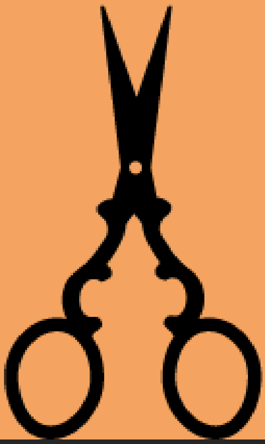


Hadoop Succinctly®

by Elton Stoneman

Syncfusion | Technology Resource Portal

Hadoop Succinctly

By
Elton Stoneman

Foreword by Daniel Jebaraj

Technical Reviewer: James McCaffrey
Copy Editor: John Elderkin
Acquisitions Coordinator: Hillary Bowling, online marketing manager, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn’t everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors’ tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Elton Stoneman is a software architect, Pluralsight author, Microsoft MVP, and Docker Captain. He has been connecting systems since 2000, and he has spent the last few years designing and building Big Data solutions with a variety of Hadoop technologies and a mixture of on-premise and cloud deliveries.

His latest Pluralsight courses cover Big Data in detail on Microsoft's Azure Cloud, which provides managed clusters for Hadoop and the Hadoop ecosystem, including Storm, HBase, and Hive. He is currently working a new course, **Hadoop for .NET Developers**, which will be released soon.

Hadoop Succinctly is Elton's third e-book for Syncfusion (following *Hive Succinctly* and *HBase Succinctly*), and it is accompanied by source code on GitHub and a Hadoop container image on the Docker Hub:

https://github.com/sixeyed/hadoop-succinctly

https://hub.docker.com/r/sixeyed/hadoop-succinctly

You'll find Elton online in other places, too—blogging at https://blog.sixeyed.com and tweeting at @EltonStoneman.

Chapter 1 Introducing Hadoop

What is Hadoop?

Hadoop is a Big Data platform with two functions—storing huge amounts of data in safe, reliable storage, and running complex queries over that data in an efficient way. Both storage and compute run on the same set of servers in Hadoop, and both functions are fault tolerant, which means you can build a high-performance cluster from commodity hardware.

Hadoop’s key feature is that it just keeps scaling. As your data needs grow, you can expand your cluster by adding more servers. The same query will run in the exact same way on a cluster with 1,000 nodes as it will on a cluster with five nodes (only much faster).

Hadoop is a free, open source project from Apache. It's been around since 2006, and the last major change to the platform came in 2013. Hadoop is a stable and mature product that has become amazingly popular, and its popularity has only increased as data becomes a core part of every business. Hadoop requires minimal investment to get started, a key feature that allows you to quickly spin up a development Hadoop environment and run some of your own data to see if it works for you.

The other major attraction of Hadoop is its ecosystem. Hadoop itself is a conceptually simple piece of software—it does distributed storage on one hand and distributed compute on the other. But the combination of the two gives us a solid base to work from that other projects have exploited to the full. Hadoop is at the core of a whole host of other Big Data tools, such as HBase for real-time Big Data, Hive for querying Big Data using an SQL-like language, and Oozie for orchestrating and scheduling work.

Hadoop sits at the core of a suite of Big Data tools, and currently trending favorites such as Spark and Nifi still put Hadoop front and center. That means a good understanding of Hadoop will help you get the best out of the other tools you use and, ultimately, get better insight from your data.

We'll learn how Hadoop works, what goes on inside the cluster, how to move data in and out of Hadoop, and how to query it efficiently. Hadoop is primarily a Java platform, and the standard query tool is the Java MapReduce framework. We won’t dive deep into MapReduce, but we will walk through a Java example and also learn how to write the same query in Python and .NET. We’ll explore the commercial distributions of Hadoop and finish with a look at some of the key technologies in the wider Hadoop ecosystem.

Do I have Big Data?

Hadoop may be conceptually simple, but it is nevertheless a complex technology. In order to run it reliably, we need multiple servers dedicated to different roles in the cluster; to work with our data efficiently, we need to think carefully about how it's stored; and to run even simple queries, we need to write a specific program, package it up, and send it to the cluster.

The beauty of Hadoop is that it doesn't get any more complex as our data gets bigger. We'll simply be applying the same concepts to larger clusters.

If you're considering adopting Hadoop, you must first determine if you really have Big Data. The answer won't simply be based on how many terabytes of data you have—it's more about the nature of your data landscape. Big Data literature focuses on the three (or four) Vs, and those are a good way of understanding whether or not the benefits of Hadoop will justify its complexity.

Think about your data in terms of the traits shown in Figure 1.

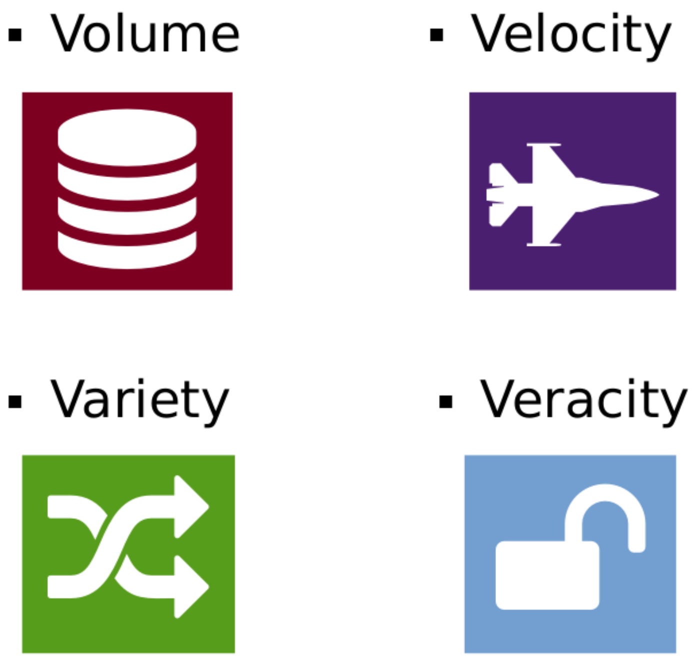


Figure 1: The Four Vs

- Volume: how much data do you have?
- Velocity: how quickly do you receive data?
- Variety: does your data come in different forms?
- Veracity: can you trust the content and context of your data?

Having a large amount of data doesn't necessarily mean you have Big Data. If you have 1 TB of log files but the log entries are all in the same format and you're only adding new files at the rate of 1 GB per month, Hadoop might not be a good fit. With careful planning, you could store all that in an SQL database and have real-time query access without the additional overhead of Hadoop.

However, if your log files are highly varied, with a mixture of CSV and JSON files with different fields within the files, and if you're collecting them at a rate of 1 TB per week, you're going to struggle to contain that in SQL, which means Hadoop would be a good fit.

In order to answer whether or not you have Big Data, think about your data in terms of the Vs, then ask yourself, “Can we solve the problems we have using any of our existing technologies?” When the answer is no, it's time to start looking into Hadoop.

The fourth V, veracity, isn't always used, but I think it adds a valuable dimension to our thinking. When we have large volumes of data coming in at high velocity, we need to think about the trustworthiness of the data.

In an Internet of Things (IoT) solution, you may get billions of events coming to you from millions of devices, and you should expect veracity problems. The data from the devices might not be accurate (especially timestamps—clocks on devices are notoriously unreliable), and you might not get the data as expected (events could be dropped or come in the wrong order). That means added complexity for your data analysis.

By looking at your data through the four Vs, you can consider whether or not you have complex storage and compute requirements, and that will help you determine if you need Hadoop.

How Hadoop works

Hadoop is a distributed technology, sharing work among many servers. Broadly speaking, a Hadoop cluster is a classic master/worker architecture, in which the clients primarily make contact with the master, as shown in Figure 2.

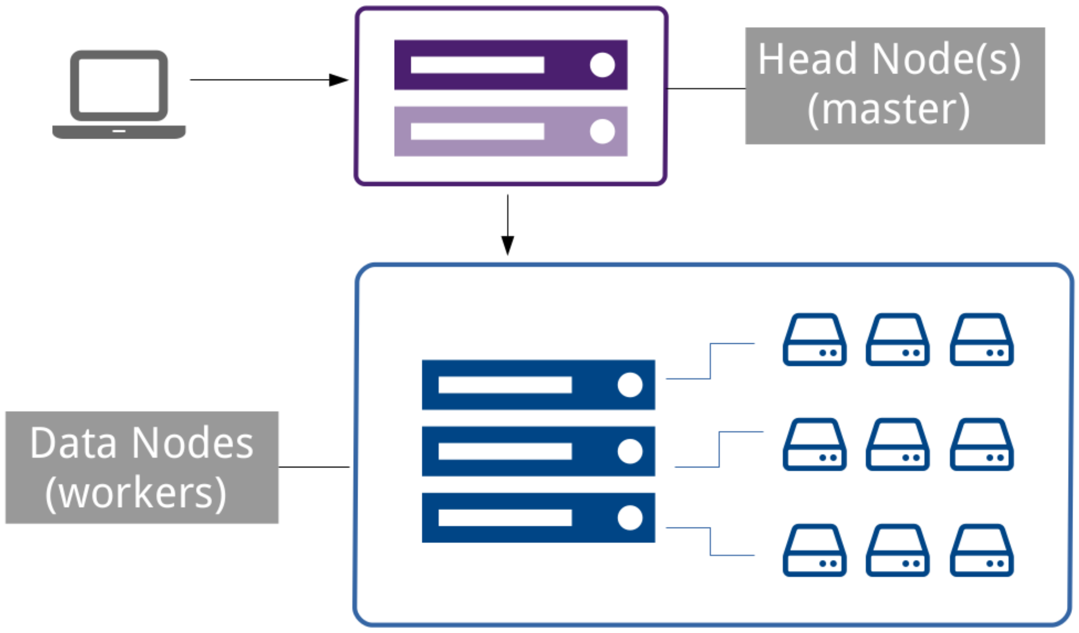


Figure 2: The Master/Worker Architecture of Hadoop

The master knows where the data is distributed among the worker nodes, and it also coordinates queries, splitting them into multiple tasks among the worker nodes. The worker nodes store the data and execute tasks sent to them by the master.

When we store a large file in Hadoop, the file gets split into many pieces, called blocks, and the blocks are shared among the slaves in the cluster. Hadoop stores multiple copies of each block for redundancy at the storage level—by default, three copies of each block are stored. If you store a 1 GB file on your cluster, Hadoop will split it into eight 128 MB blocks, and each of those eight blocks will be replicated on three of the nodes.

That original 1 GB file will actually be stored as 24 blocks of 128 MB, and in a cluster with four data nodes, each node will store six blocks across its disks, as we see in Figure 3.

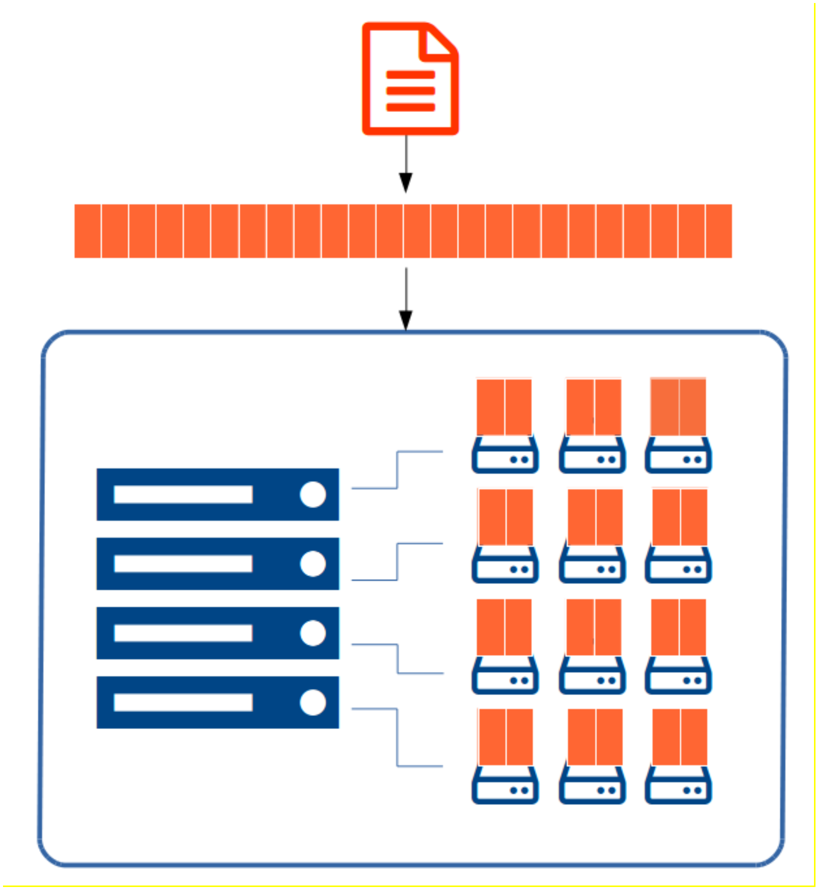


Figure 3: Large Files Split into File Blocks

Replicating data across many nodes gives Hadoop high availability for storages. In this example, we could lose two data nodes and still be able to read the entire 1 GB file from the remaining nodes. More importantly, replicating the data enables fast query performance. If we have a query that runs over all the data in the file, the master could split that query into eight tasks and distribute the tasks among the data nodes.

Because the master knows which blocks of data are on which servers, it can schedule tasks so that they run on data nodes that have a local copy of the data. This means nodes can execute their tasks by reading from the local disk and save the cost of transferring data over the network. In our example, if the data nodes have sufficient compute power to run two tasks concurrently, our single query over 1 GB of data would be actioned as eight simultaneous tasks, each running more than 128 MB of data.

Hadoop’s ability to split a very large job into many small tasks while those tasks run concurrently is what makes it so powerful—that’s called Massively Parallel Processing (MPP), and it is the basis for querying huge amounts of data and getting the response in a reasonable time.

In order to see how much benefit you get from high concurrency, consider a more realistic example—a query of over 1 TB of data that is split into 256 MB blocks (the block size is one of the many options you can configure in Hadoop). If Hadoop splits that query into 4,096 tasks and the tasks take about 90 seconds each,the query would take more than 100 hours to complete on a single-node machine.

On a powerful 12-node cluster, the same query would complete in 38 minutes. Table 1 shows how those tasks could be scheduled on different cluster sizes.

Table 1: Compute Time for Hadoop Clusters

Data Nodes	Total CPU Cores	Total Concurrent Tasks	Best Compute Time
1	1	1	100 hours
12	192	160	38 minutes
50	1200	1000	6 minutes

Hadoop aims for maximum utilization of the cluster when scheduling jobs. Each task is allocated to a single CPU core, which means that allowing for some processor overhead, a cluster with 12 data nodes, each with 16 cores, can run 160 tasks concurrently.

A powerful 50-node cluster can run more than 1000 tasks concurrently, completing our 100-hour query in under six minutes. The more nodes you add, however, the more likely it is that a node will be allocated a task for which it does not locally store the data, which means the average task-completion time will be longer, as nodes read data from other nodes across the network. Hadoop does a good job of minimizing the impact of that, as we'll see in Chapter 4 YARN—Yet Another Resource Negotiator.

Summary

This chapter has offered an overview of Hadoop and the concepts of Big Data. Simply having a lot of data doesn't mean you need to use Hadoop, but if you have numerous different types of data that are rapidly multiplying, and if you need to perform complex analytics, Hadoop can be a good fit.

We saw that there are two parts to Hadoop—the distributed file system and the intelligent job scheduler. They work together to provide high availability at the storage level and high parallelism at the compute level, which is how Hadoop enables high-performance Big Data processing without bespoke, enterprise-grade hardware.

Hadoop must understand the nature of the work at hand in order to split and distribute it, which means that in order to query Hadoop, you will need to use set patterns and a specific programming framework. The main pattern is called MapReduce, and map/reduce queries are typically written in Java. In the next chapter, we'll see how to get started with Hadoop and how to write a simple MapReduce program.

Chapter 2 Getting Started with Hadoop

Running Hadoop in a Docker container

Hadoop is a Java platform, which means installing it on a single node is a straightforward process of setting up the prerequisites, downloading the latest version, unpacking it, and running the commands that start the servers. Like other tools in the Hadoop ecosystem, it can run in three modes: local, pseudo-distributed, and distributed.

For development and exploration, the pseudo-distributed option is best—it runs the different Hadoop servers in separate Java processes, which means you get the same runtime architecture as with a full cluster while running on a single machine.

The Apache documentation covers setting up a single-node cluster and the more complex full-cluster setup.

If you've read my other Syncfusion e-books (*HBase Succinctly* and *Hive Succinctly*), you know I like to use Docker for my demo environments, and there's a Docker image for this e-book, which means you can follow along using exactly the same environment that I use without having to set up Hadoop on your machine.

You'll need to install Docker, which runs on Windows, Mac, and Linux. Then you can start a pseudo-distributed instance of Hadoop by running the command in Code Listing 1.

Code Listing 1: Running Hadoop in Docker

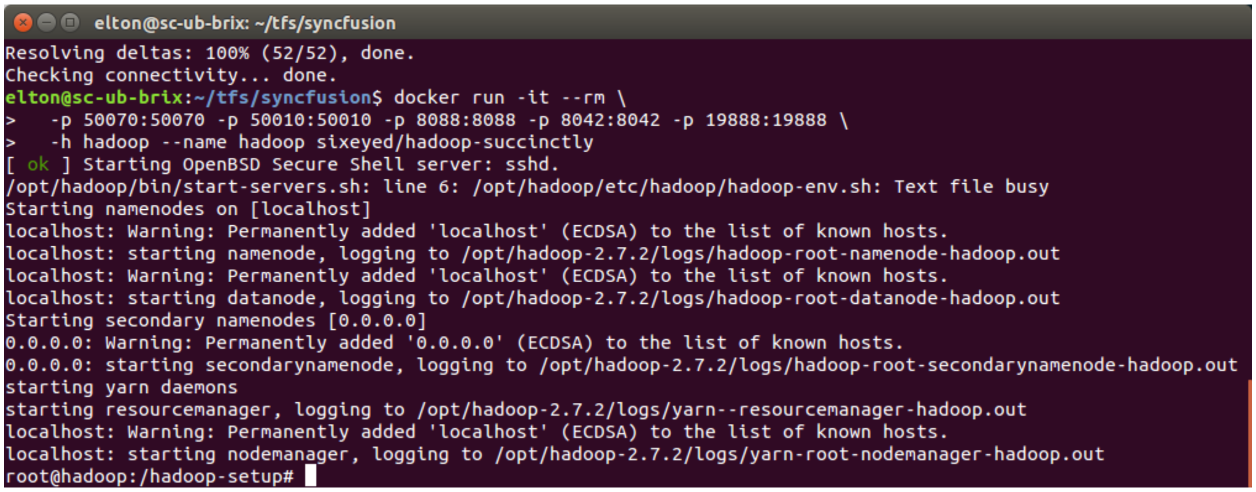
```
docker run -it --rm \
 -p 50070:50070 -p 50010:50010 -p 8088:8088 -p 8042:8042 -p 19888:19888 \
 -h hadoop --name hadoop sixeyed/hadoop-succinctly:2.7.2
```

Tip: Setting the container's host name with the *-h* ***flag makes using the Hadoop web UIs much easier. You can add an entry to your*** *hosts* ***file, associating the Docker machine IP address with the host name for the container, which means you can browse to*** *http://hadoop:8088* ***to check on job status. Doing this, the links generated by Hadoop will all work correctly.***

The first time that code runs, it will download the *Hadoop Succinctly* Docker image from the public repository. That will take a while, but then Docker caches the image locally, which means the next time you run the command, the container will start in only a few seconds.

This command launches your container interactively using the **`-it`** flag, so that you can start working with Hadoop on the command line and it will remove the container as soon as you're finished using the **`--rm`** flag. Any changes you make will be discarded, and the next container you run will return to the base-image state.

The output from starting the container will look like Figure 4, with some log entries telling you the Hadoop services are starting and a command prompt ready for you to use Hadoop.



```
elton@sc-ub-brix: ~/tfs/syncfusion
Resolving deltas: 100% (52/52), done.
Checking connectivity... done.
elton@sc-ub-brix:~/tfs/syncfusion$ docker run -it --rm \
>   -p 50070:50070 -p 50010:50010 -p 8088:8088 -p 8042:8042 -p 19888:19888 \
>   -h hadoop --name hadoop sixeyed/hadoop-succinctly
[ ok ] Starting OpenBSD Secure Shell server: sshd.
/opt/hadoop/bin/start-servers.sh: line 6: /opt/hadoop/etc/hadoop/hadoop-env.sh: Text file busy
Starting namenodes on [localhost]
localhost: Warning: Permanently added 'localhost' (ECDSA) to the list of known hosts.
localhost: starting namenode, logging to /opt/hadoop-2.7.2/logs/hadoop-root-namenode-hadoop.out
localhost: Warning: Permanently added 'localhost' (ECDSA) to the list of known hosts.
localhost: starting datanode, logging to /opt/hadoop-2.7.2/logs/hadoop-root-datanode-hadoop.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: Warning: Permanently added '0.0.0.0' (ECDSA) to the list of known hosts.
0.0.0.0: starting secondarynamenode, logging to /opt/hadoop-2.7.2/logs/hadoop-root-secondarynamenode-hadoop.out
starting yarn daemons
starting resourcemanager, logging to /opt/hadoop-2.7.2/logs/yarn--resourcemanager-hadoop.out
localhost: Warning: Permanently added 'localhost' (ECDSA) to the list of known hosts.
localhost: starting nodemanager, logging to /opt/hadoop-2.7.2/logs/yarn-root-nodemanager-hadoop.out
root@hadoop:/hadoop-setup#
```

Figure 4: The Running Docker Container

Verifying your installation

The Hadoop package comes with sample MapReduce jobs you can use to check that your installation is working correctly. We'll look at the **hadoop** command line in more detail later, but in order to verify your environment, you can run the commands in Code Listing 2.

Code Listing 2: Verifying Your Hadoop Installation

```
hadoop fs -mkdir -p /user/root/input

hadoop fs -put $HADOOP_HOME/etc/hadoop input

hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-
2.7.2.jar grep input output 'dfs[a-z.]+'
```

The first command creates a directory called **input** in the home directory for the root user in the Hadoop file system, and the second command copies all the files in the **etc/hadoop** directory to the **input** directory.

The third command submits a job to Hadoop in order to run over the data we've just stored. The job is packaged in a Java JAR file, which is a sample supplied with Hadoop, and we send it to Hadoop with the arguments **grep input output 'dfs[a-z.]+'**. That will run a job that searches all the files in the input directory and looks for words starting with “dfs,” then it counts all the matches.

As the job runs, it will produce a lot of output that tells you the current progress. When it completes, you can view the final result with the command in Code Listing 3.

Code Listing 3: Output of the Sample Job

```
# hadoop fs -cat output/*

6     dfs.audit.logger

4     dfs.class

3     dfs.server.namenode.

2     dfs.period

2     dfs.audit.log.maxfilesize

2     dfs.audit.log.maxbackupindex

1     dfsmetrics.log

1     dfsadmin

1     dfs.servers

1     dfs.replication

1     dfs.file
```

You should see a similar list of words beginning with “dfs” and the count of occurrences for each of them. If your program doesn't run as expected, you'll need to check your installation steps or simply use the Docker container.

In the rest of this chapter, we'll build our own version of the word count program as a simple introduction to MapReduce programming.

The structure of a MapReduce program

There are three parts to a MapReduce program:

- The **driver**, which configures how the program will run in Hadoop.
- The **mapper**, which is fed input data from Hadoop and produces intermediate output.
- The **reducer**, which is fed the intermediate output from the mappers and produces the final output.

The mapper runs in parallel—each input file (or, for large input files, each file block) has a dedicated mapper task that will run on one of the data nodes. Each mapper is given only a small subset of the total input data, and its job is to read through the input and transform it into intermediate output.

Each mapper produces only a part of the total output, and the reducer's job is to build the final results, aggregating the many intermediate outputs from the mappers into a single result. Simple jobs have only one reducer task. Figure 4 shows the flow of data in a MapReduce job.

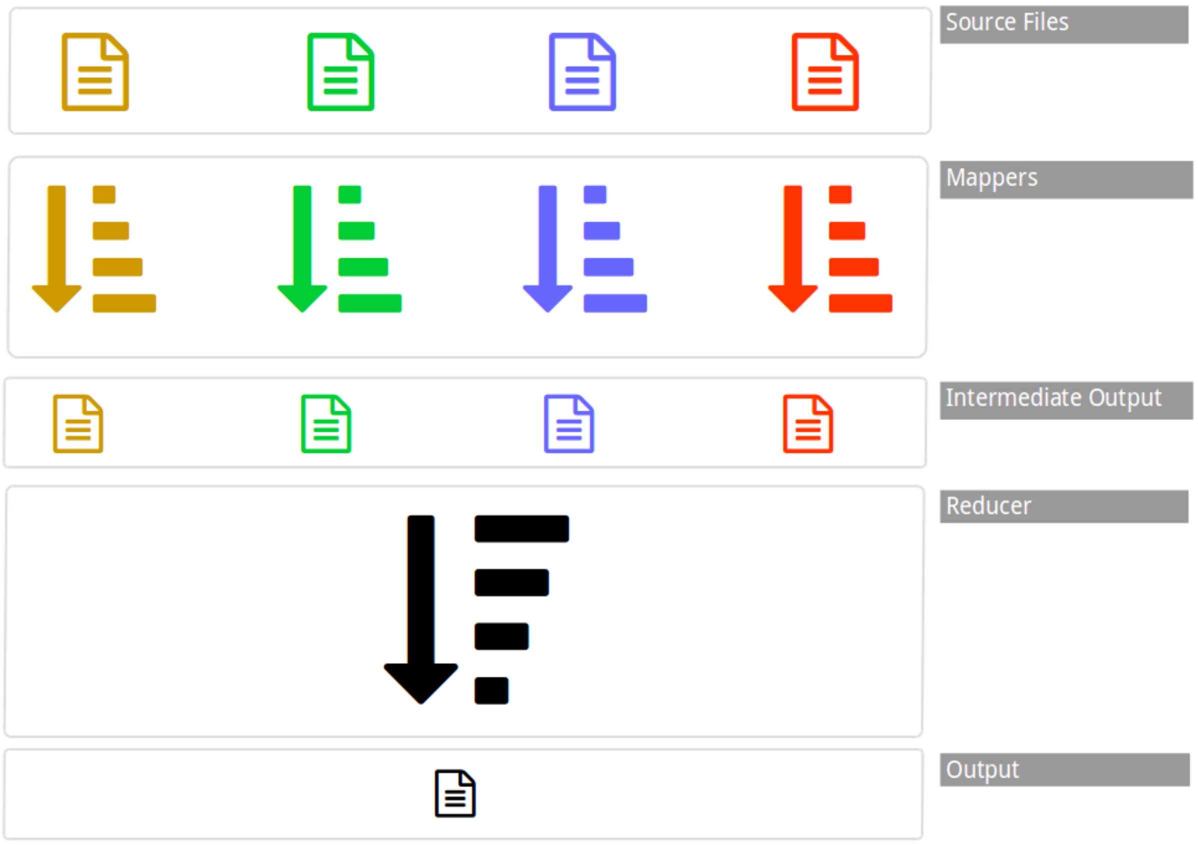


Figure 4: Data Flow in MapReduce

In our word count example, the mapper will be given the contents of a file, one line at a time. The MapReduce framework takes care of choosing the file by reading through it and sending lines to the mapper, which means we need to write the code to transform the data we receive. In this case, we'll split the incoming line into words and look for any that start with "dfs." When we find a match, we'll write the word to the output.

Hadoop collects the output from all the mappers as they complete, it sorts and merges the intermediate output, then it sends that output to the reducer. The reducer will receive the intermediate output from all the Hadoop mappers as a Java `Iterable` collection. We'll need to write code to sum the counts for each word, then write the final counts as the output.

This explains why having a single reducer is common practice. The reducer's job is to collate the intermediate results, so we usually give all the results to one instance. If the reducer needs to aggregate the output from the mappers, it will have all the available data needed to do so. We can run a job with multiple reducers, which improves performance, but that means each reducer only aggregates part of the intermediate output and our final result will be split across multiple reducer outputs.

When data moves between the mappers and reducer, it might cross the network, so all the values must be serializable. In order to store data, the MapReduce framework provides its own set of classes that are built for efficient serialization (you will use the **Text** class, for instance, instead of a normal Java **String**). Additionally, in order to validate that the job is properly configured before it starts, Hadoop must be explicitly told the data types of your classes.

The driver executes this configuration, specifying which mapper and reducer to use and the type of inputs and outputs. Inputs and outputs are always a key-value pair, which means you specify the type of the key and the type of the value in the driver.

There are a few moving parts in a MapReduce program that need to be wired up correctly, but the code is quite straightforward.

The word count mapper

We need to write three classes to implement a MapReduce program: a mapper, a reducer, and a driver. The full code is up on the GitHub repository that accompanies this e-book: sixeyed/hadoop-succinctly/java. I've used the new Visual Studio Code tool as the IDE, but the Java project is built with Maven. Information for building the JAR is in the README for the project.

Our mapper and reducer classes will inherit from base classes in the Hadoop client library, which means the project needs to reference the Apache package. The Maven configuration for the dependency is in Code Listing 4 and targets version 2.7.2 of the library, which is the current version of Hadoop at the time of writing and matches the Hadoop installation in the **hadoop-succinctly** container.

Code Listing 4: Maven Dependency for MapReduce

```
<dependency>
      <groupId>org.apache.hadoop</groupId>

      <artifactId>hadoop-client</artifactId>

      <version>2.7.2</version>
</dependency>
```

Starting with the mapper, we extend the base class **org.apache.hadoop.mapreduce.Mapper** and specify the types for the input and output key-value pairs. In Code Listing 5, the first two type arguments are the expected input key and value types, and the next two types are the key and value types the mapper will emit.

Code Listing 5: Extending the Base Mapper

```
public static class Map
              extends Mapper<LongWritable, Text, Text, IntWritable>
```

For each line in the file, Hadoop will call the **`map()`** method, passing it a key-value pair. In our driver configuration, we'll set up the input format, so that the map will be called with a key that is a **`LongWritable`** object containing the position of the line in the file and a value that is a **`Text`** object containing the actual line of text.

Code Listing 6 shows the definition of the map method in which the input key and value types match the type declaration for the mapper class. The map method is also passed a **`Context`** object, which it can use to write out values.

Code Listing 6: Signature of the Map Method

```
public void map(LongWritable key, Text value, Context context)
```

Mappers aren't required to do anything when **`map()`** is called, but they can emit zero or more key-value pairs by writing them to the **`Context`** object. Any values written are collected by Hadoop and will be sorted, merged, and sent on to the reducer.

The code for the **`map()`** method is simple—we tokenize the string into individual words, and if the word contains "dfs," we emit it to the context. The key-value pair we emit uses a **`Text`** object for the key, which is the full word where we found a match, and the value is an **`IntWritable`** object. We'll use the integer value later to sum up the counts, but we don’t do any counting in the mapper—we always emit a value of 1. Code Listing 7 shows the full **`map()`** implementation.

Code Listing 7: The Map Method

```
public void map(LongWritable key, Text value, Context context)
                throws IOException, InterruptedException {
         String line = value.toString();
         StringTokenizer tokenizer = new StringTokenizer(line);
         while (tokenizer.hasMoreTokens()) {
            String next = tokenizer.nextToken();
            if (next.contains("dfs")){
                word.set(next);
                context.write(word, one);  // 'one' is IntWritable(1)
            }
         }
      }
```

We always emit 1 as the count when we find a matching word because it's not the mapper's job to do any aggregation work (such as counting up values). You might think it would be more efficient for the mapper to keep counts in local state, with a dictionary of words, incrementing the counts as it reads the lines and emitting the counts as a batch, but this approach is problematic.

The mapper should be able to work for any file size. A mapper might run a task for a 1 MB file, but the same mapper must also work correctly if it's given a 1 GB file. Hadoop streams through the input file in order to minimize the amount of memory needed, but if your mapper holds state in a dictionary, its memory footprint is unknowable, and for large files that could cause an out-of-memory exception.

Mappers should be lightweight components that are called with a single piece of input and respond with zero or more pieces of output. You might get performance gains by microbatching inside the mapper, but doing so limits that mapper to only working with small inputs. However, if your mapper behaves correctly, your overall job can scale to any size, which is a better performance option.

Aggregation work belongs in the reducer, although you can perform some postmapper aggregation using a combiner, which works like a mini-reducer, running over the output from each mapper.

The word count reducer

As with the mapper, the reducer extends from the Hadoop base class and specifies the input and output key and value types. For the reducer, the input types need to match the output types from the mapper, but the reducer can emit different output types if need be.

In our case, the reducer receives **`Text`** keys containing matching words, and **`IntWritable`** values that are the word counts initialized to 1. The reducer emits the same types—the class definition with type arguments is shown in Code Listing 8.

Code Listing 8: Extending the Base Reducer

```
public static class Reduce
              extends Reducer<Text, IntWritable, Text, IntWritable>
```

After the map phase, the **`reduce()`** method on the reducer will be called with a key and a collection of values. Hadoop sorts and merges the keys from the mappers, and you can rely on that in your reducer. You know you will only see each key once (with the value being the collection of all the mapper values for that key), and you know the next key will have a higher value (lexicographically) than the current key.

The **`reduce()`** method defines the key and value collection types, together with a **`Context`** object that is used to write output in the same way as for the mapper. Code Listing 9 shows the **`reduce()`** method in full.

Code Listing 9: The Reduce Method

```

public void reduce(Text key, Iterable<IntWritable> values, Context context)

                   throws IOException, InterruptedException {

         int sum = 0;

         for (IntWritable val : values) {

            sum += val.get();

         }

         context.write(key, new IntWritable(sum));
      }
```

For the word count, **`reduce()`** will be called with a text value containing a string that matches “dfs” and a collection of **`IntWritable`** objects. When Hadoop merges the output from the mappers, it sorts all the keys and groups all the values for one key into a collection. The reducer is then invoked for each key, receiving the collection of values for that key in the **`reduce()`** method.

The value collection will have at least one element because the mapper will always emit a value for each key. The collection might hold any number of elements in the collection, and Hadoop will present them to the reducer as they were emitted by the mapper. In the reducer, we aggregate the intermediate output from the mappers into the final output—in this case, we just add all the values to give us a total count for each key.

When we write the output from the reducer, we do it in the same way as in the mapper, by calling **`write()`** on the **`Context`** class, passing it a key-value pair. The key is the word that contains the match, and we simply write the same key that we receive in the method call.

The value is the sum of all the 1 values from the mappers, and we write that out as an **`IntWritable`**. Within the method body, we use an integer to sum the values, but we need to use one of Hadoop's serializable types in order to write to the context, so we create an **`IntWritable`** from the integer.

The word count driver

The driver is the entry point for the MapReduce program, and it needs only to configure the rest of the components. Drivers don't have to extend from a base class or implement an interface, but they do need to be configured as the entry point for the JAR—in the sample code, that's done in the Maven setup by specifying the **`mainClass`** element, as seen in Code Listing 10.

Code Listing 10: Specifying the Main Class

```
<archive>

  <manifest>

    <addClasspath>true</addClasspath>

    <mainClass>com.sixeyed.hadoopsuccinctly.WordCount</mainClass>

  </manifest>
</archive>
```

The **`WordCount`** class has a standard **`main()`** method with a **`String`** array to receive command-line arguments. It's common to set up your driver so that any values that will change per environment, or for different runs of the program, are read from the arguments. For the word count, we'll use the arguments to specify the source for input files and the target for output files.

Configuring a **`Job`** object, i.e. defining what the job will do, is the primary work of the driver. The **`Job`** object has a dependency on a **`Configuration`** object, but for our simple example we can use default configuration values. Code Listing 11 shows the creation of the **`Job`** object.

Code Listing 11: Setting Up the Job Configuration

```
Configuration conf = new Configuration();

Job job = new Job(conf, "wordcount");
```

Here we specify a name for the job, “wordcount,” which will be shown in the Hadoop status UIs.

In order to execute the job, Hadoop requires that all the types involved be explicitly specified. First, we specify the JAR that contains our own types, the output key and value types, and the mapper and reducer types. There are various set methods on the **`Job`** class for declaring these types. In Code Listing 12, we configure the JAR as the archive that contains the **`WordCount`** class, **`Text`** class as the output key, **`IntWritable`** as the output value, and **`Map`** and **`Reduce`** classes.

Code Listing 12: Specifying Types for the Job

```
    job.setJarByClass(WordCount.class);

    job.setOutputKeyClass(Text.class);

    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(Map.class);

    job.setReducerClass(Reduce.class);
```

Note that we don't specify the input key and value types, although Hadoop needs that information, too. However, MapReduce programs work at a low level where the mapper code is tightly coupled to the expected data format.

Hadoop supports a variety of file formats commonly found in Big Data solutions—plain text files, GZipped text files, and more efficient formats such as Avro and Parquet. In the job, we configure the file format we expect and that will implicitly set up the input key and value types.

In Code Listing 13, specifying the **`TextInputFormat`** class gives us the **`LongWritable`** key and **`Text`** value input types that the mapper expects.

Code Listing 13: Specifying Input and Output Formats

```
    job.setInputFormatClass(TextInputFormat.class);

    job.setOutputFormatClass(TextOutputFormat.class);
```

The final responsibilities of the driver are to tell Hadoop where to find the source files and where the output will be written. That's done with the **`FileInputFormat`** and **`FileOutputFormat`** classes, passing them the **`Job`** object we've set up, and the directory path from the main method arguments. We can also tell Hadoop that the client should wait for this job to complete rather than returning as soon as it is submitted.

Code Listing 14: Final Job Configuration

```
    FileInputFormat.addInputPath(job, new Path(args[0]));

    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
```

Running the MapReduce program

In order to run a MapReduce program, we can submit it to Hadoop using the command line. The **`hadoop`** command lets us work with both the Hadoop file system and the job scheduler, and the **`jar`** operation submits a new job. The full command is in Code Listing 15.

Code Listing 15: Submitting the Job

```
hadoop jar /packages/wordcount-1.0-SNAPSHOT.jar input output2
```

We pass the **`jar`** command the path to the JAR file that contains our driver, mapper, and reducer, along with any arguments the driver is expecting. In this case, I've copied the JAR file to the packages folder on the system root, and I'm passing **`input`** as the source directory the mappers will read from and **`output2`** as the target directory to which the reducer will write.

When the job completes, the output from the reducer will be written to a file in the output directory. Each reducer has its own output file, and we can see the combined output from all reducers (whether there are one or many) by using the **`hadoop fs cat`** command. Code Listing 16 shows abbreviated output from the MapReduce word count.

Code Listing 16: Viewing the Job Output

```
# hadoop fs -cat output2/*

"dfs" 3

*.sink.ganglia.tagsForPrefix.dfs=   1

dfs.class=org.apache.hadoop.metrics.file.FileContext  1

dfs.fileName=/tmp/dfsmetrics.log    1

dfs.period=10     1
```

Notice that the output is different from when we ran the sample MapReduce program as we verified the Hadoop installation because the new program uses different mapper and reducer logic.

Summary

In this chapter, we got started with Hadoop and had a quick introduction to MapReduce, which is the programming approach used in Hadoop for writing jobs that can run at massive levels of parallelism.

Hadoop is straightforward to install on a single node for development and testing, but you can avoid installing it locally by using the Docker image that accompanies this e-book. However you run it, the Hadoop distribution comes with some sample programs you can use to verify that the installation works correctly.

We used one of those samples as the basis for creating our own MapReduce program for counting words—word count is the “Hello World” of Big Data programming. We saw that the custom logic in the mapper is called for each line in the input source files, and its role is to produce intermediate output. Many mappers can run concurrently, each processing a small part of the total input while generating intermediate output that Hadoop stores.

When all the mappers are complete, the reducer runs over the sorted, merged, intermediate output and produces the final output.

In order to configure the parts of the job, you need a driver that specifies the mapper and reducer to use and the types they expect for the input and output key-value pairs. The MapReduce program, containing driver, mapper, and reducer, gets built into a JAR file, and you use the Hadoop command to submit a job, supplying the path to the JAR file and any arguments the driver is expecting.

Now that we've had a hands-on introduction, we'll look more closely at the Hadoop platform, beginning with the Hadoop file system.

Chapter 3 HDFS—The Hadoop Distributed File System

Introducing HDFS

HDFS is the storage part of the Hadoop platform, and with HDFS you get reliability and scale with commodity hardware.

Commodity is important—it means you don't need specialist hardware in your Hadoop cluster, and nodes don't need to have the same—or even similar—specifications. Many Hadoop installations have started with a cluster built from beg-or-borrow servers and expanded with their own higher-spec machines when the project took off. You can add new nodes to a running cluster while increasing your storage and compute power without any downtime.

The resilience built into HDFS means servers can go offline or disks can fail without loss of data, so that you don't even need RAID storage on your machines. And because Hadoop is much more infrastructure-aware than other compute platforms, you can configure Hadoop to know which rack each server node is on. It uses that knowledge to increase redundancy—by default, data stored in Hadoop is replicated three times in the cluster. On a large cluster with sufficient capacity, Hadoop will ensure one of the replicas is on a different rack from the others.

As far as storage is concerned, HDFS is closed system. When you read or write data in Hadoop, you must do it through the HDFS interface—because of its unique architecture, there's no support for connecting directly to a data node and reading files from its disk. The data is distributed among many data nodes, but the index specifying which file is on which node is stored centrally. In this chapter, we'll get a better understanding of the architecture and see how to work with files in HDFS using the command line.

Architecture of HDFS

HDFS uses a master/slave architecture in which the master is called the “name node” and the slaves are called “data nodes.” Whenever you access data in HDFS, you do so via the name node, which owns the HDFS-equivalent of a file allocation table, called the file system namespace.

In order to write a file in HDFS, you make a PUT call to the name node and it will determine how and where the data will be stored. To read data, you make a GET call to the name node, and it will determine which data nodes get copies of the data and will direct you to read the data from those nodes.

The name node is a logical single point of failure for Hadoop. If the name node is unavailable, you can't access any of the data in HDFS. If the name node is irretrievably lost, your Hadoop journey could be at an end—you'll have a set of data nodes containing vast quantities of data but no name node capable of mapping where the data is, which means it might be impossible to get the cluster operational again and restore data access.

In order to prevent that, Hadoop clusters have a secondary name node that has a replicated file index from the primary name node. The secondary is a passive node—if the primary fails, you'll need to manually switch to the secondary, which can take tens of minutes. For heavily used clusters in which that downtime is not acceptable, you can also configure the name nodes in a high-availability setup.

Figure 5 shows a typical small-cluster configuration.

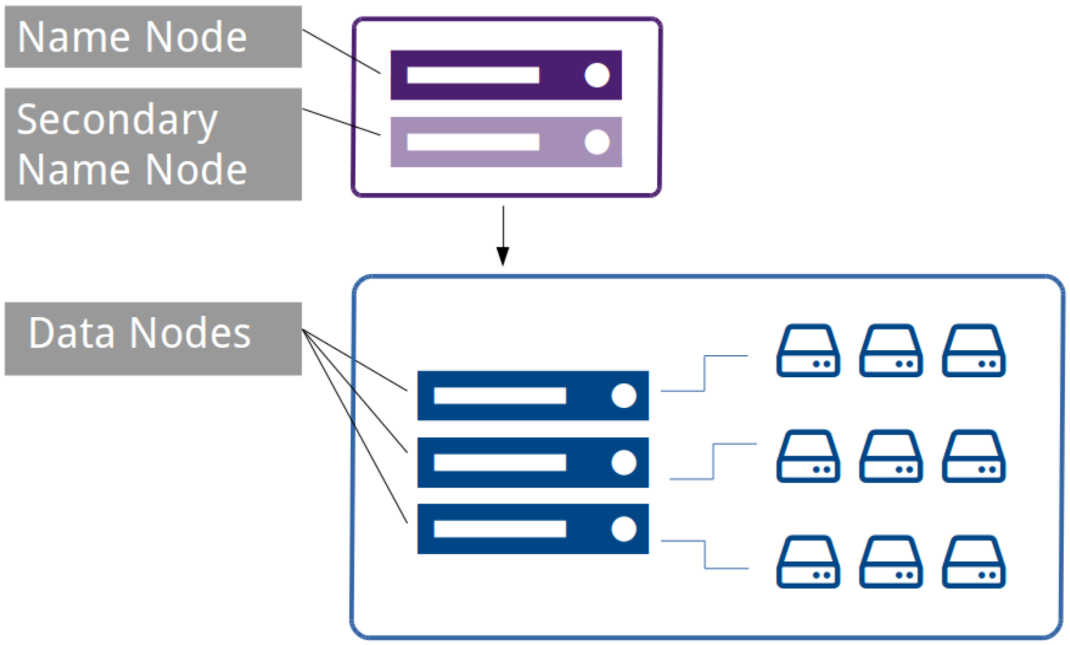


Figure 5: HDFS Architecture

For consumers, the Hadoop cluster is a single, large-file store, and the details of how files are physically stored on the data nodes is abstracted. Every file has a unique address, with the HDFS scheme and a nested folder layout—e.g., **`hdfs://[namenode]/logs/2016/04/19/20/web-app-log-201604192049.gz`**.

Accessing HDFS from the command line

The **`hadoop`** command provides access to storage as well as to compute. The storage operations begin with **`fs`** (for "file system"), then typically follow with Linux file operation names. You can use **`hadoop fs -ls`** to list all the objects in HDFS. In order to read the contents of one or more files (which we have already seen), use **`hadoop fs -cat`**.

HDFS is a hierarchical file system that can store directories and files, and its security model is similar to Linux's—objects have an owner and a group, and you can set read, write, and execute permissions on objects.

Note: Hadoop has wavered between having a single command for compute and storage or having separate ones. The *hdfs* ***command is also supported for storage access, but it provides the same operations as the*** *hadoop* ***command. Anywhere you see*** *hdfs dfs* ***in Hadoop literature, you can substitute it with*** *hadoop fs* ***for the same result.***

Users in HDFS have their own home directory, and when you access objects, you can either specify a full path from the root of the file system or a partial path assumed to start from your home directory. In Code Listing 17, we create a directory called 'ch03' in the home directory for the root user and check that the folder is where we expect it to be.

Code Listing 17: Creating Directories in HDFS

```
# hadoop fs -mkdir -p /user/root/ch03

# hadoop fs -ls

Found 1 items

drwxr-xr-x   - root supergroup  0 2016-04-15 16:44 ch03

# hadoop fs -ls /user/root

Found 1 items

drwxr-xr-x   - root supergroup  0 2016-04-15 16:44 /user/root/ch03
```

In order to store files in HDFS, we can use the **`put`** operation, which copies files from the local file system, into HDFS. If we're using the **`hadoop-succinctly`** Docker container, the commands in Code Listing 18 copy some of the setup files from the container's file system into HDFS.

Code Listing 18: Putting Files into HDFS

```
# hadoop fs -put /hadoop-setup/ ch03

# hadoop fs -ls ch03/hadoop-setup

Found 3 items

-rw-r--r--   1 root supergroup        294 2016-04-15 16:49 ch03/hadoop-
setup/install-hadoop.sh

-rw-r--r--   1 root supergroup        350 2016-04-15 16:49 ch03/hadoop-
setup/setup-hdfs.sh

-rw-r--r--   1 root supergroup        184 2016-04-15 16:49 ch03/hadoop-
setup/setup-ssh.sh
```

With the **`put`** command, we can copy individual files or whole directory hierarchies, which makes for a quick way of getting data into Hadoop.

Tip: When you're working on a single node, copying data into HDFS effectively means copying it from the local drive to a different location on the same local drive. If you're wondering why you have to do that, remember that Hadoop is inherently distributed. When you run *hadoop fs -put* ***on a single node, Hadoop simply copies the file—but run the same command against a multinode cluster and the source files will be split, distributed, and replicated among many data nodes. You use the exact same syntax for a development node running on Docker as for a production cluster with 100 nodes.***

In order to fetch files from HDFS to the local file system, use the **`get`** operation. Many of the HDFS commands support pattern matching for source paths, which means you can use wildcards to match any part of filenames and the double asterisk to match files at any depth in the folder hierarchy.

Code Listing 19 shows how to locally copy any files whose name starts with 'setup' and that have the extension 'sh' in any folder under the 'ch03' folder.

Code Listing 19: Getting Files from HDFS

```
# mkdir ~/setup

root@21243ee85227:/hadoop-setup# hadoop fs -get ch03/**/setup*.sh ~/setup

root@21243ee85227:/hadoop-setup# ls ~/setup/

setup-hdfs.sh  setup-ssh.sh
```

There are many more operations available from the command line, and **`hadoop fs -help`** lists them all. Three of the most useful are **`tail`** (which reads the last 1KB of data in a file), **`stat`** (which sees useful stats on a file, including the degree of replication), and **`count`** (which tells how many files and folders are in the hierarchy).

Write once, read many

HDFS is intended as a write-once, read-many file system that significantly simplifies data access for the rest of the platform. As we've seen, in a MapReduce job, Hadoop streams through input files, passing each line to a mapper. If the file is editable, the contents can potentially change during the execution of the task. Content already processed might change or be removed entirely, invalidating the result of the job.

Storing data in Hadoop from the command line requires three operations: the **`put`** operation, which we've already seen; **`moveFromLocal`**, which deletes the local source files after copying them; and **`appendFromLocal`**, which adds the contents of local files to existing files in HDFS.

Append is an interesting operation, because we can't actually edit data in Hadoop. In earlier versions of Hadoop, append wasn't supported—we could only create new files and add data while the file handle was open. Once we closed the file, it became immutable and, instead of adding to it, we had to create a new file and write the contents of the original along with the new data we wanted to add, then save it with a different name.

HDFS's ability to support append functionality has considerably broadened its potential—this means Apache's real-time Big Data platform, HBase, can use HDFS for reliable storage. And note that being append-only isn't the restriction it might seem if you're from a relational database background. Typically, Hadoop is used for batch processing over data that records something (an event or a fact), and that data is static.

You might have a file in Hadoop that records a day's worth of financial trades—those are a series of facts that will never change, so there's no need to edit the file. If a trade was booked incorrectly, it will be rebooked, which is a separate event that will be recorded in another day's file.

The second event may reverse the impact of the first event, but it doesn't mean the event never happened. In fact, storing both events permanently enables much richer analysis. If you allowed updates and could edit the original trade, you'd lose any record of the original state along with the knowledge that it had changed.

In order to append data without altering the original file, HDFS splits the new data into blocks and adds them to the original blocks into which the file was split.

File splits and replication

We saw in Chapter 1 that HDFS splits large files into blocks when we store them. The default block size can differ between Hadoop distributions (and we can configure it ourselves for the whole cluster and for individual files), but it's typically 128 MB. All HDFS read and write operations work at the block level, and an optimal block size can provide a useful performance boost.

The correct block size is a compromise, though—one that must balance some conflicting goals:

- The name node stores metadata entries for every file block in memory; smaller block size leads to significantly more blocks and higher metadata storage.
- Job tasks run at the block level, so having very large blocks means the system can get clogged with long-running tasks. Having smaller blocks can mean the cost of spinning up and managing tasks outweighs the benefit of having greater parallelism.
- Blocks are also the level of replication, so that having smaller blocks means having more blocks to distribute around the cluster. A greater degree of distribution can decrease the chances of scheduling a task to run on a node that keeps data locally.

Block size is one part of Hadoop configuration you can tune to your requirements. If you typically run jobs with very large inputs—running over terabytes of data—setting a higher block size can improve performance. A block size of 256 MB or even 512 MB is not unusual for larger workloads.

Another parameter we can change for our cluster, or for individual files, is the degree of replication. Figure 6 shows how some of the splits in a 1 GB file can be distributed across a cluster with 12 data nodes.

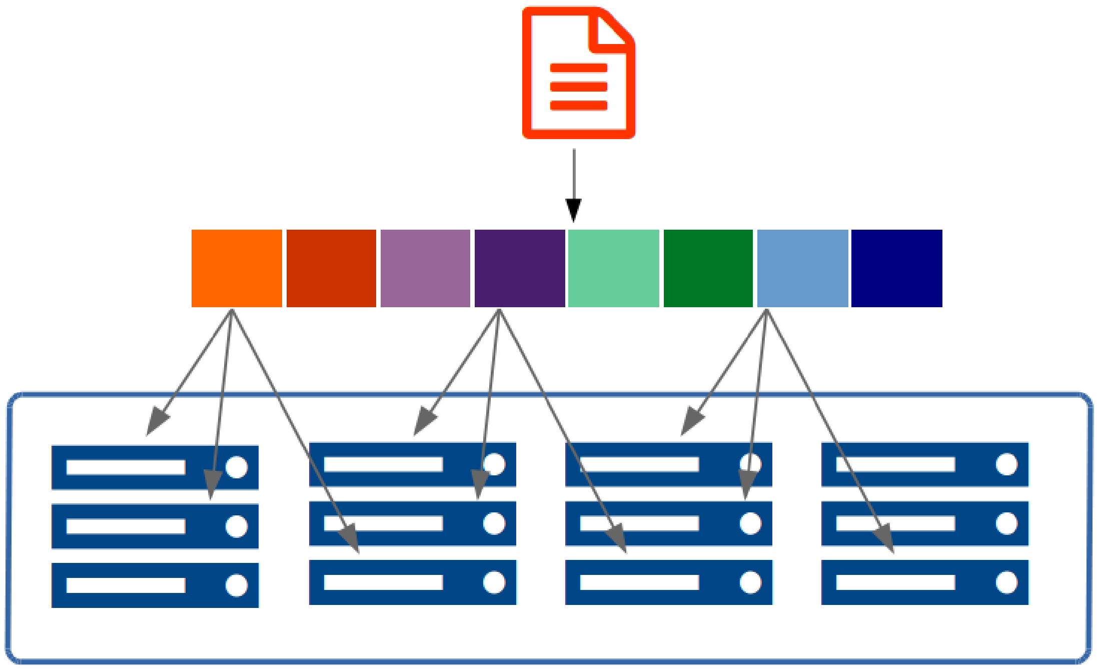


Figure 6: HDFS File Splits and Replication

With a default block size of 128 MB and a default replication factor of 3, HDFS will split the file into eight blocks, and each block will be copied to three servers: two on one rack and the third on a different rack. If you submit a job that queries the entire file, it will be split into eight tasks, each processing one block of the file.

In this example, there are more data nodes than there are tasks in the job. If the cluster is idle, Hadoop can schedule each task so it runs on a server that has the block locally. This, however, is a best-case scenario.

If the cluster isn't idle, or if you have files with a small number of blocks, or if there's a server outage, there's a much larger chance that a task must be scheduled on a node that doesn't keep the data locally. Reading data from another node in the same rack, or worse still, from a node in a different rack, might be orders of magnitude slower than reading from the local disk, which means tasks will take much longer to run.

HDFS lets you change the replication factor, but the optimal value is also a compromise between conflicting goals:

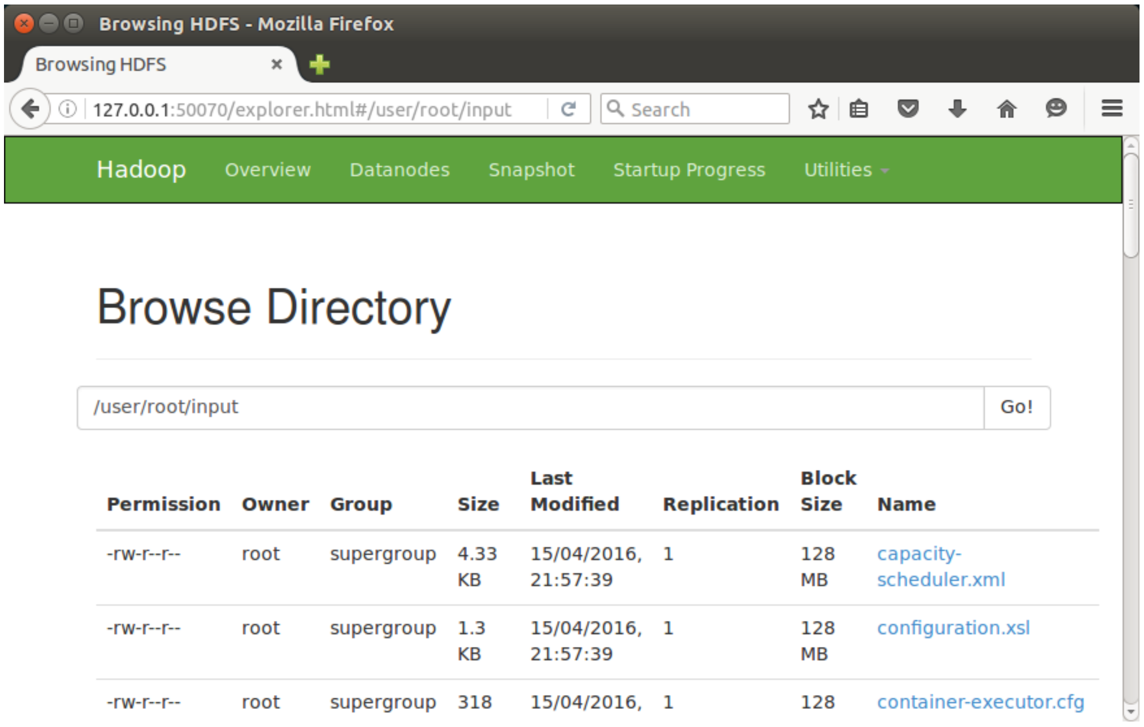
- A higher replication factor means blocks are copied to more data nodes, increasing the chances for tasks to be scheduled on a node that has the data locally.
- More replication means higher disk consumption for the same amount of raw data. A 10 GB file in HDFS will take up 30 GB of disk space with a replication factor of 3, but a 50 GB file will have a replication factor of 5.

Block size and replication factor are useful parameters to consider when it comes to tuning Hadoop for your own data landscape.

HDFS Client and the web UI

The Hadoop FS shell is the easiest way to get started with data access in HDFS, but for serious workloads you'll typically write data programmatically. HDFS supports a Java API and a REST API for programmatic access. And web UI, which you can use to explore the file system, is built into HDFS.

The web UI runs on the name node, on port 50070 (by default). That port is exposed by the **`hadoop-succinctly`** Docker container, which means you can browse to **`http://127.0.0.1:50070/explorer.html`** (substitute 127.0.0.1 with the IP address of your Docker VM if you're using Mac or Windows). Figure 7 shows how the storage browser looks.


Browsing HDFS - Mozilla Firefox
Browsing HDFS
127.0.0.1:50070/explorer.html#/user/root/input
Search
Hadoop
Overview
Datanodes
Snapshot
Startup Progress
Utilities

Browse Directory

/user/root/input Go!

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	root	supergroup	4.33 KB	15/04/2016, 21:57:39	1	128 MB	capacity-scheduler.xml
-rw-r--r--	root	supergroup	1.3 KB	15/04/2016, 21:57:39	1	128 MB	configuration.xsl
-rw-r--r--	root	supergroup	318	15/04/2016,	1	128	container-executor.cfg

Figure 7: The HDFS Explorer

From the browser, you can navigate around the directories in HDFS, see the block information for files, and download entire files. It's a useful tool for navigating HDFS, especially as it is read-only, so that the web UI can be given to users who don't otherwise have access to Hadoop.

Summary

In this chapter, we had a closer look at HDFS, the Hadoop Distributed File System. We learned how HDFS is architected with a single active name node that holds the metadata for all the files in the cluster and with multiple data nodes that actually store the data.

We got started reading and writing data from HDFS using the **`hadoop fs`** command line, and we learned how HDFS splits large files into blocks and replicates the blocks across many data nodes in order to provide resilience and enable high levels of parallelism for MapReduce tasks. We also learned that block size and replication factor are configurable parameters that you might need to flex in order to get maximum performance.

Lastly, we had a brief look at the HDFS explorer in the Hadoop Administration web UI. There are other UIs provided by embedded HTTP servers in Hadoop that we'll see in later chapters.

Next, we'll examine the other part of Hadoop—the resource negotiator and compute engine.

Chapter 4 YARN—Yet Another Resource Negotiator

Introducing YARN

Until now, I've referred to the compute part of Hadoop as a job scheduler, but more correctly it should be referred to as a job scheduler, resource manager, and task monitor combined.

The compute platform was rewritten for Hadoop version 2 (released in 2013), and the resource management part was abstracted from MapReduce, which means Hadoop clusters can run different types of jobs, not only MapReduce programs. The new resource manager is YARN, named in the style of YAML (Yet Another Markup Language) and Yaws (Yet another web server).

We will look at how YARN schedules and manages jobs because understanding this will help us check on the status of long-running jobs and work on optimizing clusters. But from a programming perspective, YARN is a black box for Hadoop programs—you submit your MapReduce job with its configuration to the cluster and let YARN do the rest.

We've seen that in MapReduce we need to build a custom mapper, reducer, and driver, but we don't need to build anything else. And we don't need to tell Hadoop how to break up the job or where to run the tasks. That's all done by the MapReduce framework and YARN.

YARN is an abstracted job scheduler, and the Hadoop ecosystem has expanded to make full use of it. In a typical Hadoop cluster, YARN might schedule standard MapReduce jobs, run Hive queries using the Tez engine, and run Spark—all on the same set of compute resources.

Just as HDFS provides resilience and scalability at the storage layer, so YARN provides the same at the compute layer. Once a job has been submitted to YARN, it can be expected to complete at some point, even if there are repeated hardware failures during the life of the job.

The structure of a YARN job

YARN provides a logical separation of roles between the resource manager, which determines where tasks should run, and the job monitors, which report on task progress. That separation was a core driver for the new architecture in Hadoop 2 because it allows the job management tasks to be distributed around the cluster, which means the platform can run at higher scale.

The resource manager is the centralized master component, and it typically runs on the same node in the Hadoop cluster as the HDFS name node. The resource manager runs as a service, and when it starts a new job it creates an application master—a software component that has the responsibility of ensuring that the job (which could be a MapReduce program or an interactive Spark session) runs successfully.

Application masters work with the resource manager to request resources when they have work to do (such as a map task in MapReduce), and they also work with node managers—which are the compute services on each of the data nodes in the Hadoop cluster—in order to monitor tasks. Figure 8 shows how the components of a job can be distributed in the cluster.

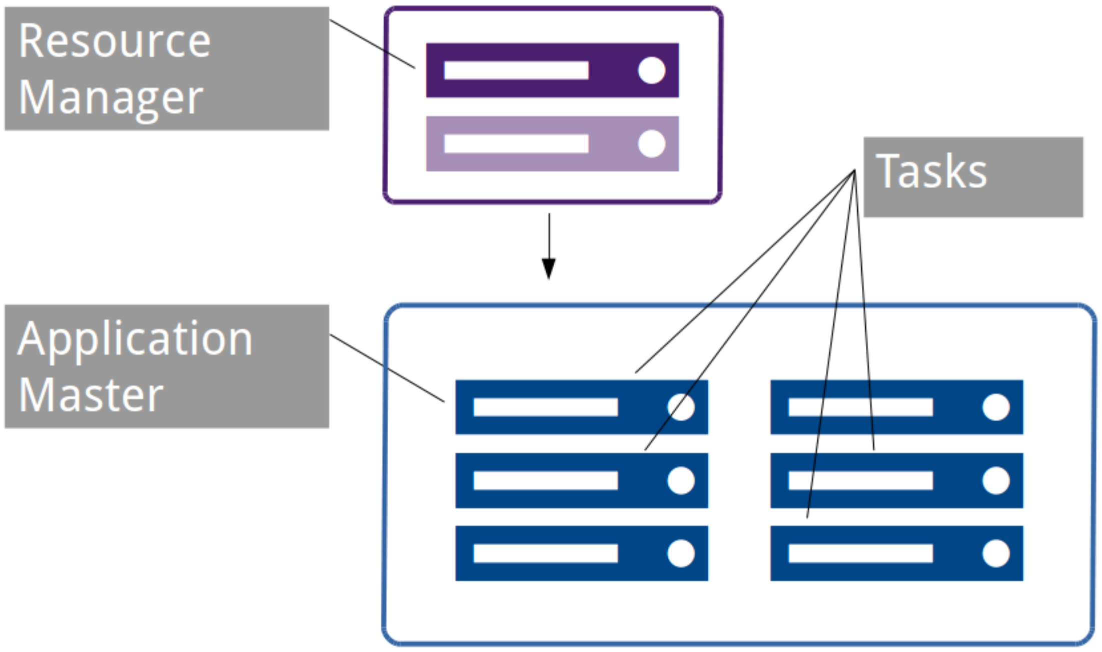


Figure 8: Distributed Job Components in YARN

Tasks are executed in containers on the data nodes. Containers are isolated units of compute with a fixed allocation of CPU and memory.

Note: YARN containers are logically similar to Docker containers, but as yet Docker is not integrated with YARN, which means tasks can't be scheduled to run as a Docker container instance.

While task containers are running, they communicate progress with the application master, and the application master communicates overall progress with the client that submitted the job. YARN permits containers to communicate using their own protocol, which allows YARN to run different types of jobs without knowing the internal workings of the framework. However, the application master must use the YARN protocol to communicate with the resource manager.

When all the task containers are complete, the application master will flag the entire job as complete to the client and to the resource manager, so that the resources used by the application master can be freed and used for other work.

A key feature of YARN is that the application master itself runs in a container that can be hosted on any node manager in the cluster so that the capacity of large clusters isn't restricted by the capacity of the resource manager (because the application masters are distributed around the cluster's worker nodes).

How MapReduce works in YARN

We've already submitted some MapReduce jobs to YARN with the **`hadoop jar`** command line. With that command, all we do is specify the JAR file containing the application we want to run and the arguments expected by our own code. That submits the job to the resource manager running on the name node, which checks capacity in the cluster and—assuming there is spare capacity—allocates a container on one of the data nodes to run the application master for the job.

The application master can start on any data node, so a registration process runs back to the resource manager, then the resource manager can inform the client where the application master is running. From then on, the client communicates directly with the application master running on the data node.

The application master starts the job, which uses its own framework to determine the compute tasks to be performed. In the case of our word count, the MapReduce framework will generate multiple map tasks—one for each input file and a single reducer task. While tasks are run, the application master makes a request to the resource manager for a new container allocation. The request can be specific about its requirements, requesting the amount of memory and CPU needed and even the preferred node name or rack name.

If there is available capacity, the resource manager identifies which node manager the application should use, and the application master communicates directly with the node to create the container and start the task. While tasks are running, the application master monitors them, and while there are more tasks queued to be scheduled, the application master keeps running.

Should any tasks fail, the application master will decide how to handle the failure. That could mean rescheduling the task via another call to the resource manager, or, in the case of multiple failures, it could mean flagging the entire job as a failure and exiting the application. In the event of failed communication between the application master and the resource manager, as in the case of data node failure, the resource manager itself can terminate the application.

However the application master ends, it is the job of the resource manager to tidy up the container allocation for the application, so that compute resource can be used for other containers such as a new application master or a task container started by another application.

Scheduler implementations

The resource manager is the central component that allocates compute resources as required. Compute resources are finite, which means there might be a point when resource requests cannot be met, and YARN has a pluggable framework that allows for different approaches to meeting requests. This is the scheduler component, which is configured at the whole-cluster level, and currently there are three implementations bundled with Hadoop.

The simplest implementation is the FIFO (First In, First Out) Scheduler that purely allocates resources based on which application asked first. If you submit a large job to a Hadoop cluster using the FIFO Scheduler, that job will use all available resources until it completes or until it reaches a point when it no longer needs all the resources.

When the application stops requesting resources and existing resources are freed up, the next application in the queue will start. Figure 9 shows how two large jobs and one small job run with the FIFO Scheduler.

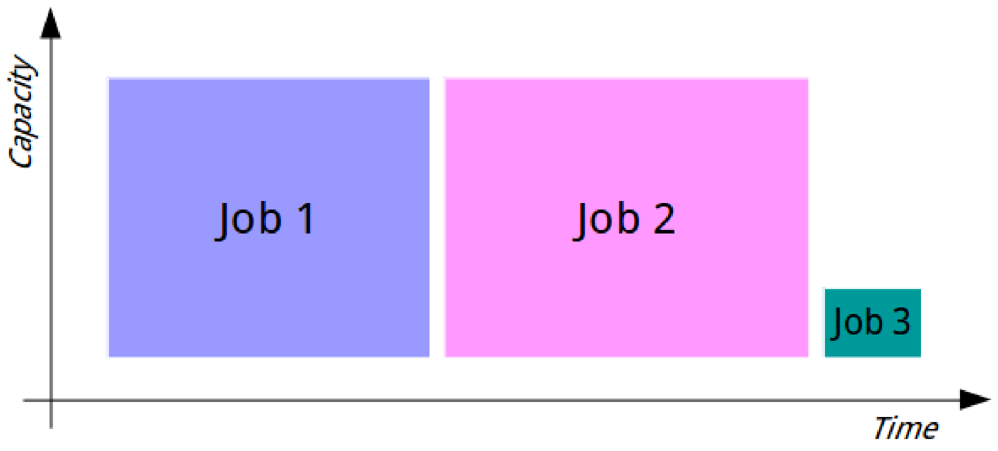


Figure 9: Job Processing with the FIFO Scheduler

With the FIFO Scheduler, large jobs all run sequentially, each one consuming the available resources until it completes. Then the next job starts and consumes all available resources. Jobs can only run concurrently if they require less than the total amount of compute available, which in practice only happens with very small jobs or with a very large cluster.

The FIFO Scheduler is crude, but it is suitable for cases in which jobs do not need to run concurrently—for example, small enterprises with batch process that must all run eventually, but in which individual jobs don't have their own Service Level Agreements (SLAs).

The other schedulers are more sophisticated, using both explicit queue configurations to allow multiple jobs to run concurrently and the cluster resources to be shared more predictably. With the Capacity Scheduler, we can define multiple queues and give each queue a fixed allocation of compute capacity. For example, we could have a cluster with three top-level queues for Marketing, Research, and Development, with a 50:30:20 split. The scheduler would ensure the entire capacity of the cluster is used as allocated, so that if one queue is empty, its share of the compute isn't reallocated and the cluster isn't fully utilized.

When the queues have outstanding work, the FIFO policy is used within the queues. That way each department (or however the top-level split is organized) has its allocated share, but within the department's queue large jobs can still clog the cluster and cause delays for smaller jobs. Figure 10 shows how the same two large jobs and one small job run using the Capacity Scheduler.

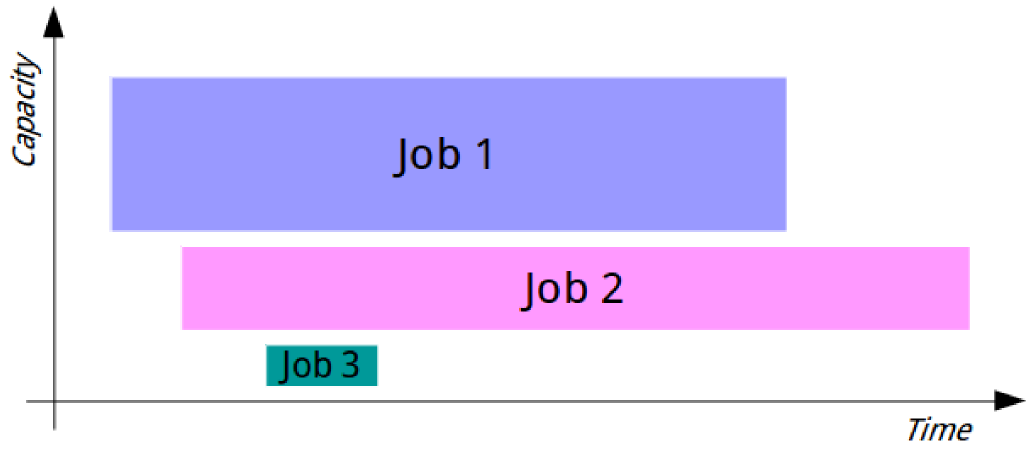


Figure 10: Job Processing with the Capacity Scheduler

In Apache Hadoop, the Capacity Scheduler is the default, but some distributions change this to use the final option—the Fair Scheduler. The Fair Scheduler attempts to fairly allocate cluster resources between all running jobs. When a single job runs with the Fair Scheduler, it gets 100% of the cluster capacity. As more jobs are submitted, each gets a share of resources as they are freed up from the original job, so that when a task from the first job completes, its resources can be given to the second job while the next task for the first job has to wait.

With the Fair Scheduler, we can address the problem of making sure small jobs don't get stuck waiting for large jobs while also making sure that large jobs don't take an excessive amount of time to run due to their resources being starved by small jobs. Figure 11 shows how the two large jobs and one small job would run under the Fair Scheduler.

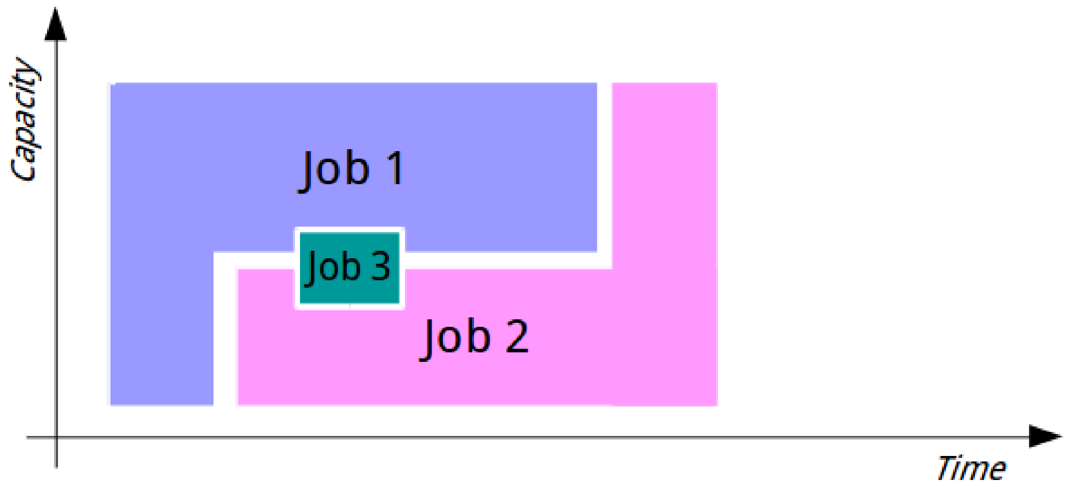


Figure 11: Job Processing with the Fair Scheduler

The Fair Scheduler can also be configured with hierarchical queues following its fair policy of running for jobs within the queues as well as between queues. In order to support complex sharing requirements, we can also define fixed allocations for queues, effectively giving them higher priority within the general fairness policy.

What is a resource?

Because YARN is a framework-independent technology, it doesn't enforce any notions of what a resource looks like or how it should be used. In MapReduce, containers are the resources requested for running specific tasks. A container can be allocated for running a mapper over a single file split. Typically, we'd aim for each mapper to complete its work and release the resources in a short timeframe—one minute is a good goal.

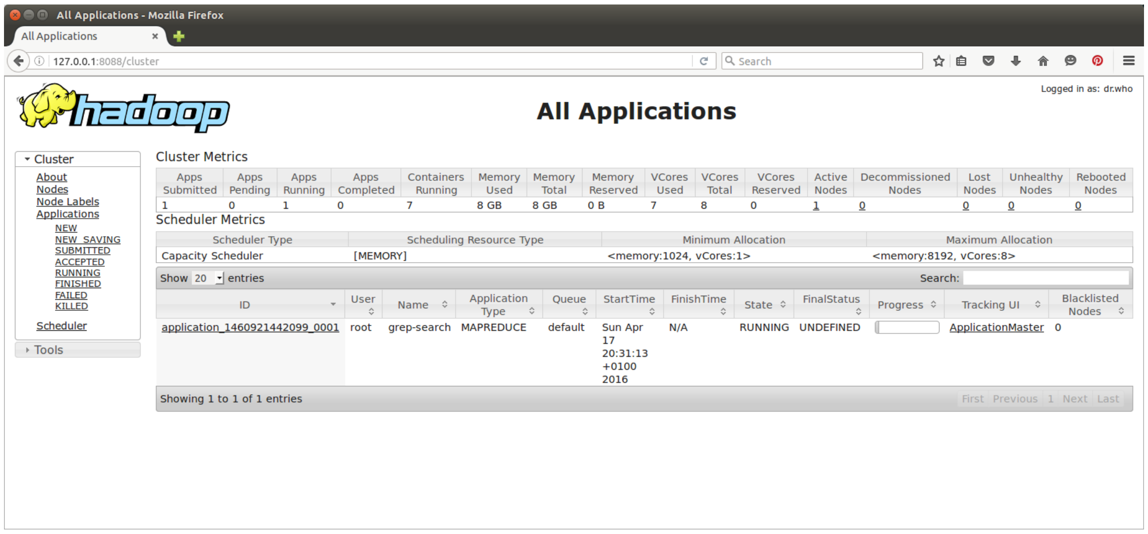
Compare that to Spark, in which a container can be used for a long-running client process (such as a user connection from a Jupyter notebook). In that case, Spark can request a resource with multiple cores and several gigabytes of RAM, and that single container may run for hours.

In the case of MapReduce, the container runs an instance of a Java VM, and by default the resources requested are a single CPU core and 1 GB of RAM. These are configurable, but be aware that when we write our mapper and reducer code, it will run in a constrained environment in which each individual task has a relatively small amount of resources allocated to it.

Monitoring jobs in YARN

Like HDFS, the various YARN services include embedded web servers that you can use to check the status of work on the cluster.

The UI runs on port 8088, so if you're using the **`hadoop-succinctly`** Docker image in Linux, you can browse to **`http://127.0.0.1:8088`** (or, if you've added the Docker machine IP address to your **`hosts`** file, you can browse to **`http://hadoop:8088`**) and see the Resource Manager UI, shown in Figure 12.



All Applications

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
1	0	1	0	7	8 GB	8 GB	0 B	7	8	0	1	0	0	0	0

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation	Maximum Allocation
Capacity Scheduler	[MEMORY]	<memory:1024, vCores:1>	<memory:8192, vCores:8>

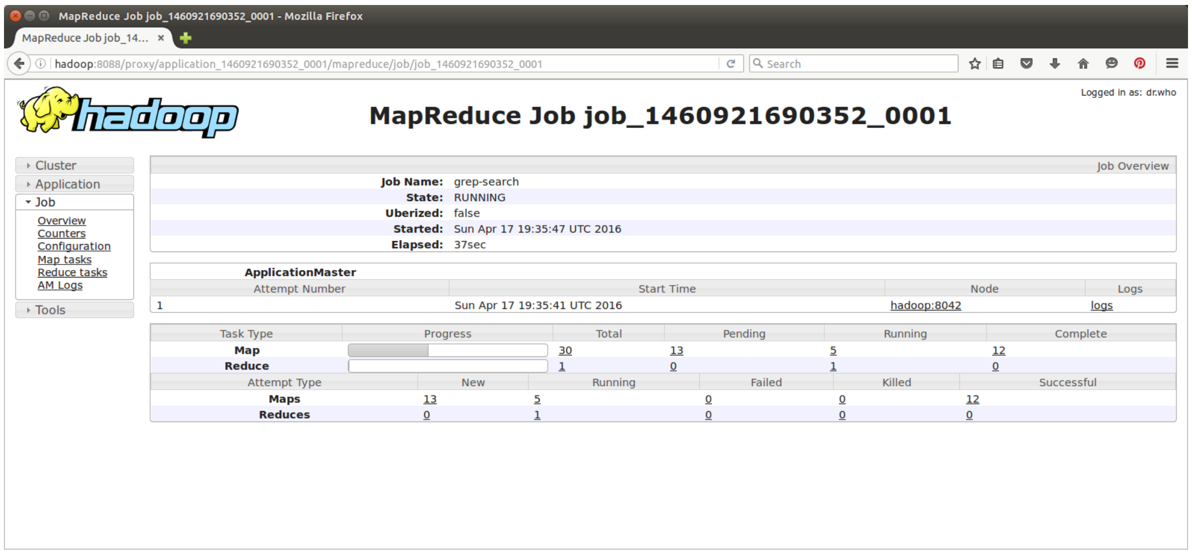
Show 20 entries   Search:

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI	Blacklisted Nodes
application_1460921442099_0001	root	grep-search	MAPREDUCE	default	Sun Apr 17 20:31:13 +0100 2016	N/A	RUNNING	UNDEFINED		ApplicationMaster	0

Showing 1 to 1 of 1 entries   First Previous 1 Next Last

Figure 12: The YARN Resource Manager UI

The default view of the resource manager is to show all running jobs with a link to their respective Application Master UI. Remember that the application master for a job runs as a container on a worker node in the cluster, but the application master communicates progress back to the resource manager. When you open the application link, you'll see the UI in Figure 13.



MapReduce Job job_1460921690352_0001 - Mozilla Firefox

hadoop:8088/proxy/application_1460921690352_0001/mapreduce/job/job_1460921690352_0001

Logged in as: dr.who

MapReduce Job job_1460921690352_0001

Job Overview

Job Name:	grep-search
State:	RUNNING
Uberized:	false
Started:	Sun Apr 17 19:35:47 UTC 2016
Elapsed:	37sec

ApplicationMaster

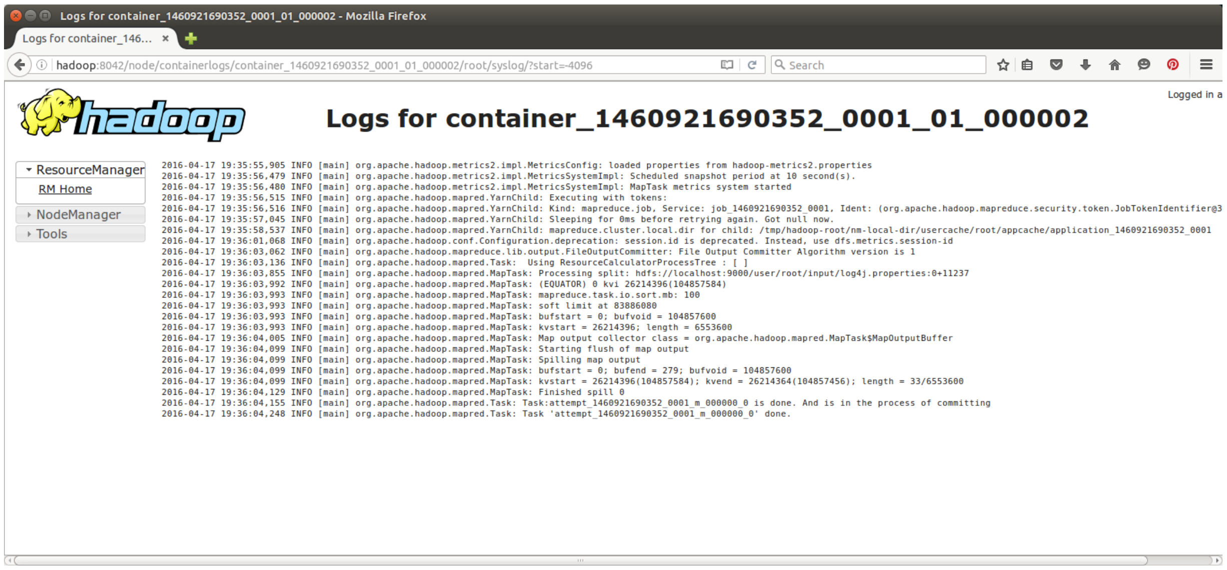
Attempt Number	Start Time	Node	Logs
1	Sun Apr 17 19:35:41 UTC 2016	hadoop:8042	logs

Task Type	Progress	Total	Pending	Running	Complete
Map		30	13	5	12
Reduce		1	0	1	0

Attempt Type	New	Running	Failed	Killed	Successful
Maps	13	5	0	0	12
Reduces	0	1	0	0	0

Figure 13: The Application Master UI

Figure 14 shows the breakdown of the work that the application runs through. In this case, it's the **`grep`** MapReduce job from Chapter 2, and the UI shows the number of map and reduce tasks by status. You can drill down for even more detail by following the links and getting a list of all tasks, then viewing the logs for a specific task, as in Figure 14.



Logs for container_1460921690352_0001_01_000002 - Mozilla Firefox

hadoop:8042/node/containerlogs/container_1460921690352_0001_01_000002/root/syslog/?start=-4096

Logged in a

Logs for container_1460921690352_0001_01_000002

```
2016-04-17 19:35:55,905 INFO [main] org.apache.hadoop.metrics2.impl.MetricsConfig: loaded properties from hadoop-metrics2.properties
2016-04-17 19:35:56,479 INFO [main] org.apache.hadoop.metrics2.impl.MetricsSystemImpl: Scheduled snapshot period at 10 second(s).
2016-04-17 19:35:56,480 INFO [main] org.apache.hadoop.metrics2.impl.MetricsSystemImpl: MapTask metrics system started
2016-04-17 19:35:56,515 INFO [main] org.apache.hadoop.mapred.YarnChild: Executing with tokens:
2016-04-17 19:35:56,516 INFO [main] org.apache.hadoop.mapred.YarnChild: Kind: mapreduce.job, Service: job_1460921690352_0001, Ident: (org.apache.hadoop.mapreduce.security.token.JobTokenIdentifier@3
2016-04-17 19:35:57,045 INFO [main] org.apache.hadoop.mapred.YarnChild: Sleeping for 0ms before retrying again. Got null now.
2016-04-17 19:35:58,537 INFO [main] org.apache.hadoop.mapred.YarnChild: mapreduce.cluster.local.dir for child: /tmp/hadoop-root/nm-local-dir/usercache/root/appcache/application_1460921690352_0001
2016-04-17 19:36:01,068 INFO [main] org.apache.hadoop.conf.Configuration.deprecation: session.id is deprecated. Instead, use dfs.metrics.session-id
2016-04-17 19:36:03,062 INFO [main] org.apache.hadoop.mapreduce.lib.output.FileOutputCommitter: File Output Committer Algorithm version is 1
2016-04-17 19:36:03,136 INFO [main] org.apache.hadoop.mapred.Task:  Using ResourceCalculatorProcessTree : [ ]
2016-04-17 19:36:03,855 INFO [main] org.apache.hadoop.mapred.MapTask: Processing split: hdfs://localhost:9000/user/root/input/log4j.properties:0+11237
2016-04-17 19:36:03,992 INFO [main] org.apache.hadoop.mapred.MapTask: (EQUATOR) 0 kvi 26214396(104857584)
2016-04-17 19:36:03,993 INFO [main] org.apache.hadoop.mapred.MapTask: mapreduce.task.io.sort.mb: 100
2016-04-17 19:36:03,993 INFO [main] org.apache.hadoop.mapred.MapTask: soft limit at 83886080
2016-04-17 19:36:03,993 INFO [main] org.apache.hadoop.mapred.MapTask: bufstart = 0; bufvoid = 104857600
2016-04-17 19:36:03,993 INFO [main] org.apache.hadoop.mapred.MapTask: kvstart = 26214396; length = 6553600
2016-04-17 19:36:04,005 INFO [main] org.apache.hadoop.mapred.MapTask: Map output collector class = org.apache.hadoop.mapred.MapTask$MapOutputBuffer
2016-04-17 19:36:04,099 INFO [main] org.apache.hadoop.mapred.MapTask: Starting flush of map output
2016-04-17 19:36:04,099 INFO [main] org.apache.hadoop.mapred.MapTask: Spilling map output
2016-04-17 19:36:04,099 INFO [main] org.apache.hadoop.mapred.MapTask: bufstart = 0; bufend = 279; bufvoid = 104857600
2016-04-17 19:36:04,099 INFO [main] org.apache.hadoop.mapred.MapTask: kvstart = 26214396(104857584); kvend = 26214364(104857456); length = 33/6553600
2016-04-17 19:36:04,129 INFO [main] org.apache.hadoop.mapred.MapTask: Finished spill 0
2016-04-17 19:36:04,155 INFO [main] org.apache.hadoop.mapred.Task: Task:attempt_1460921690352_0001_m_000000_0 is done. And is in the process of committing
2016-04-17 19:36:04,248 INFO [main] org.apache.hadoop.mapred.Task: Task 'attempt_1460921690352_0001_m_000000_0' done.
```

Figure 14: Viewing Task Logs in the YARN UI

The container logs are viewed from the node manager web server running on the node that hosted the container, which means in a cluster that will be a different machine from the resource manager. In our pseudo-distributed Docker runtime, the node manager is running on the same machine, but it uses the standard port 8042 instead of 8088 for the resource manager.

With the resource manager and node manager web servers, you will also get a lot of detail about the setup of the machine and of Hadoop. You can view the hardware allocation for nodes, check the Hadoop service logs, and view metrics and configuration all from the web UIs—they are worth getting to know well.

Summary

In this chapter, we looked at the compute part of Hadoop, learning how YARN works and about the key parts of resource negotiation in Hadoop. YARN is a generic job scheduler that supports different application frameworks—primarily MapReduce but also newer application types such as Spark and Tez. YARN allows applications to request resources that it will fulfill based on the capacity of the cluster, and it lets the applications determine which resources they need.

The flexibility and scalability of YARN come from the master/slave framework and the separation of concerns. On the master node, the resource manager's role is to accept job requests from clients, start jobs by allocating an application master, and respond to resource requests from running applications.

Application masters run in resource containers (not Docker containers!) on data nodes, so that the management overhead in YARN is distributed throughout the cluster. YARN can be configured to use various scheduling policies when resources are requested, which means you can set up your cluster to use a basic FIFO policy, a fixed-capacity policy, or a fair policy that shares compute as fairly as possible between active jobs.

We also noted briefly that the YARN resource manager and node manager services have embedded web servers. The web UIs tell us a lot of information about the nodes in the cluster and the work they have done, which means they are useful for administration and troubleshooting.

Now that we have a better understanding of how Hadoop works, next we'll look at alternative programming options for MapReduce.

Chapter 5 Hadoop Streaming

Introducing Hadoop Streaming

Although MapReduce and Hadoop itself are native Java platforms, there is support for building MapReduce components in other languages. This is called Hadoop Streaming, and it is a simple approach in which Hadoop invokes an executable as part of a task rather than hosting a JAR file in a JVM.

Hadoop Streaming uses the standard input and output streams to communicate with the executing process, so it is suitable for any platform that can build an executable binary that reads from **stdin** and writes to **stdout**. Although simple, Hadoop Streaming is a powerful technique that greatly extends the reach and flexibility of MapReduce.

Streaming allows you to use MapReduce without having to write Java code for mappers and reducers. This is particularly attractive for users who already have an investment in non-Java platforms. If you have a library of custom analytical code written in Python, or if you want to write MapReduce code in .NET, you can do that with Hadoop Streaming. You can even mix and match, using a Java mapper with a C++ reducer or an R mapper with a Python reducer.

Streaming jobs in Hadoop

A streaming job is a MapReduce job, but with a different execution mode for the job tasks. It is still submitted to YARN from a client and still follows the YARN architecture with an application master starting to manage the job. Hadoop ships with a standard JAR as the driver program, and we pass the **`hadoop`** command the details of the executables to which we want to stream.

When task containers run, a Java VM spawns the executable process. Communication back to the application master is done in the Java VM, which acts as a bridge between Hadoop and the streaming executable.

The **`hadoop-streaming.jar`** takes four arguments (as a minimum; the job can be configured with more arguments), specifying the input and output directories and the executables to run for the mapper and reducer. Code Listing 20 shows a valid Hadoop Streaming job submission that uses standard Linux shell commands for the mapper and reducer.

Code Listing 20: Hadoop Streaming Using System Commands

```
hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-2.7.2.jar \

 -input input \

 -output output4 \

 -mapper /bin/cat \

 -reducer /usr/bin/wc
```

This command runs a MapReduce program with the **`cat`** command as the mapper, which will concatenate all the input lines into one output, and with the **`wc`** command as the reducer, which runs a word count over the intermediate output.

Using MapReduce for this means all the input files will be combined by the mappers and presented to the reducer as a single input. The reducer then runs the count over the combined file. Code Listing 21 shows the output from the streaming job.

Code Listing 21: Output of the Streaming Job

```
# hadoop fs -cat output4/*

   2022    7600   76858
```

The **`wc`** command writes output in a fixed order, showing the line count, word count, and total byte count for the input, and here we have 2,022 lines containing 7,600 words in 75KB of storage. This is a trivial example, but it shows the power of streaming—here we have some real analysis generated by Hadoop and we did not have to write any code at all.

The streaming interface

Hadoop Streaming uses a very simple interface to communicate with executable mappers and reducers. Remember that all data is transferred through Hadoop as key-value pairs, and this applies to streaming applications, too. Input is fed from Hadoop to the executable in a series of tab-separated lines where:

- The string before the first tab character is the key.
- The string after the first tab character is the value.
- Each key-value pair is sent to **stdin** as a single line.
- An empty string sent to **stdin** represents the end of the input.

Hadoop collects output from the executable using the same protocol with each line in **stdout** read as a new, tab-separated key-value pair. The executable can also write to the error stream (**stderr**) to update job counters or status information.

For mappers, the streaming interface is the same as with the MapReduce API—the input contains a key-value pair. For reducers, the interface is different—in MapReduce, the reducer is called with a key and an array of values, but in the streaming API, the array is flattened, which means the executable is called with the same key multiple times with a single value from the array in each call.

In the rest of the chapter, we'll see how to run our simple word count from Chapter 2 as a Hadoop Streaming job.

Streaming word count in Python

Python is a great option for Hadoop Streaming jobs. It's a popular language among engineers and analysts, it's cross-platform, and, in most operating systems, it's a standard install. You can reasonably expect it will be pre-installed on your cluster, and you can also run a simplified version of the MapReduce pipeline locally to verify your scripts.

Python is also a very concise language. The full source code is on GitHub in sixeyed/hadoop-succinctly/python. Code Listing 22 shows the Python mapper in its entirety.

Code Listing 22: Word Count Mapper in Python

```
#!/usr/bin/env python

import sys

for line in sys.stdin:
  line = line.strip()
  words = line.split()
  for word in words:
    if 'dfs' in word:
      print '%s\t%s' % (word, 1)
```

This script loops while there is input to **stdin**—the input will be fed from the Java VM that bridges the mapper task with the Python runtime. For each line, we clear any leading or trailing whitespace, then split the line into words. For each word, if it contains the string "dfs," we print it to **stdout**.

Writing the tab-separated output to **stdout** is the equivalent of writing to Context in the Java MapReduce—it writes intermediate output that will be fed into the reducer.

Streaming reducers are a little more involved than Java reducers because the input comes as individual key-value pairs rather than a key with collection of values. But the Hadoop guarantee that data entering the reducer will be sorted still holds true, so streaming reducers know that when a key changes, the input is complete for the previous key.

Code Listing 23 shows the reducer code in Python.

Code Listing 23: Word Count Reducer in Python

```
#!/usr/bin/env python

import sys

last_match = None
total_count = 0

for line in sys.stdin:
    line = line.strip()
    match, count = line.split('\t')
    count = int(count)

    if not last_match:
        last_match = match

    if last_match == match:
        total_count += count
    else:
        print '%s\t%s' % (last_match, total_count)
        total_count = 1
        last_match = match

print '%s\t%s' % (last_match, total_count)
```

The code loops through **`stdin`**, but because it is a reducer, it will be receiving the intermediate output from the mapper. The reducer is stateful, and it will increment the running total while it receives input for a particular key. When the key changes, the reducer writes out the total for the previous key and resets its state.

By chaining together Linux **`cat`** and **`sort`** commands with calls to the Python scripts, we can mimic the behavior of Hadoop and verify that the scripts work correctly. Python is installed on the *Hadoop Succinctly* Docker container, and the Python scripts for this chapter are available in the /python directory. We can run them using the command in Code Listing 24, which also shows the abbreviated output.

Code Listing 24: Verifying the Python Scripts

```
# cat $HADOOP_HOME/etc/hadoop/* | /python/mapper.py | sort |
/python/reducer.py

"dfs" 3

#*.sink.ganglia.tagsForPrefix.dfs=  1

...
```

To submit the Python job, we use the same Hadoop-streaming.jar archive as the driver, but we also need to use the file argument in order to specify the paths to the scripts we want to run. Hadoop will copy those files to HDFS so that they are available to any node that runs a task. Code Listing 25 shows the full streaming job submission.

Code Listing 25: Submitting the Streaming Python Job

```
hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-2.7.2.jar \

 -input input \

 -output output4 \

 -mapper mapper.py \

 -reducer reducer.py \

 -file /python/mapper.py \

 -file /python/reducer.py
```

In the **jar** command, we specify an output directory, which is where the results from the reducer will be written. As before, we can view the combined results using Hadoop **fs –cat**, as shown in Code Listing 26.

Code Listing 25: Results of the Streaming Python Job

```
# hadoop fs -cat output4/*

"dfs" 3

#*.sink.ganglia.tagsForPrefix.dfs=  1

#dfs.class=org.apache.hadoop.metrics.file.FileContext 1
```

Streaming word count in .NET

For .NET programmers, Microsoft has provided a library that mimics the Hadoop Java API and lets us build MapReduce jobs in .NET using similar constructs to those used in Java. There are base classes for mappers and reducers, and we can work in a type-safe manner with our key and value pairs.

However, this is only a syntactic wrapper around Hadoop Streaming—ultimately a .NET project is built as an executable that reads and writes through standard input and output streams in the same way as a normal console application. The .NET API also hasn't been refreshed for a while, which means you must decide if taking a dependency on an older API is worth the value you get from wrapping the streaming interface.

In this chapter, we'll stick with Hadoop Streaming and the .NET source code on GitHub in sixeyed/hadoop-succinctly/dotnet, which has two console apps—one for the mapper and one for the reducer. The code is fundamentally the same as the Python variant—an instance of the mapper will be created for each task and fed lines from the input file into **stdin**.

The **`Main()`** method in the mapper class loops through **stdin** while there are more lines, as with the code in Code Listing 26.

Code Listing 26: Reading Input in the .NET Mapper

```
        static void Main(string[] args)
        {
            string line;
            while ((line = Console.ReadLine()) != null)
            {
                line = line.Trim();
                //...
            }
        }
```

When a nonempty line is received, the mapper splits the input on the space character and checks the elements for words containing the search string, as in Code Listing 27.

Code Listing 27: The .NET Mapper

```
        var words = line.Split(' ');
        foreach (var word in words)
        {
            if (word.Contains("dfs"))
             {
                Console.WriteLine(string.Format("{0}\t1", word));
            }
        }
```

Emitting a key-value pair to the context is a case of formatting a string with the key and value separated by a tab character and writing it with **`Console.WriteLine()`**.

The reducer class works in the same way, with a **`Main()`** method that loops through the console input stream. The overall input to the reducer will be the same sorted and merged input that Hadoop would provide to a Java reducer, but it will be flattened so that each item in the value collection causes another line of input to the reducer.

Code Listing 28 shows the **`Main()`** method for the reducer in which the read loop is the same as the mapper, but we initialize some variables to maintain state in the loop.

Code Listing 28: Reading Input in the .NET Reducer

```
        static void Main(string[] args)

        {

            string lastMatch = null;

            int totalCount = 0;

            string line;


            while ((line = Console.ReadLine()) != null)

            {

                line = line.Trim();

                //...

        }
```

In the read loop, the functionality is the same as in the Python script—keep a running count for each key and when the key changes, emit the previous count and reset it. Code Listing 29 shows the .NET reducer implementation.

Code Listing 29: The .NET Reducer

```
var parts = line.Split('\t');

var match = parts[0];

var count = int.Parse(parts[1]);

if (lastMatch == null)

{

    lastMatch = match;

}

if (lastMatch == match)

{

    totalCount += count;

}

else

{

    Console.WriteLine(string.Format("{0}\t{1}", lastMatch, totalCount));

    totalCount = 1;

    lastMatch = match;

}

Console.WriteLine(string.Format("{0}\t{1}", lastMatch, totalCount));
```

In order to run .NET programs, you either need a Windows machine with the .NET Framework installed or you can use the cross-platform .NET Core on any runtime. Running on a Windows cluster, you submit the streaming program to Hadoop in the usual way, specifying the name of the executable programs you want to run and shipping the binaries with the file argument.

Code Listing 30 shows the submit command and abbreviated results from the .NET Streaming job.

Code Listing 30: Submitting the Streaming .NET Job

```
hadoop jar %HADOOP_HOME%\share\hadoop\tools\lib\hadoop-streaming-2.7.2.jar \
 -mapper "c:\dotnet\Sixeyed.HadoopSuccinctly.Streaming.Mapper.exe" \
 -reducer "c:\dotnet\Sixeyed.HadoopSuccinctly.Streaming.Reducer.exe" \
 -input input \
 -output output4
```

Interoperability with Hadoop Streaming

With the streaming interface, we build the same type of MapReduce components using the language of our choice. There's no need for an explicit driver, because the **`hadoop-streaming.jar`** provided by the Hadoop installation acts as the driver. We can also write mappers and reducers in the language of our choice, and we can include whichever unit-testing tools we normally use.

Because streaming executables are simply console apps, we can also run integration tests using a subset of the input data—feeding a single file into the mapper and capturing the output, which will feed as the input for the reducer. And, because we incur the cost of running the Java VM that bridges between Hadoop and our process, a decision to use streaming will include a performance consideration.

However, be careful with interoperability in streaming apps. The protocol between Hadoop and your executable always passes data as strings, so if you are trying to mix a non-Java mapper with a Java reducer, you must be sure your mapper serializes the key and value output in the expected format so that the reducer can deserialize.

Managing dependencies in Hadoop Streaming

Another important consideration for streaming apps is ensuring that the executable can run on the data nodes. Typically, that means setting up the executable platform during the commissioning of your servers—if you know you'll be using Python or the .NET Framework, you need to have the required version installed as part of your setup.

If you use additional libraries in your executable programs, you will need to make sure they'll be available on the node where the task runs. You can use the same dependency-shipping approach that Hadoop provides for distributing JAR libraries with Java MapReduce programs. The **`hadoop-streaming.jar`** supports the **`archive`** argument that lets you specify a local package to ship along with your MapReduce program.

If your executable has library dependencies, you can bundle them into a ZIP package and specify that ZIP file in the **`archive`** argument for your job, as in Code Listing 31.

Code Listing 31: Submitting Streaming Jobs with Dependencies

```
hadoop jar %HADOOP_HOME%\share\hadoop\tools\lib\hadoop-streaming-2.7.2.jar \
 -input input \
 -output output4 \
 -mapper /lib/mapper-with-depdencies.exe \
 -reducer /lib/reducer-with-dependencies.exe \
 -archive c:\dotnet\dependencies.zip
```

Note: You need not ship your executables for streaming jobs; you can copy them to HDFS first and specify the relative HDFS path in the *mapper* ***and*** *reducer* ***arguments, as in this example.***

When this job runs, Hadoop will copy the archive file to HDFS with a high-replication factor (a default of 10)—the aim being that it will already be local to nodes that run tasks for the job. When the task runs, it extracts the archive to the working directory for the job, so that your executable will find its dependencies in the current directory.

Performance impact of Hadoop Streaming

When a Java MapReduce task runs on a node in Hadoop, it first creates a Java VM to run the JAR file where the mapper or reducer is built. For a streaming application, there is no Java code, but a JVM is nevertheless created, and unless you're using a system command, the executable will also need to bootstrap its own runtime. The cost of loading the executable runtime and the JVM will impact the overall runtime of your job.

It can be difficult to assess that cost because you can't use typical instrumentation techniques—by the time your instrumentation code runs, the runtime has already been loaded and the cost already incurred. The difference is often unimportant because you choose to use streaming for nonfunctional reasons (having existing logic you want to re-use or having greater experience in a non-Java platform). However, it's always useful to understand the cost of making that decision.

We can get a reasonable idea of cost by benchmarking similar MapReduce programs running through different platforms. Table 2 shows the results from running the word count MapReduce through Java, Python, and .NET.

Runtime	Average Total Map Time	Average Total Reduce Time	Average Total Job Time	Difference From Java Baseline
Java	102.3	31.7	134	0

Runtime	Average Total Map Time	Average Total Reduce Time	Average Total Job Time	Difference From Java Baseline
Python	112.9	34.0	146.9	12.9
.NET	108.2	33.3	141.6	7.6

Table 2: Compute Time for MapReduce Programs

For this benchmark, I used the MapReduce programs, all of which are on GitHub, and I ran each job five times to get a reasonable average. Because the .NET program uses the full framework, it must run on Windows, so I ran all the jobs on the same single-node Syncfusion Big Data Studio installation (more on that in Chapter 7).

We can see there is an impact in using streaming compared to native MapReduce—streaming jobs take 5-10% longer, depending on the runtime. If you're considering using streaming for the bulk of your Hadoop access, that's something to be aware of. The impact is not significant enough to discount streaming, but time-critical jobs may need to be done in Java.

Summary

Hadoop is a Java platform, and although Java is the native language for running MapReduce jobs, Hadoop has strong support for any platforms that can produce an executable component. That allows you to build MapReduce programs in which the mapper and reducer are written in any language you prefer, provided you can configure your Hadoop nodes with all the prerequisites they need to use the platform.

In this chapter, we looked at how to reproduce the simple MapReduce program from Chapter 2 by using Python and the .NET Framework. The streaming interface uses a simple protocol in which data is fed into **stdin** line by line, as tab-separated key-value pairs, and Hadoop expects output to come into **stdout** using the same format.

Although you should be aware of type safety and the performance impact of Hadoop Streaming, typically both concerns are insignificant compared to the ability to leverage your preferred platform and any existing technical assets you have. Hadoop Streaming supports shipping dependencies to nodes, which means you can build complex executables for your mappers and reducers and have them run without issue on distributed nodes.

The Hadoop Streaming interface is only one aspect of the cross-platform support for Hadoop. In the next chapter, we'll look more closely at the moving parts in the Hadoop cluster. In Chapter 7, we'll see how different Hadoop distributions run on different platforms.

Chapter 6 Inside the Cluster

Introduction

So far we've examined master nodes and worker nodes at a high level. Now we're going to look more closely at the Hadoop cluster in order to get a better understanding of how to set up machines and see how they work together.

A Hadoop cluster is a dynamic part of an infrastructure. If you have analytical problems that require a Big Data solution, your data collection will be ongoing and your cluster will need to grow periodically in order to meet increased storage demands. Additionally, when the benefits of rich analysis start to be realized, many companies accelerate their Big Data program, storing more data for longer periods and requiring different types of analysis.

Hadoop is inherently scalable and also richly configurable, which means you can tune your cluster to meet changing requirements and continue to store and process all your data from a single platform.

Sizing Hadoop—minimal cluster

As you initially design a Hadoop cluster, you will need some understanding of your data requirements in order to provision the correct amount of hardware.

As an absolute minimum, Hadoop clusters used for production workloads need one master that will act as the HDFS name node and the YARN resource manager, and they will need three slaves to act as the HDFS data nodes and the YARN node managers. Figure 15 shows a minimal deployment of Hadoop in which the nodes all reside on a single rack.

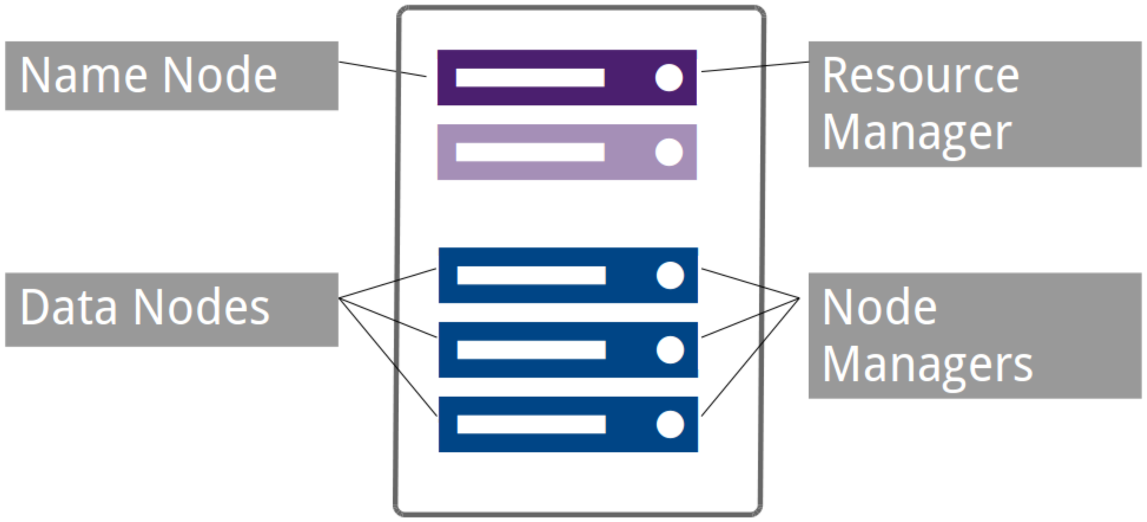


Figure 15: A Minimal Hadoop Cluster

The hardware requirements are different for the master and slave servers in these ways:

- The master node is a critical component. It needs to be reliable, so it should have hardware RAID for its storage even though it doesn't require a lot of disk space. As the name node, it maintains the HDFS file namespace (the location and metadata of all the file blocks stored on the cluster), and for performance it keeps that information in memory. As the resource manager, it’s the central point of contact for all application masters and client connections. The name node and resource manager services have multiple functions, but CPU power is not a primary factor.

- The worker nodes are not critical because Hadoop is architected with failure in mind. As HDFS data nodes, the workers should have a lot of disk storage, and RAID is not required (HDFS uses a round-robin approach to disk allocation that has benchmarked faster than RAID-0 for typical Hadoop workloads). As YARN node managers, workers should have a lot of CPU cores and a high provision of memory. If you plan to run only MapReduce applications, each task will require an allocation of one CPU core and 1 GB of RAM—figures you can use to roughly calculate your cluster's capacity.

Server specifications continuously improve, but at the time of writing a typical commodity server suitable for use as a Hadoop slave would have 12-16 CPU cores, 48-128GB of RAM, and up to 12 disks, each with up to 4 TB of storage. A reasonable budget should allow you to settle on a 16-core machine with 48 GB of RAM and 36 TB of storage.

With three slaves of the same specification, you would have a storage capacity of 36 TB (because HDFS replicates three times, every block would be replicated on every server) and a compute capacity for perhaps 36 concurrent task containers (assuming a 25% CPU overhead for running the Hadoop services, leaving 12 cores on each node for YARN to allocate).

That specification may not sound minimal if you're used to dealing with ordinary app servers or even database servers, but in Big Data terms that’s quite a limited setup. There will be enough storage to collect 100 GB of data every day for just under a year, but the compute side is more limited.

Assuming an average of 1.6 GB of data per hour, that’s 14 file blocks for each hour’s worth of data. You would use the full concurrent task capacity of the cluster with a job processing just three hours of data. If you want to run a job over an entire month's worth of data (that's 3 TB spread across 24,000 file blocks of 128 MB each), assuming each task takes 60 seconds, your job would max out the cluster for 11 hours. Analyzing the entire year's worth of data would take more than five full days.

Sizing Hadoop—the small cluster

A minimal cluster is really only a starting point, a reasonably small investment if you're evaluating Hadoop. Typically, a functional small cluster contains about 10 worker nodes. That's a comfortable amount for a single master to manage—beyond that you'd need to scale out your master services, which we'll look at shortly.

The small Hadoop cluster is likely to have servers on multiple racks, typically in a two-layer network infrastructure in which the servers have a fast network connection within the rack and an uplink switch to the rest of the network. Figure 16 shows a small cluster setup with 10 worker nodes across three racks.

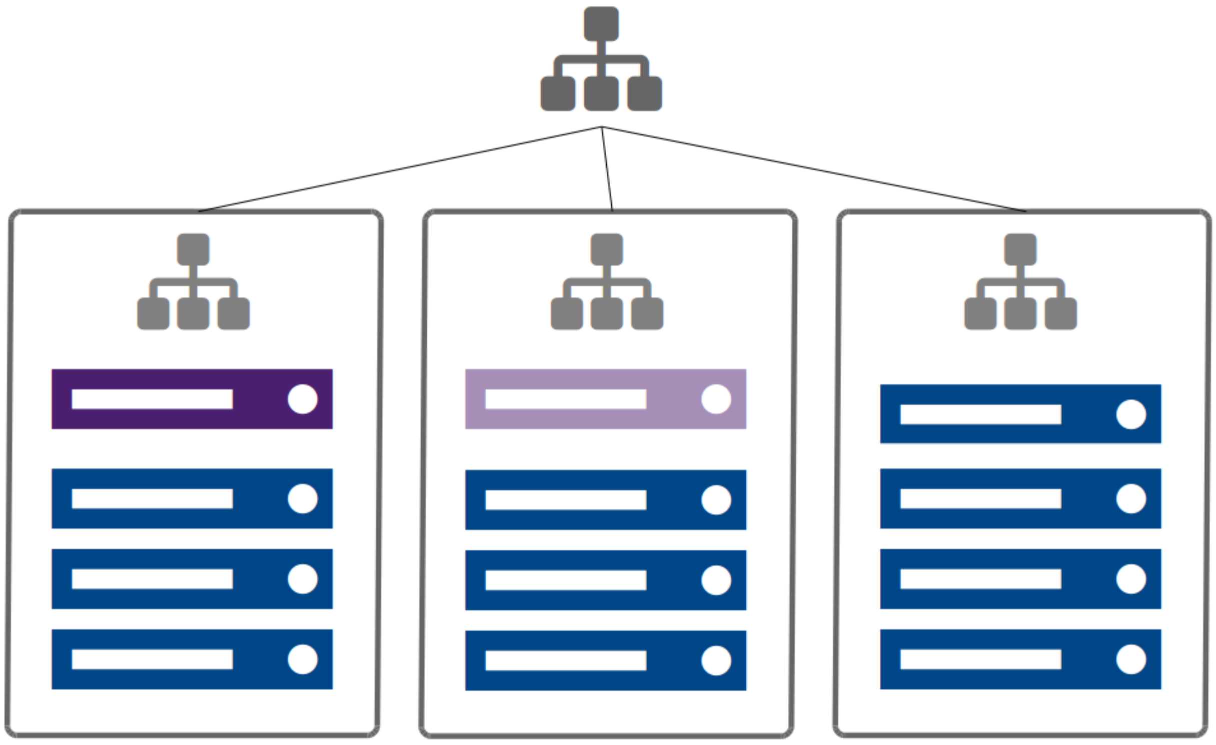


Figure 16: A Small Hadoop Cluster

Adding more slave servers gives you additional storage and compute capacity, but it's not quite as simple as a linear progression for both. If we keep our replication factor at the default of 3, adding three servers increases our storage capacity by the combined storage of the new nodes. If we added three nodes to our minimal cluster, we'd double the capacity to 72 TB, while scaling up to nine servers gives us 108 TB total storage.

Similarly, our nine-server cluster now has 144 cores, so it could execute 108 tasks simultaneously. Unfortunately, that doesn't mean our job to analyze a month's data will take one-third of the time and complete in 11/3=3.5 hours. With a replication factor of 3, and with nine servers, YARN has a smaller chance of allocating a task to a node with data locality, which means the average task time will increase as nodes are reading data over the network.

We can't predict the exact impact of losing locality, because that depends on which other jobs the cluster is running and how efficiently the scheduler works, but if we make a conservative guess that the average job time would increase from 60 seconds to 75 seconds, the monthlong job will take 4.6 hours. We've tripled our infrastructure, but the bottom-line job duration has fallen to 40% of the original time rather than 30%.

Sizing Hadoop—large clusters

In the wild, there are Hadoop deployments with thousands of data nodes that store and process petabytes of data. One of the main drivers for the re-architecture of HDFS and YARN in Hadoop 2 addressed this issue of scaling, with the goal being that limitations on cluster scale should move to extraordinarily large sizes.

From the slave side, very large Hadoop clusters have large numbers of data nodes. You can separate the storage and compute functions, so that your HDFS data nodes run on one set of machines and your YARN node managers run on a separate set, but what you gain in distribution you will lose in data locality, so this is not a common pattern (except in cloud-hosted clusters, which we'll see in Chapter 8).

From the master side, large clusters look very different because they do split out and federate the storage role. One name node cannot hold the entire file system namespace in memory if there are billions of file blocks. If you store data at the rate of 1 TB per hour, you are adding a million file blocks every week and you will quickly hit memory limitations. HDFS allows federation, which means multiple name nodes each store part of the namespace.

YARN currently doesn't support federated resource managers. A high-performance resource manager can support about 4,000 node managers, which is the current practical limit of Hadoop (designs are being discussed to support federated resource managers, which you can follow on the Apache JIRA ticket YARN-2915).

Hadoop high availability and federation is beyond the scope of this e-book, but it is a well-trodden path. The key difference with large clusters is the requirement, usually filled with Apache Zookeeper, for a fast subsystem for shared state and notifications.

Larger clusters don't only provide more storage and compute, they can also provide increased reliability because they provide more machines for HDFS and YARN to use in the case of failure.

Reliability in HDFS

Hadoop is built on the expectation that hardware will fail and that Hadoop's work will continue in the face of failure. Failure is more common in HDFS data nodes because they are more numerous and have more failure points—a server with 12 spinning disks will surely have a failure at some point. In fact, with large enough clusters, daily failures become a certainty.

Data nodes might have failed disks, they might lose network connectivity, or the server might fail altogether, but Hadoop has a simple mechanism to address all these issues—data nodes periodically send messages to the name node, which monitors them and responds to failures.

The data nodes send two types of messages—the heartbeat message, which confirms that the node is up and able to communicate, and blockreports, in which the node sends a list of every block it has stored, across all its disks. These mechanisms, shown in Figure 17, are enough for HDFS to maintain reliability.

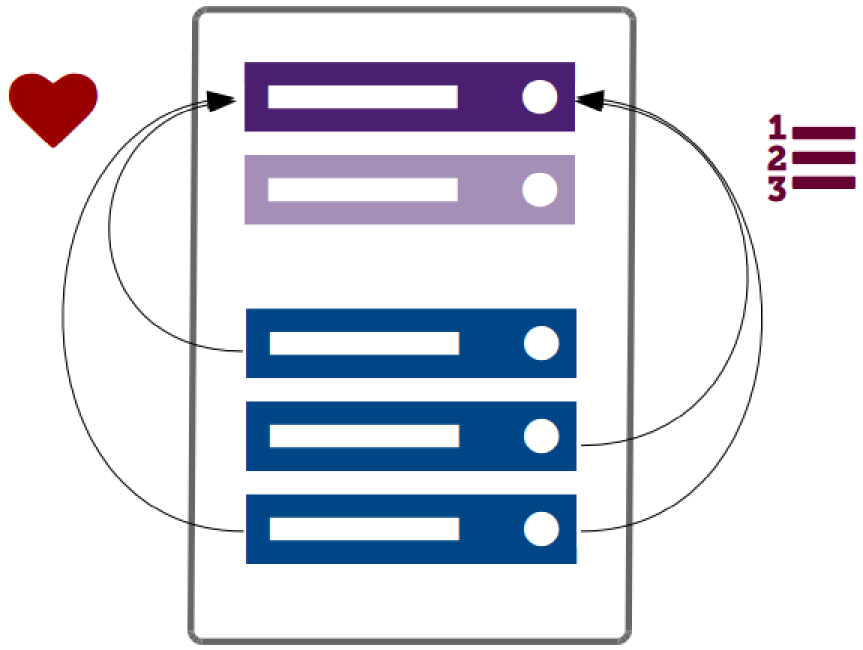


Figure 17: Heartbeat and Blockreport Messaging in HDFS

If heartbeat messages from a data node stop, the name node assumes the data node is offline and marks it as unavailable. Any entries for that node in the file system namespace are removed, which can mean files no longer have the required number of block replicas. In that case, the name node will instruct other data nodes to make replicas of all the blocks on the unavailable node, and, when the replication completes, the file system will be back to full replication.

With the default configuration, a single disk failure in a node will cause the data node service to shut down, triggering another replication by the name node. You can configure HDFS to tolerate one or more disk failures in a data node, which is a reasonable approach provided you have monitoring in place to alert you when disks fail.

HDFS endeavors to keep your data available, but it does not automatically make optimum use of space in the cluster. If a failed node is returned to the cluster or if new nodes are added, an imbalance of file blocks across the nodes will occur, and a returning node could indicate that blocks have more replicas than you need while new nodes start off empty, which means they'll have no data locality for tasks.

Hadoop has the functionality to rebalance data across the cluster without any downtime, so that client requests are still serviced. It’s designed not to be an invasive process—it isn’t compute-intensive, although rebalancing typically takes a long time.

You can run the command **`hadoop balancer`** to start rebalancing data across the cluster, but, in order to conserve cluster capacity, Hadoop uses the compute power of the machine running the balancer when deciding where blocks should move, which means you should invoke it on a fast machine with fast access to the cluster. Even so, rebalancing many terabytes of data will take days.

Reliability in YARN

Just as HDFS monitors data availability and remedies any lost blocks, YARN monitors job progress and can address issues with lost tasks and failed jobs. While tasks run, they are the responsibility of the application master, and they send heartbeat messages in the same way that data nodes send heartbeats to the name node.

If a task fails unexpectedly, or if the heartbeat stops, the application master marks the task as failed and requests another container from the resource manager—which is ideally running on a different node than where the task failed—to repeat the failed task.

YARN lets you configure how many retries are permitted for map and reduce tasks. The defaults allow four task retries, but if any one task does fail four times, the application is flagged as a failure. That is also configurable, which means you can set up YARN to allow a certain percentages of task failures while keeping the job running.

Although the responsibility of the resource manager, running applications also send heartbeat messages. If an application master ends in a failed state or stops sending heartbeats, the resource manager ends the application and will retry it with a new application master. The number of retries for an application is also configurable, which means YARN allows for a lot of tuning to ensure that transient failures don't stop job completion and that a bad job (which could be down to defects in the code or the data) doesn't clog up the cluster.

Cluster configuration

We've seen several points where Hadoop's behavior is configurable, and there are a huge number of switches we can tweak to alter the characteristics of the cluster.

Hadoop is configured using XML files that are present on each node. Configuration files are in the **`/etc/hadoop`** directory within the Hadoop installation folder (which is typically referenced with the **`HADOOP_HOME`** environment variable, e.g., **`$HADOOP_HOME`** in the **`hadooop-succinctly`** Docker image). Typically, there are four files in which we configure settings, and we only need to specify settings where we wish to override the default values.

There are far too many options to cover here, but we'll look at the main configuration files and the specific setup used in the **`hadoop-succinctly`** container. The default settings are typically suitable for a basic development environment, which means there are some settings you always override.

core-site.xml

These are core, Hadoop-wide settings used by HDFS, YARN, the web servers, and the APIs. The default configuration for the relevant version is in the Hadoop documentation—for 2.7.3 these are the core-default.xml settings.

This file configures key settings such as the security used by Hadoop and the default file system (Hadoop can use file systems other than HDFS, but only HDFS provides the performance boost of data locality). The default is the operating system's file system, which means we need to override that in order to use HDFS. Code Listing 32 shows the setup for the Docker container.

Code Listing 32: Specifying the File System in core-site.xml

```
<configuration>

    <property>

        <name>fs.defaultFS</name>

        <value>hdfs://localhost:9000</value>

    </property>

</configuration>
```

This is the entire **core-site.xml**, which means we specify only one property—all the other settings come from the default file. We specify **fs.defaultFS**, which is the default file system for Hadoop, and it is specified in terms of the base URI for the provider—in this case, an HDFS path on port 9000 of the local host (which is the Docker container).

hdfs-site.xml

This file is for site-specific overrides for properties that configure HDFS. The available properties and default values are listed in the hdfs-default.xml documentation.

HDFS configuration properties let you specify the ports the services should listen on, whether or not data transfer should be encrypted, and the behavior of the file cache. For the Docker container, we only override one property, as seen in Code Listing 33.

Code Listing 33: Specifying the Replication Factor in hdfs-site.xml

```
<configuration>

    <property>

        <name>dfs.replication</name>

        <value>1</value>

    </property>

</configuration>
```

The **`dfs.replication`** property specifies the default replication factor for file blocks in HDFS. The default factor here is 3, but our container only runs a single data node service. When Hadoop starts up, it uses a special safe mode while it checks for file system consistency. If it expects three block replicas but only one data node ever reports back, Hadoop won't leave safe mode, and the cluster is effectively read-only.

Specifying a replication factor of 1 allows the pseudo-distributed cluster on the container to run correctly with a single data node process.

mapred-site.xml

This file configures the behavior of MapReduce jobs. In v1 versions of Hadoop, this file configured the behavior of the MapReduce engine as well as the actual jobs, but with YARN, the runtime behavior lives in a separate file. The default property values are specified in the mapred-default.xml documentation.

In the mapred-site.xml, we can specify values such as the default number of reducers used for each job, whether or not the data output from mappers should be compressed, and the type of sort to perform on map keys. In the Docker setup, we have overridden one property, as in Code Listing 34.

Code Listing 34: Specifying the MapReduce Engine in mapred-site.xml

```
<configuration>

    <property>

        <name>mapreduce.framework.name</name>

        <value>yarn</value>

    </property>

</configuration>
```

This specifies which engine to use for MapReduce jobs. Because YARN introduced a very different model of computation in Hadoop v2, the default is left to use the original MapReduce v1 engine, so that when old clusters upgrade, they won't unknowingly switch to a new engine that is potentially incompatible.

yarn-site.xml

The last file we'll look at, yarn-sit.xml, configures the behavior of YARN. As usual, the available properties and default settings are listed in the yarn-default.xml documentation.

We can configure a lot of low-level settings here, such as the heartbeat interval for application masters to send to the resource manager, the type of scheduler to use, and the minimum and maximum CPU core and memory allocation requests YARN will allow. The Docker container specifies two properties here, as in Code Listing 35.

Code Listing 35: Specifying the Shuffle Service in yarn-site.xml

```
<configuration>

    <property>

        <name>yarn.nodemanager.aux-services</name>

        <value>mapreduce_shuffle</value>

    </property>

    <property>

            <name>yarn.nodemanager.aux-
services.mapreduce.shuffle.class</name>

            <value>org.apache.hadoop.mapred.ShuffleHandler</value>

      </property>

</configuration>
```

Between MapReduce v1 and YARN, there was a change as to how the output from mappers got sorted before being sent to the reducer(s). The sorting of keys is the shuffle phase, and in MapReduce v1, the shuffle implementation was fixed. YARN allows for a pluggable shuffle framework, defined as an auxiliary service running on the node managers, and in this configuration we specify the standard **`ShuffleHandler`** class as the implementation of the **`mapred_shuffle`** service.

Summary

In this chapter, we looked more closely at the details of the Hadoop cluster. We examined both the hardware requirements for master and slave nodes and how Hadoop looks in clusters of different sizes. Because Hadoop clusters are capable of expanding gracefully, a good approach to sizing a new implementation is to estimate the rate of data capture and plan for a cluster that has enough capacity for a known period.

We saw how Hadoop nodes communicate among themselves, sending heartbeats and status reports, and we looked at how that facilitates reliability in the cluster. With HDFS, the loss of a data node triggers additional replication of data among the remaining nodes. With YARN, the loss of a node manager means the tasks and applications running on that node will be rescheduled on the cluster.

Lastly, we looked at the configuration framework in Hadoop and saw some of the properties we can override. All the key decision points in the operation of Hadoop have settings we can specify, which means we can tune our cluster to operate most efficiently for our requirements.

Deploying and configuring a Hadoop cluster is not a trivial task, and there are many commercial products that wrap Hadoop in customized and supported packages. Next, we'll look at some of the popular Hadoop distributions.

Chapter 7 Hadoop Distributions

Introducing distributions

As Hadoop becomes increasingly prevalent across enterprises, the commercial ecosystem around Hadoop continues to grow. Commercial distributions of Hadoop typically bundle the core platform with additional Big Data technologies aimed at providing a single, easy-to-use platform for the entire data analysis landscape.

Commercially available distributions fall into two options—packaged solutions that we deploy on-premise and pay for either in terms of a support subscription or a product cost, and hosted solutions that run in the Cloud and that we pay for in terms of the hourly (or per-minute) compute cost of the cluster and the storage we use.

In this chapter, we'll look at the major Hadoop distributions to see what they offer over the core Apache offering and how they differentiate themselves in the market.

Cloudera

Cloudera was the first commercial provider of Hadoop, and its CEO, Doug Cutting, was the founder of Hadoop’s original project. Cloudera distribution, including Apache Hadoop (CDH), is a packaged Hadoop installation based on the core HDFS and YARN frameworks, and it also includes a broad selection of tools from the Hadoop ecosystem, including HBase and Hive.

The main differentiator for Cloudera over other distributions is the speed at which it adopts new developments in the Hadoop world. Cloudera engineers are actively involved in Apache Big Data projects, and the company is happy to promote early stage technology into its platform.

As the oldest commercial Hadoop provider, with hundreds of large-scale customer deployments, Cloudera has extensive experience in configuring Hadoop, and Cloudera Enterprise ships with default configurations that are better suited to large deployments and require less tuning than a standard Hadoop install.

CDH is free, but Cloudera Enterprise, a commercial offering, adds support and additional features, such as proactively monitoring your Hadoop cluster and comparing it to other customers' clusters. Cloudera uses that insight to power its predictive maintenance component, which can alert you to problems with your nodes before they have a serious impact.

MapR

MapR takes a very different approach for its Hadoop installation, using its own components to provide an optimized distribution rather than packaging the pure Apache components. At the core, their product uses a custom storage layer that is binary-compatible with HDFS but has a different implementation.

The file system MapR-FS exposes a Network File System (NFS) interface, so that clients can connect to the storage layer as a mapped drive, but it’s compatible with HDFS. The file system uses a distributed master for the metadata store, which means it can scale to larger sizes than HDFS and doesn't have the same single point of failure.

The MapR distribution bundles other key tools from the ecosystem but typically reworks and rebrands them, so that the Big Data-scale queue technology MapR Streams is compatible with Apache Kafka, and, for real-time access, MapR-DB is compatible with HBase.

All the MapR customized components are API-compatible with their Apache versions, but they are tuned for performance and scale at the enterprise level. MapR has its own management portal, the MapR Control System, which lets you configure the system as well as monitoring nodes and getting warning alerts.

Hortonworks

Hortonworks provides the most Hadoop-faithful of the commercial distributions. All of the included components are official Apache versions, which means the Hortonworks Data Platform (HDP) product is primarily about packaging, configuring, and supporting open source components.

The open source community has benefitted greatly from Hortonworks' approach. When there have been gaps in its product offering, rather than build a proprietary component, Hortonworks has ramped up support for Apache projects and even gone so far as to buy other companies and donate that technology to Apache. Two of the key open source tools around Hadoop are through Hortonworks—Ambari, for cluster management, and Ranger, for securing Hadoop.

The Hortonworks data platform uses Apache Ambari for configuration and monitoring of the cluster (which we’ll see in Chapter 8), and the packaged product includes HBase, Spark, Hive, etc. Typically, Hortonworks are slightly more conservative than Cloudera with their adoption of new releases, so the versions of components in HDP can be behind Cloudera (at the time of writing, Cloudera’s CDH 5.7 bundles HBase version 1.2.0, while HDP 2.4.0 has HBase 1.1.2).

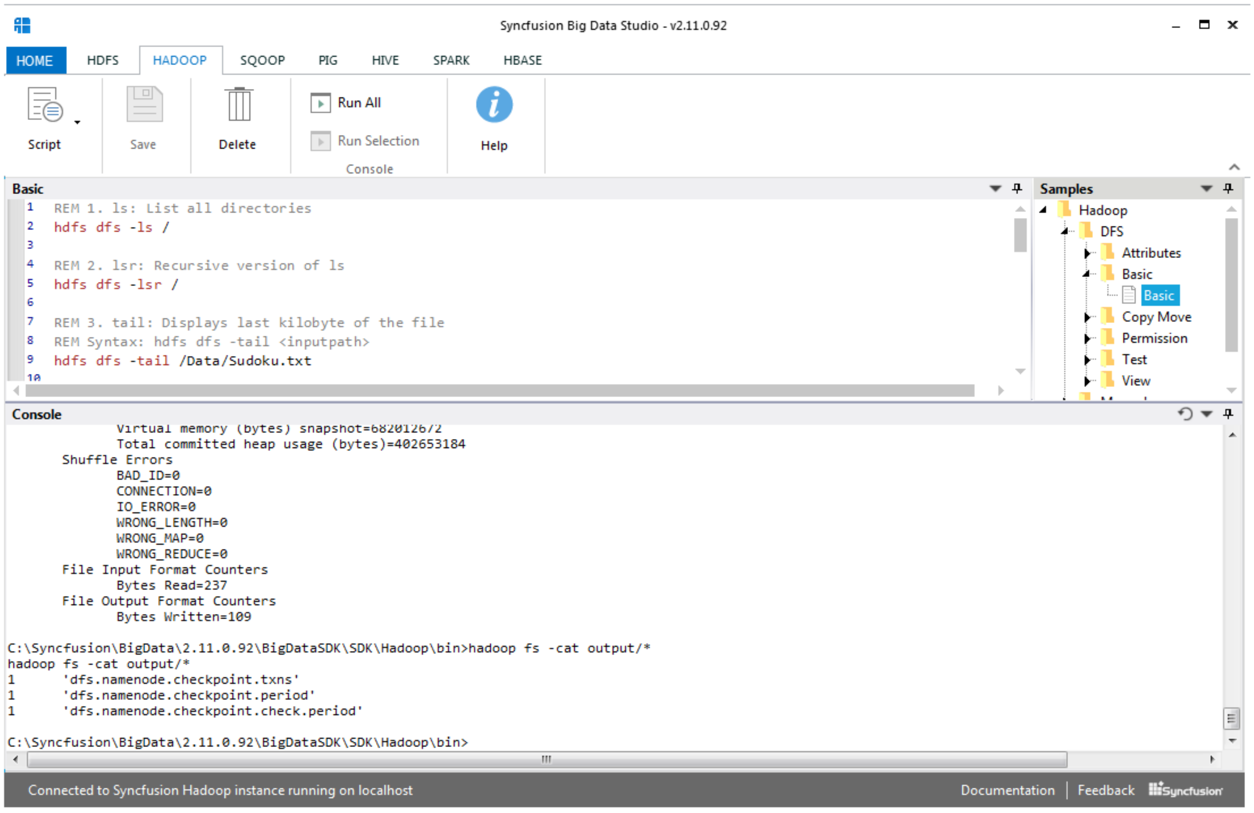
HDP is distinct from the other leading distributions because it provides a variant that runs on Microsoft Windows. Although Hadoop is Java-based and is technically platform independent, the vast majority of deployments currently run on Linux platforms. With HDP, Hortonworks provides a version of Hadoop that will appeal to the many Microsoft clients now considering Hadoop.

Syncfusion and others

Syncfusion's Big Data platform is a relative newcomer to the market, but it has some key differentiators that make it an attractive option. First, it is Windows-native, which means it is a version of Hadoop specifically built to support Microsoft technologies.

The Big Data platform lets you run as a single node or connect to a cluster running on-premise or in the Cloud. The platform comes with a custom management portal that lets you submit different types of jobs to Hadoop, and the distribution includes sample jobs written in C#, so that Microsoft .NET becomes a first-class citizen.

Part of Syncfusion's platform is the Big Data Studio, which gives you a nice front-end for navigating Hadoop and other parts of the ecosystem—we see the result from running the word count query in the Studio in Figure 18.



```
REM 1. ls: List all directories
hdfs dfs -ls /

REM 2. lsr: Recursive version of ls
hdfs dfs -lsr /

REM 3. tail: Displays last kilobyte of the file
REM Syntax: hdfs dfs -tail <inputpath>
hdfs dfs -tail /Data/Sudoku.txt
```

```
                Virtual memory (bytes) snapshot=682012672
                Total committed heap usage (bytes)=402653184
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=237
        File Output Format Counters
                Bytes Written=109

C:\Syncfusion\BigData\2.11.0.92\BigDataSDK\SDK\Hadoop\bin>hadoop fs -cat output/*
hadoop fs -cat output/*
1       'dfs.namenode.checkpoint.txns'
1       'dfs.namenode.checkpoint.period'
1       'dfs.namenode.checkpoint.check.period'

C:\Syncfusion\BigData\2.11.0.92\BigDataSDK\SDK\Hadoop\bin>
```

Connected to Syncfusion Hadoop instance running on localhost

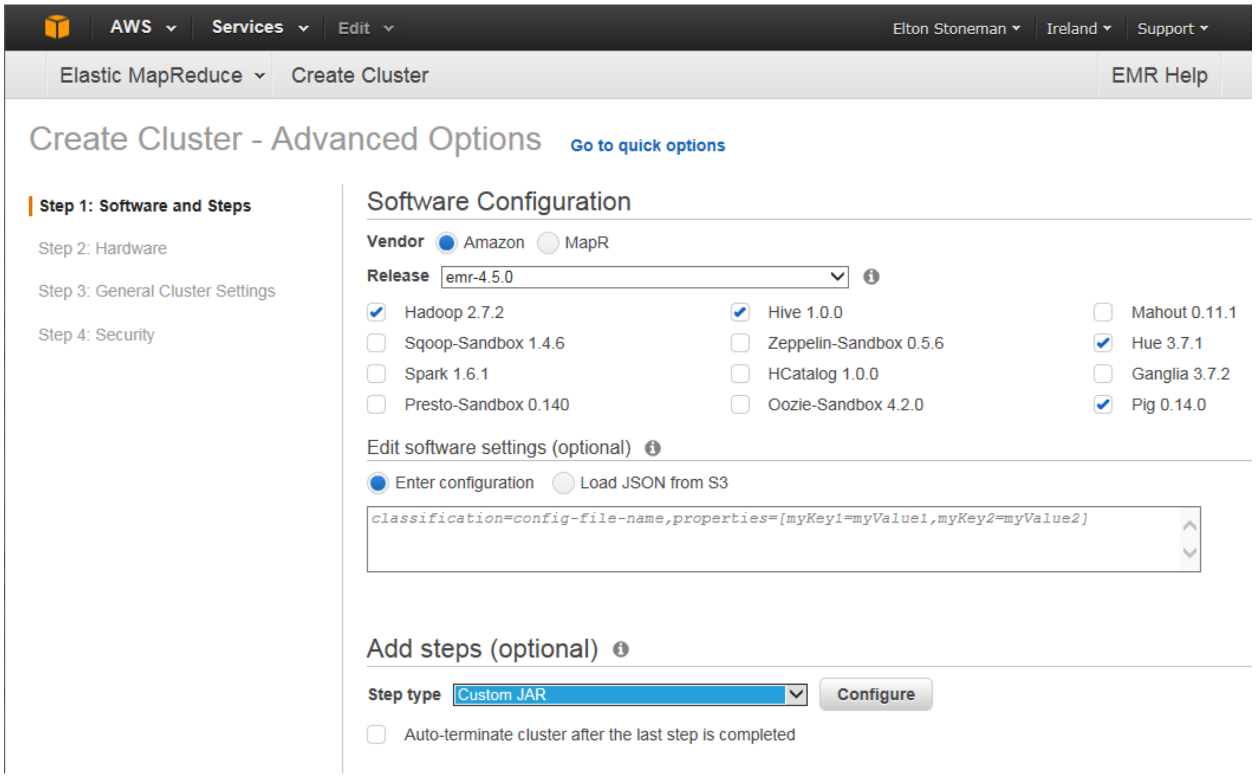
Figure 18: Syncfusion's Big Data Studio

Also worthy of mention are Pivotal, whose HD distribution is expanded with HBD—aiming to make SQL a native part of the Hadoop experience—and IBM, which has deep integration, from their BigInsights Hadoop platform to the rest of their "Big" stack.

Amazon web services

Amazon provides a packaged Hadoop distribution that runs in the Cloud—Amazon Elastic MapReduce (EMR). EMR uses Amazon's S3 for the storage layer (rather than Hadoop) and provisions compute using Elastic Compute Cloud (EC2). This is the same approach that Microsoft uses in the Azure Cloud. The main consideration with cloud-based Hadoop deployments is that you lose the benefit of data locality, although you gain with scale because you are able to easily add or remove compute nodes and have virtually limitless storage capacity.

With EMR, you can provision a cluster from some preconfigured options, including Hive or Spark, and you can specify that custom components be installed as part of the deployment. Amazon also provides a hosted MapR distribution as well as its own Hadoop bundle. Figure 19 shows the creation of a new EMR cluster through the AWS Management Console.



Create Cluster - Advanced Options

Step 1: Software and Steps
Step 2: Hardware
Step 3: General Cluster Settings
Step 4: Security

Software Configuration

Vendor [✓ Amazon] MapR

Release emr-4.5.0

✓ Hadoop 2.7.2	✓ Hive 1.0.0	Mahout 0.11.1
Sqoop-Sandbox 1.4.6	Zeppelin-Sandbox 0.5.6	✓ Hue 3.7.1
Spark 1.6.1	HCatalog 1.0.0	Ganglia 3.7.2
Presto-Sandbox 0.140	Oozie-Sandbox 4.2.0	✓ Pig 0.14.0

Edit software settings (optional)

[✓ Enter configuration] Load JSON from S3

```
classification=config-file-name,properties=[myKey1=myValue1,myKey2=myValue2]
```

Add steps (optional)

Step type Custom JAR

Auto-terminate cluster after the last step is completed

Figure 19: Creating an Elastic MapReduce Cluster

You will be charged an hourly rate to run the VMs that comprise the compute part of the cluster and a separate per-gigabyte charge for all the data you have stored in S3.

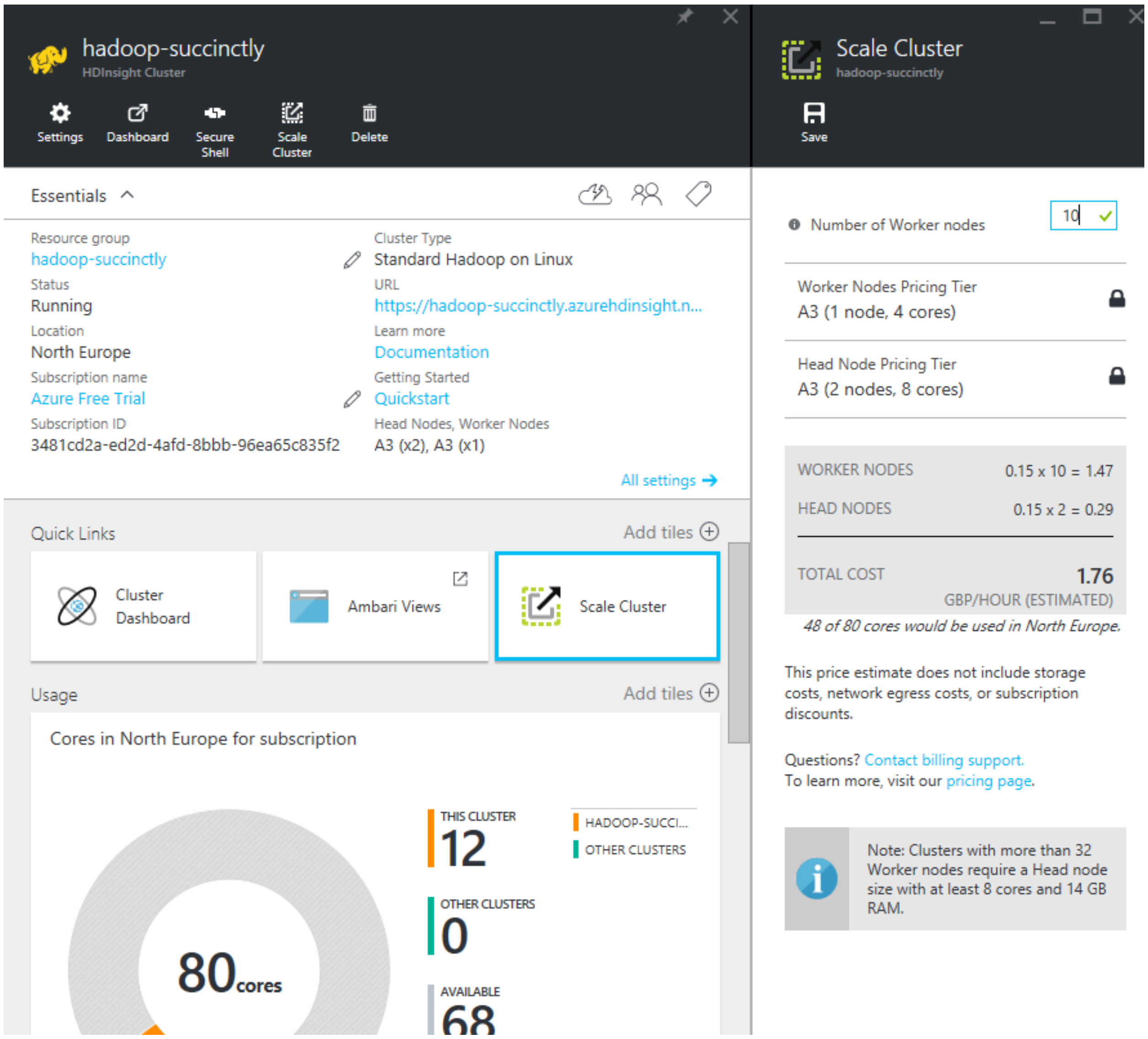
This is one of the major benefits of cloud-based Big Data platforms. If you don't need to run jobs 24/7, you can provision a cluster when you need it, run your batch jobs, then remove the cluster. All of your data remains in cheap object storage, but the expensive compute power is only used when you need it. It's also easy to scale up when you need to—if you need to add more jobs without extending the overall run time, you can simply change your provisioning scripts to create a bigger cluster.

Cloud-based Hadoop distributions also integrate easily with the rest of the provider's stack—another plus point. Both AWS and Azure provide managed cloud-scale queue technologies suitable for receiving huge quantities of data from disparate sources and scalable compute frameworks you can use to pull data from the queues and store in object storage for later querying in Hadoop.

Microsoft Azure

The managed Big Data stack on Microsoft's Azure Cloud, called HDInsight, is actually powered by the Hortonworks Data Platform, and you can spin up clusters based on Windows or Linux VMs. HDInsight clusters use Azure Storage instead of HDFS, and the compute nodes run on Azure Virtual Machines where the cluster is part of a virtual network.

You can manage HDInsight clusters with PowerShell or the Cross-Platform CLI or with the Azure Portal. Figure 20 shows the portal screen for resizing a cluster.



Resource group: hadoop-succinctly
Status: Running
Location: North Europe
Subscription name: Azure Free Trial
Subscription ID: 3481cd2a-ed2d-4afd-8bbb-96ea65c835f2
Cluster Type: Standard Hadoop on Linux
URL: https://hadoop-succinctly.azurehdinsight.n...
Learn more: Documentation
Getting Started: Quickstart
Head Nodes, Worker Nodes: A3 (x2), A3 (x1)

Cores in North Europe for subscription
80 cores
THIS CLUSTER 12
OTHER CLUSTERS 0
AVAILABLE 68
HADOOP-SUCCI...
OTHER CLUSTERS

Number of Worker nodes: 10
Worker Nodes Pricing Tier: A3 (1 node, 4 cores)
Head Node Pricing Tier: A3 (2 nodes, 8 cores)

WORKER NODES	0.15 x 10 = 1.47
HEAD NODES	0.15 x 2 = 0.29
TOTAL COST	1.76 GBP/HOUR (ESTIMATED)

48 of 80 cores would be used in North Europe.

This price estimate does not include storage costs, network egress costs, or subscription discounts.

Questions? Contact billing support.
To learn more, visit our pricing page.

Note: Clusters with more than 32 Worker nodes require a Head node size with at least 8 cores and 14 GB RAM.

Figure 20: Scaling a HDInsight Cluster

HDInsight takes a similar approach to EMR, but instead of configuring the components to add to the cluster, it only allows you to select from one of the preconfigured cluster setups. In the current platform, you can deploy a plain Hadoop cluster from a cluster running Storm (for stream processing), from HBase (for real-time data access), or from Spark (for fast in-memory analysis). Because the storage and compute layers are disconnected, you can run multiple clusters that all have access to the same shared storage.

Although HDInsight is based on the open source HDP platform, Microsoft provides customizations at the client level. The Azure SDK integrates with Visual Studio to give you monitoring and deployment from within the IDE, and there are .NET and PowerShell packages to support development.

As with AWS, you can leverage other parts of Azure to feed data into object storage and query it with a cluster. Also, Azure has the Data Factory offering that lets you orchestrate entire runbooks, including spinning up a Hadoop cluster, submitting jobs, waiting for them to complete, and removing the cluster. With any cloud service, though, you need to be aware that the PaaS components are usually proprietary, so if you take a dependency on them, you will find it difficult to migrate to a different platform.

Choosing between distributions

Adoption of Hadoop can be business-driven, from stakeholders wanting to extract more value from their data, and it can be technology-driven, from engineers wanting to make use of the data they know is there. Typically, there's a prototype phase in which you'll work with Hadoop in a local installation, and it's a good idea to run that phase on pure Hadoop, so that you start to understand the capabilities and shortcomings of running your own cluster. The output of that phase should help the business users decide if the potential returns justify the outlay.

Next, you'll need to choose between an on-premise installation or running in the Cloud. The Cloud offers a great way to start cheaply. You won't incur the upfront costs and build time for provisioning your own cluster—you can spin up a cloud cluster within 30 minutes. If your data is cloud-sourced anyway (from global clients outside your own network), it makes good sense to capture and store that data in the Cloud. Initially, you can run your cluster on a part-time basis, when you need to run batch jobs, and your operating costs will be minimal.

However, there is a crunch point in cloud computing, when the economies flip and it becomes cheaper to run your cluster on-premise. If you run a relatively large Hadoop cluster in the Cloud 24x7 and store a large quantity of data in object storage, your Cloud costs could easily be $5K per month. While $5K buys you a pretty well-specified compute node, over a year you might find that your Cloud costs are the equivalent of owning your own small cluster—although keep in mind that your on-premise cluster will have its own running and management costs.

If you're planning to run on-premise and looking at anything above the minimal Hadoop cluster, a commercial distribution is worth investing in. All the platforms have strong offerings and will likely provide you with a more reliable, more efficient, and more easily managed cluster than you can achieve by spending the same amount on setting up your own cluster from the pure Apache platform.

Ultimately, the platform choice is usually driven by cost, but you should drill down into all the potential costs when you make your decision. Buying your own kit may seem cheaper than paying the ongoing price for a cloud platform, but if you need to upgrade your cluster every six months with an additional 100 TB of storage and another five compute nodes, the cost savings may be illusory.

Summary

In this chapter, we looked at the commercial options for running Hadoop. Although Hadoop is a fully open source platform, it is a complex product with multiple moving parts, and many enterprises don't like to take a dependency on it without having commercial support available. That commercial opportunity is taken up by packaged Hadoop distributions and hosted Hadoop platforms.

In the packaged space, the major players are Cloudera, MapR, and Hortonworks. Each takes a different approach, but there's a lot of commonality in their offerings, so it's worthwhile getting to know the platforms well if you intend to run on-premise. The commercial market is also expanding, with newer products from entrants like Pivotal, Syncfusion, and IBM offering their own strengths.

If you don't have a definite longterm data roadmap based on Hadoop, a Cloud-based platform may be a better option, certainly when you're starting out. Both Amazon and Microsoft provide feature-rich Big Data platforms that have efficient pricing. Data and compute are billed separately, so if you can group all your batch jobs into one run, you can power up a cluster to do all the jobs, then power it down, which means you pay only expensive compute costs while the cluster is actually running.

Whichever option you choose, be mindful of the points at which the product departs from the standard Hadoop stack. If you focus your own investment on core Hadoop functionality (or functionality that is 100% compatible with Hadoop) and don't take a hard dependency on any proprietary technology, you'll be well placed to move to a different distribution if you later choose to do so.

Chapter 8 The Hadoop Ecosystem

Introducing the ecosystem

We can't fully discuss Hadoop without mentioning the ecosystem. Because Hadoop is an open source platform with lots of potential and relatively few drawbacks, a huge community has developed around it, extending the reach and the capabilities of Hadoop. The ecosystem encompasses a wide range of technologies in different spheres, and in the final chapter of this e-book we'll look at a few of the major ones.

Most of the technologies in the ecosystem embrace the core Hadoop platform, plugging holes or providing alternative ways to work, but a few of them use the core features of Hadoop to support a completely different approach. All the tools we'll look at are open source, often with a large community following coupled with substantial investment from businesses.

Although Hadoop is a powerful and flexible platform, it is seldom used on its own. Augmenting Hadoop with tools from the ecosystem adds value to the basic proposition, making your cluster and its workload easier to manage or giving you faster and easier access to your data.

Managing Hadoop with Ambari

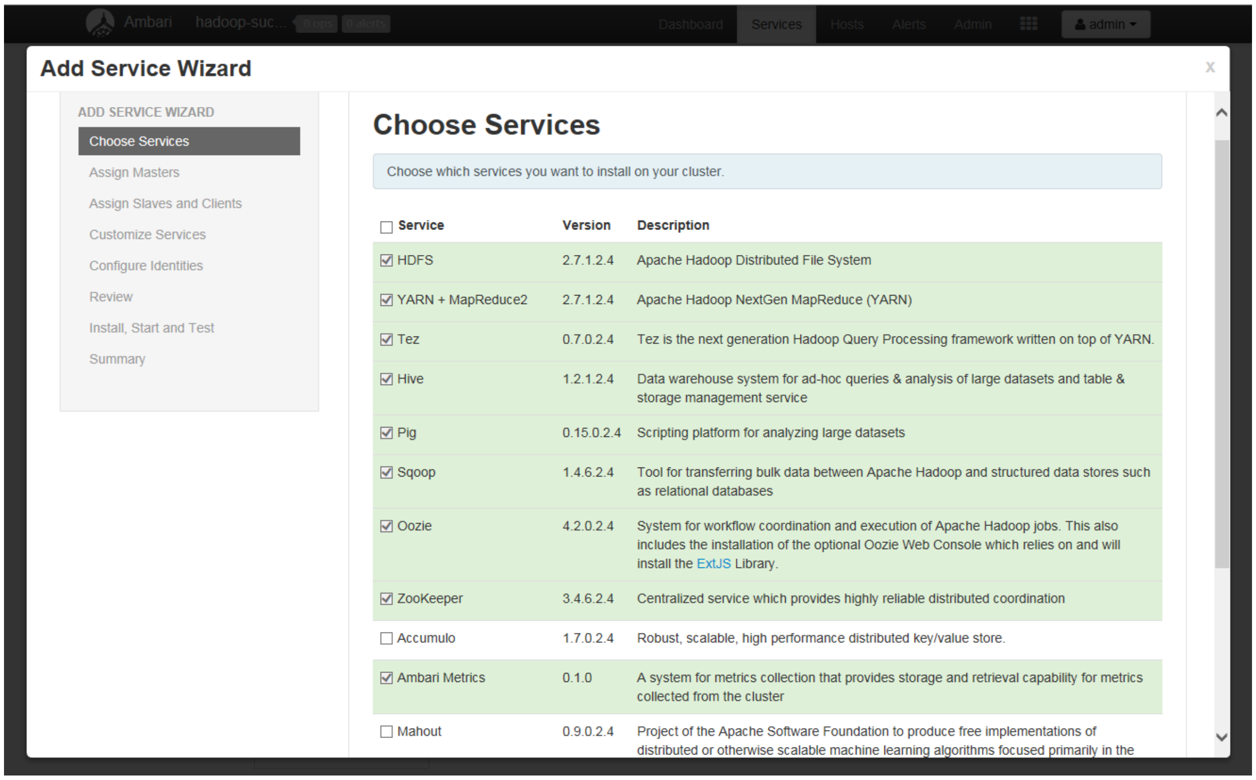
While installing Hadoop from scratch is not a daunting task, setting up a multinode cluster is far more complex, and it presents a much greater surface area for extension. If you decide to add Hive or Spark to your cluster, it's not trivial when you have 10 data nodes to deploy to and keep in sync. Ambari is the management console that addresses that problem and many more.

Ambari is a web application that aims to provide a single point of access for your Hadoop cluster. It has an extensible architecture with a growing library of components that provide functionality for administrating, monitoring, and using your cluster. Along with providing web views, it also exposes a REST API for automating access to your cluster.

In the administration space, Ambari has all the major tasks covered:

- Hadoop configuration: You can view and edit configuration settings without touching the XML files on the server.
- Cluster setup: You can add and remove nodes to the cluster, stop and start services, and enter maintenance mode.
- Component deployment: You can add and remove packaged components like Hive.
- Component configuration: You can view and edit settings for installed components in the same way as the Hadoop configuration.

Figure 21 shows the Add Service Wizard in Ambari, where you can add other components from the ecosystem to your cluster with ease.



Choose Services

Choose which services you want to install on your cluster.

☐ Service	Version	Description
☑ HDFS	2.7.1.2.4	Apache Hadoop Distributed File System
☑ YARN + MapReduce2	2.7.1.2.4	Apache Hadoop NextGen MapReduce (YARN)
☑ Tez	0.7.0.2.4	Tez is the next generation Hadoop Query Processing framework written on top of YARN.
☑ Hive	1.2.1.2.4	Data warehouse system for ad-hoc queries & analysis of large datasets and table & storage management service
☑ Pig	0.15.0.2.4	Scripting platform for analyzing large datasets
☑ Sqoop	1.4.6.2.4	Tool for transferring bulk data between Apache Hadoop and structured data stores such as relational databases
☑ Oozie	4.2.0.2.4	System for workflow coordination and execution of Apache Hadoop jobs. This also includes the installation of the optional Oozie Web Console which relies on and will install the ExtJS Library.
☑ ZooKeeper	3.4.6.2.4	Centralized service which provides highly reliable distributed coordination
☐ Accumulo	1.7.0.2.4	Robust, scalable, high performance distributed key/value store.
☑ Ambari Metrics	0.1.0	A system for metrics collection that provides storage and retrieval capability for metrics collected from the cluster
☐ Mahout	0.9.0.2.4	Project of the Apache Software Foundation to produce free implementations of distributed or otherwise scalable machine learning algorithms focused primarily in the

Figure 21: Adding Services with Ambari

For monitoring, Ambari collects stats from all the nodes in the cluster and all the jobs that have been run. You can view and drill down on graphs that show CPU, memory, disk, and network usage along with counts of all the known HDFS and YARN nodes and their status.

Figure 22 shows the Ambari home screen, which summarizes the health and status of the cluster.

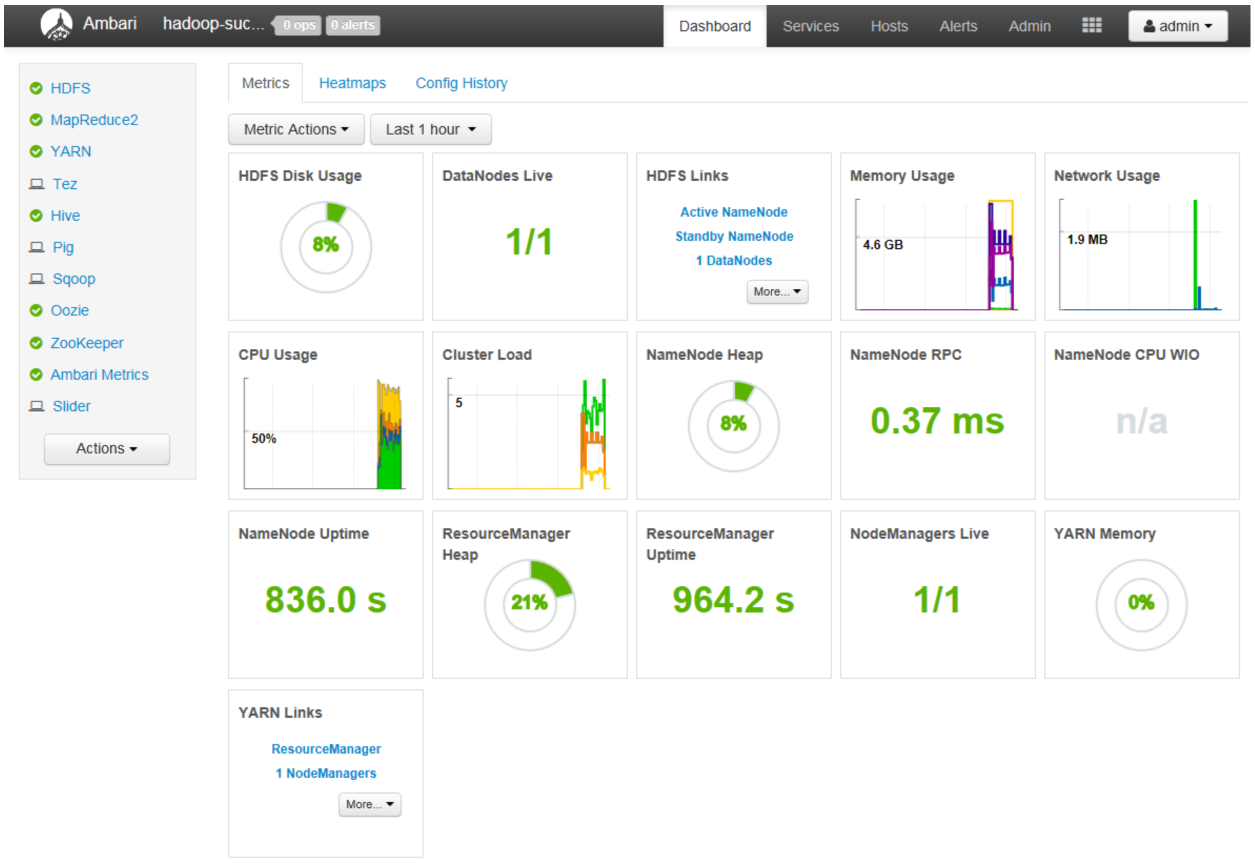
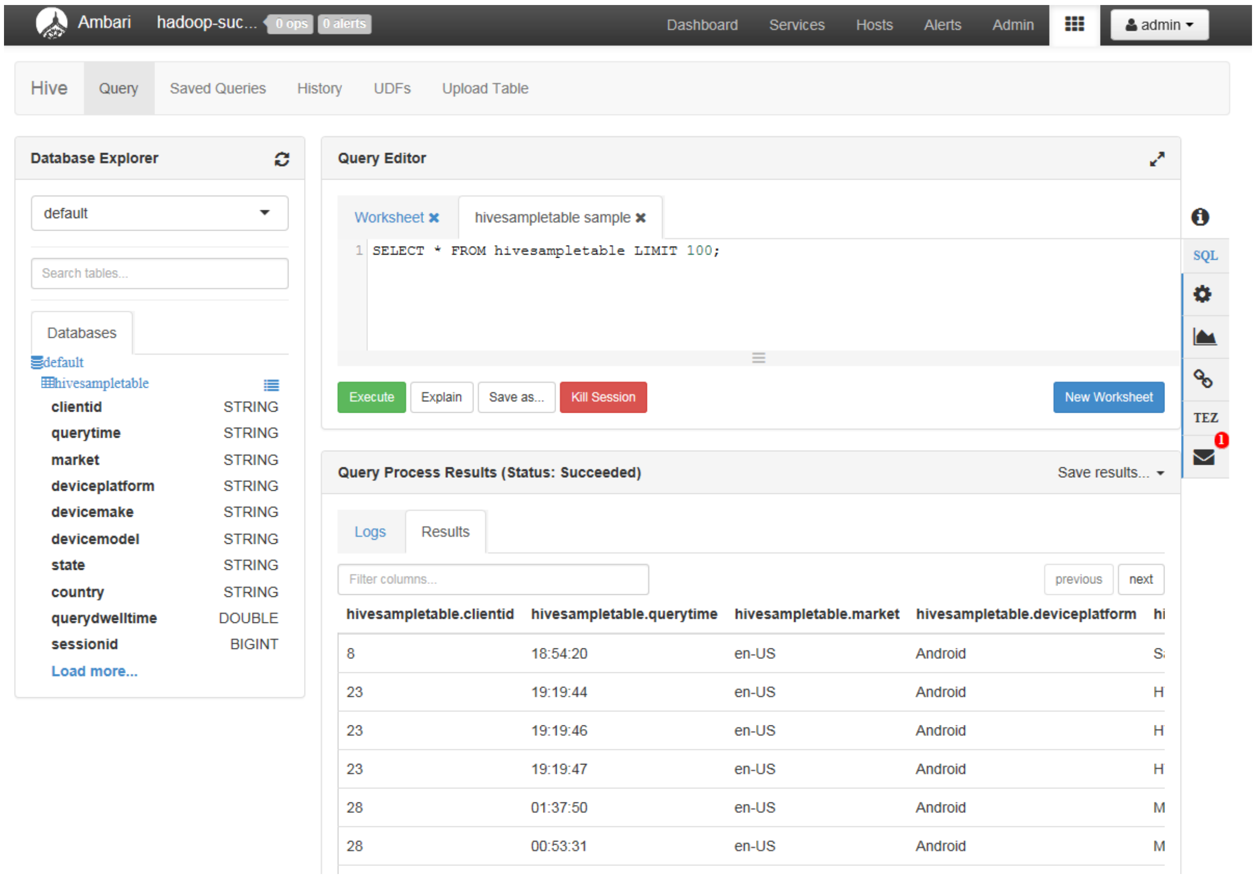


Figure 22: The Ambari Home Screen

On the usage front, Ambari has views that integrate with other parts of the ecosystem. Pig and Hive are alternative ways of querying data in Hadoop, and Ambari has views for each that let you submit queries and view the results, and views that save the queries you've run, so that you can load them again later.

Figure 23 shows the Hive view, in which you have a schema navigator, an input pane to write your query, and an output pane showing query results and logs.



```
SELECT * FROM hivesampletable LIMIT 100;
```

clientid	STRING
querytime	STRING
market	STRING
deviceplatform	STRING
devicemake	STRING
devicemodel	STRING
state	STRING
country	STRING
querydwelltime	DOUBLE
sessionid	BIGINT

hivesampletable.clientid	hivesampletable.querytime	hivesampletable.market	hivesampletable.deviceplatform	hi
8	18:54:20	en-US	Android	S
23	19:19:44	en-US	Android	H
23	19:19:46	en-US	Android	H
23	19:19:47	en-US	Android	H
28	01:37:50	en-US	Android	M
28	00:53:31	en-US	Android	M

Figure 23: The Hive View in Ambari

MapReduce alternatives: Pig

As Hadoop has gained popularity, larger numbers of users are being exposed to MapReduce and have started to feel the pain of writing multiple Java classes to access their data, even for simple-use cases.

Although MapReduce is hugely powerful, it's a complex technology that can require cooperation from multiple disciplines—analysts to define the desired results, engineers to build the programs, testers to verify the code, and so on.

The earliest alternative to the Java MapReduce API was Pig, which is a simplified Domain Specific Language (DSL) for building MapReduce queries. Although the Pig DSL (called Pig Latin) is a new language, it's a minimal one, and new users from different disciplines can be productive with it in a few hours. A sample Pig query is shown in Code Listing 36.

Code Listing 36: A Matching Line Count in Pig Latin

```
lines = LOAD 'input ' USING PigStorage() AS (line);

matching = FILTER lines BY line MATCHES '.*dfs.*';

match_groups = GROUP matching BY line;

word_count = FOREACH match_groups GENERATE matching as line,
COUNT(matching);

STORE word_count INTO 'output5';
```

This query is very similar to our custom MapReduce job from Chapter 2—reading a set of input files with the **`LOAD`** statement, using **`FILTER`** to find lines that contain the string 'dfs,' then counting the occurrences with the **`GROUP`** and **`GENERATE COUNT`** statements. The **`STORE`** statement specifies where the result should be written.

The Pig script contains much less code, is easier to read, and doesn't require a Java IDE to work with, build, or package the query. You can run Pig interactively, building up a query line by line, or you can submit entire scripts for batch execution.

You can execute Pig queries with the Pig command line (or with the Ambari Pig View), and the Pig engine will build a Java MapReduce program from your query and submit it to the cluster. As far as YARN and HDFS are concerned, this is a standard MapReduce job and it runs in the same way as a normal Java job.

Pig is usually the first tool that new Hadoop users pick up—it gives a good understanding of the structure of MapReduce without the complexity of the Java API. The scope of Pig is limited, which means you'll likely need to build custom MapReduce programs for complex jobs. But for simple jobs, Pig significantly lowers the entry barrier.

SQL on Hadoop: Hive

The next step up from Pig is an SQL-on-Hadoop framework that lets you query data in Hadoop using an SQL-like language. The open nature of Hadoop does lead to one downside—competing technologies appear in the same space, and it can be difficult to choose among them. The SQL-on-Hadoop space is a prime example, as there are several mature and well-featured alternatives, including Impala (from Cloudera), HAWQ (from Pivotal), and Apache Hive.

I'll focus on Hive here because it has a large user base, is under continuing development, and is the supported SQL-like language for Spark.

Data in Hadoop is widely variable, but groups of data often share similar traits. A folder containing web server logs might have files with different contents, but they might all be tab-separated text files with a number of key fields present in every file. With Hive, you impose a structure on your Hadoop data by defining tables in a similar way to a relational database.

Hive table definitions specify the source of the data in Hadoop, the column names and data types the files contain, and the mapping configuration, so that Hive knows how to convert the raw data into rows. In order to run our MapReduce query in Hive, we'd first need to create a table that represents the files in the input folder. Code Listing 37 shows an example.

Code Listing 37: HiveQL to Create a Table

```
CREATE EXTERNAL TABLE input(line STRING)

STORED AS TEXTFILE

LOCATION '/user/root/input'
```

Next, we can query that table with HiveQL, an SQL-like language that is very close to SQL-92 compliance. In order to count rows containing a string match, we can use a query like Code Listing 38.

Code Listing 38: HiveQL to Query a Table

```
SELECT COUNT(*) FROM input WHERE line LIKE '%dfs%'
```

Similar to Pig, Hive contains a compiler that builds a MapReduce program from the HiveQL, so that the cluster actually executes a normal MapReduce job. Hive is great for establishing a data warehouse over the main areas of your data landscape—there's a lot more to it and it's worth looking into.

Syncfusion has published *Hive Succinctly* (also by me), which covers the technology in more detail.

Real-Time Big Data: HBase

HBase is a departure from the other tools we've covered because it doesn't extend Hadoop—it's an alternative Big Data technology that uses Hadoop for its foundation. HBase is a real-time Big Data store that uses HDFS to store data so that it gains all the scalability and reliability at the storage layer without building that from scratch.

We use HBase like a NoSQL database. There are tables and rows, but the cells within rows don't belong to fixed columns. Instead, each table has one or more column families, and the column family is like a hash table—one row may have a column family with 1,000 columns while another row in the same table has only one column.

HBase is hugely scalable. It has the capability to store tables with billions of rows and millions of columns. It can provide real-time access because every row has a unique key, and HBase uses an efficient, indexed storage format to quickly find rows by key. HBase also employs a master/slave architecture, but YARN isn't used.

HDFS runs in the normal way with name and data nodes, but HBase has its own service processes. Because the requirements and access patterns are different with HBase, it typically has a dedicated cluster and the nodes aren't used for batch processing.

Other parts of the ecosystem integrate with HBase, so that we can map HBase tables in Hive and query them using HiveQL, which means we can use the Ambari Hive view to query HBase tables. HBase also uses another core Apache technology—Zookeeper—as a centralized configuration and alerting system between the nodes.

Syncfusion also has a dedicated title on HBase—again by me—*HBase Succinctly*.

The new Big Data toolkit: Spark

Spark is the new Big Thing in Big Data. It is a collection of technologies that lets you work with Big Data in new ways—there's an SQL engine, a machine-learning component, a streaming interface, and a dedicated-graph component. Spark works with explicit units of data called Resilient Distributed Datasets (RDDs). Typically, a chain of work is built up on an RDD, one that might include transforms or classic MapReduce patterns, then the RDD is executed when results are needed.

As opposed to the more complex job-splitting approach of MapReduce, in which the work is encapsulated as a Directed Acyclic Graph (DAG), RDD execution uses multiple parts that can execute concurrently. Spark moves data into memory when executing tasks, and commonly used data can be explicitly cached in memory—which means Spark is fast.

Speed of execution and breadth of functionality are driving Spark as the go-to toolkit for Big Data, but Spark is still rooted in Hadoop. Spark can be deployed on a shared Hadoop cluster, Spark queries the data in HDFS, and Spark jobs can execute on YARN, which allows us to run MapReduce and Spark jobs in the same cluster and prioritize work with the scheduler.

A key factor in Spark's adoption is that it supports interactive querying through notebooks such as Jupyter. That allows for fast, exploratory analysis of your dataset, so that you can quickly iterate queries over a small, cached dataset until you have the results you want, then you can submit the job to YARN for the entire dataset.

Spark is also covered in Syncfusion's library with *Spark Succinctly*, written by Marko Švaljek.

Summary

That quick tour of the Hadoop ecosystem ends this e-book. We looked at some key technologies that extend Hadoop or make use of it as a core platform, and most Big Data journeys will incorporate one or more of these tools.

If you're about to start your journey and you're wondering which tools you should look at, here's a suggested roadmap. Start with the **`hadoop-succinctly`** Docker container and some simple MapReduce programs in order to get a good feel for how the cluster operates in practice and to understand how to monitor what the cluster is doing.

Next, move on to a new installation with Ambari—which can be a single node—so that you can easily add components and see how the cluster services are surfaced in Ambari. Pig is a useful alternative to MapReduce if you have widely varying data, and Hive is a good technology to know if you want to provide a data warehouse that analysts can use (mastering HiveQL is much easier for nonengineers than is learning Java).

A good time to look at Spark will be when you're comfortable with running queries from different engines, monitoring job progress and the health of the cluster, and possibly extending your cluster to add new nodes. Although you might find yourself using Spark exclusively in the longterm, starting with a solid understanding of Hadoop will definitely prove helpful.