

ADDISON
WESLEY
DATA &
ANALYTICS
SERIES



Data Analytics
with **SPARK**
Using **PYTHON**



JEFFREY AVEN

Table of Contents

Preface xi

Introduction 1

I: Spark Foundations

1 Introducing Big Data, Hadoop, and Spark 5

Introduction to Big Data, Distributed Computing, and Hadoop 5

A Brief History of Big Data and Hadoop 6

Hadoop Explained 7

Introduction to Apache Spark 13

Apache Spark Background 13

Uses for Spark 14

Programming Interfaces to Spark 14

Submission Types for Spark Programs 14

Input/Output Types for Spark Applications 16

The Spark RDD 16

Spark and Hadoop 16

Functional Programming Using Python 17

Data Structures Used in Functional Python Programming 17

Python Object Serialization 20

Python Functional Programming Basics 23

Summary 25

2 Deploying Spark 27

Spark Deployment Modes 27

Local Mode 28

Spark Standalone 28

Spark on YARN 29

Spark on Mesos 30

Preparing to Install Spark 30

Getting Spark 31

Installing Spark on Linux or Mac OS X 32

Installing Spark on Windows 34

Exploring the Spark Installation 36

Deploying a Multi-Node Spark Standalone Cluster 37

Deploying Spark in the Cloud	39
Amazon Web Services (AWS)	39
Google Cloud Platform (GCP)	41
Databricks	42
Summary	43
3 Understanding the Spark Cluster Architecture	45
Anatomy of a Spark Application	45
Spark Driver	46
Spark Workers and Executors	49
The Spark Master and Cluster Manager	51
Spark Applications Using the Standalone Scheduler	53
Spark Applications Running on YARN	53
Deployment Modes for Spark Applications Running on YARN	53
Client Mode	54
Cluster Mode	55
Local Mode Revisited	56
Summary	57
4 Learning Spark Programming Basics	59
Introduction to RDDs	59
Loading Data into RDDs	61
Creating an RDD from a File or Files	61
Methods for Creating RDDs from a Text File or Files	63
Creating an RDD from an Object File	66
Creating an RDD from a Data Source	66
Creating RDDs from JSON Files	69
Creating an RDD Programmatically	71
Operations on RDDs	72
Key RDD Concepts	72
Basic RDD Transformations	77
Basic RDD Actions	81
Transformations on PairRDDs	85
MapReduce and Word Count Exercise	92
Join Transformations	95
Joining Datasets in Spark	100
Transformations on Sets	103
Transformations on Numeric RDDs	105
Summary	108

II: Beyond the Basics

5 Advanced Programming Using the Spark Core API 111

Shared Variables in Spark	111
Broadcast Variables	112
Accumulators	116
Exercise: Using Broadcast Variables and Accumulators	119
Partitioning Data in Spark	120
Partitioning Overview	120
Controlling Partitions	121
Repartitioning Functions	123
Partition-Specific or Partition-Aware API Methods	125
RDD Storage Options	127
RDD Lineage Revisited	127
RDD Storage Options	128
RDD Caching	131
Persisting RDDs	131
Choosing When to Persist or Cache RDDs	134
Checkpointing RDDs	134
Exercise: Checkpointing RDDs	136
Processing RDDs with External Programs	138
Data Sampling with Spark	139
Understanding Spark Application and Cluster Configuration	141
Spark Environment Variables	141
Spark Configuration Properties	145
Optimizing Spark	148
Filter Early, Filter Often	149
Optimizing Associative Operations	149
Understanding the Impact of Functions and Closures	151
Considerations for Collecting Data	152
Configuration Parameters for Tuning and Optimizing Applications	152
Avoiding Inefficient Partitioning	153
Diagnosing Application Performance Issues	155
Summary	159

6 SQL and NoSQL Programming with Spark 161

Introduction to Spark SQL	161
Introduction to Hive	162
Spark SQL Architecture	166

Getting Started with DataFrames	168
Using DataFrames	179
Caching, Persisting, and Repartitioning DataFrames	187
Saving DataFrame Output	188
Accessing Spark SQL	191
Exercise: Using Spark SQL	194
Using Spark with NoSQL Systems	195
Introduction to NoSQL	196
Using Spark with HBase	197
Exercise: Using Spark with HBase	200
Using Spark with Cassandra	202
Using Spark with DynamoDB	204
Other NoSQL Platforms	206
Summary	206
7 Stream Processing and Messaging Using Spark	209
Introducing Spark Streaming	209
Spark Streaming Architecture	210
Introduction to DStreams	211
Exercise: Getting Started with Spark Streaming	218
State Operations	219
Sliding Window Operations	221
Structured Streaming	223
Structured Streaming Data Sources	224
Structured Streaming Data Sinks	225
Output Modes	226
Structured Streaming Operations	227
Using Spark with Messaging Platforms	228
Apache Kafka	229
Exercise: Using Spark with Kafka	234
Amazon Kinesis	237
Summary	240
8 Introduction to Data Science and Machine Learning Using Spark	243
Spark and R	243
Introduction to R	244
Using Spark with R	250
Exercise: Using RStudio with SparkR	257

Machine Learning with Spark	259
Machine Learning Primer	259
Machine Learning Using Spark MLlib	262
Exercise: Implementing a Recommender Using Spark MLlib	267
Machine Learning Using Spark ML	271
Using Notebooks with Spark	275
Using Jupyter (IPython) Notebooks with Spark	275
Using Apache Zeppelin Notebooks with Spark	278
Summary	279
Index	281

Sample pages

Understanding the Spark Cluster Architecture

It is not the beauty of a building you should look at; it's the construction of the foundation that will stand the test of time.

David Allan Coe, American songwriter

In This Chapter:

- Detailed overview of the Spark application and cluster components
- Spark resource schedulers and Cluster Managers
- How Spark applications are scheduled on YARN clusters
- Spark deployment modes

Before you begin your journey as a Spark programmer, you should have a solid understanding of the Spark application architecture and how applications are executed on a Spark cluster. This chapter closely examines the components of a Spark application, looks at how these components work together, and looks at how Spark applications run on Standalone and YARN clusters.

Anatomy of a Spark Application

A Spark application contains several components, all of which exist whether you're running Spark on a single machine or across a cluster of hundreds or thousands of nodes.

Each component has a specific role in executing a Spark program. Some of these roles, such as the client components, are passive during execution; other roles are active in the execution of the program, including components executing computation functions.

The components of a Spark application are the *Driver*, the *Master*, the *Cluster Manager*, and the *Executor(s)*, which run on worker nodes, or *Workers*. Figure 3.1 shows all the Spark components in the context of a Spark Standalone application. You will learn more about each component and its function in more detail later in this chapter.

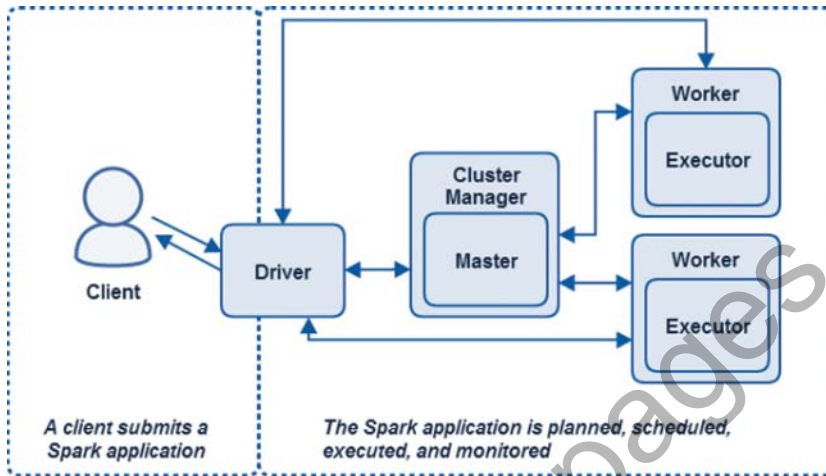


Figure 3.1 Spark Standalone cluster application components.

All Spark components, including the Driver, Master, and Executor processes, run in Java virtual machines (JVMs). A JVM is a cross-platform runtime engine that can execute instructions compiled into Java bytecode. Scala, which Spark is written in, compiles into bytecode and runs on JVMs.

It is important to distinguish between Spark's runtime application components and the locations and node types on which they run. These components run in different places using different deployment modes, so don't think of these components in physical node or instance terms. For instance, when running Spark on YARN, there would be several variations of Figure 3.1. However, all the components pictured are still involved in the application and have the same roles.

Spark Driver

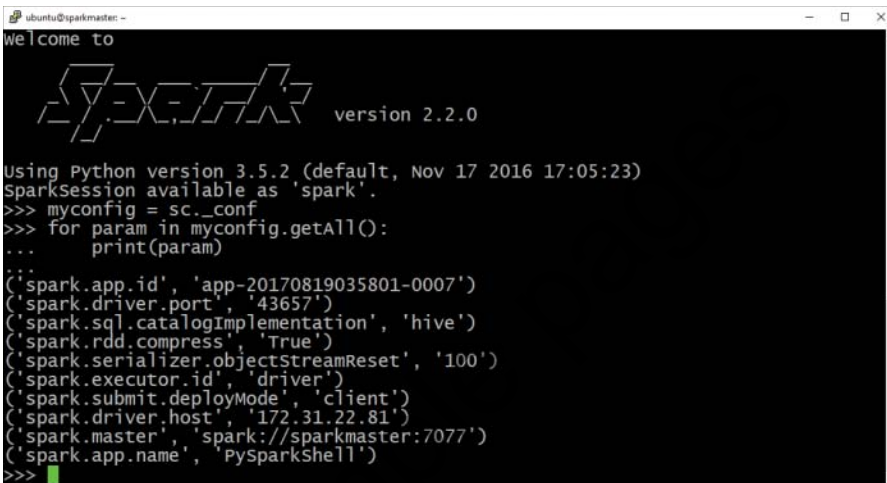
The life of a Spark application starts and finishes with the Spark Driver. The Driver is the process that clients use to submit applications in Spark. The Driver is also responsible for planning and coordinating the execution of the Spark program and returning status and/or results (data) to the client. The Driver can physically reside on a client or on a node in the cluster, as you will see later.

SparkSession

The Spark Driver is responsible for creating the *SparkSession*. The *SparkSession* object represents a connection to a Spark cluster. The *SparkSession* is instantiated at the beginning of a Spark application, including the interactive shells, and is used for the entirety of the program.

Prior to Spark 2.0, entry points for Spark applications included the `SparkContext`, used for Spark core applications; the `SQLContext` and `HiveContext`, used with Spark SQL applications; and the `StreamingContext`, used for Spark Streaming applications. The `SparkSession` object introduced in Spark 2.0 combines all these objects into a single entry point that can be used for all Spark applications.

Through its `SparkContext` and `SparkConf` child objects, the `SparkSession` object contains all the runtime configuration properties set by the user, including configuration properties such as the Master, application name, number of Executors, and more. Figure 3.2 shows the `SparkSession` object and some of its configuration properties within a `pyspark` shell.



```

ubuntu@sparkmaster: ~
Welcome to

      _____
     /  _  /  Spark  version 2.2.0
    /  /  /
   /  /  /

Using Python version 3.5.2 (default, Nov 17 2016 17:05:23)
SparkSession available as 'spark'.
>>> myconfig = sc._conf
>>> for param in myconfig.getAll():
...     print(param)
...
('spark.app.id', 'app-20170819035801-0007')
('spark.driver.port', '43657')
('spark.sql.catalogImplementation', 'hive')
('spark.rdd.compress', 'True')
('spark.serializer.objectStreamReset', '100')
('spark.executor.id', 'driver')
('spark.submit.deployMode', 'client')
('spark.driver.host', '172.31.22.81')
('spark.master', 'spark://sparkmaster:7077')
('spark.app.name', 'pysparkShell')
>>>

```

Figure 3.2 SparkSession properties.

SparkSession Name

The object name for the `SparkSession` instance is arbitrary. By default, the `SparkSession` instantiation in the Spark interactive shells is named `spark`. For consistency, you always instantiate the `SparkSession` as `spark`; however, the name is up to the developer's discretion.

Listing 3.1 demonstrates how to create a `SparkSession` within a non-interactive Spark application, such as a program submitted using `spark-submit`.

Listing 3.1 Creating a SparkSession

```

from pyspark.sql import SparkSession
spark = SparkSession.builder \
    .master("spark://sparkmaster:7077") \
    .appName("My Spark Application") \

```

```

    .config("spark.submit.deployMode", "client") \
    .getOrCreate()
numlines = spark.sparkContext.textFile("file:///opt/spark/licenses") \
    .count()
print("The total number of lines is " + str(numlines))

```

Application Planning

One of the main functions of the Driver is to plan the application. The Driver takes the application processing input and plans the execution of the program. The Driver takes all the requested *transformations* (data manipulation operations) and *actions* (requests for output or prompts to execute programs) and creates a *directed acyclic graph (DAG)* of *nodes*, each representing a transformational or computational step.

Directed Acyclic Graph (DAG)

A DAG is a mathematical construct that is commonly used in computer science to represent dataflows and their dependencies. DAGs contain vertices, or nodes, and edges. Vertices in a dataflow context are steps in the process flow. Edges in a DAG connect vertices to one another in a directed orientation and in such a way that it is impossible to have circular references.

A Spark application DAG consists of *tasks* and *stages*. A task is the smallest unit of schedulable work in a Spark program. A stage is a set of tasks that can be run together. Stages are dependent upon one another; in other words, there are *stage dependencies*.

In a process scheduling sense, DAGs are not unique to Spark. For instance, they are used in other Big Data ecosystem projects, such as Tez, Drill, and Presto for scheduling. DAGs are fundamental to Spark, so it is worth being familiar with the concept.

Application Orchestration

The Driver also coordinates the running of stages and tasks defined in the DAG. Key driver activities involved in the scheduling and running of tasks include the following:

- Keeping track of available resources to execute tasks
- Scheduling tasks to run “close” to the data where possible (the concept of data locality)

Other Functions

In addition to planning and orchestrating the execution of a Spark program, the Driver is also responsible for returning the results from an application. These could be return codes or data in the case of an action that requests data to be returned to the client (for example, an interactive query).

The Driver also serves the application UI on port 4040, as shown in Figure 3.3. This UI is created automatically; it is independent of the code submitted or how it was submitted (that is, interactive using `pyspark` or non-interactive using `spark-submit`).

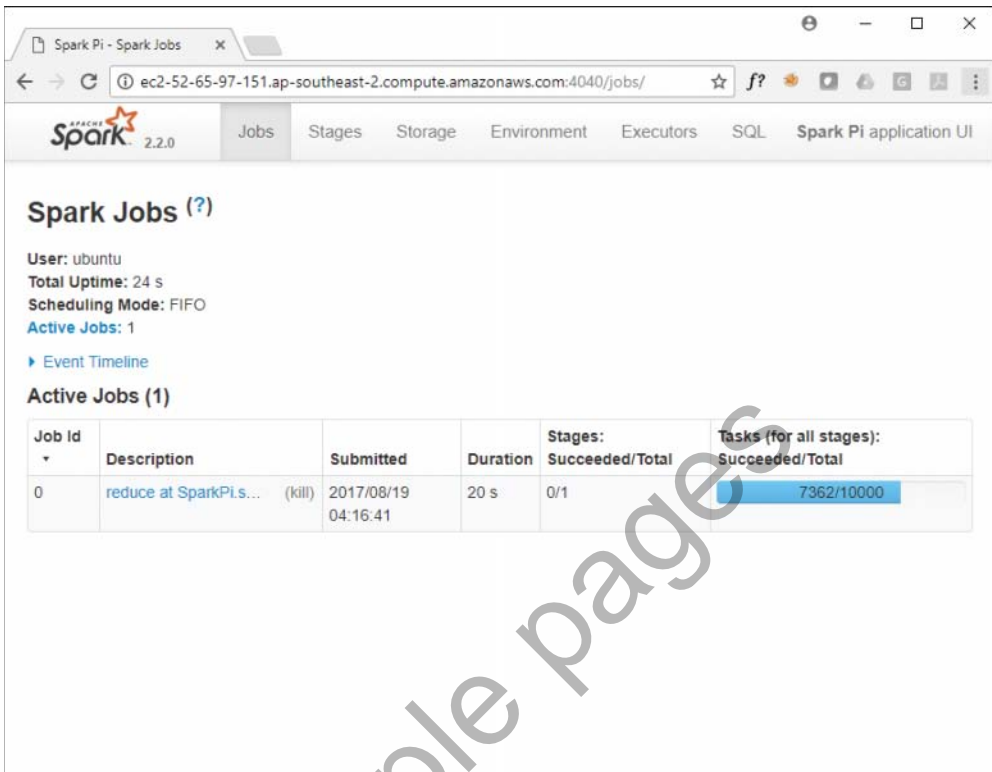


Figure 3.3 Spark application UI.

If subsequent applications launch on the same host, successive ports are used for the application UI (for example, 4041, 4042, and so on).

Spark Workers and Executors

Spark Executors are the processes on which Spark DAG tasks run. Executors reserve CPU and memory resources on slave nodes, or Workers, in a Spark cluster. An Executor is dedicated to a specific Spark application and terminated when the application completes. A Spark program normally consists of many Executors, often working in parallel.

Typically, a Worker node—which hosts the Executor process—has a finite or fixed number of Executors allocated at any point in time. Therefore, a cluster—being a known number of nodes—has a finite number of Executors available to run at any given time. If an application requires Executors in excess of the physical capacity of the cluster, they are scheduled to start as other Executors complete and release their resources.

As mentioned earlier in this chapter, JVMs host Spark Executors. The JVM for an Executor is allocated a *heap*, which is a dedicated memory space in which to store and manage objects.

The amount of memory committed to the JVM heap for an Executor is set by the property `spark.executor.memory` or as the `--executor-memory` argument to the `pyspark`, `spark-shell`, or `spark-submit` commands.

Executors store output data from tasks in memory or on disk. It is important to note that Workers and Executors are aware only of the tasks allocated to them, whereas the Driver is responsible for understanding the complete set of tasks and the respective dependencies that comprise an application.

By using the Spark application UI on port 404x of the Driver host, you can inspect Executors for the application, as shown in Figure 3.4.

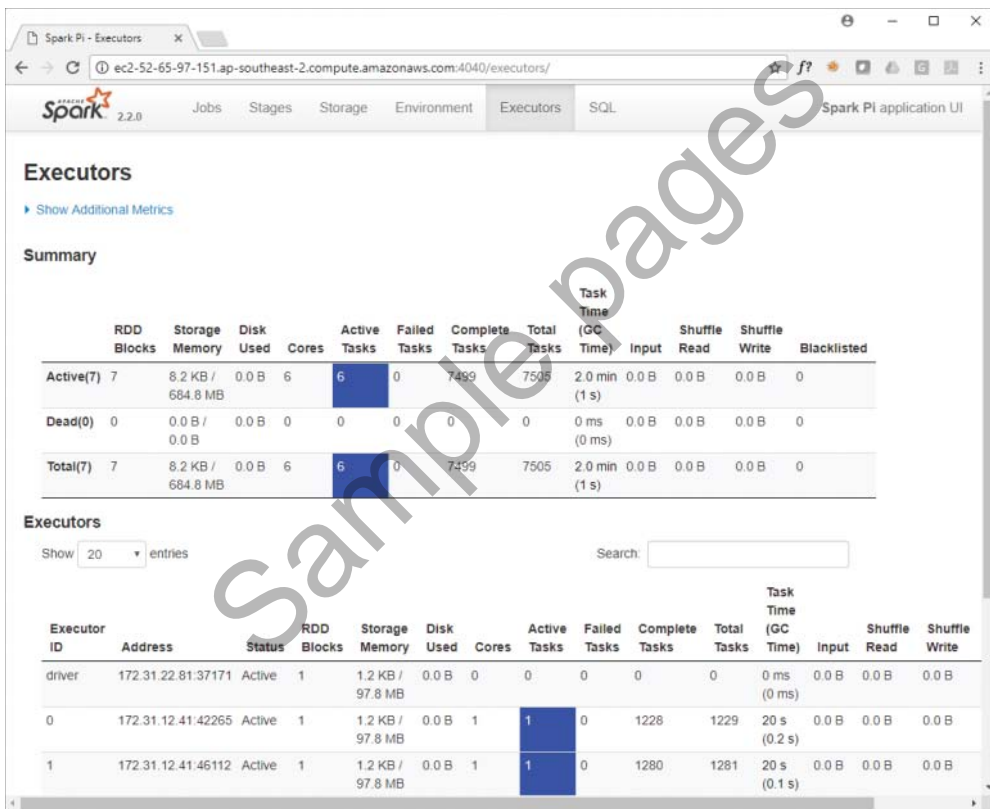


Figure 3.4 Executors tab in the Spark application UI.

For Spark Standalone cluster deployments, a worker node exposes a user interface on port 8081, as shown in Figure 3.5.

The screenshot shows the Spark Worker UI for a worker at IP 172.31.12.41:37870. The page displays the following information:

- Worker ID:** worker-20170819033739-172.31.12.41-37870
- Master URL:** spark://sparkmaster:7077
- Cores:** 2 (0 Used)
- Memory:** 2.9 GB (0.0 B Used)
- Running Executors:** 0
- Finished Executors:** 18

The 'Finished Executors' table is as follows:

ExecutorID	Cores	State	Memory	Job Details	Logs
1	1	KILLED	512.0 MB	ID: app-20170819033846-0000 Name: Spark Pi User: ubuntu	stdout stderr
0	1	KILLED	512.0 MB	ID: app-20170819033846-0000 Name: Spark Pi User: ubuntu	stdout stderr
0	2	KILLED	1024.0 MB	ID: app-20170819034534-0001 Name: PySparkShell User: ubuntu	stdout stderr

Figure 3.5 Spark Worker UI.

The Spark Master and Cluster Manager

The Spark Driver plans and coordinates the set of tasks required to run a Spark application. The tasks themselves run in Executors, which are hosted on Worker nodes.

The Master and the Cluster Manager are the central processes that monitor, reserve, and allocate the distributed cluster resources (or containers, in the case of YARN or Mesos) on which the Executors run. The Master and the Cluster Manager can be separate processes, or they can combine into one process, as is the case when running Spark in Standalone mode.

Spark Master

The Spark Master is the process that requests resources in the cluster and makes them available to the Spark Driver. In all deployment modes, the Master negotiates resources or containers with Worker nodes or slave nodes and tracks their status and monitors their progress.

When running Spark in Standalone mode, the Spark Master process serves a web UI on port 8080 on the Master host, as shown in Figure 3.6.

The screenshot shows the Spark Master web interface. At the top, it displays the Spark logo and version 2.2.0, followed by the URL 'Spark Master at spark://sparkmaster:7077'. Below this, it lists various cluster metrics: URL, REST URL, Alive Workers (3), Cores in use (6 Total, 0 Used), Memory in use (8.6 GB Total, 0.0 B Used), Applications (0 Running, 15 Completed), Drivers (0 Running, 0 Completed), and Status (ALIVE).

The 'Workers' section contains a table with the following data:

Worker Id	Address	State	Cores	Memory
worker-20170819033739-172.31.12.41-37870	172.31.12.41:37870	ALIVE	2 (0 Used)	2.9 GB (0.0 B Used)
worker-20170819033747-172.31.9.150-38844	172.31.9.150:38844	ALIVE	2 (0 Used)	2.9 GB (0.0 B Used)
worker-20170819033751-172.31.12.22-34084	172.31.12.22:34084	ALIVE	2 (0 Used)	2.9 GB (0.0 B Used)

The 'Running Applications' section shows a table with columns: Application ID, Name, Cores, Memory per Executor, Submitted Time, User, State, and Duration. It is currently empty.

The 'Completed Applications' section shows a table with columns: Application ID, Name, Cores, Memory per Executor, Submitted Time, User, State, and Duration. It contains two entries:

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20170819042055-0014	Spark PI	6	512.0 MB	2017/08/19 04:20:55	ubuntu	FINISHED	27 s
app-20170819041639-0013	Spark PI	6	512.0 MB	2017/08/19 04:16:39	ubuntu	FINISHED	27 s

Figure 3.6 Spark Master UI.

Spark Master Versus Spark Driver

It is important to distinguish the runtime functions of the Driver and the Master. The name *Master* may be inferred to mean that this process is governing the execution of the application—but this is not the case. The Master simply requests resources and makes those resources available to the Driver. Although the Master monitors the status and health of these resources, it is not involved in the execution of the application and the coordination of its tasks and stages. That is the job of the Driver.

Cluster Manager

The Cluster Manager is the process responsible for monitoring the Worker nodes and reserving resources on these nodes upon request by the Master. The Master then makes these cluster resources available to the Driver in the form of Executors.

As discussed earlier, the Cluster Manager can be separate from the Master process. This is the case when running Spark on Mesos or YARN. In the case of Spark running in Standalone mode, the Master process also performs the functions of the Cluster Manager. Effectively, it acts as its own Cluster Manager.

A good example of the Cluster Manager function is the YARN ResourceManager process for Spark applications running on Hadoop clusters. The ResourceManager schedules, allocates, and monitors the health of containers running on YARN NodeManagers. Spark applications then use these containers to host Executor processes, as well as the Master process if the application is running in `cluster` mode; we will look at this shortly.

Spark Applications Using the Standalone Scheduler

In Chapter 2, “Deploying Spark,” you learned about the Standalone scheduler as a deployment option for Spark. You also deployed a fully functional multi-node Spark Standalone cluster in one of the exercises in Chapter 2. As discussed earlier, in a Spark cluster running in Standalone mode, the Spark Master process performs the Cluster Manager function as well, governing available resources on the cluster and granting them to the Master process for use in a Spark application.

Spark Applications Running on YARN

As discussed previously, Hadoop is a very popular and common deployment platform for Spark. Some industry pundits believe that Spark will soon supplant MapReduce as the primary processing platform for applications in Hadoop. Spark applications on YARN share the same runtime architecture but have some slight differences in implementation.

ResourceManager as the Cluster Manager

In contrast to the Standalone scheduler, the Cluster Manager in a YARN cluster is the YARN ResourceManager. The ResourceManager monitors resource usage and availability across all nodes in a cluster. Clients submit Spark applications to the YARN ResourceManager. The ResourceManager allocates the first container for the application, a special container called the *ApplicationMaster*.

ApplicationMaster as the Spark Master

The ApplicationMaster is the Spark Master process. As the Master process does in other cluster deployments, the ApplicationMaster negotiates resources between the application Driver and the Cluster Manager (or ResourceManager in this case); it then makes these resources (containers) available to the Driver for use as Executors to run tasks and store data for the application. The ApplicationMaster remains for the lifetime of the application.

Deployment Modes for Spark Applications Running on YARN

Two deployment modes can be used when submitting Spark applications to a YARN cluster: Client mode and Cluster mode. Let’s look at them now.

Client Mode

In Client mode, the Driver process runs on the client submitting the application. It is essentially unmanaged; if the Driver host fails, the application fails. Client mode is supported for both interactive shell sessions (`pyspark`, `spark-shell`, and so on) and non-interactive application submission (`spark-submit`). Listing 3.2 shows how to start a `pyspark` session using the Client deployment mode.

Listing 3.2 YARN Client Deployment Mode

```

$SPARK_HOME/bin/pyspark \
--master yarn-client \
--num-executors 1 \
--driver-memory 512m \
--executor-memory 512m \
--executor-cores 1
# OR
$SPARK_HOME/bin/pyspark \
--master yarn \
--deploy-mode client \
--num-executors 1 \
--driver-memory 512m \
--executor-memory 512m \
--executor-cores 1

```

Figure 3.7 provides an overview of a Spark application running on YARN in Client mode.

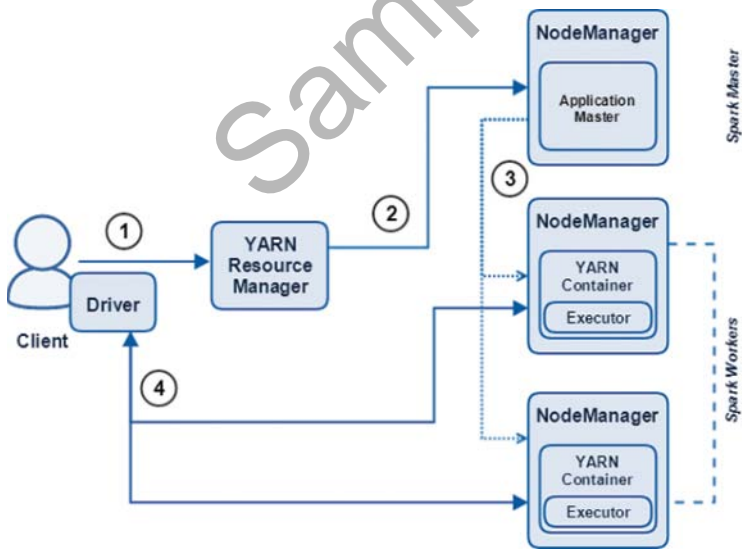


Figure 3.7 Spark application running in YARN Client mode.

The steps shown in Figure 3.7 are described here:

1. The client submits a Spark application to the Cluster Manager (the YARN ResourceManager). The Driver process, SparkSession, and SparkContext are created and run on the client.
2. The ResourceManager assigns an ApplicationMaster (the Spark Master) for the application.
3. The ApplicationMaster requests containers to be used for Executors from the ResourceManager. With the containers assigned, the Executors spawn.
4. The Driver, located on the client, then communicates with the Executors to marshal processing of tasks and stages of the Spark program. The Driver returns the progress, results, and status to the client.

The Client deployment mode is the simplest mode to use. However, it lacks the resiliency required for most production applications.

Cluster Mode

In contrast to the Client deployment mode, with a Spark application running in YARN Cluster mode, the Driver itself runs on the cluster as a subprocess of the ApplicationMaster. This provides resiliency: If the ApplicationMaster process hosting the Driver fails, it can be re-instantiated on another node in the cluster.

Listing 3.3 shows how to submit an application by using `spark-submit` and the YARN Cluster deployment mode. Because the Driver is an asynchronous process running in the cluster, Cluster mode is not supported for the interactive shell applications (`pyspark` and `spark-shell`).

Listing 3.3 YARN Cluster Deployment Mode

```
$SPARK_HOME/bin/spark-submit \  
--master yarn-cluster \  
--num-executors 1 \  
--driver-memory 512m \  
--executor-memory 512m \  
--executor-cores 1  
$SPARK_HOME/examples/src/main/python/pi.py 10000  
# OR  
$SPARK_HOME/bin/spark-submit \  
--master yarn \  
--deploy-mode cluster \  
--num-executors 1 \  
--driver-memory 512m \  
--executor-memory 512m \  
--executor-cores 1  
$SPARK_HOME/examples/src/main/python/pi.py 10000
```

Figure 3.8 provides an overview of a Spark application running on YARN in Cluster mode.

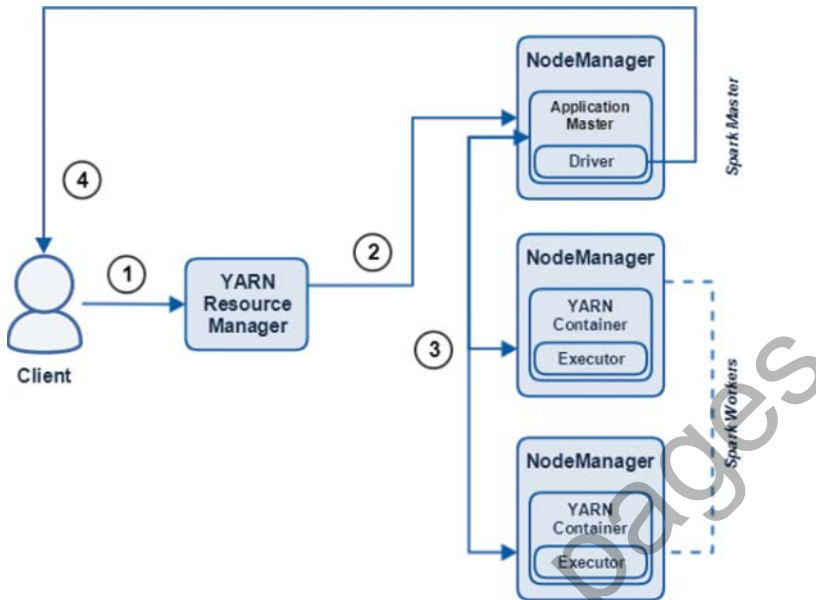


Figure 3.8 Spark application running in YARN Cluster mode.

The steps shown in Figure 3.8 are described here:

1. The client, a user process that invokes `spark-submit`, submits a Spark application to the Cluster Manager (the YARN Resource Manager).
2. The Resource Manager assigns an Application Master (the Spark Master) for the application. The Driver process is created on the same cluster node.
3. The Application Master requests containers for Executors from the Resource Manager. Executors are spawned within the containers allocated to the Application Master by the Resource Manager. The Driver then communicates with the Executors to marshal processing of tasks and stages of the Spark program.
4. The Driver, running on a node in the cluster, returns progress, results, and status to the client.

The Spark application web UI, as shown previously, is available from the Application Master host in the cluster; a link to this user interface is available from the YARN Resource Manager UI.

Local Mode Revisited

In Local mode, the Driver, the Master, and the Executor all run in a single JVM. As discussed earlier in this chapter, this is useful for development, unit testing, and debugging, but it has

limited use for running production applications because it is not distributed and does not scale. Furthermore, failed tasks in a Spark application running in Local mode are not re-executed by default. You can override this behavior, however.

When running Spark in Local mode, the application UI is available at <http://localhost:4040>. The Master and Worker UIs are not available when running in Local mode.

Summary

In this chapter, you have learned about the Spark runtime application and cluster architecture, the components of a Spark application, and the functions of these components. The components of a Spark application include the Driver, Master, Cluster Manager, and Executors. The Driver is the process that the client interacts with when launching a Spark application, either through one of the interactive shells or through the `spark-submit` script. The Driver is responsible for creating the `SparkSession` object (the entry point for any Spark application) and planning an application by creating a DAG consisting of tasks and stages. The Driver communicates with a Master, which in turn communicates with a Cluster Manager to allocate application runtime resources (containers) on which Executors will run. Executors are specific to a given application and run all tasks for the application; they also store output data from completed tasks. Spark's runtime architecture is essentially the same regardless of the cluster resource scheduler used (Standalone, YARN, Mesos, and so on).

Now that we have explored Spark's cluster architecture, it's time to put the concepts into action starting in the next chapter.