TP: Getting Started with *Hadoop*

Alexandru Costan

MapReduce has emerged as a leading programming model for data-intensive computing. It was originally proposed by Google to simplify development of web search applications on a large number of machines. Hadoop is a java open source implementation of MapReduce. The two fundamental subprojects are the Hadoop MapReduce framework and the HDFS.

HDFS is a distributed file system that provides high throughput access to application data. It is inspired by the GFS. HDFS has master/slave architecture. The master server, called NameNode, splits files into blocks and distributes them across the cluster with replications for fault tolerance. It holds all metadata information about stored files. The HDFS slaves, the actual store of the data blocks called DataNodes, serve read/write requests from clients and propagate replication tasks as directed by the NameNode.

The Hadoop MapReduce is a software framework for distributed processing of large data sets on compute clusters. It runs on the top of the HDFS. Thus data processing is collocated with data storage. It also has master/slave architecture.

The goal of this TP is to study the implementation and the operation of the Hadoop Platform. We will see how to deploy the platform and how to send and to retrieve the data to/from the HDFS. Finally we will run simple examples using the MapReduce paradigm.

Exercise 1: Installing Hadoop platform on your local machine

The goal of this exercise is to learn how to set up and configure a single-node Hadoop installation so that you can quickly perform simple operations using Hadoop MapReduce and the Hadoop Distributed File System (HDFS).

Hadoop can run in one of the three supported modes:

- **Local (Standalone) Mode**: running in a non-distributed mode, as a single Java process;
- **Pseudo-Distributed Mode**: running also on a single-node but each Hadoop daemon runs in a separate Java process;
- **Fully-Distributed Mode**: distributed on large-scale clusters.

In this TP we will deploy Hadoop in **Pseudo-Distributed Mode**.

Question 1.1

To avoid typing your password each time you start the Hadoop daemons, create (if not already in place) a couple of `ssh` keys using the `ssh-keygen` command and add the public key to the authorized keys.

```
ssh-keygen -q -N '' -f $HOME/.ssh/id_rsa
cat $HOME/.ssh/id_rsa.pub ≫ $HOME/.ssh/authorized_keys
ssh-add
```

Check the success of this step by running:
`ssh localhost`
Normally, you should be able to connect without being asked for a password.

Question 1.2

Download the Hadoop platform from (http://hadoop.apache.org/releases.html). Choose the latest stable version (currently 3.2.0) **binary** (not sources!). Extract the contents of `hadoop-3.2.0.tar.gz` in your home. To run, Hadoop needs a `JAVA_HOME` environment variable specifying the directory of `jvm`, and a `HADOOP_HOME` environment variable specifying the directory of Hadoop. Add the following lines in your ∼/`.bashrc` file located in your home directory (create it, if it doesn't exist):

```
export JAVA_HOME = <java path>
export HADOOP_HOME = <your Hadoop directory path>
export PATH = $PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin
```

To make these changes in your open terminal type:
`source ~/.bashrc`

Question 1.3

Before starting the Hadoop platform, edit its configuration files located in the `etc/hadoop` folder. As you are using your own machine, you must configure Hadoop as follows:

- In `etc/hadoop/core-site.xml` add:

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

  This indicates the name of the machine and the associated network port on which the Namenode process will run. You can check that the 9000 port on your machine is not already in use:
  `netstat -a | grep tcp | grep LISTEN | grep 9000`

- In `etc/hadoop/hdfs-site.xml` add:

```
<configuration>
   <property>
      <name>dfs.replication</name>
         <value>1</value>
      </property>
</configuration>
```

- In `etc/hadoop/mapred-site.xml` add:

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

This indicates the name of the resource manager used for the MapReduce application. In our case, this is of course Yarn.

- In `etc/hadoop/yarn-site.xml` add:

```
<configuration>
  <property>
    <name>yarn.nodemanager.aux-services</name>
      <value>mapreduce_shuffle</value>
    </property>
</configuration>
```

This property tells NodeManagers that there will be an auxiliary service called mapreduce.shuffle that they need to implement. Because NodeManagers won't shuffle data for a non-MapReduce job by default (i.e. remember they can run other job types as well, which might not need shuffle), we need to configure such a service for MapReduce.

Question 1.4

We will now start the platform:

- Format a new distributed-filesystem (HDFS):
  $ `hdfs namenode -format`

  If the command worked, you should see the following near the end of a long list of messages:

  `INFO common.Storage: Storage directory /tmp/hadoop-alex/dfs/name has been successfully formatted.`

  where the storage directory is the place where HDFS will be mounted on your local file-system.

- Start the hadoop daemons (HDFS and Yarn):
  $ `start-dfs.sh`
  $ `start-yarn.sh`

- The hadoop daemon logs output is written to the `$HADOOP_HOME/logs`. Check them.

- A nifty tool for checking whether the expected Hadoop processes are running is `jps` (displaying all java processes running). Normally you should see a process running (and its corresponding pid) for each hadoop instace: NodeManager, NameNode, SecondaryNameNode, ResourceManager and DataNode.)

Question 1.5

The web interfaces that show you the status of HDFS and YARN are available by default at:

- NameNode (status of HDFS) - http://localhost:9870/
- ResourceManager (status of your YARN cluster) - http://localhost:8088/

Question 1.6

To stop all the daemons running on your machine:
$ `stop-dfs.sh`
$ `stop-yarn.sh`

Exercise 2: Using the Hadoop Distributed File System (HDFS)

The goal of this exercise is to learn how to perform simple operations using the Hadoop Distributed File System (HDFS).

The HDFS shell is invoked by `hdfs dfs <args>`. All the FS shell commands take path `URIs` as arguments. The `URI` format is `scheme://autority/path`. For HDFS the `scheme` is `hdfs`, and for the local filesystem the scheme is `file`. The scheme and authority are optional. An HDFS file or directory such as `/parent/child` can be specified as

- `hdfs://namenodehost/parent/child` OR
- `/parent/child`

A selection of HDFS shell operations:

```
hdfs dfs -cat URI [URI ...] –Copies source paths to stdout.
hdfs dfs -copyFromLocal <localsrc> URI
hdfs dfs -copyToLocal [-ignorecrc] [-crc] URI <localdst>
hdfs dfs -get [-ignorecrc] [-crc] <src> <localdst>
hdfs dfs -ls <args>
hdfs dfs -mkdir <paths>
hdfs dfs -put <localsrc> ...  <dst>
hdfs dfs -rmr URI [URI ...]
```

Upload a few files and directories from your local file system to HDFS (i.e. using the `put` command). Browse through these files in the HDFS web interface.

Question 2.1

We will now generate three files LoremIpsum-128, LoremIpsum-256 and LoremIpsum-512. Their sizes are 128MB, 256MB and 512MB respectively. For this you use three times LoremIpsum and generator.sh script file provided in the TP archive.

- `./generator.sh LoremIpsum 128`

- `./generator.sh LoremIpsum 256`
- `./generator.sh LoremIpsum 512`

Now copy the three files to HDFS (`put`) and then show their content (`cat`) and finally remove (`rmr`) them.

Exercise 3: Running your first MapReduce programs

The goal of this exercise is to execute a few MapReduce examples, typically used for benchmarking, which come with the default Hadoop distribution.

Question 3.1

Run the wordcount example (application that counts the number of occurrences of each word in a given input set):

- `hadoop jar $HADOOP_PREFIX/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.2.0.jar wordcount /input /output`

where `input` is the directory in HDFS containing the input files (e.g. LoremIpsum-256) while the `output` is the directory in HDFS that will be created to store the results; `hadoop-mapreduce-examples-3.2.0.jar` is the `jar` archive containing the default examples that come with every Hadoop distribution.

Use the 256 MB data set and check the output. Then, inspect the log files generated by this job.

Question 3.2

Run the Pi estimator. This program estimates the value of Pi using a Monte Carlo method. Mapper: Generate points in a unit square and then count points inside/outside of the inscribed circle of the square. Reducer: Accumulate points inside/outside results from the mappers. Let numTotal = numInside + numOutside. The fraction numInside/numTotal is a rational approximation of the value (Area of the circle)/(Area of the square), where the area of the inscribed circle is Pi/4 and the area of unit square is 1. Then, Pi is estimated value to be 4(numInside/numTotal).

- `hadoop jar $HADOOP_PREFIX/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.2.0.jar pi maps samples_per_map`

where `maps` represents the number of map tasks used and `samples_per_map` the number of points processed by each map task.

Run the Pi estimator with 20 maps and 10,000 points per map. Check the log files.

Question 3.3

Run the grep example. This program searches plain-text data sets for lines matching a regular expression (the last argument of the command). You could use as input, for instance, the file hierarchy in your Hadoop directory.

- `hdfs dfs -put etc/hadoop input`
- `hadoop jar $HADOOP_PREFIX/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.0.0.jar grep /input /output 'dfs[a-z.]+'`

Check the output.

Exercise 4: Hadoop optimization

Question 4.1

The HDFS **chunk (block) size** is the smallest unit of data that the file system can store. This value has an influence on the MapReduce jobs execution. HDFS is meant to handle large files. If you have small files, smaller block sizes are better. If you have large files, larger block sizes are better.

Let's focus on the latter case. A large chunk size offers several important advantages. First, it reduces clients' need to interact with the master because reads and writes on the same chunk require only one initial request to the master for chunk location information. The reduction is especially significant for our workloads because applications mostly read and write large files sequentially. Second, since on a large chunk, a client is more likely to perform many operations on a given chunk, it can reduce network overhead by keeping a persistent TCP connection to the chunkserver over an extended period of time. Third, it reduces the size of the metadata stored on the master. This allows to keep the metadata in memory, which in turn brings several advantages. Finally, the chunk size directly impacts on the number of map tasks, as the input data is split into several blocks (of chunk size), each assigned to a map task.

Clearly, the chunk size is a crucial factor for Hadoop performance. Currently, the default block size in Hadoop is set to **128 MB**. To change the block size for **new files**, you can update the `conf/hdfs-site.xml` file:

```
<configuration>
  <property>
    <name>dfs.block.size</name>
      <value>67108864</value>
    </property>
</configuration>
```

Use the 256MB dataset and run some benchmarks (e.g. wordcount and pi) using different block sizes (i.e., 4, 32 and 128MB). Analyze the execution times, what do you conclude?

Question 4.2

In a Hadoop cluster, each task (e.g. a map task, a reduce task) runs in a separate Container. Each node can run several Containers in parallel. A set of system resources (CPU cores, RAM) are allocated for each container. It's vital to balance the usage of RAM, CPU

and disk so that processing is not constrained by any one of these cluster resources. As a general recommendation, allowing for 1-2 Containers per disk and per core gives the best balance for cluster utilization. In order to control the number of Containers running in parallel on each node we have to define the maximum resource allocation per Container.

Assume your machine has 16 GB of RAM (to find the amount of memory of your specific machine use the `free` command). Some of this RAM should be reserved for Operating System usage. On each node, we'll assign 8 GB RAM for YARN to use and keep 8 GB for the Operating System. The following property in the `yarn-site.xml` file sets the maximum memory YARN can utilize on the node:

```
<configuration>
  <property>
    <name>yarn.nodemanager.resource.memory-mb</name>
      <value>8192</value>
    </property>
</configuration>
```

The next step is to provide YARN guidance on how to break up the total resources available into Containers. You do this by specifying the minimum unit of RAM to allocate for a Container. We want to allow for a maximum of 8 Containers, and thus need (8 GB total RAM) / (8 Containers) = 1 GB minimum per container:

```
<configuration>
  <property>
    <name>yarn.scheduler.minimum-allocation-mb</name>
      <value>1024</value>
    </property>
</configuration>
```

YARN will allocate Containers with RAM amounts greater than the `yarn.scheduler.minimum-allocation-mb`.

When running MapReduce jobs, you can further set how much maximum memory each Map and Reduce task will take. Since each Map and each Reduce will run in a separate Container, these maximum memory settings should be at least equal to or more than the YARN minimum Container allocation. For our example cluster, we have the minimum RAM for a Container = 1 GB. We'll thus assign 2 GB for Map task Containers, and 4 GB for Reduce tasks Containers. In `mapred-site.xml`:

```
<configuration>
  <property>
    <name>mapreduce.map.memory.mb</name>
      <value>2048</value>
    </property>
  <property>
    <name>mapreduce.reduce.memory.mb</name>
      <value>4096</value>
    </property>
</configuration>
```

With the above configurations, YARN will be able to allocate on each node up to 4 mappers (8/2) or 2 reducers (8/4) or a permutation within that.

Don't forget to re-start Hadoop! Now, run the bencnhmarks using different Containers configuration (i.e., 1 mapper or 1 reducer vs. 4 mappers or 2 reducers). See the execution time, what do you conclude?