

Apache Spa-R-k

Chapter 1, Simple Parallelism with R introduced the `segue` package as a simple means to directly utilize a Hadoop cluster, hosted within the cloud on Amazon Web Services, direct from your local laptop R session. However, during the writing of the book, it became clear that the new Big Data kid-on-the-block, **Apache Spark**, had not just arrived on the scene, but was potentially going to eclipse Hadoop and its MapReduce implementation as the platform of choice for running large scale data analysis on clusters of commodity hardware, not only reliably, but also with interactive levels of performance.

As the author, I decided that I wanted to include a description of this important and increasingly popular new parallel distributed big data processing technology in the book. As the writing of the book developed so also has Apache Spark and its integration with R. My original effort at documenting Spark was quickly superseded by its rapid technical development – the interface available from R has been through several revisions. The decision was therefore made to create a soft format bonus chapter describing the latest available version of Apache Spark and its accompanying **SparkR** package, enabling this aspect of the book to be as late breaking as possible. A new major release of Spark, version 2.0, which will improve its mixed workload ability for processing both real-time and bulk data, is also due for release later in the year. Therefore, Apache Spark is going to continue to maintain and grow in its importance as a high-performance processing framework for big data analytics.

At the time of writing this chapter, the latest available release of Spark is version 1.6.1, released in March 2016, and it is this version that we will focus on in this chapter.

So, with our most modern data science hats now adorned, let's find out what all the fuss has been about with Apache Spark, and discover how it can be utilized from R.

In this bonus chapter, we will cover the following topics:

- Installing and setting up a Spark compute cluster
- Using the SparkR package to interact with a Spark cluster from R
- Spark's core Resilient Distributed Datasets programming abstraction (RDD)
- Partitioning RDDs for parallelism
- Importing and Exporting data from Spark
- RDD transformations and actions

About Spark

The development of Spark started in 2009 at Berkeley University's AMPLab, where researchers sought to improve on MapReduce to enable iterative and real-time data processing at scale. Spark was open sourced in 2010, and migrated to becoming an Apache-shepherded project in 2013. By late 2014, Apache Spark had been fully adopted by the main big data platform distribution providers including Hortonworks and Cloudera. Today, it has the most active development community of all the big data technologies, and has demonstrated 100x speed-up, compared to Hadoop MapReduce, for various types of processing workload.

Apache Spark achieves its impressive performance through a combination of in-memory processing, controllable caching of reusable data, and maintaining a computational dependency tree that, through lazy evaluation, ensures that only the required data is accessed when required for a given complex processing task.

Apache Spark's popularity is also due to its support for a number of different processing models within a single platform architecture including: an optimized machine learning library (**MLlib**), a parallel SQL engine that supports both structured and semi-structured data, a graph processing engine (**GraphX**) for computing relationships between entities (for example, for analyzing social media networks), and a micro-batch mechanism (**Streaming**) for real-time data processing. Apache Spark also has some compatibility with Hadoop, with its own implementation of map/reduce operations, and is able to directly read and write data in many different formats including direct interoperation with the Hadoop filesystem (HDFS), thereby easing data application migration from Hadoop to Spark. All these capabilities make Apache Spark a highly attractive one-stop-shop for a wide range of both explorative analysis and production data processing pipelines.

At the core of Spark is its parallelized data abstraction, **Resilient Distributed Datasets (RDD)**. An RDD is a collection of data elements partitioned across the *nodes* in the Spark cluster, which can be operated on in parallel. Although RDDs are how the internals of Spark work with data, for R itself, a higher level interface abstraction is made available from the **SparkR** package, namely, the **DataFrame**. The Spark DataFrame enables implicit parallel manipulation of the potentially massive tables of data distributed across the Spark cluster. Unsurprisingly, Spark's DataFrame has much similarity with R's own native tabular data namesake, `data.frame`, and supports a similar set and style of operations that you may already be familiar with from the popular R package, **dplyr**. There is also interface support from R for utilizing aspects of Spark's MLLib.

The SparkR package is now made available as part of the standard Spark distribution, both binary and source.

The remainder of this chapter will describe how to install and set up Spark, and how you can take advantage of its large-scale, reliable, and high-performance in-memory processing directly from R by using the SparkR API.

Installing and setting up Spark and SparkR

The first component to download and install is Spark itself from the main Apache site at <http://spark.apache.org/downloads.html>. The version of Spark I have used in this chapter is 1.2.1, compiled with the default option for Hadoop 1.X. You will need to uncompress and unarchive the pre-built binaries and download them into a suitable directory on your computer.

Although Spark is written in a language called Scala, Scala itself compiles to Java bytecode. Since we are using the pre-built binaries, you only need to have Java (version 6 or later) installed on your system in order to run Spark.

The second component to download and install is the SparkR package. At the time of writing, SparkR is still in its pre-production phase and is made available in source form from GitHub. By using another R package, `devtools` (itself available directly from CRAN), you can use the following command directly from within your R session to both install and compile SparkR:

```
> library(devtools)
> install_github("amplab-extras/SparkR-pkg", subdir="pkg")
```

This will take several minutes to download the package and auto-run the SparkR build script, and will generate substantial logging output to the console as it proceeds.



Note that SparkR has a dependency on the rJava and testthat packages. You should review the current details on the prerequisites and installation instructions for SparkR, as described on the project's GitHub page at <http://amplab-extras.github.io/SparkR-pkg/>.

The simplest SparkR program

Once the installation process completes, you can run the following sequence of R commands to test all is well:

```
> sc = sparkR.init(master="local")
Launching java with command java ...
01/03/12 19:45:12 INFO Slf4jLogger: Slf4jLogger started
> rdd <- parallelize(sc, 1:100)
> rdd2 <- lapply(rdd, function(x) { x + 100 })
> cache(rdd2)
> take(rdd2,1)
[[1]]
[1] 101
> sparkR.stop()
Stopping SparkR
01/03/12 19:47:31 INFO RemoteActorRefProvider$RemotingTerminator:
  Remote daemon shut down; proceeding with flushing remote
  transports.
```

Some of the output in the preceding code has been removed for brevity. This simple sequence of half a dozen statements touches on the main features of a SparkR program including lazy evaluation of RDDs.

The call to `sparkR.init()` creates a special local standalone Spark quasi-cluster within the context of what is known as the **Driver Program**, and returns a handle to the specific **Spark Context** for our newly created R Spark application instance. The next statement, `parallelize()`, creates an RDD containing 100 elements with the values 1 to 100, and returns a handle to the RDD. The third statement is an RDD transformation, and uses SparkR's `lapply()` to add 100 to each element of `rdd`, creating a second separate RDD with the reference handle `rdd2`. Note that `rdd2` is not itself evaluated at this point.

The fourth statement effectively marks `rdd2` for caching according to the default persistency setting, which, in this case, is the default of in-memory only. Again, nothing is calculated or persisted at this point; this will only happen when the RDD is next evaluated. The very next statement, `take()` (an action), retrieves the first (optimal access) element value in `rdd2` from the cluster, causing the RDD to be evaluated, and materializes (in this case) the value `101` into the R session itself. Finally, the call to `sparkR.stop()` terminates our Spark application instance; in this case, because the cluster was explicitly created (`master="local"`) rather than pre-existing, the "cluster" is also itself completely shut down rather than just disconnected.

If you wish to quickly look ahead in this chapter, take a look at the bottom-half of *Figure 5*—you will see a depiction of the Spark system level components that are active under this example of a default local configuration of a Spark cluster within the driver program. We will see later how we can create a persistent running Spark cluster to which multiple running R sessions can potentially connect and execute parallel operations.



Testing SparkR programs:

For testing the correctness of SparkR programs on a small machine such as a laptop, using the `master="local"` setting can be very useful, though it is important to remember that you will be much more limited in performance, particularly with regard to holding RDDs in memory. So, it is best to use this approach with relatively small sample datasets.

The Spark application web console

But first, let's rerun the previous script one statement at a time, and take a look at the additional web-based system monitoring that comes built-in with Spark.

```
> sc = sparkR.init(master="local")
```

Once this command has been run, start up your favorite web browser, and type in the URL: `http://localhost:4040`. You should see a web page similar to the *Figure 1* given next. This is the application monitoring console showing the currently active and past jobs executed by our SparkR application instance.

Note that the default name given to our application is displayed in the top-right corner on the menu bar of the page preceding **application UI**—in this case, **SparkR**. An option to `sparkR.init()` allows you to set the name of your application instance explicitly.

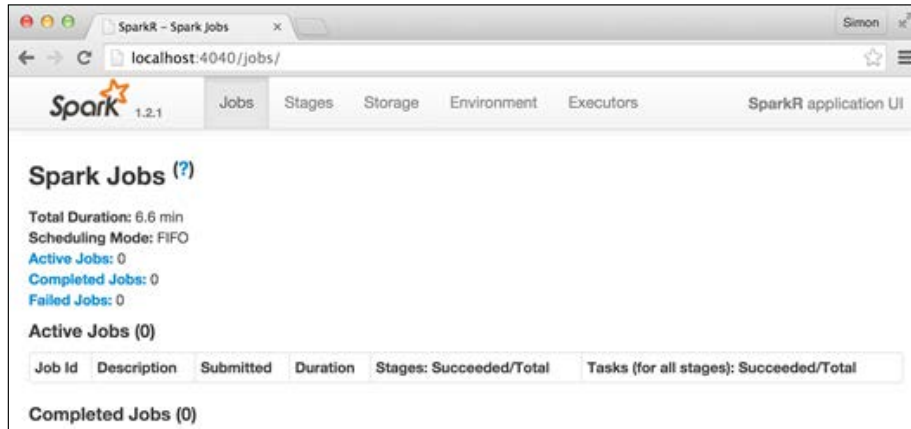


Figure 1: SparkR application monitoring web console showing initial jobs start-up state.

If you switch to the **Executors** tab, you will see a screen similar to *Figure 2*, where a Java Virtual Machine, designated "driver", has been started up. Since we are running a local cluster, this is the executor that will be used to perform any parallel data processing operations. The Spark framework is itself implemented in Scala/Java, but is capable of executing code written in other languages. SparkR ensures that the JVM will be able to run any R code necessary to carry out the computation by automatically transferring the required variables and function definitions across the wire to the executing JVM. There are some aspects of this we need to be more aware of when running our R session against a remote cluster, which we will cover later in the chapter.

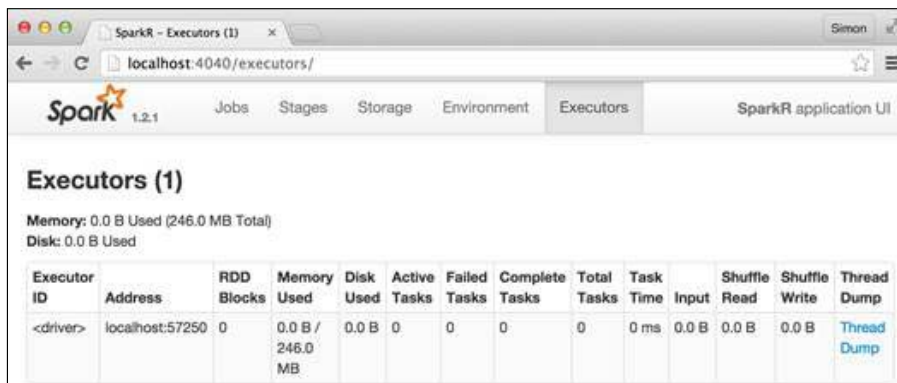


Figure 2: SparkR Application monitoring showing local "driver" JVM task executor.

Now run the following command:

```
> rdd <- parallelize(sc, 1:100)
```

If you look at the main **Jobs** tab in the web console, you will see three completed jobs associated with this one RDD creation statement. Spark maps the high-level RDD API calls into an efficient sequence of tasks that are run by the allocated cluster executors.

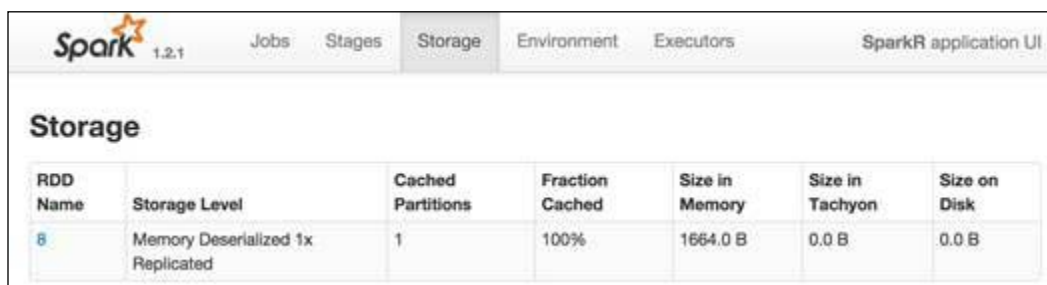
Run the next two statements:

```
> rdd2 <- lapply(rdd, function(x) { x + 100 })
> cache(rdd2)
```

And now look at the **Storage** tab in the web console—it will be empty. Until `rdd2` is evaluated, for example, by running an action, it will not be manifested in the cache. So let's do that by executing the `take()` statement:

```
> take(rdd2, 1)
```

And then look at the **Storage** tab—it should now have a single entry similar to the one shown in the following screenshot:



| RDD Name | Storage Level | Cached Partitions | Fraction Cached | Size in Memory | Size in Tachyon | Size on Disk |
|----------|-----------------------------------|-------------------|-----------------|----------------|-----------------|--------------|
| 8 | Memory Deserialized 1x Replicated | 1 | 100% | 1664.0 B | 0.0 B | 0.0 B |

Figure 3: SparkR web console showing an RDD has been manifested in the cache.

Partitioning RDDs for parallelism

RDDs are distributed in terms of partitions. As you can see in the **Cached Partitions** field in *Figure 3*, our simple example RDD has been cached with only one partition, which effectively means that parallelism will be limited for this dataset to a single processing task. A Spark task is a single unit of work, and even if we had more than one worker, a single task cannot be split between multiple workers.

Helpfully, you can directly repartition an existing RDD into more (or less) partitions with the following (note the use of the `L` qualifier to indicate to R that an integer, and not a numeric value, must be passed as the parameter type for the number of partitions requested):

```
> rdd4 <- repartition(rdd2,4L)
> numPartitions(rdd4)
[1] 4
> numPartitions(rdd2)
[1] 1
```

Notice how the number of partitions for `rdd2` has remained unchanged – we have created a new `rdd4` with the same set of elements, but now split into four separately taskable smaller parts. One thing to be aware of is that repartitioning large datasets can be a time consuming process requiring data to be shuffle-exchanged amongst the workers. Spark does this in order to spread data evenly, and ensure a balanced parallel workload. You can see just how much shuffling has been invoked with an RDD operation through reviewing the **Stages** tab in the web console. As we can see in *Figure 4*, the repartition (stage number 8) invoked a shuffle write of nearly the size of the whole dataset.

It is useful to observe this aspect of Spark execution with the web monitoring console, particularly if you find your program taking a long time to run, since this could be indicative of significant data shuffling – something that we want to reduce from happening in order to maintain highest performance.

A key aspect of partitioning an RDD is the relationship between partitions which result in an equivalent number of tasks, and the number of tasks to be computed by each worker in the cluster. The general rule of thumb – though "caveat emptor!" still applies – is to have a minimum of two to four tasks per partition per separate worker node. So, if we have a cluster of 10 workers, then splitting RDDs into at least 40 partitions will ensure efficient use of the cluster resources, reduce the impact of workload imbalance, and deliver best overall performance.

The screenshot shows the Spark Stages UI for a job. At the top, there are navigation tabs: Jobs, Stages (selected), Storage, Environment, and Executors. The Spark logo and version 1.2.1 are on the left, and 'SparkR application UI' is on the right.

Spark Stages (for all jobs)

Total Duration: 13 min
 Scheduling Mode: FIFO
 Active Stages: 0
 Completed Stages: 13
 Failed Stages: 0

Active Stages (0)

| Stage Id | Description | Submitted | Duration | Tasks: Succeeded/Total | Input | Output | Shuffle Read | Shuffle Write |
|------------------------------|--|------------------------------|----------|------------------------|----------|--------|--------------|---------------|
| Completed Stages (13) | | | | | | | | |
| Stage Id | Description | Submitted | Duration | Tasks: Succeeded/Total | Input | Output | Shuffle Read | Shuffle Write |
| 15 | collect at NativeMethodAccessorImpl.java:2 | +details 2015/03/13 14:02:19 | 1 s | 4/4 | | | | |
| 13 | collect at NativeMethodAccessorImpl.java:2 | +details 2015/03/13 14:02:18 | 1 s | 4/4 | | | | |
| 11 | collect at NativeMethodAccessorImpl.java:2 | +details 2015/03/13 14:02:17 | 1 s | 4/4 | | | | |
| 9 | collect at NativeMethodAccessorImpl.java:2 | +details 2015/03/13 14:02:15 | 1 s | 4/4 | | | | |
| 8 | RDD at RRDD.scala:18 | +details 2015/03/13 14:02:14 | 0.7 s | 1/1 | 1664.0 B | | | 1563.0 B |

Figure 4: Observation of the need for potentially expensive data shuffling (stage 8) when repartitioning RDDs.

Now all we have to do is increase the number of cores available to our cluster from the default of just one. To do that, we are going to shut down our current local cluster by calling `sparkR.stop()`, and fire up a separate standalone cluster with multiple workers to provide full parallel compute.

Firing up your own Spark cluster

The arrangement of the Spark cluster that we are going to create is depicted in *Figure 5*. We will launch a separate **Master** and two **Workers** using the appropriate Spark command line shell scripts. The diagram shows the key components highlighting the default web monitoring URLs for each component, and the basic steps involved in running up your own cluster, which we will now explore in detail.

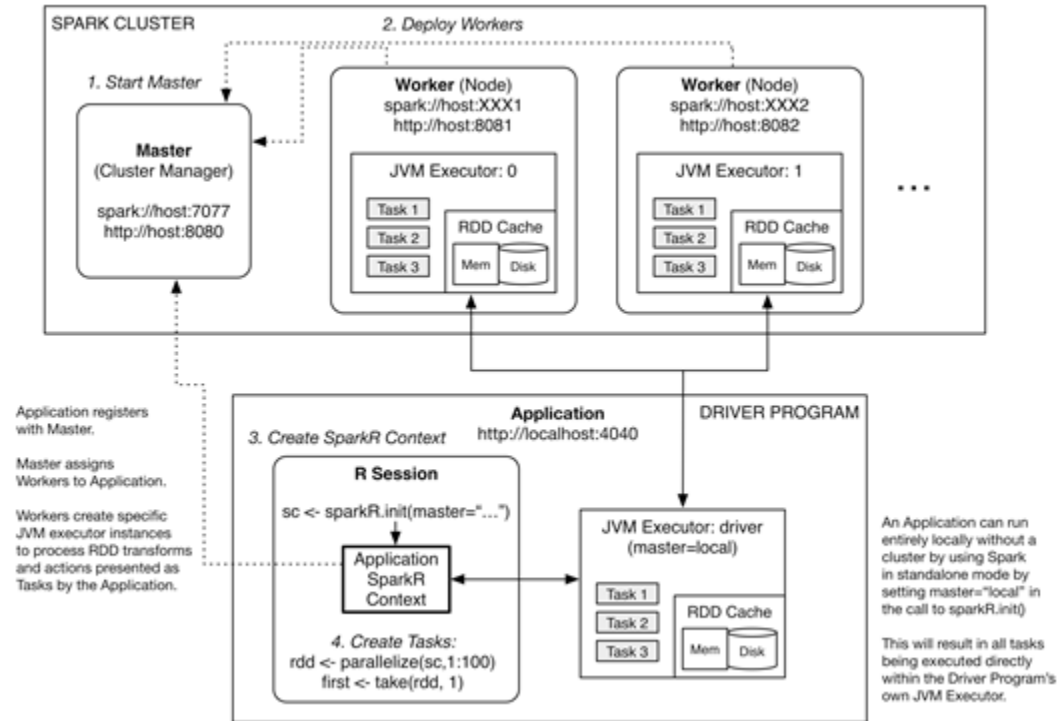


Figure 5: SparkR driver program, and Spark cluster Master/Worker configuration.

Starting the Master

First fire up a terminal window on your system, change directory to the location of your downloaded Spark installation, and then type the following at the command-line prompt to start the Master:

```
prompt$ ./sbin/start-master.sh
starting org.apache.spark.deploy.master.Master, logging to
/Users/simon/Downloads/spark-1.2.1/sbin/./logs/
spark-simon-org.apache.spark.deploy.master.Master-1-Simons-
MacBook-Pro.local.out
```

The Master has its own web monitoring console, and, by default, this will be accessible at `http://localhost:8080`. Typing this into your favorite web browser should yield a web page similar to *Figure 6*:

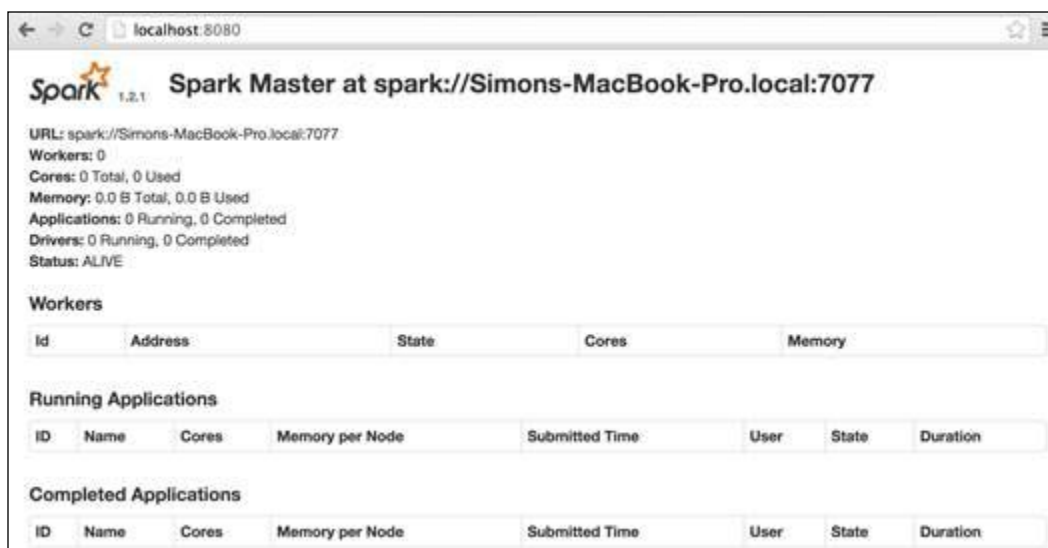


Figure 6: Master's web monitoring console at its default URL of localhost:8080 in its initial state with no Workers.

Note the Spark URL being advertised by the Master is `spark://Simons-MacBook-Pro.local:7077`; we use this when starting up a Worker so that it knows which Master to register itself with.

Starting the Workers

There are two key aspects to Workers that we want to control—how much memory and how many processor cores they use. The start script for a Worker allows us to control these two aspects directly with command-line parameters. Spark also operates with configuration files. TODO add reference link for how to manipulate configuration files.

On my MacBook Pro, there is a maximum of 16 GB memory space that is available along with four processing cores. Let's start the two Workers with 6 GB of memory (`-m` option) and two processing cores (`-c` option) each (note that Spark terminology uses `slave` and `worker` interchangeably):

```
prompt$ ./sbin/start-slave.sh 1 spark://Simons-MacBook-Pro.local:7077
-m 6G -c 2
starting org.apache.spark.deploy.worker.Worker, logging to
/Users/simon/Downloads/spark-1.2.1/sbin/./
```

```
logs/spark-simon-org.apache.spark.deploy.worker.Worker-1
-Simons-MacBook-Pro.local.out
```

Let's take a quick glance at the Worker's log file, you should output similar to the following (some of the output has been trimmed for brevity):

```
...
15/03/14 17:59:06 INFO Remoting: Starting remoting
15/03/14 17:59:06 INFO Remoting: Remoting started; listening on
  addresses :[akka.tcp://sparkWorker@localhost:50287]
15/03/14 17:59:06 INFO Remoting: Remoting now listens on addresses:
  [akka.tcp://sparkWorker@localhost:50287]
15/03/14 17:59:06 INFO Utils: Successfully started service
  'sparkWorker' on port 50287.
15/03/14 17:59:07 INFO Worker: Starting Spark worker localhost:50287
  with 2 cores, 6.0 GB RAM
15/03/14 17:59:07 INFO Worker: Spark home:
  /Users/simon/Downloads/spark-1.2.1
15/03/14 17:59:07 INFO Utils: Successfully started service 'WorkerUI'
  on port 8081.
15/03/14 17:59:07 INFO WorkerWebUI: Started WorkerWebUI at
  http://localhost:8081
15/03/14 17:59:07 INFO Worker: Connecting to master spark://Simons-
  MacBook-Pro.local:7077...
15/03/14 17:59:07 INFO Worker: Successfully registered with master
  spark://Simons-MacBook-Pro.local:7077
```

You can see from the logging output that the Worker is communicating with the Master through a Spark-specific protocol channel.

We launch the second worker in the same way as previously, except that the first argument to the launch script is 2 rather than 1 in order to assign a distinct numeric ID to the second worker:

```
prompt$ ./sbin/start-slave.sh 2 spark://Simons-MacBook-Pro.local:7077
  -m 6G -c 2
starting org.apache.spark.deploy.worker.Worker, logging to /
  Users/simon/Downloads/spark-1.2.1/sbin/./logs/
  spark-simon-org.apache.spark.deploy.worker.Worker-2-Simons-
  MacBook-Pro.local.out
```

Some of the content of this second Worker's log file is as follows:

```
15/03/14 18:00:12 INFO Worker: Starting Spark worker localhost:50357
with 2 cores, 6.0 GB RAM
...
15/03/14 18:00:12 INFO WorkerWebUI: Started WorkerWebUI at
http://localhost:8082
15/03/14 18:00:12 INFO Worker: Connecting to master spark://
Simons-MacBook-Pro.local:7077...
15/03/14 18:00:12 INFO Worker: Successfully registered with
master spark://Simons-MacBook-Pro.local:7077
```

You can also see from their logging output that each of the Workers has its own web monitoring page (WorkerWebUI) – the first worker providing its monitor at `http://localhost:8081`, and the second at `http://localhost:8082`, similar to *Figure 7* that follows:



Figure 7: Worker's web monitoring console in its initial state with no application executors created.

If we also now look back at the Master's web console, we will see the two Workers are indeed registered, as in *Figure 8*.



Figure 8: Master's web monitoring console showing both Workers registered.

Note how the Master has listed the total amount of resources at its disposal in the top half of the page: two Workers, four cores and 12 GB of memory. Now that we have our Spark cluster up and running, we just have to connect our R session to it...

Connecting SparkR to a running cluster

Connecting to a running Spark cluster and creating our SparkR context is really very simple, and is done by passing in the Master's spark URL as the value of the `master` parameter. However, we also need to take into consideration how much of the Workers' resources we want to use when running our application.

A Spark cluster is designed to accommodate many running applications by creating separate executors for each application instance with their own private portion of memory. This separation introduces a layer of both safety and security; an executor from one application cannot interfere with the data being processed in another. We can tell Spark how much executor resource to use when we connect to the Master by passing in additional `sparkEnvir` environment arguments as follows:

```
> sc = sparkR.init(master="spark://Simons-MacBook-Pro.local:7077",
appName="MyRApp", sparkEnvir=list(spark.executor.memory="6g"))
```

Easy! You can now verify our R session is connected by once again reviewing the Master's web console, as shown in [Figure 9](#) that follows. You should notice that the application entry is now present for our explicitly named App `MyRApp`, and that both the workers have allocated the 6 GB of memory we had requested for our application.

To round out our view of the web monitoring consoles, [Figure 10](#) and [Figure 11](#) show the executors in use by our application on both of the workers. This is also reflected in the application web monitoring console itself, as in [Figure 12](#) which shows the **Executor** tab including the driver program's JVM executor.

Hopefully, by now you have a fairly complete view of what a Spark cluster looks like on the inside so-to-speak, how to start one up, and how to navigate the various monitoring screens that are available. All of this cluster structure is as depicted earlier in [Figure 5](#). Working with a remotely located Spark cluster running in the cloud is a very similar experience.

Now, however, its time to look at SparkR API in detail, and the various RDD transformations and actions it provides within its 70+ function interface.

Shutdown cluster? Start cluster with ability to shut it down from web console?

Worker states? Separation of worker configuration from application configuration.

Spark on AWS?

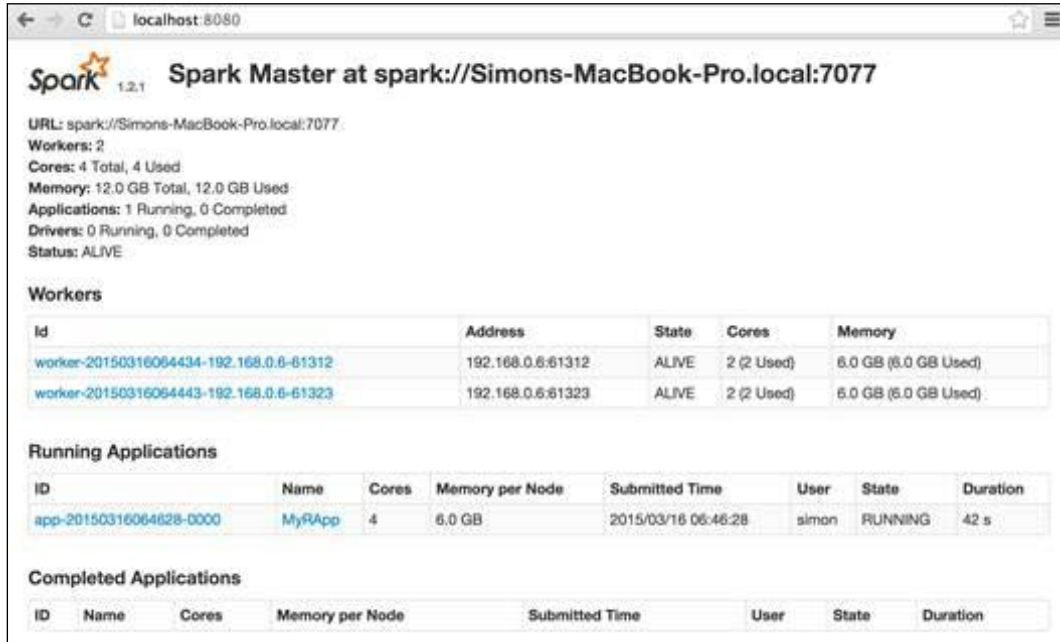


Figure 9: Master's web monitoring console showing our R session as a newly registered application.

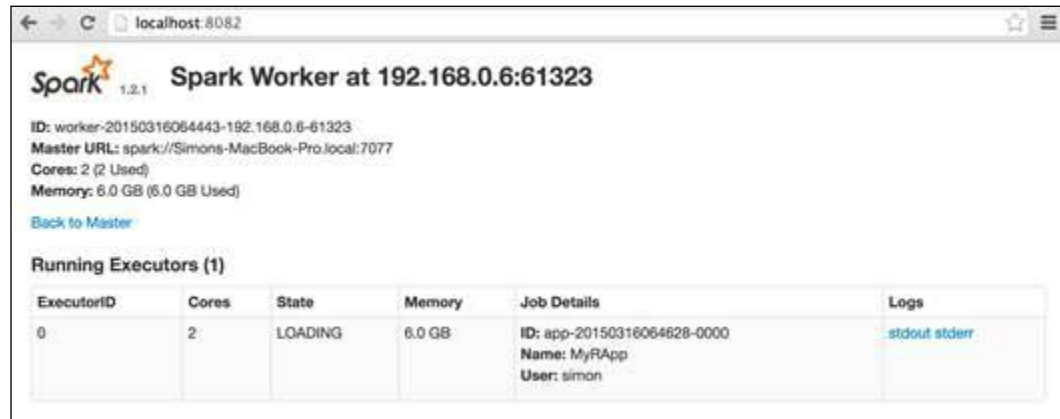


Figure 10: Worker 1's web monitoring console showing the Application Executor JVM its running.



Figure 11: Worker-1's web monitoring console showing the Application Executor JVM its running.

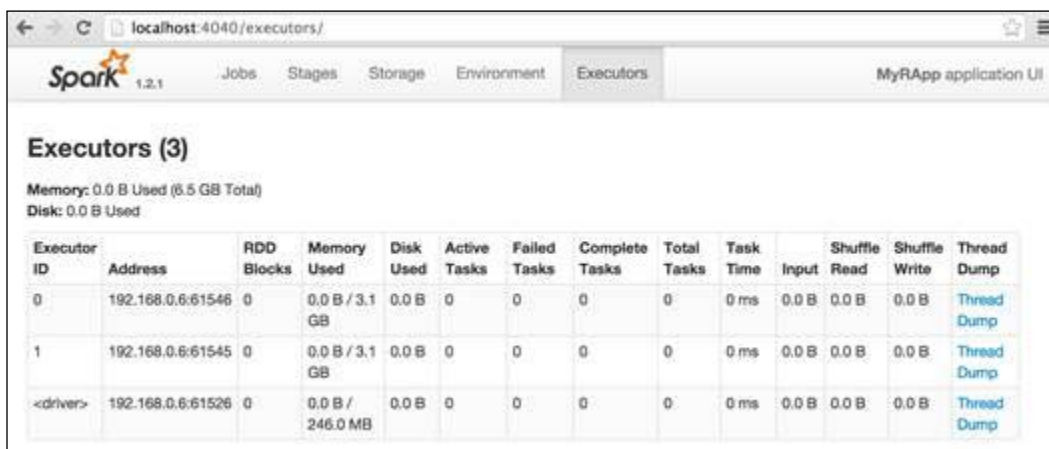


Figure 12: Our application's web monitoring console showing the three Executor JVMs, one from each Worker, and its own driver program JVM executor.

Types of RDD

An RDD is essentially a table of data. Each column of data can be made up of a numeric or string value, but must remain consistent across all rows in the table. The RDD abstraction itself does not name individual columns or provide simple typing information as a **DataTable** in R.



Note:

Spark 1.3 introduces a new abstraction called **Data Frame**, which adds naming and typing metadata to the underlying RDD abstraction, and provides an extended API for easy manipulation of data frames. Spark 1.4, was released in July 2015, is due to include this new Data Frame API for SparkR. Data frames do not detract from RDD itself – they provide additional convenience and some extra efficiency for applying basic operations to tabular data.

SparkR RDD Transformations

The following table provides a short description of each of the RDD transformations available in SparkR. Recall that a transformation is an operation on an RDD that generates another separate RDD, creating a computational dependency chain, which is only evaluated by Spark whenever an action is applied to generate a result for consumption by an application, that is, Spark applies lazy evaluation.

For the mini examples presented in the table, the following R script preamble is assumed:

| RDD Transformation | Description |
|--|-----------------------------|
| aggregateByKey | |
| cogroup | |
| combineByKey | |
| coalesce Related: repartition | Equivalent to repartition() |
| distinct | |
| filterRDD | |
| flatMap | |
| flatMapValues | |
| foldByKey Related action: fold | |
| groupByKey | |
| join fullOuterJoin leftOuterJoin rightOuterJoin | |
| keyBy | |

| RDD Transformation | Description |
|--|--|
| keys | |
| lapply lapplyPartition lapplyPartitionsWithIndex Related: map | Equivalent to map. The duplicated naming reflects the intersection of the R (lapply) world with that of the Hadoop (map/reduce) fraternity. |
| map mapPartitions mapPartitionsWithIndex mapValues Related: lapply | map(), mapPartitions(), and mapPartitionsWithIndex() are all interchangeable with their similarly named lapply() cousins: see previous entry in table for equivalent description. mapValues |
| parallelize numPartitions | |
| partitionBy hashCode | |
| reduceByKey | |
| repartition | |
| sampleRDD | |
| sortBy sortByKey | |
| unionRDD | |
| values | |
| zipWithIndex zipWithUniqueId | |

SparkR RDD Actions

The following table provides a short description and examples of each of the RDD actions available in SparkR. An action returns a result to R from a potentially complex and large chain of RDD transformations. If repeated actions on an RDD are expected, then performance can be significantly improved by persisting the end of chain RDDs and action intermediate RDDs.

For the mini examples presented in the table, the following R script preamble is assumed:

```
> sparkR.init() # Starts a local Spark cluster instance
> rdd123 <- parallelize(sc,c(11,11,11,22,22,33))
```

```

> rdd09 <- parallelize(sc,0:9)
> rdd092 <- repartition(rdd09,2L)
> rddAZ <- parallelize(sc,c('A','B','C','D','E','V','W','X','Y','Z'))
> rddAZ2 <- repartition(rddAZ,2L)
> rddAE <- parallelize(sc,c('A','Z','B','X','D','V','C','Y','W','E'))
> tuples <- list(list('A',10),list('B',20))
> rddABtuples <- parallelize(sc,tuples)
> rddABduples <- parallelize(sc,append(tuples,tuples))

```

| RDD action | Description |
|---------------------------------------|---|
| aggregateRDD | |
| collect (RDD) | collect () returns the contents of the RDD as an R list in its natural order, that is, the order in which the data was presented when the RDD was created. |
| collectPartition(RDD, index: integer) | <pre> > collect(rddAZ2) [[1]] [1] "A" ... [[10]] [1] "Z" </pre> |
| collectAsMap (KV_RDD) | <p>collectPartition() allows you to select the partition of the RDD that is returned: note that partitions are numbered according to JAVA indices, that is, from 0 to $N-1$, as opposed to 1 to N:</p> <pre> > unlist(collectPartition(rdd092,0L)) [1] 0 2 4 6 8 > unlist(collectPartition(rdd092,1L)) [1] 1 3 5 7 9 </pre> <p>collectAsMap() returns the contents of a key-value RDD as an R named-list:</p> <pre> > map <- collectAsMap(rddABtuples) > map\$A [1] 10 > map\$B [1] 20 </pre> |

| RDD action | Description |
|---|---|
| <p>count (RDD) countByKey (KV_RDD) countByValue (RDD)</p> | <p>count () simply returns the number of elements in the RDD, and is equivalent to length (rdd) :</p> <pre>> count (rdd123) [1] 6 > count (distinct (rdd123)) [1] 3</pre> <p>countByKey () counts the number of distinct keys in a key-value RDD, returning a nested list of results:</p> <pre>> countByKey (rddABduples) [[1]] [[1]] [[1]] [1] "A" [[1]] [[2]] [1] 2 [[2]] [[2]] [[1]] [1] "B" [[2]] [[2]] [1] 2</pre> <p>countByValue () counts the number of distinct occurrences of each value in an RDD – useful for creating frequency distributions:</p> <pre>> unlist (countByValue (rdd123)) [1] 11 3 22 2 33 1 # 11=3,22=2,33=1</pre> |
| <p>fold Related: reduce</p> | |
| <p>lookup</p> | |
| <p>maximum (RDD) minimum (RDD)</p> | <p>maximum () and minimum () scan the whole RDD to locate the numerically highest and lowest values:</p> <pre>> minimum (rdd09) [1] 0 > maximum (rddAZ) [1] "Z" > maximum (values (rddABtuples)) [1] 20</pre> |

| RDD action | Description |
|---|-------------|
| reduce Related: fold reduceByKeyLocally Related action: reduceByKey | |
| take takeOrdered takeSample | |
| top | |

SparkR System & Data Management API

The following table deals with, essentially, the "leftovers" in the SparkR API—those operations which are about the messy practicalities of making the clean RDD abstraction work, including system interaction and data management.

| API call | Description |
|---|--|
| broadcast value setBroadcastValue | Use value to access the broadcast variable inside your compute function. |
| cache persist unpersist | Equivalent to persist () |
| checkpoint setCheckpointDir | |
| foreach foreachPartition | |
| includePackage | |
| objectFile saveAsObjectFile | |
| pipeRDD | |
| print.jobj | |
| textFile saveAsTextFile | |