Chapter 4. Working with Key/Value Pairs

This chapter covers how to work with RDDs of key/value pairs, which are a common data type required for many operations in Spark. Key/value RDDs are commonly used to perform aggregations, and often we will do some initial ETL (extract, transform, and load) to get our data into a key/value format. Key/value RDDs expose new operations (e.g., counting up reviews for each product, grouping together data with the same key, and grouping together two different RDDs).

We also discuss an advanced feature that lets users control the layout of pair RDDs across nodes: *partitioning*. Using controllable partitioning, applications can sometimes greatly reduce communication costs by ensuring that data will be accessed together and will be on the same node. This can provide significant speedups. We illustrate partitioning using the PageRank algorithm as an example. Choosing the right partitioning for a distributed dataset is similar to choosing the right data structure for a local one—in both cases, data layout can greatly affect performance.

Motivation

Spark provides special operations on RDDs containing key/value pairs. These RDDs are called pair RDDs. Pair RDDs are a useful building block in many programs, as they expose operations that allow you to act on each key in parallel or regroup data across the network. For example, pair RDDs have a `reduceByKey()` method that can aggregate data separately for each key, and a `join()` method that can merge two RDDs together by grouping elements with the same key. It is common to extract fields from an RDD (representing, for instance, an event time, customer ID, or other identifier) and use those fields as keys in pair RDD operations.

Creating Pair RDDs

There are a number of ways to get pair RDDs in Spark. Many formats we explore loading from in Chapter 5 will directly return pair RDDs for their key/value data. In other cases we have a regular RDD that we want to turn into a pair RDD. We can do this by running a `map()` function that returns key/value pairs. To illustrate, we show code that starts with an RDD of lines of text and keys the data by the first word in each line.

The way to build key-value RDDs differs by language. In Python, for the functions on keyed data to work we need to return an RDD composed of tuples (see Example 4-1).

Example 4-1. Creating a pair RDD using the first word as the key in Python

```
pairs = lines.map(lambda x: (x.split(" ")[0], x))
```

In Scala, for the functions on keyed data to be available, we also need to return tuples (see Example 4-2). An implicit conversion on RDDs of tuples exists to provide the additional key/value functions.

Example 4-2. Creating a pair RDD using the first word as the key in Scala

```
val pairs = lines.map(x => (x.split(" ")(0), x))
```

Java doesn't have a built-in tuple type, so Spark's Java API has users create tuples using the `scala.Tuple2` class. This class is very simple: Java users can construct a new tuple by writing `new Tuple2(elem1, elem2)` and can then access its elements with the `._1()` and `._2()` methods.

Java users also need to call special versions of Spark's functions when creating pair RDDs. For instance, the `mapToPair()` function should be used in place of the basic `map()` function. This is discussed in more detail in "Java", but let's look at a simple case in Example 4-3.

Example 4-3. Creating a pair RDD using the first word as the key in Java

```
PairFunction<String, String, String> keyData =
  new PairFunction<String, String, String>() {
  public Tuple2<String, String> call(String x) {
    return new Tuple2(x.split(" ")[0], x);
  }
};
JavaPairRDD<String, String> pairs = lines.mapToPair(keyData);
```

When creating a pair RDD from an in-memory collection in Scala and Python, we only need to call `SparkContext.parallelize()` on a collection of pairs. To create a pair RDD in Java from an in-memory collection, we instead use `SparkContext.parallelizePairs()`.

Transformations on Pair RDDs

Pair RDDs are allowed to use all the transformations available to standard RDDs.

`combineByKey(createCombiner, mergeValue, mergeCombiners, partitioner)`	Combine values with the same key using a different result type.	See Examples 4-12 through 4-14.
`mapValues(func)`	Apply a function to each value of a pair RDD without changing the key.	`rdd.mapValues(x => x+1)`
`flatMapValues(func)`	Apply a function that returns an iterator to each value of a pair RDD, and for each element returned, produce a key/value entry with the old key. Often used for tokenization.	`rdd.flatMapValues(x => (x to 5)`
`keys()`	Return an RDD of just the keys.	`rdd.keys()`
`values()`	Return an RDD of just the values.	`rdd.values()`
`sortByKey()`	Return an RDD sorted by the key.	`rdd.sortByKey()`

Table 4-2. Transformations on two pair RDDs (rdd = {(1, 2), (3, 4), (3, 6)} other = {(3, 9)})

Function name	Purpose	Example	Resu
`subtractByKey`	Remove elements with a key	`rdd.subtractByKey(other)`	`{(1,`

	RDD.		
`leftOuterJoin`	Perform a join between two RDDs where the key must be present in the other RDD.	`rdd.leftOuterJoin(other)`	`{(1,` `(2,No` `(3,` `(4,So` `(3,` `(6,So`
`cogroup`	Group data from both RDDs sharing the same key.	`rdd.cogroup(other)`	`{(1,(` `[])),` `6],[9`

We discuss each of these families of pair RDD functions in more detail in the upcoming sections.

Pair RDDs are also still RDDs (of Tuple2 objects in Java/Scala or of Python tuples), and thus support the same functions as RDDs. For instance, we can take our pair RDD from the previous section and filter out lines longer than 20 characters, as shown in Examples 4-4 through 4-6 and Figure 4-1.

Example 4-4. Simple filter on second element in Python

```python
result = pairs.filter(lambda keyValue: len(keyValue[1]) < 20)
```

Example 4-5. Simple filter on second element in Scala

```scala
pairs.filter{case (key, value) => value.length < 20}
```

Example 4-6. Simple filter on second element in Java

```java
Function<Tuple2<String, String>, Boolean> longWordFilter =
  new Function<Tuple2<String, String>, Boolean>() {
    public Boolean call(Tuple2<String, String> keyValue) {
      return (keyValue._2().length() < 20);
    }
  };
JavaPairRDD<String, String> result = pairs.filter(longWordFilter
```

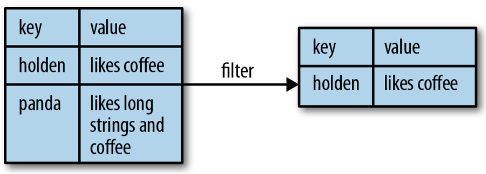


Figure 4-1. Filter on value

Sometimes working with pairs can be awkward if we want to access only the value part of our pair RDD. Since this is a common pattern, Spark provides the `mapValues(func)` function, which is the same as `map{case (x, y): (x, func(y))}`. We will use this function in many of our examples.

We now discuss each of the families of pair RDD functions, starting with aggregations.

Aggregations

When datasets are described in terms of key/value pairs, it is common to want to aggregate statistics across all elements with the same key. We have looked at the `fold()`, `combine()`, and `reduce()` actions on basic RDDs, and similar per-key transformations exist on pair RDDs. Spark has a similar set of operations that combines values that have the same key. These operations return RDDs and thus are transformations rather than actions.

```
rdd.mapValues(x => (x, 1)).reduceByKey((x, y) => (x._1 + y._1,
```

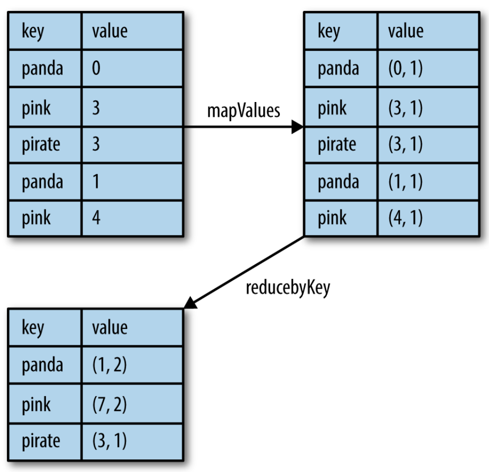


Figure 4-2. Per-key average data flow

TIP

Those familiar with the combiner concept from MapReduce should note that calling `reduceByKey()` and `foldByKey()` will automatically perform combining locally on each machine before computing global totals for each key. The user does not need to specify a combiner. The more general `combineByKey()` interface allows you to customize combining behavior.

We can use a similar approach in Examples 4-9 through 4-11 to also implement the classic distributed word count problem. We will use `flatMap()` from the previous chapter so that we can produce a pair RDD of words and the number 1 and then sum together all of the words using `reduceByKey()` as in Examples 4-7 and 4-8.

Example 4-9. Word count in Python

```
rdd = sc.textFile("s3://...")
words = rdd.flatMap(lambda x: x.split(" "))
result = words.map(lambda x: (x, 1)).reduceByKey(lambda x, y:
```

Example 4-10. Word count in Scala

```
val input = sc.textFile("s3://...")
val words = input.flatMap(x => x.split(" "))
val result = words.map(x => (x, 1)).reduceByKey((x, y) => x +
```

Example 4-11. Word count in Java

```
JavaRDD<String> input = sc.textFile("s3://...")
JavaRDD<String> words = rdd.flatMap(new FlatMapFunction<String,
  public Iterable<String> call(String x) { return Arrays.asLi
});
JavaPairRDD<String, Integer> result = words.mapToPair(
  new PairFunction<String, String, Integer>() {
    public Tuple2<String, Integer> call(String x) { return ne
}).reduceByKey(
  new Function2<Integer, Integer, Integer>() {
    public Integer call(Integer a, Integer b) { return a + b;
});
```

TIP

We can actually implement word count even faster by using the `countByValue()` function on the first RDD: `input.flatMap(x => x.split(" ")).countByValue()`.

TIP

We can disable map-side aggregation in `combineByKey()` if we know that our data won't benefit from it. For example, `groupByKey()` disables map-side aggregation as the aggregation function (appending to a list) does not save any space. If we want to disable map-side combines, we need to specify the partitioner; for now you can just use the partitioner on the source RDD by passing `rdd.partitioner`.

Since `combineByKey()` has a lot of different parameters it is a great candidate for an explanatory example. To better illustrate how `combineByKey()` works, we will look at computing the average value for each key, as shown in Examples 4-12 through 4-14 and illustrated in Figure 4-3.

Example 4-12. Per-key average using combineByKey() in Python

```
sumCount = nums.combineByKey((lambda x: (x,1)),
                             (lambda x, y: (x[0] + y, x[1] + 1)
                             (lambda x, y: (x[0] + y[0], x[1] +
sumCount.map(lambda key, xy: (key, xy[0]/xy[1])).collectAsMap()
```

Example 4-13. Per-key average using combineByKey() in Scala

```
val result = input.combineByKey(
  (v) => (v, 1),
  (acc: (Int, Int), v) => (acc._1 + v, acc._2 + 1),
  (acc1: (Int, Int), acc2: (Int, Int)) => (acc1._1 + acc2._1
  ).map{ case (key, value) => (key, value._1 / value._2.toFloat
  result.collectAsMap().map(println(_))
```

Example 4-14. Per-key average using combineByKey() in Java

```
public static class AvgCount implements Serializable {
  public AvgCount(int total, int num) {
          total_ = total;
          num_ = num; }
  public int total_;
  public int num_;
  public float avg() {
          return total_ / (float) num_; }
}

Function<Integer, AvgCount> createAcc = new Function<Integer, A
  public AvgCount call(Integer x) {
    return new AvgCount(x, 1);
  }
};
Function2<AvgCount, Integer, AvgCount> addAndCount =
  new Function2<AvgCount, Integer, AvgCount>() {
  public AvgCount call(AvgCount a, Integer x) {
    a.total_ += x;
    a.num_ += 1;
    return a;
  }
};
Function2<AvgCount, AvgCount, AvgCount> combine =
  new Function2<AvgCount, AvgCount, AvgCount>() {
  public AvgCount call(AvgCount a, AvgCount b) {
    a.total_ += b.total_;
    a.num_ += b.num_;
    return a;
  }
};
AvgCount initial = new AvgCount(0,0);
JavaPairRDD<String, AvgCount> avgCounts =
  nums.combineByKey(createAcc, addAndCount, combine);
Map<String, AvgCount> countMap = avgCounts.collectAsMap();
for (Entry<String, AvgCount> entry : countMap.entrySet()) {
  System.out.println(entry.getKey() + ":" + entry.getValue().avg
}
```

ing one of the specialized aggregation functions in Spark can be much faster than the naive approach of grouping our data and then reducing it.

TUNING THE LEVEL OF PARALLELISM

So far we have talked about how all of our transformations are distributed, but we have not really looked at how Spark decides how to split up the work. Every RDD has a fixed number of *partitions* that determine the degree of parallelism to use when executing operations on the RDD.

When performing aggregations or grouping operations, we can ask Spark to use a specific number of partitions. Spark will always try to infer a sensible default value based on the size of your cluster, but in some cases you will want to tune the level of parallelism for better performance.

Most of the operators discussed in this chapter accept a second parameter giving the number of partitions to use when creating the grouped or aggregated RDD, as shown in Examples 4-15 and 4-16.

Example 4-15. reduceByKey() with custom parallelism in Python

```
data = [("a", 3), ("b", 4), ("a", 1)]
sc.parallelize(data).reduceByKey(lambda x, y: x + y)      # De
sc.parallelize(data).reduceByKey(lambda x, y: x + y, 10)  # Cu
```

Example 4-16. reduceByKey() with custom parallelism in Scala

```
val data = Seq(("a", 3), ("b", 4), ("a", 1))
sc.parallelize(data).reduceByKey((x, y) => x + y)    // Default
sc.parallelize(data).reduceByKey((x, y) => x + y)    // Custom
```

Sometimes, we want to change the partitioning of an RDD outside the context of grouping and aggregation operations. For those cases, Spark provides the `repartition()` function, which shuffles the data across the network to create a new set of partitions. Keep in mind that repartitioning your data is a fairly expensive operation. Spark also has an optimized version of `repartition()` called `coalesce()` that allows avoiding data movement, but only if you are decreasing the number of RDD partitions. To know whether you can safely call `coalesce()`, you can check the size of the RDD using `rdd.partitions.size()` in Java/Scala and `rdd.getNumPartitions()` in Python and make sure that you are coalescing it to fewer partitions than it currently has.

Grouping Data

With keyed data a common use case is grouping our data by key—for example, viewing all of a customer's orders together.

If our data is already keyed in the way we want, `groupByKey()` will group our data using the key in our RDD. On an RDD consisting of keys of type `K` and values of type `V`, we get back an RDD of type `[K, Iterable[V]]`.

`groupBy()` works on unpaired data or data where we want to use a different condition besides equality on the current key. It takes a function that applies to every element in the source RDD and uses the result to determine the key.

TIP

If you find yourself writing code where you `groupByKey()` and then use a `reduce()` or `fold()` on the values, you can probably achieve the same result more efficiently by using one of the per-key aggregation functions. Rather than reducing the RDD to an in-memory value, we reduce the data per key and get back an RDD with the reduced values corresponding to each key. For example, `rdd.reduceByKey(func)` produces the same RDD as `rdd.groupByKey().mapValues(value => value.reduce(func))` but is more efficient as it avoids the step of creating a list of values for each key.

In addition to grouping data from a single RDD, we can group data sharing the same key from multiple RDDs using a function called `cogroup()`. `cogroup()` over two RDDs sharing the same key type, `K`, with the respective value types `V` and `W` gives us back `RDD[(K, (Iterable[V], Iterable[W]))]`. If one of the RDDs doesn't have elements for a given key that is present in the other RDD, the corresponding `Iterable` is simply empty. `cogroup()` gives us the power to group data from multiple RDDs.

`cogroup()` is used as a building block for the joins we discuss in the next section.

```
storeAddress = {
  (Store("Ritual"), "1026 Valencia St"), (Store("Philz"), "748
  (Store("Philz"), "3101 24th St"), (Store("Starbucks"), "Seat

storeRating = {
  (Store("Ritual"), 4.9), (Store("Philz"), 4.8))}

storeAddress.join(storeRating) == {
  (Store("Ritual"), ("1026 Valencia St", 4.9)),
  (Store("Philz"), ("748 Van Ness Ave", 4.8)),
  (Store("Philz"), ("3101 24th St", 4.8))}
```

Sometimes we don't need the key to be present in both RDDs to want it in our result. For example, if we were joining customer information with recommendations we might not want to drop customers if there were not any recommendations yet. `leftOuterJoin(other)` and `rightOuterJoin(other)` both join pair RDDs together by key, where one of the pair RDDs can be missing the key.

With `leftOuterJoin()` the resulting pair RDD has entries for each key in the source RDD. The value associated with each key in the result is a tuple of the value from the source RDD and an `Option` (or `Optional` in Java) for the value from the other pair RDD. In Python, if a value isn't present `None` is used; and if the value is present the regular value, without any wrapper, is used. As with `join()`, we can have multiple entries for each key; when this occurs, we get the Cartesian product between the two lists of values.

TIP

`Optional` is part of Google's Guava library and represents a possibly missing value. We can check `isPresent()` to see if it's set, and `get()` will return the contained instance provided data is present.

`rightOuterJoin()` is almost identical to `leftOuterJoin()` except the key must be present in the other RDD and the tuple has an option for the source rather than the other RDD.

We can revisit Example 4-17 and do a `leftOuterJoin()` and a `rightOuterJoin()` between the two pair RDDs we used to illustrate `join()` in Example 4-18.

Example 4-18. leftOuterJoin() and rightOuterJoin()

```
storeAddress.leftOuterJoin(storeRating) ==
{(Store("Ritual"),("1026 Valencia St",Some(4.9))),
  (Store("Starbucks"),("Seattle",None)),
  (Store("Philz"),("748 Van Ness Ave",Some(4.8))),
  (Store("Philz"),("3101 24th St",Some(4.8)))}

storeAddress.rightOuterJoin(storeRating) ==
{(Store("Ritual"),(Some("1026 Valencia St"),4.9)),
  (Store("Philz"),(Some("748 Van Ness Ave"),4.8)),
  (Store("Philz"), (Some("3101 24th St"),4.8))}
```

Sorting Data

Having sorted data is quite useful in many cases, especially when you're producing downstream output. We can sort an RDD with key/value pairs provided that there is an ordering defined on the key. Once we have sorted our data, any subsequent call on the sorted data to `collect()` or `save()` will result in ordered data.

Since we often want our RDDs in the reverse order, the `sortByKey()` function takes a parameter called `ascending` indicating whether we want it in ascending order (it defaults to `true`). Sometimes we want a different sort order entirely, and to support this we can provide our own comparison function. In Examples 4-19 through 4-21, we will sort our RDD by converting the integers to strings and using the string comparison functions.

Example 4-19. Custom sort order in Python, sorting integers as if strings

```
rdd.sortByKey(ascending=True, numPartitions=None, keyfunc = lam
```

Example 4-20. Custom sort order in Scala, sorting integers as if strings

```
val input: RDD[(Int, Venue)] = ...
implicit val sortIntegersByString = new Ordering[Int] {
  override def compare(a: Int, b: Int) = a.toString.compare
}
rdd.sortByKey()
```

	elements for each key.		2)}
`collectAsMap()`	Collect the result as a map to provide easy lookup.	`rdd.collectAsMap()`	`Map{(1, 2), (3, 4), (3, 6)}`
`lookup(key)`	Return all values associated with the provided key.	`rdd.lookup(3)`	`[4, 6]`

There are also multiple other actions on pair RDDs that save the RDD, which we will describe in Chapter 5.

Data Partitioning (Advanced)

The final Spark feature we will discuss in this chapter is how to control datasets' partitioning across nodes. In a distributed program, communication is very expensive, so laying out data to minimize network traffic can greatly improve performance. Much like how a single-node program needs to choose the right data structure for a collection of records, Spark programs can choose to control their RDDs' partitioning to reduce communication. Partitioning will not be helpful in all applications—for example, if a given RDD is scanned only once, there is no point in partitioning it in advance. It is useful only when a dataset is reused *multiple times* in key-oriented operations such as joins. We will give some examples shortly.

Spark's partitioning is available on all RDDs of key/value pairs, and causes the system to group elements based on a function of each key. Although Spark does not give explicit control of which worker node each key goes to (partly because the system is designed to work even if specific nodes fail), it lets the program ensure that a *set* of keys will appear together on *some* node. For example, you might choose to hash-partition an RDD into 100 partitions so that keys that have the same hash value modulo 100 appear on the same node. Or you might range-partition the RDD into sorted ranges of keys so that elements with keys in the same range appear on the same node.

As a simple example, consider an application that keeps a large table of user information in memory—say, an RDD of `(UserID, UserInfo)` pairs, where `UserInfo` contains a list of topics the user is subscribed to. The application periodically combines this table with a smaller file representing events that happened in the past five minutes—say, a table of `(UserID, LinkInfo)` pairs for users who have clicked a link on a website in those five minutes. For example, we may wish to count how many users visited a link that was *not* to one of their subscribed topics. We can perform this combination with Spark's `join()` operation, which can be used to group the `UserInfo` and `LinkInfo` pairs for each `UserID` by key. Our application would look like Example 4-22.

Example 4-22. Scala simple application

```scala
// Initialization code; we load the user info from a Hadoop Sequ
// This distributes elements of userData by the HDFS block where
// and doesn't provide Spark with any way of knowing in which pa
// particular UserID is located.
val sc = new SparkContext(...)
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...

// Function called periodically to process a logfile of events i
// we assume that this is a SequenceFile containing (UserID, Lin
def processNewLogs(logFileName: String) {
  val events = sc.sequenceFile[UserID, LinkInfo](logFileName
  val joined = userData.join(events)// RDD of (UserID, (UserInf
  val offTopicVisits = joined.filter {
    case (userId, (userInfo, linkInfo)) => // Expand the tuple
      !userInfo.topics.contains(linkInfo.topic)
  }.count()
  println("Number of visits to non-subscribed topics: " + offTop
}
```

This code will run fine as is, but it will be inefficient. This is because the `join()` operation, called each time `processNewLogs()` is invoked, does not know anything about how the keys are partitioned in the datasets. By default, this operation will hash all the keys of *both* datasets, sending elements with the same key hash across the network to the same machine, and then join together the elements with the same key on that machine (see Figure 4-4). Because we expect the `userData` table to be much larger than the small log of `events` seen every five minutes, this wastes a lot

Figure 4-4. Each join of userData and events without using partitionBy()

Fixing this is simple: just use the `partitionBy()` transformation on `userData` to hash-partition it at the start of the program. We do this by passing a `spark.HashPartitioner` object to `partitionBy`, as shown in Example 4-23.

Example 4-23. Scala custom partitioner

```
val sc = new SparkContext(...)
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...
                 .partitionBy(new HashPartitioner(100))   //
                 .persist()
```

The `processNewLogs()` method can remain unchanged: the `events` RDD is local to `processNewLogs()`, and is used only once within this method, so there is no advantage in specifying a partitioner for `events`. Because we called `partitionBy()` when building `userData`, Spark will now know that it is hash-partitioned, and calls to `join()` on it will take advantage of this information. In particular, when we call `userData.join(events)`, Spark will shuffle only the `events` RDD, sending events with each particular `UserID` to the machine that contains the corresponding hash partition of `userData` (see Figure 4-5). The result is that a lot less data is communicated over the network, and the program runs significantly faster.

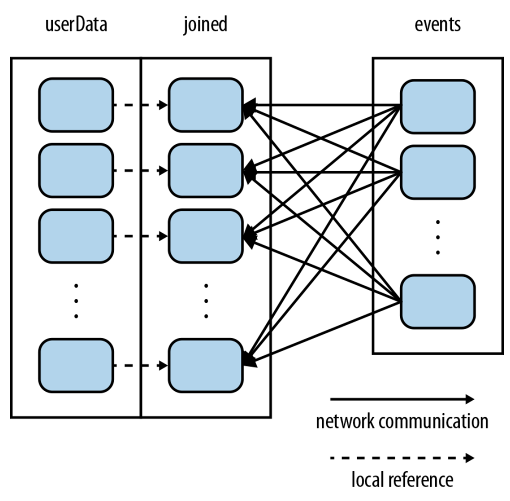


Figure 4-5. Each join of userData and events using partitionBy()

Note that `partitionBy()` is a transformation, so it always returns a *new* RDD—it does not change the original RDD in place. RDDs can never be modified once created. Therefore it is important to persist and save as `userData` the result of `partitionBy()`, not the original `sequenceFile()`. Also, the 100 passed to `partitionBy()` represents the number of partitions, which will control how many parallel tasks perform further operations on the RDD (e.g., joins); in general, make this at least as large as the number of cores in your cluster.

WARNING

Failure to persist an RDD after it has been transformed with `partitionBy()` will cause subsequent uses of the RDD to repeat the partitioning of the data. Without persistence, use of the partitioned RDD will cause reevaluation of the RDDs complete lineage. That would negate the advantage of `partitionBy()`, resulting in repeated partitioning and shuffling of data across the network, similar to what occurs without any specified partitioner.

In fact, many other Spark operations automatically result in an RDD with known partitioning information, and many operations other than `join()` will take advantage of this information. For example, `sortByKey()` and `groupByKey()` will result in range-partitioned and hash-partitioned RDDs, respectively. On the other hand, operations like `map()` cause the new RDD to *forget* the parent's partitioning information, because such operations could theoretically modify the key of each record. The next few sections describe how to determine how an RDD is partitioned, and exactly how partitioning affects the various Spark operations.

Spark operations affect partitioning, and to check that the operations you want to do in your program will yield the right result (see Example 4-24).

Example 4-24. Determining partitioner of an RDD

```
scala> val pairs = sc.parallelize(List((1, 1), (2, 2), (3, 3))
pairs: spark.RDD[(Int, Int)] = ParallelCollectionRDD[0]

scala> pairs.partitioner
res0: Option[spark.Partitioner] = None

scala> val partitioned = pairs.partitionBy(new spark.HashPart
partitioned: spark.RDD[(Int, Int)] = ShuffledRDD[1] at par

scala> partitioned.partitioner
res1: Option[spark.Partitioner] = Some(spark.HashPartit
```

In this short session, we created an RDD of `(Int, Int)` pairs, which initially have no partitioning information (an `Option` with value `None`). We then created a second RDD by hash-partitioning the first. If we actually wanted to use `partitioned` in further operations, then we should have appended `persist()` to the third line of input, in which `partitioned` is defined. This is for the same reason that we needed `persist()` for `userData` in the previous example: without `persist()`, subsequent RDD actions will evaluate the entire lineage of `partitioned`, which will cause `pairs` to be hash-partitioned over and over.

Operations That Benefit from Partitioning

Many of Spark's operations involve shuffling data by key across the network. All of these will benefit from partitioning. As of Spark 1.0, the operations that benefit from partitioning are `cogroup()`, `groupWith()`, `join()`, `leftOuterJoin()`, `rightOuterJoin()`, `groupByKey()`, `reduceByKey()`, `combineByKey()`, and `lookup()`.

For operations that act on a single RDD, such as `reduceByKey()`, running on a prepartitioned RDD will cause all the values for each key to be computed *locally* on a single machine, requiring only the final, locally reduced value to be sent from each worker node back to the master. For binary operations, such as `cogroup()` and `join()`, pre-partitioning will cause at least one of the RDDs (the one with the known partitioner) to not be shuffled. If both RDDs have the *same* partitioner, and if they are cached on the same machines (e.g., one was created using `mapValues()` on the other, which preserves keys and partitioning) or if one of them has not yet been computed, then no shuffling across the network will occur.

Operations That Affect Partitioning

Spark knows internally how each of its operations affects partitioning, and automatically sets the `partitioner` on RDDs created by operations that partition the data. For example, suppose you called `join()` to join two RDDs; because the elements with the same key have been hashed to the same machine, Spark knows that the result is hash-partitioned, and operations like `reduceByKey()` on the join result are going to be significantly faster.

The flipside, however, is that for transformations that *cannot* be guaranteed to produce a known partitioning, the output RDD will not have a `partitioner` set. For example, if you call `map()` on a hash-partitioned RDD of key/value pairs, the function passed to `map()` can in theory change the key of each element, so the result will not have a `partitioner`. Spark does not analyze your functions to check whether they retain the key. Instead, it provides two other operations, `mapValues()` and `flatMapValues()`, which guarantee that each tuple's key remains the same.

All that said, here are all the operations that result in a partitioner being set on the output RDD: `cogroup()`, `groupWith()`, `join()`, `leftOuterJoin()`, `rightOuterJoin()`, `groupByKey()`, `reduceByKey()`, `combineByKey()`, `partitionBy()`, `sort()`, `mapValues()` (if the parent RDD has a partitioner), `flatMapValues()` (if parent has a partitioner), and `filter()` (if parent has a partitioner). All *other* operations will produce a result with no partitioner.

Finally, for binary operations, *which* partitioner is set on the output depends on the parent RDDs' partitioners. By default, it is a hash partitioner, with the number of partitions set to the level of parallelism of the operation. However, if one of the parents has a `partitioner` set, it will be that partitioner; and if both parents have a `partitioner` set, it will be the partitioner of the first parent.

Example: PageRank

As an example of a more involved algorithm that can benefit from RDD partitioning, we consider PageRank. The PageRank algorithm, named after Google's Larry Page, aims to assign a measure of importance (a "rank") to each document in a set based on how many documents have links to it. It can be used to rank web pages, of course, but also scientific articles, or influential users in a social network.

PageRank is an iterative algorithm that performs many joins, so it is a good use case

```
// Initialize each page's rank to 1.0; since we use mapValues, t
// will have the same partitioner as links
var ranks = links.mapValues(v => 1.0)

// Run 10 iterations of PageRank
for (i <- 0 until 10) {
  val contributions = links.join(ranks).flatMap {
    case (pageId, (links, rank)) =>
      links.map(dest => (dest, rank / links.size))
  }
  ranks = contributions.reduceByKey((x, y) => x + y).mapValues(
}

// Write out the final ranks
ranks.saveAsTextFile("ranks")
```

That's it! The algorithm starts with a `ranks` RDD initialized at 1.0 for each element, and keeps updating the `ranks` variable on each iteration. The body of PageRank is pretty simple to express in Spark: it first does a `join()` between the current `ranks` RDD and the static `links` one, in order to obtain the link list and rank for each page ID together, then uses this in a `flatMap` to create "contribution" values to send to each of the page's neighbors. We then add up these values by page ID (i.e., by the page receiving the contribution) and set that page's rank to `0.15 + 0.85 * contributionsReceived`.

Although the code itself is simple, the example does several things to ensure that the RDDs are partitioned in an efficient way, and to minimize communication:

1. Notice that the `links` RDD is joined against `ranks` on each iteration. Since `links` is a static dataset, we partition it at the start with `partitionBy()`, so that it does not need to be shuffled across the network. In practice, the `links` RDD is also likely to be much larger in terms of bytes than `ranks`, since it contains a list of neighbors for each page ID instead of just a `Double`, so this optimization saves considerable network traffic over a simple implementation of PageRank (e.g., in plain MapReduce).

2. For the same reason, we call `persist()` on `links` to keep it in RAM across iterations.

3. When we first create `ranks`, we use `mapValues()` instead of `map()` to preserve the partitioning of the parent RDD (`links`), so that our first join against it is cheap.

4. In the loop body, we follow our `reduceByKey()` with `mapValues()`; because the result of `reduceByKey()` is already hash-partitioned, this will make it more efficient to join the mapped result against `links` on the next iteration.

TIP

To maximize the potential for partitioning-related optimizations, you should use `mapValues()` or `flatMapValues()` whenever you are not changing an element's key.

Custom Partitioners

While Spark's `HashPartitioner` and `RangePartitioner` are well suited to many use cases, Spark also allows you to tune how an RDD is partitioned by providing a custom `Partitioner` object. This can help you further reduce communication by taking advantage of domain-specific knowledge.

For example, suppose we wanted to run the PageRank algorithm in the previous section on a set of web pages. Here each page's ID (the key in our RDD) will be its URL. Using a simple hash function to do the partitioning, pages with similar URLs (e.g., *http://www.cnn.com/WORLD (http://www.cnn.com/WORLD)* and *http://www.cnn.com/US (http://www.cnn.com/US)*) might be hashed to completely different nodes. However, we know that web pages within the same domain tend to link to each other a lot. Because PageRank needs to send a message from each page to each of its neighbors on each iteration, it helps to group these pages into the same partition. We can do this with a custom `Partitioner` that looks at just the domain name instead of the whole URL.

To implement a custom partitioner, you need to subclass the `org.apache.spark.Partitioner` class and implement three methods:

- `numPartitions: Int`, which returns the number of partitions you will create.

- `getPartition(key: Any): Int`, which returns the partition ID (0 to `numPartitions-1`) for a given key.

- `equals()`, the standard Java equality method. This is important to implement because Spark will need to test your `Partitioner` object against other in-

```
      case dnp: DomainNamePartitioner =>
        dnp.numPartitions == numPartitions
      case _ =>
        false
    }
  }
```

Note that in the `equals()` method, we used Scala's pattern matching operator (`match`) to test whether `other` is a `DomainNamePartitioner`, and cast it if so; this is the same as using `instanceof()` in Java.

Using a custom `Partitioner` is easy: just pass it to the `partitionBy()` method. Many of the shuffle-based methods in Spark, such as `join()` and `groupByKey()`, can also take an optional `Partitioner` object to control the partitioning of the output.

Creating a custom `Partitioner` in Java is very similar to Scala: just extend the `spark.Partitioner` class and implement the required methods.

In Python, you do not extend a `Partitioner` class, but instead pass a hash function as an additional argument to `RDD.partitionBy()`. Example 4-27 demonstrates.

Example 4-27. Python custom partitioner

```python
import urlparse

def hash_domain(url):
  return hash(urlparse.urlparse(url).netloc)

rdd.partitionBy(20, hash_domain)   # Create 20 partitions
```

Note that the hash function you pass will be compared *by identity* to that of other RDDs. If you want to partition multiple RDDs with the same partitioner, pass the same function object (e.g., a global function) instead of creating a new `lambda` for each one!

Conclusion

In this chapter, we have seen how to work with key/value data using the specialized functions available in Spark. The techniques from Chapter 3 also still work on our pair RDDs. In the next chapter, we will look at how to load and save data.

3 "Join" (http://en.wikipedia.org/wiki/Join_(SQL)) is a database term for combining fields from two tables using common values.

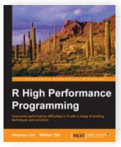
4 The Python API does not yet offer a way to query partitioners, though it still uses them internally.

PREV
3. Programming with RDDs

26 MINS

R

BOOK SECTION



Simple Tweaks to Use Less RAM

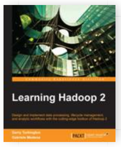
from: R High Performance Programming by Aloysius Lim...

Released: January 2015

7 MINS

R

BOOK SECTION



Kite Data

from: Learning Hadoop 2 by Gabriele Modena...

Released: February 2015

5 MINS

Hadoop

BOOK SECTION

Using HBase Tables for Single Entities

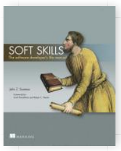
from: HBase Design Patterns by Mark Kerzner...

Released: December 2014

7 MINS

HBase

BOOK SECTION



Chapter 53. Do you really understand your retirement plan?

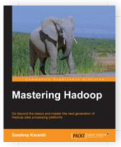
from: Soft Skills: The software developer's life manual by John Z. Sonmez

Released: December 2014

15 MINS

Personal & Professional Development

BOOK SECTION



YARN – Bringing Other Paradigms to Hadoop

from: Mastering Hadoop by Sandeep Karanth

Released: December 2014

8 MINS

Hadoop

62 MINS

Software Development