CHAPTER 6

Advanced Spark Programming

Introduction

This chapter introduces a variety of advanced Spark programming features that we didn't get to cover in the previous chapters. We introduce two types of shared variables: *accumulators* to aggregate information and *broadcast variables* to efficiently distribute large values. Building on our existing transformations on RDDs, we introduce batch operations for tasks with high setup costs, like querying a database. To expand the range of tools accessible to us, we cover Spark's methods for interacting with external programs, such as scripts written in R.

Throughout this chapter we build an example using ham radio operators' call logs as the input. These logs, at the minimum, include the call signs of the stations contacted. Call signs are assigned by country, and each country has its own range of call signs so we can look up the countries involved. Some call logs also include the physical location of the operators, which we can use to determine the distance involved. We include a sample log entry in Example 6-1. The book's sample repo includes a list of call signs to look up the call logs for and process the results.

Example 6-1. Sample call log entry in JSON, with some fields removed

```
{"address":"address here", "band":"40m","callsign":"KK6JLK","city":"SUNNYVALE",
"contactlat":"37.384733","contactlong":"-122.032164",
"county":"Santa Clara","dxcc":"291","fullname":"MATTHEW McPherrin",
"id":57779,"mode":"FM","mylat":"37.751952821","mylong":"-122.4208688735",...}
```

The first set of Spark features we'll look at are shared variables, which are a special type of variable you can use in Spark tasks. In our example we use Spark's shared variables to count nonfatal error conditions and distribute a large lookup table.

When our task involves a large setup time, such as creating a database connection or random-number generator, it is useful to share this setup work across multiple data items. Using a remote call sign lookup database, we examine how to reuse setup work by operating on a per-partition basis.

In addition to the languages directly supported by Spark, the system can call into programs written in other languages. This chapter introduces how to use Spark's language-agnostic `pipe()` method to interact with other programs through standard input and output. We will use the `pipe()` method to access an R library for computing the distance of a ham radio operator's contacts.

Finally, similar to its tools for working with key/value pairs, Spark has methods for working with numeric data. We demonstrate these methods by removing outliers from the distances computed with our ham radio call logs.

Accumulators

When we normally pass functions to Spark, such as a `map()` function or a condition for `filter()`, they can use variables defined outside them in the driver program, but each task running on the cluster gets a new copy of each variable, and updates from these copies are not propagated back to the driver. Spark's shared variables, *accumulators* and *broadcast variables*, relax this restriction for two common types of communication patterns: aggregation of results and broadcasts.

Our first type of shared variable, accumulators, provides a simple syntax for aggregating values from worker nodes back to the driver program. One of the most common uses of accumulators is to count events that occur during job execution for debugging purposes. For example, say that we are loading a list of all of the call signs for which we want to retrieve logs from a file, but we are also interested in how many lines of the input file were blank (perhaps we do not expect to see many such lines in valid input). Examples 6-2 through 6-4 demonstrate this scenario.

Example 6-2. Accumulator empty line count in Python

```python
file = sc.textFile(inputFile)
# Create Accumulator[Int] initialized to 0
blankLines = sc.accumulator(0)

def extractCallSigns(line):
    global blankLines   # Make the global variable accessible
    if (line == ""):
        blankLines += 1
    return line.split(" ")

callSigns = file.flatMap(extractCallSigns)
```

```
callSigns.saveAsTextFile(outputDir + "/callsigns")
print "Blank lines: %d" % blankLines.value
```

Example 6-3. Accumulator empty line count in Scala

```
val sc = new SparkContext(...)
val file = sc.textFile("file.txt")

val blankLines = sc.accumulator(0)  // Create an Accumulator[Int] initialized to 0

val callSigns = file.flatMap(line => {
  if (line == "") {
    blankLines += 1 // Add to the accumulator
  }
  line.split(" ")
})

callSigns.saveAsTextFile("output.txt")
println("Blank lines: " + blankLines.value)
```

Example 6-4. Accumulator empty line count in Java

```
JavaRDD<String> rdd = sc.textFile(args[1]);

final Accumulator<Integer> blankLines = sc.accumulator(0);
JavaRDD<String> callSigns = rdd.flatMap(
  new FlatMapFunction<String, String>() { public Iterable<String> call(String line) {
      if (line.equals("")) {
        blankLines.add(1);
      }
      return Arrays.asList(line.split(" "));
  }});

callSigns.saveAsTextFile("output.txt")
System.out.println("Blank lines: "+ blankLines.value());
```

In these examples, we create an `Accumulator[Int]` called `blankLines`, and then add 1 to it whenever we see a blank line in the input. After evaluating the transformation, we print the value of the counter. Note that we will see the right count only *after* we run the `saveAsTextFile()` action, because the transformation above it, `map()`, is lazy, so the side-effect incrementing of the accumulator will happen only when the lazy `map()` transformation is forced to occur by the `saveAsTextFile()` action.

Of course, it is possible to aggregate values from an entire RDD back to the driver program using actions like `reduce()`, but sometimes we need a simple way to aggregate values that, in the process of transforming an RDD, are generated at different scale or granularity than that of the RDD itself. In the previous example, accumulators let us count errors as we load the data, without doing a separate `filter()` or `reduce()`.

To summarize, accumulators work as follows:

- We create them in the driver by calling the `SparkContext.accumulator(initialValue)` method, which produces an accumulator holding an initial value. The return type is an `org.apache.spark.Accumulator[T]` object, where `T` is the type of `initialValue`.
- Worker code in Spark closures can add to the accumulator with its `+=` method (or `add` in Java).
- The driver program can call the `value` property on the accumulator to access its value (or call `value()` and `setValue()` in Java).

Note that tasks on worker nodes cannot access the accumulator's `value()`—from the point of view of these tasks, accumulators are *write-only* variables. This allows accumulators to be implemented efficiently, without having to communicate every update.

The type of counting shown here becomes especially handy when there are multiple values to keep track of, or when the same value needs to increase at multiple places in the parallel program (for example, you might be counting calls to a JSON parsing library throughout your program). For instance, often we expect some percentage of our data to be corrupted, or allow for the backend to fail some number of times. To prevent producing garbage output when there are too many errors, we can use a counter for valid records and a counter for invalid records. The value of our accumulators is available only in the driver program, so that is where we place our checks.

Continuing from our last example, we can now validate the call signs and write the output only if most of the input is valid. The ham radio call sign format is specified in Article 19 by the International Telecommunication Union, from which we construct a regular expression to verify conformance, shown in Example 6-5.

Example 6-5. Accumulator error count in Python

```
# Create Accumulators for validating call signs
validSignCount = sc.accumulator(0)
invalidSignCount = sc.accumulator(0)

def validateSign(sign):
    global validSignCount, invalidSignCount
    if re.match(r"\A\d?[a-zA-Z]{1,2}\d{1,4}[a-zA-Z]{1,3}\Z", sign):
        validSignCount += 1
        return True
    else:
        invalidSignCount += 1
        return False
```

```
# Count the number of times we contacted each call sign
validSigns = callSigns.filter(validateSign)
contactCount = validSigns.map(lambda sign: (sign, 1)).reduceByKey(lambda (x, y): x + y)

# Force evaluation so the counters are populated
contactCount.count()
if invalidSignCount.value < 0.1 * validSignCount.value:
    contactCount.saveAsTextFile(outputDir + "/contactCount")
else:
    print "Too many errors: %d in %d" % (invalidSignCount.value, validSignCount.value)
```

Accumulators and Fault Tolerance

Spark automatically deals with failed or slow machines by re-executing failed or slow tasks. For example, if the node running a partition of a `map()` operation crashes, Spark will rerun it on another node; and even if the node does not crash but is simply much slower than other nodes, Spark can preemptively launch a "speculative" copy of the task on another node, and take its result if that finishes. Even if no nodes fail, Spark may have to rerun a task to rebuild a cached value that falls out of memory. The net result is therefore that the same function may run multiple times on the same data depending on what happens on the cluster.

How does this interact with accumulators? The end result is that *for accumulators used in actions, Spark applies each task's update to each accumulator only once*. Thus, if we want a reliable absolute value counter, regardless of failures or multiple evaluations, we must put it inside an action like `foreach()`.

For accumulators used in RDD transformations instead of actions, this guarantee does not exist. An accumulator update within a transformation can occur more than once. One such case of a probably unintended multiple update occurs when a cached but infrequently used RDD is first evicted from the LRU cache and is then subsequently needed. This forces the RDD to be recalculated from its lineage, with the unintended side effect that calls to update an accumulator within the transformations in that lineage are sent again to the driver. Within transformations, accumulators should, consequently, be used only for debugging purposes.

While future versions of Spark may change this behavior to count the update only once, the current version (1.2.0) does have the multiple update behavior, so accumulators in transformations are recommended only for debugging purposes.

Custom Accumulators

So far we've seen how to use one of Spark's built-in accumulator types: integers (`Accumulator[Int]`) with addition. Out of the box, Spark supports accumulators of type `Double`, `Long`, and `Float`. In addition to these, Spark also includes an API to define custom accumulator types and custom aggregation operations (e.g., finding the maximum of the accumulated values instead of adding them). Custom accumulators need to extend `AccumulatorParam`, which is covered in the Spark API documentation. Beyond adding to a numeric value, we can use any operation for add, provided that operation is commutative and associative. For example, instead of adding to track the total we could keep track of the maximum value seen so far.

An operation *op* is commutative if *a op b = b op a* for all values *a*, *b*.

An operation *op* is associative if *(a op b) op c = a op (b op c)* for all values *a*, *b*, and *c*.

For example, `sum` and `max` are commutative and associative operations that are commonly used in Spark accumulators.

Broadcast Variables

Spark's second type of shared variable, *broadcast variables*, allows the program to efficiently send a large, read-only value to all the worker nodes for use in one or more Spark operations. They come in handy, for example, if your application needs to send a large, read-only lookup table to all the nodes, or even a large feature vector in a machine learning algorithm.

Recall that Spark automatically sends all variables referenced in your closures to the worker nodes. While this is convenient, it can also be inefficient because (1) the default task launching mechanism is optimized for small task sizes, and (2) you might, in fact, use the same variable in *multiple* parallel operations, but Spark will send it separately for each operation. As an example, say that we wanted to write a Spark program that looks up countries by their call signs by prefix matching in an array. This is useful for ham radio call signs since each country gets its own prefix, although the prefixes are not uniform in length. If we wrote this naively in Spark, the code might look like Example 6-6.

Example 6-6. Country lookup in Python

```python
# Look up the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country code to support this lookup.
signPrefixes = loadCallSignTable()

def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes)
    count = sign_count[1]
    return (country, count)

countryContactCounts = (contactCounts
                        .map(processSignCount)
                        .reduceByKey((lambda x, y: x+ y)))
```

This program would run, but if we had a larger table (say, with IP addresses instead of call signs), the `signPrefixes` could easily be several megabytes in size, making it expensive to send that `Array` from the master alongside each task. In addition, if we used the same `signPrefixes` object later (maybe we next ran the same code on *file2.txt*), it would be sent *again* to each node.

We can fix this by making `signPrefixes` a broadcast variable. A broadcast variable is simply an object of type `spark.broadcast.Broadcast[T]`, which wraps a value of type `T`. We can access this value by calling `value` on the `Broadcast` object in our tasks. The value is sent to each node only once, using an efficient, BitTorrent-like communication mechanism.

Using broadcast variables, our previous example looks like Examples 6-7 through 6-9.

Example 6-7. Country lookup with Broadcast values in Python

```
# Look up the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country code to support this lookup.
signPrefixes = sc.broadcast(loadCallSignTable())

def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes.value)
    count = sign_count[1]
    return (country, count)

countryContactCounts = (contactCounts
                        .map(processSignCount)
                        .reduceByKey((lambda x, y: x+ y)))

countryContactCounts.saveAsTextFile(outputDir + "/countries.txt")
```

Example 6-8. Country lookup with Broadcast values in Scala

```
// Look up the countries for each call sign for the
// contactCounts RDD.  We load an array of call sign
// prefixes to country code to support this lookup.
val signPrefixes = sc.broadcast(loadCallSignTable())
val countryContactCounts = contactCounts.map{case (sign, count) =>
  val country = lookupInArray(sign, signPrefixes.value)
  (country, count)
}.reduceByKey((x, y) => x + y)
countryContactCounts.saveAsTextFile(outputDir + "/countries.txt")
```

Example 6-9. Country lookup with Broadcast values in Java

```java
// Read in the call sign table
// Look up the countries for each call sign in the
// contactCounts RDD
final Broadcast<String[]> signPrefixes = sc.broadcast(loadCallSignTable());
JavaPairRDD<String, Integer> countryContactCounts = contactCounts.mapToPair(
  new PairFunction<Tuple2<String, Integer>, String, Integer> (){
    public Tuple2<String, Integer> call(Tuple2<String, Integer> callSignCount) {
      String sign = callSignCount._1();
      String country = lookupCountry(sign, callSignInfo.value());
      return new Tuple2(country, callSignCount._2());
    }}).reduceByKey(new SumInts());
countryContactCounts.saveAsTextFile(outputDir + "/countries.txt");
```

As shown in these examples, the process of using broadcast variables is simple:

1. Create a `Broadcast[T]` by calling `SparkContext.broadcast` on an object of type `T`. Any type works as long as it is also `Serializable`.
2. Access its value with the `value` property (or `value()` method in Java).
3. The variable will be sent to each node only once, and should be treated as read-only (updates will *not* be propagated to other nodes).

The easiest way to satisfy the *read-only* requirement is to broadcast a primitive value or a reference to an immutable object. In such cases, you won't be able to change the value of the broadcast variable except within the driver code. However, sometimes it can be more convenient or more efficient to broadcast a mutable object. If you do that, it is up to you to maintain the read-only condition. As we did with our call sign prefix table of `Array[String]`, we must make sure that the code we run on our worker nodes does not try to do something like `val theArray = broadcastArray.value; theArray(0) = newValue`. When run in a worker node, that line will assign `newValue` to the first array element only in the copy of the array local to the worker node running the code; it will not change the contents of `broadcastArray.value` on any of the other worker nodes.

Optimizing Broadcasts

When we are broadcasting large values, it is important to choose a data serialization format that is both fast and compact, because the time to send the value over the network can quickly become a bottleneck if it takes a long time to either serialize a value or to send the serialized value over the network. In particular, Java Serialization, the default serialization library used in Spark's Scala and Java APIs, can be very inefficient out of the box for anything except arrays of primitive types. You can optimize serialization by selecting a different serialization library using the `spark.serializer` property (Chapter 8 will describe how to use *Kryo*, a faster serialization library), or by implementing your own serialization routines for your data types (e.g., using the `java.io.Externalizable` interface for Java Serialization, or using the `reduce()` method to define custom serialization for Python's pickle library).

Working on a Per-Partition Basis

Working with data on a per-partition basis allows us to avoid redoing setup work for each data item. Operations like opening a database connection or creating a random-number generator are examples of setup steps that we wish to avoid doing for each element. Spark has *per-partition* versions of `map` and `foreach` to help reduce the cost of these operations by letting you run code only once for each partition of an RDD.

Going back to our example with call signs, there is an online database of ham radio call signs we can query for a public list of their logged contacts. By using partition-based operations, we can share a connection pool to this database to avoid setting up many connections, and reuse our JSON parser. As Examples 6-10 through 6-12 show, we use the `mapPartitions()` function, which gives us an iterator of the elements in each partition of the input RDD and expects us to return an iterator of our results.

Example 6-10. Shared connection pool in Python

```
def processCallSigns(signs):
    """Lookup call signs using a connection pool"""
    # Create a connection pool
    http = urllib3.PoolManager()
    # the URL associated with each call sign record
    urls = map(lambda x: "http://73s.com/qsos/%s.json" % x, signs)
    # create the requests (non-blocking)
    requests = map(lambda x: (x, http.request('GET', x)), urls)
    # fetch the results
    result = map(lambda x: (x[0], json.loads(x[1].data)), requests)
    # remove any empty results and return
    return filter(lambda x: x[1] is not None, result)

def fetchCallSigns(input):
    """Fetch call signs"""
    return input.mapPartitions(lambda callSigns : processCallSigns(callSigns))

contactsContactList = fetchCallSigns(validSigns)
```

Example 6-11. Shared connection pool and JSON parser in Scala

```
val contactsContactLists = validSigns.distinct().mapPartitions{
  signs =>
  val mapper = createMapper()
  val client = new HttpClient()
  client.start()
  // create http request
  signs.map {sign =>
    createExchangeForSign(sign)
 // fetch responses
  }.map{ case (sign, exchange) =>
      (sign, readExchangeCallLog(mapper, exchange))
  }.filter(x => x._2 != null) // Remove empty CallLogs
}
```

Example 6-12. Shared connection pool and JSON parser in Java

```
// Use mapPartitions to reuse setup work.
JavaPairRDD<String, CallLog[]> contactsContactLists =
  validCallSigns.mapPartitionsToPair(
  new PairFlatMapFunction<Iterator<String>, String, CallLog[]>() {
    public Iterable<Tuple2<String, CallLog[]>> call(Iterator<String> input) {
      // List for our results.
      ArrayList<Tuple2<String, CallLog[]>> callsignLogs = new ArrayList<>();
      ArrayList<Tuple2<String, ContentExchange>> requests = new ArrayList<>();
      ObjectMapper mapper = createMapper();
      HttpClient client = new HttpClient();
      try {
        client.start();
        while (input.hasNext()) {
          requests.add(createRequestForSign(input.next(), client));
        }
        for (Tuple2<String, ContentExchange> signExchange : requests) {
          callsignLogs.add(fetchResultFromRequest(mapper, signExchange));
        }
      } catch (Exception e) {
      }
      return callsignLogs;
    }});
System.out.println(StringUtils.join(contactsContactLists.collect(), ","));
```

When operating on a per-partition basis, Spark gives our function an `Iterator` of the elements in that partition. To return values, we return an `Iterable`. In addition to `mapPartitions()`, Spark has a number of other per-partition operators, listed in Table 6-1.

Table 6-1. Per-partition operators

Function name	We are called with	We return	Function signature on RDD[T]
`mapPartitions()`	Iterator of the elements in that partition	Iterator of our return elements	`f: (Iterator[T]) → Iterator[U]`
`mapPartitionsWithIndex()`	Integer of partition number, and Iterator of the elements in that partition	Iterator of our return elements	`f: (Int, Iterator[T]) → Iterator[U]`

Function name	We are called with	We return	Function signature on RDD[T]
`foreachPartition()`	Iterator of the elements	Nothing	`f: (Iterator[T]) → Unit`

In addition to avoiding setup work, we can sometimes use `mapPartitions()` to avoid object creation overhead. Sometimes we need to make an object for aggregating the result that is of a different type. Thinking back to Chapter 3, where we computed the average, one of the ways we did this was by converting our RDD of numbers to an RDD of tuples so we could track the number of elements processed in our reduce step. Instead of doing this for each element, we can instead create the tuple once per partition, as shown in Examples 6-13 and 6-14.

Example 6-13. Average without mapPartitions() in Python

```python
def combineCtrs(c1, c2):
    return (c1[0] + c2[0], c1[1] + c2[1])

def basicAvg(nums):
    """Compute the average"""
    nums.map(lambda num: (num, 1)).reduce(combineCtrs)
```

Example 6-14. Average with mapPartitions() in Python

```python
def partitionCtr(nums):
    """Compute sumCounter for partition"""
    sumCount = [0, 0]
    for num in nums:
        sumCount[0] += num
        sumCount[1] += 1
    return [sumCount]

def fastAvg(nums):
    """Compute the avg"""
    sumCount = nums.mapPartitions(partitionCtr).reduce(combineCtrs)
    return sumCount[0] / float(sumCount[1])
```

Piping to External Programs

With three language bindings to choose from out of the box, you may have all the options you need for writing Spark applications. However, if none of Scala, Java, or Python does what you need, then Spark provides a general mechanism to pipe data to programs in other languages, like R scripts.

Spark provides a `pipe()` method on RDDs. Spark's `pipe()` lets us write parts of jobs using any language we want as long as it can read and write to Unix standard streams. With `pipe()`, you can write a transformation of an RDD that reads each

RDD element from standard input as a `String`, manipulates that `String` however you like, and then writes the result(s) as `Strings` to standard output. The interface and programming model is restrictive and limited, but sometimes it's just what you need to do something like make use of a native code function within a map or filter operation.

Most likely, you'd want to pipe an RDD's content through some external program or script because you've already got complicated software built and tested that you'd like to reuse with Spark. A lot of data scientists have code in R,[1] and we can interact with R programs using `pipe()`.

In Example 6-15 we use an R library to compute the distance for all of the contacts. Each element in our RDD is written out by our program with newlines as separators, and every line that the program outputs is a string element in the resulting RDD. To make it easy for our R program to parse the input we will reformat our data to be `mylat, mylon, theirlat, theirlon`. Here we have a comma as the separator.

Example 6-15. R distance program

```
#!/usr/bin/env Rscript
library("Imap")
f <- file("stdin")
open(f)
while(length(line <- readLines(f,n=1)) > 0) {
  # process line
  contents <- Map(as.numeric, strsplit(line, ","))
  mydist <- gdist(contents[[1]][1], contents[[1]][2],
                  contents[[1]][3], contents[[1]][4],
                  units="m", a=6378137.0, b=6356752.3142, verbose = FALSE)
  write(mydist, stdout())
}
```

If that is written to an executable file named *./src/R/finddistance.R*, then it looks like this in use:

```
$ ./src/R/finddistance.R
37.75889318222431,-122.42683635321838,37.7614213,-122.4240097
349.2602
coffee
NA
ctrl-d
```

So far, so good—we've now got a way to transform every line from `stdin` into output on `stdout`. Now we need to make `finddistance.R` available to each of our worker

1 The SparkR project also provides a lightweight frontend to use Spark from within R.

nodes and to actually transform our RDD with our shell script. Both tasks are easy to accomplish in Spark, as you can see in Examples 6-16 through 6-18.

Example 6-16. Driver program using pipe() to call finddistance.R in Python

```
# Compute the distance of each call using an external R program
distScript = "./src/R/finddistance.R"
distScriptName = "finddistance.R"
sc.addFile(distScript)
def hasDistInfo(call):
    """Verify that a call has the fields required to compute the distance"""
    requiredFields = ["mylat", "mylong", "contactlat", "contactlong"]
    return all(map(lambda f: call[f], requiredFields))
def formatCall(call):
    """Format a call so that it can be parsed by our R program"""
    return "{0},{1},{2},{3}".format(
        call["mylat"], call["mylong"],
        call["contactlat"], call["contactlong"])

pipeInputs = contactsContactList.values().flatMap(
    lambda calls: map(formatCall, filter(hasDistInfo, calls)))
distances = pipeInputs.pipe(SparkFiles.get(distScriptName))
print distances.collect()
```

Example 6-17. Driver program using pipe() to call finddistance.R in Scala

```
// Compute the distance of each call using an external R program
// adds our script to a list of files for each node to download with this job
val distScript = "./src/R/finddistance.R"
val distScriptName = "finddistance.R"
sc.addFile(distScript)
val distances = contactsContactLists.values.flatMap(x => x.map(y =>
  s"$y.contactlay,$y.contactlong,$y.mylat,$y.mylong")).pipe(Seq(
    SparkFiles.get(distScriptName)))
println(distances.collect().toList)
```

Example 6-18. Driver program using pipe() to call finddistance.R in Java

```
// Compute the distance of each call using an external R program
// adds our script to a list of files for each node to download with this job
String distScript = "./src/R/finddistance.R";
String distScriptName = "finddistance.R";
sc.addFile(distScript);
JavaRDD<String> pipeInputs = contactsContactLists.values()
  .map(new VerifyCallLogs()).flatMap(
  new FlatMapFunction<CallLog[], String>() {
    public Iterable<String> call(CallLog[] calls) {
      ArrayList<String> latLons = new ArrayList<String>();
      for (CallLog call: calls) {
        latLons.add(call.mylat + "," + call.mylong +
                "," + call.contactlat + "," + call.contactlong);
      }
      return latLons;
    }
  });
JavaRDD<String> distances = pipeInputs.pipe(SparkFiles.get(distScriptName));
System.out.println(StringUtils.join(distances.collect(), ","));
```

With `SparkContext.addFile(path)`, we can build up a list of files for each of the worker nodes to download with a Spark job. These files can come from the driver's local filesystem (as we did in these examples), from HDFS or other Hadoop-supported filesystems, or from an HTTP, HTTPS, or FTP URI. When an action is run in the job, the files will be downloaded by each of the nodes. The files can then be found on the worker nodes in `SparkFiles.getRootDirectory`, or located with `SparkFiles.get(filename)`. Of course, this is only one way to make sure that `pipe()` can find a script on each worker node. You could use another remote copying tool to place the script file in a knowable location on each node.

All the files added with `SparkContext.addFile(path)` are stored in the same directory, so it's important to use unique names.

Once the script is available, the `pipe()` method on RDDs makes it easy to pipe the elements of an RDD through the script. Perhaps a smarter version of `findDistance` would accept `SEPARATOR` as a command-line argument. In that case, either of these would do the job, although the first is preferred:

- `rdd.pipe(Seq(SparkFiles.get("finddistance.R"), ","))`
- `rdd.pipe(SparkFiles.get("finddistance.R") + " ,")`

In the first option, we are passing the command invocation as a sequence of positional arguments (with the command itself at the zero-offset position); in the second, we're passing it as a single command string that Spark will then break down into positional arguments.

We can also specify shell environment variables with `pipe()` if we desire. Simply pass in a map of environment variables to values as the second parameter to `pipe()`, and Spark will set those values.

You should now at least have an understanding of how to use `pipe()` to process the elements of an RDD through an external command, and of how to distribute such command scripts to the cluster in a way that the worker nodes can find them.

Numeric RDD Operations

Spark provides several descriptive statistics operations on RDDs containing numeric data. These are in addition to the more complex statistical and machine learning methods we will describe later in Chapter 11.

Spark's numeric operations are implemented with a streaming algorithm that allows for building up our model one element at a time. The descriptive statistics are all computed in a single pass over the data and returned as a `StatsCounter` object by calling `stats()`. Table 6-2 lists the methods available on the `StatsCounter` object.

Table 6-2. Summary statistics available from StatsCounter

Method	Meaning
`count()`	Number of elements in the RDD
`mean()`	Average of the elements
`sum()`	Total
`max()`	Maximum value
`min()`	Minimum value
`variance()`	Variance of the elements
`sampleVariance()`	Variance of the elements, computed for a sample
`stdev()`	Standard deviation
`sampleStdev()`	Sample standard deviation

If you want to compute only one of these statistics, you can also call the corresponding method directly on an RDD—for example, `rdd.mean()` or `rdd.sum()`.

In Examples 6-19 through 6-21, we will use summary statistics to remove some outliers from our data. Since we will be going over the same RDD twice (once to compute the summary statistics and once to remove the outliers), we may wish to cache the RDD. Going back to our call log example, we can remove the contact points from our call log that are too far away.

Example 6-19. Removing outliers in Python

```
# Convert our RDD of strings to numeric data so we can compute stats and
# remove the outliers.
distanceNumerics = distances.map(lambda string: float(string))
stats = distanceNumerics.stats()
stddev = std.stdev()
mean = stats.mean()
reasonableDistances = distanceNumerics.filter(
  lambda x: math.fabs(x - mean) < 3 * stddev)
print reasonableDistances.collect()
```

Example 6-20. Removing outliers in Scala

```
// Now we can go ahead and remove outliers since those may have misreported locations
// first we need to take our RDD of strings and turn it into doubles.
val distanceDouble = distance.map(string => string.toDouble)
val stats = distanceDoubles.stats()
val stddev = stats.stdev
val mean = stats.mean
val reasonableDistances = distanceDoubles.filter(x => math.abs(x-mean) < 3 * stddev)
println(reasonableDistance.collect().toList)
```

Example 6-21. Removing outliers in Java

```
// First we need to convert our RDD of String to a DoubleRDD so we can
// access the stats function
JavaDoubleRDD distanceDoubles = distances.mapToDouble(new DoubleFunction<String>() {
    public double call(String value) {
      return Double.parseDouble(value);
    }});
final StatCounter stats = distanceDoubles.stats();
final Double stddev = stats.stdev();
final Double mean = stats.mean();
JavaDoubleRDD reasonableDistances =
  distanceDoubles.filter(new Function<Double, Boolean>() {
    public Boolean call(Double x) {
      return (Math.abs(x-mean) < 3 * stddev);}});
System.out.println(StringUtils.join(reasonableDistance.collect(), ","));
```

With that final piece we have completed our sample application, which uses accumulators and broadcast variables, per-partition processing, interfaces with external programs, and summary statistics. The entire source code is available in *src/python/ChapterSixExample.py*, *src/main/scala/com/oreilly/learningsparkexamples/scala/ChapterSixExample.scala*, and *src/main/java/com/oreilly/learningsparkexamples/java/ChapterSixExample.java*, respectively.

Conclusion

In this chapter, you have been introduced to some of the more advanced Spark programming features that you can use to make your programs more efficient or expressive. Subsequent chapters cover deploying and tuning Spark applications, as well as built-in libraries for SQL and streaming and machine learning. We'll also start seeing more complex and more complete sample applications that make use of much of the functionality described so far, and that should help guide and inspire your own usage of Spark.