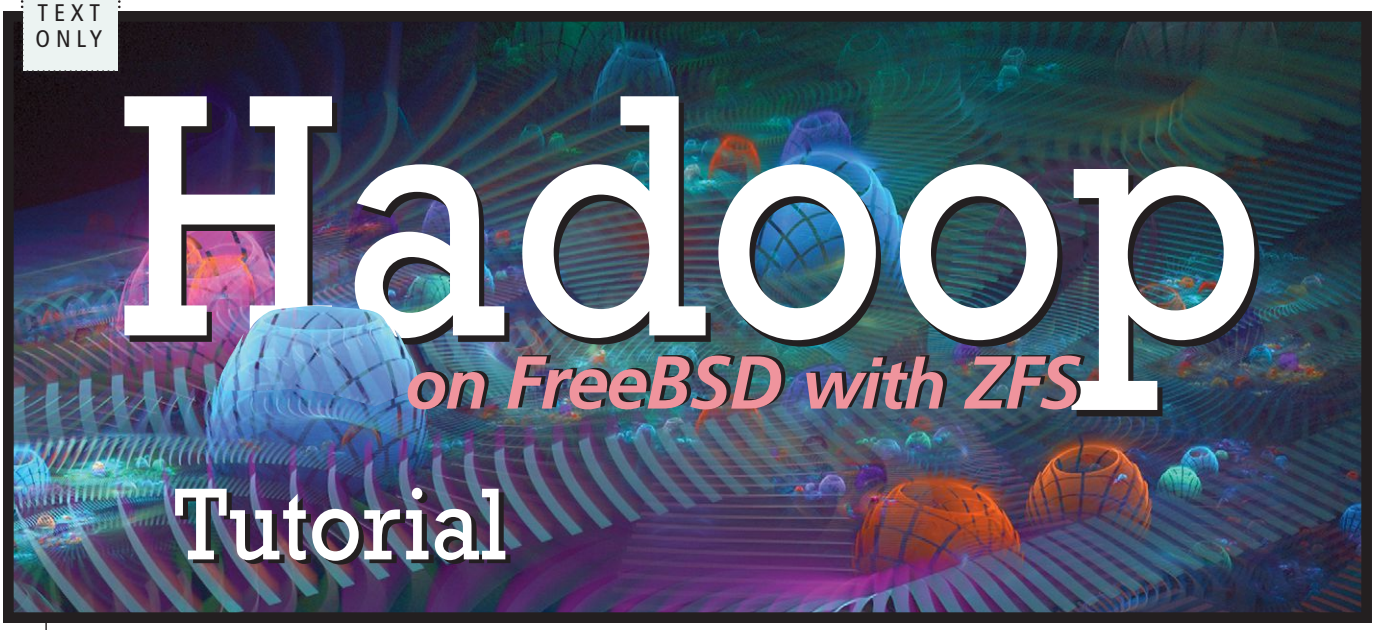
SEE TEXT ONLY



Hadoop

on FreeBSD with ZFS

Tutorial

By Benedict **Reuschling**

This article provides a scripted tutorial using an Ansible playbook to build the cluster from the ground up. Manual installation and configuration quickly becomes tedious the more nodes are involved, which is why automating the installation removes some of the burden for BSD system administrators. The tutorial builds hadoop on ZFS to make use of the powerful features of that filesystem with an integrated volume manager to enhance HDFS's feature set.

Hadoop is an open-source, distributed framework using the map-reduce programming paradigm to split big computing jobs into smaller pieces. Those pieces are then distributed to all the nodes in the cluster (map step), where the participating datanodes calculate parts of the problem in parallel. In the reduce step, those partial results are collected and the final result is computed. Results are stored in the hadoop distributed filesystem (HDFS). Coordination is done via a master node called the namenode. Hadoop consists of many components and can run on commodity hardware without requiring many resources. The more nodes that participate in the cluster and if the problem can be expressed in map-reduce terms, the better the performance than just running the calculation on a single node. Mostly written in Java, hadoop aims to provide enough redundancy to allow nodes to fail while still maintaining a functional compute cluster. A rich ecosystem of additional software grew up around hadoop, which makes the task of setting up a cluster of hadoop machines for big data applications a difficult one.

Requirements

This tutorial uses three FreeBSD 11.2 machines, either physical or virtual. Other and older BSD versions should work as well, as long as they support a recent version of Java/OpenJDK. OpenZFS with regular directories is fine, If OpenZFS is not available, regular directories are fine, too. One machine will serve as the master node (called namenode) and the other two will serve as compute nodes (or datanodes in hadoop terms). They need to be able to connect to each other on the network. Also, an Ansible setup must be available for the playbook work. This involves an inventory file that contains the three machines and the necessary software on the target machines (python 2.7 or higher) for Ansible to send commands to them.

Note that this playbook does not use the default paths used by the FreeBSD port/package of hadoop. This way, a higher version of hadoop can be used before the port gets updated. The default FreeBSD paths can be easily substituted when required. The configuration files presented in this tutorial contain only the minimal sections required to get a basic hadoop setup going. The FreeBSD port/package contains sample configuration files that have many more configuration options than are initially needed. However, the port is a great resource for extending and learning about the hadoop cluster once it is set up.

Readers unfamiliar with Ansible should be able to abstract the setup steps and either implement them with a different configuration management system (puppet, chef, saltstack) or execute the steps manually.

Defining Playbook Variables

To make management easier, a separate `vars.yml` file that holds all the variables is used. This file contains all the information in a central location that is needed to install the cluster. For example, when a higher version of hadoop should be used, only the `hdp_ver` variable must be changed.

```
java_home: "/usr/local/openjdk8"
hdp: "hadoop"
hdp_zpool_name: {{hdp}}pool
hdp_ver: "2.9.0"
hdp_destdir: "/usr/local/{{hdp}}{{hdp_ver}}"
hdp_home: "/home/{{hdp}}"
hdp_tmp_dir: "/{{hdp}}/tmp"
hdp_name_dir: "/{{hdp}}/hdfs/namenode"
hdp_data_dir: "/{{hdp}}/hdfs/datanode"
hdp_mapred_dir: "/{{hdp}}/mapred.local.dir"
hdp_namesec_dir: "/{{hdp}}/namesecondary"
hdp_keyname: "my_hadoop_key"
hdp_keytype: "ed25519"
hdp_user_password: "{{vault_hdp_user_pw}}"
```

File: `vars.yml`

The first line stores the location of the installed OpenJDK from ports. To save a bit of typing in the playbook and to replace common occurances of the word hadoop, a shorter variable `hdp` is used as a prefix to all the rest of the variables. The zpool name (`hadooppool` in this tutorial) already makes use of the variable we defined for hadoop's name. As mentioned above, the hadoop version is used to keep track of which version this cluster is based on. The variables describe ZFS datasets (or directories) for the hadoop user that the software is using while runnning. The last couple of lines are defining the SSH key that hadoop needs to securely connect between the nodes of the cluster. Secrets like passwords are also stored for the hadoop user in Ansible-vault. This way, the playbook can be shared with others without exposing the passwords set for that individual cluster.

To create a new vault, the `ansible-vault(1)` command with the `create` subcommand is used, followed by the path where the encrypted vault file should be stored.

```
$ ansible-vault create vault.yml
```

After being being prompted to create a passphrase to open the vault, an editor is opened in the vault file and secrets can be stored within. Refer to (https://docs.ansible.com/ansible/latest/reference_appendices/faq.html#how-do-i-generate-crypted-passwords-for-the-user-module) on how to generate encrypted passwords that the Ansible user module can understand. The line in the vault should look like this, with `<password>` replaced by the password hash:

```
vault_hdp_user_pw: "<yourpassword>"
```

File: `vault.yml`

Playbook Contents

The playbook itself is divided into several sections to help better understand what is being done in each of them. The first part is the beginning of the playbook where it describes what the playbook does (`name`), which hosts to work on (`hosts`), and where the variables and the vault are stored (`vars_files`):

```
#!/usr/local/bin/ansible-playbook
- name: "Install a {{hdp}} {{hdp_ver}} multi node cluster"
    hosts: "{{host}}"

    vars_files:
    - vault.yml
    - vars.yml
```

The first line will ensure that the playbook can run like a regular shell script by making it executable (`chmod +x`). The `name:` describes what this playbook is doing and uses the variables defined in `vars.yml`. The hosts are provided on the commandline later to make it more flexible to add more machines. Alternatively, when there is a predetermined number of hosts for the cluster, they can also be entered in the `hosts:` line.

Next, the tasks that the playbook should execute are defined (be careful not to use tabs for indentations, this is YAML syntax):

```
  tasks:
    - name: "Install required software for {{hdp}}"
      package:
        name: "{{item}}"
      with_items:
        - openjdk8
        - bash
        - gtar
```

The first task is to install OpenJDK from FreeBSD packages, bash for the hadoop user's shell, and gtar to extract the source tarball (the `unarchive` step later on) that was downloaded from the hadoop website.

The datasets (or directories if ZFS can not be used) are created in the next step:

```
- name: "Create ZFS datasets for the {{hdp}} user"
  zfs:
    name: "{{hdp_zpool_name}}{{item}}"
    state: present
    extra_zfs_properties:
      mountpoint: "{{item}}"
      recordsize: "1M"
      compression: "lz4"
    with_items:
      - "{{hdp_home}}"
      - "/opt"
      - "{{hdp_tmp_dir}}"
      - "{{hdp_name_dir}}"
      - "{{hdp_data_dir}}"
      - "{{hdp_namesec_dir}}"
      - "{{hdp_mapred_dir}}"
      - "{{hdp_zoo_dir}}"
```

The datasets are each using LZ4 for compression and are able to use a record size of up to 1 megabyte. This is important to increase compression as the hadoop distributed filesystem (HDFS) is using 128MB records by default. The paths to the mount points are defined in the `vars.yml` file and will be used in the hadoop-specific config files again later on.

```
- name: "Create the {{hdp}} User"
  user:
    name: "{{hdp}}"
    comment: "{{hdp}} User"
    home: "{{hdp_localhome}}/{{hdp}}"
    shell: /usr/local/bin/bash
    createhome: yes
    password: "{{vault_hdp_user_pw}}"
```

The hadoop processes should all be started and run under a separate user account, aptly named hadoop. This task will create that designated user in the system. The result looks like the following in the password database (the user ID might be a different one):

```
$ grep hadoop /etc/passwd
hadoop:*:31707:31707:hadoop User:/home/hadoop:/usr/local/bin/bash
```

Next, the SSH keys need to be distributed for the hadoop user to be able to log into each cluster machine without requiring a password. Ansible's lookup functionality is used to read an SSH key that was generated earlier on the machine running the playbook (it is recommended to generate this kind of separate key for hadoop using `ssh-keygen`). The SSH key must not have a passphrase, as the hadoop processes will perform the logins without any user interaction to enter it. The task will add the SSH public key to the `authorized_keys` file in `/home/hadoop`.

```
- name: "Add SSH key for {{hdp}} User to authorized_keys file"
  authorized_key:
    user: "{{hdp}}"
    key: "{{ lookup('file', './{{hdp_keyname}}.pub') }}"
```

The public and private key must be placed in hadoop's home directory under `.ssh`. Since a variable has been defined for the key, it is easy to provide the public (`.pub` extension) as well as the private key (no extension) without having to spell out its real name in this task. Additionally, the key is secured by setting a proper mode and ownership so that no one else but hadoop has access to it.

```
- name: "Copy public and private key to {{hdp}}'s .ssh directory"
  copy:
    src: "./{{item.name}}"
    dest: "{{hdp_localhome}}/{{hdp}}/.ssh/{{item.type}}"
    owner: "{{hdp}}"
    group: "{{hdp}}"
    mode: 0600
  with_items:
    - { type: "id_{{hdp_keytype}}", name: "{{hdp_keyname}}" }
    - { type: "id_{{hdp_keytype}}.pub", name: "{{hdp_keyname}}.pub" }
```

The hadoop user is added to the `AllowUsers` line in `/etc/ssh/sshd_config` to allow it access to each machine. The regular expression will make sure that any previous entries in the `AllowUsers` line are preserved and that the hadoop user is added to the end of the preexisting user list.

```
- name: "Add {{hdp}} to AllowedUsers line in /etc/ssh/sshd_config"
  replace:
    backup: no
    dest: /etc/ssh/sshd_config
    regexp: '^(AllowUsers(?!.*\b{{ hdp }}\b).*)$'
    replace: '\1 {{ hdp }}'
    validate: 'sshd -T -f %s'
```

SSH is restarted explicitly afterwards, as the playbook is going to make use of the hadoop SSH login soon. Note that an Ansible handler can't be used here, because it would be executed too late (at the end of the playbook when all tasks have been executed).

```
- name: Restart SSH to make changes to AllowUsers take effect
  service:
    name: sshd
    state: restarted
```

The next task deals with collecting SSH key information from the node so that hadoop does not have to confirm the host key of the target system upon establishing the first connection. We need to be able to locally ssh into the master node itself, so we have to add `0.0.0.0, localhost`, the IP address of each machine, the master IP address (so that the client nodes know about it and don't require an additional task) to `.ssh/known_hosts`. That is what `ssh-keyscan` is doing in this task step. The variable `{{workers}}` will be provided on the commandline later and contains all the machines that will act as datanodes to run map-reduce jobs. (Of course, these can also be placed in `vars.yml` when the number of machines is static and do not change.)

```
- name: "Scan SSH Keys"
  shell: ssh-keyscan 0.0.0.0 localhost \
"{{hostvars[inventory_hostname]'ansible_default_ipv4']['address']}} {{master}}" >>
"{{hdp_home}}/.ssh/known_hosts"

- name: "Scan worker SSH Keys one by one"
  shell: "ssh-keyscan {{item}} {{master}} >> {{hdp_home}}/.ssh/known_hosts"
  with_items: "{{workers}}"
```

To function properly, hadoop requires setting a number of environment variables. These include `JAVA_HOME, HADOOP_HOME` and other variables that the hadoop user needs to make the hadoop cluster work with Java. The environment variables are stored in the `.bashrc` file that is deployed from the local Ansible control machine to the hadoop home directory on the remote systems.

The `.bashrc` file itself will be provided as a template. This powerful functionality in Ansible makes it possible to store configuration files filled with Ansible variables (utilizing Jinja2 syntax). When deploying them, it is not just a simple copy operation. During transport to the remote machine, the variables are replaced with their actual values. In this case, `{{hdp_destdir}}` is replaced by `/usr/local/hadoop2.9.0`.

```
- name: "Copy BashRC over to {{hdp_localhome}}/{{hdp}}/.bashrc"
  template:
    src: "./hadoop.bashrc.template"
    dest: "{{hdp_home}}/.bashrc"
    owner: "{{hdp}}"
    group: "{{hdp}}"
```

The template file itself needs to have the following additional content at the end of the file:

```
export JAVA_HOME={{java_home}}
export HADOOP_HOME={{hdp_destdir}}
export HADOOP_INSTALL={{hdp_destdir}}
export PATH=$PATH:$HADOOP_INSTALL/bin
export HADOOP_PREFIX=/opt/{{hdp}}
export PATH=$PATH:$HADOOP_PREFIX/bin
export PATH=$PATH:$HADOOP_HOME/bin
export PATH=$PATH:$JAVA_HOME/bin
export PATH=$PATH:$HADOOP_HOME/sbin
export HADOOP_CLASSPATH=$JAVA_HOME/lib/tools.jar
export HADOOP_MAPRED_HOME=$HADOOP_HOME
```

```
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export YARN_HOME=$HADOOP_HOME
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
export HADOOP_OPTS="$HADOOP_OPTS -Djava.library.path=$HADOOP_HOME/lib/native"
```

Any users other than hadoop that want to run map-reduce jobs would also need these environment variables set, so consider also copying that file to `/etc/skel/.bashrc`.

Deploying Hadoop Configuration Files

It is time to deploy the files that make up the hadoop distribution. It is basically a tarball that can be extracted to any directory, as it mostly contains JARs and config files. This way, hadoop can easily be copied around as a whole, since the directory contains everything needed to run hadoop. The files are available for download from the hadoop webpage (http://hadoop.apache.org/releases.html). There are a lot of supported versions that keep evolving rapidly, meaning that there will be new releases coming out at regular intervals. The bottom of that page lists how many bugs were fixed in each of the releases. Contrary to how it might sound, there is no need to keep up with the pace that the hadoop project sets and an older release of hadoop can run for years if desired. A fairly recent release (2.9.0) was chosen for this article. Make sure to pick the binary distribution to download, as it takes additional time to build hadoop from sources. The file is called `hadoop-2.9.0.tar.gz`, and the name will be constructed again in the playbook by using the definitions in the `vars.yml` file. Ansible's unarchive module takes care of extracting the tarball on the remote machine into `{{hdp_destdir}}`, which resolves to `/usr/local/hadoop2.9.0`. With the version included in the directory/dataset name, it is possible to install different versions of hadoop side by side for testing purposes.

```
- name: "Unpack Hadoop {{hdp_ver}}"
  unarchive:
    src: "./{{hdp}}-{{hdp_ver}}.tar.gz"
    dest: "{{hdp_destdir}}"
    remote_src: yes
    owner: "{{hdp}}"
    group: "{{hdp}}"
```

Core Hadoop Configuration

The time has come to edit the fleet of configuration files that ship with hadoop. It can be overwhelming for beginners starting out with hadoop to understand which file needs to be changed. Unfortunately, the hadoop website does not do a good job of explaining what files need to be changed for a fully distributed hadoop cluster. In our experience, the documentation on the hadoop website is incomplete, and, even if followed to the letter, the result is not a functioning hadoop cluster. After a lot of trial and error, the author identified the important files needed to create a fully functional map-reduce cluster with the underlying HDFS on FreeBSD.

At its core, 4 files form the site-specific configuration parts for this cluster and they are named *-site.xml. They need to be changed and all of them reside in the configuration directory of the hadoop distribution. In this tutorial, that path is `/usr/local/hadoop2.9.0/etc/hadoop/` and contains `core-site.xml`, `yarn-site.xml`, `hdfs-site.xml`, and `mapred-site.xml`.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>
 <property>
  <name>fs.defaultFS</name>
  <value>hdfs://{{master}}:9000</value>
 </property>
 <property>
```

```
  <name>io.file.buffer.size</name>
  <value>131072</value>
 </property>
 <property>
  <name>hadoop.tmp.dir</name>
  <value>{{hdp_tmp_dir}}</value>
  <description>A base for other temporary directories.</description>
 </property>
</configuration>
```

core-site.xml Template

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
<!-- Site specific YARN configuration properties -->
  <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>
  <property>
    <name>yarn.resourcemanager.hostname</name>
    <value>{{master}}</value>
  </property>
  <property>
    <name>yarn.resourcemanager.resource-tracker.address</name>
    <value>{{master}}:8025</value>
  </property>
  <property>
    <name>yarn.resourcemanager.scheduler.address</name>
    <value>{{master}}:8030</value>
  </property>
<property>
    <name>yarn.resourcemanager.address</name>
    <value>{{master}}:8050</value>
  </property>
  <property>
    <name>yarn.resourcemanager.webapp.address</name>
    <value>0.0.0.0:8088</value>
  </property>
</configuration>
```

yarn-site.xml Template

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
 <property>
  <name>dfs.namenode.name.dir</name>
  <value>file://{{hdp_name_dir}}</value>
 </property>
```

```
 <property>
  <name>dfs.datanode.data.dir</name>
  <value>file://{{hdp_data_dir}}</value>
 </property>
 <property>
  <name>dfs.replication</name>
  <value>2</value>
 </property>
 <property>
  <name>dfs.checkpoint.dir</name>
  <value>file://{{hdp_freebsd_namesec_dir}}</value>
  <final>true</final>
 </property>
</configuration>
```

hdfs-site.xml Template

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>mapreduce.job.tracker</name>
    <value>{{inventory_hostname}}:8021</value>
  </property>
  <property>
    <name>mapred.job.tracker</name>
    <value>{{inventory_hostname}}:54311</value>
  </property>
  <property>
    <name>mapred.local.dir</name>
    <value>{{hdp_mapred_dir}}</value>
  </property>
  <property>
    <name>mapred.system.dir</name>
    <value>/mapredsystemdir</value>
  </property>
  <property>
    <name>mapred.tasktracker.map.tasks.maximum</name>
    <value>2</value>
  </property>
  <property>
    <name>mapred.tasktracker.reduce.tasks.maximum</name>
    <value>2</value>
  </property>
  <property>
    <name>mapred.child.java.opts</name>
    <value>-Xmx200m</value>
  </property>
  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>
  <property>
    <name>mapreduce.jobhistory.address</name>
```

```
    <value>{{inventory_hostname}}:10020</value>
  </property>
</configuration>
```

mapred-site.xml Template

The following task in the playbook will take care of putting them in the right place with properly replaced variables from the `vars.yml` definition file. (Particularly at this point, having a central Ansible variables file becomes invaluable, as typos and errors in these files cause a lot of headaches debugging an already complex distributed system like hadoop.)

```
- name: "Templating *-site.xml files for the node"
  template:
    src: "./Hadoop275/freebsd/{{item}}.j2"
    dest: "{{hdp_destdir}}/etc/{{hdp}}/{{item}}"
    owner: "{{hdp}}"
    group: "{{hdp}}"
  with_items:
    - core-site.xml
    - hdfs-site.xml
    - yarn-site.xml
    - mapred-site.xml
```

A file called `slaves` (newer versions renamed it to `workers`) contains the names of hosts that should serve as datanodes. The machine defined as the master can also participate and work on map-reduce jobs, hence the localhost in the file. The task here adds the workers that are defined as parameters to the Ansible playbook to that file:

```
- name: "Create and populate the slaves file"
  lineinfile:
    dest: "{{hdp_destdir}}/etc/{{hdp}}/slaves"
    owner: "{{hdp}}"
    group: "{{hdp}}"
    line: "{{item}}"
  with_items: "{{ workers }}"
```

Now that a bunch of file changes have been made to the installation, we need to be sure that the files are still owned by hadoop and not the user running the Ansible script. This last task recursively sets ownership and group to the hadoop user on the files and directories the playbook that has touched so far.

```
- name: "Give ownership to {{hdp}}"
  file:
    path: "{{item}}"
    owner: "{{hdp}}"
    group: "{{hdp}}"
    recurse: yes
  with_items:
    - "{{hdp_home}}"
    - "{{hdp_destdir}}"
    - "/{{hdp}}"
```

That's the complete playbook called `freebsd_hadoop2.9.0.yml` and it can be executed with the following commandline.

```
$ ./freebsd_hadoop2.9.0.yml -Kbe 'host=namenode:datanode1:datanode2
master=namenode' -e '{"workers":{"datanode1","datanode2"}}' --vault-id @prompt
```

The hosts `namenode`, `datanode1`, and `datanode2` all need to be defined in Ansible's inventory file. The `--vault-id @prompt` parameter will ask for the vault password that was defined when creating the vault.

Starting Hadoop and the First Map-reduce Job

After the playbook has been run and there are no errors in the deployment, it is time to log into the namenode host and switch to the hadoop user (using the password that was set). A first test is to verify that this user can log into each `datanode1` and `datanode2` without being prompted to confirm the hostkey or provide a password. If the login completes without any of these, then the hadoop services can be started. The first step is to format the distributed filesystem using the `hdfs namenode` command (the path to hadoop is in the `.bashrc` file, so the full path to the hdfs executable is omitted):

```
hadoop@namenode$ hdfs namenode -format
```

A couple of initialization messages scroll by, but there should be no errors at the end. Be careful when running this command a second time. Each time, a unique ID is generated to identify the HDFS from others. Unfortunately, the format is only done on the master node, not throughout the other cluster nodes. Hence, running it a second time will confuse the datanodes because they still retain the old ID. The solution is to wipe the directories defined in `{{hdp_data_dir}}` and `{{hdp_tmp}}` of any previous content, both on the datanodes and the namenode.

Next, all the services that make up the hadoop system must be started in order. The following commands will take care of that:

```
hadoop@namenode$ start-dfs.sh && start-yarn.sh && mr-jobhistory-daemon.sh start
historyserver
```

To make sure all the processes have started successfully, run `jps` to verify that the following services have started on the namenode: `NameNode`, `ResourceManager`, `JobHistoryServer`, and `SecondaryNameNode`.

The datanodes must have these processes in the `jps` output: `NodeManager` and `DataNode`. (Running `jps` during a map-reduce job means more processes will be spawned on the datanodes that form the units of work the node is processing using the YARN framework.)

The cluster is ready to run its first map-reduce job. Hadoop provides sample jobs to get to know the framework without having to write a Java program first and packing it in a jar file to be executed as a job. One of these example files will try to calculate the value of pi using a Monte Carlo simulation. The following shell script can do that:

```
#!/bin/sh
export HADOOP_CLASSPATH=${JAVA_HOME}/lib/tools.jar
hadoop jar /usr/local/hadoop2.9.0/share/hadoop/mapreduce/hadoop-mapreduce-examples-
2.9.0.jar pi 16 1000000
```

Executing the shell script will spawn mappers to calculate a subset of the Monte Carlo simulation. Depending on how many mappers are chosen (16 in this example), the accuracy of the result varies. The job can be monitored using a browser that's pointed to the URL `http://<the.namenode.ip.address>:8088`. Browsing to `http://<the.namenode.ip.address>:50070` displays the overall cluster status along with a filesystem browser for the HDFS and logs (manual refresh is required to get updated information on both pages).

Another interesting sample is the `random-text-writer` that creates a bunch of files in the HDFS across the nodes. A `timestamp` is used to make it possible to run this command multiple times in a row:

```
#!/bin/sh
export HADOOP_CLASSPATH=${JAVA_HOME}/lib/tools.jar
timestamp="`date +%Y%m%d%H%M`"
hadoop jar /usr/local/hadoop-2.9.0/share/hadoop/mapreduce/hadoop-mapreduce-examples-
2.9.0.jar randomtextwriter /random-text_$timestamp
```

After both jobs have run without errors, the hadoop cluster is ready to accept more sophisticated map-reduce jobs. This is left as a learning exercise for the reader, the internet is full of tutorials about how to write map-reduce jobs.

This tutorial closes with a view of the ZFS compression ratios achieved with the two jobs being completed (results may vary):

```
hadoop@namenode$ zfs get refcompressratio hadooppool/hadoop/hdfs/namenode
NAME                                PROPERTY             VALUE         SOURCE
hadooppool/hadoop/hdfs/namenode     refcompressratio     26.28x        -
```

The datanodes also achieved quite a good compression ratio from the random-text-writer example:

```
NAME                                PROPERTY             VALUE         SOURCE
hadooppool/hadoop/hdfs/datanode     refcompressratio     2.35x         -
```

This shows that running hadoop on FreeBSD has benefits. OpenZFS is able to add additional protection to the data stored in HDFS and makes it possible to store more data on the underlying disks. In a big data world, this is an enormous win.

BENEDICT REUSCHLING joined the FreeBSD Project in 2009. After receiving his full documentation commit bit in 2010, he actively began mentoring other people to become FreeBSD committers. He is a proctor for the BSD Certification Group and joined the FreeBSD Foundation in 2015, where he is currently serving as vice president. Benedict has a Master of Science degree in Computer Science and is teaching a UNIX for software developers class at the University of Applied Sciences, Darmstadt, Germany.