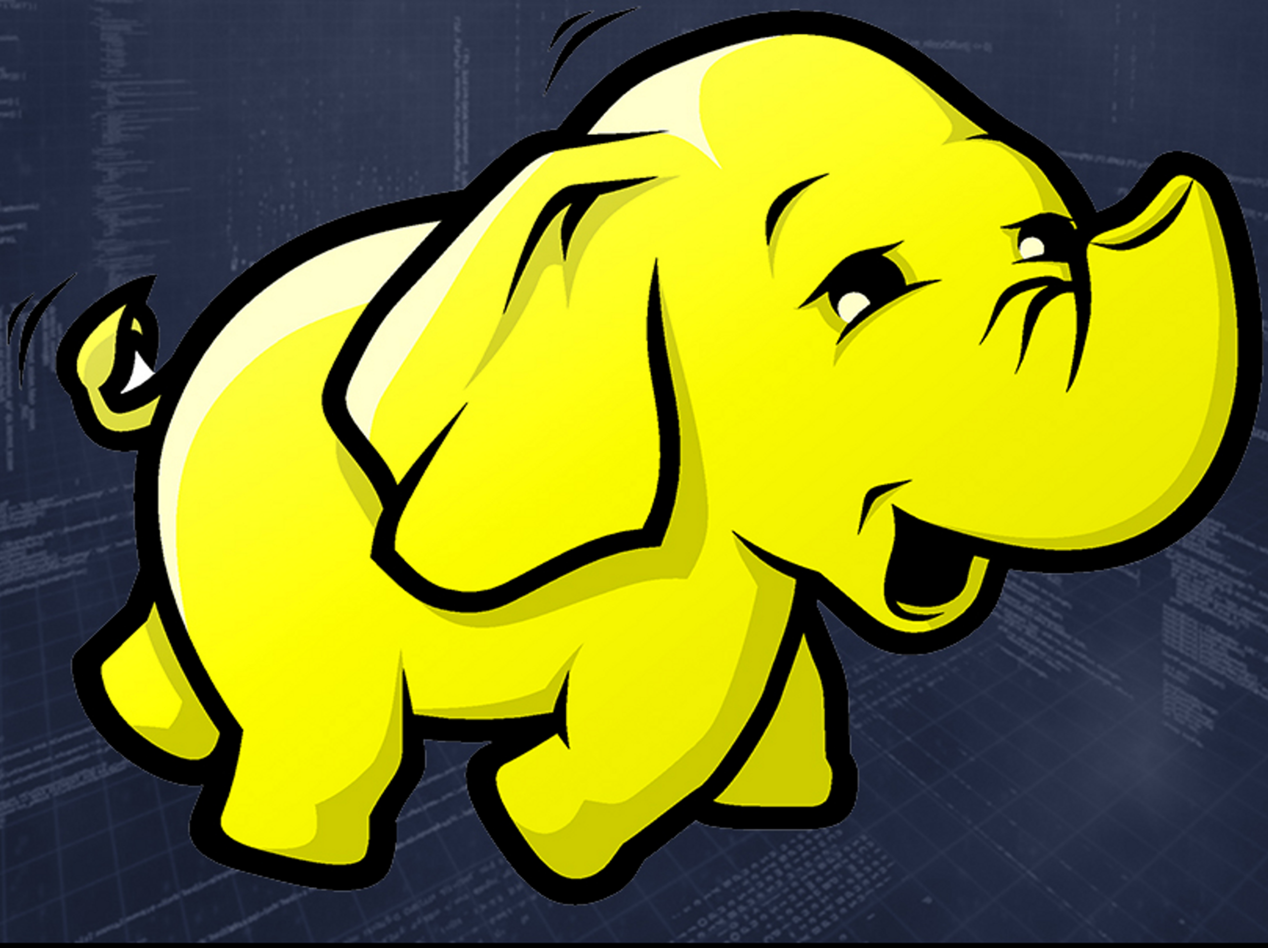
APACHE hadoop TUTORIAL

The Ultimate Guide



MARTIN MOIS

WEB CODE GEEKS
WEB DEVELOPERS RESOURCE CENTER

Apache Hadoop Tutorial

Contents

Preface

Apache Hadoop is an open-source software framework written in Java for distributed storage and distributed processing of very large data sets on computer clusters built from commodity hardware. All the modules in Hadoop are designed with a fundamental assumption that hardware failures are common and should be automatically handled by the framework.

Hadoop has become the de-facto tool used for Distributed computing. For this reason we have provided an abundance of tutorials here at Java Code Geeks, most of which can be found here: http://examples.javacodegeeks.com/category/enterprise-java/apache-hadoop/

Now, we wanted to create a standalone, reference post to provide a framework on how to work with Hadoop and help you quickly kick-start your own applications. Enjoy!

About the Author

Martin is a software engineer with more than 10 years of experience in software development. He has been involved in different positions in application development in a variety of software projects ranging from reusable software components, mobile applications over fat-client GUI projects up to larg-scale, clustered enterprise applications with real-time requirements.

After finishing his studies of computer science with a diploma, Martin worked as a Java developer and consultant for international operating insurance companies. Later on he designed and implemented web applications and fat-client applications for companies on the energy market. Currently Martin works for an international operating company in the Java EE domain and is concerned in his day-to-day work with larg-scale big data systems.

His current interests include Java EE, web applications with focus on HTML5 and performance optimizations. When time permits, he works on open source projects, some of them can be found at this github account. Martin is blogging at Martin's Developer World.

Chapter 1

Introduction

Apache Hadoop is a framework designed for the processing of big data sets distributed over large sets of machines with commodity hardware. The basic ideas have been taken from the Google File System (GFS or GoogleFS) as presented in this paper and the MapReduce paper.

A key advantage of Apache Hadoop is its design for scalability, i.e. it is easy to add new hardware to extend an existing cluster in means of storage and computation power. In contrast to other solutions the used principles do not rely on the hardware and assume it is highly available, but rather accept the fact that single machines can fail and that in such case their job has to be done by other machines in the same cluster without any interaction by the user. This way huge and reliable clusters can be build without investing in expensive hardware.

The Apache Hadoop project encompasses the following modules:

- Hadoop Common: Utilities that are used by the other modules.
- Hadoop Distributed File System (HDFS): A distributed file system similar to the one developed by Google under the name GFS.
- Hadoop YARN: This module provides the job scheduling resources used by the MapReduce framework.
- Hadoop MapReduce: A framework designed to process huge amount of data

The modules listed above form somehow the core of Apache Hadoop, while the ecosystem contains a lot of Hadoop-related projects like Avro, HBase, Hive or Spark.

Chapter 2

Setup

2.1 Setup "Single Node"

In order to get started, we are going to install Apache Hadoop on a single cluster node. This type of installation only serves the purpose to have a running Hadoop installation in order to get your hands dirty. Of course you don't have the benefits of a real cluster, but this installation is sufficient to work through the rest of the tutorial.

While it is possible to install Apache Hadoop on a Windows operating system, GNU/Linux is the basic development and production platform. In order to install Apache Hadoop, the following two requirements have to be fulfilled:

- Java >= 1.7 must be installed.
- ssh must be installed and sshd must be running.

If ssh and sshd are not installed, this can be done using the following commands under Ubuntu:

```
$ sudo apt-get install ssh
$ sudo apt-get install rsync
```

Now that ssh is installed, we create a user named `hadoop` that will later install and run the HDFS cluster and the MapReduce jobs:

```
$ sudo useradd -s /bin/bash -m -p hadoop hadoop
```

Once the user is created, we open a shell for it, create a SSH keypair for it, copy the content of the public key to the file `authorized_keys` and check that we can login to localhost using ssh without password:

```
$ su - hadoop
$ ssh-keygen -t rsa -P ""
$ cat $HOME/.ssh/id-rsa.pub >> $HOME/.ssh/authorized_keys
$ ssh localhost
```

Having setup the basic environment, we can now download the Hadoop distribution and unpack it under `/opt/hadoop`. Starting HDFS commands just from the command line requires that the environment variables `JAVA_HOME` and `HADOOP_HOME` are set and the HDFS binaries are added to the path (please adjust the paths to your environment):

```
$ export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-amd64
$ export HADOOP_HOME=/opt/hadoop/hadoop-2.7.1
$ export PATH=$PATH:$HADOOP_HOME/bin
```

These lines can also be added to the file `.bash_profile` to not type them each time again.

In order to run the so called "pseudo-distributed" mode, we add the following lines to the file `$HADOOP_HOME/etc/hadoop/core-site.xml`:

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:9000</value>
    </property>
</configuration>
```

The following lines are added to the file `$HADOOP_HOME/etc/hadoop/hdfs-site.xml` (please adjust the paths to your needs):

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
                <property>
      <name>dfs.namenode.name.dir</name>
      <value>/opt/hadoop/hdfs/namenode</value>
   </property>
   <property>
      <name>dfs.datanode.data.dir</name>
      <value>/opt/hadoop/hdfs/datanode</value>
   </property>
</configuration>
```

As user hadoop we create the paths we have configured above as storage:

```
mkdir -p /opt/hadoop/hdfs/namenode
mkdir -p /opt/hadoop/hdfs/datanode
```

Before we start the cluster, we have to format the file system:

```
$ $HADOOP_HOME/bin/hdfs namenode -format
```

Now its time to start the HDFS cluster:

```
$ $HADOOP_HOME/sbin/start-dfs.sh
Starting namenodes on [localhost]
localhost: starting namenode, logging to /opt/hadoop/hadoop-2.7.1/logs/hadoop-hadoop- ←↩
    namenode-m1.out
localhost: starting datanode, logging to /opt/hadoop/hadoop-2.7.1/logs/hadoop-hadoop- ←↩
    datanode-m1.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /opt/hadoop/hadoop-2.7.1/logs/hadoop-hadoop ←↩
    -secondarynamenode-m1.out
```

If the start of the cluster was successful, we can point our browser to the following URL: http://localhost:50070/. This page can be used to monitor the status of the cluster and to view the content of the file system using the menu item `Utilities > Browse the file system`.

2.2 Setup "Cluster"

The setup of Hadoop as a cluster is very similar to the "single node" setup. Basically the same steps as above have to be performed. The only difference is that a cluster needs only one NameNode, i.e. we have to create and configure the directory for the NameNode only on the node that is supposed to run the NameNode instance (master node).

The file `$HADOOP_HOME/etc/hadoop/slaves` can be used to tell Hadoop about all machines in the cluster. Just enter the name of each machine as a separate line in this file where the first line denotes the node that is supposed to be the master (i.e. runs the NameNode):

```
master-node.mydomain.com
slave-node.mydomain.com
another-slave-node.mydomain.com
...
```

Before you continue, please make sure that you can ssh to all machines in the slave file using their DNS name (as stored in the `slaves` file) without providing a password. As a prerequisite this means that you have created a hadoop user on all machines and copied the hadoop user's public key file from the master node to the `authorized_keys` file to all other machines in the cluster using for example the following command:

```
$ ssh-copy-id -i $HOME/.ssh/id_rsa.pub hadoop@slave-node.mydomain.com
```

We change the following lines in the file `$HADOOP_HOME/etc/hadoop/core-site.xml` such that it contains the DNS name of the master node on all machines:

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://master-node.mydomain.com:9000</value>
    </property>
</configuration>
```

Additionally we can now also increase the replication factor appropriately to our cluster setup. Here we use the value 3 assuming that we have at least three different DataNodes:

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>3</value>
    </property>
```

If the `slaves` files is configured properly and trusted access (i.e. access without passwords for the hadoop user) is configured for all machines listed in the `slaves` file, the cluster can be started with the following command line on that master node:

```
$ $HADOOP_HOME/sbin/start-dfs.sh
```

Optionally, instead of starting the cluster automatically from the master node, one can also use the `hadoop-daemon.sh` script to start either a NameNode or a DataNode on each of the machines:

```
$ $HADOOP_HOME/sbin/hadoop-daemon.sh --config $HADOOP_CONF_DIR --script hdfs start namenode
$ $HADOOP_HOME/sbin/hadoop-daemons.sh --config $HADOOP_CONF_DIR --script hdfs start  ↩
   datanode
```

By replacing the command `start` with `stop`, one can later on stop the dedicates servers:

```
$ $HADOOP_HOME/sbin/hadoop-daemons.sh --config $HADOOP_CONF_DIR --script hdfs stop datanode
$ $HADOOP_HOME/sbin/hadoop-daemon.sh --config $HADOOP_CONF_DIR --script hdfs stop namenode
```

Please note that first all DataNodes should be stopped. The NameNode is stopped after all DataNodes have been shutdown.

Chapter 3

HDFS

3.1 HDFS Architecture

HDFS (Hadoop Distributed File System) is, as the name already states, a distributed file system that runs on commodity hardware. Like other distributed file systems it provides access to files and directories that are stored over different machines on the network transparently to the user application. But in contrast to other similar solutions, HDFS makes a few assumptions that are not so common:

- Hardware Failure is seen more as a norm than as an exception. Instead of relying on expensive fault-tolerant hardware systems, commodity hardware is chosen. This not only reduces the investment for the whole cluster setup but also enables the easy replacement of failed hardware.
- As Hadoop is designed for batch processing of large data sets, requirements that stem from the more user-centered POSIX standard are relaxed. Hence low-latency access to random parts of a file is less desirable than streaming files with high throughput.
- The applications using Hadoop process large data sets that reside in large files. Hence Hadoop is tuned to handle big files instead of a lot of small files.
- Most big data applications write the data once and read it often (log files, HTML pages, user-provided images, etc.). Therefore Hadoop makes the assumption that a file is once created and then never updated. This simplifies the coherency model and enables high throughput.

In HDFS there are two different types of servers: NameNodes and DataNodes. While there is only one NameNode, the number of DataNodes is not restricted.

The NameNode serves all metadata operations on the file system like creating, opening, closing or renaming files and directories. Therefore it manages the complete structure of the file system. Internally a file is broken up into one or more data blocks and these data blocks are stored on one or more DataNodes. The knowledge which data blocks form a specific file resides on the NameNode, hence the client receives the list of data blocks from the NameNode and can later on contact the DataNodes directly in order to read or write the data.

The fact that the whole cluster has only one NameNode makes the complete architecture very simple but also introduces a single point of failure (SPOF). Before Hadoop 2.0 it was not possible to run two NameNodes in a failover mode, i.e. when one NameNode was down due to a crash of the machine or due to a maintenance operation, the whole cluster was out of service. Newer versions of Hadoop allow to run two NameNode instances in an active/passive configuration with hot standby. As a requirement for this configuration, both machines running the NameNodes servers are supposed to have the same hardware and a shared storage (that again needs to be high-available).

HDFS uses a traditional hierarchical file system. This means that data resides in files that are grouped into directories. One can create and remove files and directories and move files from one directory to another one. In contrast to other file systems HDFS does currently not support hard or soft links.

Data replication is a key element for fault tolerance in HDFS. The replication factor of a file determines how many copies of this file should be stored within the cluster. How these replicas are distributed over the cluster is determined by the replication policy. The current default policy tries to store one replica of a block on the same local rack as the original one and the second replica on another remote rack.

If there should be another replica this one gets stored on the same remote rack as the second replica. As network bandwidth between nodes that run in the same rack is typically greater than between different racks (which has to go through switches), this policy allows applications to read all replicas from the same rack and therewith to not utilize the network switches that connect the racks. The underlying assumption here is that a rack failure is much less probable than a node failure.

All information about the HDFS namespace is stored on the NameNode and kept in memory. Changes to this data structure are written as entries to a transaction log called EditLog. This EditLog is kept as a normal file in the file system of the operating system the NameNode is running on. Using a transaction log to record all changes makes it possible to restore these changes if the NameNode crashes. Hence, when the NameNode starts, it reads the transaction log from the local disc and applies all changes stored in it to the last version of the namespace.

Once all changes have been applied it stores the updated data structure on the local file system in a file called FsImage. The transaction log can then be deleted as it has been applied and its information has been stored in the latest FsImage file. From then on all futher changes are again stored within a new transaction log. The process of applying the transaction log to an older version of FsImage and then replacing its latest version is called "checkpoint". Currently such checkpoints are only executed when the NameNode starts.

The DataNodes store the blocks within their local file system and distribute the files over directories such that the local file system can deal efficiently with them. At startup the DataNode scans the local file system structure and afterwards sends a list of all blocks it stores to the NameNode (the BlockReport).

HDFS is robust against a number of failure types:

- DataNode failure: Each DataNode sends from time to time a heartbeat message to the NameNode. If the NameNode does not receive any heartbeat for a specific amount of time from a DataNode, the node is seen to be dead and no further operations are scheduled for it. A dead DataNode decreases the replication factor of the data blocks it stored. To prevent data loss, the NameNode can start new replication processes to increase the replication factor for these blocks.
- Network partitions: If the cluster breaks up into two or more partitions, the NameNode loses the connection to a set of DataNodes. These DataNodes are seen as dead and no further I/O operations are scheduled for them.
- Data integrity: When a client uploads data into the file system, it computes a checksum for each data block. This checksum gets stored within a hidden file in the same namespace. If the same file is read later on, the client reading the data blocks also retrieves the hidden file and compares the checksums with the ones it computes for the received blocks.
- NameNode failure: As the NameNode is a single point of failure, it is critical that its data (EditLog and FsImage) can be restored. Therefore one can configure to store more than one copy of the EditLog and FsImage file. Although this decreases the speed with which the NameNode can process operations, it ensures at the same time that multiple copies of the critical files exist.

3.2 HDFS User Guide

HDFS can be accessed in different ways. Next to a Java API and a C wrapper around this API, the distribution also ships with a shell that has commands that are similar to other known shells like the bash or csh.

The following command shows how to list all files in the root directory:

```
[hdfs@m1 ~]$ hdfs dfs -ls /
Found 2 items
drwxr-xr-x   - hdfs hdfs          0 2016-01-01 09:13 /apps
drwx------   - hdfs hdfs          0 2016-01-01 08:22 /user
```

A new directory can be created by specifying `-mkdir` as third parameter and the directory name as fourth parameter:

```
[hdfs@m1 ~]$ hdfs dfs -mkdir /foo
[hdfs@m1 ~]$ hdfs dfs -ls /
Found 3 items
drwxr-xr-x   - hdfs hdfs          0 2015-08-06 09:13 /apps
drwxr-xr-x   - hdfs hdfs          0 2016-01-18 07:02 /foo
drwx------   - hdfs hdfs          0 2015-07-21 08:22 /user
```

The commands `-put` and `-get` can be used to upload or download a file from the file system:

```
[hdfs@m1 ~]$ echo "Hello World" > /tmp/helloWorld.txt
[hdfs@m1 ~]$ hdfs dfs -put /tmp/helloWorld.txt /foo/helloWorld.txt
[hdfs@m1 ~]$ hdfs dfs -get /foo/helloWorld.txt /tmp/helloWorldDownload.txt
[hdfs@m1 ~]$ cat /tmp/helloWorldDownload.txt
Hello World
[hdfs@m1 ~]$ hdfs dfs -ls /foo
Found 1 items
-rw-r--r--   3 hdfs hdfs         12 2016-01-01 18:04 /foo/helloWorld.txt
```

Now that we have seen how to list files using the HDFS shell, it is time to implement the simple `-ls` command in Java. Therefore we create a simple maven project and add the following line to the `pom.xml`:

```
<properties>
        <hadoop.version>2.7.1</hadoop.version>
</properties>

<dependencies>
        <dependency>
                <groupId>org.apache.hadoop</groupId>
                <artifactId>hadoop-client</artifactId>
                <version>${hadoop.version}</version>
        </dependency>
</dependencies>

<build>
        <plugins>
                <plugin>
                        <artifactId>maven-assembly-plugin</artifactId>
                        <configuration>
                                <archive>
                                        <manifest>
                                                <mainClass>ultimate.hdfs.HdfsClient</ ↩
                                                    mainClass>
                                        </manifest>
                                </archive>
                                <descriptorRefs>
                                        <descriptorRef>jar-with-dependencies</descriptorRef ↩
                                            >
                                </descriptorRefs>
                                <finalName>${project.artifactId}-${project.version}</ ↩
                                    finalName>
                                <appendAssemblyId>true</appendAssemblyId>
                        </configuration>
                        <executions>
                                <execution>
                                        <id>make-assembly</id>
                                        <phase>package</phase>
                                        <goals>
                                                <goal>single</goal>
                                        </goals>
                                </execution>
                        </executions>
                </plugin>
        </plugins>
</build>
```

The lines above tell maven to use version 2.7.1 of the Hadoop client library. The code to list the contents of an arbitrary directory provided on the command line looks like the following:

```
public class HdfsClient {

    public static void main(String[] args) throws IOException {
        if (args.length < 1) {
            System.err.println("Please provide the HDFS path (hdfs://hosthdfs:port/path)");
            return;
        }
        String hadoopConf = System.getProperty("hadoop.conf");
        if (hadoopConf == null) {
            System.err.println("Please provide the system property hadoop.conf");
            return;
        }
        Configuration conf = new Configuration();
        conf.addResource(new Path(hadoopConf + "/core-site.xml"));
        conf.addResource(new Path(hadoopConf + "/hdfs-site.xml"));
        conf.addResource(new Path(hadoopConf + "/mapred-site.xml"));
        conf.set("fs.hdfs.impl", org.apache.hadoop.hdfs.DistributedFileSystem.class.getName ↩
            ());
        conf.set("fs.file.impl", org.apache.hadoop.fs.LocalFileSystem.class.getName());
        FileSystem fileSystem = FileSystem.get(conf);
        RemoteIterator<LocatedFileStatus> iterator = fileSystem.listFiles(new Path(args[0]) ↩
            , false);
        while (iterator.hasNext()) {
            LocatedFileStatus fileStatus = iterator.next();
            Path path = fileStatus.getPath();
            System.out.println(path);
        }
    }
}
```

After a check if the necessary argument has been provided on the command line, a `Configuration` object is created. This is filled with the available Hadoop resources. Additionally the properties `fs.hdfs.impl` and `fs.file.impl` are set as it might happen when using the assembly plugin that the values get overridden. Finally the `FileSystem` object is retrieved from the static `get()` method and its `listFiles()` method is invoked. Iterating over the returned entries provides a list of files within the given directory:

```
$ java -Dhadoop.conf=/opt/hadoop/hadoop-2.7.1/etc/hadoop/ -jar target/hdfs-client-0.0.1- ↩
    SNAPSHOT-jar-with-dependencies.jar hdfs://localhost:9000/foo
hdfs://m1:9000/foo/helloWorld.txt
```

Chapter 4

MapReduce

4.1 MapReduce Architecture

Having a distributed file system to store huge amounts of data, a framework to analyze this data becomes necessary. MapReduce is such a framework that was first described by Jeffrey Dean and Sanjay Ghemawat (both working at Google). It basically consists of two functions: `Map` and `Reduce`.

The `Map` function takes as input a key/value pair and computes an intermediate key/value pair. Key and value of the intermediate pair can be completely different to those ones passed into the function, the only point to consider is that the MapReduce framework will group intermediate values with the same key together.

The `Reduce` function takes one key from the intermediate keys and all the values that belong to this key and computes an output value for this list of values. As the list of values can become quite huge, the framework provides an iterator for them such that not all values have to be loaded into memory.

The interesting point of this concept is that both the `Map` and the `Reduce` function are stateless. This way the framework can create an arbitrary amount of instances for each function and let them work concurrently. In conjunction with the HDFS file system this offers the possibility to start on each DataNode a `Map` instance that processes the data stored on this local DataNode.

Without having to transfer the input data to an external machine, the data processing can happen just in the same place where the data resides. Next to this, the intermediate results can also be stored as files in the HDFS file system and therefore used by the `Reduce` jobs. All together the complete framework can scale to huge amounts of data as with every DataNode added to the cluster also new computation power get available that can process that data stored on this new node.

A very simple example to gain a better understanding of the MapReduce approach is the word counting use case. In this case we assume that we have hundreds of text files and that we want to compute how often each word appears in these texts. The idea how to solve this problem with the MapReduce framework is to implement a `Map` function that takes as input the name of a text file as key and the contents of this file as value (name/content):

```
map(String key, String value):
        for word in value:
                submitIntermediate(word, 1)
```

Each invocation of `submitIntermediate()` produces an intermediate key/value pair with the word as key and the value "one":

```
"Hello"/1
"World"/1
...
"Hello"/1
"World"/1
...
```

As the MapReduce framework now groups all intermediate key/value pairs by the key, all key/value pairs for the key "World" are passed into the `Reduce` function. This function now only has the task to count the number of "one" values:

```
reduce(String key, Iterator values):
        result = 0
        for value in values:
                result += v
        submitFinal(result)
```

The final output will then look like this:

```
"Hello"/2
"World"/2
...
```

Sometimes it makes sense to introduce an additional phase after the `Map` function that is invoked on the same node and that already groups all intermediate key/value pairs with the same key together before they are transferred to the `Reduce` function. This phase is often called `Combine` phase and would in the example above be implemented similar to the `reduce` function.

The goal of this additional phase is to reduce the amount of data that is passed from one node to the other as in practice all values with the same key have to be transferred to the node that is executing the `Reduce` function for this key. In our example above there could be hundreds of intermediate ("Hello"/1) pairs that would have to be transferred to the `Reduce` node processing the key "Hello". If the `Combine` step before would have already reduced them, only one pair from each `Map` node for the key "Hello" would have to be transferred to the `Reduce` node.

Although the transportation of the intermediate key/value pairs to the `Reduce` nodes and their concatenation to one list for each key is part of the framework, it is often referred to has `Shuffle` phase. This phase can therefore be implemented and optimized once for all kinds of use cases.

4.2 MapReduce example

Now that we have learned the basics about the MapReduce framework, it is time to implement a simple example. In this tutorial we are going to create an "inverted index" from a couple of text files. "Inverted index" means that we create for each word a list of text files it appears in. This kind of index is used by search engines like Google, Elasticsearch or Solr in order to lookup all the pages that are listed for the search term. In real world usage the ordering of the list is the most important work, as we expect that the most "relevant" pages are listed first. In this example we are of course only implementing the first basic step that scans a set of text files and creates the corresponding "inverted index" for it.

We start by creating a maven project on the command line (and assume that maven is setup properly):

```
mvn archetype:create -DgroupId=ultimate-tutorials -DartifactId=mapreduce-example
```

This will generate the following structure in the file system:

```
|-- src
|   |-- main
|   |   `-- java
|   |       `-- ultimatetutorials
|   `-- test
|   |   `-- java
|   |       `-- ultimatetutorials
`-- pom.xml
```

We add the following lines to the file `pom.xml`:

```
<dependencies>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-client</artifactId>
        <version>2.7.1</version>
    </dependency>
</dependencies>
```

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <configuration>
                <archive>
                    <manifest>
                        <mainClass>ultimatetutorial.InvertedIndex</mainClass>
                    </manifest>
                </archive>
            </configuration>
        </plugin>
    </plugins>
</build>
```

The `dependencies` section defines the maven artifacts we are going to use (here: the `hadoop-client` library in version 2.7.1). In order to start our application without having to specify the main class on the command line, we define the main class using the `maven-jar-plugin`.

This main class looks like the following:

```
public class InvertedIndex {

    public static void main(String[] args) throws IOException, ClassNotFoundException,  ↩
        InterruptedException {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "inverted index");
        job.setJarByClass(InvertedIndex.class);
        job.setMapperClass(InvertedIndexMapper.class);
        job.setCombinerClass(InvertedIndexReducer.class);
        job.setReducerClass(InvertedIndexReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

After having constructed a `Configuration` object, a new `Job` instance is created by passing the configuration and the name of the job. Next we specify the class that implements the `MapReduce` job as well as the classes that implement the `Mapper`, `Combiner` and `Reducer`. As the `Combiner` step is basically only a `Reduce` step that is executed locally, we choose the same class as for the `Reducer` step. Now the MapReduce job needs to now of which type the output key and values are. As we are going to provide a list of documents for each term, we choose the `Text` class. Finally we provide the path for the input and output documents and start the job by calling `waitForCompletion`.

In the Map step of our MapReduce implementation we are going to split the input document into terms and create intermediate key/value pairs in the form (term/document):

```
public class InvertedIndexMapper extends Mapper<Object, Text, Text, Text> {
    private static final Log LOG = LogFactory.getLog(InvertedIndexMapper.class);

    @Override
    protected void map(Object key, Text value, Context context) throws IOException,  ↩
        InterruptedException {
        StringTokenizer tokenizer = new StringTokenizer(value.toString(), ".,; \\t\\n?!\\" ↩
            /()[]$%");
        while (tokenizer.hasMoreTokens()) {
            String token = tokenizer.nextToken();
            Text keyOut = new Text(token);
            String fileName = ((FileSplit) context.getInputSplit()).getPath().toString();
            Text valueOut = new Text(fileName);
            LOG.info("Key/Value: " + keyOut + "/" + valueOut);
            context.write(keyOut, valueOut);
        }
    }
}
```

The mapper implementation basically subclasses the `Mapper` class and overrides the `map()` method. Within this method the value (i.e. the document) is split into tokens and each token is written to the `Context` as key. Creating the tokens by using a `StringTokenizer` is of course a very simple way and not sufficient for many other types of documents, but in this example it demonstrates how to create different keys for one input key/value pair. The value of the intermediate key/value pair should be the file name of the document, hence we invoke the method `getInputSplit()` on the context, cast it to `FileSplit` and write the resulting path into the context.

The intermediate key/value pairs created by this `Mapper` implementation are then passed to the `Reducer`:

```
public class InvertedIndexReducer extends Reducer<Text,Text,Text,Text> {

    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context) throws ↩
        IOException, InterruptedException {
        List<String> valueList = new LinkedList<String>();
        for (Text value : values) {
            String valueAsString = new String(value.getBytes(), Charset.forName("UTF-8"));
            if (!valueList.contains(valueAsString)) {
                valueList.add(value.toString());
            }
        }
        StringBuilder sb = new StringBuilder();
        for (String value : valueList) {
            if (sb.length() > 0) {
                sb.append(",");
            }
            sb.append(value);
        }
        context.write(key, new Text(sb.toString()));
    }
}
```

Like the `Mapper`, the `Reducer` subclasses the corresponding class from the Hadoop framework and overrides the `reduce()` method. This `reduce` method gets for each intermediate key the list of values. In our case this is the list of document paths that contain the term (key). As this list can be so long that it does not fit into memory, the framework provides an iterator over the values. But in this simple example the list of documents will be short enough such that we can hold all document paths within a `LinkedList` in order to filter out duplicates. The output key/value pair is then just the term with a comma separated list of documents that contain this term.

After having compiled the source code using `mvn install`, we can load a set of documents to the HDFS. The example project contains a few text documents in the folder `src/test/resources` that can be loaded into the HDFS with the following command:

```
hdfs dfs -put src/test/resources/ input
```

This creates a new folder named `input` in the home directory of the `hadoop` user (assuming that the command is issued as hadoop user). If the folder already exists, it can be deleted by calling:

```
hdfs dfs -rm -r /user/hadoop/input
```

Now we can start the MapReduce job:

```
hadoop jar target/mapreduce-example-0.0.1-SNAPSHOT.jar /user/hadoop/input /user/hadoop/ ↩
    output
```

If the job finishes successfully, we will see a line similar to the following one in the output:

```
...
INFO mapreduce.Job: Job job_local498794269_0001 completed successfully
...
```

The following command tells us how the output files are named:

```
hadoop@m1:~/mapreduce-example$ hdfs dfs -ls output
Found 2 items
-rw-r--r--   1 hadoop supergroup          0 2016-01-01 14:55 output/_SUCCESS
-rw-r--r--   1 hadoop supergroup     121228 2016-01-01 14:55 output/part-r-00000
```

The file named `_SUCCESS` is only a marker file and does not contain any data. The actual output is stored in the file `part-r-00000`:

```
hadoop@m1:~/mapreduce-example$ hdfs dfs -cat output/part-r-00000
API     hdfs://localhost:9000/user/hadoop/input/4 ↩
    FreeWeatherProvidersAPIToDevelopAWeatherApp.txt
APIs    hdfs://localhost:9000/user/hadoop/input/ ↩
    WildflySwarmTowardsMaturityAndASmallContribution.txt
AWS     hdfs://localhost:9000/user/hadoop/input/ExampleOfUsingExtensionsInSwift.txt
About   hdfs://localhost:9000/user/hadoop/input/ExampleOfUsingExtensionsInSwift.txt
Accept  hdfs://localhost:9000/user/hadoop/input/RunningAnyDockerImageOnOpenshiftOrigin.txt
Add     hdfs://localhost:9000/user/hadoop/input/RunningAnyDockerImageOnOpenshiftOrigin.txt
After   hdfs://localhost:9000/user/hadoop/input/RunningAnyDockerImageOnOpenshiftOrigin.txt
All     hdfs://localhost:9000/user/hadoop/input/4 ↩
    FreeWeatherProvidersAPIToDevelopAWeatherApp.txt
Alma    hdfs://localhost:9000/user/hadoop/input/ExampleOfUsingExtensionsInSwift.txt
Amazon  hdfs://localhost:9000/user/hadoop/input/ExampleOfUsingExtensionsInSwift.txt
And     hdfs://localhost:9000/user/hadoop/input/UseJUnitsExpectedExceptionsSparingly.txt, ↩
    hdfs://localhost:9000/user/hadoop/input/MicroservicesUseCases.txt
...
```

As expected this file now contains an ordered list of all terms together with the paths to the documents that contain this word. So for example the term "And" is contained in the two documents "UseJUnitsExpectedExceptionsSparingly.txt" and "Microservice-sUseCases.txt".

Chapter 5

YARN

5.1 YARN Architecture

YARN (Yet Another Resource Negotiator) has been introduced to Hadoop with version 2.0 and solves a few issues with the resources scheduling of MapReduce in version 1.0. In order to understand the benefits of YARN, we have to review how resource scheduling worked in version 1.0.

A MapReduce job is split by the framework into tasks (Map tasks, Reducer tasks) and each task is run on of the DataNode machines on the cluster. For the execution of tasks, each DataNode machine provided a predefined number of slots (map slots, reducers slots). The JobTracker was responsible for the reservation of execution slots for the different tasks of a job and monitored their execution. If the execution failed, it reserved another slot and re-started the task. It also cleand up temoprary resources and make the reserved slot available to other tasks.

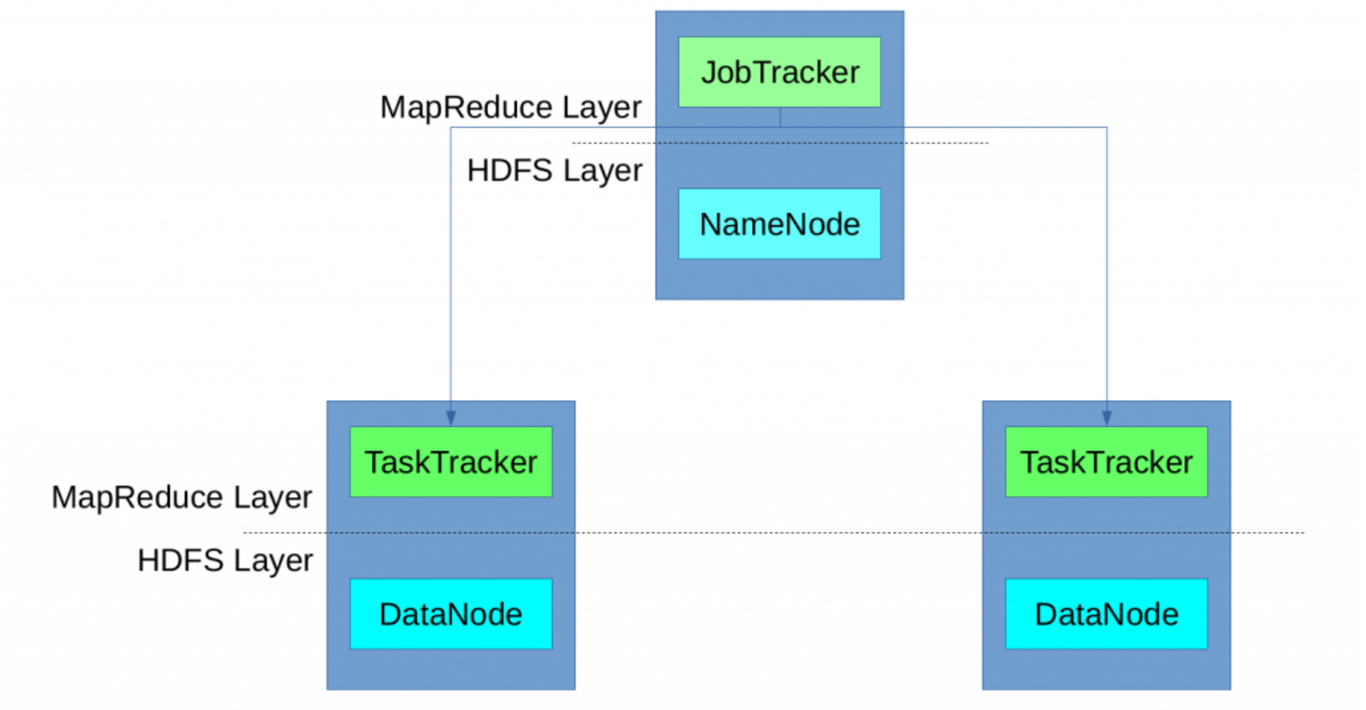


Figure 5.1: MapReduce

The fact that there was only one JobTracker instance in Hadoop 1.0 led to the problem that the whole MapReduce execution could fail, if the the JobTracker fails (single point of failure). Beyond that, having only one instance of the JobTracker limits scalability (for very large clusters with thousands of nodes).

The concept of predefined map and reduce slots also caused resource problems in case all map slots are used while reduce slots are still available and vice versa. In general it was not possible to reuse the MapReduce infrastructure for other types of computation like real-time jobs. While MapReduce is a batch framework, applications that want to process large data sets stored in HDFS and immediately inform the user about results cannot be implemented with it. Beneath the fact that MapReduce 1.0 did not offer real-time provision of computation results, all other types of applications that want to perform computations on the HDFS data had to be implemented as Map and Reduce jobs, which was not always possible.

Hence Hadoop 2.0 introduced YARN as resource manager, which no longer uses slots to manage resources. Instead nodes have "resources" (like memory and CPU cores) which can be allocated by applications on a per request basis. This way MapReduce jobs can run together with non-MapReduce jobs in the same cluster.

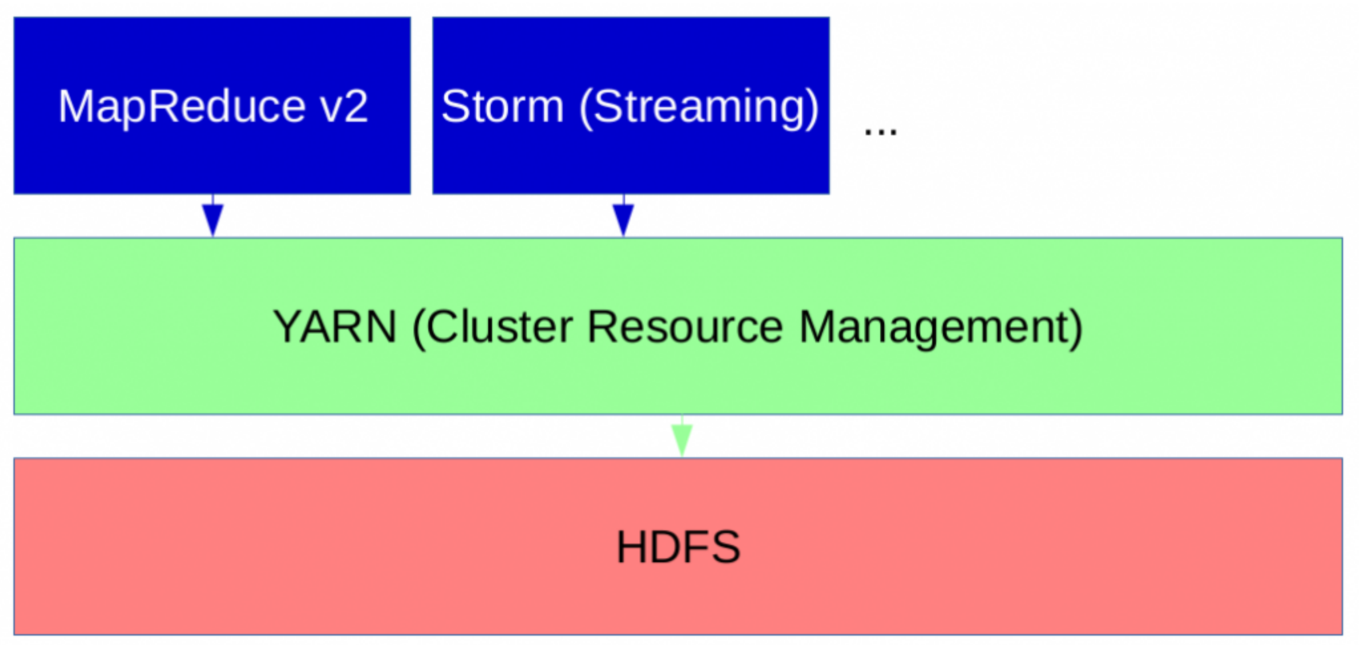


Figure 5.2: YARN

The heart of YARN is the Resource Manager (RM) which runs on the master node and acts as a global resource scheduler. It also arbitrates resources between competing applications. In contrast to the Resource Manager, the Node Managers (NM) run on slave nodes and communicate with the RM. The NodeManager is responsible for creating containers in which the applications run, monitors their CPU and memory usage and reports them to the RM.

Each application has its own ApplicationMaster (AM) which runs within a container and negotiates resources with the RM and works with the NM to execute and monitor tasks. The MapReduce implementation of Hadoop 2.0 therefore ships with an AM (named MRAppMaster) that requests containers for the execution of the map tasks from the RM, receives the container IDs from the RM and then executes the map tasks within the provided containers. Once the map tasks have finished, it requests new containers for the execution of the reduce tasks and starts their execution on the provided containers.

If the execution of a task fails, it is restarted by the ApplicationMaster. Should the ApplicationMaster fail, the RM will attempt to the restart the whole application (up to two times per default). Therefore the ApplicationMaster can signal if it supports job recovery. In this case the ApplicationMaster receives the previous state from the RM and can only restart incomplete tasks.

If a NodeManager fails, i.e the RM does not receive any heartbeats from it, it is removed from the list of active nodes and all its tasks are treated as failed. In contrast to version 1.0 of Hadoop, the ResourceManager can be configured for High Availability.

5.2 YARN Example

Now that we have learned about the architecture of YARN, it is time to execute a MapReduce job on YARN. Therefore we configure within the file `$HADOOP_HOME/etc/hadoop/mapred-site.xml` that the YARN framework should be used:

```
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
</configuration>
```

Then we add the following property to the file `$HADOOP_HOME/etc/hadoop/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>
```

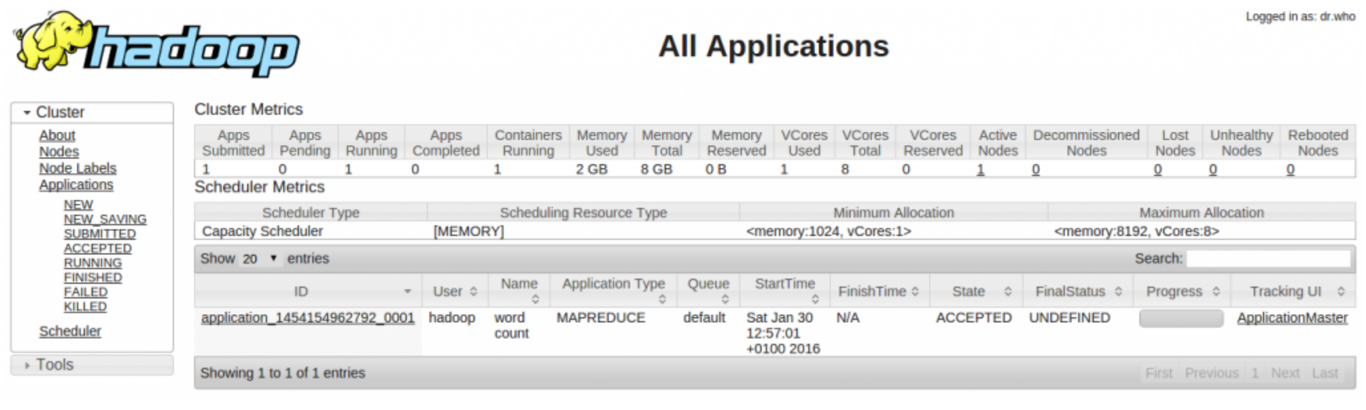
Now it is time to start the resource manager:

```
$ sbin/start-yarn.sh
starting yarn daemons
starting resourcemanager, logging to /opt/hadoop/hadoop-2.7.1/logs/yarn-hadoop- ↩
    resourcemanager-m1.out
localhost: starting nodemanager, logging to /opt/hadoop/hadoop-2.7.1/logs/yarn-hadoop- ↩
    nodemanager-m1.out
```

When we start a MapReduce job using YARN the output is a little different:

```
$ hadoop jar target/mapreduce-example-0.0.1-SNAPSHOT.jar input output
...
The url to track the job: http://m1:8088/proxy/application_1454013410608_0001/
Running job: job_1454013410608_0001
Job job_1454013410608_0001 running in uber mode : false
 map 0% reduce 0%
 map 86% reduce 0%
 map 100% reduce 0%
 map 100% reduce 100%
Job job_1454013410608_0001 completed successfully
```

As the log messages indicate, it is possible to track the jobs using the web interface running at port 8088:



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	1	2 GB	8 GB	0 B	1	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
application_1454154962792_0001	hadoop	word count	MAPREDUCE	default	Sat Jan 30 12:57:01 +0100 2016	N/A	ACCEPTED	UNDEFINED		ApplicationMaster

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

Figure 5.3: YARN RM

Chapter 6

Download

This was a tutorial on Apache Hadoop.

Download You can download the full source code of this tutorial here: **hadoop-tutorial-code.zip**