Table 3-2. Basic RDD transformations on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`map()`	Apply a function to each element in the RDD and return an RDD of the result.	`rdd.map(x => x + 1)`	`{2, 3, 4, 4}`
`flatMap()`	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	`rdd.flatMap(x => x.to(3))`	`{1, 2, 3, 2, 3, 3, 3}`
`filter()`	Return an RDD consisting of only elements that pass the condition passed to `filter()`.	`rdd.filter(x => x != 1)`	`{2, 3, 3}`
`distinct()`	Remove duplicates.	`rdd.distinct()`	`{1, 2, 3}`
`sample(withReplacement, fraction, [seed])`	Sample an RDD, with or without replacement.	`rdd.sample(false, 0.5)`	Nondeterministic

Table 3-3. Two-RDD transformations on RDDs containing {1, 2, 3} and {3, 4, 5}

Function name	Purpose	Example	Result
`union()`	Produce an RDD containing elements from both RDDs.	`rdd.union(other)`	`{1, 2, 3, 3, 4, 5}`
`intersection()`	RDD containing only elements found in both RDDs.	`rdd.intersection(other)`	`{3}`
`subtract()`	Remove the contents of one RDD (e.g., remove training data).	`rdd.subtract(other)`	`{1, 2}`
`cartesian()`	Cartesian product with the other RDD.	`rdd.cartesian(other)`	`{(1, 3), (1, 4), … (3,5)}`

Actions

The most common action on basic RDDs you will likely use is `reduce()`, which takes a function that operates on two elements of the type in your RDD and returns a new element of the same type. A simple example of such a function is +, which we can use to sum our RDD. With `reduce()`, we can easily sum the elements of our RDD, count the number of elements, and perform other types of aggregations (see Examples 3-32 through 3-34).

The simplest and most common operation that returns data to our driver program is `collect()`, which returns the entire RDD's contents. `collect()` is commonly used in unit tests where the entire contents of the RDD are expected to fit in memory, as that makes it easy to compare the value of our RDD with our expected result. `collect()` suffers from the restriction that all of your data must fit on a single machine, as it all needs to be copied to the driver.

`take(n)` returns *n* elements from the RDD and attempts to minimize the number of partitions it accesses, so it may represent a biased collection. It's important to note that these operations do not return the elements in the order you might expect.

These operations are useful for unit tests and quick debugging, but may introduce bottlenecks when you're dealing with large amounts of data.

If there is an ordering defined on our data, we can also extract the top elements from an RDD using `top()`. `top()` will use the default ordering on the data, but we can supply our own comparison function to extract the top elements.

Sometimes we need a sample of our data in our driver program. The `takeSample(withReplacement, num, seed)` function allows us to take a sample of our data either with or without replacement.

Sometimes it is useful to perform an action on all of the elements in the RDD, but without returning any result to the driver program. A good example of this would be posting JSON to a webserver or inserting records into a database. In either case, the `foreach()` action lets us perform computations on each element in the RDD without bringing it back locally.

The further standard operations on a basic RDD all behave pretty much exactly as you would imagine from their name. `count()` returns a count of the elements, and `countByValue()` returns a map of each unique value to its count. Table 3-4 summarizes these and other actions.

Table 3-4. Basic actions on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`collect()`	Return all elements from the RDD.	`rdd.collect()`	`{1, 2, 3, 3}`
`count()`	Number of elements in the RDD.	`rdd.count()`	`4`
`countByValue()`	Number of times each element occurs in the RDD.	`rdd.countByValue()`	`{(1, 1), (2, 1), (3, 2)}`

Function name	Purpose	Example	Result
`take(num)`	Return `num` elements from the RDD.	`rdd.take(2)`	`{1, 2}`
`top(num)`	Return the top `num` elements the RDD.	`rdd.top(2)`	`{3, 3}`
`takeOrdered(num)(ordering)`	Return `num` elements based on provided ordering.	`rdd.takeOrdered(2)(myOrdering)`	`{3, 3}`
`takeSample(withReplacement, num, [seed])`	Return `num` elements at random.	`rdd.takeSample(false, 1)`	Nondeterministic
`reduce(func)`	Combine the elements of the RDD together in parallel (e.g., `sum`).	`rdd.reduce((x, y) => x + y)`	9
`fold(zero)(func)`	Same as `reduce()` but with the provided zero value.	`rdd.fold(0)((x, y) => x + y)`	9
`aggregate(zeroValue)(seqOp, combOp)`	Similar to `reduce()` but used to return a different type.	`rdd.aggregate((0, 0))((x, y) => (x._1 + y, x._2 + 1), (x, y) => (x._1 + y._1, x._2 + y._2))`	`(9, 4)`
`foreach(func)`	Apply the provided function to each element of the RDD.	`rdd.foreach(func)`	Nothing

Converting Between RDD Types

Some functions are available only on certain types of RDDs, such as `mean()` and `variance()` on numeric RDDs or `join()` on key/value pair RDDs. We will cover these special functions for numeric data in Chapter 6 and pair RDDs in Chapter 4. In Scala and Java, these methods aren't defined on the standard RDD class, so to access this additional functionality we have to make sure we get the correct specialized class.

partitions. If a node that has data persisted on it fails, Spark will recompute the lost partitions of the data when needed. We can also replicate our data on multiple nodes if we want to be able to handle node failure without slowdown.

Spark has many levels of persistence to choose from based on what our goals are, as you can see in Table 3-6. In Scala (Example 3-40) and Java, the default `persist()` will store the data in the JVM heap as unserialized objects. In Python, we always serialize the data that persist stores, so the default is instead stored in the JVM heap as pickled objects. When we write data out to disk or off-heap storage, that data is also always serialized.

Table 3-6. Persistence levels from org.apache.spark.storage.StorageLevel and pyspark.StorageLevel; if desired we can replicate the data on two machines by adding _2 to the end of the storage level

Level	Space used	CPU time	In memory	On disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

Off-heap caching is experimental and uses Tachyon. If you are interested in off-heap caching with Spark, take a look at the Running Spark on Tachyon guide.

Example 3-40. persist() in Scala

```
val result = input.map(x => x * x)
result.persist(StorageLevel.DISK_ONLY)
println(result.count())
println(result.collect().mkString(","))
```

Notice that we called `persist()` on the RDD before the first action. The `persist()` call on its own doesn't force evaluation.