The R in Spark: Learning Apache Spark with R

Javier Luraschi

2018-05-28

Contents

The R in Spark

In this book you will learn Apache Spark using R with sparklyr. The book intends to take someone unfamiliar with Spark or R and help them become intermediate users by teaching a set of tools, skills and practices applicable to data science.

Chapter 1

Introduction

This chapters covers the historical background that lead to the development of Spark, introduces R in the context of Spark and `sparklyr` as a project bridging Spark and R.

1.1 Background

Humans have been storing, retrieving, manipulating, and communicating information since the Sumerians in Mesopotamia developed writing in about 3000 BC. Based on the storage and processing technologies employed, it is possible to distinguish four distinct phases development: pre-mechanical (3000 BC – 1450 AD), mechanical (1450–1840), electromechanical (1840–1940), and electronic (1940–present).

As humanity moves from traditional industries to an economy based on information technology our footprint of digital information has kept growing at exponential rates:

With the ambition to provide a searchable tool to all this new digital information, many companies attempted to provide such functionality with what we now know as web search or search engines. Managing information at this scale was a challenging problem that companies had to tackle from the very beginning. Given the vast amount of digital information, search engiens were unble to store all the web page information required to support web searches in a single computer. This meant that they had to split information across many machines, which was accomlished by splitting this data and storing it as many files across many machines, this approach became known as the Google File System from a research paper published in 2003 by Google which has served for others to build on.

One year later, in 2004, Google published a new paper describing how to perform operations across the Google File System, this approach came to be known as **MapReduce**. As you would expect, there are two operations in MapReduce: Map and Reduce. We can think of the mapping operation as a way to transform each file into a new file and, reuduce as a way of combining two files into a new one. It happens to be the case that using these two operations is sufficient to perform interesting operations; for instance, MapReduce can be used to rank web pages efficietly across a cluster of machines.

Since the papers were released by Google, a team in Yahoo worked on implementing the Google File System and MapReduce as free open source projects. This project was released in 2006 as **Hadoop** and the Google File System became implemented as the Hadoop File System, or HDFS for short. The Hadoop project made distributed file-based computing accessible to many users and organizations.

While Hadoop provided support to perform map/reduce operations over a distributed file system, it still required each map/reduce operation to be written with code every time a data analysys was run. The **Hive** project, released in 2008 by Facebook, brought Structured Query Language (SQL) support to Hadoop. This meant that data analysis could now be performed at large-scale without the need to write code for each

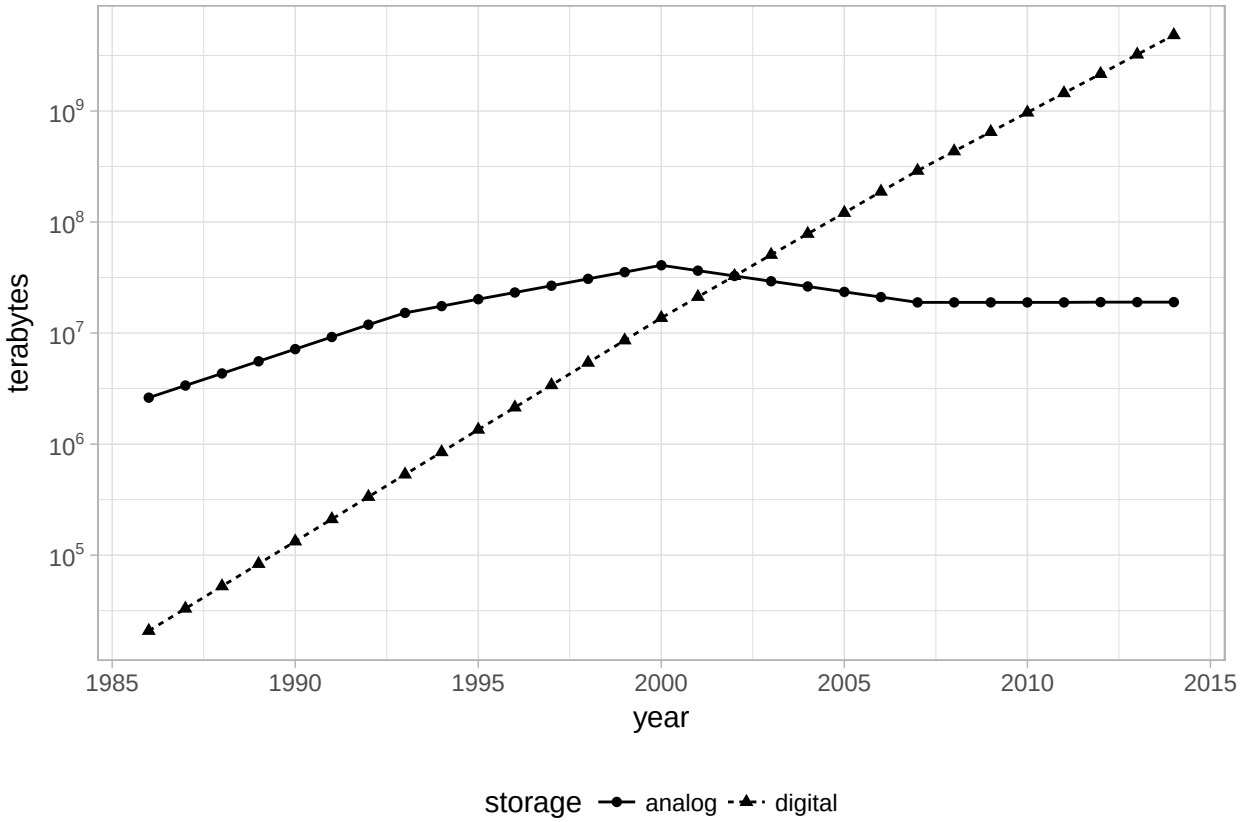


Figure 1.1: World's capacity to store information.

map/reduce operation, but instead, one could write generic data analysis statements that are much easier to understand and write.

1.2 Spark

While Hadoop with Hive was a powerful tool, it was still working over a distributed file system and was dependent on map/reduce operations. This meant that it was running using disk drives which tend to be significantly slower than using a computer's memory. In 2009, the **Apache Spark** projects starts in Berkeley to improve over Hadoop. Specifically, by making use of memory (instead of disk drives) and by providing a richer set of verbs beyond map/reduce, this allowed it to be much faster and generic than its predecessor. For instance, one can sort 100TB of data in 72min and 2100 computers using Hadoop, but only 206 computers in 23min using Spark. Spark was build using the Scala programming language, but interfaces to other programming languages are also provided today. Spark was released as an open source project in 2010 with the scope of the project defined as follows:

> "Apache Spark is a fast and general engine for large-scale data processing."
>
> — spark.apache.org

meaning that Spark is a tool designed to support:

- **Data Processing**: Data processing is the collection and manipulation of items of data to produce meaningful information (French, 1996).
- **Large-Scale**: What *large* means is hard to quantify, but one can interpret this as cluster-scale instead, which represents a set of connected computers that work together.
- **General**: Spark optimizes and executes parallel generic code, as in, there is no restriction as to what type of code one can write in Spark.
- **Fast**: Spark is much faster than its predecessor by making efficient use of memory to speed data access while running algorithms at scale.

Spark is good at tackling large-scale data processing problems, this usually known as **big data** (data sets that are more voluminous and complex that traditional ones, but also is good at tackling large-scale computation problems, known as **big compute** (tools and approaches using a large amount of CPU and memory resources in a coordinated way). There is a third problem space where data nor compute are necessarily large scale and yet, there are significant benefits from using the same tools.

Big data and big compute problems are usually easy to spot, if the data does not fit into a single machine, you might have a big data problem; if the data fits into a single machine but a process over the data takes days, weeks or months to compute, you might have a big compute problem.

For the third problem space, there are a few use cases this breaks to:

1. **Velocity**: One can have a dataset of 10GB in size and a process that takes 30min to run over this data, this is by no means big-compute nor big-data; however, if a data scientist is researching ways to improve accuracy for their models, reducing the runtime down to 3min it's a 10X improvement, this improvement can lead to significant advances and productivity gains by increasing the velocity at which one can analyze data.

2. **Variety**: One can have an efficient process to collect data from many sources into a single location, usually a database, this process could be already running efficiently and close to realtime. Such processes are known at ETL (Extract-Transform-Load); data is extracted from multiple sources, transformed to the required format and loaded in a single data store. While this has worked for years, the tradeoff from this system is that adding a new data source is expensive, the system is centralized and tightly controlled. Since making changes to this type of systems could cause the entire process to come to a halt, adding new data sources usually takes long to be implemented. Instead, one can store all data its natural format and process it as needed using cluster computing, this architecture is currently known as a data lake.

Some people refer to some of these benefits as the four 'V's of big data: Velocity, Variety, Volume and Veracity. Others have gone as far as expending this to five or even as the 10 Vs of Big Data. Mnemonics set aside, cluster computing is being used today in more innovative ways and and is not uncommon to see organizations experimenting with new workflows and a variety of tasks that were traditionally uncommon for cluster computing. Much of the hype attributed to big data falls into this space, where some will argue that everything should be considered big data and where others will argue than almost nothing should. My hope is that this book will help you understand the opportunities and limitations of Apache Spark with R.

1.3 R

R is a computing language with it's inception dating back to Bell Laboratories. At that time, computing was done by calling Fortran subroutines which, apparently, were not pleasant to deal with. The S computing language was designed as an interface language to support higher abstractions to perform statistical computing over existing subroutines:

```
knitr::include_graphics("images/01-intro-s-algorithm-interface.png")
```

R is a modern and free implementation of S, specifically:

> R is a programming language and free software environment for statistical computing and graphics.
>
> — The R Project for Statistical Computing

There are two strong arguments for choosing R over other computing languages while working with data:

- The **R Language** was designed by statisticians for statisticians, meaning, this is one of the few successful languages designed for non-programmers; so learning R will probably feel more natural. Additionally, since the R language was designed to be an interface to other tools and languages, R allows you to focus more on modeling and less on the peculiarities of computer science and engineering.
- The **R Community** provides a rich package archive provided by CRAN (The Comprehensive R Archive Network) which allows you to install ready-to-use packages to perform many tasks, most notably, high-quality statistic models with many only available in R. In addition, the R community is a welcoming and active group of talented individuals motivated to help you succeed. Many packages provided by the R community make R, by far, the place to do statistical computing.

One can argue to what degree other fields, like machine learning, overlap with statistics; so far, most people will argue that the overlap is non-trivial. Similar arguments can be made for data science, big data, deep learning and beyond. With the continuous rise of popularity of R, I can only expect R's influence and scope to keep growing over time; we can take a look at the historic downloads of R packages in CRAN to get some sense of R's recent growth (see Section 12.8):

1.4 sparklyr

Back in 2016, there was a need in the R community to support Spark through an interface compatible with other R packages, easy to use and available in CRAN. To this end, development of `sparklyr` started in 2016 by RStudio under JJ Allaire, Kevin Ushey and Javier Luraschi, version 0.4 was released in summer during the *useR!* conference, this first version added support for `dplyr`, `DBI`, modeling with `MLlib` and an extensible API that enabled extensions like H2Os rsparkling package. Since then, many new features have been added and support across many Spark distributions and cloud services has made available.

Officially,

> `sparklyr` is an R interface for Apache Spark.
>
> —github.com/rstudio/sparklyr

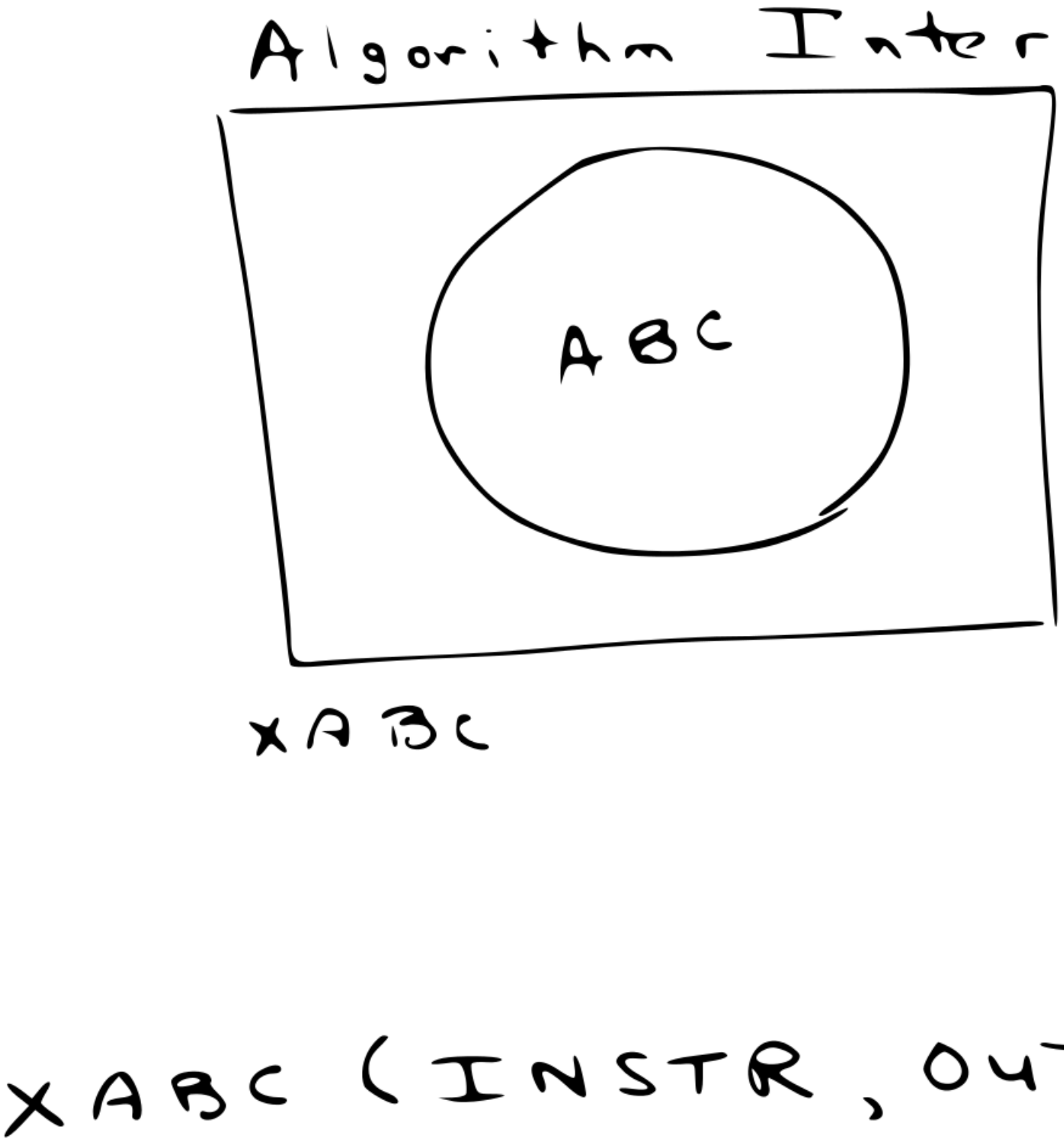


Figure 1.2: Interface language diagram by John Chambers from useR 2016.

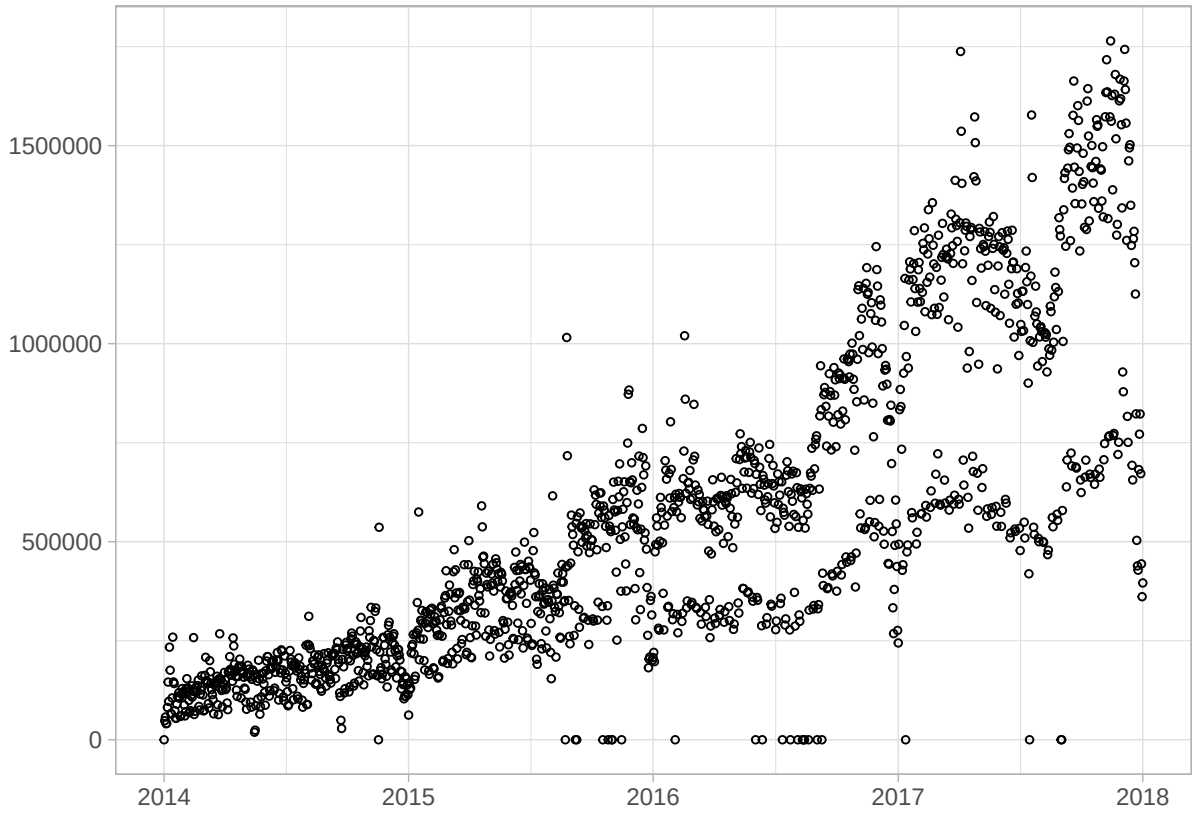


Figure 1.3: Daily downloads of CRAN packages.

It's available in CRAN and works like any other CRAN package, meaning that: it's agnostic to Spark versions, it's easy to install, it serves the R community, it embraces other packages and practices from the R community and so on. It's hosted in GitHub under https://github.com/rstudio/sparklyr and licensed under Apache 2.0 which is allows you to clone, modify and contribute back to this project.

While thinking of who and why should use `sparklyr`, the following roles come to mind:

- **New Users**: For new users, I'm going to argue that `sparklyr` is the best way to get started with Spark. My hope is that the first chapters of this book will get you up running with ease and set you up for long term success.
- **Data Scientists**: I do believe, strongly, that `sparklyr` in combination with many other R packages and tools is the most productive environment for the modern data scientists. `sparklyr` allows support for high-level tasks and low-level extensibility mechanisms to match the needs and skills of every data scientists.
- **Expert Users**: For those users that are already immersed in Spark and can write code natively inScala, I'm going to argue that making their work available as an `sparklyr` extension is very desirable for them and the community. The R community is one of the most welcoming and supportive communities I've known, so I can't think of better ways of helping the expert users share their work and knowledge than by making it available in CRAN to R community.

This book is titled "The R in Spark" as a way to describe and teach that area of overlap between Spark and R. The R package that represents this overlap is `sparklyr`; however, the overlap goes beyond a package. It's an overlap of communities, expectations, future directions, packages and package extensions as well. Naming this book `sparklyr` or "Introduction to sparklyr" would have left behind a much more exciting opportunity, an opportunity to present this book as an intersection of the R and Spark communities. Both are solving very similar problems with a set of different skills and backgrounds; therefore, it is my hope that `sparklyr` can be a fertile ground for innovation, a welcoming place to newcomers, a productive place for experienced data scientists and an open community where cluster computing and modeling can come together.

Here are some resources to help you get involved:

- **Documentation**: This should be your entry point to learn more about sparklyr, the documentation is kept up to date with examples, reference functions and many more relevant resources (https://spark.rstudio.com).
- **Github**: If you believe something needs to get fixed, open a GitHub issue or send us a pull request (https://github.com/rstudio/sparklyr).
- **Stack Overflow**: For general questions, Stack Overflow is a good place to start (stackoverflow.com/tags/sparklyr).
- **Gitter**: For urgent issues or to keep in touch you can chat with us in Gitter (https://gitter.im/rstudio/sparklyr).

Chapter 2

Installation

From R, installing and launching a local Spark cluster using `sparklyr` is as easy as running:

```
spark_install()
sc <- spark_connect(master = "local")
```

However, to make sure we can all run the code above and understand it, this chapter will walk you through installing the prerequisites, installing Spark and connecting to a local Spark cluster.

2.1 Prerequisites

As briefly mentioned in the Introduction chapter, R is a programming language that can run in many platforms and environments. Most people making use of a programming language also choose tools to make them more productive in it; for R, RStudio would be such tool. Strictly speaking, RStudio is an Integrated Development Environment or IDE for short, which also happens to support many platforms and environments. R and RStudio are the free software tools this book will make use of and therefore, I strongly recommend you get those installed if you haven't done so already.

Additionally, since Spark is build in the Scala programming language which is run by the Java Virtual Machine, you also need to install Java 7 or newer in your system. It is likely that your system already has Java installed, but is probably worth updating with the steps bellow.

2.1.1 Install R

From r-project.org, download and launch the installer for your platform, Windows, Macs or Linux available.

2.1.2 Install Java

From oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html, download and launch the installer for your platform, Windows, Macs or Linux available. While installing the JRE (Java Runtime Environment) is sufficient for most operations, in order to build extensions you will need the JDK (Java Developer Kit); therefore, I rather recommend installing the JDK in the first place.

Starting with Spark 2.1, Java 8 is required; however, previous versions of Spark support Java 7. Regardless, we recommend installing Java 8 as described in this chapter

[Home]

Download

CRAN

R Project

About R
Logo
Contributors
What's New?
Reporting Bugs
Development Site
Conferences
Search

R Foundation

Foundation
Board
Members
Donors
Donate

Help With R

Getting Help

Documentation

Manuals
FAQs
The R Journal
Books

The R Project for Stat
Computing

Getting Started

R is a free software environment for statistical computing
and runs on a wide variety of UNIX platforms, Windows a
please choose your preferred CRAN mirror.

If you have questions about R like how to download and
the license terms are, please read our answers to freque
you send an email.

News

- **R version 3.5.0 (Joy in Playing)** has been released o
- **R version 3.4.4 (Someone to Lean On)** has been rel
- **useR! 2018** (July 10 - 13 in Brisbane) is open for regis
  https://user2018.r-project.org
- **The R Journal Volume 9/2** is available.
- **useR! 2017** took place July 4 - 7 in Brussels **https://**
- The **R Logo** is available for download in high-resoluti

Figure 2.1: The R Project for Statistical Computing.

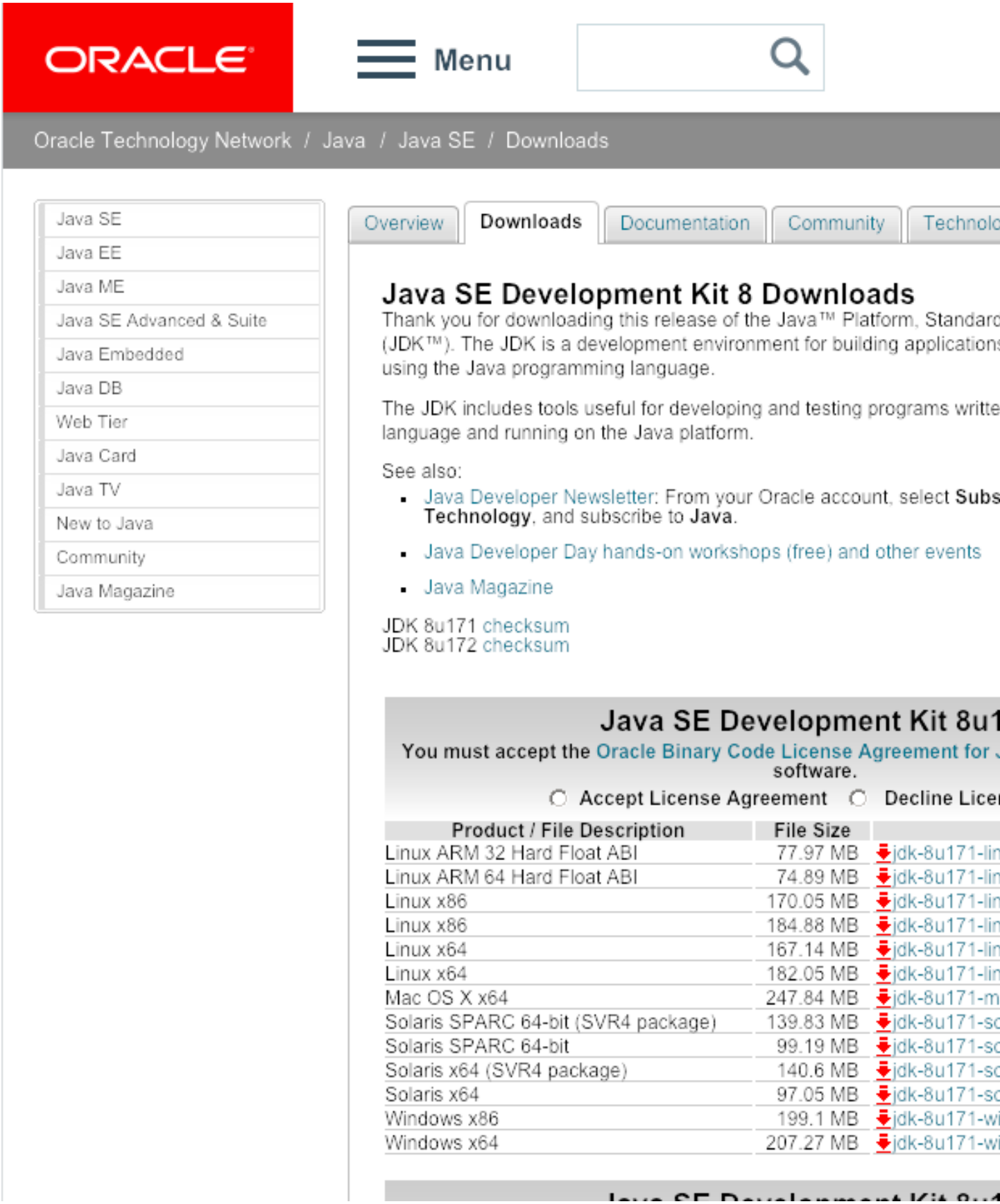


Figure 2.2: Java Download.

2.1.3 Install RStudio

While installing RStudio is not strictly required to work with `sparklyr` in R, it will make you much more production and therefore, I highly recommend you take the time to install RStudio from rstudio.com/products/rstudio/download/, then download and launch the installer for your platform, Windows, Macs or Linux available.

After launching RStudio, identify the Console panel since this is where most of the code will be executed in this book. For additional learning resources on R and RStudio consider visiting: rstudio.com/online-learning/.

2.1.4 Install sparklyr

First of all, we would want to install `sparkylr`. As many other R packages, `sparklyr` is available in CRAN and can be easily installed as follows:

```
install.packages("sparklyr")
```

The CRAN release of `sparklyr` contains the most stable version and it's the recommended version to use; however, for those that need or might want to try newer features being developed in `sparklyr` you can install directly from GitHub using the `devtools` package. First install the `devtools` package and then `sparklyr` as follows:

```
install.packages("devtools")
devtools::install_github("rstudio/sparklyr")
```

2.2 Installing Spark

Start by loading `sparklyr`,

```
library(sparklyr)
```

This will makes all `sparklyr` functions available in R, which is really helpful; otherwise, we would have to run each `sparklyr` command prefixed with `sparklyr::`.

As mentioned, Spark can be easily installed by running `spark_install()`; this will install the latest version of Spark locally in your computer, go ahead and run `spark_install()`. Notice that this command requires internet connectivity to download Spark.

```
spark_install()
```

All the versions of Spark that are available for installation can be displayed with `spark_available_versions()`:

```
spark_available_versions()
```

```
##    spark
## 1  1.6.3
## 2  1.6.2
## 3  1.6.1
## 4  1.6.0
## 5  2.0.0
## 6  2.0.1
## 7  2.0.2
## 8  2.1.0
## 9  2.1.1
## 10 2.2.0
```

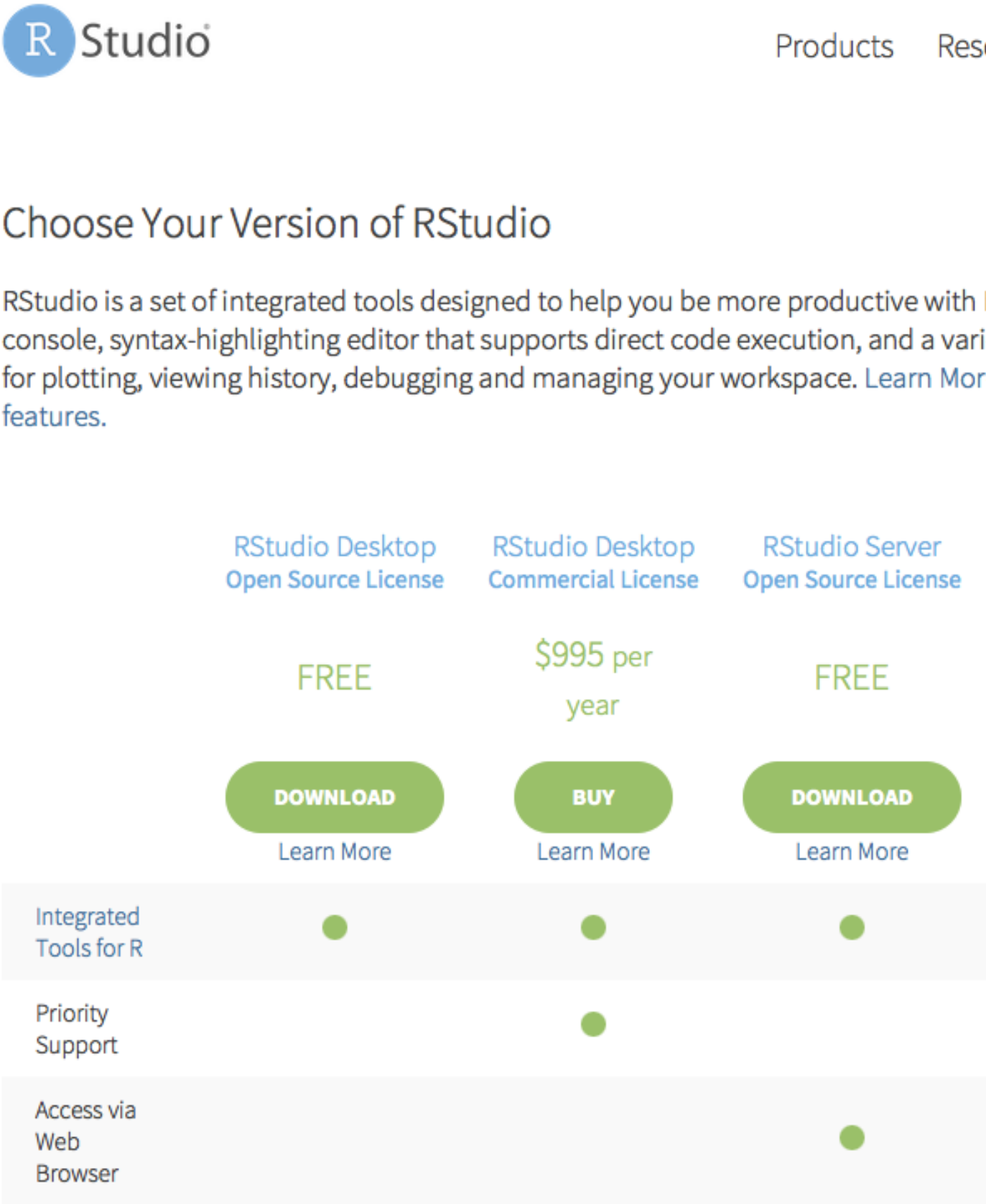


Figure 2.3: RStudio Downloads.

```
## 11 2.2.1
## 12 2.3.0
```

A specific version can be installed using the Spark version and, optionally, by also specifying the Hadoop version. For instance, to install Spark 1.6.3, we would run `spark_install("1.6.3")`.

You can also check which versions are installed by running:

```
spark_installed_versions()
```

Finally, in order to uninstall an specific version of Spark you can run `spark_uninstall()` by specifying the Spark and Hadoop versions, for instance:

```
spark_uninstall(version = "1.6.0", hadoop = "2.6")
```

2.3 Connecting to Spark

It's important to mention that, so far, we've only installed a local Spark cluster. A local cluster is really helpful to get started, test code and troubleshoot with ease; further chapters will explain where to find, install and connect to real Spark clusters with many machines; but for the first few chapters, we will focus on using local clusters.

Threfore, to connect to this local cluster we simple run:

```
sc <- spark_connect(master = "local")
```

The `master` parameter helps `sparklyr` find which is the "main" machine from the Spark cluster, this machine is often call the driver node. While working with real clusters using many machines, most machines will be worker machines and one will be the master. Since we only have a local cluster with only one machine, we will default to use `"local"` for now.

2.4 Using Spark

Now that you are connected, we can run a simple commands. For instance, let's start by loading some text.

First, lets create a text file by running:

```
write("Hello World!", "hello.txt")
```

Which we can read back in Spark by running:

```
spark_read_text(sc, "hello", "hello.txt")
```

```
## # Source:   table<hello> [?? x 1]
## # Database: spark_connection
##   line
##   <chr>
## 1 Hello World!
```

2.4.1 Web Interface

Most of the Spark commands will get started from the R console; however, it is often the case that monitoring and analizing execution is done through Spark's web interface. This interface is a web page provided by the driver node which can be accessed from `sparklyr` by running:

```
spark_web(sc)
```

2.4.2 Logs

Another common tool is to read through the Spark logs, a log is just a text file where Spark will append information relevant to the execution of tasks in the cluster. For local clusters, we can retrieve the `sparklyr` related log entries by running:

```
spark_log(sc, filter = "sparklyr", n = 5)
```

```
## 18/05/28 14:17:55 INFO SparkContext: Submitted application: sparklyr
## 18/05/28 14:17:55 INFO SparkContext: Added JAR file:/Library/Frameworks/R.framework/Versions/3.5/Resour
## 18/05/28 14:17:58 INFO Executor: Fetching spark://localhost:64297/jars/sparklyr-2.3-2.11.jar with times
## 18/05/28 14:17:59 INFO Utils: Fetching spark://localhost:64297/jars/sparklyr-2.3-2.11.jar to /private/v
## 18/05/28 14:17:59 INFO Executor: Adding file:/private/var/folders/fz/v6wfsg2x1fb1rw4f6r0x4jwm0000gn/T/s
```

2.5 Disconnecting

For local clusters and, really, any cluster; once you are done processing data you should disconnect by running:

```
spark_disconnect(sc)
```

this will terminate the connection to the cluster but also terminate the cluster tasks as well. If multiple Spark connections are active, or if the conneciton instance `sc` is no longer available, you can also disconnect all your Spark connections by running `spark_disconnect_all()`.

2.6 Recap

This chapter walked you through installing R, Java, RStudio and `sparklyr` as the main tools required to use Spark from R. We covered installing local Spark clusters using `spark_install()` and learned how to launch the web interface using `spark_web(sc)` and view logs using `spark_log(sc)`.

It is my hope that this chapter will help anyone interested in learning cluster computing using Spark and R to get you started, ready to experiment on your own and ready to tackle actual data analysis and modeling tasks without any makor blockers. However, if you hit any installation or connection issues, start by browsing online for the error message or open a GitHub issue under https://github.com/rstudio/sparklyr/issues to help you get going.

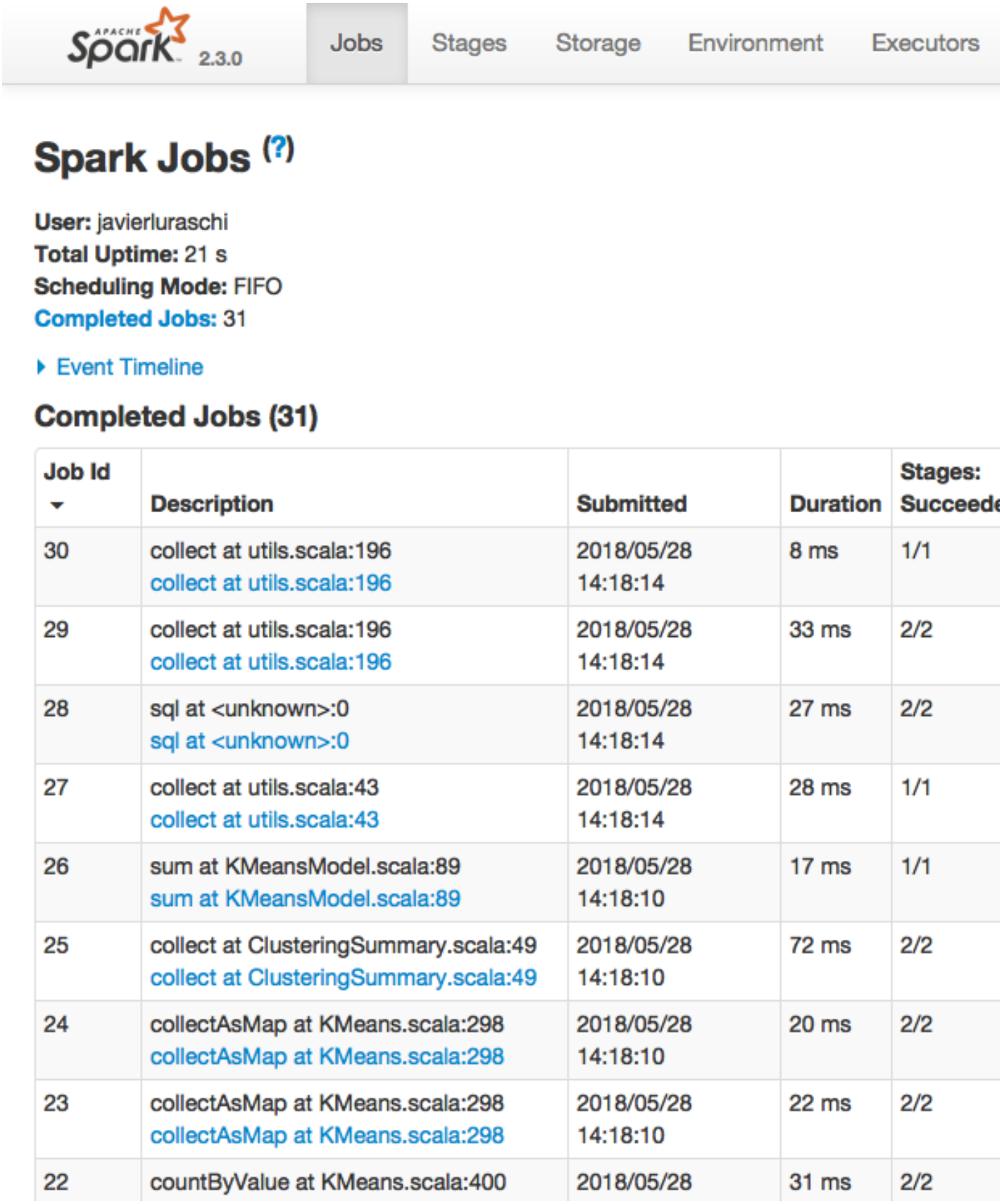


Figure 2.4: Apache Spark Web Interface.

Chapter 3

Analysis

While **this chatper has not been written**, a few resources and basic examples were made available to help out until this chapter is written.

3.1 dplyr

Using `sparklyr`, you can apply the same data analysis techniques described in Chapter 5 - Data transformation - R for Data Science by Garrett Grolemund and Hadley Wickham.

Once you understand `dplyr`, you can make use of `dplyr` and `sparklyr` as follows:

```
library(sparklyr)
library(dplyr)

# Connect to Spark
sc <- spark_connect(master = "local")

# Use dplyr's copy_to() to copy the iris dataset to Spark
iris_tbl <- copy_to(sc, iris, overwrite = TRUE)

# The iris_tbl is a Spark data frame compatible with dplyr
iris_tbl
```

```
## # Source:   table<iris> [?? x 5]
## # Database: spark_connection
##    Sepal_Length Sepal_Width Petal_Length Petal_Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <chr>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
##  8          5           3.4          1.5         0.2 setosa
##  9          4.4         2.9          1.4         0.2 setosa
## 10          4.9         3.1          1.5         0.1 setosa
## # ... with more rows
```

```
# Transform iris_tbl with dplyr as usual
iris_tbl %>%
  group_by(Species) %>%
  summarise_all(funs(mean))
```

```
## # Source:   lazy query [?? x 5]
## # Database: spark_connection
##   Species    Sepal_Length Sepal_Width Petal_Length Petal_Width
##   <chr>             <dbl>       <dbl>        <dbl>       <dbl>
## 1 versicolor         5.94        2.77         4.26       1.33
## 2 virginica          6.59        2.97         5.55       2.03
## 3 setosa             5.01        3.43         1.46       0.246
```

To understand `dplyr` further, I would recommend taking a look at the following vignettes:

- Introduction to dplyr
- Two-table verbs
- Window functions
- Programming with dplyr

3.2 DBI

The `DBI` provides an database interface for R, meaning, if you are familiar with SQL, you can make use of `DBI` to perform SQL queries in Spark using `sparklyr`. To learn more about `DBI`, I would recommend reading first A Common Database Interface (DBI). Once you are familiar with `DBI`, you can use this package with `sparklyr` as follows:

```
library(DBI)

dbGetQuery(sc,
  "SELECT mean(Sepal_Length), mean(Sepal_Width),
          mean(Petal_Length), mean(Petal_Width)
   FROM iris
   GROUP BY Species")
```

```
##   avg(Sepal_Length) avg(Sepal_Width) avg(Petal_Length) avg(Petal_Width)
## 1             5.936            2.770             4.260            1.326
## 2             6.588            2.974             5.552            2.026
## 3             5.006            3.428             1.462            0.246
```

More advanced `DBI` resources are available in the following vignettes:

- A Common Interface to Relational Databases from R and S – A Proposal
- Implementing a new backend
- DBI specification

Chapter 4

Modeling

While **this chatper has not been written**, I want to provide a few resources and basic examples to help out while the chapter is written.

4.1 Overview

MLlib is Apache Spark's scalable machine learning library and is available through `sparklyr`, mostly, with functions prefixed with `ml_`. The following table describes some of the modeling algorithms supported:

Algorithm	Function
Accelerated Failure Time Survival Regression	ml_aft_survival_regression
Alternating Least Squares Factorization	ml_als
Correlation Matrix	ml_corr
Decision Trees	ml_decision_tree
Generalized Linear Regression	ml_generalized_linear_regression
Gradient-Boosted Trees	ml_gradient_boosted_trees
Isotonic Regression	ml_isotonic_regression
K-Means Clustering	ml_kmeans
Latent Dirichlet Allocation	ml_lda
Linear Regression	ml_linear_regression
Linear Support Vector Machines	ml_linear_svc
Logistic Regression	ml_logistic_regression
Multilayer Perceptron	ml_multilayer_perceptron
Naive-Bayes	ml_naive_bayes
One vs Rest	ml_one_vs_rest
Principal Components Analysis	ml_pca
Random Forests	ml_random_forest
Survival Regression	ml_survival_regression

Here is an example to get you started with K-Means:

```
library(sparklyr)

# Connect to Spark in local mode
sc <- spark_connect(master = "local")
```

```
## Re-using existing Spark connection to local
```

```
# Copy iris to Spark
iris_tbl <- sdf_copy_to(sc, iris, overwrite = TRUE)

# Run K-Means for Species using only Petal_Width and Petal_Length as features
iris_tbl %>%
  ml_kmeans(centers = 3, Species ~ Petal_Width + Petal_Length)
```

```
## K-means clustering with 3 clusters
##
## Cluster centers:
##   Petal_Width Petal_Length
## 1    1.359259     4.292593
## 2    0.246000     1.462000
## 3    2.047826     5.626087
##
## Within Set Sum of Squared Errors =  31.41289
```

More examples are reosurces are available in spark.rstudio.com/mlib.

Chapter 5

Clusters

Previous chapters focused on using Spark over a single computing instance, your personal computer. In this chapter we will introduce techniques to run Spark over multiple computing instances, also known as a computing cluster, to analyze data at scale.

If you already have a Spark cluster in your organization, you could consider skipping to the next chapter, Connections, which will teach you how to connect to an existing cluster.

If don't have a cluster or are considering improvements to your existing infrastructure, this chapter will introduce some of the cluster trends, managers and providers available today.

5.1 Overview

There are three major trends in cluster computing worth discussing: **on-premise**, **cloud computing** and **kubernetes**. Framing these trends over time will help us understand how they came to be, what they are and what their future might be:

For **on-premise** clusters, someone, either yourself or someone in your organiation purchased physical computers that are intended to be used for cluster computing. The computers in this cluster can made of *off-the-shelf* hardware, meaning that someone placed an order to purchase computers usually found in stores shelves or, *high-performance* hardware, meaning that a computing vendor provided highly customized computing hardware which also comes optimized for high-performance network connectivity, power consumption, etc. When purchasing hundreds or thousands of computing instances, it doesn't make sense to keep them in the usual computing case that we are all familiar with, but rather, it makes sense to stack them as efficient as possible on top of each other to minimize room space. This group of efficiently stacked computing instances is known as a rack. Once a cluster grows to thousands of computers, you will also need to host hundreds of racks of computing devices, at this scale, you would also need significant physical space to hosts those racks. A building that provides racks of computing instances is usually known as a *data-center*. At the scale of a data center, optimizing the building that holds them, their heating system, power suply, network connectivity, etc. becomes also relevant to optimize. In 2011, Facebook announced the Open Compute Project inniciative which provides a set of data center blueprints free for anyone to use.

There is nothing preventing us from building our own data centers and in fact, many organizations have followed this path. For instance, Amazon started as an online book store, over the years Amazon grew to sell much more than just books and, with it's online store growth, their data centers also grew in size. In 2002, Amazon considered selling access to virtual servers, in their data centers to the public and, in 2004, Amazon Web Services launched as a way to let anyone rent a subset of their datacenters on-demand, meaning that one did not have to purchase, configure, maintain nor teardown it's own clusters but could rather rent them from Amazon directly.

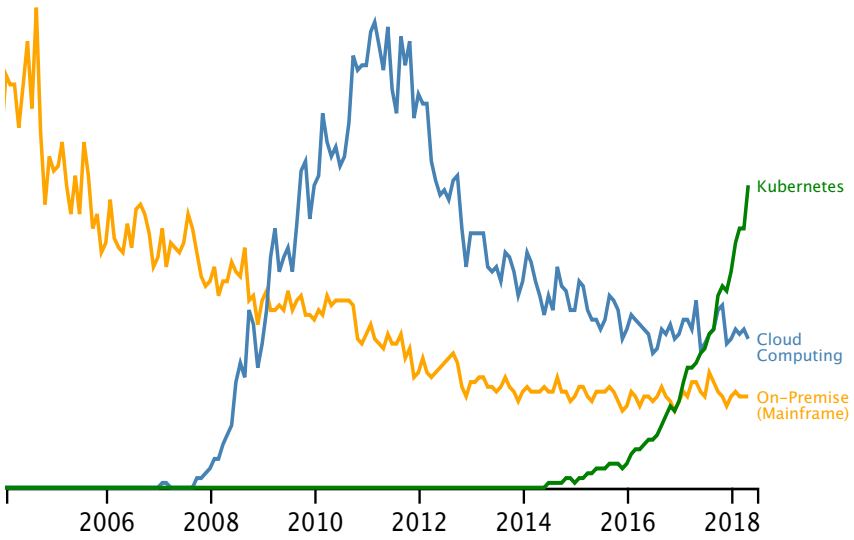


Figure 5.1: Google trends for on-premise (mainframe), cloud computing and kubernetes.

The on-demand compute model is what we know today as **Cloud Computing**. It's a concept that evolved from Amazon Web Services providing their data centers as a service. In the cloud, the cluster you use is not owned by you and is neither in your physical building, but rather, it's a data center owned and managed by someone else. Today, there are many cloud providers in this space ranging from Amazon, Microsoft, Google, IBM and many others. Most cloud computing platforms provide a user interface either through a web applciation and command line to request and manage resources.

While the bennefits of processing data in the *cloud* were obvious for many years, picking a cloud provider had the unintended side-effect of locking organizations with one particular provider, making it hard to switch between provideers or back to on-premise clusters. **Kubernetes**, announced by Google in 2014, is an open source system for managing containerized applications across multiple hosts. In practice, it provides common infrastructure otherwise proprietary to cloud providers making it much easier to deploy across multiple cloud providers and on-premise as well. However, being a much newer paradigm than on-premise or cloud computing, it is still in it's adoption phase but, nevertheless, promising for cluster computing in general and, specifically, for Apache Spark.

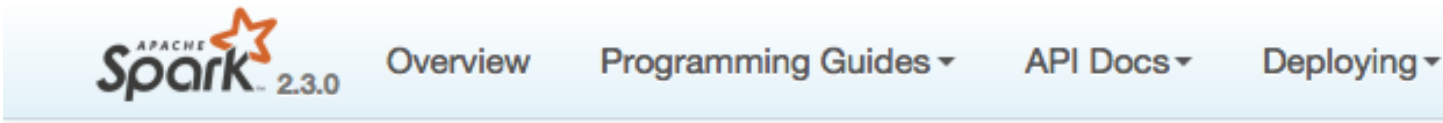
5.2 Managers

In order to run Spark within a computing cluster, one needs to run something capable of initializing Spark over each compute instance, this is known as a cluster manager. The available cluster managers in Spark are: **Spark Standalone**, **YARN**, **Mesos** and **Kubernetes**.

5.2.1 Standalone

In **Spark Standalone**, Spark works on it's own without additional software requirements since it provides it's own cluster manager as part of the Spark installation.

By completing the Installation chapter, you should have a local Spark installation available, which we can use to initialize a local stanalone Spark cluster. First, retrieve the `SPARK_HOME` directory by running `sparklyr::spark_home_dir()` from R and then, from a terminal or R, use `start-master.sh` and `start-slave.sh` as follows:



Spark Standalone Mode

- Installing Spark Standalone to a Cluster
- Starting a Cluster Manually
- Cluster Launch Scripts
- Connecting an Application to the Cluster
- Launching Spark Applications
- Resource Scheduling
- Executors Scheduling
- Monitoring and Logging
- Running Alongside Hadoop
- Configuring Ports for Network Security
- High Availability
  - Standby Masters with ZooKeeper
  - Single-Node Recovery with Local File System

In addition to running on the Mesos or YARN cluster managers, Spark also provides a simple sta
standalone cluster either manually, by starting a master and workers by hand, or use our provide
daemons on a single machine for testing.

Installing Spark Standalone to a Cluster

To install Spark Standalone mode, you simply place a compiled version of Spark on each node o
Spark with each release or build it yourself.

Starting a Cluster Manually

You can start a standalone master server by executing:

```
./sbin/start-master.sh
```

Once started, the master will print out a `spark://HOST:PORT` URL for itself, which you can use to

Figure 5.2: Spark Standalone Site.

```
# Retrieve the Spark installation directory
spark_home <- sparklyr::spark_home_dir()

# Build path to start-master.sh
start_master <- file.path(spark_home, "sbin", "start-master.sh")

# Execute start-master.sh to start the cluster manager master node
system2(start_master)

# Build path to start-slave
start_slave <- file.path(spark_home, "sbin", "start-slave.sh")

# Execute start-slave.sh to start a worker and register in master node
system2(start_slave, paste0("spark://", system2("hostname", stdout = TRUE), ":7077"))
```

The previous command initialized the master node and a worker node, the master node interface can be accessed under localhost:8080 and looks like the following:

Notice that there is one worker register in Spark standalone, you can follow the link to this worker node to see additional information:

Once data analysis is complete, one can simply stop all the running nodes in this local cluster by running:

```
stop_all <- file.path(spark_home, "sbin", "stop-all.sh")
system2(stop_all)
```

A similar approach can be followed to configure a cluster by running each `start-slave.sh` command over each machine in the cluster.

Further reading: Spark Standalone Mode

5.2.2 Yarn

YARN for short, or Hadoop YARN, is the resource manager introduced in 2012 to the Hadoop project. As mentioned in in the Introduction chapter, Spark was built initially to speed up computation over Hadoop; then, when Hadoop 2 was launched, it introduced YARN as a component to manage resources in the cluster, to this date, using Hadoop YARN with Apache Spark is still very common.

YARN applications can be submitted in two modes: **yarn-client** and **yarn-cluster**. In yarn-cluster mode the driver is running remotely, while in yarn-client mode, the driver is on the machine that started the job, `sparklyr` supports both modes.

Further reading: Running Spark on YARN

5.2.3 Mesos

Apache Mesos is an open-source project to manage computer clusters. Mesos began as a research project in the UC Berkeley RAD Lab by then PhD students Benjamin Hindman, Andy Konwinski, and Matei Zaharia, as well as professor Ion Stoica. Mesos uses Linux Cgroups to provide isolation for CPU, memory, I/O and file system.

Further reading: Running Spark on Mesos

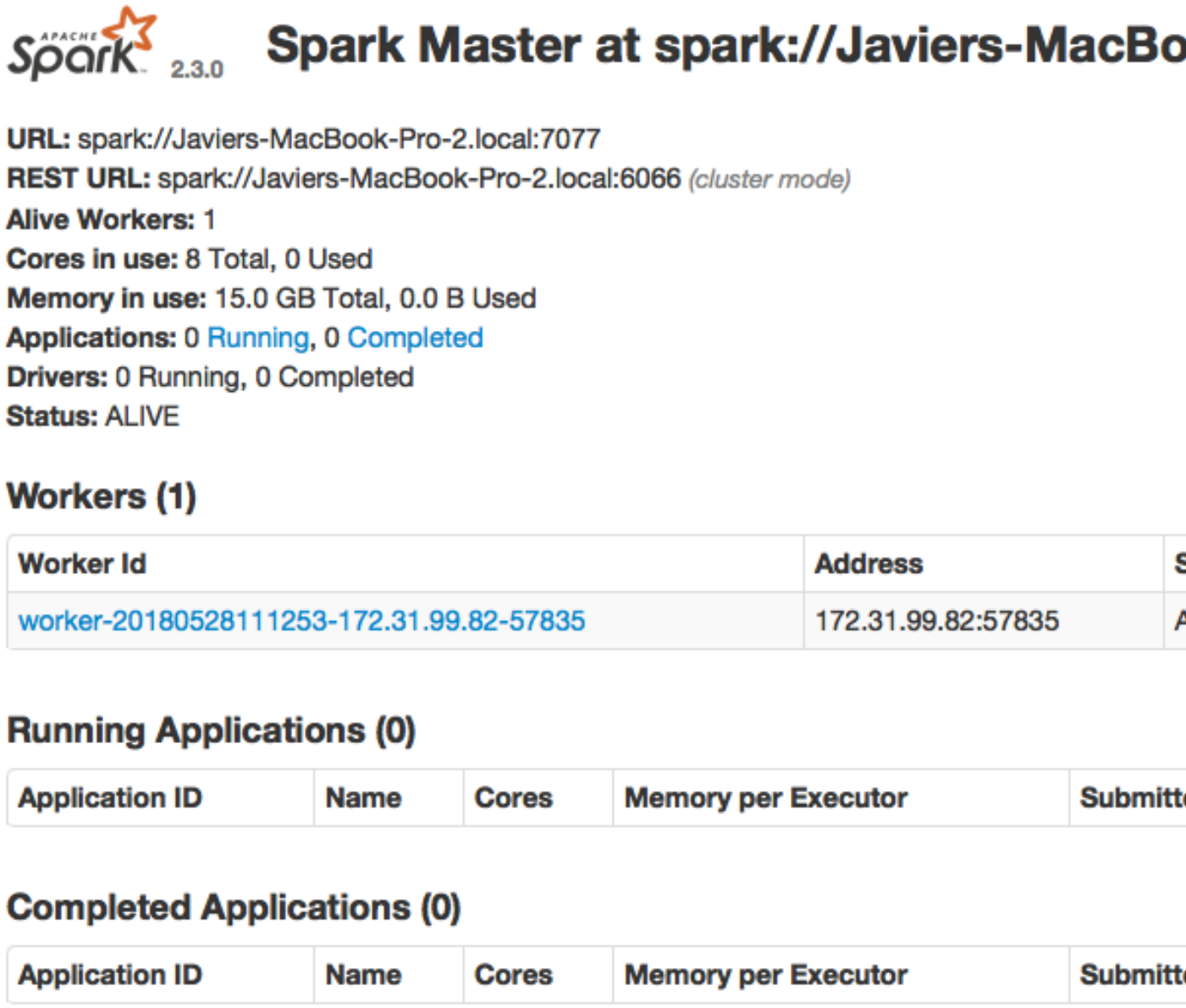


Figure 5.3: Spark Standalone Web Interface.

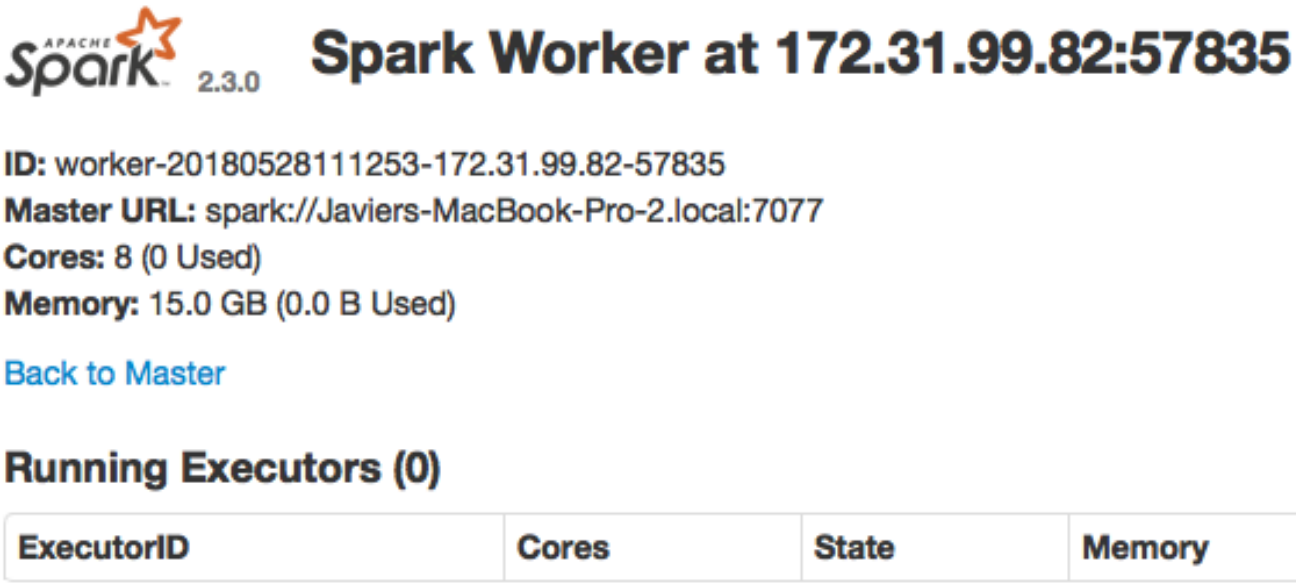


Figure 5.4: Spark Standalone Worker Web Interface.

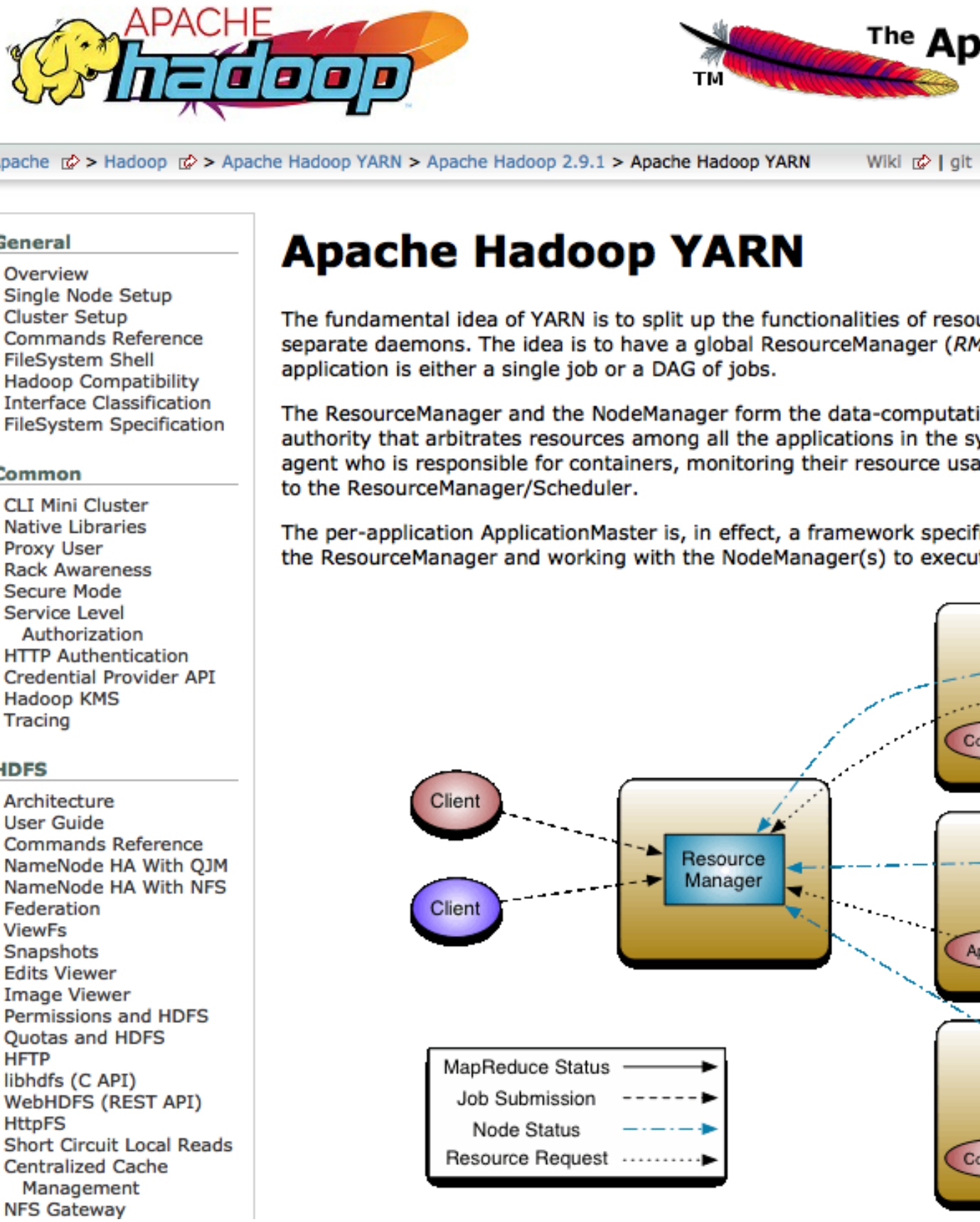


Figure 5.5: Hadoop YARN Site

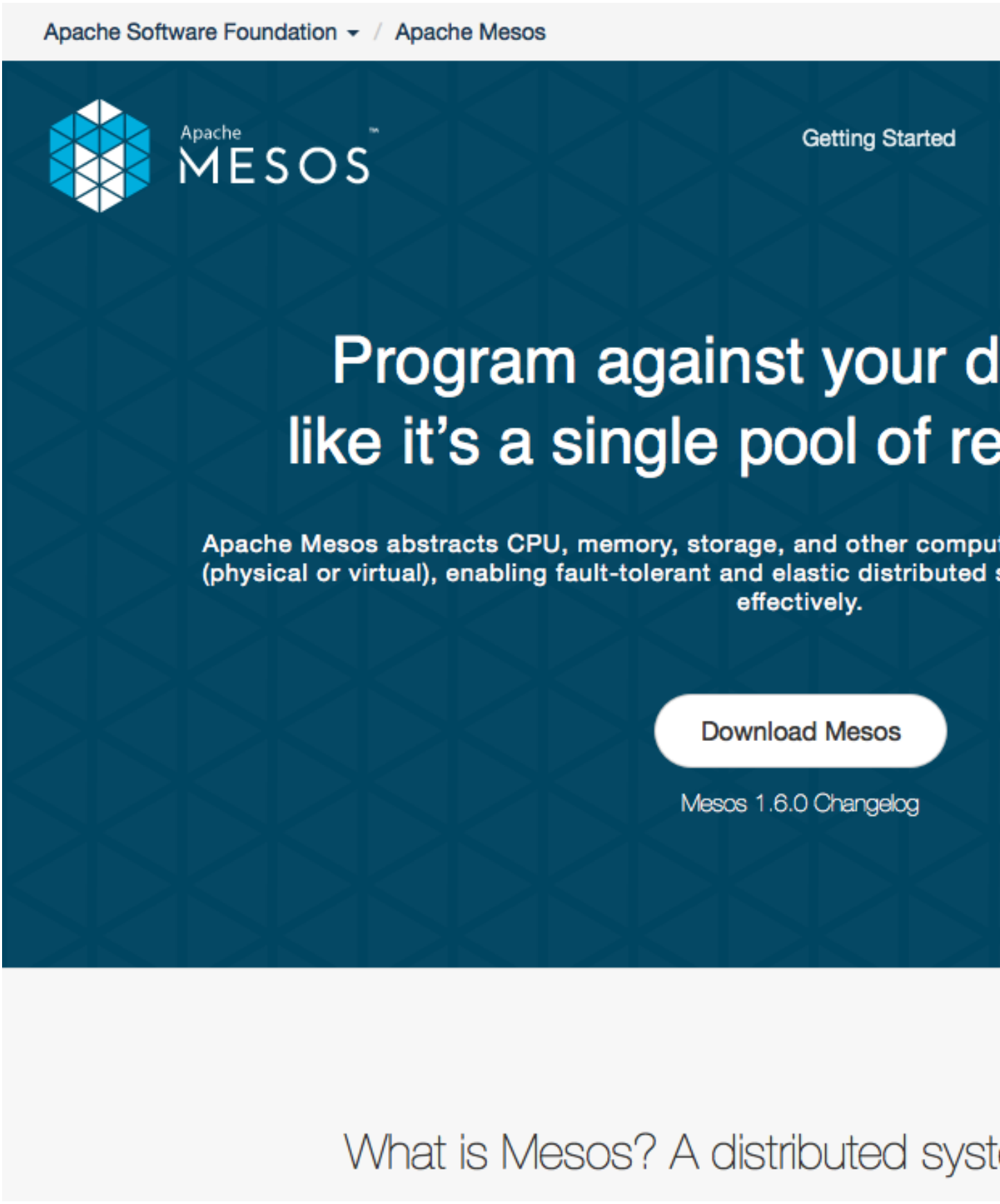


Figure 5.6: Mesos Landing Site

5.2.4 Kubernetes

Kubernetes is an open-source container-orchestration system for automating deployment, scaling and management of containerized applications that was originally designed by Google and now maintained by the Cloud Native Computing Foundation.

Further reading: Running Spark on Kubernetes

5.3 On-Premise

As mentioned in the overview section, on-premise clusters represent a set of computing instances procured, colocated and managed by staff members from your organization. These clusters can be highly customized and controlled; however, they can also inccur significant initial expenses and maintenance costs.

One can use a cluster manager in on-premise clusters as described in the previous section; however, many organizations choose to partner with companies providing additional management software, services and resources to manage software in their cluster including, but not limited to, Apache Spark. Some of the on-premise cluster providers include: Cloudera, Hortonworks and MapR to mention a few which will be briefly introduced.

5.3.1 Cloudera

Cloudera, Inc. is a United States-based software company that provides Apache Hadoop and Apache Spark-based software, support and services, and training to business customers.

Cloudera's hybrid open-source Apache Hadoop distribution, CDH (Cloudera Distribution Including Apache Hadoop), targets enterprise-class deployments of that technology. Cloudera says that more than 50% of its engineering output is donated upstream to the various Apache-licensed open source projects (Apache Hive, Apache Avro, Apache HBase, and so on) that combine to form the Apache Hadoop platform. Cloudera is also a sponsor of the Apache Software Foundation.

5.3.2 Hortonworks

Hortonworks is a big data software company based in Santa Clara, California. The company develops, supports, and provides expertise on an expansive set of entirely open source software designed to manage data and processing for everything from IOT, to advanced analytics and machine learning. Hortonworks believes it is a data management company bridging the cloud and the datacenter.

5.3.3 MapR

MapR is a business software company headquartered in Santa Clara, California. MapR provides access to a variety of data sources from a single computer cluster, including big data workloads such as Apache Hadoop and Apache Spark, a distributed file system, a multi-model database management system, and event stream processing, combining analytics in real-time with operational applications. Its technology runs on both commodity hardware and public cloud computing services.

5.4 Cloud

For those readers that don't have a cluster yet, it is likely that you will want to choose a cloud cluster, this section will briefly mention some of the major cloud infrastructure providers as a starting point to choose

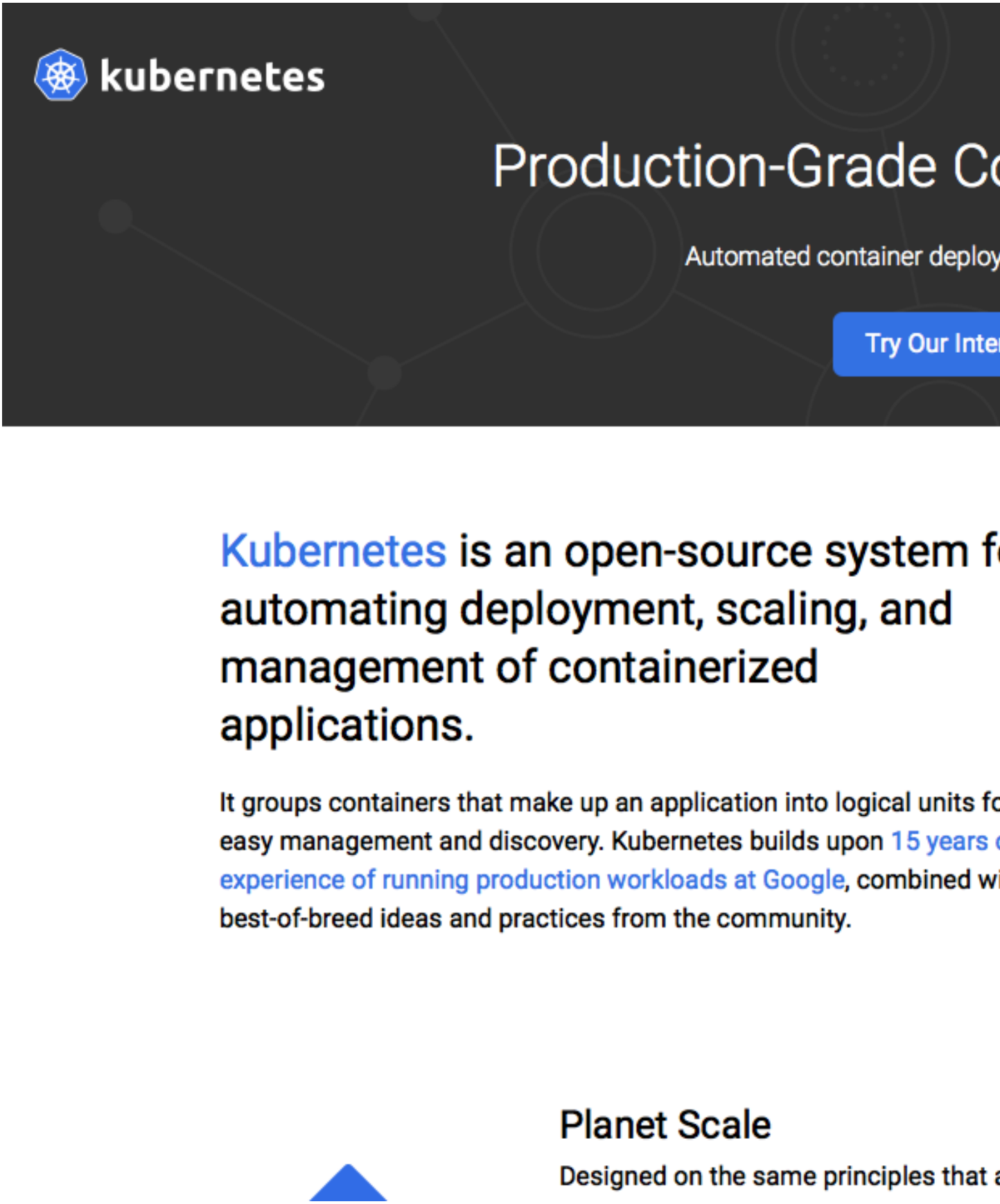


Figure 5.7: Kubernetes Landing Site.

Figure 5.8: Cloudera Landing Site.

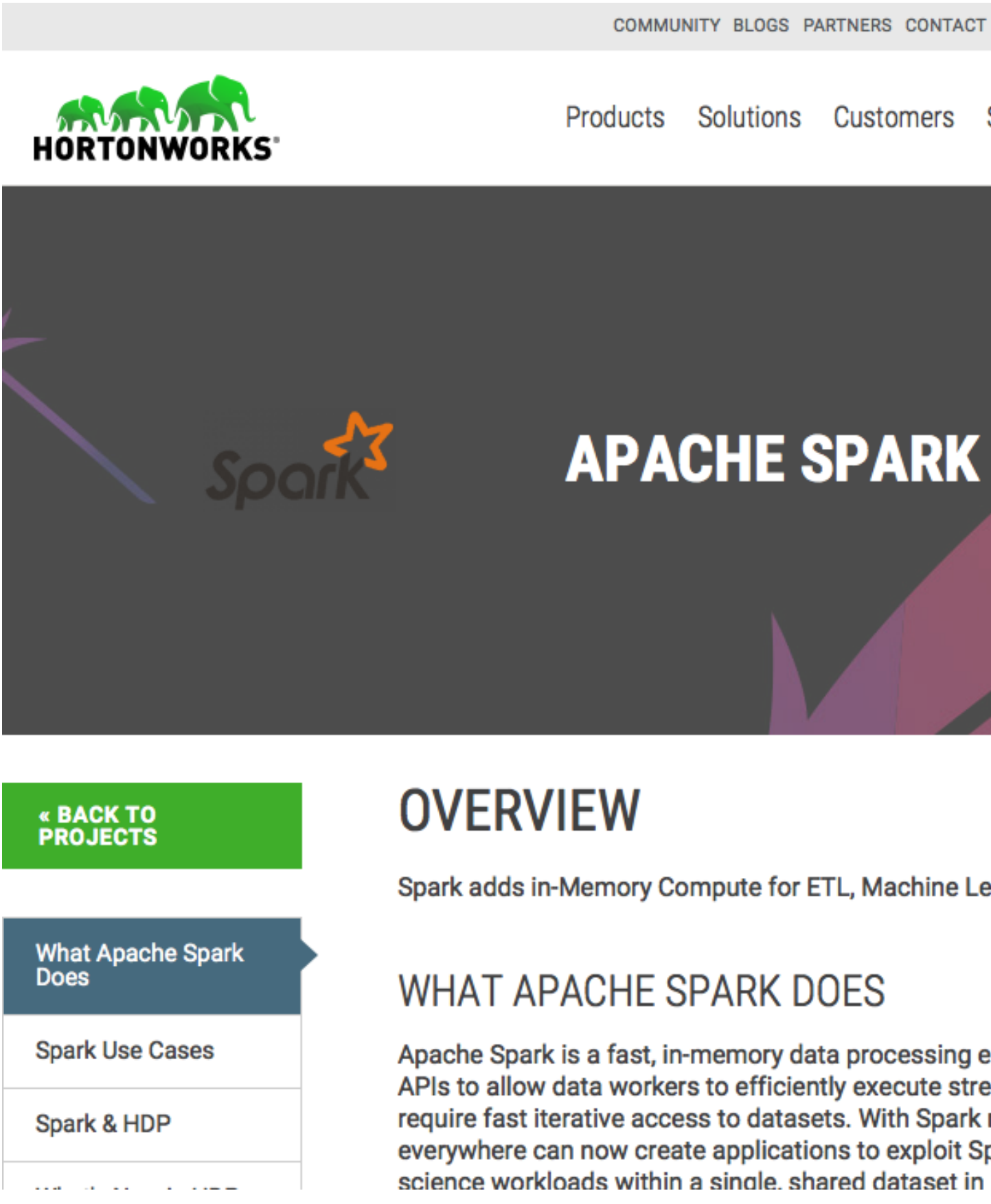


Figure 5.9: Hortonworks Landing Site.

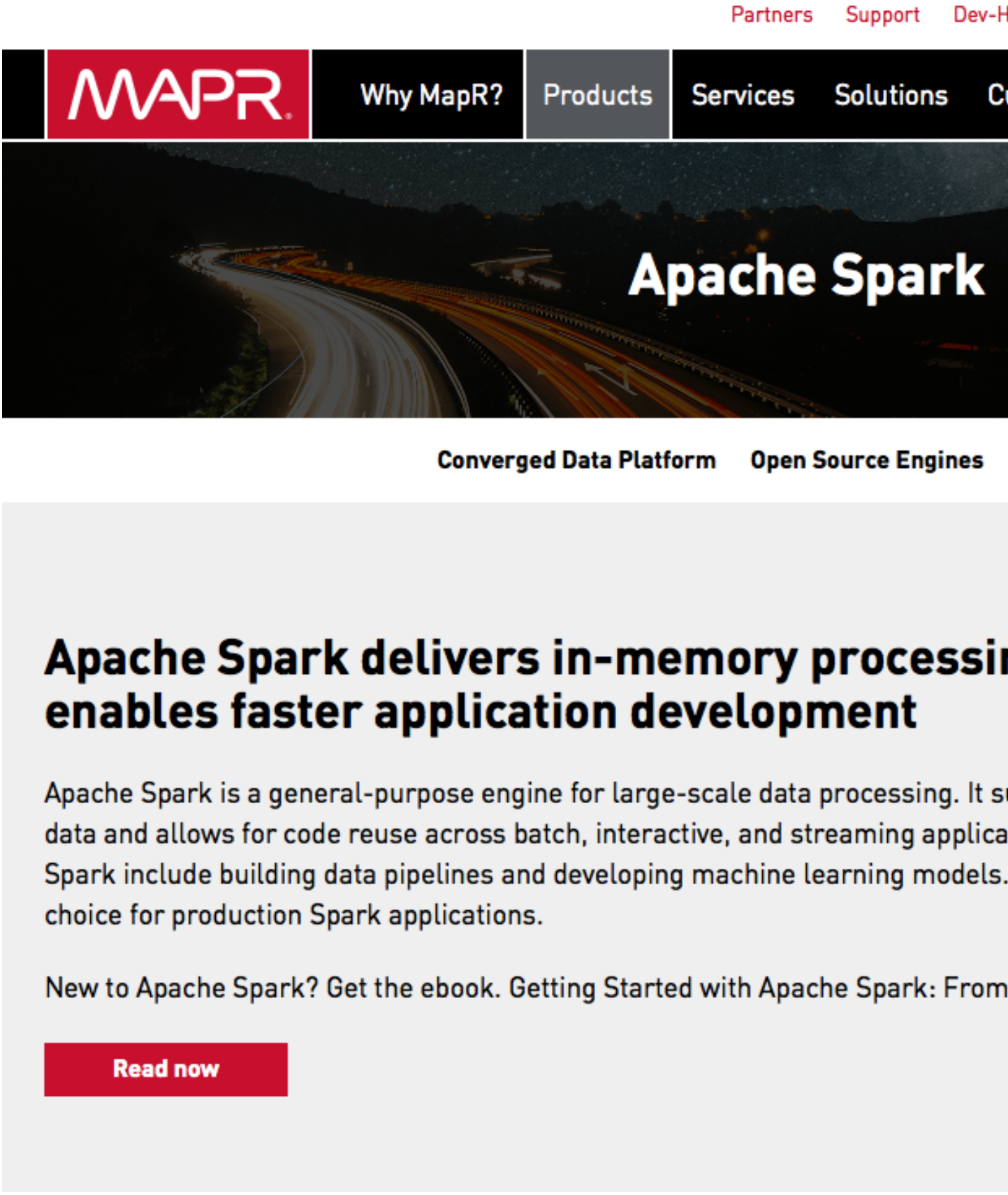


Figure 5.10: MapR Landing Site.

the right one for you.

It is worth mentioning that in a cloud service model, the compute instances are charged by the hour and times the number of instances reserved for your cluster. Since the cluster size is flexible, it is a good practice to start with small clusters and scale compute resources as needed. Even if you know in advance that a cluster of significant size will be required, starting small provides an opportunity to troubleshoot issues at a lower cost since it's unlikely that your data analysis will run at scale flawlessly on the first try.

The major providers of cloud computing infrastructure are: Amazon, Google and Microsoft that this section will briefly introduce.

5.4.1 Amazon

Amazon provides cloud services through Amazon Web Services; more specifically, they provide an on-demand Spark cluster through Amazon Elastic Map Reduce or EMR for short.

5.4.2 Google

Google provides their on-demand computing services through their Google Cloud, on-demand Spark cluster are provided by Google Dataproc.

5.4.3 Microsoft

Microsoft provides cloud services thorugh Microsft Azure and Spark clusters through Azure HDInsight.

5.5 Tools

While using only R and Spark can be sufficient for some clusters, it is common to install complementary tools in your cluster to improve: monitoring, sql analysis, workflow coordination, etc. with applications like Ganglia, Hue and Oozie respectevly. This secton is not meant to cover all, but rather mention two that are relevant to R and `sparklyr`.

5.5.1 RStudio

RStudio's open source and professional products, like: RStudio Server, RStudio Server Pro, Shiny Server, Shiny Server Pro, or RStudio Connect; can be installed within the cluster to support many R workflows, while `sparklyr` does not require any additional tools, they provide significant productivity gains worth considering.

5.5.2 Livy

Apapche Livy is an incubation project in Apache providing support to use Spark clusters remotely through a web interface. It is ideal to connect directly into the Spark cluster; however, there are times where connecting directly to the cluster is not feasible. When facing those constraints, one can consider installing Livy in their cluster and secure it properly to enable remote use over web protocols.

However, there is a significant performance overhead from using Livy in `sparklyr` for experimentation, meaning that, executing many client comamnds over Livy has a significant overhead; however, running a few commands to generate complex analysis is usually performant since the performance overhead of starting computation can be insignificant compared to the actual cluster computation.

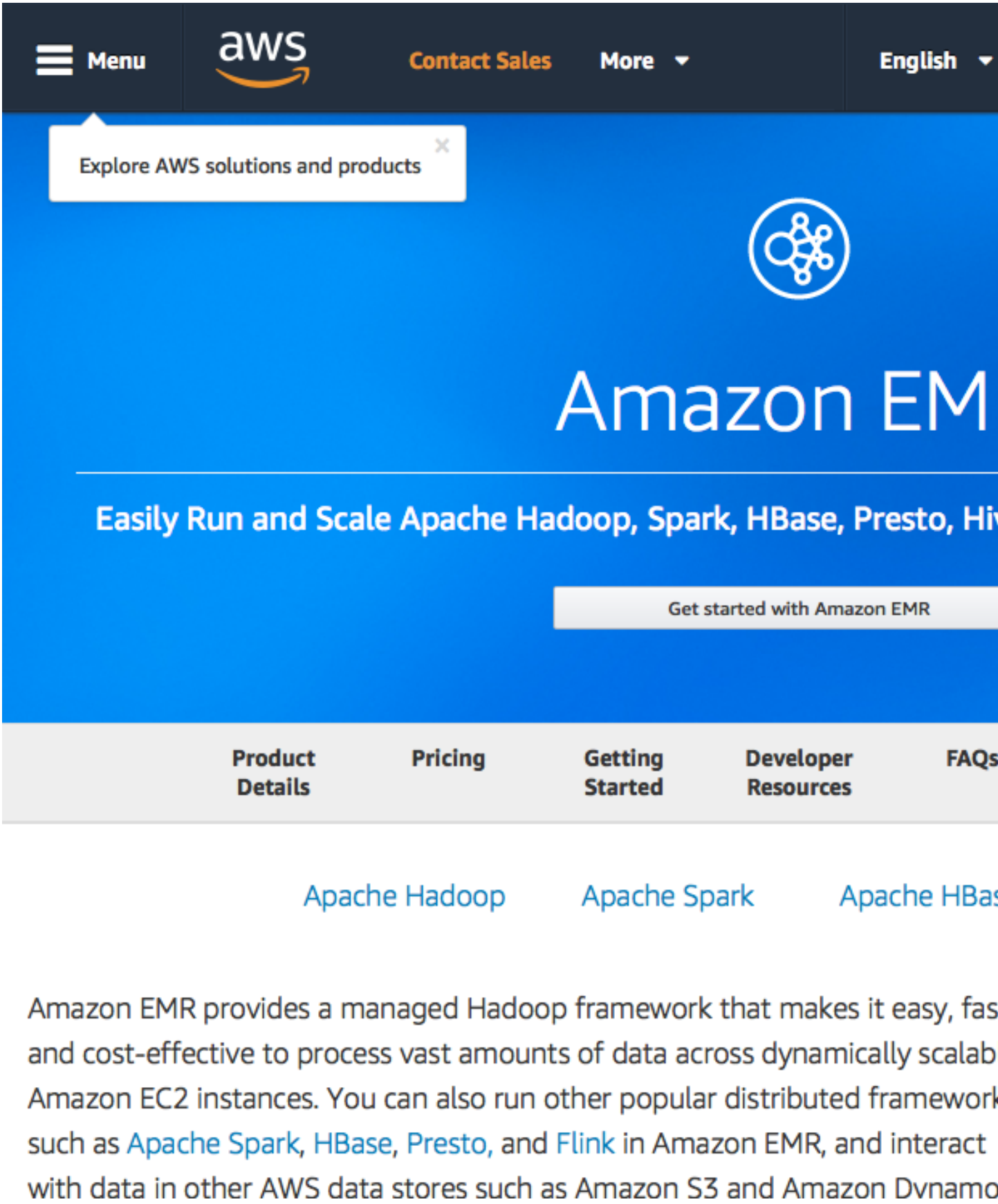


Figure 5.11: Amazon EMR Landing Site.

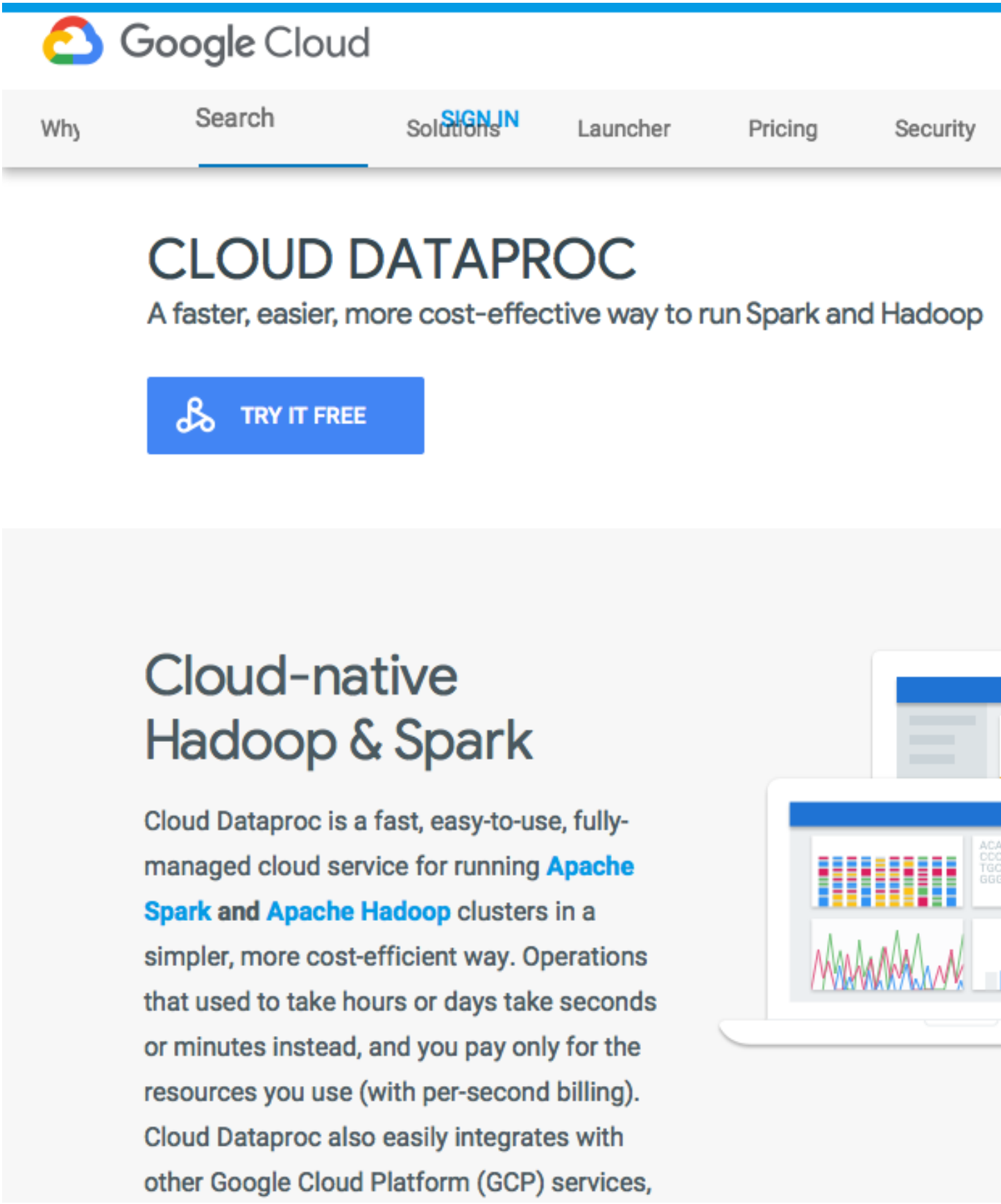


Figure 5.12: Google Dataprox Landing Site.

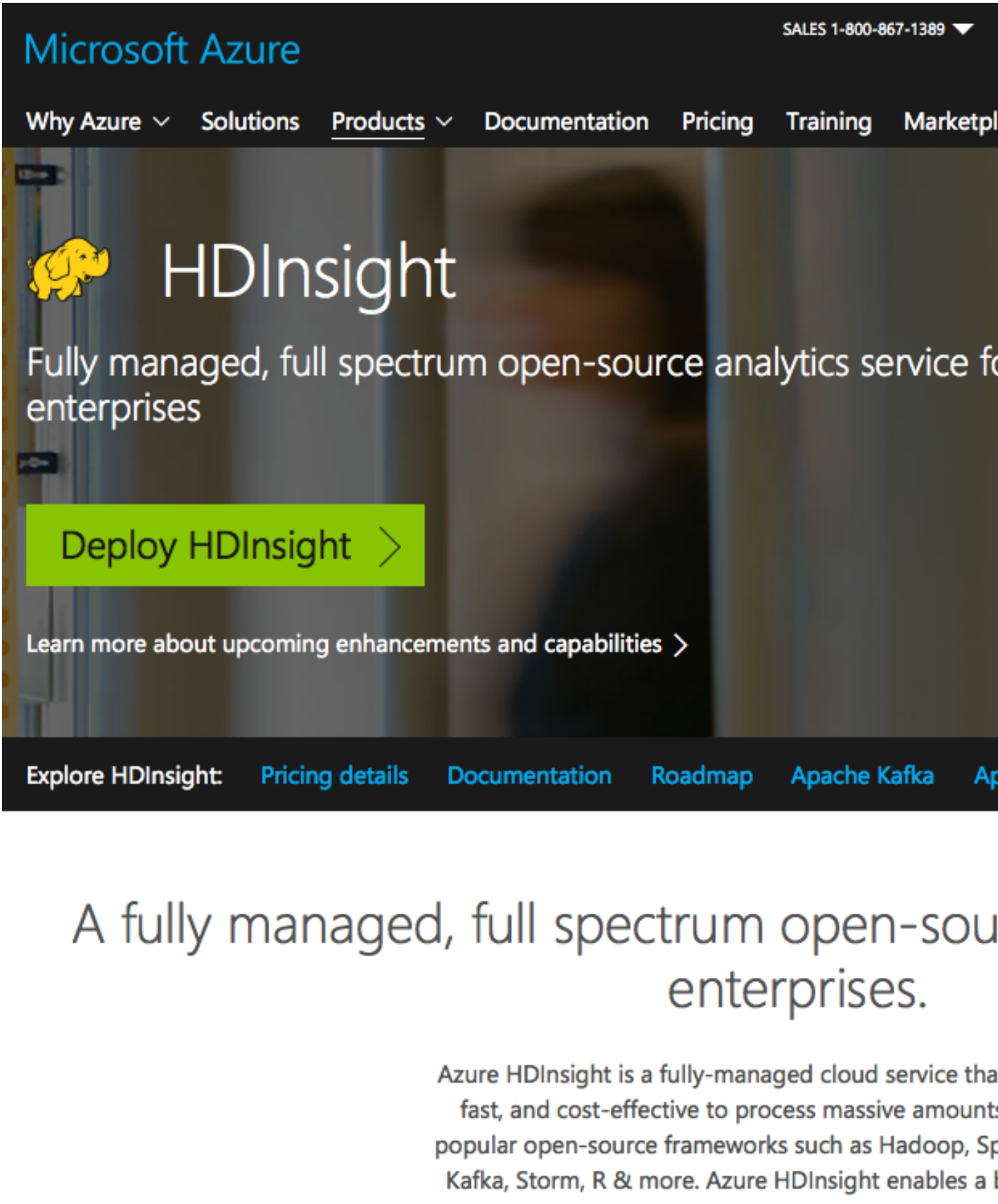


Figure 5.13: Azure HDInsight Landing Site.

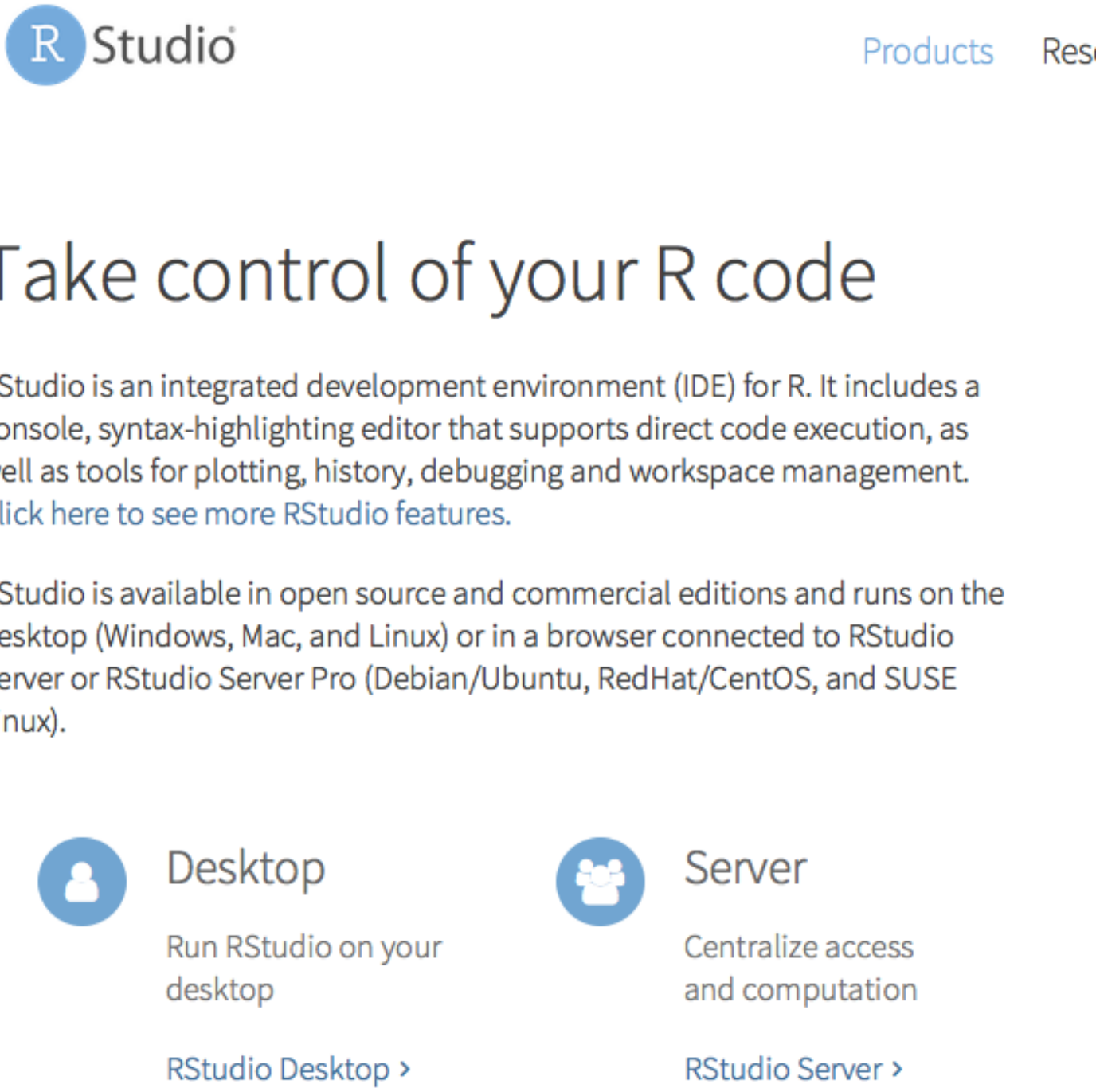


Figure 5.14: Rstudio Server

Submit Jobs from Anywhere

Livy enables programmatic, fault-tolerant, multi-tenant submission of Spark jobs f
needed). So, multiple users can interact with your Spark cluster concurrently and

Use Interactive Scala or Python

Livy speaks either Scala or Python, so clients can communicate with your Spark
batch job submissions can be done in Scala, Java, or Python.

No Code Changes Needed

Don't worry, no changes to existing programs are needed to use Livy. Just build
to your Spark cluster, and you're off! Check out Get Started to get going.

What is Apache Livy?

Figure 5.15: Apache Livy Landing Site.

5.6 Recap

This chapter explained the history and tradeoffs of on-premise, cloud computing and presented Kubernetes as a promising framework to provide flexibility across on-premise and cloud providers. It also introduced cluster managers (Spark Standalone, YARN, Mesos and Kubernetes) as the software needed to run Spark as a cluster application. This chapter briefly mentioned on-premise cluster providers like Cloudera, Hortonworks and MapR as well as the major cloud providers: Amazon, Google and Microsoft.

While this chapter provided a solid foundation to understand current computing trends, cluster tools and providers useful to perform data science; it falls short to help those tasked with deliberately choosing a cluster manager, service provider or architecture. If you have this task assigned to you, use this chapter as a starting point to reach to many more resources to complete your understanding of the platform your organization needs.

The next chapter, Connections, assumes a Spark cluster is already available to you and will focus on understanding how to connect to it from sparklyr.

Chapter 6

Connections

The previous chapter, Clusters, presented the major cluster computing paradigms, cluster managers and cluster providers; this section explains the internal components of a Spark cluster and the how to perform connections to any cluster running Apache Spark.

6.1 Overview

Before explaining how to connect to Spark clusters, it is worth discussing the components of a Spark cluster and how they interact, this is often known as the cluster architecture of Apache Spark.

First, lets go over a couple definitions. As you know form previous chapters, a cluster is a collection of machines to perform analysis beyond a single computer. However, in distributed systems and clusters literature, we often reffer to each physical machine as a compute instance, compute node, or simply instance or node for short. It is helpful to remind this while reading through this chapter and making use of external resource.

In a Spark cluster, there are three types of compute instances that are relevant to Spark: The **driver node**, the **worker nodes** and the **cluster manager**. A cluster manager is a service that allos Spark to be executed in the clsuter and was explained in the Cluster Managers section. The driver node is tasked with delegating work to the worker nodes, but also for aggregating their results and iterating further if needed. For the most part, aggregation happens in the worker nodes; however, even after the nodes aggregate data, it is often the case that the driver node would have to aggregate the worker results. Therefore, the driver node has at least, but often, much more compute resources (read RAM, CPU, Local Storage, etc.) then the worker node.

Strictly speaking, the driver node and worker nodes are just names assigned to machines with particular roles, while the actual computation in the driver node is performed by the **spark context**. The Spark context is a Spark component tasked with scheduling tasks, managing data and so on. In the worker nodes, the actual computation is performed under a **spark executor**, which is also a Spark component tasked with executing subtasks against a data partition.

If you already have an Spark cluster in their organization, you should ask your cluster administrator to provide connection information for this cluster and read carefully their usage policies and constraints. A cluster is usually shared among many users so you want to be respectful of others time and resources while using a shared cluster environment. Your system administrator will describe if it's an **on-premise** vs **cloud** cluster, the **cluster manager** being used, supported **connections** and supported **tools**. You can use this information to jump directly to Local, Standalone, Yarn, Mesos, Livy or Kubernetes based on the information provided to you.

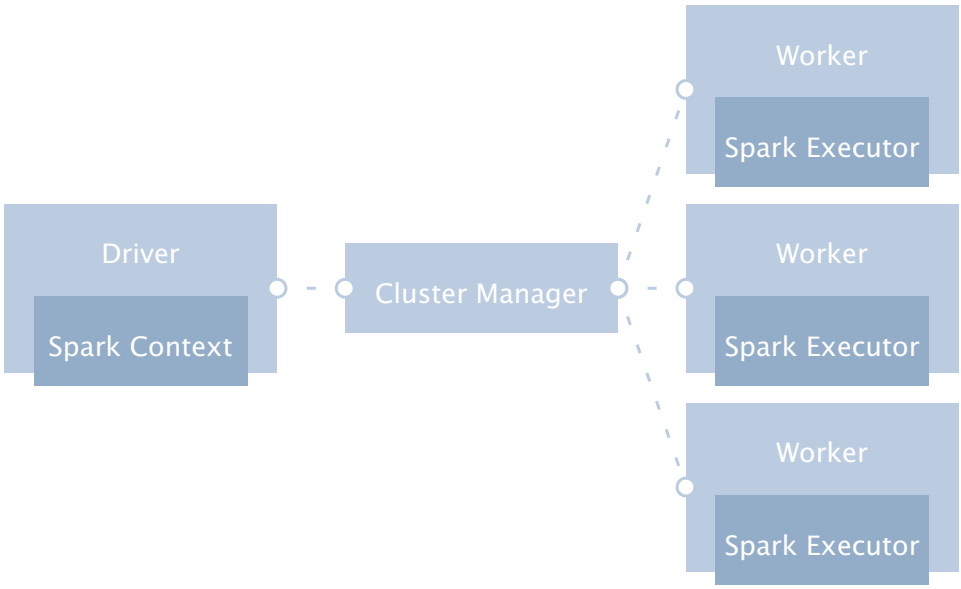


Figure 6.1: Apache Spark Architecture

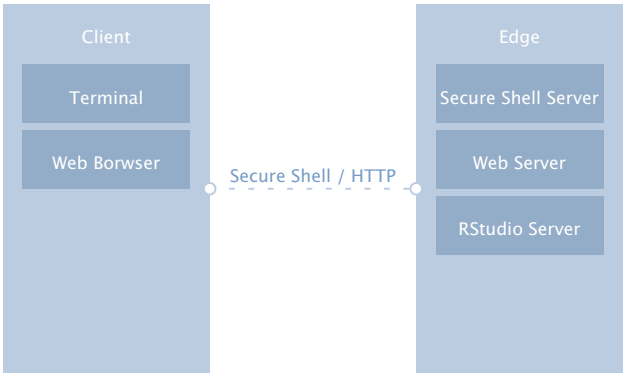


Figure 6.2: Using a Spark Cluster from an Edge Node

6.1.1 Edge Nodes

Before connecting to Apache Spark, you will first have to connect to the cluster. Usually, by connecting to an edge node within the cluster. An edge node, is a machine that can accessed from outside the cluster but which is also part of the cluster. There are two methods to connect to this edge instance:

- **Terminal**: Using a computer terminal applicaiton, one can use a secure shell to establish a remote connection into the cluster, once you connect into the cluster, you can launch R and then use `sparklyr`.
- **Web Browser**: While using `sparklyr` from a terminal is possible, it is usually more producty to install a **web server** in an edge node that provides more tools and functionality to run R with `sparklyr`. Most likely, you will want to consider using RStudio Server rather than connecting from the terminal.

6.1.2 Spark Home

It is important to mention that, while connecting to a Spark cluster, you will need to find out the correct `SPARK_HOME` path which contains the installation of Spark in the given instance. The `SPARK_HOME` path must

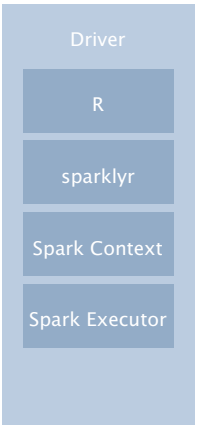


Figure 6.3: Local Connection Diagram

be set as an environment variable before connecting or explicitly specified in `spark_connect()` using the `spark_home` parameter.

For system administrators, we recommend you set `SPARK_HOME` for all the users in your cluster; however, if this is not set in your cluster you can also specify `SPARK_HOME` while using `spark_connect()` as follows:

```
sc <- spark_connect(master = "cluster-master", spark_home = "local/path/to/spark")
```

Where `cluster-master` is set to the correct cluster manager master for Spark Standalone, YARN, Mesos, etc.

6.2 Local

When connecting to Spark in local mode, Spark starts as a single application simulating a cluster with a single node, this is not a proper computing cluster but is ideal to perform work offline and troubleshoot issues. A local connection to Spark is represented in the following diagram:

Notice that in the local connections diagram, there is no cluster manager nor worker process since, in local mode, everything runs inside the driver application. It's also worth pointing out that `sparklyr` starts the Spark Context through `spark-submit`, a script available in every Spark installation to enable users to submit custom application to Spark which `sparklyr` makes use of to submit itself to Spark. For the curious reader, the Contributing chapter explains the internal processes that take place in `sparklyr` to submit this application and connect properly from R.

To perform this local connection, we can connect with the following familiar code used in previous chapters:

```
# Connect to local Spark instance
sc <- spark_connect(master = "local")
```

By default, `sparklyr`, will connect using as many CPU cores are available in your compute instance; however, this can be customized by connecting using `master="local[n]"`, where `n` is the desired number of cores to use. For example, we can connect using only 2 CPU cores as follows:

```
# Connect to local Spark instance using 2 cores
sc <- spark_connect(master = "local[2]")
```

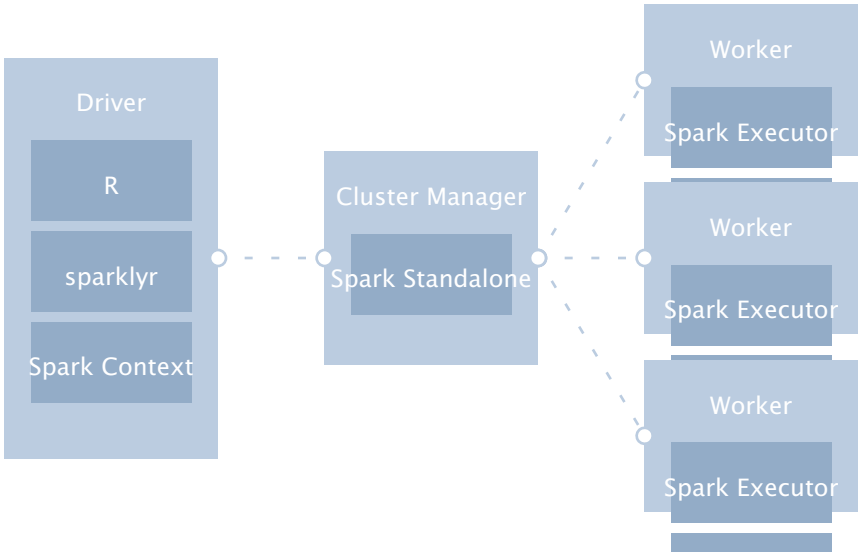


Figure 6.4: Spark Standalone Connection Diagram

6.3 Standalone

Connecting to a Spark Standalone cluster requires the location of the cluster manager's master instance, this location can be found in the cluster manager web interface as described in the clusters-standalone section, you can find this location by looking for a URL starting with `spark://`.

A connection in standalone mode starts from `sparklyr` launching `spark-submit` to submit the `sparklyr` application and creating the Spark Context, which requests executors to Spark's standalone cluster manager in the `master` location:

In order to connect, use `master="spark://hostname:port"` in `spark_connect()` as follows:

```
sc <- spark_connect(master = "spark://hostname:port")
```

6.4 Yarn

Hadoop YARN supports two connection modes: YARN Client and YARN Cluster. However, YARN Client mode is much more common that YARN Cluster since it's more efficient and easier to set up.

6.4.1 Yarn Client

When connecting in YARN Client mode, the driver instance runs R, sparklyr and the Spark Context which requests worker nodes from YARN to run Spark executors as follows:

To connect, one can simply run with `master = "yarn"` as follows:

```
sc <- spark_connect(master = "yarn-client")
```

Once connected, you can use all techniques described in previous chapters using the `sc` connection; for instances, you can do data analysis or modeling.

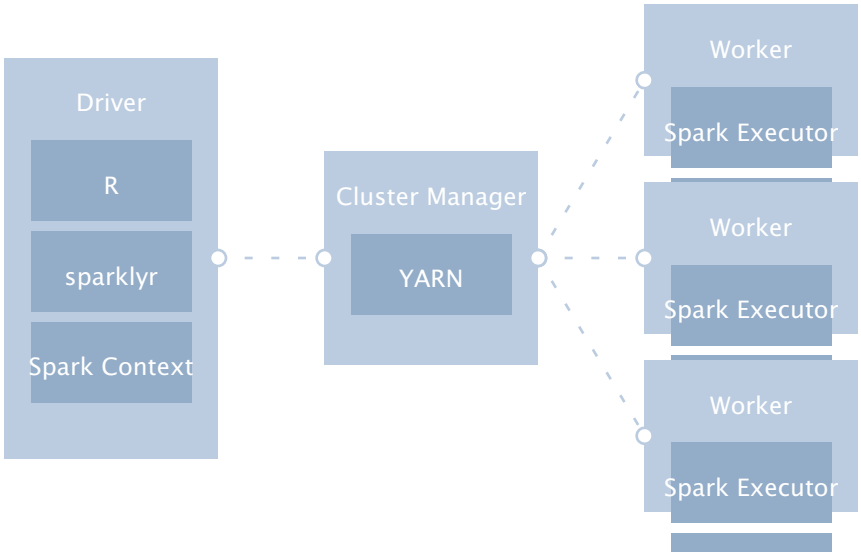


Figure 6.5: YARN Client Connection Diagram

6.4.2 Yarn Cluster

The main difference between YARN Cluster mode and YARN Client mode is that in YARN Cluster mode, the driver node is not required to be the node where R and sparklyr get started; instead, the driver node remains the designated driver node which is usually a different node than the edge node where R is running. It can be helpful to consider using YARN Cluster when the edge node has too many concurrent users, is lacking computing resources or where tools (like RStudio) need to be managed independently of other clsuter resources.

To connect in YARN Cluster mode, we can simple run:

```
sc <- spark_connect(master = "yarn-cluster")
```

This connection assumes that the node running `spark_connect()` is properly configured, meaning that, `yarn-site.xml` exists and the `YARN_CONF_DIR` environment variable is properly set. When using Hadoop as a file system, one would also need the `HADOOP_CONF_DIR` environment variable properly configured. This configuration is usually provided by your system administrator and is not something that you would have to manually configure.

6.5 Livy

As opposed to other connection methods which require using an edge node in the cluster, Livy Livy provides a **Web API** that makes the Spark cluster accessible from outside the cluster and neither requires a local installation in the client. Once connected through the Web API, the **Livy Service** starts the Spark context by requesting reosurces from the cluster manager and distributing work as usual.

Conencting through Livy requires the URL to the Livy service which should be similar to `https://hostname:port/livy`. Since remote connections are allowed, connections usually requires, at the very least, basic authentication:

```
sc <- spark_connect(master = "https://hostname:port/livy", method = "livy", config = livy_config(
  username="<username>",
  password="<password>"
))
```

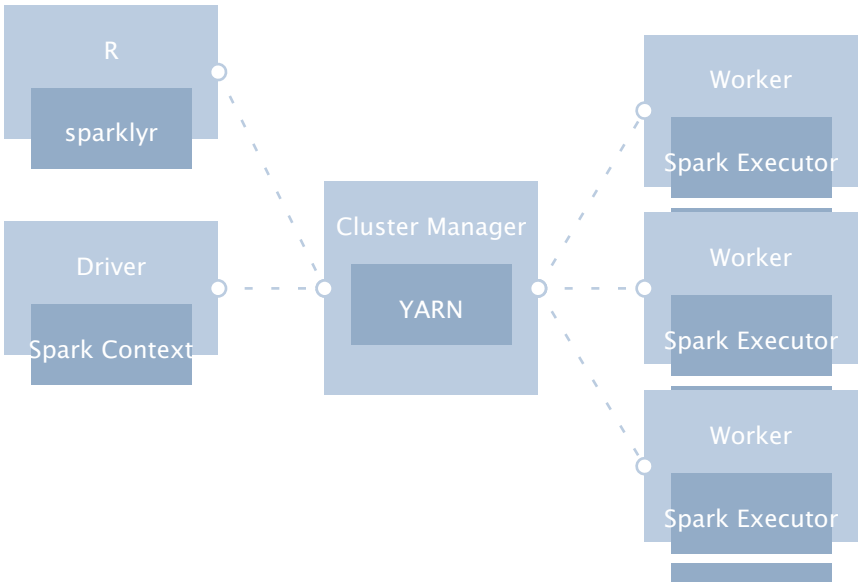


Figure 6.6: YARN Cluster Connection Diagram

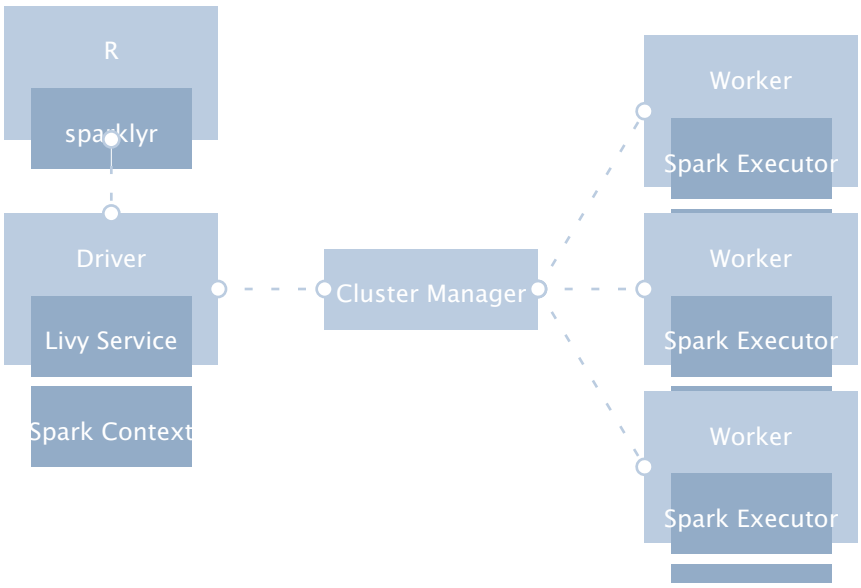


Figure 6.7: Livy Connection Diagram

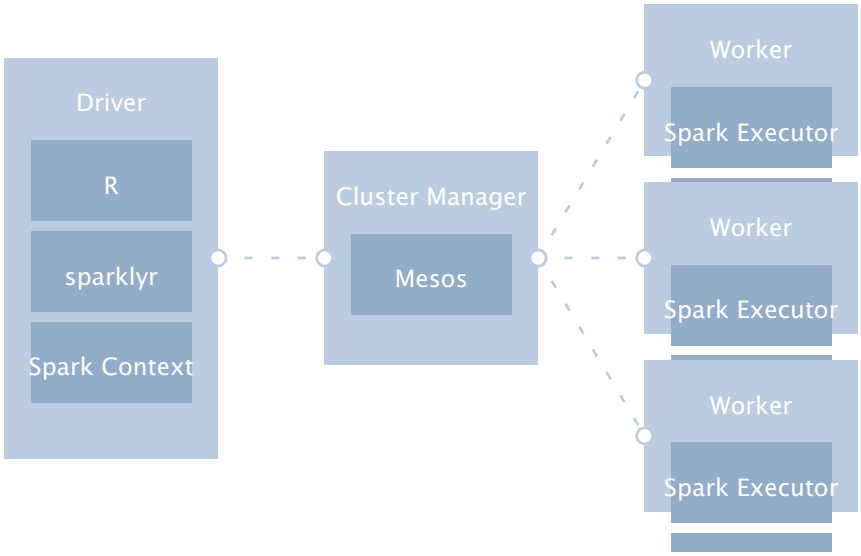


Figure 6.8: Mesos Connection Diagram

Once connected through Livy, operations you can make use of an other `sparklyr` feature; however, Livy is not suitable for experimental data analysis, since executing commands have a significant delay; that said, while running long running computations, this overhead could be considered irrelevant. In general, it is preffered to avoid using Livy and work directly within an edge node in the cluster; if this is not feasible, using Livy could be a reasonable approach.

6.6 Mesos

Similar to YARN, Mesos supports client mode and a cluster mode. However, `sparklyr` currently only supports client mode for Mesos.

Connecting requires the address to the Mesos master node, usually in the form of `mesos://host:port` or `mesos://zk://host1:2181,host2:2181,host3:2181/mesos` for Mesos using ZooKeeper.

```
sc <- spark_connect(master = "mesos://host:port")
```

6.7 Kubernetes

Kubernetes cluster do not support client modes similar to Mesos or YARN, instead, the connection model is similar to YARN Cluster, where the driver node is assigned by Kubernetes.

Kubernetes support is scheduled to be added to `sparklyr` with sparklyr/issues/1525, please follow progress for this feature directly in github. Once Kubernetes becomes supported in `sparklyr`, connecting to Kubernetes will work as follows:

```
sc <- spark_connect(
  master = "k8s://https://<apiserver-host>:<apiserver-port>"
  config = list(
    spark.executor.instances = 2,
    spark.kubernetes.container.image = "spark-image"
```

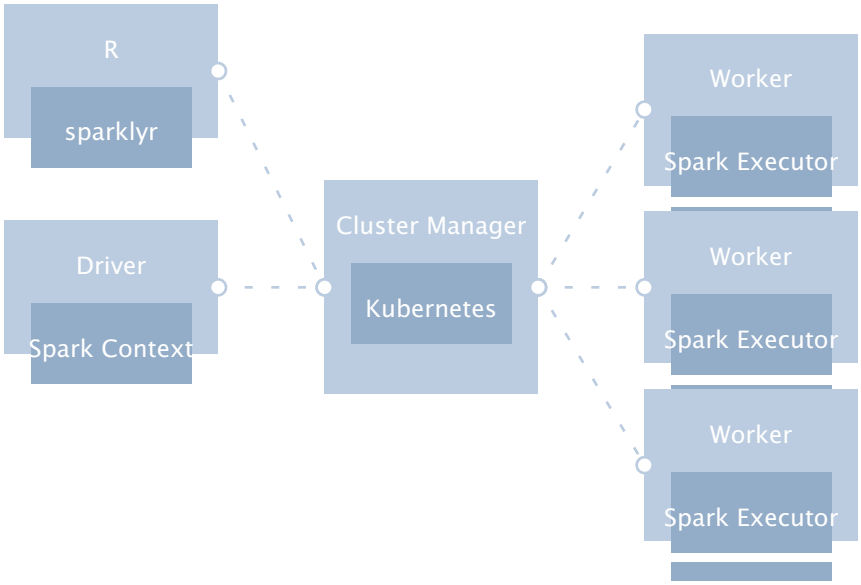


Figure 6.9: Kubernetes Connection Diagram

```
  )
)
```

If your computer is already configured to use a Kubernetes cluster, you can use the following commmand to find the `apiserver-host` and `apiserver-port`:

```
system2("kubectl", "cluster-info")
```

6.8 Multiple

It is common to connect once, and only once, to Spark. However, you can also open multiple connections to Spark by connecting to different clusters or by specifying the `app_name` parameter, this can be helpful to compare Spark versions or validate you analysis before submitting to the cluster. The following example opens connections to Spark 1.6.3, 2.3.0 and Spark Standalone:

```
# Connect to local Spark 1.6.3
sc_1_6_3 <- spark_connect(master = "local", version = "1.6.3")

# Connect to local Spark 2.3.0
sc_2_3_0 <- spark_connect(master = "local", version = "2.3.0", appName = "Spark23")

# Connect to local Spark Standalone
sc_standalone <- spark_connect(master = "spark://host:port")
```

Finally, we can disconnect from each connection:

```
spark_disconnect(sc_1_6_3)
spark_disconnect(sc_2_3_0)
spark_disconnect(sc_standalone)
```

Alternatevely, you can disconnect from all connections at once:

```
spark_disconnect_all()
```

6.9 Recap

This chapter presented an overview of Spark's architecture and detailed connections concepts and examples to connect in local mode, standalone, YARN, Mesos, Kubernetes and Livy. It also presented edge nodes and their role while connecting to Spark clusters. This information should give you enough information to effectevely connect to any cluster with Apache Spark enabled.

To troubleshoot connection problems, it is recommended to search for the connection problem in StackOverflow, the sparklyr github issues and, if needed, open a new GitHub issue in sparklyr to assist further.

The next chapter, Configuration, will present the tools and best practices to configure the resources and behavior of your Spark connection.

Chapter 7

Configuration

This chatper has not been written.

The previous chapter, Connections, explained how to connect to Spark clusters; however, it is often the case that connection is not sufficient to run proper analysis at scale since the default settings (say number of executors, avaialble memory, etc) that a system administrator configured need to be modified to your particular needs. This chapter will explain how to configure, what can be configured and the important use cases you want to consider while configuring your Spark environment.

7.1 Overview

```
config <- spark_config()
sc <- spark_connect(master = "local")

spark_context_config(sc)

spark_session_config(sc)

# Previous versions
spark_hive_config(sc)
```

7.2 Tools

7.3 Resources

https://spark.apache.org/docs/latest/configuration.html

Chapter 8

Sources

This chatper has not been written.

8.1 Text

(What is a text file?) (Diagram)

8.2 CSV

(What is CSV?) (Diagram)

8.3 Parquet

(What is Parquet?) (Diagram)

8.4 Avro

(What is Avro?) (Diagram)

8.5 ORC

(What is ORC?) (Diagram)

8.6 JDBC

(What is JDBC?) (Diagram)

8.7 Generic Sources

Chapter 9

Tuning

This chatper has not been written.

9.1 Overview

(explain RDDs)

9.2 Caching

Most sparklyr operations that retrieve a Spark data frame, cache the results in-memory, for instance, running `spark_read_parquet()` or `sdf_copy_to()` will provide a Spark dataframe that is already cached in-memory. As a Spark data frame, this object can be used in most sparklyr functions, including data analysis with dplyr or machine learning.

```
library(sparklyr)
sc <- spark_connect(master = "local")
```

```
iris_tbl <- sdf_copy_to(sc, iris, overwrite = TRUE)
```

You can inspect which tables are cached by navigating to the Spark UI using `spark_web(sc)`, opening the storage tab, and clicking on a given RDD:

Data loaded in memory will be released when the R session terminates either explicitly or implicitly with a restart or disconnection; however, to free up resources, you can use `tbl_uncache()`:

```
tbl_uncache(sc, "iris")
```

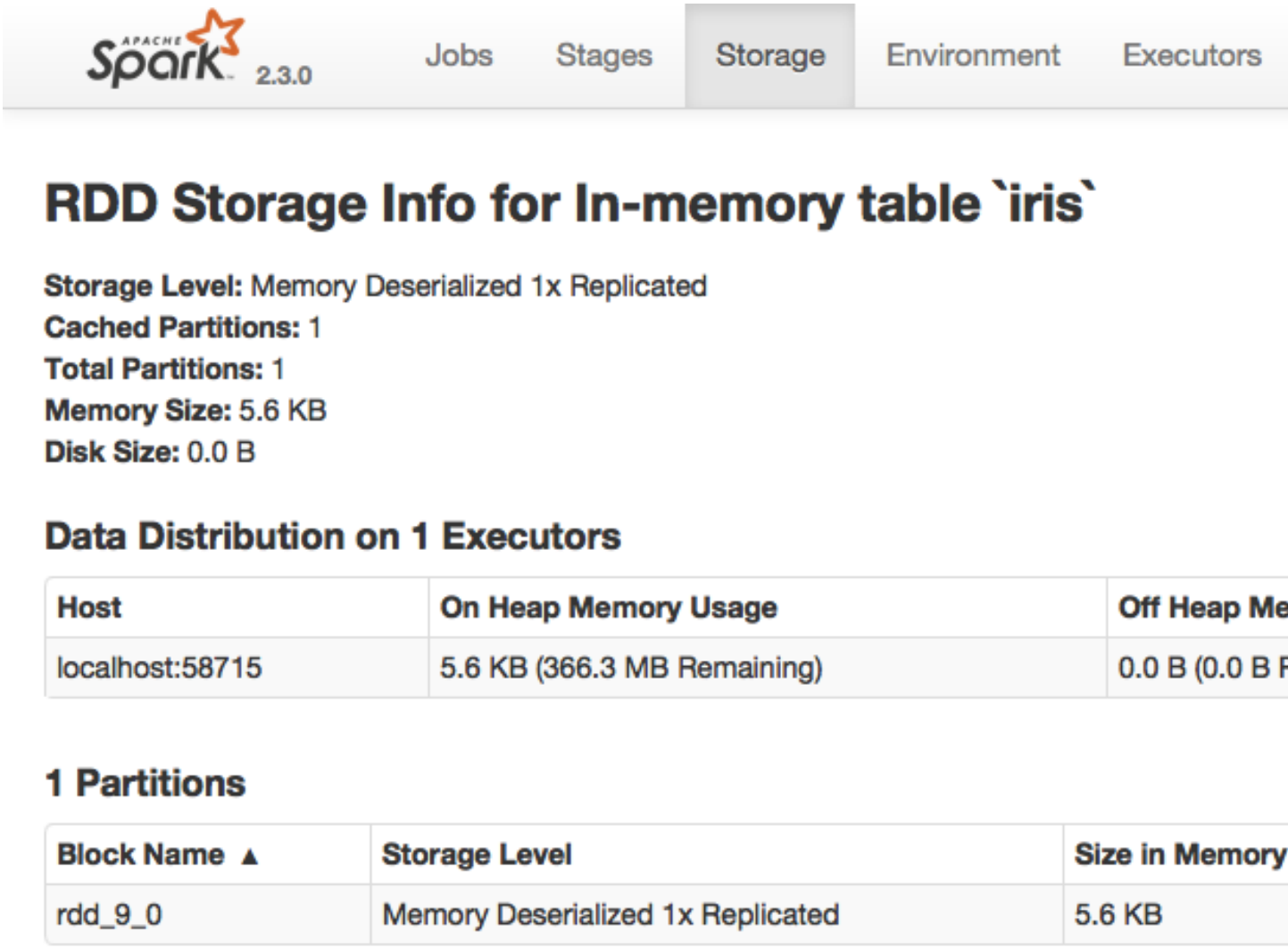


Figure 9.1: Cached RDD in Spark Web Interface.

9.3 Partitions

9.4 Shuffling

9.5 Checkpointing

9.6 Troubleshooting

Logs, `spark_debug_string`

Chapter 10

Extensions

While **this chatper has not been written.**, a few resources are available to help explore these topics until this chapter gets written.

10.1 Using Extensions

10.1.1 GraphFrames

See spark.rstudio.com/graphframes.

10.1.2 Mleap

See spark.rstudio.com/guides/mleap.

10.1.3 H2O

See spark.rstudio.com/guides/h2o.

10.1.4 Others

See also github.com/chezou/sparkavro.

10.2 Writting Extensions

See spark.rstudio.com/extensions.

10.3 Pipelines

This chatper has not been written.

10.4 Overview

Spark's ML Pipelines provide a way to easily combine multiple transformations and algorithms into a single workflow, or pipeline.

Take a look at spark.rstudio.com/guides/pipelines to learn about their purpose and functionality.

Chapter 11

Distributed R

While **this chatper has not been written.**, use spark.rstudio.com/guides/distributed-r to learn how to use R directly over each worker node.

11.1 Use Cases

11.2 Packages

11.3 Restrictions

11.4 Troubleshooting

Chapter 12

Contributing

This chatper has not been written.

While there are many ways to contribute to `sparklyr` from helping community members to opening github issues, this chapter focuses on those readers interested in contributing fixes and features to `sparklyr` but will also help those who want to understand how `sparklyr` works internally.

12.1 Overview

(architecture overview)

12.2 Serialization

12.3 Invocations

12.4 R Packages

(dbi, dplyr, broom, etc)

12.5 Connections

12.6 Distributed R

Appendix

12.7 Worlds Store Capacity

```
library(tidyverse)
read_csv("data/01-worlds-capacity-to-store-information.csv", skip = 8) %>%
  gather(key = storage, value = capacity, analog, digital) %>%
  mutate(year = X1, terabytes = capacity / 1e+12) %>%
  ggplot(aes(x = year, y = terabytes, group = storage)) +
    geom_line(aes(linetype = storage)) +
    geom_point(aes(shape = storage)) +
    scale_y_log10(
      breaks = scales::trans_breaks("log10", function(x) 10^x),
      labels = scales::trans_format("log10", scales::math_format(10^x))
    ) +
    theme_light() +
    theme(legend.position = "bottom")
```

12.8 Daily downloads of CRAN packages

```
downloads_csv <- "data/01-intro-r-cran-downloads.csv"
if (!file.exists(downloads_csv)) {
  downloads <- cranlogs::cran_downloads(from = "2014-01-01", to = "2018-01-01")
  readr::write_csv(downloads, downloads_csv)
}

cran_downloads <- readr::read_csv(downloads_csv)

ggplot(cran_downloads, aes(date, count)) +
  geom_point(colour="black", pch = 21, size = 1) +
  scale_x_date() +
  xlab("") +
  ylab("") +
  theme_light()
```

12.9 Google trends for mainframes, cloud computing and kubernetes

```
library(r2d3)

read.csv("data/05-cluster-trends.csv") %>%
  mutate(month = as.Date(paste(month, "-01", sep = ""))) %>%
  r2d3(script="images/05-clusters-trends.js")
```

Bibliography

French, C. (1996). *Data Processing and Information Technology.* Cengage Learning Business Press.