CHAPTER
1

Finishing Your Spark Job

COPYRIGHTED MATERIAL

When you scale out a Spark application for the first time, one of the more common occurrences you will encounter is the application's inability to merely succeed and finish its job. The Apache Spark framework's ability to scale is tremendous, but it does not come out of the box with those properties. Spark was created, first and foremost, to be a framework that would be easy to get started and use. Once you have developed an initial application, however, you will then need to take the additional exercise of gaining deeper knowledge of Spark's internals and configurations to take the job to the next stage.

In this chapter we lay the groundwork for getting a Spark application to succeed. We will focus primarily on the hardware and system-level design choices you need to set up and consider before you can work through the various Spark-specific issues to move an application into production.

We will begin by discussing the various ways you can install a production-grade cluster for Apache Spark. We will include the scaling efficiencies you will need depending on a given workload, the various installation methods, and the common setups. Next, we will take a look at the historical origins of Spark in order to better understand its design and to allow you to best judge when it is the right tool for your jobs. Following that, we will take a look at resource management: how memory, CPU, and disk usage come into play when creating and executing Spark applications. Next, we will cover storage capabilities within Spark and their external subsystems. Finally, we will conclude with a discussion of how to instrument and monitor a Spark application.

Installation of the Necessary Components

Before you can begin to migrate an application written in Apache Spark you will need an actual cluster to begin testing it on. You can download, compile, and install Spark in a number of different ways within its system (some will be easier than others), and we'll cover the primary methods in this chapter.

Let's begin by explaining how to configure a *native* installation, meaning one where *only* Apache Spark is installed, then we'll move into the various Hadoop distributions (Cloudera and Hortonworks), and conclude by providing a brief explanation on how to deploy Spark on Amazon Web Services (AWS).

Before diving too far into the various ways you can install Spark, the obvious question that arises is, "What type of hardware should I leverage for a Spark cluster?" We can offer various possible answers to this question, but we'd like to focus on a few resounding truths of the Spark framework rather than necessitating a given layout.

It's important to know that Apache Spark is an *in-memory* compute grid. Therefore, for maximum efficiency, it is highly recommended that the system, as a whole, maintain enough memory *within the framework* for the largest workload (or dataset) that will be conceivably consumed. We are not saying that you cannot scale a cluster later, but it is always better to plan ahead, especially if you work inside a larger organization where purchase orders might take weeks or months.

On the concept of memory it is necessary to understand that when computing the amount of memory you need to understand that the computation does not equate to a one-to-one fashion. That is to say, for a given 1TB dataset, you will need *more* than 1TB of memory. This is because when you create objects within Java from a dataset, the object is typically much larger than the original data element. Multiply that expansion times the number of objects created for a given dataset and you will have a much more accurate representation of the amount of memory a system will require to perform a given task.

To better attack this problem, Spark is, at the time of this writing, working on what Apache has called *Project Tungsten*, which will greatly reduce the memory overhead of objects by leveraging off heap memory. You don't need to know more about Tungsten as you continue reading this book, but this information may apply to future Spark releases, because Tungsten is poised to become the de facto memory management system.

The second major component we want to highlight in this chapter is the number of CPU cores you will need per physical machine when you are determining hardware for Apache Spark. This is a much more fragmented answer in that, once the data load normalizes into memory, the application is typically network or CPU bound. That said, the easiest solution is to test your Spark application on a smaller dataset and measure its bounding case, be it either network or CPU, and then plan accordingly from there.

Native Installation Using a Spark Standalone Cluster

The simplest way to install Spark is to deploy a Spark Standalone cluster. In this mode, you deploy a Spark binary to each node in a cluster, update a small set of configuration files, and then start the appropriate processes on the master and slave nodes. In Chapter 2, we discuss this process in detail and present a simple scenario covering installation, deployment, and execution of a basic Spark job.

Because Spark is not tied to the Hadoop ecosystem, this mode does not have any dependencies aside from the Java JDK. Spark currently recommends the Java 1.7 JDK. If you wish to run alongside an existing Hadoop deployment, you can launch the Spark processes on the same machines as the Hadoop installation and configure the Spark environment variables to include the Hadoop configuration.

NOTE **For more on a Cloudera installation of Spark try** `http://www.cloudera.com/content/www/en-us/documentation/enterprise/latest/topics/cdh_ig_spark_installation.html`**. For more on the Hortonworks installation try** `http://hortonworks.com/hadoop/spark/#section_6`**. And for more on an Amazon Web Services installation of Spark try** `http://aws.amazon.com/articles/4926593393724923`.

The History of Distributed Computing That Led to Spark

We have introduced Spark as a distributed compute framework; however, we haven't really discussed what this means. Until recently, most computer systems available to both individuals and enterprises were based around single machines. These single machines came in many shapes and sizes and differed dramatically in terms of their performance, as they do today.

We're all familiar with the modern ecosystem of personal machines. At the low-end, we have tablets and mobile phones. We can think of these as relatively weak, un-networked computers. At the next level we have laptops and desktop computers. These are more powerful machines, with more storage and computational ability, and potentially, with one or more graphics cards (GPUs) that support certain types of massively parallel computations. Next are those machines that some people have networked with in their home, although generally these machines were not networked to share their computational ability, but rather to provide shared storage—for example, to share movies or music across a home network.

Within most enterprises, the picture today is still much the same. Although the machines used may be more powerful, most of the software they run, and most of the work they do, is still executed on a single machine. This fact limits the scale and the potential impact of the work they can do. Given this limitation, a few select organizations have driven the evolution of modern parallel computing to allow networked systems of computers to do more than just share data, and to collaboratively utilize their resources to tackle enormous problems.

In the public domain, you may have heard of the SETI at Home program from Berkeley or the Folding@Home program from Stanford. Both of these programs were early initiatives that let individuals dedicate their machines to solving parts of a massive distributed task. In the former case, SETI has been looking for unusual signals coming from outer space collected via radio telescope. In the latter, the Stanford program runs a piece of a program computing permutations of proteins—essentially building molecules—for medical research.

Because of the size of the data being processed, no single machine, not even the massive supercomputers available in certain universities or government agencies, have had the capacity to solve these problems within the scope of a project or even a lifetime. By distributing the workload to multiple machines, the problem became potentially tractable—solvable in the allotted time.

As these systems became more mature, and the computer science behind these systems was further developed, many organizations created *clusters* of machines—coordinated systems that could distribute the workload of a particular problem across many machines to extend the resources available. These systems first grew in research institutions and government agencies, but quickly moved into the public domain.

Enter the Cloud

The most well-known offering in this space is of course the proverbial "cloud." Amazon introduced AWS (Amazon Web Services), which was later followed by comparable offerings from Google, Microsoft, and others. The purpose of a cloud is to provide users and organizations with scalable clusters of machines that can be started and expanded upon on-demand.

At about the same time, universities and certain companies were also building their own clusters in-house and continuing to develop frameworks that focused on the challenging problem of parallelizing arbitrary types of tasks and computations. Google was born out of its PageRank algorithm—an extension of the MapReduce framework that allowed a general class of problems to be solved in parallel on clusters built with commodity hardware.

This notion of building algorithms, that, while not the most efficient, could be massively parallelized and scaled to thousands of machines, drove the next stage of growth in this area. The idea that you could solve massive problems by building clusters, not of supercomputers, but of relatively weak and inexpensive machines, democratized distributed computing.

Yahoo, in a bid to compete with Google, developed, and later open-sourced under the Apache Foundation, the Hadoop platform—an ecosystem for distributed computing that includes a file system (HDFS), a computation framework

(MapReduce), and a resource manager (YARN). Hadoop made it dramatically easier for any organization to not only create a cluster but to also create software and execute parallelizable programs on these clusters that can process huge amounts of distributed data on multiple machines.

Spark has subsequently evolved as a replacement for MapReduce by building on the idea of creating a framework to simplify the difficult task of writing parallelizable programs that efficiently solve problems at scale. Spark's primary contribution to this space is that it provides a powerful and simple API for performing complex, distributed operations on distributed data. Users can write Spark programs as if they were writing code for a single machine, but under the hood this work is distributed across a cluster. Secondly, Spark leverages the memory of a cluster to reduce MapReduce's dependency on the underlying distributed file system, leading to dramatic performance gains. By virtue of these improvements, Spark has achieved a substantial amount of success and popularity and has brought you here to learn more about how it accomplishes this.

Spark is not the right tool for every job. Because Spark is fundamentally designed around the MapReduce paradigm, its focus is on excelling at Extract, Transform, and Load (ETL) operations. This mode of processing is typically referred to as batch processing—processing large volumes of data efficiently in a distributed manner. The downside of batch processing is that it typically introduces larger latencies for any single piece of data. Although Spark developers have been dedicating a substantial amount of effort to improving the Spark Streaming mode, it remains fundamentally limited to computations on the order of seconds. Thus, for truly low-latency, high-throughput applications, Spark is not necessarily the right tool for the job. For a large set of use cases, Spark nonetheless excels at handling typical ETL workloads and provides substantial performance gains (as much as 100 times improvement) over traditional MapReduce.

Understanding Resource Management

In the chapter on cluster management you will learn more about how the operating system handles the allocation and distribution of resources amongst the processes on a single machine. However, in a distributed environment, the cluster manager handles this challenge. In general, we primarily focus on three types of resources within the Spark ecosystem. These are disk storage, CPU cores, and memory. Other resources exist, of course, such as more advanced abstractions like virtual memory, GPUs, and potentially different tiers of storage, but in general we don't need to focus on those within the context of building Spark applications.

Disk Storage

The first type of resource, disk, is vital to any Spark application since it stores persistent data, the results of intermediate computations, and system state.

When we refer to disk storage, we are referring to data stored on a hard drive of some kind, either the traditional rotating spindle, or newer SSDs and flash memory. Like any other resource, disk is finite. Disk storage is relatively cheap and most systems tend to have an abundance of physical storage, but in the world of big data, it's actually quite common to use up even this cheap and abundant storage! We tend to enable replication of data for the sake of durability and to support more efficient parallel computation. Also, you'll usually want to persist frequently used intermediate dataset(s) to disk to speed up long-running jobs. Thus, it generally pays to be cognizant of disk usage, and treat it as any other finite resource.

Interaction with physical disk storage on a single machine is abstracted away by the file system—a program that provides an API to read and write files. In a distributed environment, where data may be spread across multiple machines, but still needs to be accessed as a single logical entity, a *distributed* file system fulfills the same role. Managing the operation of the *distributed* file system and monitoring its state is typically the role of the cluster administrator, who tracks usage, quotas, and re-assigns resources as necessary. Cluster managers such as YARN or Mesos may also regulate access to the underlying file system to better distribute resources between simultaneously executing applications.

CPU Cores

The central processing unit (CPU) on a machine is the processor that actually executes all computations. Modern machines tend to have multiple CPU cores, meaning that they can execute multiple processes in parallel. In a cluster, we have multiple machines, each with multiple cores. On a single machine, the operating system handles communication and resource sharing between processes. In a distributed environment, the cluster manager handles the assignment of CPU resources (cores) to individual tasks and applications. In the chapter on cluster management, you'll learn specifically how YARN and Mesos ensure that multiple applications running in parallel can have access to this pool of available CPUs and share it fairly.

When building Spark applications, it's helpful to relate the number of CPU cores to the parallelism of your program, or how many tasks it can execute simultaneously. Spark is based around the resilient distributed dataset (RDD)—an abstraction that treats a distributed dataset as a single entity consisting of multiple partitions. In Spark, a single Spark task will processes a single partition of an RDD on a single CPU core.

Thus, the degree to which your data is partitioned—and the number of available cores—essentially dictates the parallelism of your program. If we consider a hypothetical Spark job consisting of five stages, each needing to run 500 tasks, if we only have five CPU cores available, this may take a long time to complete! In contrast, if we have 100 CPU cores available, and the data is sufficiently partitioned, for example into 200 partitions, Spark will be able to parallelize much more effectively, running 100 tasks simultaneously, completing the job much more quickly. By default, Spark only uses two cores with a single executor—thus when launching a Spark job for the first time, it may unexpectedly take a very long time. We discuss executor and core configuration in the next chapter.

Memory

Lastly, memory is absolutely critical to almost all Spark applications. Memory is used for internal Spark mechanisms such as the shuffle, and the JVM heap is used to persist RDDs in memory, minimizing disk I/O and providing dramatic performance gains. Spark acquires memory per executor—a worker abstraction that you'll learn more about in the next chapter. The amount of memory that Spark requests per executor is a configurable parameter and it is the job of the cluster manager to ensure that the requested resources are provided to the requesting application.

Generally, cluster managers assign memory the same way that the cluster manager assigns CPU cores as discrete resources. The total available memory in a cluster is broken up into blocks or containers, and these containers are assigned (or offered in the case of Mesos) to specific applications. In this way, the cluster manager can act to both assign memory fairly, and schedule resource usage to avoid starvation.

Each assigned block of memory in Spark is further subdivided based on Spark and cluster manager configurations. Spark makes tradeoffs between the memory allocated for dynamic memory allocated during shuffle, the memory used to store cached RDDs, and the amount of memory available for off-heap storage.

Most applications will require some degree of tuning to determine the appropriate balance of memory based on the RDD transformations executed within the Spark program. A Spark application with improperly configured memory settings may run inefficiently, for example, if RDDs cannot be fully persisted in memory and instead are swapped back and forth from disk. Insufficient memory allocated for the shuffle operation can also lead to slowdown since internal tables may be swapped to disk, if they cannot fit entirely into memory.

In the next chapter on cluster management, we will discuss in detail the memory structure of a block of memory allocated to Spark. Later, when we cover performance tuning, we'll show how to set the parameters associated with memory to ensure that Spark applications run efficiently and without failures.

In newer versions of Spark, starting with Spark 1.6, Spark introduces dynamic automatic memory tuning. As of 1.6, Spark will automatically adjust the fraction of memory allocated for shuffle and caching, as well as the total amount of allocated memory. This allows you to fit larger datasets into a smaller amount of memory, as well as to more easily create programs that execute successfully out of the box, without extensive tuning of a multitude of memory parameters.

Using Various Formats for Storage

When solving a distributed processing problem sometimes we get tempted to focus more on the solution, on how to get the best from the cluster resources, or on how to improve the code to be more efficient. All of these things are great but they are not all we can do to improve the performance of our application.

Sometimes, the way we choose to store the data we are processing, highly impacts the execution. This subchapter proposes to bring some light on how to decide which file format to choose when storing data.

There are several aspects we must consider when loading or storing data with Spark: What is the most suitable file format to choose? Is the file format splittable? Meaning, can splits of this file be processed in parallel? Do we compress the data and if so, which compression codec to use? How large should our files be?

The first thing you should be careful of is the file sizes your dataset is divided into. Even if in Chapter 3 you will read about parallelism and how it affects the performance of your application, it is important to mention how the file sizes determine the level of parallelism. As you already might know, on HDFS each file is stored in blocks. When reading these files with Spark, each HDFS block will be mapped to one Spark partition. For each partition, a Spark task will be launched to read and process it. A high level of parallelism is usually beneficial if you have the necessary resources and if the data is properly partitioned. However, a very large number of tasks come with a scheduling overhead that should be avoided if it is not necessary. In conclusion, the size of the files we are reading causes a proportional number of tasks to be launched and a significant scheduling overhead.

Besides the large number of tasks that are launched, reading a lot of small files also brings a serious time penalty inflicted by opening them. You should also consider the fact that all the file paths are handled on the driver. So if your data consists of a huge amount of small files, then you risk placing memory pressure on the driver.

On the other hand, if the dataset is composed of a set of huge files, then you must make sure the files are splittable. Otherwise, they will have to be handled by single tasks resulting in very large partitions. This will highly decrease performance.

Most of the time, saving space is important. So, to minimize the data's disk footprint, we compress it. If we plan to process this data later on with Spark, we have to be careful which compression format we choose. It is important to know if it is splittable or not. Let's imagine we have a 5 GB file stored on HDFS with a block size of 128 MB. The file will be composed of 40 blocks. When we read it with Spark, a task will be launched for each block, so there will be 40 parallel tasks that will process the data. If this file would be a compressed file in gzip format, then it is not supported to decompress a block independently from the other blocks. This means that Spark is not able to process each block in parallel, so only one task will process the entire file. It is obvious that the performance is highly impacted and we might even face memory issues.

There are many compression codecs having different features and advantages. When choosing between them we trade off between compression ratio and speed. The most common ones are gzip, bzip2, lzo, lz4, and Snappy.

- Gzip is a compression codec that uses the DEFLATE algorithm. It is a wrapper around the Zlib compression format having the advantage of a good compression ratio.
- Bzip2 compression format uses the burrows wheeler transform algorithm and it is block oriented. This codec has a higher compression ratio than gzip.
- There are also the LZO and the LZ4 block oriented compression codecs that both are based on the LZ77 algorithm. They have modest compression ratios but they excel at compression and decompression speeds.

The fastest compression and decompression speed is provided by the Snappy compression codec. It is a block-oriented codec based on the LZ77 algorithm. Because of its decompression speed, it is desirable to use Snappy for datasets that are frequently used.

If we were to separate compression codecs into splittable or not splittable we would refer to Table 1-1. However, making this separation is confusing because it strongly depends on the file format that they are compressing. If the non splittable codecs are used with file formats that support block structure like Sequence files or ORC files, then the compression will be applied for each block. In this case, Spark will be able to launch in parallel tasks for each block. So you might consider them splittable. But, on the other hand, if they are used to compress text files, then the entire file will be compressed in a single block, therefore only one task will be launched per file.

This means that not only the compression codec is important but also the file's storage format. Spark supports a variety of input and output formats, structured or unstructured, starting with text files, sequence files, or any other Hadoop file formats. Is important to underline that making use of the `hadoopRDD` and `newHadoopRDD` methods, you can read in Spark any existent Hadoop file format.

Table 1-1: Splittable Compression Codecs

COMPRESSION CODEC	IS SPLITTABLE
Gzip	No
Bzip2	Yes
LZO	No, unless indexed
Snappy	Yes

Text Files

You can easily read text files with Spark using the `textFile` method. You can either read a single file or all of the files within a folder. Because this method will split the documents into lines, you have to keep the lines at a reasonable size.

As mentioned above, if the files are compressed, depending on the compression codec, they might not be splittable. In this case, they should have sizes small enough to be easily processed within a single task.

There are some special text file formats that must be mentioned: the structured text files. CSV files, JSON files and XML files all belong to this category.

To easily do some analytics over data stored in CSV format you should create a DataFrame on top of it. To do this you have two options: You can either read the files with the classic `textFile` method or programmatically specify the schema, or you could use one of the Databricks packages spark-csv. In the example below, we read a csv file, remove the first line that represents the header, and map each row to a Car object. The resulted RDD is transformed to a DataFrame.

```
import sqlContext.implicits._
case class Pet(name: String, race : String)
val textFileRdd = sc.textFile("file.csv")
val schemaLine = textFileRdd.first()
val noHeaderRdd = textFileRdd.filter(line => ↵
!line.equals(schemaLine))
val petRdd = noHeaderRdd.map(textLine => {
             val columns = textLine.split(",")
             Pet(columns(0), columns(1))})
val petDF = petRdd.toDF()
```

An easier way to process CSV files is to use the spark-csv package from Databricks. You just read the file specifying the csv format:

```
val df = sqlContext.read
    .format("com.databricks.spark.csv")
    .option("header", "true")
    .option("inferSchema", "true")
    .load("file.csv")
```

To read and process JSON files, Spark SQL exposes a dedicated method. You have the possibility to leave Spark SQL to infer the schema from the dataset or you can specify it programmatically. If you know the schema in advance, it is recommended to provide it. This is to avoid making Spark go through the input once more to determine it. Another advantage of providing the schema yourself is that you have the possibility of working only with the fields you need. If you have JSON files with lots of fields that are not in your interest, you can specify only the relevant ones and the other ones will be ignored.

Here is an example of how to read a JSON file with and without specifying the schema of your dataset:

```
val schema = new StructType(Array(
    new StructField("name", StringType, false),
    new StructField("age", IntegerType, false)))
val specifiedSchema= sqlContext.jsonFile("file.json",schema)
val inferedSchema = sqlContext.jsonFile("file.json")
```

This way of handling JSON files assumes that you have a JSON object per line. If there are some JSON objects that miss several fields then the fields are replaced with nulls. In the case when we infer the schema and there are malformed inputs, Spark SQL creates a new column called _corrupt_record. The erroneous inputs will have this column populated with their data and will have all the other columns null.

The XML file formats are not an ideal format for distributed processing because they usually are very verbose and don't have an XML object per line. Because of this they cannot be processed in parallel. Spark doesn't have for now a built-in library for processing these files. If you try to read an XML file with the `textFile` method it is not useful because Spark will read the file line by line. If your XML files are small enough to fit in memory, then you could read them using the `wholeTextFile` method. This will output a pair RDD that will have the file's path as key and the entire text file as value. Processing large files in this manner is allowed but it might cause a bad performance.

Sequence Files

Sequence files are a commonly used file format, consisting of binary key value pairs that must be subclasses of the Hadoop Writable interface. They are very popular in distributed processing because they have sync markers. This allows you to identify record boundaries, thus making it possible to parallelize the process. Sequence files are an efficient way of storing your data because they can be efficiently processed compressed or uncompressed.

Spark offers a dedicated API for loading sequence files:

```
val seqRdd = sc.sequenceFile("filePath", classOf[Int], classOf[String])
```

Avro Files

The avro file format is a binary data format that relies on a schema. When storing data into an avro format, the schema is always stored with the data. This feature makes possible for files in avro file format to be read from different applications.

There is a Spark package to read/write avro files: spark-avro (`https://github.com/databricks/spark-avro`). This package handles the schema conversion from avro schema to the Spark SQL schema. To load an avro file is pretty straight forward: You have to include the spark-avro package and then you read the file as follows:

```
import com.databricks.spark.avro._
val avroDF = sqlContext.read.avro("pathToAvroFile")
```

Parquet Files

Parquet file format is a columnar file format that supports nested data structures. Being in a columnar format makes it very good for aggregation queries, because only the required columns are read from disk. Parquet files support really efficient compression and encoding schemes, since they can be specified per-column. This being said, it is clear why using this file format gives you the advantage of decreasing the disk IO operations and saving more storage space.

Spark SQL provides methods for reading and writing Parquet files maintaining the data's schema. This file format supports schema evolution. One can start with some columns and then add more columns. These schema differences are automatically detected and merged. However if you can, you should avoid schema merging, because it is an expensive operation. Below is an example of how to read a parquet file, having the schema merging enabled:

```
val parquetDF = sqlContext.read
              .option("mergeSchema","true")
              .parquet("parquetFolder")
```

In Spark SQL, the Parquet Datasource is able to detect if data is partitioned and to determine the partitions. This is an important optimization in data analysis because during a query, only the needed partitions are scanned based on the predicates inside the query. In the example below, only the folder for company A will be scanned in order to serve the requested employees.

```
Folder/company=A/file1.parquet
Folder/company=B/fileX.parquet

SELECT employees FROM myTable WHERE company=A
```

The Parquet file format is encouraged as a best practice for Spark SQL.

Making Sense of Monitoring and Instrumentation

One of the most important things when running a distributing application is monitoring. You want to identify as soon as possible anomalies and to troubleshoot them. You want to analyze the application's behavior so you can determine how to improve its performance. Knowing how your application uses the cluster resources and how the load is distributed might make you gain some important insights and save you a lot of time and money.

The purpose of this section is to identify the monitoring options we have and what we learn from the metrics we inspect.

Spark UI

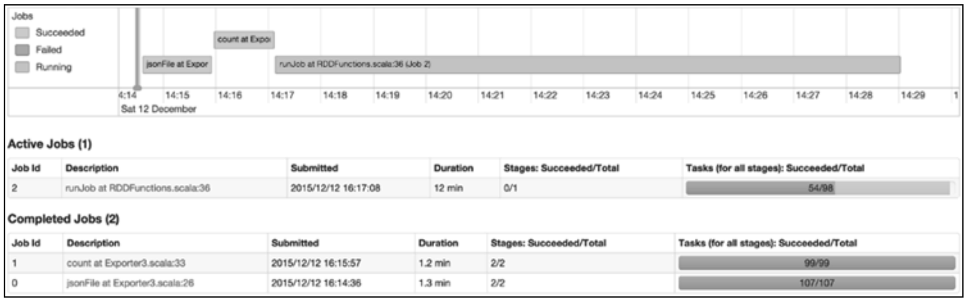
Spark comes with a built-in UI that exposes useful information and metrics about the application you are running. When you launch a Spark application, a web user interface is launched, having the default port set on 4040. If there are multiple Spark drivers running on the node, then an exception will be displayed reporting the fact that the 4040 port is unavailable. In this case, the web UI will try to bind to the next ports starting with 4040: 4041, 4042 until an available one is found.

To access the Spark UI for your application, you will open the following page in your web browser: `http://<driver-node-ip>:<allocatedPort-default4040>.`

The default behavior is to provide access to the job execution metrics only during the execution of your application. So, you will be able to see the Spark UI as long as the application is still running. To continue seeing this information in the UI even after the process finishes, you can change the default behavior by setting the `spark.eventLog.enabled` to true.

This feature is really useful, because you can understand better the behavior of your Spark application. In this web user interface you can see information such as:

- In the Jobs tab you can see the list of jobs that were executed and the job that is still in progress with their execution timeline. It displays how many stages and tasks were successful from the total number and information about the duration of each job (see Figure 1-1).



Active Jobs (1)

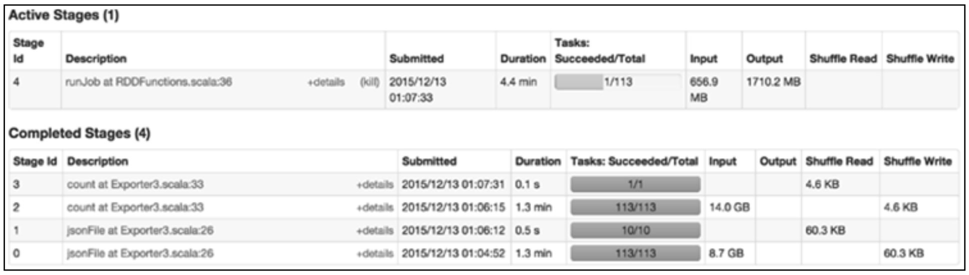
Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	runJob at RDDFunctions.scala:36	2015/12/12 16:17:08	12 min	0/1	54/98

Completed Jobs (2)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	count at Exporter3.scala:33	2015/12/12 16:15:57	1.2 min	2/2	99/99
0	jsonFile at Exporter3.scala:26	2015/12/12 16:14:36	1.3 min	2/2	107/107

Figure 1-1: The Spark UI showing job progress

- In the Stages tab you can see the list of stages that were executed and the one that is still active for all of the jobs (see Figure 1-2). This page offers relevant information about how your data is being processed: You can see the amount of data that is received as an input and its size as an output. Also, here you can see the amount of data that is being shuffled. This information is valuable since it might signal that you are not using the right operators for processing your data or that you might need to partition your data. In Chapter 3 there are more details about the shuffle phase and how it impacts the performance of your Spark application.



Active Stages (1)

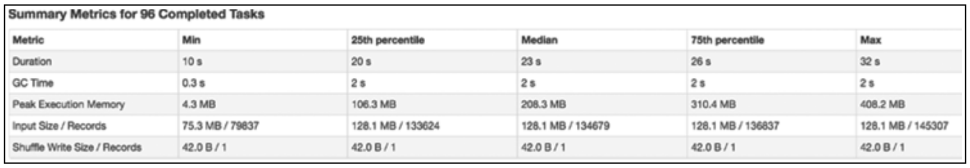
Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
4	runJob at RDDFunctions.scala:36	+details (kill)	2015/12/13 01:07:33	4.4 min	1/113	656.9 MB	1710.2 MB		

Completed Stages (4)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
3	count at Exporter3.scala:33	+details	2015/12/13 01:07:31	0.1 s	1/1			4.6 KB	
2	count at Exporter3.scala:33	+details	2015/12/13 01:06:15	1.3 min	113/113	14.0 GB			4.6 KB
1	jsonFile at Exporter3.scala:26	+details	2015/12/13 01:06:12	0.5 s	10/10			60.3 KB	
0	jsonFile at Exporter3.scala:26	+details	2015/12/13 01:04:52	1.3 min	113/113	8.7 GB			60.3 KB

Figure 1-2: Spark UI job execution information

- In the task metrics stage, you can analyze metrics about the tasks that were executed. You can see reports about their duration, about garbage collection, memory, and the size of the data that is being processed (see Figure 1-3). The information about the duration of the running tasks might signal that your data is not uniformly distributed. If the maximum task duration is a lot larger than the medium duration it means that you have a task on which the load is much higher than on the others.



Summary Metrics for 96 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	10 s	20 s	23 s	26 s	32 s
GC Time	0.3 s	2 s	2 s	2 s	2 s
Peak Execution Memory	4.3 MB	106.3 MB	208.3 MB	310.4 MB	408.2 MB
Input Size / Records	75.3 MB / 79837	128.1 MB / 133624	128.1 MB / 134679	128.1 MB / 136837	128.1 MB / 145307
Shuffle Write Size / Records	42.0 B / 1	42.0 B / 1	42.0 B / 1	42.0 B / 1	42.0 B / 1

Figure 1-3: Spark UI task metrics

- The DAG schedules stages for a certain job (see Figure 1-4). This information is important for you to understand the way your job is scheduled for running. You can identify the operations that trigger shuffles and are stage boundaries. Chapter 3 goes into more detail about the Spark Execution Engine.

- Information about the execution environment: In the Environment tab you can see all the configuration parameters used when starting your Spark context and the JARs used.
- Logs gathered from each executor are also important.

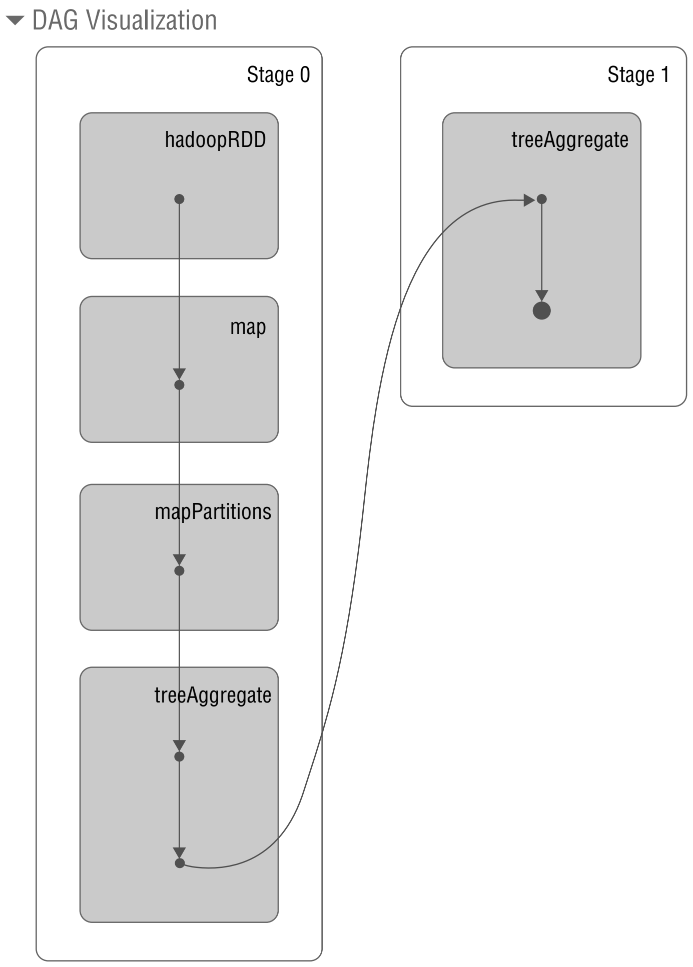


Figure 1-4: The DAG stage scheduling

Spark Standalone UI

When running Spark in standalone mode, you have another build in the web user interface that exposes information about clusters and about the executed jobs and detailed logs. You can access this UI at the following address: `http://<master-ip>:<defaultPort: 8080>`.

If you are running Spark on top of YARN or Mesos cluster managers, you can start a history server that allows you to see the UI for applications that finished executing. To start the server use the following command: `./sbin/start-history-server.sh`.

The history server is available at the following address: `http://<server-url>:18080`.

Metrics REST API

Spark also provides REST APIs for retrieving metrics about your application for you to use programmatically or to build your own visualizations based on them. The information is provided in JSON format for running applications and for apps from history.

The API endpoints are :

```
http://<server-url>:18080/api/v1
http://<driver-node-ip>:<allocatedPort-default4040>
/api/v1
```

You can find more information about the available APIs at `http://spark.apache.org/docs/latest/monitoring.html#rest-api`.

Metrics System

A useful Spark component is the Metrics System. This system is available on the driver and on each executor and can expose information about several Spark components to different syncs. In this way you can obtain metrics sent by the master process, by your application, by worker processes, and by the driver and executors.

Spark offers the freedom to monitor your application using a different set of third-party tools using this Metrics System.

External Monitoring Tools

There are several external Spark monitoring applications used for profiling. A widely used open source tool for displaying time series data is Graphite. The Spark Metrics System has a built-in Graphite sink that sends metrics about your application to a Graphite node.

You could also use Ganglia, a scalable distributed monitoring system to keep an eye on your application. Among other metrics' syncs, Spark supports a Ganglia sync that sends the metrics to a Ganglia node or to a multicast group. Because of licensing reasons this sync is not included in the default Spark build.

Another performance monitoring tool for Spark is SPM. This tool collects all the metrics from your Spark application and provides monitoring charts.

Summary

In this chapter we detailed the ways that you can install a production-grade cluster for Apache Spark. We also covered a bit about scaling efficiencies, along with installation and setup. You should now have a good idea about how Spark handles resource management and its various storage capabilities and external subsystems. And, we showed you how to instrument and monitor a Spark application. Now, in Chapter 2 you will learn all about cluster management, Spark's physical processes and how they are managed by components inside the Spark engine.