19 Putting into Practice: Large-Scale Data Management with HADOOP

The chapter proposes an introduction to HADOOP and suggests some exercises to initiate a practical experience of the system. The following assumes that you dispose of a Unix-like system (Mac OS X works just fine; Windows requires Cygwin). HADOOP can run in a pseudo-distributed mode which does not require a cluster infrastructure for testing the software, and the main part of our instructions considers this mode. Switching to a real cluster requires some additional configurations that are introduced at the end of the chapter. Since HADOOP is a relatively young system that steadily evolves, looking at the on-line, up-to-date documentation is of course recommended if you are to use it on a real basis. We illustrate HADOOP, MAPREDUCE and PIG manipulations on the DBLP data set, which can be retrieved from the following URL:

http://dblp.uni-trier.de/xml/

Download the *dblp.dtd* and *dblp.xml* files. The latter is, at the time of writing, almost 700 MB. Put both files in a *dblp* directory. In addition, you should take some smaller files extracted from the DBLP archive, that we make available on the book web site.

The content of the *dblp* directory should be similar to the following:

```
ls -l dblp/
total 705716
-rw-r--r-- 1 webdam webdam    108366  author-medium.txt
-rw-r--r-- 1 webdam webdam     10070  author-small.txt
-rw-r--r-- 1 webdam webdam      7878  dblp.dtd
-rw-r--r-- 1 webdam webdam 720931885  dblp.xml
-rw-r--r-- 1 webdam webdam    130953  proceedings-medium.txt
-rw-r--r-- 1 webdam webdam     17151  proceedings-small.txt
```

19.1 Installing and running HADOOP

First, get a stable release from the HADOOP site (`http://hadoop.apache.org/`) and unpack the archive on your machine. In order to set up your environment, you need a `HADOOP_HOME` variable that refers to the HADOOP installation directory. For instance:

```
export HADOOP_HOME=/users/webdam/hadoop
```

This can also be set up in the */path/to/hadoop/conf/hadoop-env.sh* script. HADOOP features a command-line interpreter, written in Java, named *hadoop*. Add the `HADOOP_HOME/bin` directory to your path, as follows:

```
export PATH=$PATH:$HADOOP_HOME/bin
```

You should be able to run `hadoop`:

```
bash-3.2$ hadoop version
Hadoop 0.20.2
```

Now, you are ready to make some preliminary tests with HADOOP. We will begin with simple filesystem manipulations, in a mode called "pseudo-distributed" which runs the HADOOP servers on the local machine. In order to set up this mode, you need first to edit the *conf/core-site.xml* configuration file (all relative paths are rooted at `$HADOOP_HOME`) to add the following parameters.

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

This tells HADOOP that the current installation runs with a local HDFS Master node (the "NameNode" in HADOOP terminology) on the port 9000. The file system is initialized with the `format` command:

```
$ hadoop namenode -format
```

This initializes a directory in */tmp/hadoop-<username>/dfs/name*. The *hadoop* program is a Java command-line interpreter used to execute HADOOP commands. The general usage is:

```
$ hadoop <command> [parameters]
```

where `command` is one of `namenode` (commands sent to the Master node), `fs` (filesystem commands), `job` (commands that control MAPREDUCE jobs), etc.

Once the file system is formatted, you must launch the HDFS Master node (*namenode*). The name node is a process responsible for managing file server nodes (called *datanodes* in HADOOP) in the cluster, and it does so with *ssh* commands. You must check that SSH is properly configured and, in particular, that you can log in the local machine with SSH without having to enter a passphrase. Try the following command:

```
$ ssh localhost
```

If it does not work (or if you are prompted for a passphrase), you must execute the following commands:

```
$ ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

This generates a new SSH key with an empty password. Now, you should be able to start the namenode server:

```
$ start-dfs.sh &
```

This launches a namenode process, and a datanode process on the local machine. You should get the following messages

```
starting namenode, logging to (...)
localhost: starting datanode, logging to (...)
localhost: starting secondarynamenode, logging to (...)
```

The secondary namenode is a mirror of the main one, used for failure recovery. At this point you can check that the HDFS is up and running with

```
$ hadoop fs -ls /
```

and look for errors If anything goes wrong, you must look at the log files that contain a lot of report messages on the initialization steps carried out by the start procedure. Once the servers are correctly launched, we can copy the *dblp* directory in the HDFS file system with the following command:

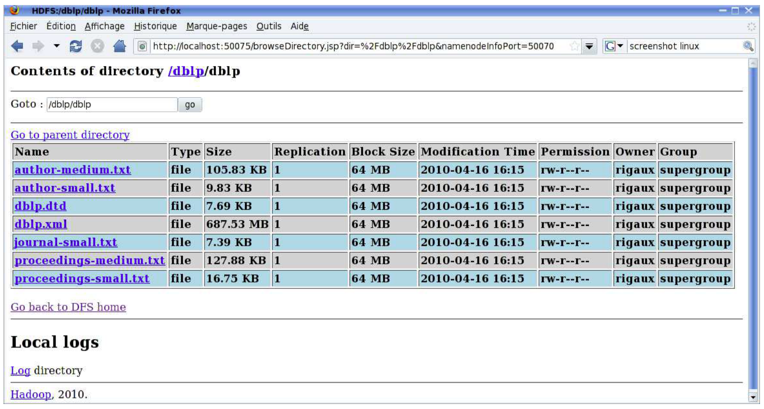
```
$ hadoop fs -put dblp/ /dblp
```

This creates a *dblp* directory under the root of the HDFS filesystem hierarchy. All the basic filesystem commands can be invoked through the *hadoop* interface. Here is a short session that shows typical Unix-like file manipulations.

```
$ hadoop fs -ls /dblp/dblp*
Found 2 items
-rw-r--r--   1 wdmd supergroup       7878 2010-03-29 10:40 /DBLP/dblp.dtd
-rw-r--r--   1 wdmd supergroup  719304448 2010-03-29 10:40 /DBLP/dblp.xml
$ hadoop fs -mkdir /DBLP/dtd
$ hadoop fs -cp /DBLP/dblp.dtd  /DBLP/dtd/
$ hadoop fs -ls /DBLP/dtd
Found 1 items
-rw-r--r--   1 wdmd supergroup       7878 2010-03-29 10:55 /DBLP/dtd/dblp.dtd
$ hadoop fs -get /DBLP/dtd/dblp.dtd my.dtd
bash-3.2$ ls -l my.dtd
-rw-r--r--  1 wdmd  staff  7878 29 mar 10:56 my.dtd
```

The output of the `ls` command is quite similar to the standard Unix one, except for the second value of each line that indicates the replication factor in the distributed file system. The value is, here, 1 (no replication), the default parameter that can be set in the *conf/hdfs-site.xml* configuration file.

```
<?xml version="1.0"?>
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
</configuration>
```


HDFS:/dblp/dblp - Mozilla Firefox
Fichier Édition Affichage Historique Marque-pages Outils Aide
http://localhost:50075/browseDirectory.jsp?dir=%2Fdblp%2Fdblp&namenodeInfoPort=50070
screenshot linux

Contents of directory /dblp/dblp

Goto : /dblp/dblp go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
author-medium.txt	file	105.83 KB	1	64 MB	2010-04-16 16:15	rw-r--r--	rigaux	supergroup
author-small.txt	file	9.83 KB	1	64 MB	2010-04-16 16:15	rw-r--r--	rigaux	supergroup
dblp.dtd	file	7.69 KB	1	64 MB	2010-04-16 16:15	rw-r--r--	rigaux	supergroup
dblp.xml	file	687.53 MB	1	64 MB	2010-04-16 16:15	rw-r--r--	rigaux	supergroup
journal-small.txt	file	7.39 KB	1	64 MB	2010-04-16 16:15	rw-r--r--	rigaux	supergroup
proceedings-medium.txt	file	127.88 KB	1	64 MB	2010-04-16 16:15	rw-r--r--	rigaux	supergroup
proceedings-small.txt	file	16.75 KB	1	64 MB	2010-04-16 16:15	rw-r--r--	rigaux	supergroup

Go back to DFS home

Local logs

Log directory

Hadoop, 2010.

Figure 19.1: Browsing the HDFS file system

The Namenode also instantiates a rudimentary web server at *http://localhost:50070/*. It shows some information on the current status of the file system and provides a simple Web interface to browse the file hierarchy. Figure 19.1 shows a screen shot of this interface, with the list of our sample files loaded in the HDFS server. Note the replication factor (here, 1) and the large block size of 64 MBs.

19.2 Running MAPREDUCE jobs

We can now run MAPREDUCE job to process data files stored in HDFS. The following gives first an example that scans text files extracted from the DBLP data set. We then suggest some improvements and experiments. You must first start the MAPREDUCE servers:

```
start-mapred.sh
```

Our example processes data files extracted from the DBLP data set and transformed in flat text files for simplicity. You can take these data inputs of various sizes, named *authors-xxx.txt* from the book web site, along with the Java code. The smallest file size is a few KBs, the largest 300 MBs. These are arguably small data sets for HADOOP, yet sufficient for an initial practice.

The file format is pretty simple. It consists of one line for each pair (author, title), with tab-separated fields, as follows.

```
<author name>  <title> <year>
```

Our MAPREDUCE job counts the number of publications found for each author. We decompose the code in two Java files, available as usual from the site. The first one, below, provides an implementation of both the MAP and REDUCE operations.

```
package myHadoop;

/**
 * Import the necessary Java packages
 */

import java.io.IOException;
import java.util.Scanner;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;

/**
 * A Mapreduce example for Hadoop. It extracts some basic
 * information from a text file derived from the DBLP data set.
 */
public class Authors {

  /**
   * The Mapper class -- it takes a line from the input file and
   * extracts the string before the first tab (= the author name)
   */
  public static class AuthorsMapper extends
          Mapper<LongWritable, Text, Text, IntWritable> {

        private final static IntWritable one = new IntWritable(1);
        private Text author = new Text();

        public void map(LongWritable key, Text value, Context context)
                throws IOException, InterruptedException {

          /* Open a Java scanner object to parse the line */
          Scanner line = new Scanner(value.toString());
          line.useDelimiter("\t");
          author.set(line.next());
          context.write(author, one);
        }
  }

  /**
   * The Reducer class -- receives pairs (author name, <list of counts>)
   * and sums up the counts to get the number of publications per author
   */
  public static class CountReducer extends
          Reducer<Text, IntWritable, Text, IntWritable> {
        private IntWritable result = new IntWritable();

        public void reduce(Text key, Iterable<IntWritable> values,
              Context context)
                throws IOException, InterruptedException {

          /* Iterate on the list to compute the count */
          int count = 0;
          for (IntWritable val : values) {
                count += val.get();
          }
          result.set(count);
          context.write(key, result);
        }
  }
}
```

HADOOP provides two abstract classes, `Mapper` and `Reducer`, which must be extended and specialized by the implementation of, respectively, a *map()* and *reduce()* methods. The formal parameters of each abstract class describe respectively the types of the input key, input value, output key and output value. The framework also comes with a list of serializable data types that must be used to represent the values exchanged in a MAPREDUCE workflow. Our example relies on three such types: `LongWritable` (used for the input key, i.e., the line number), `IntWritable` (used for counting occurrences) and `Text` (a generic type for character strings). Finally, the `Context` class allows the user code to interact with the MAPREDUCE system.

So, consider first the *map()* method of our (extended) `Mapper` class `AuthorMapper`. It takes as input pairs *(key, value)*, *key* being here the number of the line from the input file (automatically generated by the system, and not used by our function), and *value* the line itself. Our code simply takes the part of the line that precedes the first tabulation, interpreted as the author name, and produces a pair *(author, 1)*.

The *reduce()* function is almost as simple. The input consists of a key (the name of an author) and a list of the publication counts found by all the mappers for this author. We simply iterate on this list to sum up these counts.

The second Java program shows how a job is submitted to the MAPREDUCE environment. The comments in the code should be explicit enough to inform the reader. An important aspect is the `Configuration` object which loads the configuration files that describe our MAPREDUCE setting. The same job can run indifferently in local or distributed mode, depending on the configuration chosen at run time. This allows to test a (map, reduce) pair of functions on small, local data sets, before submitted a possibly long process.

```
package myHadoop;

/**
 * Example of a simple MapReduce job: it reads
 * file containing authors and publications, and
 * produce each author with her publication count.
 */

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import myHadoop.Authors;

/**
 * The follozing class implements the Job submission, based on
 * the Mapper (AuthorsMapper) and the Reducer (CountReducer)
 */
public class AuthorsJob {

  public static void main(String[] args) throws Exception {

        /*
         * Load the Haddop configuration. IMPORTANT: the
         * $HADOOP_HOME/conf directory must be in the CLASSPATH
         */
        Configuration conf = new Configuration();

        /* We expect two arguments */

        if (args.length != 2) {
          System.err.println("Usage: AuthorsJob <in> <out>");
          System.exit(2);
        }

        /* Allright, define and submit the job */
        Job job = new Job(conf, "Authors count");

        /* Define the Mapper and the Reducer */
        job.setMapperClass(Authors.AuthorsMapper.class);
        job.setReducerClass(Authors.CountReducer.class);

        /* Define the output type */
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        /* Set the input and the output */
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        /* Do it! */
        System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

The second object of importance is the instance of `Job` that acts as an interface with the MAPREDUCE environment for specifying the input, the output, and the *map()* and *reduce()* functions. Our example presents the bare minimal specification.

The Job can be directly run as `java AuthorsJob <inputfile> <outputdir>`. It produces an output directory `outputdir` (which must not exist prior to the job execution) with a set of files, one for each reducer, containing the result. Be sure to add all the HADOOP Jar files (found in HADOOP home directory) in your `CLASSPATH` before running the job. Here is for instance a part of the result obtained by processing the *author-small.txt* data file:

```
(...)
Dominique Decouchant    1
E. C. Chow      1
E. Harold Williams      1
```

```
Edward Omiecinski       1
Eric N. Hanson  1
Eugene J. Shekita       1
Gail E. Kaiser  1
Guido Moerkotte 1
Hanan Samet     2
Hector Garcia-Molina    2
Injun Choi      1
(...)
```

Note that the authors are alphabetically ordered, which is a desirable side effect of the map reduce framework. Sorting is done during the shuffle phase to bring together intermediate pairs that share the same key value.

19.3 PIGLATIN scripts

PIGLATIN can be found at *http://hadoop.apache.org/pig/*. Download a recent and stable archive, and uncompress it somewhere (say, in *$home/pig*). Add the *pig/bin* subdirectory to your path variable (change the directories to reflect your own setting):

```
$ export PATH=$HOME/pig/bin:$PATH
```

Check that the command-line interface is ready. The `pig -x local` should produce the following output.

```
$ pig -x local
[main] INFO  org.apache.pig.Main - Logging error messages to: xxx.log
grunt>
```

PIG accepts commands from an interpreter called `grunt` which runs either in “local” mode (files are read from the local filesystem) or “MAPREDUCE” mode. The former is sufficient to take a grasp on the main features of PIGLATIN. It is also useful for testing new scripts that could run for hours on large collections.

We refer to the presentation of PIGLATIN that can be found in the chapter devoted to distributed computing. Running PIGLATIN with the command line interpreter is a piece of cake. As an initial step, we invite the reader to run a script equivalent to the MAPREDUCE Job described in the previous section.

19.4 Running in cluster mode (optional)

We give now some hints to run HADOOP in a real cluster. As far as experimental data manipulations are involved, this is not really necessary, because neither the principles nor the code change depending on the pseudo-distributed mode. If you need to process really large data sets, and/or use HADOOP in real-life environment, a real cluster of machines is of course required. Understanding the architecture of a real HADOOP cluster may also be interesting on its own. You need, of course, at least two connected machines, preferably sharing a regular distributed filesystem like NFS, with system administration rights. Before going further, please note that if your objective is real-life processing of large data sets, you do not need to set up your own cluster, but can (at least for some non-committing experiments) use a cloud computing environment supporting HADOOP, e.g., Amazon Web Services (*http://aws.amazon.com*) or Cloudera (*http://www.cloudera.com*) to name a few.

19.4.1 Configuring HADOOP in cluster mode

Most of the parameters that affect the running mode of HADOOP are controlled from the configuration files located in the *conf* directory. In order to switch easily from one mode to the other, you can simply copy *conf* as (say) *conf-cluster*. The choice between the two configurations is set by the environement variable `HADOOP_CONF_DIR`. Set this variable to the chosen value:

```
export HADOOP_CONF_DIR=$HADOOP_HOME/conf-cluster
```

For simplicity, we assume that all the nodes in the cluster share the same configuration file (accessible thanks to a NFS-like distribution mechanism). If your machines are heterogeneous, you may have to refine the configuration for each machine.

The *slaves* file contains the list of nodes in the cluster, referred to by their name or IP. Here is for instance the content of *slaves* for a small 10-node cluster located at INRIA:

```
node1.gemo.saclay.inria.fr
node2.gemo.saclay.inria.fr
node3.gemo.saclay.inria.fr
...
node10.gemo.saclay.inria.fr
```

There exists a *masters* file, which contains the name of the secondary Nameserver. You can leave it unchanged.

Before attempting to start your cluster, you should look at the XML configuration file *core-site.xml*, *hdfs-site.xml* and *mapred-site.xml*. They contain parameter (or "properties") relative respectively to core HADOOP, HDFS and MAPREDUCE. Here is for instance a self-commented *hdfs-site.xml* file with some important properties.

```
 <?xml version="1.0"?>

<configuration>
  <! Amount of replication of
                 each data chunk -->
  <property>
      <name>dfs.replication</name>
       <value>3</value>
    </property>

    <!-- Disk(s) and directory/ies
              for filesystem info -->
  <property>
      <name>dfs.name.dir</name>
      <value>/disk1/hdfs/name</value>
    </property>
  <!-- Disk(s) and directory/ies
            for data chunks -->
<property>
    <name>dfs.data.dir</name>
    <value>/disk1/hdfs/data</value>
  </property>
</configuration>
```

19.4.2 Starting, stopping and managing HADOOP

HADOOP servers are launched with the *start-dfs.sh* script (located in *bin*). It starts the Namenode on the local machine (that is, the machine the script is run on), one datanode on each of the machines listed in the *slaves* file, and a secondary Namenode on the machine listed in the *masters* file. These scripts report error messages in log files located in the *HADOOP_HOME/logs* directory. Before your cluster is up and running, you will probably have to inspect these files more than once to find and correct the (hopefully small) problems specific to your environment. The HDFS system is of course halted with *stop-dfs.sh*.

A MAPREDUCE environment is launched with the *start-mapred.sh* script which starts a JobTracker (a MAPREDUCE Master node) on the local machine, and a tasktracker (the Workers in Google MAPREDUCE terminology) on the machines listed in *slaves*.

Another useful script is *hadoop-env.sh* where many parameters that affect the behavior of HADOOP can be set. The memory buffer used by each node is for instance determined by `HADOOP_HEAPSIZE`. The list of these parameters goes beyond the scope of this introduction: we refer the reader to the online documentation.

19.5 Exercises

If you succesfully managed to run the above examples, you are ready to go further in the discovery of HADOOP and its associated tools.

Exercise 19.5.1 (Combiner functions) *Once a map() function gets executed, it stores its result on the local filesystem. This result is then transferred to a Reducer. The performance of the Job may therefore be affected by the size of the data. A useful operation is thus to limit the size of the* MAP *result before network transmission:* HADOOP *allows the specification of Combiner functions to this end. This can be seen as performing locally (that is, on the Mapper) a part of the* REDUCE *task at the end of the* MAP *phase.*

Not all Jobs are subject to Combiner optimization. Computing the average of the intermediate pairs value for instance can only be done by the Reducer. In the case of associate functions like count(), a Combiner is quite appropriate. The exercise consists in defining a Combiner function for the MAPREDUCE *job of Section 19.2. We let the reader investigate the* HADOOP *documentation (in particular Java APIs) to learn the interface that allows to define and run Combiner functions.*

Exercise 19.5.2 *Consider the XML files representing movies. Write* MAPREDUCE *jobs that take these files as input and produce the following flat text files with tab-separated fields:*

- `title-and-actor.txt`: *each line contains the title, the actor's name, year of birth and role. Example:*

```
The Social network  Jesse Eisenberg 1983  Mark Zuckerberg
The Social network  Mara Rooney   1985  Erica Albright
Marie Antoinette    Kirsten Dunst 1982  Marie-Antoinette
```

- `director-and-title.txt`: *each line contains the director's name and the movie title. Example:*

```
David Fincher   The Social network 2010
Sofia Coppola   Lost in translation 2003
David Fincher   Seven              1995
```

You must write an input function that reads an XML file and analyzes its content with either SAX or DOM: refer to the PiP chapter on the XML programming APIs.

Exercise 19.5.3 *Run the following* PIGLATIN *queries on the files obtained from the previous exercise.*

1. *Load* `title-and-actor.txt` *and group on the title. The actors (along with their roles) should appear as a nested bag.*
2. *Load* `director-and-title.txt` *and group on the director name. Titles should appear as a nested bag.*
3. *Apply the* ***cogroup*** *operator to associate a movie, its director and its actors from both sources.*
4. *Write a* PIG *program that retrieves the actors that are also director of some movie: output a tuple for each artist, with two nested bags, one with the movies s/he played a role in, and one with the movies s/he directed.*
5. *write a modified version that looks for artists that were both actors and director of a same movie.*

Exercise 19.5.4 (Inverted file project) *The goal of the project is to build a simple inverted file using a* MAPREDUCE *job. You can either use Java programming, or* PIG *programs with a few additional functions used to process character strings and compute* tf *and* idf *indicators.*

Design and implement a process that takes as input a set of text files (consider for instance the abstracts of our movies collection) and outputs a list of the terms found in the texts, along with their frequency. Associate to each term the list of documents, along with the idf *indicator.*