**Chapter 17. Deploying Spark**

This chapter explores the infrastructure you need in place for you and your team to be able to run Spark Applications:

* Cluster deployment choices
* Spark’s different cluster managers
* Deployment considerations and configuring deployments

For the most, part Spark should work similarly with all the supported cluster managers; however, customizing the setup means understanding the intricacies of each of the cluster management systems. The hard part is deciding on the cluster manager (or choosing a managed service). Although we would be happy to include all the minute details about how you can configure different cluster with different cluster managers, it’s simply impossible for this book to provide hyper-specific details for every situation in every single enviroment. The goal of this chapter, therefore, is not to discuss each of the cluster managers in full detail, but rather to look at their fundamental differences and to provide a reference for a lot of the material already available on the Spark website. Unfortunately, there is no easy answer to “which is the easiest cluster manager to run” because it varies so much by use case, experience, and resources. The [Spark documentation site](http://spark.apache.org/docs/latest/cluster-overview.html) offers a lot of detail about deploying Spark with actionable examples. We do our best to discuss the most relevant points.

As of this writing, Spark has three officially supported cluster managers:

* Standalone mode
* Hadoop YARN
* Apache Mesos

These cluster managers maintain a set of machines onto which you can deploy Spark Applications. Naturally, each of these cluster managers has an opinionated view toward management, and so there are trade-offs and semantics that you will need to keep in mind. However, they all run Spark applications the same way (as covered in [Chapter 16](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch16.html#s4c1---spark-applications)). Let’s begin with the first point: where to deploy your cluster.

**Where to Deploy Your Cluster to Run Spark Applications**

There are two high-level options for where to deploy Spark clusters: deploy in an on-premises cluster or in the public cloud. This choice is consequential and is therefore worth discussing.

**On-Premises Cluster Deployments**

Deploying Spark to an on-premises cluster is sometimes a reasonable option, especially for organizations that already manage their own datacenters. As with everything else, there are trade-offs to this approach. An on-premises cluster gives you full control over the hardware used, meaning you can optimize performance for your specific workload. However, it also introduces some challenges, especially when it comes to data analytics workloads like Spark. First, with on-premises deployment, your cluster is fixed in size, whereas the resource demands of data analytics workloads are often elastic. If you make your cluster too small, it will be hard to launch the occasional very large analytics query or training job for a new machine learning model, whereas if you make it large, you will have resources sitting idle. Second, for on-premises clusters, you need to select and operate your own storage system, such as a Hadoop file system or scalable key-value store. This includes setting up georeplication and disaster recovery if required.

If you are going to deploy on-premises, the best way to combat the resource utilization problem is to use a cluster manager that allows you to run many Spark applications and dynamically reassign resources between them, or even allows non-Spark applications on the same cluster. All of Spark’s supported cluster managers allow multiple concurrent applications, but YARN and Mesos have better support for dynamic sharing and also additionally support non-Spark workloads. Handling resource sharing is likely going to be the biggest difference your users see day to day with Spark on-premise versus in the cloud: in public clouds, it’s easy to give each application its own cluster of exactly the required size for just the duration of that job.

For storage, you have several different options, but covering all the trade-offs and operational details in depth would probably require its own book. The most common storage systems used for Spark are distributed file systems such as Hadoop’s HDFS and key-value stores such as Apache Cassandra. Streaming message bus systems such as Apache Kafka are also often used for ingesting data. All these systems have varying degrees of support for management, backup, and georeplication, sometimes built into the system and sometimes only through third-party commercial tools. Before choosing a storage option, we recommend evaluating the performance of its Spark connector and evaluating the available management tools.

**Spark in the Cloud**

While early big data systems were designed for on-premises deployment, the cloud is now an increasingly common platform for deploying Spark. The public cloud has several advantages when it comes to big data workloads. First, resources can be launched and shut down elastically, so you can run that occasional “monster” job that takes hundreds of machines for a few hours without having to pay for them all the time. Even for normal operation, you can choose a different type of machine and cluster size for each application to optimize its cost performance—for example, launch machines with Graphics Processing Units (GPUs) just for your deep learning jobs. Second, public clouds include low-cost, georeplicated storage that makes it easier to manage large amounts of data.

Many companies looking to migrate to the cloud imagine they’ll run their applications in the same way that they run their on-premises clusters. All the major cloud providers (Amazon Web Services [AWS], Microsoft Azure, Google Cloud Platform [GCP], and IBM Bluemix) include managed Hadoop clusters for their customers, which provide HDFS for storage as well as Apache Spark. This is actually *not* a great way to run Spark in the cloud, however, because by using a fixed-size cluster and file system, you are not going to be able to take advantage of elasticity. Instead, it is generally a better idea to use global storage systems that are decoupled from a specific cluster, such as Amazon S3, Azure Blob Storage, or Google Cloud Storage and spin up machines dynamically for each Spark workload. With decoupled compute and storage, you will be able to pay for computing resources only when needed, scale them up dynamically, and mix different hardware types. Basically, keep in mind that running Spark in the cloud need not mean migrating an on-premises installation to virtual machines: you can run Spark natively against cloud storage to take full advantage of the cloud’s elasticity, cost-saving benefit, and management tools without having to manage an on-premise computing stack within your cloud environment.

Several companies provide “cloud-native” Spark-based services, and all installations of Apache Spark can of course connect to cloud storage. Databricks, the company started by the Spark team from UC Berkeley, is one example of a service provider built specifically for Spark in the cloud. Databricks provides a simple way to run Spark workloads without the heavy baggage of a Hadoop installation. The company provides a number of features for running Spark more efficiently in the cloud, such as auto-scaling, auto-termination of clusters, and optimized connectors to cloud storage, as well as a collaborative environment for working on notebooks and standalone jobs. The company also provides a [free Community Edition](https://databricks.com/try-databricks) for learning Spark where you can run notebooks on a small cluster and share them live with others. A fun fact is that this *entire book* was written using the free Community Edition of Databricks, because we found the integrated Spark notebooks, live collaboration, and cluster management the easiest way to produce and test this content.

If you run Spark in the cloud, much of the content in this chapter might not be relevant because you can often create a separate, short-lived Spark cluster for each job you execute. In that case, the standalone cluster manager is likely the easiest to use. However, you may still want to read this content if you’d like to share a longer-lived cluster among many applications, or to install Spark on virtual machines yourself.

**Cluster Managers**

Unless you are using a high-level managed service, you will have to decide on the cluster manager to use for Spark. Spark supports three aforementioned cluster managers: standalone clusters, Hadoop YARN, and Mesos. Let’s review each of these.

**Standalone Mode**

Spark’s standalone cluster manager is a lightweight platform built specifically for Apache Spark workloads. Using it, you can run multiple Spark Applications on the same cluster. It also provides simple interfaces for doing so but can scale to large Spark workloads. The main disadvantage of the standalone mode is that it’s more limited than the other cluster managers—in particular, your cluster can *only* run Spark. It’s probably the best starting point if you just want to quickly get Spark running on a cluster, however, and you do not have experience using YARN or Mesos.

**STARTING A STANDALONE CLUSTER**

Starting a standalone cluster requires provisioning the machines for doing so. That means starting them up, ensuring that they can talk to one another over the network, and getting the version of Spark you would like to run on those sets of machines. After that, there are two ways to start the cluster: by hand or using built-in launch scripts.

Let’s first launch a cluster by hand. The first step is to start the master process on the machine that we want that to run on, using the following command:

$SPARK\_HOME/sbin/start-master.sh

When we run this command, the cluster manager master process will start up on that machine. Once started, the master prints out a spark://HOST:PORT URI. You use this when you start each of the worker nodes of the cluster, and you can use it as the master argument to your SparkSession on application initialization. You can also find this URI on the master’s web UI, which is *[http://master-ip-address:8080](http://master-ip-address:8080/)* by default. With that URI, start the worker nodes by logging in to each machine and running the following script using the URI you just received from the master node. The master machine must be available on the network of the worker nodes you are using, and the port must be open on the master node, as well:

$SPARK\_HOME/sbin/start-slave.sh <master-spark-URI>

As soon as you’ve run that on another machine, you have a Spark cluster running! This process is naturally a bit manual; thankfully there are scripts that can help to automate this process.

**CLUSTER LAUNCH SCRIPTS**

You can configure cluster launch scripts that can automate the launch of standalone clusters. To do this, create a file called *conf/slaves* in your Spark directory that will contain the hostnames of all the machines on which you intend to start Spark workers, one per line. If this file does not exist, everything will launch locally. When you go to actually start the cluster, the master machine will access each of the worker machines via Secure Shell (SSH). By default, SSH is run in parallel and requires that you configure password-less (using a private key) access. If you do not have a password-less setup, you can set the environment variable SPARK\_SSH\_FOREGROUND and serially provide a password for each worker.

After you set up this file, you can launch or stop your cluster by using the following shell scripts, based on Hadoop’s deploy scripts, and available in $SPARK\_HOME/sbin:

$SPARK\_HOME/sbin/start-master.sh

Starts a master instance on the machine on which the script is executed.

$SPARK\_HOME/sbin/start-slaves.sh

Starts a slave instance on each machine specified in the *conf/slaves* file.

$SPARK\_HOME/sbin/start-slave.sh

Starts a slave instance on the machine on which the script is executed.

$SPARK\_HOME/sbin/start-all.sh

Starts both a master and a number of slaves as described earlier.

$SPARK\_HOME/sbin/stop-master.sh

Stops the master that was started via the *bin/start-master.sh* script.

$SPARK\_HOME/sbin/stop-slaves.sh

Stops all slave instances on the machines specified in the *conf/slaves* file.

$SPARK\_HOME/sbin/stop-all.sh

Stops both the master and the slaves as described earlier.

**STANDALONE CLUSTER CONFIGURATIONS**

Standalone clusters have a number of configurations that you can use to tune your application. These control everything from what happens to old files on each worker for terminated applications to the worker’s core and memory resources. These are controlled via environment variables or via application properties. Due to space limitations, we cannot include the entire configuration set here. Refer to the relevant table on [Standalone Environment Variables](http://spark.apache.org/docs/latest/spark-standalone.html#cluster-launch-scripts) in [the Spark documentation](http://spark.apache.org/docs/latest/index.html).

**SUBMITTING APPLICATIONS**

After you create the cluster, you can submit applications to it using the spark:// URI of the master. You can do this either on the master node itself or another machine using spark-submit. There are some specific command-line arguments for standalone mode, which we covered in [“Launching Applications”](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch16.html#running-applications).

**Spark on YARN**

Hadoop YARN is a framework for job scheduling and cluster resource management. Even though Spark is often (mis)classified as a part of the “Hadoop Ecosystem,” in reality, Spark has little to do with Hadoop. Spark does natively support the Hadoop YARN cluster manager but it requires nothing from Hadoop itself.

You can run your Spark jobs on Hadoop YARN by specifying the master as YARN in the spark-submit command-line arguments. Just like with standalone mode, there are a number of knobs that you are able to tune according to what you would like the cluster to do. The number of knobs is naturally larger than that of Spark’s standalone mode because Hadoop YARN is a generic scheduler for a large number of different execution frameworks.

Setting up a YARN cluster is beyond the scope of this book, but there are some [great books](http://shop.oreilly.com/product/0636920033448.do) on the topic as well as managed services that can simplify this experience.

**SUBMITTING APPLICATIONS**

When submitting applications to YARN, the core difference from other deployments is that --master will become yarn as opposed the master node IP, as it is in standalone mode. Instead, Spark will find the YARN configuration files using the environment variable HADOOP\_CONF\_DIR or YARN\_CONF\_DIR. Once you have set those environment variables to your Hadoop installation’s configuration directory, you can just run spark-submit like we saw in [Chapter 16](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch16.html#s4c1---spark-applications).

**NOTE**

There are two deployment modes that you can use to launch Spark on YARN. As discussed in previous chapters, cluster mode has the spark driver as a process managed by the YARN cluster, and the client can exit after creating the application. In client mode, the driver will run in the client process and therefore YARN will be responsible only for granting executor resources to the application, not maintaining the master node. Also of note is that in cluster mode, Spark doesn’t necessarily run on the same machine on which you’re executing. Therefore libraries and external jars must be distributed manually or through the --jars command-line argument.

There are a few YARN-specific properties that you can set by using spark-submit. These allow you to control priority queues and things like keytabs for security. We covered these in [“Launching Applications”](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch16.html#running-applications) in [Chapter 16](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch16.html#s4c1---spark-applications).

**Configuring Spark on YARN Applications**

Deploying Spark as YARN applications requires you to understand the variety of different configurations and their implications for your Spark applications. This section covers some best practices for basic configurations and includes references to some of the important configuration for running your Spark applications.

**HADOOP CONFIGURATIONS**

If you plan to read and write from HDFS using Spark, you need to include two Hadoop configuration files on Spark’s classpath: *hdfs-site.xml*, which provides default behaviors for the HDFS client; and *core-site.xml*, which sets the default file system name. The location of these configuration files varies across Hadoop versions, but a common location is inside of */etc/hadoop/conf*. Some tools create these configurations on the fly, as well, so it’s important to understand how your managed service might be deploying these, as well.

To make these files visible to Spark, set HADOOP\_CONF\_DIR in *$SPARK\_HOME/spark-env.sh* to a location containing the configuration files or as an environment variable when you go to spark-submit your application.

**APPLICATION PROPERTIES FOR YARN**

There are a number of Hadoop-related configurations and things that come up that largely don’t have much to do with Spark, just running or securing YARN in a way that influences how Spark runs. Due to space limitations, we cannot include the configuration set here. Refer to the relevant table on [YARN Configurations](http://spark.apache.org/docs/latest/running-on-yarn.html#configuration) in [the Spark documentation](http://spark.apache.org/docs/latest/index.html).

**Spark on Mesos**

Apache Mesos is another clustering system that Spark can run on. A fun fact about Mesos is that the project was also started by many of the original authors of Spark, including one of the authors of this book. In the Mesos project’s own words:

*Apache Mesos abstracts CPU, memory, storage, and other compute resources away from machines (physical or virtual), enabling fault-tolerant and elastic distributed systems to easily be built and run effectively.*

For the most part, Mesos intends to be a datacenter scale-cluster manager that manages not just short-lived applications like Spark, but long-running applications like web applications or other resource interfaces. Mesos is the heaviest-weight cluster manager, simply because you might choose this cluster manager only if your organization already has a large-scale deployment of Mesos, but it makes for a good cluster manager nonetheless.

Mesos is a large piece of infrastructure, and unfortunately there’s simply too much information for us to cover how to deploy and maintain Mesos clusters. There are many great books on the subject for that, including Dipa Dubhashi and Akhil Das’s *Mastering Mesos* (O’Reilly, 2016). The goal here is to bring up some of the considerations that you’ll need to think about when running Spark Applications on Mesos.

For instance, one common thing you will hear about Spark on Mesos is fine-grained versus coarse-grained mode. Historically Mesos supported a variety of different modes (fine-grained and coarse-grained), but at this point, it supports only coarse-grained scheduling (fine-grained has been deprecated). Coarse-grained mode means that each Spark executor runs as a single Mesos task. Spark executors are sized according to the following application properties:

* spark.executor.memory
* spark.executor.cores
* spark.cores.max/spark.executor.cores

**SUBMITTING APPLICATIONS**

Submitting applications to a Mesos cluster is similar to doing so for Spark’s other cluster managers. For the most part you should favor cluster mode when using Mesos. Client mode requires some extra configuration on your part, especially with regard to distributing resources around the cluster.

For instance, in client mode, the driver needs extra configuration information in *spark-env.sh* to work with Mesos.

In *spark-env.sh* set some environment variables:

export MESOS\_NATIVE\_JAVA\_LIBRARY=<path to libmesos.so>

This path is typically *<prefix>/lib/libmesos.so* where the prefix is */usr/local* by default. On Mac OS X, the library is called *libmesos.dylib* instead of *libmesos.so*:

export SPARK\_EXECUTOR\_URI=<URL of spark-2.2.0.tar.gz uploaded above>

Finally, set the Spark Application property spark.executor.uri to <URL of spark-2.2.0.tar.gz>. Now, when starting a Spark application against the cluster, pass a mesos:// URL as the master when creating a SparkContex, and set that property as a parameter in your SparkConf variable or the initialization of a SparkSession:

*// in Scala*

**import** **org.apache.spark.sql.SparkSession**

**val** spark **=** **SparkSession**.builder

.master("mesos://HOST:5050")

.appName("my app")

.config("spark.executor.uri", "<path to spark-2.2.0.tar.gz uploaded above>")

.getOrCreate()

Submitting cluster mode applications is fairly straightforward and follows the same spark-submit structure you read about before. We covered these in [“Launching Applications”](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch16.html#running-applications).

**CONFIGURING MESOS**

Just like any other cluster manager, there are a number of ways that we can configure our Spark Applications when they’re running on Mesos. Due to space limitations, we cannot include the entire configuration set here. Refer to the relevant table on [Mesos Configurations](http://bit.ly/2DPmLTf) in [the Spark documentation](http://bit.ly/1qnQ26w).

**Secure Deployment Configurations**

Spark also provides some low-level ability to make your applications run more securely, especially in untrusted environments. Note that the majority of this setup will happen outside of Spark. These configurations are primarily network-based to help Spark run in a more secure manner. This means authentication, network encryption, and setting TLS and SSL configurations. Due to space limitations, we cannot include the entire configuration set here. Refer to the relevant table on [Security Configurations](http://bit.ly/2DJ0BTp) in [the Spark documentation](http://bit.ly/1qnQ26w).

**Cluster Networking Configurations**

Just as shuffles are important, there can be some things worth tuning on the network. This can also be helpful when performing custom deployment configurations for your Spark clusters when you need to use proxies in between certain nodes. If you’re looking to increase Spark’s performance, these should not be the first configurations you go to tune, but may come up in custom deployment scenarios. Due to space limitations, we cannot include the entire configuration set here. Refer to the relevant table on [Networking Configurations](http://bit.ly/2DGfT7v) in [the Spark documentation](http://bit.ly/1qnQ26w).

**Application Scheduling**

Spark has several facilities for scheduling resources between computations. First, recall that, as described earlier in the book, each Spark Application runs an independent set of executor processes. Cluster managers provide the facilities for scheduling across Spark applications. Second, within each Spark application, multiple jobs (i.e., Spark actions) may be running concurrently if they were submitted by different threads. This is common if your application is serving requests over the network. Spark includes a *fair scheduler* to schedule resources within each application. We introduced this topic in the previous chapter.

If multiple users need to share your cluster and run different Spark Applications, there are different options to manage allocation, depending on the cluster manager. The simplest option, available on all cluster managers, is static partitioning of resources. With this approach, each application is given a maximum amount of resources that it can use, and holds onto those resources for the entire duration. In spark-submit there are a number of properties that you can set to control the resource allocation of a particular application. Refer to [Chapter 16](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch16.html#s4c1---spark-applications) for more information. In addition, *dynamic allocation* (described next) can be turned on to let applications scale up and down dynamically based on their current number of pending tasks. If, instead, you want users to be able to share memory and executor resources in a fine-grained manner, you can launch a single Spark Application and use thread scheduling within it to serve multiple requests in parallel.

**DYNAMIC ALLOCATION**

If you would like to run multiple Spark Applications on the same cluster, Spark provides a mechanism to dynamically adjust the resources your application occupies based on the workload. This means that your application can give resources back to the cluster if they are no longer used, and request them again later when there is demand. This feature is particularly useful if multiple applications share resources in your Spark cluster.

This feature is disabled by default and available on all coarse-grained cluster managers; that is, standalone mode, YARN mode, and Mesos coarse-grained mode. There are two requirements for using this feature. First, your application must set spark.dynamicAllocation.enabled to true. Second, you must set up an external shuffle service on each worker node in the same cluster and set spark.shuffle.service.enabled to true in your application. The purpose of the external shuffle service is to allow executors to be removed without deleting shuffle files written by them. This is set up differently for each cluster manager and is described in [the job scheduling configuration](http://bit.ly/2DQ3ocB). Due to space limitations, we cannot include the configuration set for dynamic allocation. Refer to the relevant table on [Dynamic Allocation Configurations](http://bit.ly/2ne8jL3).

**Miscellaneous Considerations**

There several other topics to consider when deploying Spark applications that may affect your choice of cluster manager and its setup. These are just things that you should think about when comparing different deployment options.

One of the more important considerations is the number and type of applications you intend to be running. For instance, YARN is great for HDFS-based applications but is not commonly used for much else. Additionally, it’s not well designed to support the cloud, because it expects information to be available on HDFS. Also, compute and storage is largely coupled together, meaning that scaling your cluster involves scaling both storage and compute instead of just one or the other. Mesos does improve on this a bit conceptually, and it supports a wide range of application types, but it still requires pre-provisioning machines and, in some sense, requires buy-in at a much larger scale. For instance, it doesn’t really make sense to have a Mesos cluster for only running Spark Applications. Spark standalone mode is the lightest-weight cluster manager and is relatively simple to understand and take advantage of, but then you’re going to be building more application management infrastructure that you could get much more easily by using YARN or Mesos.

Another challenge is managing different Spark versions. Your hands are largely tied if you want to try to run a variety of different applications running different Spark versions, and unless you use a well-managed service, you’re going to need to spend a fair amount of time either managing different setup scripts for different Spark services or removing the ability for your users to use a variety of different Spark applications.

Regardless of the cluster manager that you choose, you’re going to want to consider how you’re going to set up logging, store logs for future reference, and allow end users to debug their applications. These are more “out of the box” for YARN or Mesos and might need some tweaking if you’re using standalone.

One thing you might want to consider—or that might influence your decision making—is maintaining a metastore in order to maintain metadata about your stored datasets, such as a table catalog. We saw how this comes up in Spark SQL when we are creating and maintaining tables. Maintaining an Apache Hive metastore, a topic beyond the scope of this book, might be something that’s worth doing to facilitate more productive, cross-application referencing to the same datasets.

Depending on your workload, it might be worth considering using Spark’s external shuffle service. Typically Spark stores shuffle blocks (shuffle output) on a local disk on that particular node. An external shuffle service allows for storing those shuffle blocks so that they are available to all executors, meaning that you can arbitrarily kill executors and still have their shuffle outputs available to other applications.

Finally, you’re going to need to configure at least some basic monitoring solution and help users debug their Spark jobs running on their clusters. This is going to vary across cluster management options and we touch on some of the things that you might want to set up in [Chapter 18](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch18.html#s4c3---monitoring-and-debugging).

**Conclusion**

This chapter looked at the world of configuration options that you have when choosing how to deploy Spark. Although most of the information is irrelevant to the majority of users, it is worth mentioning if you’re performing more advanced use cases. It might seem fallacious, but there are other configurations that we have omitted that control even lower-level behavior. You can find these in the Spark documentation or in the Spark source code. [Chapter 18](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch18.html#s4c3---monitoring-and-debugging) talks about some of the options that we have when monitoring Spark Applications.