees).collect
res67: Array[(Int, (String, Option[String]))] = Array((34, (Clerical, Some
(Robinson))), (34, (Clerical, Some(Smith))), (35, (Marketing, None)), (33,
(Engineering,Some(Jones))), (33,(Engineering,Some(Heisenberg))), (31,(Sales,
Some(Rafferty))))

scala>departments.rightOuterJoin(employees).collect
res68: Array[(Int, (Option[String], String))] = Array((34, (Some(Clerical),
Robinson)), (34, (Some(Clerical), Smith)), (30, (None, Williams)), (33, (Some
(Engineering), Jones)), (33, (Some(Engineering), Heisenberg)), (31, (Some
(Sales),Rafferty)))
```

代码 3-31

- Option、Some 和 None

强大的 Scala 语言可以使用 Option 类，定义函数返回值，其值可能为 null。简单地说，如果函数成功时返回一个对象，而失败时返回 null，那么可以定义函数的返回值为一个 Option 实例，其中 Option 对象是 Some 类的实例或 None 类的实例。因为 Some 和 None 都是 Option 的子项，所有的函数签名只是声明返回一个包含某种类型的 Option（如下面显示的 Int 类型）。至少，这让用户知道发生了什么。以下是使用 Scala Option 语法的示例。

```
def toInt(in: String): Option[Int]={
  try{
      Some(Integer.parseInt(in.trim))
   }catch{
      case e: NumberFormatException =>None
   }
}
```

代码 3-32

以下是 toInt 函数的工作原理：它需要一个 String 作为参数。如果它可以将 String 转换为 Int，那么它将返回 Some(Int)；如果 String 不能转换为 Int，则返回 None。调用此函数的代码如下所示。

```
toInt(someString) match{
    case Some(i) =>println(i)
    case None =>println("That didn't work.")
}
```

代码 3-33

3.1.2.4 排序

对数据进行排序在很多情况下非常有用，特别是在产生后续的输出时。可以使用键

值对 RDD 进行排序，前提是在键上定义了一个排序。一旦对数据进行了排序，对排序后的数据进行 collect()或 save()操作，将导致有序的数据。

sortByKey 函数作用于键值对形式的 RDD 上，并对键进行排序。它是在 org.apache.spark.rdd.OrderedRDDFunctions 中实现的，具体操作如下。

➢ sortByKey（ascending：Boolean = true，numPartitions：Int = self.partitions.length)：RDD[(K，V)]

从函数的实现可以看出，它主要接受两个函数，其含义和 sortBy 一样，这里就不进行解释了。该函数返回的 RDD 一定是 ShuffledRDD 类型，因为对源 RDD 进行排序，必须进行洗牌操作，而洗牌操作的结果 RDD 就是 ShuffledRDD。其实，这个函数的实现很优雅，里面用到了 RangePartitioner，它可以使得相应的范围键数据分到同一个分区中，然后内部用到 mapPartitions 对每个分区中的数据进行排序，而每个分区中数据的排序用到标准的排序机制，避免了大量数据的 shuffle。下面对 sortByKey 的使用进行说明。

```
scala>val a=sc.parallelize(List("wyp", "iteblog", "com", "397090770",
"test"), 2)
a: org.apache.spark.rdd.RDD[String] =
ParallelCollectionRDD[30] at parallelize at <console>:12

scala>val b=sc. parallelize (1 to a.count.toInt, 2)
b: org.apache.spark.rdd.RDD[Int]=ParallelCollectionRDD[31] at parallelize at
<console>:14

scala>val c=a.zip(b)
c: org.apache.spark.rdd.RDD[(String, Int)]=ZippedPartitionsRDD2[32] at zip at
<console>:16

scala>c.sortByKey().collect
res11: Array[(String, Int)]=Array((397090770,4), (com,3), (iteblog,2), (test,
5), (wyp,1))
```

代码 3-34

上面对键进行了排序，sortBy()函数中可以对排序方式进行重写，sortByKey()也有这样的功能。通常在 OrderedRDDFunctions 类中有一个变量 ordering，它是隐式的。

```
private val ordering=implicitly[Ordering[K]]
```

代码 3-35

这就是默认的排序规则，可以对它进行重写，代码如下。

```
scala>val b=sc.parallelize(List(3,1,9,12,4))
b: org.apache.spark.rdd.RDD[Int]=ParallelCollectionRDD[38] at parallelize at
<console>:12
```

```
scala>val c=b.zip(a)
c: org.apache.spark.rdd.RDD[(Int, String)]=ZippedPartitionsRDD2[39] at zip at
<console>:16

scala>c.sortByKey().collect
res15: Array[(Int, String)]=Array((1,iteblog), (3,wyp), (4,test), (9,com),
(12,397090770))

scala>implicit val sortIntegersByString=new Ordering[Int]{
    | override def compare(a: Int, b: Int)=
    | a.toString.compare(b.toString)}
sortIntegersByString: Ordering [Int] = $iwC $$iwC $$iwC $$iwC $$iwC $$anon $1
@5d533f7a

scala>  c.sortByKey().collect
res17: Array[(Int, String)]=Array((1,iteblog), (12,397090770), (3,wyp), (4,
test), (9,com))
```

代码 3-36

例子中的 sortIntegersByString 就是修改了默认顺序的排序规则。这样,将默认顺序按照 Int 的大小排序改成对字符串的排序,所以 12 会排序在 3 之前。

3.1.3 动作

与转换一样,所有在基础 RDD 上提供的传统转换操作也可用在键值对 RDD 上。当然,键值对 RDD 可以使用一些额外的操作,首先创建一个 RDD。

```
scala>val rdd=sc.parallelize(List((1, 2), (3, 4), (3, 6)))
rdd: org.apache.spark.rdd.RDD[(Int, Int)]=ParallelCollectionRDD[15] at
parallelize at <console>:24
```

代码 3-37

➢ countByKey():Map[K,Long]

对每个键进行计数,只有当返回的结果 Map 预计很小时,才使用此方法,因为整个内容都会加载到驱动程序的内存中。要处理非常大的结果,可以考虑使用:

```
rdd.mapValues(_ =>1L).reduceByKey(_ +_)
```

代码 3-38

map Values 将返回 RDD[T,Long],而不是 Map。

```
scala>rdd.countByKey()
res74: scala.collection.Map[Int,Long]=Map(1 ->1, 3 ->2)
```

代码 3-39

➢ collectAsMap()：Map[K，V]

此函数与 collect()类似，但对关键值 RDD 起作用并将其转换为 Scala Map，以保留其键值结构，如果键值对 RDD 中同一个键有多个值，则每个键中只有一个值会保留在返回的 Map 中。因为所有的数据都加载到驱动程序的内存中，所以只有在结果数据很小时才使用此方法。

```
scala>rdd.collectAsMap()
res80: scala.collection.Map[Int,Int]=Map(1 ->2, 3 ->6)
```

代码 3-40

➢ lookup(key：K)：Seq[V]

返回与提供键关联的所有值。如果 RDD 具有已知的分区程序，则只搜索该键映射到的分区即可高效地执行此操作。

```
scala>rdd.lookup(3)
res91: Seq[Int]=WrappedArray(4, 6)
```

代码 3-41

3.2 分区和洗牌

我们已经了解了 Apache Spark 如何比 Hadoop 更好地处理分布式计算，还看到了内部工作原理主要是称为弹性分布式数据集的基本数据结构。RDD 是代表数据集的不可变集合，并具有可靠性和故障恢复的内在能力。实际上，RDD 对数据的操作不是基于整个数据块，数据分布于整个集群的分区中，通过 RDD 的抽象层管理和操作数据。因此，数据分区的概念对于 Apache Spark 作业的正常运行至关重要，并且会对性能产生很大影响，决定了资源的利用情况。本节将深入讨论分区和洗牌的概念。

RDD 由数据分区组成，基于 RDD 的所有操作都在数据分区上执行，诸如转换之类的几种操作是在执行器上运行的函数，特定的数据分区也在此执行器上。但是，并非所有操作过程都可以仅由所在的执行器包含的数据分区孤立完成，像聚合这样的操作要求将数据跨节点移到整个集群中。在下面的简单整数 RDD 的操作中，SparkContext 的 parallelize()函数根据整数序列创建 RDD，然后使用 getNumPartitions()函数可以获得该 RDD 的分区数。

```
scala>val rdd_one=sc.parallelize(Seq(1,2,3))
rdd_one: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at
parallelize at <console>:24

scala>rdd_one.getNumPartitions
res0: Int=24
```

分区的数量很重要，因为该数量直接影响将要运行 RDD 转换的任务数量。如果分区的数量太少，那么大量数据可能仅使用几个 CPU 内核，从而降低性能，并使集群利用率不足。另一方面，如果分区的数量太大，那么将使用比实际需要更多的资源，并且在多用户的环境中，可能导致正在运行的其他作业的资源匮乏。如果要查看 CPU 的核数，可以使用下面的命令：

```
root@bb8bf6efccc9:~#lscpu | egrep 'CPU\(s\)'
CPU(s):                24
On-line CPU(s) list:   0-23
NUMA node0 CPU(s):     0-11
NUMA node1 CPU(s):     12-23
```

3.2.1 分区

Spark 的分区是存储在集群中节点上的原始数据块，即逻辑划分。RDD 是这种分区的集合，通过 RDD 的抽象概念隐藏了正在处理的分段数据。这种分区结构可帮助 Spark 实现并行化分布式数据处理，并以最小的网络流量在执行程序之间发送数据。

分区的数量对于一个良好的集群性能来说非常重要。如果有很少的分区，将不能充分利用集群中的内存和 CPU 资源，因为某些资源可能处于空闲状态。例如，假设有一个 Spark 集群具有 10 个 CPU 内核，一般来说，一个 CPU 内核负责一个分区的计算，在这种情况下如果有少于 10 个分区，那么一些 CPU 内核将处于空闲状态，所以会浪费资源。此外，由于分区较少，每个分区中就会有更多的数据，这样会造成集群中某些节点内存增加的压力。另一方面，如果有太多的分区，那么每个分区可能具有太少的数据或根本没有数据，也可能降低性能，因为集群中的数据分区可能是跨节点的，从多个节点上汇总分区中的数据需要更多的计算和传输时间。因此，根据 Spark 集群配置情况设置合适的分区非常重要。Spark 只能一次为 RDD 的每个分区分配运行一个并发任务，一次最多的并发任务为集群中的最大 CPU 内核数。所以，如果有一个 10 核 CPU 的集群，那么至少要为 RDD 定义 10 个分区，分区总数一般为内核的 2～4 倍。默认情况下，Spark 会创建等于集群中 CPU 内核数的分区数，也可以随时更改分区数。下面的例子创建了具有指定分区数的 RDD。

```
scala>val names=Seq("Steve","Andrew","Bob","John","Quinton")
names: Seq[String]=List(Steve, Andrew, Bob, John, Quinton)

scala>val regularRDD=sc.parallelize(names)
regularRDD: org.apache.spark.rdd.RDD[String]=ParallelCollectionRDD[114] at
parallelize at <console>:27

scala>regularRDD.partitions.size
res32: Int=24
```

```
scala>val regularRDD=sc.parallelize(names,48)
regularRDD: org.apache.spark.rdd.RDD[String]=ParallelCollectionRDD[116] at
parallelize at <console>:27

scala>regularRDD.partitions.size
res100: Int=48
```

代码 3-42

正如代码中看到的，regularRDD 的默认分区数量等于 24，这是由于当前环境是通过本地模式启动的 spark-shell，本地模式是在具有 24 核 CPU 的 Docker 虚拟实验环境中运行。如果在创建 RDD 时指定了分区数 48，regularRDD 的分区就变成 48。在创建 RDD 时，第二个参数定义了为该 RDD 创建的分区数。一个分区从不跨越多台机器，即同一分区中的所有元组都保证在同一台机器上。集群中的每个工作节点都可以包含一个或多个 RDD 的分区。分区总数是可配置的，默认情况下，它等于所有执行器节点上的内核总数。

Spark 提供了两个内置分区器，分别是哈希分区器和范围分区器。创建 RDD 时，可以通过两种方式指定特定的分区器：一种方式是通过在 RDD 上调用 partitionBy()方法提供显式指定的分区器；另一种方式是通过转换操作返回新创建的 RDD，其使用转换操作特定的分区器。带有分区器的转换操作有 join()、leftOuterJoin()、rightOuterJoin()、groupByKey()、reduceByKey()、cogroup()、foldByKey()、combineByKey()、sort()、partitionBy()、groupWith()；另外，mapValues()、flatMapValues()和 filter()的分区方式与父级 RDD 有关。而像 map()这样的操作会导致新创建的 RDD 忘记父分区信息，因为像这样的操作理论上可以修改每个记录的键，所以，在这种情况下如果操作在结果 RDD 中保留了分区器，则不再有任何意义，因为现在的键都是不同的。所以，Spark 提供像 mapValues()和 flatMapValues()这样的操作，如果不想改变键，可以使用这些操作，从而保留分区器。partitionBy()是一个转化操作，因此它的返回值总是一个新的 RDD，但它不会改变原来的 RDD。RDD 一旦创建，就无法修改，因此应该对 partitionBy()的结果进行持久化。如果没有将 partitionBy() 转化操作的结果持久化，那么后面每次用到这个 RDD 时，都会重复对数据进行分区操作。不进行持久化会导致整个 RDD 谱系图重新求值。那样的话，partitionBy() 带来的好处就会被抵消，导致重复对数据进行分区以及跨节点的混洗，和没有指定分区方式时发生的情况十分相似。

哈希分区是 Spark 中的默认分区程序，通过计算 RDD 元组中每个键的哈希值工作，具有相同哈希码的元素最终都位于相同的分区中，如以下代码片段所示。

```
partitionIndex=hashcode(key) %numPartitions
```

如果键相同，则其 hashcode 的结果相同，其对应的值保存在相同的分区上。哈希分区是 Spark 的默认分区器。如果没有提到任何分区器，那么 Spark 将使用哈希分区器对数据进行分区。下面的例子便于更好地理解以上内容。