Tessera Stack Cluster Installation

Ashrith Barthur
Doug Crabill
Xiaosu Tong

1 Tessera Installation on Ubuntu

This is a guide to installing the Tessera stack consisting of Hadoop, RHIPE, datadr, trelliscope and other supporting packages on a multi-node cluster running Ubuntu Linux version 14.04. A Red Hat Enterprise Linux version of this installation guide is available at `http://tessera.io/docs-install-cluster/`

2 The R, RHIPE, Hadoop Setting

2.1 Overview

The setting has three components: remote computer, one or more Unix R-session servers, and a Unix Hadoop cluster. The second two components are running R and RHIPE. You work on the remote computer, say your laptop, and login to an R-session server. This is home base, where you do all of your programming of R and RHIPE R commands. The R commands you write for division, application of analytic methods, and recombination that are destined for Hadoop on the cluster are passed along by RHIPE R commands.

The remote computer is typically for you to maintain. The R-session servers require IT staff to help install software, configure, and maintain. However you install packages too on the R-session servers, just you do when you want to use an R CRAN package in R. There is an extra task though; you want packages you install to be pushed up the Hadoop cluster so they can be used there too. Except for this push by you, the Hadoop cluster is the domain of the systems administrators who must, among other tasks, install Hadoop.

2.2 The R-Session Server and RStudio

Now the R-session server can be separate from the Hadoop cluster, handling only R sessions, or it can be one of the servers on the Hadoop cluster. If it is on the Hadoop cluster, there must be some precautions taken in the Hadoop configuration to protect the programming of the R session. This is needed because the RHIPE Hadoop jobs compete with the R sessions. There are never full guarantees though, so "safe mode" is separate R session servers. The last thing you want is for R sessions to get bogged down. If the cluster option is chosen, then you want to mount a file server on the cluster that contains the files associated with the R session such as .RData and files read into to R or written by R.

There is a vast segment of the R community that uses RStudio, for good reason. RStudio can join the setting. You have RStudio server installed on the R-session servers by system administrators. A web browser on the R server runs the RStudio interface which is accessed by you on your remote device via the remote login.

2.3 The Remote Computer

The remote computer is just a communication device, and does not carry out data analysis, so it can run any operating system, such as Windows. This is especially important for teaching, since Windows labs are typically very plentiful at academic institutions, but Unix labs much less so. Whatever the operating system, a common communication protocol that is used is the SSH protocol. SSH is typically used to log into a remote machine and execute commands or to transfer files. But a critical capability of it for our purposes here is that it supports both your R session command-line window, showing both input and output, and a separate window to show graphics.

2.4 Where Are the Data Analyzed

Obviously, much data analysis is carried out by Hadoop on the Hadoop cluster. Your R commands are given to RHIPE, passed along to Hadoop, and the outputs are written by Hadoop to the HDFS.

But in many analyses of larger and more complex data, it is common to have (1) outputs of a recombination method that constitute a relatively small dataset, and (2) the outputs are further analyzed as part of the overall analysis. If they are small enough to be readily analyzed in your R session, then for sure that is where you want to be. RHIPE commands allow you to write the recombination outputs from the HDFS to the R global environment of your R session. They become a dataset in .RData. While programming R and RHIPE is easy, it is not as easy as plain old serial R. The point is that a lot of data analysis can be carried out in just R even when the data are large and complex.

2.5 A Few Basic Hadoop Features

The two principal computational operations of Hadoop are Map and Reduce. The first runs parallel computations on subsets without communication among them. The second can compute across subset outputs. So Map carries out the analytic method computation. Reduce takes the outputs from Map and runs the recombination computation. A division is typically carried out both by Map and Reduce, sometimes each used several times, and can occur as part of the reading of the data into R at the start of the analysis.

Usage of Map and Reduce involves the critical Hadoop element of key-value pairs. We give one instance here. The Map operation, instructed by the analyst R code, puts a key on each subset output. This forms a key-value pair with the output as the value. Each output can have a unique key, or each key can be given to many outputs, or all outputs can have the same key. When Reduce is given the Map outputs, it assembles the key-value pairs by key, which forms groups, and then the R recombination code is applied to the values of each group independently; so the running of the code on the different groups is embarrassingly parallel. This framework provides substantial flexibility for the recombination method.

Hadoop attempts to optimize computation in a number of ways. One example is Map. Typically, there are vastly more subsets than cores on the cluster. When Map finishes the application of the analytic method to a subset on a core, Hadoop seeks to assign a subset on the same node as the core to avoid transmission of the subset across the network connecting the nodes, which is more time consuming.

3 Tessera Stack Cluster Installation Guide

3.1 Introduction

This guide is intended to be used by the Systems Administrator to quickly install and configure the software necessary to run the full Tessera stack on a multi-node cluster. It is not focused on tuning and optimally configuring the Tessera stack. Most components of Tessera require no tuning. However, the Systems Administrator is encouraged to consult the Cloudera Hadoop documentation for further tuning of Hadoop and HDFS.

3.2 Example Configuration

Running the Tessera stack on a Hadoop cluster requires each Hadoop node to be configured to fill one or more of the roles of ResourceManager, NameNode, Secondary NameNode, DataNode, and NodeManager. The role descriptions are as follows.

- NameNode - Manages the directory tree and metadata of the Hadoop File System (HDFS), a distributed virtual file system that allows data to be replicated and stored across many nodes in the cluster.
- SecondaryNameNode - Offloads HDFS checkpoint support for the NameNode. It is not a NameNode failover or backup as the name may imply.
- ResourceManager - Schedules and issues map reduce jobs for NodeManagers across the cluster.
- DataNode - Stores data on the local drives of nodes as part of the distributed HDFS. DataNodes are almost always also NodeManagers.
- NodeManager - Executes the map and reduce jobs issued by the ResourceManager. NodeManagers are almost always also DataNodes.

There can be just one NameNode, one ResourceManager, and one Secondary NameNode, but there will be many DataNodes and NodeManagers. The node acting as NameNode cannot also be the Secondary NameNode, but could perform one or more of other roles of ResourceManager, DataNode, and NodeManager with a potential performance penalty. Most cluster nodes will be compute nodes with multiple hard drives and thus will run both the DataNode and NodeManager services.

For our example we will have a single front-end server that acts as the R-session server. It is where we will run R, which in turn runs RHIPE. The front-end server is aware of our Hadoop configuration and can issue Hadoop commands, though it isn't technically a Hadoop node itself. The front-end server will also be our Shiny server and optional Rstudio server. We will have five nodes in our example Hadoop cluster. Each of the Hadoop nodes in the cluster has a fully qualified domain name of the form `nodeNNN.example.com`. In our example,

- `node001.example.com` will be the NameNode and the ResourceManager.
- `node002.example.com` will be the Secondary NameNode, a NodeManager, and a DataNode.
- `node003.example.com` to `node005.example.com` will be NodeManagers and DataNodes.
- `frontend.example.com` will be the front end R-session server, the Shiny server, and optional Rstudio Server.

3.3 Packages

3.3.1 Required Packages

The Tessera stack consists of some packages provided by the Tessera group and some provided by other sources.

Packages provided by the Tessera group

1. RHIPE - 0.75
2. datadr
3. trelliscope

Packages provided by others

1. pkg-config
2. Sun/Oracle Java - 7
3. Protocol Buffers - 2.5.0
4. Cloudera Hadoop version cdh5.3.2
5. R - 3.0.1
6. rJava - 0.9-6
7. OpenSSL (latest stable version)
8. R packages - codetools, MASS, ggplot2, lattice, boot, shiny, devtools
9. gdebi
10. Shiny server

3.3.2 Optional Packages

These optional packages can be installed along with the Tessera stack providing convenience for the analyst.

1. Rstudio server - Rstudio server creates a persistent interactive R session for the analyst by running R on a remote server and providing access through a the analyst's local browser.

3.4 Installation

Some packages need only be installed on the Hadoop nodes, some need only be installed on the frontend server, and some must be installed on both. This is specified on a per-package basis below. This guide is command line friendly.

3.4.1 System Update (all servers)

First update all currently installed system packages.

```
sudo apt-get update
sudo apt-get upgrade
```

3.4.2 pkg-config (all servers)

(`http://www.freedesktop.org/wiki/Software/pkg-config/`)
Following the system update, pkg-config is installed.

```
sudo apt-get install pkg-config
```

3.4.3 Java (all servers)

(`http://www.java.com/en/`)
A Java repository for Sun/Oracle Java is first added to the Ubuntu APT package management system.

```
sudo apt-get install python-software-properties
sudo add-apt-repository ppa:webupd8team/java
```

The system repository store is updated and Java is installed.

```
sudo apt-get update
sudo apt-get install oracle-java7-installer
```

3.4.4 Protocol Buffers (all servers)

(`https://code.google.com/p/protobuf/`)
Install Protocol Buffers version 2.5.0. It is important that version 2.5.0 be installed and not a newer or older version. Ubuntu 14.04 should have version 2.5.0 available by default. Confirm this first by installing the `aptitude` package manager and ensuring it is version 2.5.0:

```
sudo apt-get install aptitude
aptitude show protobuf-compiler
```

If it says it is version 2.5.0-something, then proceed to install it via:

```
sudo apt-get install protobuf-compiler protobuf-c-compiler libprotobuf-dev
```

If it does NOT say it is version 2.5.0-something, then manually install it. First cd to a directory the software can be downloaded and built, and type the following:

```
# Only install manually if the steps above did not say version 2.5.0-something
sudo apt-get install libtool
wget https://github.com/google/protobuf/archive/v2.5.0.zip
unzip v2.5.0.zip
cd protobuf-2.5.0
./autogen.sh
./configure --prefix=/usr/local
make
sudo make install
```

3.4.5 Hadoop (all servers)

(`http://www.cloudera.com/content/cloudera/en/products-and-services/cdh.html`) We use the Hadoop distribution built by Cloudera. It is a Hadoop MRv2 / YARN implementation. Download the Cloudera repository update package and install it. This will add the Cloudera package repository and repository key to the Ubuntu repositories searched when installing packages:

```
wget http://archive.cloudera.com/cdh5/one-click-install/trusty/\
amd64/cdh5-repository_1.0_all.deb
sudo dpkg -i cdh5-repository_1.0_all.deb
sudo apt-get update
```

We install different packages on each node based on the Hadoop roles taken by that node. Note that when these installs complete you will see warnings such as "Failed to start Hadoop namenode. Return value: 1". This is because we have not yet completed the configuration of these packages so the initial attempt to launch them will fail.

node001.example.com:

```
sudo apt-get install hadoop-yarn-resourcemanager hadoop-hdfs-namenode \
hadoop-client
```

node002.example.com:

```
sudo apt-get install hadoop-yarn-nodemanager hadoop-hdfs-secondarynamenode \
hadoop-hdfs-datanode hadoop-mapreduce
```

node003.example.com through node005.example.com:

```
sudo apt-get install hadoop-yarn-nodemanager hadoop-hdfs-datanode \
hadoop-mapreduce
```

frontend.example.com (doesn't run any hadoop services, but needs the hadoop commands):

```
sudo apt-get install hadoop-client
```

As part of the above steps, three new users and groups named yarn, hdfs, and mapred will be added to the system. All three users are added to the new group hadoop.

Configuration of Hadoop settings is done in another section, but for now we will prepare the configuration directory and set it as the default Hadoop configuration on the system. On each of node001 through node005, and on the frontend execute the following. We choose the configuration directory name of "conf.tessera" in this guide:

```
sudo cp -r /etc/hadoop/conf.empty /etc/hadoop/conf.tessera
sudo update-alternatives --install /etc/hadoop/conf hadoop-conf \
/etc/hadoop/conf.tessera 50
sudo update-alternatives --set hadoop-conf /etc/hadoop/conf.tessera
```

Add Hadoop environment variables and an updated path to the user environment for future logins. This assumes users use an sh/ksh/bash/zsh based login shell.

```
echo "export HADOOP=/usr/lib/hadoop
export HADOOP_HOME=\$HADOOP
export HADOOP_BIN=\$HADOOP/bin
export HADOOP_LIB=\$HADOOP/lib
export HADOOP_LIBS=\‘hadoop classpath | tr -d \\*\‘
export HADOOP_CONF_DIR=/etc/hadoop/conf.tessera
export HADOOP_COMMON_HOME=/usr/lib/hadoop
export HADOOP_MAPRED_HOME=/usr/lib/hadoop-mapreduce
export HADOOP_HDFS_HOME=/usr/lib/hadoop-hdfs
export YARN_HOME=/usr/lib/hadoop-yarn
export PATH=\$PATH:\$HADOOP_BIN" |
sudo bash -c "cat > /etc/profile.d/hadoop.sh"
```

3.4.6 R (all nodes)

(http://cran.r-project.org/)
R is installed and configured to know about Java.

```
sudo apt-get install r-base r-base-dev r-recommended r-cran-rodbc
sudo R CMD javareconf
```

3.4.7 rJava (all nodes)

(http://www.rforge.net/rJava/)
rJava is also required by RHIPE. It is installed by:

```
sudo apt-get install r-cran-rjava
```

3.4.8 RHIPE (all nodes)

(http://www.tessera.io/)
RHIPE is downloaded and installed.

```
wget http://ml.stat.purdue.edu/rhipebin/Rhipe_0.75.0_cdh5mr2.tar.gz
sudo R CMD INSTALL Rhipe_0.75.0_cdh5mr2.tar.gz
```

3.4.9 OpenSSL (all nodes)

(https://www.openssl.org/)
OpenSSL package is required by the Github installer/package handler for R.

```
sudo apt-get install libcurl4-openssl-dev
```

3.4.10 datadr and trelliscope support packages (all nodes)

(http://cran.r-project.org/web/packages/available_packages_by_name.html)
R packages - codetools, latttice, MASS, ggplot2, boot, shiny and devtools are needed for trelliscope and datadr. They are installed using the R internal package installer **install.packages**. Use the command line to enter the following commands.

```
sudo apt-get install r-cran-codetools r-cran-mass r-cran-ggplot2 \
r-cran-lattice r-cran-boot
sudo R -e "install.packages('RCurl', repos='http://cran.rstudio.com/')"
sudo R -e "install.packages('shiny', repos='http://cran.rstudio.com/')"
sudo R -e "install.packages('devtools', repos='http://cran.rstudio.com/')"
```

3.4.11 datadr and trelliscope (all nodes)

(http://www.tessera.io)
datadr and trelliscope are installed using install_github package from R.

```
sudo R -e "options(repos = 'http://cran.rstudio.com/');\
 library(devtools); install_github('tesseradata/datadr')"
sudo R -e "options(repos = 'http://cran.rstudio.com/');\
 library(devtools); install_github('tesseradata/trelliscope')"
```

3.4.12 gdebi-core (frontend server)

(https://apps.ubuntu.com/cat/applications/precise/gdebi/)
gdebi is a deb package installer. It automatically resolves and installs library dependencies.

```
sudo apt-get install gdebi-core
```

3.4.13 Shiny server (frontend server)

(http://www.rstudio.com/products/shiny/shiny-server/)

Shiny server is only installed on frontend.example.com, our designated Shiny server.

```
sudo R -e "install.packages('shiny', repos='http://cran.rstudio.com/')"
wget http://download3.rstudio.org/ubuntu-12.04/x86_64/shiny-server-1.3.0.403-amd64.deb
sudo gdebi --n shiny-server-1.3.0.403-amd64.deb
```

3.4.14 Rstudio Server (frontend server)

(http://www.rstudio.com/)
Rstudio Server is installed on just the shiny server, frontend.example.com.

```
wget http://download2.rstudio.org/rstudio-server-0.98.1103-amd64.deb \
-O /tmp/rstudio-server.deb
sudo gdebi --n /tmp/rstudio-server.deb
rm /tmp/rstudio-server.deb
```

Rstudio Server should now be running on frontend.example.com. Connect to it by browsing to http://frontend.example.com:8787 and logging in as a normal user with accounts on that server. This may require firewall changes.

3.5 Configuration

Hadoop's HDFS will use space on the local disks on node002 through node005 to create a single large distributed filesystem. In our example we will say each of node002 through node005 has four 2TB drives per node, for a total of 32TB across the four servers. These local drives must be already been formatted by Linux as normal Linux filesystems (ext4, ext3, etc.). We will further say these filesystems have been mounted with the same mount point locations on all nodes: /hadoop/disk1, /hadoop/disk2, /hadoop/disk3, and /hadoop/disk4. HDFS actually permits different sized drives and different numbers of drives on each server, but doing so requires each server to have an individually maintained hdfs-site.xml file, described later. If possible, keep the number of drives the same on each node for easier administration. The names of the mount points aren't special and could be anything, but these are the names we will use in this example. HDFS creates its own directory structure and files within the drives that are part of HDFS. If one were to look inside one of these filesystems after Hadoop is running and files have been written to HDFS, one would see a file and directory structure with computer generated directory names and filenames that don't correspond in any way to their HDFS filenames (eg. blk_1073742354, blk_1073742354_1530.meta, etc.). Access to these files must be made through Hadoop. Paths to files in HDFS begin with '/' just as they do on a normal Linux filesystem, but the namespace is completely independent from all other files on the nodes. This means the directory /tmp on HDFS is completely different from the normal /tmp visible at a bash prompt. Some paths in the configuration below are in the Linux filesystem namespace an some are in the HDFS filesystem namespace.

The frontend server does not directly participate in Hadoop computation, HDFS storage, etc. but it must have knowledge of the Hadoop configuration in order to interact with Hadoop. Thus all Hadoop configuration files must exist on the frontend in addition to the nodeNNN servers.

The amount of RAM per node must be known for some configuration settings, so lets say we have 64GB per node in our example configuration. We will say each node has 16 CPU cores.

3.5.1 Hosts File or DNS entries

The hostnames and IP addresses of all machines in the Hadoop cluster must be added to the `/etc/hosts` file on each machine of the cluster, or be know to the DNS server each of the nodes uses. These are our example hostnames and IP addresses.

```
10.0.0.001 node001.example.com node001
10.0.0.002 node002.example.com node002
10.0.0.003 node003.example.com node003
10.0.0.004 node004.example.com node004
10.0.0.005 node005.example.com node005
10.0.0.006 frontend.example.com frontend
```

3.5.2 Firewalls

The new network services for Hadoop, including access to files in HDFS, will be visible to the Internet unless already blocked due to NAT or some other form of firewall elsewhere on the network. It is advised that a host-based firewall be erected using iptables on each of the nodes and the fronend server. It would permit full communication for all ports among the six servers in our example: node001 through node005 and the frontend server. It would restrict access to these servers from external hosts based on your local needs. For example, you may wish to permit ssh connections to the frontend from anywhere in the world, but allow ssh to the nodeNNN servers only from local subnets. You may want to permit Rstudio Server connections to the frontend (port 8787) from just three local subnets only, but permit no other network access to any other servers. Other ports you may wish to open to local subnets include port 50070 on node001 to see the status of HDFS, and port 8088 on node001 to see the status of the ResourceManager, the jobs that are running, have been completed, etc.

3.5.3 Hadoop core-site.xml

The `core-site.xml` file is placed in `/etc/hadoop/conf.tessera` on every server in the cluster (nodeNNN and frontend). If a change is ever made to this config file, it must be made to that file on every node in the cluster as well.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <!-- This is the NameNode -->
    <name>fs.defaultFS</name>
    <value>hdfs://node001.example.com:8020</value>
  </property>
</configuration>
```

3.5.4 Hadoop mapred-site.xml

The `mapred-site.xml` file in in `/etc/hadoop/conf.tessera` is configured to tell Hadoop we are using the MRv2 / YARN framework. This file should be copied to all servers (nodeNNN and frontend).

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>
```

```
</configuration>
```

3.5.5 Hadoop yarn-site.xml

The `yarn-site.xml` file in in `/etc/hadoop/conf.tessera` is used to configure Hadoop settings related to temporary storage, number of CPU cores to use per node, memory per node, etc. This file should be copied to all servers (nodeNNN and frontend). Changes to this file can have significant performance impact, such as running 16 tasks per node rather than running 2 per node. Tuning these settings optimally is beyond the scope of this guide. In our example cluster, the hardware configuration of all nodes is identical, thus the same yarn-site.xml file can be copied around to all nodes. If in your configuration you have some nodes with different amounts of RAM or different numbers of CPU cores, they must have an individually maintained and updated `yarn-site.xml` file with those settings configured differently.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <!-- This should match the name of the resource manager in your local deployment -->
    <name>yarn.resourcemanager.hostname</name>
    <value>node001.example.com</value>
  </property>

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

  <property>
    <!-- How much RAM on this server can be used for Hadoop -->
    <!-- We will use (total RAM - 2GB).  We have 64GB in our example, so use 62GB -->
    <name>yarn.nodemanager.resource.memory-mb</name>
    <value>62000</value>
  </property>

  <property>
    <!-- How many CPU cores on this server can be used for Hadoop -->
    <!-- We will use them all, which is 16 per node in our example cluster -->
    <name>yarn.nodemanager.resource.cpu-vcores</name>
    <value>16</value>
  </property>

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

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

<property>
  <name>yarn.resourcemanager.scheduler.class</name>
  <value>org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair.FairScheduler</va
</property>

<property>
  <name>yarn.log-aggregation-enable</name>
  <value>true</value>
</property>

<property>
  <!-- List of directories to store temporary localized files. -->
  <!-- Spread these across all local drives on all nodes -->
  <name>yarn.nodemanager.local-dirs</name>
  <value>file:///hadoop/disk1/yarn/local,file:///hadoop/disk2/yarn/local,
         file:///hadoop/disk3/yarn/local,file:///hadoop/disk4/yarn/local</value>
</property>

<property>
  <!-- Where to store temporary container logs. -->
  <!-- Spread these across all local drives on all nodes -->
  <name>yarn.nodemanager.log-dirs</name>
  <value>file:///hadoop/disk1/yarn/log,file:///hadoop/disk2/yarn/log,
         file:///hadoop/disk3/yarn/log,file:///hadoop/disk4/yarn/log</value>
</property>

<property>
  <!-- This should match the name of the NameNode in your local deployment -->
  <name>yarn.nodemanager.remote-app-log-dir</name>
  <value>hdfs://node001.example.com/var/log/hadoop-yarn/apps</value>
</property>

<property>
  <name>yarn.application.classpath</name>
  <value>
     $HADOOP_CONF_DIR,
     $HADOOP_COMMON_HOME/*,$HADOOP_COMMON_HOME/lib/*,
     $HADOOP_HDFS_HOME/*,$HADOOP_HDFS_HOME/lib/*,
     $HADOOP_MAPRED_HOME/*,$HADOOP_MAPRED_HOME/lib/*,
     $HADOOP_YARN_HOME/*,$HADOOP_YARN_HOME/lib/*
  </value>
</property>
```

```
</configuration>
```

We reference new directories above in the local Linux filesystem into which logs and other files are placed. We must create these directories now and set the proper owner, group, and permissions. This must be done on node001 - node005:

```
sudo mkdir -p /hadoop/disk1/yarn/local /hadoop/disk2/yarn/local
sudo mkdir -p /hadoop/disk3/yarn/local /hadoop/disk4/yarn/local
sudo mkdir -p /hadoop/disk1/yarn/log /hadoop/disk2/yarn/log
sudo mkdir -p /hadoop/disk3/yarn/log /hadoop/disk4/yarn/log
sudo chown -R yarn.yarn /hadoop/disk1/yarn/local /hadoop/disk2/yarn/local
sudo chown -R yarn.yarn /hadoop/disk3/yarn/local /hadoop/disk4/yarn/local
sudo chown -R yarn.yarn /hadoop/disk1/yarn/log /hadoop/disk2/yarn/log
sudo chown -R yarn.yarn /hadoop/disk3/yarn/log /hadoop/disk4/yarn/log
```

3.5.6 Hadoop secondary namenode file masters

The file `/etc/hadoop/conf.tessera/masters` must contain the hostname of the secondary namenode. This file should be copied to all servers. For our example, it contains just one line:

```
node002.example.com
```

3.5.7 Hadoop hdfs-site.xml

The `hdfs-site.xml` file is used to configure the Hadoop file system (HDFS). This file should be copied to the `/etc/hadoop/conf.tessera` directory on all nodes (nodeNNN and frontend). As with `yarn-site.xml`, if your number of disks and size of disk varies among nodes, you must independently maintain copies of `hdfs-site.xml` on each node.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <!--  Number of times each HDFS block is replicated.  Default is 3. -->
    <name>dfs.replication</name>
    <value>3</value>
  </property>

  <property>
    <!-- Size in bytes of each HDFS block.  Should be a power of 2. -->
    <!--  We use 2^27 -->
    <name>dfs.blocksize</name>
    <value>134217728</value>
```

```
</property>

<!-- Where the namenode stores HDFS metadata on its local drives -->
<!-- These are Linux filesystem paths that must already exist. -->
<property>
  <name>dfs.namenode.name.dir</name>
  <value>file:///hadoop/disk1/dfs/nn,file:///hadoop/disk2/dfs/nn,
         file:///hadoop/disk3/dfs/nn,file:///hadoop/disk4/dfs/nn</value>
</property>

<!-- Where the secondary namenode stores HDFS metadata on its local drives -->
<!-- These are Linux filesystem paths that must already exist. -->
<property>
  <name>dfs.namenode.checkpoint.dir</name>
  <value>file:///hadoop/disk1/dfs/nn,file:///hadoop/disk2/dfs/nn,
         file:///hadoop/disk3/dfs/nn,file:///hadoop/disk4/dfs/nn</value>
</property>

<property>
  <!-- Where each datanode stores HDFS blocks on its local drives. -->
  <!-- These are Linux filesystem paths that must already exist. -->
  <name>dfs.datanode.data.dir</name>
  <value>file:///hadoop/disk1/dfs/dn,file:///hadoop/disk2/dfs/dn,
         file:///hadoop/disk3/dfs/dn,file:///hadoop/disk4/dfs/dn</value>
</property>

<property>
  <!-- This should match the name of the NameNode in your local deployment -->
  <name>dfs.namenode.http-address</name>
  <value>node001.example.com:50070</value>
</property>

<property>
   <name>dfs.permissions.superusergroup</name>
   <value>hadoop</value>
</property>
<property>
  <name>dfs.client.read.shortcircuit</name>
  <value>true</value>
</property>
<property>
  <name>dfs.client.read.shortcircuit.streams.cache.size</name>
  <value>1000</value>
</property>
<property>
```

```
    <name>dfs.client.read.shortcircuit.streams.cache.expiry.ms</name>
    <value>10000</value>
  </property>

  <property>
    <!-- Leave the dn._PORT as is, do not try to make this a number -->
    <name>dfs.domain.socket.path</name>
    <value>/var/run/hadoop-hdfs/dn._PORT</value>
  </property>
</configuration>
```

We must pre-create the local metadata storage directories on the NameNode (node001), and on the Secondary NameNode (node002):

```
sudo mkdir -p /hadoop/disk1/dfs/nn /hadoop/disk2/dfs/nn
sudo mkdir -p /hadoop/disk3/dfs/nn /hadoop/disk4/dfs/nn
sudo chown -R hdfs.hdfs /hadoop/disk1/dfs /hadoop/disk2/dfs
sudo chown -R hdfs.hdfs /hadoop/disk3/dfs /hadoop/disk4/dfs
sudo chmod 700 /hadoop/disk1/dfs /hadoop/disk2/dfs /hadoop/disk3/dfs /hadoop/disk4/dfs
```

We must pre-create the local storage directories on the datanodes, node002 through node005:

```
sudo mkdir -p /hadoop/disk1/dfs/dn /hadoop/disk2/dfs/dn
sudo mkdir -p /hadoop/disk3/dfs/dn /hadoop/disk4/dfs/dn
sudo chown -R hdfs.hdfs /hadoop/disk1/dfs /hadoop/disk2/dfs
sudo chown -R hdfs.hdfs /hadoop/disk3/dfs /hadoop/disk4/dfs
sudo chmod 700 /hadoop/disk1/dfs /hadoop/disk2/dfs
sudo chmod 700 /hadoop/disk3/dfs /hadoop/disk4/dfs
```

3.6 Hadoop Pre-run Setup

At this point the configuration files created in `/etc/hadoop/conf.tessera` on node001 should be copied to all other nodes. Any future changes to any configuration files should be done on node001 and then copied from there to all other nodes.

Note: In the event that some nodes have different number of drives, or the paths to those drives differ, or they have a different number of CPUs, different amounts of RAM, etc. then separate, independent `hdfs-site.xml` and `yarn-site.xml` files may be necessary on each node.

Next we format the new HDFS filesystem before actually starting the NameNode service for the first time. This is done on the NameNode (node001) only. It must be done as the user "hdfs":

```
sudo -u hdfs hdfs namenode -format
```

Manually start hdfs first by starting the NameNode service on node001:

```
sudo service hadoop-hdfs-namenode start
```

Then start the datanode service on node002 - node005 by typing this on each of them one at a time:

```
sudo service hadoop-hdfs-datanode start
```

Once the NameNode service is running on node001 and the DataNode service is running on the other nodes, it is time to create new folders in HDFS for Hadoop to use and set permissions correctly. Note these are HDFS paths, NOT normal Linux filesystem paths:

```
sudo -u hdfs hadoop fs -mkdir -p /tmp/hadoop-yarn/staging
sudo -u hdfs hadoop fs -chmod -R 1777 /tmp
sudo -u hdfs hadoop fs -mkdir -p /user/history
sudo -u hdfs hadoop fs -chmod -R 1777 /user/history
sudo -u hdfs hadoop fs -chown mapred:hadoop /user/history
sudo -u hdfs hadoop fs -mkdir -p /var/log/hadoop-yarn
sudo -u hdfs hadoop fs -chown yarn:mapred /var/log/hadoop-yarn
```

The administrator much make some choices about where users will be storing their files, what permissions should exist on the /user directory, etc. If privacy is important, the administrator must create user directories in HDFS individually for every user on the system and set the permissions on those directories accordingly. For example, to create a private storage location for users joe and bob, the adminsitrator would type:

```
sudo -u hdfs hadoop fs -mkdir /user/joe /user/bob
sudo -u hdfs hadoop fs -chown joe /user/joe
sudo -u hdfs hadoop fs -chown bob /user/bob
sudo -u hdfs hadoop fs -chmod  700 /user/joe /user/bob
```

This would permit joe and only joe to read, write, and create files in `/user/joe`. Only bob could read, write, and create files in `/user/bob`.

In a research group where all users are permitted to see files created by all other users, the easiest approach is just to set permissions such that anyone can create new directories inside the /user directory themselves. To do this, the administrator would set the permissions like this:

```
sudo -u hdfs hadoop fs -chmod 1777 /user
```

Then an individual user, like bob, could create his own directory where he can store data, rather than having the adminsitrator create directories for everyone individually. Bob could just log into the frontend server and type this himself:

```
hadoop fs -mkdir /user/bob
```

3.7 Starting the Hadoop Cluster

Configuration settings and environment varaibles have been changed such that rebooting all servers should cause all Hadoop services to automatically start and the environment to be correctly set for all new logins. Do that now by rebooting node001 through node005, and the frontend server, by typing this on each node:

```
sudo reboot
```

3.8 Checking the status of the Hadoop Cluster

You can see the status of the ResourceManager, the jobs that are running, have been completed, etc. by browsing to `http://node001.example.com:8088`. This may require firewall settings to be adjusted before this is visible to your browser. Clicking on the Nodes link in the left sidebar should show that node002 through node005 are running and reporting in.

To see the status of HDFS, which nodes are up, how much space is used and available, etc., browse to `http://node001.example.com:50070`. This may also require firewall settings to be adjusted before this is visible to your browser. Clicking the Datanodes link should show that node002 through node005 are running and reoprting in.

3.9 Notes

3.9.1 RAID and Redundancy Design under Hadoop/HDFS

- RAID configurations are usually not recommended for HDFS data drives. HDFS already handles fault tolerance by distributing the blocks it writes to local drives among all nodes for both performance and redundancy. RAID won't improve performance, and could even slow things down. In some configurations it will reduce overall HDFS capacity.
- The default block redundancy setting for HDFS is three replicates, as specified by the `dfs.replication` variable in `hdfs-site.xml`. This means that each data block is copied to three drives, optimally on three different nodes. Hadoop has shown high availability is possible with three replicates. The downside is the total capacity of HDFS is divided by the number of replicates used. This means our 32 TB example cluster with three replicates can only hold 10.67 TB of data. Decreasing `dfs.replication` below 3 is not recommended and should be avoided. Increasing it above 3 could increase performance for large clusters under certain workloads, but at the cost of capacity.

3.9.2 R Package Design

- Most R packages can be completely provided by the system administrator by installing them as root, which implicitly places them in a system wide accessible location, for example `/usr/lib/R/library` or `/usr/local/lib64/R/library`.
- Alternately, the system administrator can install just the core default R packages at a system wide location and allow individual users to install specific R library packages in their home directory. This permits users the flexibility to easily change versions of packages they are using and update them when they choose.

3.9.3 RHIPE_HADOOP_TMP_FOLDER environment variable

- It has been observed that some versions of Linux, such as Red Hat Enterprise Linux, may have an issue with RHIPE where it will give false errors about being unable to write files in HDFS, even where the directory in question is clearly writable. This can be corrected by creating a directory somewhere in HDFS that is readable only by that user, and then setting the RHIPE_HADOOP_TMP_FOLDER environment variable to point to that HDFS directory. The user bob for example, would type this on frontend:

  ```
  hadoop fs -mkdir -p /user/bob/tmp
  hadoop fs -chmod 700 /user/bob/tmp
  ```

  He would then add this to his environment by including it in his .bashrc or .bash_profile, or whatever location is appropriate for his shell:

  ```
  export RHIPE_HADOOP_TMP_FOLDER=/user/bob/tmp
  ```

3.9.4 Consider using HDFS High Availability rather than a Secondary NameNode

The primary role of the Secondary NameNode is to perform periodic checkpointing so the NameNode doesn't have to, which makes reboots of the cluster go much more quickly. The Secondary NameNode could also be used to reconstruct the majority of HDFS if the NameNode were to have a catastrophic failure, but through a manual, imperfect process prone to error. For a more robust, fault tolerant Hadoop configuration, consider using a High Availability HDFS configuration, which uses a Standby NameNode rather than a Secondary NameNode, and can be configured for automatic failover in the event of a NameNode failure. This configuration is more complex, requires the use of Zookeeper, three or more JournalNode hosts (these can be regular nodes), and another node dedicated to act as the Standby NameNode. The documentation at cloudera.com describes this in more detail.

4 Installing R Packages and Examples for Users

4.1 Background

At this point, the Tessera stack, including Hadoop (a YARN based distribution, such as Cloudera CDH5.x), R, and RHIPE should be installed. However, at some sites the administrator may choose to only maintain Hadoop, Java, Protocol Buffers, and R and rely on the end users to install the various R packages, like RHIPE and rJava to their own personal accounts.

Ideally, the versions of Linux R, and Java, are the same on both the front-end R session server and the Hadoop cluster nodes. Java in particular plays a critical roll in RHIPE, and Java likes homogeneity.

4.2 Install and Push

If the administrator has not already installed RHIPE, you first first download the package file by typing:

```
wget http://ml.stat.purdue.edu/rhipebin/Rhipe_0.75.1.4_cdh5.tar.gz
```

Then launch R and type the following to install rJava and RHIPE:

```
install.packages("rJava")
install.packages("Rhipe_0.75.1.4_cdh5.tar.gz", repos=NULL, type="source")
```

RHIPE is now installed. Each time you start an R session and you want RHIPE to be available, type:

```
library(Rhipe)
rhinit()
```

As a one-time configuration step, you push all the R packages you have installed on the R session server, including RHIPE, onto the cluster HDFS. First, you need the system administrator to configure a directory in HDFS that is writable by you. We will assume the administrator has created for you the writable directory `/user/loginname` using your login name, and has done the same thing for other users. Suppose in `/user/loginname` you want to create a directory `bin` on HDFS where you will push your installations on the R session server. You can do this and carry out the push by

```
rhmkdir("/user/loginname/bin")
hdfs.setwd("/user/loginname/bin")
bashRhipeArchive("R.Pkg")
```

`rhmkdir()` creates your directory `bin` in the directory `/user/loginname`. `hdfs.setwd()` declares `/user/loginname/bin` to be the directory with your choice of installations. `bashRhipeArchive()` creates the actual archive of your installations and names it as `R.Pkg`.

Each time your R code will require the installations on the HDFS, you must in your R session run

```
library(Rhipe) rhinit()
rhoptions(zips = "/user/loginname/bin/R.Pkg.tar.gz")
rhoptions(runner = "sh ./R.Pkg/library/Rhipe/bin/RhipeMapReduce.sh")
```

4.3 Example: Housing Data

4.3.1 The Data

The housing data consist of 7 monthly variables on housing sales from Oct 2008 to Mar 2014, which is 66 months. The measurements are for 2883 counties in 48 U.S. states, excluding Hawaii and Alaska, and also for the District of Columbia which we treat as a state with one county. The data were derived from sales of housing units from Quandl's Zillow Housing Data (www.quandl.com/c/housing). A housing unit is a house, an apartment, a mobile home, a group of rooms, or a single room that is occupied or intended to be occupied as a separate living quarter.

The variables are

- `FIPS`: FIPS county code, an unique identifier for each U.S. county
- `county`: county name
- `state`: state abbreviation
- `date`: time of sale measured in months, from 1 to 66
- `units`: number of units sold
- `listing`: monthly median listing price (dollars per square foot)
- `selling`: monthly median selling price (dollars per square foot)

Many observations of the last three variables are missing: units 68%, listing 7%, and selling 68%.

The number of measurements (including missing), is 7 x 66 x 2883 = 1,331,946. So this is in fact a small dataset that could be analyzed in the standard serial R. However, we can use them to illustrate how RHIPE R commands implement Divide and Recombine. We simply pretend the data are large and complex, break into subsets, and continuing on with D&R. The small size let's you easily pick up the data, follow along using the R commands in the tutorial, and explore RHIPE yourself with other RHIPE R commands.

"housing.txt" is available in our Tesseradata Github repository of the RHIPE documentation, or at:

`https://raw.githubusercontent.com/tesseradata/docs-RHIPE/gh-pages/housing.txt`

The file is a table with 190,278 rows (66 months x 2883 counties) and 7 columns (the variables). The fields in each row are separated by a comma, and there are no headers in the first line. Here are the first few lines of the file:

```
01001,Autauga,AL,1,27,96.616541353383,99.1324
01001,Autauga,AL,2,28,96.856993190152,95.8209
01001,Autauga,AL,3,16,98.055555555556,96.3528
01001,Autauga,AL,4,23,97.747480735033,95.2189
01001,Autauga,AL,5,22,97.747480735033,92.7127
```

4.3.2 Write housing.txt to the HDFS

To get started, we need to make `housing.txt` available as a text file within the HDFS file system. This puts it in a place where it can be read into R, form subsets, and write the subsets to the HDFS. This is similar to what we do using R in the standard serial way; if we have a text file to read into R, we move put it in a place where we can read it into R, for example, in the working directory of the R session.

To set this up, the system administrator must do two tasks. On the R session server, set up a home directory where you have write permission; let's call it `/home/loginname`. In the HDFS, the administrator does a similar thing, creates, say, `/user/loginname` which is in the root directory.

Your first step, as for the standard R case, is to copy `housing.txt` to a directory on the R-session server where your R session is running. Suppose in your login directory you have created a directory `housing` for your analysis of the housing data. You can copy `housing.txt` to

```
"/home/loginname/housing/"
```

The next step is to get `housing.txt` onto the HDFS as a text file, so we can read it into R on the cluster. There are Hadoop commands that could be used directly to copy the file, but our promise to you is that you never need to use Hadoop commands. There is a RHIPE function, `rhput()` that will do it for you.

```
rhput("/home/loginname/housing/housing.txt", "/user/loginname/housing/housing.txt")
```

The `rhput()` function takes two arguments. The first is the path name of the R server file to be copied. The second argument is the path name HDFS where the file will be written. Note that for the HDFS, in the directory `/user/loginname` there is a directory `housing`. You might have created `housing` already with the command

```
rhmkdir("/user/loginname/housing")
```

If not, then `rhput()` creates the directory for you.

We can confirm that the housing data text file has been written to the HDFS with the `rhexists()` function.

```
rhexists("/user/loginname/housing/housing.txt")
[1] TRUE
```

We can use `rhls()` to get more information about files on the HDFS. It is similar to the Unix command `ls`. For example,

```
rhls("/user/loginname/housing")
  permission         owner      group     size          modtime
1 -rw-rw-rw- loginname supergroup 7.683 mb 2014-09-17 11:11
                                file
/user/loginname/housing/housing.txt
```

4.3.3 Read and Divide by County

Our division method for the housing data will be to divide by county, so there will be 2883 subsets. Each subset will be a `data.frame` object with 4 column variables: `date`, `units`, `listing`, and `selling`. `FIPS`, `state`, and `county` are not column variables because each has only one value for each county; their values are added to the `data.frame` as attributes.

The first step is to read each line of the file `housing.txt` into R. By convention, RHIPE takes each line of a text file to be a key-value pair. The line number is the key. The value is the data for the line, in our case the 7 observations of the 7 variables of the data for one month and one county.

Each line is read as part of Map R code written by the user. The Map input key-value pairs are the above line key-value pairs. Each line also has a Map output key-value pair. The key identifies the county. `FIPS` could have been enough to do this, but it is specified as a character vector with three elements: the 3-vector values of `FIPS`, `state`, and `county`. This is done so that later all three can be added to the subset `data.frame`. The output value for each output key is the observations of `date`, `units`, `listing`, and `selling` from the line for that key.

The Map output key-value pairs are the input key-value pairs for the Reduce R code written by the user. Reduce assembles these into groups by key, that is, the county. Then the Reduce R code is applied to the output values of each group collectively to create the subset `data.frame` object for each county. Each row is the value of one Reduce input key-value pair: observations of `date`, `units`, `listing`, and `selling` for one housing unit. `FIPS`, `state`, and `county` are added to the `data.frame` as attributes. Finally, Reduce writes each subset `data.frame` object to a directory in the HDFS specified by the user. The subsets are written as Reduce output key-value pairs. The output keys are the the values of `FIPS`. The output values are the county `data.frame` objects.

The RHIPE Manager: rhwatch()

We begin with the RHIPE R function `rhwatch()`. It runs the R code you write to specify Map and Reduce operations, takes your specification of input and output files, and manages key-value pairs for you.

The code for the county division is

```
mr1 <- rhwatch(
```

```
  map      = map1,
  reduce   = reduce1,
  input    = rhfmt("/user/loginname/housing/housing.txt", type = "text"),
  output   = rhfmt("/user/loginname/housing/byCounty", type = "sequence"),
  readback = FALSE
)
```

Arguments `map` and `reduce` take your Map and Reduce R code, which will be described below. `input` specifies the input to be the text file in the HDFS that we put there earlier using `rhput()`. The file supplies input key-value pairs for the Map code. `output` specifies the file name into which final output key-value pairs of the Reduce code that are written to the HDFS. `rhwatch()` creates this file if it does not exist, or overwrites it if it does not.

In our division by county here, the Reduce recombination outputs are the 2883 county `data.frame` R objects. They are a `list` object that describes the key-value pairs: `FIPS` key and `data.frame` value. There is one `list` element per pair; that element is itself a list with two elements, the `FIPS` key and then the `data.frame` value.

The Reduce `list` output can also be written to the R global environment of the R session. One use of this is analytic recombination in the R session when the outputs are a small enough dataset. You can do this with the argument `readback`. If `TRUE`, the list is also written to the global environment. If `FALSE`, it is not. If FALSE, it can be written latter using the RHIPE R function `rhread()`.

```
countySubsets <- rhread("/user/loginname/housing/byCounty")
```

Suppose you just want to look over the `byCounty` file on the HDFS just to see if all is well, but that this can be done by looking at a small number of key-value pairs, say 10. The code for this is

```
countySubsets <- rhread("/user/loginname/housing/byCounty", max = 10)
Read 10 objects(31.39 KB) in 0.04 seconds
```

Then you can look at the list of length 10 in various was such as

```
keys <- unlist(lapply(countySubsets, "[[", 1))
keys
[1] "01013" "01031" "01059" "01077" "01095" "01103" "01121" "04001" "05019" "05037"

attributes(countySubsets[[1]][[2]])
$names
[1] "date"             "units"            "listing"             "selling"
$row.names
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
[25] 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
[49] 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66
$state
```

```
[1] "AL"
$FIPS
[1] "01013"
$county
[1] "Butler"
$class
[1] "data.frame"
```

Map R Code The Map R code for the county division is

```
map1 <- expression({
  lapply(seq_along(map.keys), function(r) {
    line = strsplit(map.values[[r]], ",")[[1]]
    outputkey <- line[1:3]
    outputvalue <- data.frame(
      date = as.numeric(line[4]),
      units =  as.numeric(line[5]),
      listing = as.numeric(line[6]),
      selling = as.numeric(line[7]),
      stringsAsFactors = FALSE
    )
  rhcollect(outputkey, outputvalue)
  })
})
```

Map has input key-value pairs, and output key-value pairs. Each pair has an identifier, the key, and numeric-categorical information, the value. The Map R code is applied to each input key-value pair, producing one output key-value pair. Each application of the Map code to a key-value pair is carried out by a mapper, and there are many mappers running in parallel without communication (embarrassingly parallel) until the Map job completes.

RHIPE creates input key-value pair `list` objects, `map.keys` and `map.values`, based on information that it has. Let `r` be an integer from 1 to the number of input key-value pairs. `map.values[[r]]` is the value for key `map.keys[[r]]`. The housing data inputs come from a text file in the HDFS, housing.txt, By RHIPE convention, for a text file, each Map input key is a text file line number, and the corresponding Map input value is the observations in the line, read into R as a single text string. In our case each line value is the observations of the 7 county variables for the line.

This Map code is really a "for loop" with `r` as the looping variable, but is done by `lapply()` because it is in general faster than `for r in 1:length(map.keys)`. The loop proceeds through the input keys, specified by the first argument of `lapply`. The second argument of the above `lapply` defines the Map expression with the argument `r`, an index for the Map keys and values.

The function `strsplit()` splits each character-string line input value into the individual observations of the text line. The result, `line`, is a `list` of length one whose element is a `character vector` whose elements are the line observations. In our case, the observations are a `character vector` of length 7, in order: `FIPS`, `county`, `state`, `date`, `units`, `listing`, `selling`.

Next we turn to the Map output key-value pairs. `outputkey` for each text line is a character vector of length 3 with `FIPS`, `county`, and `state`. `utputvalue` is a `data.frame` with one row and 4 columns, the observations of `date`, `units`, `listing`, and `selling`, each a `numeric` object.

The argument of `data.frame`, `stringsAsFactors`, is is given the value `FALSE`. This leaves character vectors in the `data.frame` as is, and does on convert to a `factor`.

The RHIPE function `rhcollect()` forms a Map output key-value pair for each line, and writes the results to the HDFS as a key-value pair `list` object.

Reduce R Code The Reduce R code for the county division is

```
reduce1 <- expression(
  pre = {
    reduceoutputvalue <- data.frame()
  },
  reduce = {
    reduceoutputvalue <- rbind(reduceoutputvalue, do.call(rbind, reduce.values))
  },
  post = {
    reduceoutputkey <- reduce.key[1]
    attr(reduceoutputvalue, "location") <- reduce.key[1:3]
    names(attr(reduceoutputvalue, "location")) <- c("FIPS","county","state")
    rhcollect(reduceoutputkey, reduceoutputvalue)
  }
)
```

The output key-value pairs of Map are the input key-value pairs to Reduce. The first task of Reduce is to group its input key-value pairs by unique key. The Reduce R code is applied to the key-value pairs of each group by a reducer. The number of groups varies in applications from just one, with a single Reduce output, to many. For multiple groups, the reducers run in parallel, without communication, until the Reduce job completes.

RHIPE creates two list objects `reduce.key` and `reduce.values`. Each element of `reduce.key` is the key for one group, and the corresponding element of `reduce.values` has the values for the group to which the Reduce code is applied. Now in our case, the key is county and the values are the observations of `date`, `units`, `listing`, and `selling` for the all housing units in the county.

Note the Reduce code has a certain structure: expressions `pre`, `reduce`, and `post`. In our case `pre` initializes `reduceoutputvalue` to a `data.frame()`. `reduce` assembles the county `data.frame` as the reducer receives the values through `rbind(reduceoutputvalue, do.call(rbind, reduce.values))`; this uses `rbind()` to add rows to the `data.frame` object. `post` operates further on the result of `reduce`. In our case it first assigns the observation of `FIPS` as the key. Then it adds `FIPS`,`county`, and `state` as `attributes`. Finally the RHIPE function `rhcollect()` forms a Reduce output key-value pair `list`, and writes it to the HDFS.

4.3.4 Compute County Min, Median, Max

With the county division subsets now in the HDFS we will illustrate using them to carry out D&R with a very simple recombination procedure based on a summary statistic for each county of the variable `listing`. We do this for simplicity of explanation of how RHIPE works. However, we emphasize that in practice, initial analysis would almost always involve comprehensive analysis of both the detailed data for all subset variables and summary statistics based on the detailed data.

Our summary statistic consists of the minimum, median, and maximum of `listing`, one summary for each county. Map R code computes the statistic. The output key of Map, and therefore the input key for Reduce is `state`. The Reduce R code creates a `data.frame` for each state where the columns are `FIPS`, `county`, `min`, `median`, and `max`. So our example illustrates a scenario where we create summary statistics, and then analyze the results. This is an analytic recombination. In addition, we suppose that in this scenario the summary statistic dataset is small enough to analyze in the standard serial R. This is not uncommon in practice even when the raw data are very large and complex.

The RHIPE Manager: rhwatch()

Here is the code for `rhwatch()`.

```
CountyStats <- rhwatch(
  map      = map2,
  reduce   = reduce2,
  input    = rhfmt("/user/loginname/housing/byCounty", type = "sequence"),
  output   = rhfmt("/user/loginname/housing/CountyStats", type = "sequence"),
  readback = TRUE
)
```

Our Map and Reduce code, `map2` and `reduce2`, is given to the arguments `map` and `reduce`. The code will be will be discussed later.

The input key-value pairs for Map, given to the argument `input`, are our county subsets which were written to the HDFS directory `/user/loginname/housing` as the key-value pairs `list` object `byCounty`. The final output key-value pairs for Reduce, specified by the argument `output`, will be written to the `list` object `CountyStats` in the same directory as the subsets. The keys are the states, and the values are the `data.frame` objects for the states.

The argument `readback` is given the value TRUE, which means `CountyStats` is also written to the R global environment of the R session. We do this because our scenario is that analytic recombination is done in R.

The argument `mapred.reduce.tasks` is given the value 10, as in our use of it to create the county subsets.

The Map R Code

The Map R code is

```
map2 <- expression({
```

```
  lapply(seq_along(map.keys), function(r) {
    outputvalue <- data.frame(
      FIPS = map.keys[[r]],
      county = attr(map.values[[r]], "location")["county"],
      min = min(map.values[[r]]$listing, na.rm = TRUE),
      median = median(map.values[[r]]$listing, na.rm = TRUE),
      max = max(map.values[[r]]$listing, na.rm = TRUE),
      stringsAsFactors = FALSE
    )
    outputkey <- attr(map.values[[r]], "location")["state"]
    rhcollect(outputkey, outputvalue)
  })
})
```

`map.keys` is the Map input keys, the county subset identifiers `FIPS`. `map.values` is the Map input values, the county subset `data.frame` objects. The `lapply()` loop goes through all subsets, and the looping variable is `r`. Each stage of the loop creates one output key-value pair, `outputkey` and `outputvalue`. `outputkey` is the observation of `state`. `outputvalue` is a `data.frame` with one row that has the variables `FIPS`, `county`, `min`, `median`, and `max` for county `FIPS`. `rhcollect(outputkey, outputvalue)` emits the pairs to reducers, becoming the Reduce input key-value pairs.

The Reduce R Code The Reduce R code for the `listing` summary statistic is

```
reduce2 <- expression(
  pre = {
    reduceoutputvalue <- data.frame()
  },
  reduce = {
    reduceoutputvalue <- rbind(reduceoutputvalue, do.call(rbind, reduce.values))
  },
  post = {
    rhcollect(reduce.key, reduceoutputvalue)
  }
)
```

The first task of Reduce is to group its input key-value pairs by unique key, in this case by `state`. The Reduce R code is applied to the key-value pairs of each group by a reducer.

Expression `pre`, initializes `reduceoutputvalue` to a `data.frame()`. `reduce` assembles the state `data.frame` as the reducer receives the values through `rbind(reduceoutputvalue, do.call(rbind, reduce.values))`; this uses `rbind()` to add rows to the `data.frame` object. `post` operates further on the result of `reduce`; `rhcollect()` forms a Reduce output key-value pair for each state. RHIPE then writes the Reduce output key-value pairs to the HDFS.

Recall that we told RHIPE in `rhwatch()` to also write the Reduce output to `CountyStats` in both the R server global environment. There, we can have a look at the results to make sure all is well. We can look at a summary

```
str(CountyStats)
List of 49
 $ :List of 2
  ..$ : Named chr "AL"
  .. ..- attr(*, "names")= chr "state"
  ..$ :'data.frame':    64 obs. of  5 variables:
  .. ..$ FIPS  : chr [1:64] "01055" "01053" "01051" "01049" ...
  .. ..$ county: chr [1:64] "Etowah" "Escambia" "Elmore" "DeKalb" ...
  .. ..$ min   : num [1:64] 62.1 60.4 94.7 59.2 41.2 ...
  .. ..$ median: num [1:64] 67.6 66.2 99.2 71.9 50.6 ...
  .. ..$ max   : num [1:64] 77.8 79.8 102.2 82.3 60.4 ...
 $ :List of 2
  ..$ : Named chr "AR"
  .. ..- attr(*, "names")= chr "state"
  ..$ :'data.frame':    71 obs. of  5 variables:
  .. ..$ FIPS  : chr [1:71] "05025" "05023" "05021" "05019" ...
  .. ..$ county: chr [1:71] "Cleveland" "Cleburne" "Clay" "Clark" ...
  .. ..$ min   : num [1:71] 46.2 99.9 28.1 61.6 58.5 ...
  .. ..$ median: num [1:71] 60.2 108.2 38.7 67.3 82.1 ...
  .. ..$ max   : num [1:71] 73.5 125 48.8 72.7 117.4 ...
......
```

We can look at the first key-value pair

```
CountyStats[[1]][[2]]
[[1]]
state
 "AL"
```

We can look at the `data.frame` for state "AL"

```
head(CountyStats[[1]][[2]])
        min    median       max  FIPS     county
1  34.88372  51.88628  73.46257 01093     Marion
2  94.66667  99.20582 102.23077 01051     Elmore
3  83.93817  88.59977  94.67041 01031     Coffee
4  92.87617  97.53306 105.71429 01125 Tuscaloosa
5  60.34774  72.46377  93.53741 01027       Clay
6 108.97167 119.66207 128.13390 01117     Shelby
```