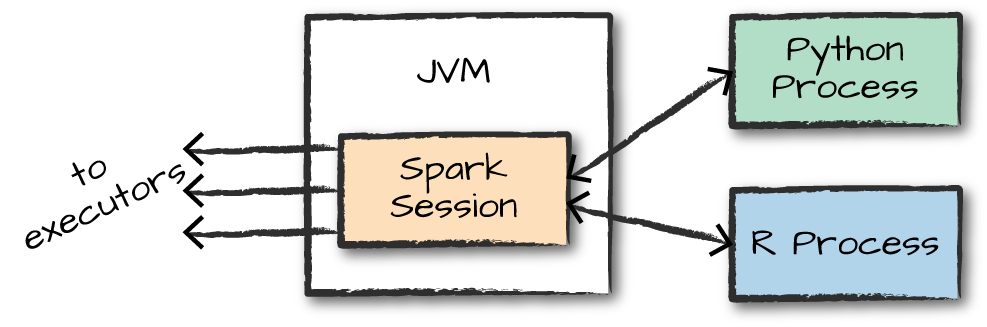
**Chapter 32. Language Specifics: Python (PySpark) and R (SparkR and sparklyr)**

This chapter will cover some of the more nuanced language specifics of Apache Spark. We’ve seen a huge number of PySpark examples throughout the book. In [Chapter 1](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch01.html#s1c1---defining-spark), we discussed at a high level how Spark runs code from other languages. Let’s talk through some of the more specific integrations:

* PySpark
* SparkR
* sparklyr

As a reminder, Figure 32-1 shows the fundamental architecture for these specific languages.



*Figure 32-1. The Spark Driver*

Now let’s cover each of these in depth.

**PySpark**

We covered a ton of PySpark throughout this book. In fact, PySpark is included alongside Scala and SQL in nearly every chapter in this book. Therefore, this section will be short and sweet, covering only the details that are relevant to Spark itself. As we discussed in [Chapter 1](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch01.html#s1c1---defining-spark), Spark 2.2 included a way to install PySpark with pip. Simply, pip install pyspark will make it available as a package on your local machine. This is new, so there may be some bugs to fix, but it is something that you can leverage in your projects today.

**Fundamental PySpark Differences**

If you’re using the structured APIs, your code should run just about as fast as if you had written it in Scala, except if you’re not using UDFs in Python. If you’re using a UDF, you may have a performance impact. Refer back to [Chapter 6](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch06.html#s2c3---working-with-different-types-of-data) for more information on why this is the case.

If you’re using the unstructured APIs, specifically RDDs, then your performance is going to suffer (at the cost of a bit more flexibility). We touch on this reasoning in [Chapter 12](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch12.html#s3c1---rdd-basics), but the fundamental idea is that Spark is going to have to work a lot harder converting information from something that Spark and the JVM can understand to Python and back again. This includes both functions as well as data and is a process known as *serialization*. We’re not saying it never makes sense to use them; it’s just something to be aware of when doing so.

**Pandas Integration**

One of the powers of PySpark is its ability to work across programming models. For instance, a common pattern is to perform very large-scale ETL work with Spark and then collect the (single-machine-sized) result to the driver and then leverage Pandas to manipulate it further. This allows you to use a best-in-class tool for the best task at hand—Spark for big data and Pandas for small data:

**import** **pandas** **as** **pd**

df = pd.DataFrame({"first":range(200), "second":range(50,250)})

sparkDF = spark.createDataFrame(df)

newPDF = sparkDF.toPandas()

newPDF.head()

These niceties make working with data big and small easy with Spark. Spark’s community continues to focus on improving this interoperability with various other projects, so the integration between Spark and Python will continue to improve. For example, at the time of writing, the community is actively working on Vectorized UDFs ([SPARK-21190](https://issues.apache.org/jira/browse/SPARK-21190)), which add a mapBatches API to let you process a Spark DataFrame as a series of Pandas data frames in Python instead of converting each individual row to a Python object. This feature is targeted to appear in Spark 2.3.

**R on Spark**

The rest of this chapter will cover R, Spark’s newest officially supported language. R is a language and environment for statistical computing and graphics. It is similar to the S language and environment developed at Bell Laboratories by John Chambers (of no relation to one of the authors of this book) and colleagues. The R language has been around for decades and is consistently popular among statisticians and those doing research in numerical computing. R is steadily becoming a first-class citizen in Spark and provides the simplest open source interface for distributed computation to the R language.

The popularity of R for performing single-machine data analysis and advanced analytics makes it an excellent complement to Spark. There are two core initiatives to making this partnership a reality: SparkR and sparklyr. These packages take slightly different approaches to provide similar functionality. SparkR provides a DataFrame API similar to R’s data.frame, while sparklyr is based on the popular dplyr package for accessing structured data. You can use whichever you prefer in your code, but over time we expect that the community might converge toward a single integrated package.

We will cover both packages here to let you choose which API you prefer. For the most part, both of these projects are mature and well supported, albeit by slightly different communities. They both support Spark’s structured APIs and allow for machine learning. We will elaborate on their differences in the next sections.

**SparkR**

SparkR is an R package (originating as a collaborative research project between UC Berkeley, Databricks, and MIT CSAIL) that provides a frontend to Apache Spark based on familiar R APIs. SparkR is *conceptually* similar to R’s built-in data.frame API, except for some departures from the API semantics, such as lazy evaluation. SparkR is a part of the official Spark project and is supported as such. See the documentation for SparkR [for more information](http://spark.apache.org/docs/latest/sparkr.html).

**PROS AND CONS OF USING SPARKR INSTEAD OF OTHER LANGUAGES**

The reasons we would recommend that you use SparkR as opposed to PySpark are the following.

* You are familiar with R and want to take the smallest step to leverage the capabilities of Spark:
* You want to leverage R-specific functionality or libraries (say the excellent ggplot2 library) and would like to work with big data in the process.

R is a powerful programming language that provides a lot of advantages over other languages when it comes to certain tasks. However, it has its share of shortcomings like natively working with distributed data. SparkR aims to fill this gap and does a great job enabling users to be successful on both small and large data, in a conceptual way similar to PySpark and Pandas.

**SETUP**

Let’s take a look at how to use SparkR. Naturally, you will need to have R installed on your system to follow along in this chapter. To start up the shell, in your Spark home folder, run **./bin/sparkR** to start SparkR. This will automatically create a SparkSession for you. If you were to run SparkR from RStudio, you would have to do something like the following:

library(SparkR)

spark <- sparkR.session()

Once we’ve started the shell, we can run Spark commands. For instance, we can read in a CSV file like we saw in [Chapter 9](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch09.html#s2c6---data-sources):

retail.data <- read.df(

"/data/retail-data/all/",

"csv",

header="true",

inferSchema="true")

print(str(retail.data))

We can take some rows from this SparkDataFrame and convert them to a standard R data.frame type:

local.retail.data <- take(retail.data, 5)

print(str(local.retail.data))

**KEY CONCEPTS**

Now that we saw some very basic code, let’s reiterate key concepts. First, SparkR is still Spark. Basically, all the tools that you have seen across the entire book apply directly to SparkR. It runs according to the same principles as PySpark and has almost all of the same functionality available as PySpark.

As shown in Figure 32-1, there is a gateway that connects the R process to the JVM that contains a SparkSession, and SparkR converts user code into structured Spark manipulations across the cluster. This makes its efficiency on par with Python and Scala when using the structured APIs. SparkR has *no support* for RDDs or other low-level APIs.

While SparkR is used less than PySpark or Scala, it’s still popular and continues to grow. For those that want to know enough Spark to leverage SparkR effectively, we recommend reading the following section, along with Parts [I](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/part01.html#part1) and [II](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/part02.html#part2) of this book. When working through those other chapters, feel free to try and use SparkR in place of Python or Scala. You’ll see that once you get the hang of it, it’s easy to translate between the various languages.

The rest of this chapter will explain the most important differences between SparkR and “standard” R to make it easier to be productive with SparkR faster.

The first thing we should cover is the different between local types and Spark types. A data.frame type’s core difference with the Spark version is that it is available in memory and is usually directly available in that particular process. A SparkDataFrame is just a logical representation of a series of manipulations. Therefore when we manipulate a data.frame, we’ll see our results right away. On a SparkDataFrame, we are going to logically manipulate the data using the same *transformation* and *action* concepts that we saw throughout the book.

Once we have a SparkDataFrame, we can collect it to a data.frame similar to how we can read in data using Spark. We can also collect it into a local data.frame with the following code (using the SparkDataFrame we created in “Setup”):

*# collect brings it from Spark to your local environment*

collect(count(groupBy(retail.data, "country")))

*# createDataFrame comverts a data.frame*

*# from your local environment to Spark*

This difference is of consequence for end users. Certain functions or assumptions that apply to local data.frames do not apply in Spark. For instance, we cannot index a SparkDataFrame according to a particular row. Additionally, we cannot change point values in a SparkDataFrame but can do that in a local data.frame.

**FUNCTION MASKING**

One frequent “gotcha” when users come to SparkR is that certain functions are masked by SparkR. When I imported SparkR, I received the following message:

The following objects are masked from ‘package:stats’:

cov, filter, lag, na.omit, predict, sd, var, window

The following objects are masked from ‘package:base’:

as.data.frame, colnames, ...

This means that if we wish to call these masked functions, we need to be explicit about the package that we’re calling them from or at least understand which function masks another. The ? can be helpful in determining these conflicts:

?na.omit *# refers to SparkR due to package loading order*

?stats::na.omit *# refers explicitly to stats*

?SparkR::na.omit *# refers explicitly to sparkR's null value filtering*

**SPARKR FUNCTIONS ONLY APPLY TO SPARKDATAFRAMES**

One implication of function masking is that functions that worked on objects previously may no longer work on them after you bring in the SparkR package. This is because SparkR functions only apply on Spark objects. For instance, we cannot use the sample function on a standard data.frame because Spark takes that function name:

sample(mtcars) *# fails*

What you have to do instead is explicitly use the base sample function. Additionally the function signatures differ between the two functions, which means that even if you are familiar with the syntax and argument order for one particular library, it does not necessarily mean it’s the same order for SparkR:

base::sample(some.r.data.frame) *# some.r.data.frame = R data.frame type*

**DATA MANIPULATION**

Data manipulation in SparkR is conceptually the same as Spark’s DataFrame API in other languages. The core difference is in the syntax, largely due to us running R code and not another language. Aggregations, filtering, and many of the functions that you can find in the other chapters throughout this book are also available in R. For the most part, you can look at the names of functions or manipulations that you find throughout this book and find out if they are available in SparkR by running *?<function-name>*. This should work the vast majority of the time, as there is good coverage of structured SQL functions:

?to\_date *# to Data DataFrame column manipulation*

SQL is largely the same. We can specify SQL commands that we can then manipulate as DataFrames. For instance, we can find all tables that contain the word “production” in them:

tbls <- sql("SHOW TABLES")

collect(

select(

filter(tbls, like(tbls$tableName, "%production%")),

"tableName",

"isTemporary"))

We can also use the popular magrittr package to make this code more readable, leveraging the piping operator to chain our transformations in a more functional and readable syntax:

library(magrittr)

tbls %>%

filter(like(tbls$tableName, "%production%")) %>%

select("tableName", "isTemporary") %>%

collect()

**DATA SOURCES**

SparkR supports all of the data sources that Spark supports, including third-party packages. We can see in the following snippet that we simply specify the options using a slightly different syntax:

retail.data <- read.df(

"/data/retail-data/all/",

"csv",

header="true",

inferSchema="true")

flight.data <- read.df(

"/data/flight-data/parquet/2010-summary.parquet",

"parquet")

Refer back to [Chapter 9](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch09.html#s2c6---data-sources) for more information.

**MACHINE LEARNING**

Machine learning is a fundamental part of the R language, as well as of Spark. From SparkR there is a decent availability of Spark MLlib algorithms. Typically they arrive in R one or two versions after they are introduced in Scala or Python. As of Spark 2.1, the following algorithms are supported in SparkR:

* spark.glm or glm: Generalized linear model
* spark.survreg: Accelerated failure time (AFT) survival regression model
* spark.naiveBayes: Naive Bayes model
* spark.kmeans: 𝘬-means model
* spark.logit: Logistic regression model
* spark.isoreg: Isotonic regression model
* spark.gaussianMixture: Gaussian mixture model
* spark.lda: Latent Dirichlet allocation (LDA) model
* spark.mlp: Multilayer perceptron classification model
* spark.gbt: Gradient boosted tree model for regression and classification
* spark.randomForest: Random forest model for regression and classification
* spark.als: Alternating least squares (ALS) matrix factorization model
* spark.kstest: Kolmogorov-Smirnov test

Under the hood, SparkR uses MLlib to train the model, which means that most everything covered in [Part VI](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/part06.html#part6) is relevant for SparkR users. Users can call summary to print a summary of the fitted model, predict to make predictions on new data, and write.ml/read.ml to save/load fitted models. SparkR supports a subset of the available R formula operators for model fitting, including ~, ., :, +, and -. Here’s an example of running a simple regression on the retail dataset:

model <- spark.glm(retail.data, Quantity ~ UnitPrice + Country,

family='gaussian')

summary(model)

predict(model, retail.data)

write.ml(model, "/tmp/myModelOutput", overwrite=T)

newModel <- read.ml("/tmp/myModelOutput")

The API is consistent across models, although not all models support detailed summary outputs like we saw with glm. For more information about specific models or preprocessing techniques, see the corresponding chapters in [Part VI](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/part06.html#part6).

While this pales in comparison to R’s extensive collection of statistical algorithms and analysis libraries, many users do not require the scale that Spark provides for the actual training and usage of their machine learning algorithms. Users have the opportunity to build training sets on large data using Spark and then collect that dataset to their local environment for training on a local data.frame.

**USER-DEFINED FUNCTIONS**

In SparkR, there are several ways of running user-defined functions. A *user-defined function* is one that is created in the native language and run on the server in that same native language. These run, for the most part, in the same way that a Python UDF runs, by performing serialization into and out of the JVM of the function.

The different kinds of UDFs you can define are as follows:

First, spark.lapply lets you run multiple instances of a function in Spark on different parameter values provided in an R collection. This is a great way of performing grid search and comparing the results:

families <- c("gaussian", "poisson")

train <- function(family) {

model <- glm(Sepal.Length ~ Sepal.Width + Species, iris, family = family)

summary(model)

}

*# Return a list of model's summaries*

model.summaries <- spark.lapply(families, train)

*# Print the summary of each model*

print(model.summaries)

Second, dapply and dapplyCollect let you process SparkDataFrame data using custom code. In particular, these functions will take each partition of the SparkDataFrame, convert it to an R data.frame inside of an executor, and then call your R code over that partition (represented as an R data.frame). They will then return the results: a SparkDataFrame for dapply or a local data.frame for dapplyCollect.

To use dapply, which returns a SparkDataFrame, you must specify the output schema that will result from the transformation so that Spark understands what kind of data you will return. For example, the following code will allow you to train a local R model per partition in your SparkDataFrame, assuming you partition your data according to the correct keys:

df <- withColumnRenamed(createDataFrame(as.data.frame(1:100)), "1:100", "col")

outputSchema <- structType(

structField("col", "integer"),

structField("newColumn", "double"))

udfFunc <- function (remote.data.frame) {

remote.data.frame['newColumn'] = remote.data.frame$col \* 2

remote.data.frame

}

*# outputs SparkDataFrame, so it requires a schema*

take(dapply(df, udfFunc, outputSchema), 5)

*# collects all results to a, so no schema required.*

*# however this will fail if the result is large*

dapplyCollect(df, udfFunc)

Finally, the gapply and gapplyCollect functions apply a UDF to a group of data in a fashion similar to dapply. In fact, these two methods are largely the same, except that one operates on a generic SparkDataFrame, and the other applies to a grouped DataFrame. The gapply function will apply this function on a per-group basis and by passing in the key as the first parameter to the function that you define. In this way, you can be sure to have a function customized according to each particular group:

local <- as.data.frame(1:100)

local['groups'] <- c("a", "b")

df <- withColumnRenamed(createDataFrame(local), "1:100", "col")

outputSchema <- structType(

structField("col", "integer"),

structField("groups", "string"),

structField("newColumn", "double"))

udfFunc <- function (key, remote.data.frame) {

if (key == "a") {

remote.data.frame['newColumn'] = remote.data.frame$col \* 2

} else if (key == "b") {

remote.data.frame['newColumn'] = remote.data.frame$col \* 3

} else if (key == "c") {

remote.data.frame['newColumn'] = remote.data.frame$col \* 4

}

remote.data.frame

}

*# outputs SparkDataFrame, so it requires a schema*

take(gapply(df,

"groups",

udfFunc,

outputSchema), 50)

gapplyCollect(df,

"groups",

udfFunc)

SparkR will continue to grow as a part of Spark; and if you’re familiar with R and a little bit of Spark, this can be a very powerful tool.

**sparklyr**

sparklyr is a newer package from the RStudio team based on the popular dplyr package for structured data. This package is fundamentally different from SparkR and its authors take a more opinionated stance toward what the integration between Spark and R should do. This means that sparklyr sheds some of the Spark concepts that are available throughout this book, like the SparkSession, and uses its own ideas instead. For some, this means that sparklyr takes a R-first approach instead of SparkR’s approach of closely matching Python and Scala APIs. That approach speaks to its origins as a framework; sparklyr was created within the R community by the folks at RStudio (the popular R IDE), rather than being created by the Spark community. Whether sparklyr’s or SparkR’s approach is better or worse completely depends on the end user’s preference.

In short, sparklyr provides an improved experience for R users familiar with dplyr, with slightly less overall functionality than SparkR (which may change over time). Specifically, sparklyr provides a complete dplyr backend to Spark, making it easy to take the dplyr code that you run today on your local machine and make it distributed. The implication of the dplyr backend architecture is that the same functions you use on local data.frame objects apply in a distributed manner to distributed Spark DataFrames. In essence, scaling up requires no code changes. Since functions apply to both single node and distributed DataFrames, this architecture addresses one of the core challenges with SparkR today, where function masking can lead to strange debugging scenarios. In addition, this architectural choice makes sparklyr an easier transition than simply using SparkR. Like SparkR, sparklyr is an evolving project; and when this book is published, the sparklyr project will have evolved further. For the most up-to-date reference, you should see the [sparklyr website](http://spark.rstudio.com/index.html). The following sections provide a lightweight comparison and won’t go into depth on this particular project. Let’s get started with some hands-on examples of sparklyr. The first thing we need to do is install the package:

install.packages("sparklyr")

library(sparklyr)

**KEY CONCEPTS**

sparklyr ignores some of the fundamental concepts that Spark has and that we discussed throughout this book. We posit that this is because these concepts are unfamiliar (and potentially irrelevant) to the typical R user. For instance, rather than a SparkSession, there’s simply spark\_connect, which allows you to connect to a Spark cluster:

sc <- spark\_connect(master = "local")

The returned variable is a remote dplyr data source. This connection, even though it resembles a SparkContext, is *not* the same SparkContext we mentioned in this book. This is a purely sparklyr concept that represents a Spark cluster connect. This function is largely the entire interface for how you will define configurations that you would like to use in your spark environment. Through this interface, you can specify initialization configurations for the spark cluster as a whole:

spark\_connect(master = "local", config = spark\_config())

This works by using the config package in R to specify the configurations you would like to set on your Spark cluster. These details are covered in the [sparklyr deployment documentation](http://spark.rstudio.com/deployment.html).

Using this variable, we can manipulate remote Spark data from a local R process, thus the result of spark\_connect performs roughly the same administrative role for end users as a SparkContext.

**NO DATAFRAMES**

sparklyr ignores the concept of a unique SparkDataFrame type. Instead it leverages tables (which are still mapped to DataFrames inside Spark) similar to other dplyr data sources and allows you to manipulate those. This aligns more with the typical R workflow, which is to use dplyr and magrittr to functionally define transformations from a source table. However, it means that some of Spark’s built-in functions and APIs may not be accessible unless dplyr also supports them.

**DATA MANIPULATION**

Once we connect to our cluster, we can run all the available dplyr functions and manipulations as if they were a local dplyr data.frame. This architectural choice gives those familiar with R the ability to do the same transformations using the same code, at scale. This means there’s no new syntax or concepts for R users to learn.

While sparklyr does improve the R end-user experience, it comes at a cost of reducing the overall power available to sparklyr users, since the concepts are R concepts, not necessarily Spark concepts. For instance, sparklyr does not support user-defined functions that you can create and apply in SparkR using dapply, gapply, and lapply. As sparklyr continues to mature, it may add this sort of functionality, but at the time of this writing this capability does not exist. sparklyr is under very active development and more functionality is being added so refer to the [sparklyr homepage](https://spark.rstudio.com/index.html) for more information.

**EXECUTING SQL**

While there is less direct Spark integration, users can execute arbitrary SQL code against the cluster using the DBI library corresponding to almost the same SQL interface we have seen in previous chapters:

library(DBI)

allTables <- dbGetQuery(sc, "SHOW TABLES")

This SQL interface provides a convenient lower-level interface to the SparkSession. For instance, users can use DBI’s interface to set Spark SQL specific properties on the Spark cluster:

setShufflePartitions <- dbGetQuery(sc, "SET spark.sql.shuffle.partitions=10")

Unfortunately, neither DBI nor spark\_connect does not give you an interface for setting Spark-specific properties, which you are going to have to specify when you connect to your cluster.

**DATA SOURCES**

Users can leverage many of the same data sources available in Spark using sparklyr. For example, you should be able to create table statements using arbitrary data sources. However, only CSV, JSON, and Parquet formats are supported as first-class citizens using the following function definitions:

spark\_write\_csv(tbl\_name, location)

spark\_write\_json(tbl\_name, location)

spark\_write\_parquet(tbl\_name, location)

**MACHINE LEARNING**

sparklyr also has support for some of the core machine learning algorithms that we saw in previous chapters. A list of the supported algorithms (at the time of this writing) includes:

* ml\_kmeans: 𝘬-means clustering
* ml\_linear\_regression: Linear regression
* ml\_logistic\_regression: Logistic regression
* ml\_survival\_regression: Survival regression
* ml\_generalized\_linear\_regression: Generalized linear regression
* ml\_decision\_tree: Decision trees
* ml\_random\_forest: Random forests
* ml\_gradient\_boosted\_trees: Gradient-boosted trees
* ml\_pca: Principal components analysis
* ml\_naive\_bayes: Naive-Bayes
* ml\_multilayer\_perceptron: Multilayer perceptron
* ml\_lda: Latent Dirichlet allocation
* ml\_one\_vs\_rest: One versus rest (allowing you to make a binary classifier into a multiclass classifier)

However, development does continue, so check [MLlib](http://spark.rstudio.com/mllib.html) for more information.

**Conclusion**

SparkR and sparklyr are areas of rapid growth in the Spark project, so visit their websites to find out the latest updates about each one. Moreover, the entire Spark project continues to grow as new members, tools, integrations, and packages join the community. The next chapter will discuss the Spark community and some of the other resources available to you.