ISIT312 Big Data Management

SIM, Session 2, 2021

Exercise 1

Hadoop and HDFS

In this exercise, you will get familiar with using the Linux Shell and Zeppelin to interact with Hadoop.

DO NOT attempt to copy the Linux commands in this document to your working Terminal, because it is error-prone. Type those commands by yourself.

Laboratory Instructions.

First you must install download and install VirtualBox on your system.

You can download VirtualBox from here:

https://www.virtualbox.org/wiki/Downloads

You can find installation procedure here:

https://documents.uow.edu.au/~jrg/115/cookbook/e1-1-A1-frame.html

Connect to Moodle and to download `ova` image of virtual machine use a link

Image of Virtual Machine for ISIT312

available in WEB LINKS section.

When downloaded start VirtualBox and use option `File->Import` to import a file `BigDataVM-07-SEP-2020.ova` into VirtualBox.

After the importation is completed, a VM named `BigDataVM-07-SEP-2020` appears in a column on the left-hand side of VirtualBox window. Use `Settings` option and later on `System` option and allow the virtual machine to use all available `Base Memory` on the "green side" of a scale. In `Display` option grant to the virtual machine all available `Video Memory`. Finally, in `Network` option, check (if necessary, and change) the `Attached to` option to `NAT`.

Use `Start` option to run `BigDataVM-07-SEP-2020`.

Note. There is no need for user name and password.

Note. Next time you use the VM a process of changing `Settings` does not need to be repeated.

(1) Start Shell and Zeppelin

Start a Shell window with `Ctrl + Alt + T` or use the second from bottom icon in a sidebar on the left-hand side of a screen.

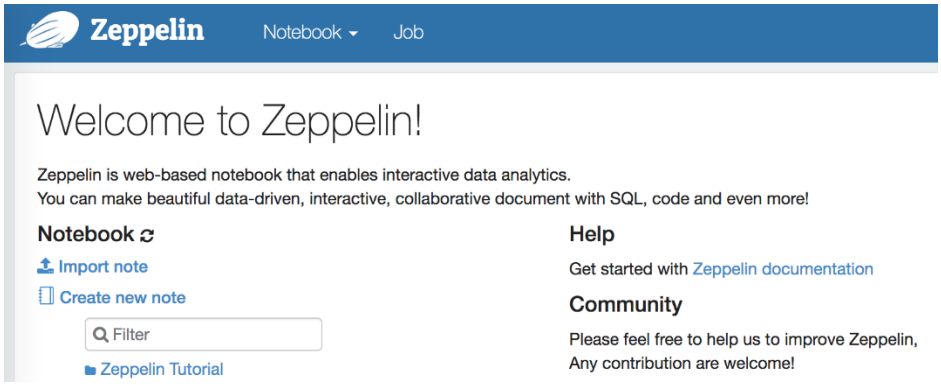
You can use the Linux Shell to interact with Hadoop. However, it is recommended that you use Zeppelin, which provides a better interface. To start Zeppelin, enter in the Shell window:

```
$ZEPPELIN_HOME/bin/zeppelin-daemon.sh start
```

Then use the fourth from bottom icon in the sidebar to start the Firefox browser and use access the following link.

```
127.0.0.1:8080
```

You may need to refresh the browser until you see the following information:



Then, use a link `Create new note` to create a new note and named it as, say, `Lab1`. At the moment you can use `sh` as a default interpreter. Do not create any folders and click at `Create` button to create a note.

A note comprises of many paragraphs. In the first line of each paragraph, you need to indicate the interpreter by inputting `%`*<interpreter>* where a metasymbol *<interpreter>* should be replaced with interpreter name. In this laboratory class, we use the Shell interpreter with command is `%sh`. Type

```
%sh
pwd
```

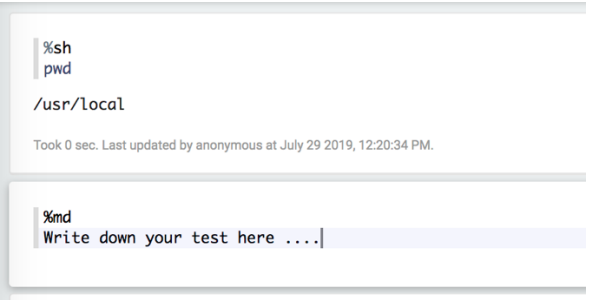
inside a note window (see a picture below). We would like to process `pwd` command (`p`rint `w`orking `d`irectory) in the Shell window.

When you are ready to run your code in a Zeppelin paragraph, click the `Run` button (a small triangle next to `READY` text on the right -hand side of a note window) or use `Shift + Return` keyboard shortcut.

Zeppelin processes pwd command in the Shell window and displays a text:

```
/home/bigdata
```

You can also try to use the `Markdown` interpreter to write down some text.



For example, type the following lines inside Zeppelin paragraph.

```
%md
<b>Hello</b>, <i>Hello</i>, Hello !!!
```

Can you guess from the outcomes what does Markdown do ?

It is always possible top re-edit the contents of Zeppelin's paragraphs and re-run a new code many times. If you do not provide a name of interpreter, for example `%sh` or `%md` then Zeppelin uses a default interpreter that has been determined as `%sh` when we created a new note.

Now you can interact with Hadoop.

(2) Hadoop files and scripts

Have a look at what are contained in the $HADOOP_HOME. Type and process the following lines one by one in the separate Zeppelin's paragraphs.

```
ls $HADOOP_HOME      # view the root directory
ls $HADOOP_HOME/bin  # view the bin directory
ls $HADOOP_HOME/sbin # view the sbin directory
```

Note that # denotes a start of inline comment and `ls` is a Shell command that list the contents of a folder. In another paragraph try the following command to learn what does $HADOOP_HOME mean.

```
echo $HADOOP_HOME
```

The `bin` and `sbin` folders contain scripts for Hadoop.

(3) Hadoop Initialization

Now you can start all Hadoop processes. First, start `HDFSNameNode` and `DataNode` processes.

```
$HADOOP_HOME/sbin/hadoop-daemon.sh start namenode
$HADOOP_HOME/sbin/hadoop-daemon.sh start datanode
```

Next, start YARN `ResourceManager` and `NodeManager`.

```
$HADOOP_HOME/sbin/yarn-daemon.sh start resourcemanager
$HADOOP_HOME/sbin/yarn-daemon.sh start nodemanager
```

Finally, start the `Job History Server`.

```
$HADOOP_HOME/sbin/mr-jobhistory-daemon.sh start historyserver
```

To view the running daemon processes the following command.

```
jps
```

The following Hadoop processes should be listed (note that the process numbers may be different).

```
2897 JobHistoryServer
2993 Jps
2386 NameNode
2585 ResourceManager
2654 NodeManager
2447 DataNode
```

(4) HDFS Shell commands

Create a folder `myfolder` in HDFS process the following command.

```
$HADOOP_HOME/bin/hadoop fs -mkdir myfolder
```

To check if the folder has been created process the following command.

```
$HADOOP_HOME/bin/hadoop fs -ls
```

Copy a file from the local filesystem to HDFS. The following command copy all files with the `.txt` extension in `$HADOOP_HOME` to a folder `myfolder` in HDFS:

```
$HADOOP_HOME/bin/hadoop fs -put $HADOOP_HOME/*.txt myfolder
```

To verify the results, list all `*.txt` files in `$HADOOP_HOME`.

```
ls $HADOOP_HOME/*.txt
```

Next, list the files in `myfolder` folder in HDFS:

```
$HADOOP_HOME/bin/hadoop fs -ls myfolder
```

To view a file in HDFS process the following command.

```
$HADOOP_HOME/bin/hadoop fs -cat myfolder/README.txt
```

Copy a file from HDFS to the `Desktop` folder in a local filesystem process the following command.

```
$HADOOP_HOME/bin/hadoop fs -copyToLocal myfolder/README.txt /home/bigdata/Desktop
```

To verify the results, process the following command.

```
ls /home/bigdata/Desktop
```

To remove a file `README.txt` from HDFS process the following command.

```
$HADOOP_HOME/bin/hadoop fs -rm myfolder/README.txt
```

(5) HDFS user interface

Open another tab in the Firefox Web Browser, and use a link `localhost:50070`.

You will see `localhost:8020`. This is the location of the HDFS. It is specified in a configuration file named `core-site.xml`.

Check this file in a folder `$HADOOP_HOME/etc/hadoop`, that contains Hadoop's configuration files.

Process the following Shell command to view the contents of a file `core-site.xml` in Zeppelin paragraph.

```
cat $HADOOP_HOME/etc/hadoop/core-site.xml
```

Return to and browse the Web user interface, for example, you can see the location of the `Datanode`. Use the options `Utilities` and then `Browser the file system`. Check the `.txt` files uploaded to HDFS previously. Note that the root directory of `bigdata` is in the `user` directory.

To view all root folders in HDFS in Terminal, you can also enter the following command in a new Zeppelin paragraph.

```
$HADOOP_HOME/bin/hadoop fs -ls /
```

(6) HDFS's Java Interface

A Java program listed below retrieves the contents of a file in the HDFS. This program is equivalent to a command `hadoop fs -cat`. A source code of the program (`FileSystemCat.java`) is available in `exercise-1` folder and it is also provided below. Read and understand the source code.

```
// cc FileSystemCat
// Displays files from a Hadoop filesystem on standard output
// by using the FileSystem directly

import java.io.InputStream;
import java.net.URI;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;

// vv FileSystemCat
public class FileSystemCat {

  public static void main(String[] args) throws Exception {
    String uri = args[0];
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(uri), conf);
    InputStream in = null;
    try {
      in = fs.open(new Path(uri));
      IOUtils.copyBytes(in, System.out, 4096, false);
    } finally {
      IOUtils.closeStream(in);
    }
  }
}
// ^^ FileSystemCat
```

Now, create a file `FileSystemCat.java` with the source code and try to compile it. It is important that all commands must be processed in the same Zeppelin paragraph. First, use the following command to define environment variable `HADOOP_CLASSPATH`.

```
export HADOOP_CLASSPATH=$($HADOOP_HOME/bin/hadoop classpath)
```

This environment variable points to all basic Hadoop libraries. Note that each time of open the Terminal and you want to use the environment variable then you must `export` it. To view the libraries, process in the same Zeppelin paragraph the following command.

```
echo $HADOOP_CLASSPATH
```

Now, make sure that the file `FileSystemCat.java` is in your current folder. Now, compile the program and create `jar` file in the following way (still in the same Zeppelin paragraph).

```
javac -cp $HADOOP_CLASSPATH FileSystemCat.java
jar cvf FileSystemCat.jar FileSystemCat*.class
```

The first above command above moves to the current folder that contains the Java source (so that the compilation does not create any package namespace for the main class). The second command compiles the source code. The last command creates a `jar` file that includes the Java class(es).

Now, you can run the jar file by using the `hadoop` script and `jar` command (still in the same Zeppelin paragraph).

```
$HADOOP_HOME/bin/hadoop jar /home/bigdata/Desktop/FileSystemCat.jar FileSystemCat myfolder/LICENSE.txt
```

Check whether the uploaded file is same as the local file.

(7) Shut down Hadoop

When finishing your practice with Hadoop, it is good practice to terminate the Hadoop daemons before turning off the VM.

Use the following command to terminate the Hadoop daemons.

```
$HADOOP_HOME/sbin/hadoop-daemon.sh stop namenode
$HADOOP_HOME/sbin/hadoop-daemon.sh stop datanode
$HADOOP_HOME/sbin/yarn-daemon.sh stop resourcemanager
$HADOOP_HOME/sbin/yarn-daemon.sh stop nodemanager
$HADOOP_HOME/sbin/mr-jobhistory-daemon.sh stop historyserver
```

(8) Make a typescript of information in Terminal

If you work in Shell but not Zeppelin, you can use the `script` command to make a typescript of everything printed in your Terminal.

```
script a-file-name-you-want-to-save-the-typescript-to.txt
<work with your Terminal..>
exit
```

Check the contents of `a-file-name-you-want-to-save-the-typescript-to.txt`.

It is good idea to export Zeppelin note, such that you can import it before the next exercise and reuse the commands processed up to now.

End of Exercise 1

ISIT312 Big Data Management

Session 4, 2020

Exercise 2

Programming MapReduce Applications in Java

In this exercise, you will get familiar with programming of simple MapReduce applications.

Be careful when copying the Linux commands in this document to your working Terminal, because it is error-prone. Maybe you should type those commands by yourself.

Prologue

Login to your system and start VirtualBox.

When ready start a virtual machine `ISIT312-BigDataVM-07-SEP-2020`.

When the virtual machine is running, leftclick at Terminal icon to open Terminal window.

In this practice you can either use Zeppelin interface described in the previous practice or use can use command line interface in Terminal window. If you plan to use Zeppelin then perform the following actions.

(1) Start Zeppelin and create a new notebook named, say, `Lab2`.

(2) Import `Lab1` notebook created and saved in the previous exercise.

(3) Copy the statements processed in `Lab1` into `Lab2` notebook to Start the services of HDFS, YARN, and Job History Server.

If you plan to use command line in terminal window then type all commands at `$` prompt in terminal window.

(1) How to start Hadoop ?

Open a new Terminal window or Zeppelin paragraph and start all Hadoop processes in the following way.

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

```
jps
```

When ready minimize the `Terminal` window used to start Hadoop. We shall refer to this window as to "Hadoop window" and we shall use it later on.

(2) Run MapReduce Applications

In this step, we process two MapReduce applications available in the Hadoop installation. Both applications are included in the `hadoop-mapreduce-examples-2.7.3.jar` file in `$HADOOP_HOME/share/hadoop/mapreduce`.

This first application is the computation of Pi by executing the following command (type 3 lines listed blow in one line):

```
$HADOOP_HOME/bin/hadoop jar
$HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-
2.7.3.jar pi 10 20
```

Another application that uses a regular expression to search for all strings that start from a string `dfs`. First create a folder `input` in HDFS in the following way.

```
$HADOOP_HOME/bin/hadoop fs -mkdir input
```

Next, process the following command to copy the files from etc/Hadoop at a local file system into input folder at HDFS.

```
$HADOOP_HOME/bin/hadoop fs -put /etc/hadoop/conf/* input
```

Next process an application that uses a regular expression to search for all strings that start from a string `dfs`.

```
$HADOOP_HOME/bin/hadoop jar
$HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-
2.7.3.jar grep input output 'dfs[a-z.]+'
```

Note, that to perform the above operation, if the `output` folder exists in your HDFS, you need to remove it first.

Check the outcome of this application through listing the contents of a folder `output`.

```
$HADOOP_HOME/bin/hadoop fs -cat output/part*
```

Finally, remove the folders input and output from HDFS.

```
$HADOOP_HOME/bin/hadoop fs -rm input/*
$HADOOP_HOME/bin/hadoop fs -rmdir input
$HADOOP_HOME/bin/hadoop fs -rm output/*
$HADOOP_HOME/bin/hadoop fs -rmdir output
```

(3) YARN user interface

Enter a link `bigdata-virtualbox:8088` into your web browser. Check the list of applications you just submitted and completed.

(4) How to compile `WordCount.java` Java program

Download a file `WordCount.java` and use editor (`gedit`) to browse its code. The functionality of **WordCount** application and its implementation has been thoroughly discussed during the lecture classes.

To set appropriate environment for compilation of a program process the following command.

```
export HADOOP_CLASSPATH=$($HADOOP_HOME/bin/hadoop classpath)
```

To compile a program process the following command.

```
javac -classpath ${HADOOP_CLASSPATH}  WordCount.java
```

Finally, we have to create jar file in the following way.

```
jar cf WordCount.jar WordCount*.class
```

(5) How to process `WordCount.java` Java program

Next, upload to HDFS home folder any text file. For example we can use a file `WordCount.java`, please see below.

```
$HADOOP_HOME/bin/hadoop fs -put WordCount.java .
```

WordCount application will count the total number occurrences for each word in the file. Finally, process **WordCount** application in the following way.

```
$HADOOP_HOME/bin/hadoop jar WordCount.jar WordCount WordCount.java /user/output/
```

Note, that an input parameter `WordCount.java` determines an input file located in HDFS in `/user/bigdata` folder. An input parameter `/user/output` determines a location of the output files in HDFS.

To see the results from processing of **WordCount** application process the following commands that lists the contents of a file created in HDFS.

```
$HADOOP_HOME/bin/hadoop fs -ls /user/output
```

```
$HADOOP_HOME/bin/hadoop fs -cat /user/output/part-r-00000
```

The first command lists the contents of output folder created by Hadoop and the second command lists the contents of a file with the results of counting.

(6) How to clean after processing of WordCount application

Before we can process **WordCount** application again we have to remove output folder created by Hadoop. Of course, it is also possible to re-process **WordCount** application with a different value of parameter that determines a location of the results.

To delete a folder output we have make it empty first with the following command.

```
$HADOOP_HOME/bin/hadoop fs -rm /user/output/*
```

Then, we can delete the folder with the following command.

```
$HADOOP_HOME/bin/hadoop fs -rmdir /user/output
```

And verify the results with

```
$HADOOP_HOME/bin/hadoop fs -ls /user
```

To remove `WordCount.java` from HDFS process the following command.

```
$HADOOP_HOME/bin/hadoop fs -rm WordCount.java
```

(7) How to process `MinMax.java` Java program

`MinMax` application reads the key-value pairs and for each distinct key it finds the largest value associated with a key. Its functionality is the same as the following SQL statement.

```
SELECT key, MIN(value), MAX(value)
FROM Sequence-of-key-value-pairs
GROUP BY key;
```

Use an editor to examine the contents of a file `MinMax.java`. The application has a very similar structure to WordCount application. The Mapper is almost identical. The Reduce instead of summations find minimal and maximal values associated with each key.

To set appropriate environment for compilation of a program process the following command.

```
export HADOOP_CLASSPATH=$($HADOOP_HOME/bin/hadoop classpath)
```

To compile a program process the following command.

```
javac -classpath ${HADOOP_CLASSPATH}  MinMax.java
```

To create jar file process the following command.

```
jar cf MinMax.jar MinMax*.class
```

Next, upload to HDFS home folder a file `sales.txt` and list its contents in HDFS.

```
$HADOOP_HOME/bin/hadoop fs -put sales.txt .
$HADOOP_HOME/bin/hadoop fs -cat sales.txt
```

Before processing **MinMax** application, make sure that a folder `/user/output` does not exist in HDFS.

```
$HADOOP_HOME/bin/hadoop fs -ls /user
```

If the folder exists then repeat step (2) above.

Finally, start **MinMax** application in the following way.

```
$HADOOP_HOME/bin/hadoop jar MinMax.jar MinMax sales.txt /user/output/
```

To see the results from processing of **MinMax** application process the following command.

```
$HADOOP_HOME/bin/hadoop fs -cat /user/output/part-r-00000
```

Repeat step (2) to remove the results of processing **MinMax** application.

(8) How to process Filter application

An objective of **Filter** application is to select from the contents of a file `sales.txt` all pairs such that amount of sales is greater than a given value. Its functionality is the same as the following SQL statement.

```
SELECT key, value
FROM Sequence-of-key-value-pairs
```

```
WHERE value > given-value;
```

Use an editor to examine the contents of a file `Filter.java`. The application has a simpler structure than WordCount and MinMax applications. The Mapper is almost identical and there is no Reducer. Please see the following line in a Driver `main()`.

```
job.setNumReduceTasks(0); // Set number of reducers to zero
```

To set appropriate environment for compilation of a program process the following command.

```
export HADOOP_CLASSPATH=$($HADOOP_HOME/bin/hadoop classpath)
```

To compile a program process the following command.

```
javac -classpath ${HADOOP_CLASSPATH}  Filter.java
```

To create jar file process the following command.

```
jar cf Filter.jar Filter*.class
```

A file `sales.txt` is already uploaded to HDFS, such that there is no need to do it again.

Before processing **Filter** application, make sure that a folder `/user/output` does not exist in HDFS.

```
$HADOOP_HOME/bin/hadoop fs -ls /user
```

If the folder does exist then repeat step (2) above.

Next, we create a parameterized shell script to process **Filter** application with a parameter that determines the minimal sales from which the transactions are listed. Start gedit editor and open an empty file run in the following way.

```
gedit run.sh
```

Next, type into a file `run.sh` the following line.

```
$HADOOP_HOME/bin/hadoop jar Filter.jar Filter $1 sales.txt /user/output/
```

Note, that `$1` represents a parameter of shell script `run.sh`. In the future we shall use the shell script to process a line listed above and to replace `$1` with a value of actual parameter. Save a file `run.sh` and quit an editor.

Next, we have to grant processing (execute) privileges to a user on a shell script run.sh. To do so process the following command.

```
chmod u+x run.sh
```

Finally, to run **Filter** application process the following command.

```
./run.sh 45
```

Note a value of actual parameter `45` at the end of a command line. The parameter means that we would like to retrieve only sales whose value is greater than 45. When the shell script is processed a value `45` replaces a formal parameter `$1`.

To see the results from processing of **Filter** application process the following command.

```
$HADOOP_HOME/bin/hadoop fs -cat /user/output/part-r-00000
```

Repeat step (2) to remove the results of processing **Filter** application.

(9) How to process `Grep` and `InvIndx` applications ?

`Grep` application finds the words that match a given template provided as a regular expression. You can use command shell scripts `c.sh`, `j.sh`, `load.sh`, `r.sh`, `show.sh`, `clean.sh` in the following way.

To compile the application, process a shell script `c.sh` in the following way.

```
./c.sh
```

To create jar file, process a shell script `j.sh` in the following way.

```
./j.sh
```

To load a file `grep.txt`, process a shell script `load.sh` in the following way.

```
./load.sh grep.txt
```

To run the application process shell script `r.sh` in the following way.

```
./r.sh grep.txt
```

To display the results process a shell script `show.sh` in the following way.

```
./show.sh
```

To remove the results process a shell script `clean.sh` in the following way.

```
./clean.sh
```

It is quite easy to adopt the shell scripts listed above to process an application `InvInd` that creates an inverted index.

End of Exercise 2

ISIT312 Big Data Management

Session 2, 2021

Exercise 3

Introduction to Hive

In this exercise, you will get familiar with how to start Hive Server 2, how to use command line and graphical user interfaces to Hive, how to create internal and external tables in Hive, and how the relational view of data provided by Hive is implemented in HDFS.

Be careful when copying the Linux commands in this document to your working Terminal, because it is error-prone. Maybe you should type those commands by yourself

Prologue

Login to your system and start VirtualBox.

When ready start a virtual machine `ISIT312-BigDataVM-07-SEP-2020`.

(1) Where is Hive Server 2 ?

Open a new Terminal window and process the following commands in Terminal window.

```
echo $HIVE_HOME
echo $HIVE_CONF_DIR
```

The messages tell you where Hive is installed and where is Hive's configuration folder. Configuration folder contains a file `hive-site.xml` that includes the values of Hive configuration parameters.

Process a statement that lists the contents of `hive-site.xml`.

```
cat $HIVE_CONF_DIR/hive-site.xml
```

At the beginning of quite long list of messages you will get the following fragment of XML document.

```
<property>
    <name>javax.jdo.option.ConnectionURL</name>
    <value>jdbc:derby:;databaseName=/usr/share/hive/metastore_db;
    create=true</value>
    <description>
      JDBC connect string for a JDBC metastore.
      To use SSL to encrypt/authenticate the connection, provide
      database-specific SSL flag in the connection URL.
      For example, jdbc:postgresql://myhost/db?ssl=true for
     postgres database.
    </description>
  </property>
```

A value of a property `javax.jdo.option.ConnectionURL` is `jdbc:derby:;databaseName=/usr/share/hive/metastore_db;create=true` and it tells us what relational DBMS is used to implement Metastore (Derby) and where Metastore is located (`/usr/share/hive/metastore_db`). Metastore (data dictionary or data repository in

traditional DBMSs) contains all information about the mappings of Hive relational tables into the files in HDFS. **Deletion or re-initialization of Metastore means that all such mappings are lost !** Data located in HDFS is not changed. If you would like to reinitialize metastore (you probably do not need to do it now) then you have to first remove the present metastore_db folder from $HIVE_HOME

```
rm –rf $HIVE_HOME/metastore_db
```

and the process `schematool` program in the following way.

```
$HIVE_HOME/bin/schematool -initSchema -dbType derby
```

Process a statement:

```
ls $HIVE_HOME/bin
```

`hiveserver2` is Hive2 Thrift server that will be used to access HDFS visible as a collection of relational tables. `beeline` is a command line interface to Hive2 server, but we use Zeppelin in this lab. `schematool` is a program for initialization of Hive Metastore.

(2) How to start Metastore service and Hive Server 2 ?

To start Hive's metastore service, open ***a Terminal window***, type:

```
$HIVE_HOME/bin/hive --service metastore
```

A message shows that metastore is up and running:

```
SLF4J: Actual binding is of type
[org.apache.logging.slf4j.Log4jLoggerFactory]
```

To start hiveserver2, open ***another Terminal window*** and type:

```
$HIVE_HOME/bin/hiveserver2
```

The same message shows that hiveserver2 is up and running.

[IMPORTANT] Don't use Zeppelin to run the above two commands, because the %sh interpreter has a timeout threshold.

You can use Hive's own interface to interact with Hive. Open another new Terminal window and process the following command.

```
$HIVE_HOME/bin/beeline
```

Process the following cmmand in front of `beeline>` prompt.

```
!connect jdbc:hive2://localhost:10000
```

The statement connects you through JDBC interface to Hive 2 server running on a `localhost` and listening to a port `10000`.

Press Enter when prompted about user name. Press Enter when prompted about password. The system should reply with a prompt

```
0: jdbc:hive2://localhost:10000>
```

To find what databases are available process the following command.

```
show databases;
```

At the moment only `default` database is available. We shall create new databases in the future.
To find what tables have been created so far process a statement

```
show tables;
```

(3) How to create an internal relational table ?

To work with Hive in Zeppelin, you need to use the hive interpreter in the Zeppelin paragraphs.

Open a new Zeppelin paragraph. Type the following in the beginning of the paragraph:

```
%hive
```

which indicates that the Hive interpreter is used.

In the following contents, we use Beeline to interact with Hive.

To create a single column relational table process the following statement in "beeline window".

```
create table hello(message varchar(50));
```

Try again

```
show tables;
```

and

```
describe hello;
```

in "beeline window".

Hive created an internal relational table hello in HDFS. Is it possible to find a location of the table in HDFS ?

(4) How to find a location of internal relational table in HDFS ?

If you use Zeppelin then open a new paragraph and use %sh as the interpreter.

If you use command line interface then move to "Hadoop window"

Next, process the following command.

```
$HADOOP_HOME/bin/hadoop fs -ls /user
```

Note a new folder hive created in HDFS folder. Use ↑(up) arrow key to redisplay a command processed just now and at the end of the command add `/hive`, i.e. process the following command.

```
$HADOOP_HOME/bin/hadoop fs -ls /user/hive
```

The results show that a folder `hive` is not empty and it contains a folder `warehouse`. Use the following command to investigate what are the contents of a folder `warehouse`.

```
$HADOOP_HOME/bin/hadoop fs -ls /user/hive/warehouse
```

And here we find our relational table `hello` implemented as a folder in HDFS. One more time, try to find what is in `hello` folder.

```
$HADOOP_HOME/bin/hadoop fs -ls /user/hive/warehouse/hello
```

There is nothing there because the table is empty.

(5) How to insert a row into an internal relational table ?

If you use Zeppelin then return to %hive paragraph,

If you use command line interface then return to "beeline window".

To insert a row into a relational table hello process the following statement.

```
insert into hello values ('Hello world !');
```

Note, that insertion of a row takes some time. In the future we shall not use this way to populate Hive tables.

What about HDFS ? What has changed in HDFS after insertion of a row ? Return to "Hadoop window" and process the most recently processed command again.

```
$HADOOP_HOME/bin/hadoop fs -ls /user/hive/warehouse/hello
```

A new file has been added to `/user/hive/warehouse/hello` HDFS folder.

Process the following command to list the contents of a new file.

```
$HADOOP_HOME/bin/hadoop fs -cat /user/hive/warehouse/hello/000000_0
```

And here we have a row recently inserted into a table `hello`.

Return to "beeline window" and insert few more rows. Then return to "Hadoop window" and list the contents of `hello` folder in the following way.

```
$HADOOP_HOME/bin/hadoop fs -ls /user/hive/warehouse/hello
```

Try the following command to list the contents of all rows.

```
$HADOOP_HOME/bin/hadoop fs -cat /user/hive/warehouse/hello/000000*
```

(6) How to create and how to process SQL script ?

Right click Terminal icon at the bottom of task bar and open a new Terminal window.

Use a command

```
gedit hello.hql
```

to open a text editor with an empty file hello.hql.

Insert into the file the following lines.

```
insert into hello values('Hello HQL !');
select * from hello;
```

Save a file and quit gedit editor. When you open a new Terminal window and you start gedit editor your current folder is your home folder and because of that the edited file is saved in your home folder. Note, that beeline has been started from you home folder as well. This is why you can process a script file hello.hql through beeline without providing a path to the script file. Now return to "beeline window" and process HQL script just created with the following command.

```
!run hello.hql
```

If you would like to save a report from processing of HQL script then you should first process a command

```
!record hello.rpt
```

then process HQL script with

```
!run hello.hql
```

and finally stop recording with

```
!record
```
(no file name !)

Your report from processing of HQL script is stored in a file hello.rpt in the current folder of beeline which in this case is your home folder. Return to Terminal window used to create HQL script and process the following command to list the contents of a report.

```
cat hello.rpt
```

Now, you can close a Terminal window used to create HQL script file.

(7) How to use SQL Developer ?

To start SQL Developer leftclick at the second icon from top of task bar (an icon with a pretty large and green triangle) and wait for a while.

When ready look at Connections subwindow in the left upper corner of SQL Developer window and find hive connection there. It is a connection to Hive Server 2 we have already created for you. Rightclick at a connection icon (hive) and pick Connect option.

SQL Developer asks about Connection Information. Type any name you like and any password you like, and click at OK button.

SQL Developer adds default database to hive connection in Connections subwindow. Leftclick at a small + in front of default name. SQL Developer add Tables item. Leftclick at a small + in front of Table item, and so on ... Finally, left click at hello. SQL Developer lists the contents of hello table in its main window.

Now, retun to hive panel in SQL Developer main window. To process HQL statement type

```
select * from hello where message like '%world !';
```

into worksheet window and leftclick at a large green triangle in the left upper corner of `hive` panel.

To access HQL script use option `File->Open` from the main SQL Developer menu and pick a file `hello.hql` in `bigdata` home folder.

To process the script click at an icon with a small green triangle next to an icon with a large green triangle. Confirm that you would like to use `hive` connection. A report from processing appears at the bottom of `hello.hql` panel. You can use a "floppy disk" icon to save it.

To disconnect SQL Developer from Hive Server 2 rightclick at `hive` connection icon and pick `Disconnect` option.

Use `File->Exit` option to quit SQL Developer. You can save HQL from worksheets into the files if you like.

(8) How to load data into an internal relational table ?

In this step you can use either Zeppelin interface or beeline command interface or SQL Developer to Hive Server 2. In Zeppelin interface use %hive paragraph.

A specification below is consistent with Zeppelin or beeline command interfaces.

As we found earlier loading data to an internal relational table with `insert` statement takes considerable amount of time. Additionally, Hive is not a typical database system that should be used to process online transactions where a small amount of data is collected from a user and inserted into a relational table. Hive supposed to operate on the large amounts of data that should be inserted into the relational table for more convenient processing with HQL. In this step, we practice a way how a large and well formatted data set can be loaded into an internal relational table.

Right click at Terminal icon at the bottom of task bar and open a new window. Start `gedit` editor and create a new file called `names.tbl`. Insert into the file the following 3 lines, save the file, and quit `gedit`.

```
James,Bond,35
Harry,Potter,16
Robin,Hood,120
```

Start `gedit` again and create HQL script `names.hql` with the following `create table` statement.

```
create table names(
 first_name VARCHAR(30),
 last_name  VARCHAR(30),
 age        DECIMAL(3) )
  row format delimited fields terminated by ',' stored as textfile;
```

Save the file, quit `gedit`, and close Terminal window.

Now, move to "beeline window" and process HQL script `names.hql` script with `!run` command.

```
!run names.hql
```

To load into a relational table `names` data from a file `names.tbl` process the following statement in "beeline window".

```
load data local inpath 'names.tbl' into table names;
```

Verify the results with

```
select * from names;
```

(9) How to load data into an external relational table ?

When loading data into an internal relational table data located in a local file system get replicated in HDFS. It is much better to move data into HDFS and "overlap" ("cover") it with a definition of an external relational table.

To do so copy a file `names.tbl` to HDFS. Move to "Hadoop window" and process the following commands.

```
$HADOOP_HOME/bin/hadoop fs -put names.tbl /user/bigdata
$HADOOP_HOME/bin/hadoop fs -ls /user/bigdata
```

Right click at Terminal icon at the bottom of task bar and open a new window. Start `gedit` editor and create HQL script `enames.hql` with the following create table statement.

```
create external table enames(
 first_name VARCHAR(30),
 last_name  VARCHAR(30),
 age        DECIMAL(3) )
  row format delimited fields terminated by ','
  stored as textfile location '/user/bigdata';
```

Save the file, quit `gedit`, and close Terminal window.

Move to "beeline window" and process a script file `enames.hql` in the following way.

```
!run enames.hql
```

Finally, list the contents of an external table enames.

```
select * from enames;
```

If you move to "Hadoop window" and list the names of relational tables located in HDFS `/user/hive/warehouse` with

```
$HADOOP_HOME/bin/hadoop fs -ls /user/hive/warehouse/
```

then only the relational tables `hello` and `names` are listed. This is because an external relational table `enames` is located in `/user/bigdata/names.tbl`. An external relational table is equivalent to a definition of the tables stored in Metastore and mapped on a file in HDFS `/user/bigdata/names.tbl`.

Move to "beeline window" and drop an external relational table `enames`.

```
drop table enames;
```

Then, move to "Hadoop window" and check if a file `names.tbl` still exists in `/user/bigdata` and it is not empty.

```
$HADOOP_HOME/bin/hadoop fs -ls /user/bigdata
$HADOOP_HOME/bin/hadoop fs -cat /user/bigdata/names.tbl
```

Deletion of an external relational table deletes its definition from Metastore only. Now, move to "beeline window" and drop an internal table `names`.

```
drop table names;
```

Finally, move to "Hadoop window" and check if a file `names.tbl` still exists in `/user/hive/warehouse`.

```
$HADOOP_HOME/bin/hadoop fs -ls /user/hive/warehouse/
```

Deletion of an internal relational table deletes its definition in Metastore and a file in HDFS that implements the table.

(10) How to create a database ?

In this step you can use either Zeppelin interface or beeline command interface or SQL Developer to Hive Server 2. In Zeppelin interface use %hive paragraph .

Move to "beeline window" if you use command line interface. To create a new database `tpchr` process the following statement.

```
create database tpchr;
```

A database is created as a new folder `tpchr.db` in `/user/hive/warehouse` folder in HDFS. To verify location the database move to "Hadoop window" and process the following command.

```
$HADOOP_HOME/bin/hadoop fs -ls /user/hive/warehouse
```

A database can be created in a location different from `/user/hive/warehouse`. For example, process the following statement in "beeline window".

```
create database other location '/user/hive';
```

A database `other` is created in `/user/hive` folder however no new folder is created. Process the following command in "Hadoop window".

```
$HADOOP_HOME/bin/hadoop fs -ls /user/hive
```

To get more information about `default`, `tpchr`, and `other` databases process the following commands in "beeline window".

```
show databases;
describe database default;
describe database tpchr;
describe database other;
```

Move to SQL Developer and refresh with blue arrows `Connections` subwindow. To restore information about the databases leftclick at small plus in front of `hive` connection.

When connected beeline assumes that initially you are connected to `default` database.

The current database is a `default` database. It is possible to change a current database with `use` command. Process the following statement in "beeline window".

```
use other;
create table hello(message varchar(50));
```

Refresh `Connections` subwindow in SQL Developer and leftclick at small `+` in front of `other` database icon and later on at `Tables` item to show the tables created in `other` database.

Move to "Hadoop window" and process a command:

```
$HADOOP_HOME/bin/hadoop fs -ls /user/hive
```

to find location of `hello` table in HDFS. Note, that it is possible to have many tables with the same names in different databases (like in MySQL).

(11) How to access a database ?

To access a database as your current database use a command `use`. For example, to make `other` database your current database process a statement `use` in "beeline window".

```
use other;
insert into hello values( 'Hello James !');
```

To access a relational table located in a database that is not your currrent database you have to prefix a table name with a database name and with `.` (dot). For example, change a current database in "beeline window" to `default`.

```
use deafult;
```

To access a relational table `hello` located in `other` database process the following statement.

```
select * from other.hello;
```

(12) How to drop a database ?

To drop `other` database process a statement:

```
drop database other;
```

The system cannot drop a nonempty database. Drop a table `hello` first and then drop a database `other`.

```
drop table hello;
drop database other;
```

(13) How to create and how to access a relational table with a column of type `array` ?

It is possible in Hive to create relational tables with columns of type `array`, `map`, `struct`, `named struct`, and `union`.

From the relational database theory point of view such tables are not in 1NF and they are also called as nested tables, unnormalized tables or 0NF tables.

Make `default` database your current database. First, we create a table `friend` that contains information about friends.

```
create table friend(
 name varchar(30),
 friends array<string> );
```

Use `describe` command to verify the structures of the table. Next, we shall insert few rows into the table.

```
insert into friend select 'James',array('Kate','John');
insert into friend select 'John', array(NULL);
insert into friend select 'Kate', array();
insert into friend select 'Harry',array('James');
```

Use `select` statement to verify the contents of a relational tables `friend`. Note a difference between `NULL` and `array[]`). It is possible to bind different interpretations to `NULL` and empty array (`array([])`).

```
select * from friend;
```

To select a particular element from an array we provide a number of an element in an array. For example, to list the first 3 friends of each person process the following `select` statement.

```
select  name, friends[0], friends[1], friends[2]
from friend;
```

(14) How to create and how to access a relational table with a column of type `map` ?

Create a relational table `workshop` to keep information about the workshops, types of tools available at each workshop and total number of tools for each type.

```
create table workshop(
 name varchar(50),
 tools map<string,int> );
```

Use `describe` command to verify the structures of the table. Next, we shall insert few rows into the table.

```
describe workshop;

insert into workshop
            select 'XYZ Ltd.',map('screwdriver',30,'hammer',1);
insert into workshop select 'Mitra10', map('hammer',1);

select * from workshop;
```

We use a key value to select a particular element from a map. For example, to select the total number of hammers in each workshop process the following `select` statement.

```
select name, tools['hammer'] hammers
from workshop;
```

To select workshops that have at least one hammer process the following `select` statement.

```
select name, tools['hammer'] hammers
from workshop
where tools['hammer'] > 0;
```

(15) How to create and how to access a relational table with a column of type `struct` ?

Create a relational table `employee` to keep information about employees.

```
create table employee(
  enumber decimal(7),
  address struct<city:string,street:string,house:int,flat:int> );
```

Use `describe` command to verify the structures of the table. Next, we shall insert few rows into the table.

```
describe employee;

insert into employee
       select 007, named_struct('city','London',
                   'street','Victoria St.','house',7,'flat',77);
insert into employee
       select 123, named_struct('city','Dapto',
                   'street','Station St.','house',1,'flat',0);

select * from employee;
```

To select a particular element from a structure we provide a field name. For example, to select the names of cities the employees live in we process the following statement.

```
select enumber, address.city city
from employee;
```

To find all employees living London we process the following statement.

```
select *
from employee
where address.city = 'London';
```

(16) How to create a new connection in SQL Developer ?

Assume that we would like to create a new connection in SQL Developer to a new database. To create a new connection leftclick at `Connections` icon in `Connection` subwindow of SQL Developer window.

Next leftclick at large green + (plus) at the top of Connections subwindow. SQL Developer opens New/Select Database Conmnection window.

Type a name of a new connection into Connection Name field. A connection name is up to you.

A user name and password are also up to you.

Next, left click at Hive tab to make it active (initially Oracle tab is active).

Type localhost into Host name field.

Type 10000 into Port field.

Type a name of a database created earlier, e.g. tpchr into Database field.

Do not change the contents of Driver field.

When ready leftclick at Test button. When "Success" word is displayed in a left lower corner of New/Select Database Connection window leftclick at Connect button. SQLDeveloper creates a new connection in Connections subwindow and opens a new tab in the main window.

By choosing tabs in the main Window of SQL Developer you can use different connections to process HQL statements.

End of Exercise 3

ISIT312 Big Data Management

Session 2, 2021

Exercise 4

Conceptual and Logical Modelling of Data Warehouse and more of Hive

In this exercise, you will learn how to use UMLet to create a conceptual and logical schema of a data warehouse, how to create partitioned tables, how to create bucket tables, how to export and import tables, and how to find a query processing plan in Hive.

Be careful when copying the Linux commands in this document to your working Terminal, because it is error-prone. Maybe you should type those commands by yourself.

Prologue

Login to your system and start VirtualBox.

When ready start a virtual machine `ISIT312-BigDataVM-07-SEP-2020`.

(1) How to start UMLet ?

Find UMLet icon on desktop and use the icon to start UMLet 14.3.

(2) How to create a conceptual schema of a data warehouse ?

(2.1) Read he following specification of a sample data warehouse domain.

A multinational company consists of departments located in different countries. A department is described by a name and mission statement. The employees work on the projects. A project is described by a name and deadline. An employee is described by an employee number and full name. Employees work on projects. When an employee completes a project then he/she is re-employed by a company to work on another project. The company would like to record in a data warehouse information about the total number of employees, length of each employment, total salary paid on each employment per year, per month, per project, per department, per city and per country. For example, it should be possible to find the projects and the total number of employees working on each project, or the total number of employees employed in each in each month of a given year in each department, or average salary in each month of each year and for each project etc.

(2.2) First, we identify a fact entity and the measures.

Consider the following fragment of a specification given above.

The company would like to record in a data warehouse information about the total number of employees, length of each employment, total salary paid on each employment per year, per project, per department, per city and per country.

The fragment contributes to a fact entity EMPLOYMENT described by the measures length and salary.

To create a fact entity EMPLOYMENT drag a graphical component fact entity that looks like a cube (Fact name and Measure1, Measure 2,…) from a panel with the graphical widgets to the drawing area of UMLet. Next, change a name of fact from Fact name to EMPLOYMENT and measures Measure1, Measure 2, … to length and salary (the measures are yellow highlighted in a fragment of the specification above). To do so you have to leftclick at the fact entity to make it blue and then modify its properties in a panel "Properties" in the right lower corner of UMLet window.

(2.3) Next, we add the dimensions and attributes describing the entity types in dimensions.

The following fragment of specification indicates the existence of Time dimension that consists of entity types MONTH and YEAR (see a yellow highlighted text in a fragment below).

The company would like to record in a data warehouse information about the total number of employees, length of each employment, total salary paid on each employment per year, per month, per project, per department, per city and per country.

First, we create Time dimension that consists of the entity types MONTH and YEAR. To do so drag two entity type graphical components (Level name, attribute, …) from a panel with graphical widgets to the drawing area of UMLet. Then, change the names of entity types to MONTH and YEAR and connect the entities with one-to-many relationship graphical component, Finally, use one-to-many relationship graphical component to connect MONTH entity to a fact entity EMPLOYMENT.

Next, add an attribute name to an entity MONTH and attribute number to an entity YEAR. To do so you have to leftclick at an entity to make it blue and then modify its properties in a panel "Properties" in the right lower corner of UMLet window.

In the same way add three more dimensions: PROJECT, EMPLOYEE, and DEPARTMENT and describe each entity type with the attributes listed in the specification (see below).

A project is described by a name and deadline.

An employee is described by an employee number and full name.
A department is described by a name and mission statement.

(2.4) Finally, we create the hierarchies over the dimensions.

In this step we create Location and Time hierarchies over the dimensions DEPARTMENT and TIME(MONTH-YEAR). First add a hierarchy blob Criterion between entity type MONTH and many-to-one relationship leading to an entity YEAR. Replace a text Criterion with a text Time.

Next, create two more entity types CITY and COUNTRY and describe both of them with an attribute name. Add one-to-many relationship between COUNTRY and CITY and one-to-many relationship between CITY and EMPLOYMENT.

Finally, to create Location hierarchy add a blob Criterion between entity DEPARTMENT and many-to-one relationship leading to an entity CITY. Replace a text Criterion with a text Location.

To save your design in a file employment.uxf use File->Save option from the main menu. To create pdf file use File->Export as … option from the main menu. When creating pdf file make sure that none of the graphical component in the main window of UMLet is blue highlighted !

(3) How to create a logical schema of a data warehouse ?

To create a logical schema (a collection of relational schemas) of a data warehouse we start from a diagram of conceptual schema created in the previous step. First, we change in a panel located in the right upper corner of UMLet window a collection of graphical widgets from "Conceptual modeling" to "Logical modeling".

Next, we add to each entity type, except fact entity EMPLOYMENT, a surrogate key. For example, we add a surrogate key year_ID to entity YEAR, month_ID to entity MONTH, project_ID to entity PROJECT, etc. We nominate all _ID attributes to be primary keys in the respective relational tables. The names of relational tables remain the same as the names of the respective entity types.

Next, we migrate the primary keys from one side of one-to-many relationships to the relational tables to many side of the relationships and we nominate the migrated _ID attribute as foreign keys.

Next, we nominate a collection of all foreign keys in a table obtained from fact entity type EMPLOYEMENT as a composite primary key in a relational table EMPLOYMENT.

Finally, we replace one-to-many relationships with arrows directed from the locations of foreign keys to the locations of the respective primary keys. A logical schema can be saved and exported in the same way as a conceptual schema.

(4) How to start Hadoop, Hive Server 2, beeline or SQL Developer ?

Open a new Terminal window and start Hadoop in the following way.

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

When ready minimize the Terminal window used to start Hadoop. We shall refer to this window as to "Hadoop window" and we shall use it later on.

When ready navigate to a folder where you plan to keep your HQL scripts from this lab (you may have to create such folder now) and start Hive Server 2 in the following way.

To start Hive's metastore service, open ***a Terminal window***, type:

```
$HIVE_HOME/bin/hive --service metastore
```

A message shows that metastore is up and running:

```
SLF4J: Actual binding is of type
[org.apache.logging.slf4j.Log4jLoggerFactory]
```

To start hiveserver2, open ***another Terminal window*** and type:

```
$HIVE_HOME/bin/hiveserver2
```

The same message shows that hiveserver2 is up and running.

[IMPORTANT] Don't use Zeppelin to run the above two commands, because the %sh interpreter has a timeout threshold.

You can use Hive's own interface to interact with Hive. Open another new Terminal window and process the following command.

```
$HIVE_HOME/bin/beeline
```

Process the following cmmand in front of `beeline>` prompt.

```
!connect jdbc:hive2://localhost:10000
```

The statement connects you through JDBC interface to Hive 2 server running on a `localhost` and listening to a port `10000`.

Press Enter when prompted about user name. Press Enter when prompted about password. The system should reply with a prompt

```
0: jdbc:hive2://localhost:10000>
```

To find what databases are available process the following command.

```
show databases;
```

At the moment only `default` database is available. We shall create new databases in the future. To find what tables have been created so far process a statement

```
show tables;
```

If you plan to use SQL Developer then leftclick at SQL Developer Icon at the top of task bar. Then rightclick at `hive` connection icon, pick "Connection" option from a menu, enter any user name and password and leftclick at OK button.

We shall use a `default` database for Hive tables created in this laboratory exercise.

(5) How to create and load data into an internal table ?

Create the following internal table to store information about the items.

```
create table item(
 code  char(7),
 name   varchar(30),
 brand  varchar(30),
 price  decimal(8,2) )
    row format delimited fields terminated by ','
    stored as textfile;
```

Next create a text file `items.txt` with sample data given below and save a file in a folder where you plan to keep HQL scripts from this lab (you already started Hive Server 2 from this folder).

```
B000001,bolt,Golden Bolts,12.34
B000002,bolt,Platinum Parts,20.0
B000003,bolt,Unbreakable Spares,17.25
S000001,screw,Golden Bolts,45.00
S000002,screw,Platinum Parts,12.0
N000001,nut,Platinum Parts,21.99
```

When ready process the following HQL statement to load the contents of a file `items.txt` into `item` table.

```
load data local inpath 'items.txt' into table item;
```

Verify the contents of a table `item` with a simple

```
select * from item;
```

(6) How to create a partitioned internal table ?

We would like to improve performance of processing a table `item` through partitioning We expect that a lot of queries will retrieve only the items that have a given name, like for example a query

```
select min(price) from item where name ='bolt';
```

Therefore, we shall create a new table `pitem` as a partitioned table. Process the following statement to create a partitioned table `pitem`.

```
create table pitem(
 code  char(7),
 brand  varchar(30),
 price  decimal(8,2) )
    partitioned by (name varchar(30))
    row format delimited fields terminated by ','
    stored as textfile;
```

Note, that a column `name` has been removed from a list of columns and added as a partition key.

Next, we shall create the partitions for the values in a column `name` of `item` table. Process the following `alter table` statements to create the partitions for `bolt`, `screw`, and `nut`.

```
alter table pitem add partition (name='bolt');
alter table pitem add partition (name='screw');
alter table pitem add partition (name='nut');
```

Next, process the following statement to check if all partitions have been created.

```
show partitions pitem;
```

Now, we shall copy data from a table `item` into a partitioned table `pitem`. Process the following `INSERT` statements.

```
insert into table pitem partition (name='bolt') select code, brand,
price
from item
```

```
where name='bolt';

insert into table pitem partition (name='screw') select code,
brand, price
from item
where name='screw';

insert into table pitem partition (name='nut') select code, brand,
price
from item
where name='nut';
```

Finally, try

```
select * from pitem;
```

to check if all data has been copied.

(7) How to "explain" a query processing plan ?

Are there any differences in the ways how Hive processes an internal table and an internal partitioned table ?

`explain` statement can be used to find the differences in processing of the following `select` statements.

```
explain extended select * from item where name='bolt';
explain extended select * from pitem where name='bolt';
```

The first statement processed on a table item returns the following processing plan.

```
Explain
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------
STAGE DEPENDENCIES:
  Stage-0 is a root stage

STAGE PLANS:
  Stage: Stage-0
    Fetch Operator
      limit: -1
      Processor Tree:
        TableScan
          alias: item
          Statistics: Num rows: 1 Data size: 203 Basic stats:
COMPLETE Column stats: NONE
          GatherStats: false
          Filter Operator
            isSamplingPred: false
            predicate: (UDFToString(name) = 'bolt') (type: boolean)
            Statistics: Num rows: 1 Data size: 203 Basic stats:
COMPLETE Column stats: NONE
```

```
            Select Operator
              expressions: code (type: char(7)), 'bolt' (type:
varchar(30)), brand (type: varchar(30)), price (type: decimal(8,2))
              outputColumnNames: _col0, _col1, _col2, _col3
              Statistics: Num rows: 1 Data size: 203 Basic stats:
COMPLETE Column stats: NONE
              ListSink


22 rows selected.
```

The second statement processed on a partitioned table `pitem` returns a plan.

```
Explain
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------
STAGE DEPENDENCIES:
  Stage-0 is a root stage

STAGE PLANS:
  Stage: Stage-0
    Fetch Operator
      limit: -1
      Partition Description:
          Partition
            input format: org.apache.hadoop.mapred.TextInputFormat
            output format:
org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat
            partition values:
              name bolt
            properties:
              COLUMN_STATS_ACCURATE {"BASIC_STATS":"true"}
              bucket_count -1
              columns code,brand,price
              columns.comments
              columns.types char(7):varchar(30):decimal(8,2)
              field.delim ,
              file.inputformat
org.apache.hadoop.mapred.TextInputFormat
              file.outputformat
org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat
              location
hdfs://localhost:8020/user/hive/warehouse/pitem/name=bolt
              name default.pitem
              numFiles 1
              numRows 3
              partition_columns name
              partition_columns.types varchar(30)
              rawDataSize 86
              serialization.ddl struct pitem { char(7) code,
varchar(30) brand, decimal(8,2) price}
              serialization.format ,
```

```
              serialization.lib
org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe
              totalSize 89
              transient_lastDdlTime 1534561429
            serde:
org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe

              input format:
org.apache.hadoop.mapred.TextInputFormat
              output format:
org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat
              properties:
                bucket_count -1
                columns code,brand,price
                columns.comments
                columns.types char(7):varchar(30):decimal(8,2)
                field.delim ,
                file.inputformat
org.apache.hadoop.mapred.TextInputFormat
                file.outputformat
org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat
                location
hdfs://localhost:8020/user/hive/warehouse/pitem
                name default.pitem
                partition_columns name
                partition_columns.types varchar(30)
                serialization.ddl struct pitem { char(7) code,
varchar(30) brand, decimal(8,2) price}
                serialization.format ,
                serialization.lib
org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe
                transient_lastDdlTime 1534561343
              serde:
org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe
              name: default.pitem
            name: default.pitem
      Processor Tree:
        TableScan
          alias: pitem
          Statistics: Num rows: 3 Data size: 86 Basic stats:
COMPLETE Column stats: NONE
          GatherStats: false
          Select Operator
            expressions: code (type: char(7)), brand (type:
varchar(30)), price (type: decimal(8,2)), 'bolt' (type:
varchar(30))
            outputColumnNames: _col0, _col1, _col2, _col3
            Statistics: Num rows: 3 Data size: 86 Basic stats:
COMPLETE Column stats: NONE
            ListSink


68 rows selected.
```

Comparison of `Statistics: Num rows` indicates a smaller amount of data processed in the second case.

```
Statistics: Num rows: 1 Data size: 203 Basic stats: COMPLETE
Column stats: NONE
Statistics: Num rows: 3 Data size: 86 Basic stats: COMPLETE Column
stats: NON
```

(8) How partitions are implemented in HDFS ?

To find how partitions are implemented in HDFS move to "Hadoop window" and process the following command.

```
$HADOOP_HOME/bin/hadoop fs -ls /user/hive/warehouse/pitem
```

The partitions of `pitem` tables are implemented as subfolders in `/user/hive/warehouse/pitem`.

(8) How to create dynamically partitioned tables ?

The partitions of dynamically partitioned table are determined when data is loaded to the table.

Process the following statements to set the parameters that allow for dynamic partitioning.

```
set hive.exec.dynamic.partition.mode=nonstrict;
set hive.exec.max.dynamic.partitions=5;
```

Next, drop a table `pitem` and recreate it in the same way as at the beginning of step 6.

```
create table pitem(
 code  char(7),
 brand  varchar(30),
 price  decimal(8,2) )
    partitioned by (name varchar(30))
    row format delimited fields terminated by ','
    stored as textfile;
```

Next, process the following statement to dynamically the partitions and to copy the rows from a table `item` into a table `pitem`.

```
insert overwrite table pitem partition(name)
select code, brand, price, name
from item;
```

Then process the statements to display the partitions and the contents of a table pitem.

```
show partitions pitem;
```

```
select * from pitem;
```

(9) How to create a bucket table ?

Another method to distribute a table in HDFS is to partition it into *bucket*s. A *bucket* is equivalent to a segment of file in HDFS. A column in a table can be selected to determine a distribution of rows over buckets. The value of this column will be hashed into a number and a row will be stored in a bucket with the same number. The rows with the same value in the selected column will be stored in the same bucket. However, the same bucket may also contain the rows with other values from the selected column. A concept of *bucket* is equivalent to a concept of *clustered hash index* on the traditional relational database systems.

Process the following statement to create a table `bitem` distributed over two buckets.

```
create table bitem(
 code  char(7),
 name   varchar(30),
 brand  varchar(30),
 price  decimal(8,2) )
    clustered by (name) into 2 buckets
    row format delimited fields terminated by ','
    stored as textfile;
```

Process the following statement to copy the contents of a table `item` into a table `bitem`.

```
insert overwrite table bitem
select code, name, brand, price
from item;
```

Display a query processing plan for a bucket table.

```
explain extended select * from bitem where name='bolt';
```

```
Explain
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
------------------------------------------------------
STAGE DEPENDENCIES:
  Stage-0 is a root stage

STAGE PLANS:
  Stage: Stage-0
    Fetch Operator
      limit: -1
      Processor Tree:
        TableScan
          alias: bitem
          Statistics: Num rows: 6 Data size: 199 Basic stats:
COMPLETE Column stats: NONE
          GatherStats: false
          Filter Operator
            isSamplingPred: false
```

```
              predicate: (UDFToString(name) = 'bolt') (type: boolean)
              Statistics: Num rows: 3 Data size: 99 Basic stats:
  COMPLETE Column stats: NONE
              Select Operator
                expressions: code (type: char(7)), 'bolt' (type:
  varchar(30)), brand (type: varchar(30)), price (type: decimal(8,2))
                outputColumnNames: _col0, _col1, _col2, _col3
                Statistics: Num rows: 3 Data size: 99 Basic stats:
  COMPLETE Column stats: NONE
                ListSink
22 rows selected.
```

(10) How to export a table ?

To transfer Hive tables from one Hadoop installation to another one can use `export` and `import` statements. To export an internal table `item` into a folder `exp-item` in HDFS process the following statement.

```
export table item to '/tmp/exp-item'
```

To verify the storage structures created by `export` statement process the following commands in "Hadoop window".

```
$HADOOP_HOME/bin/hadoop fs -ls /tmp/exp*
$HADOOP_HOME/bin/hadoop fs -ls /tmp/exp-item/data
```

Just for fun you can also list the contents of metadata.

```
$HADOOP_HOME/bin/hadoop fs -cat /tmp/exp-item/_meta.
```

(11) How to import a table ?

Assume that we would like to import the contents of exported table `item` into a new table `imported_item`. Process the following commands.

```
import table imported_item from '/tmp/exp-item';
select * from imported_item;
```

End of Exercise 4

ISIT312 Big Data Management

Session 2, 2021

Exercise-5

SQL for Data Warehousing and still more of Hive

In this exercise, you will learn how to use Hive HQL extensions for Data Warehousing. A laboratory includes application of `SELECT` statement with GROUP BY clause, advanced features of `GROUP BY` clause (`ROLLUP`, `CUBE`, `GROUPING SETS`), windowing and analytics functions.

Be careful when copying the Linux commands in this document to your working Terminal, because it is error-prone. Maybe you should type those commands by yourself.

Prologue
Login to your system and start VirtualBox.

When ready start a virtual machine `ISIT312-BigDataVM-07-SEP-2020`.

(1) How to start Hadoop, Hive Server 2, beeline or SQL Developer ?

Open a new Terminal window and start Hadoop in the following way.

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

When ready minimize the Terminal window used to start Hadoop. We shall refer to this window as to "Hadoop window" and we shall use it later on.

When ready navigate to a folder where you plan to keep your HQL scripts from this lab (you may have to create such folder now) and start Hive Server 2 in the following way.
To start Hive's metastore service, open ***a Terminal window***, type:

```
$HIVE_HOME/bin/hive --service metastore
```

A message shows that metastore is up and running:

```
SLF4J: Actual binding is of type
[org.apache.logging.slf4j.Log4jLoggerFactory]
```

To start hiveserver2, open ***another Terminal window*** and type:

```
$HIVE_HOME/bin/hiveserver2
```

The same message shows that hiveserver2 is up and running.

[IMPORTANT] Don't use Zeppelin to run the above two commands, because the %sh interpreter has a timeout threshold.

You can use Hive's own interface to interact with Hive. Open another new Terminal window and process the following command.

```
$HIVE_HOME/bin/beeline
```

Process the following command in front of `beeline>` prompt.

```
!connect jdbc:hive2://localhost:10000
```

The statement connects you through JDBC interface to Hive 2 server running on a `localhost` and listening to a port `10000`.

Press Enter when prompted about user name. Press Enter when prompted about password. The system should reply with a prompt

```
0: jdbc:hive2://localhost:10000>
```

To find what databases are available process the following command.

```
show databases;
```

At the moment only `default` database is available. We shall create new databases in the future. To find what tables have been created so far process a statement

```
show tables;
```

If you plan to use SQL Developer then leftclick at SQL Developer Icon at the top of task bar. Then rightclick at hive connection icon, pick "Connection" option from a menu, enter any user name and password and leftclick at OK button.

We shall use a default database for Hive tables created in this laboratory exercise.

(2) How to create and how to load data into an internal table ?

We shall use default database for Hive table created, loaded, and used in this laboratory exercise. Create the following internal table to store information about items.

```
create table ORDERS(
 part     char(7),
 customer varchar(30),
 amount   decimal(8,2),
 oyear    decimal(4),
 omonth   decimal(2),
 oday     decimal(2) )
   row format delimited fields terminated by ','
              stored as textfile;
```

The table represents a three-dimensional data cube. A fact entity `orders` is described by a measure `amount`. The dimensions include `part`, `customer`, and obviously

`time(oyear,omonth,oday)` dimension. There is a hierarchy over `time` dimension where years consist of months and months consist of days.

Next create a text file `items.txt` with sample data given below and save a file in a folder where you plan to keep HQL scripts from this lab (you already started Hive Server 2 from this folder).

```
bolt,James,200,2016,01,01
bolt,Peter,100,2017,01,30
bolt,Bob,300,2018,05,23
screw,James,20,2017,05,11
screw,Alice,55,2018,01,01
nut,Alice,23,2018,03,16
washer,James,45,2016,04,24
washer,Peter,100,2016,05,12
bolt,James,200,2018,01,05
bolt,Peter,100,2018,01,05
```

To load data into a table `orders` process the following `load` statement.

```
load data local inpath 'items.txt' into table orders;
```

To verify the contents of a table orders process the following select statement.

```
select * from orders;
```

(3) How to perform a simple aggregation with `group by` and `having` clauses ?

We start from implementation of a query that *finds the total number of orders per each part*, i.e. we perform aggregation along a dimension part. Create a process the following `select` statement.

```
select part, count(*)
from orders
group by part;
```

Next, we *find the total number of ordered parts summarized per each part*. It is another aggregation along a dimension `part`.

```
select part, sum(amount)
from orders
group by part;
```

Next, we *find the total number of orders per customer* and we list only the customers who submitted more than one order.

```
select customer, count(*)
from orders
group by customer
having count(*) > 1;
```

Now, assume that we would like *to find in one query the total number of orders per each part, per each customer and per both part and customer*.

```
select part, NULL, count(*)
from orders
group by part
union
select NULL, customer, count(*)
from orders
group by customer
union
select part, customer, count(*)
from orders
group by part, customer;
```

(4) How to perform aggregations with `rollup` operator ?

Assume that we would like to perform aggregation over two dimensions, then over one of the two dimensions used earlier and then aggregation over all orders. For example, *find the total number of parts ordered and summarized per part and per customer, then per part and then total number of all parts ordered*.

```
select part, customer, sum(amount)
from orders
group by part, customer
union
select part, null, sum(amount)
from orders
group by part
union
select null, null,sum(amount)
from orders;
```

The same query can be implemented as a single `select` statement with `rollup` operator.

```
select part, customer, sum(amount)
from orders
group by part, customer with rollup;
```

In the next example we use `rollup` operator to implement a query that *finds the total number of parts ordered and summarized per year and month, per year, and the total number of parts ordered.*

```
select oyear, omonth, sum(amount)
from orders
group by oyear,omonth with rollup;
```

(5) How to perform aggregation with `cube` operator ?

Assume that we would like to *find an average number of parts ordered and summarized per part, per customer, per both part and customer and an average number of parts per order.*

```
select part, customer, avg(amount)
from orders
group by part, customer with cube;
```

It is possible to verify some of the results with the following queries.

```
select avg(amount)
from orders;
```

and

```
select part,avg(amount)
from orders
where part='bolt'
group by part;
```

(6) How to perform aggregations with `grouping sets` operator ?

Assume that we would like to *find the total number of orders per part and per customer and both per year and customer*.

```
select part, customer, oyear, count(*)
from ORDERS
group by part, customer, oyear
grouping sets ((part),(customer), (oyear,customer));
```

In another example we *find the total number of parts ordered and summarized per year, month, day, per year and month , and per year, and the total number of parts ordered*.

```
SELECT oyear, omonth, oday, sum(amount)
from orders
group by oyear,omonth,oday grouping
sets((oyear,omonth,oday),(oyear,omonth), (oyear),() );
```

(7) How to perform window based aggregations ?

It is possible to use `group by` clause of select statement to find the total number of ordered parts summarized per each part.

```
select part, sum(amount)
from orders
group by part;
```

It is possible to get the similar result as from a query with `group by` clause with so called *windowing*.

```
select part, SUM(amount) over (partition by part)
from orders;
```

```
bolt        900.00
bolt        900.00
bolt        900.00
bolt        900.00
bolt        900.00
nut         23.00
```

```
screw       75.00
screw       75.00
washer      145.00
washer      145.00
```

To get the same results we have to use distinct keyword.

```
select distinct part, SUM(amount) over (partition by part)
from orders;
```

```
bolt        900.00
nut         23.00
screw       75.00
washer      145.00
```

Now we use windowing to implement a query that *finds for each part, for each customer, and for each amount ordered by customer the largest total number of parts ordered and aggregated per part*.

```
select part, customer, amount, MAX(amount) over (partition by part)
from orders;
```

```
bolt        Peter     100.00     300.00
bolt        James     200.00     300.00
bolt        Bob       300.00     300.00
bolt        Peter     100.00     300.00
bolt        James     200.00     300.00
nut         Alice     23.00      23.00
screw       Alice     55.00      55.00
screw       James     20.00      55.00
washer      Peter     100.00     100.00
washer      James     45.00      100.00
```

A table orders is partitioned (grouped by) the values in column part and for each part the largest amount is found and added to each output row that consists of `part`, `customer` and `amount`.

It is possible to use more than one aggregation. For example, we extend a query above with the *summarization of the amounts per each part*.

```
select part, customer, amount,
       MAX(amount) over (partition by part),
       SUM(amount) over (partition by part)
from orders;
```

(7) How to perform window aggregations and window ordering ?

We start from a query that *finds for each part and amount ordered the total number of parts ordered and summarized per each part*.

```
select part, amount, SUM(amount) over (partition by part)
from orders;
```

The results are the following.

```
bolt      100.00      900.00
bolt      200.00      900.00
bolt      300.00      900.00
bolt      100.00      900.00
bolt      200.00      900.00
nut       23.00       23.00
screw     55.00       75.00
screw     20.00       75.00
washer    100.00      145.00
washer    45.00       145.00
```

Now, we add a clause `order by` to *windowing*. Process the following statement.

```
select part, amount,
       SUM(amount) over (partition by part order by amount)
from orders;
```

```
bolt      100.00      200.00 |<-- 100+100
bolt      100.00      200.00 |<-- 100+100
bolt      200.00      600.00   |<-- 200+200
bolt      200.00      600.00   |<-- 200+200
bolt      300.00      900.00
nut       23.00       23.00
screw     20.00       20.00
screw     55.00       75.00
washer    45.00       45.00
washer    100.00      145.00
```

Addition of `order by` clause computes the increasing results of summarization over the amounts and orders the rows in each partition by the summarized amount. If two rows have the same values of `order by amount` then the rows are treated as one row with summarized `amount`. For example the first two rows have the same values of `order by amount` and because of that a value of `SUM(amount) = 100+100`. The same applies to the next two rows. If two or more rows have the same values of part and amount then summarization is performed in one step over all such rows. This problem (if it is really a problem ?) can be solved with more selective `order by` key. For example, the rows in each window can be ordered by `amount`, `oyear`, `omonth`, and `oday`.

```
select part,amount,
       SUM(amount) over (partition by part
                         order by amount, oyear, omonth, oday)
from orders;
```

In this case the rows in each window are ordered by `amount`, `oyear`, `omonth`, oday and summarization is performed in a row-by-row way.

```
bolt      100.00      100.00
bolt      100.00      200.00
bolt      200.00      400.00
bolt      200.00      600.00
bolt      300.00      900.00
nut       23.00       23.00
screw     20.00       20.00
screw     55.00       75.00
washer    45.00       45.00
```

```
washer     100.00     145.00
```

To *find how the ordered amounts of ordered parts changed year by year* process the following `select` statement.

```
select part, amount, oyear,
       SUM(amount) over (partition by part order by oyear)
from orders;
```

```
bolt        200.00     2016     200.00
bolt        100.00     2017     300.00
bolt        100.00     2018     900.00
bolt        200.00     2018     900.00
bolt        300.00     2018     900.00
nut         23.00      2018     23.00
screw       20.00      2017     20.00
screw       55.00      2018     75.00
washer      100.00     2016     145.00
washer      45.00      2016     145.0
```

Now, we change aggregation function to `AVG`.

```
select part, amount,
             AVG(amount) over (partition by part
                               order by amount, oyear, omonth, oday)
from orders;
```

The statement finds so called *walking average*.

```
bolt     100.00     100.000000 AVG(100)
bolt     100.00     100.000000 AVG(100+100)
bolt     200.00     133.333333 AVG(100+100+200)
bolt     200.00     150.000000 AVG(100+100+200+200)
bolt     300.00     180.000000 AVG(100+100+200+200+300)
nut      23.00      23.000000
screw    20.00      20.000000
screw    55.00      37.500000
washer   45.00      45.000000
washer   100.00     72.500000
```

(8) How to perform window aggregations and window framing ?

Next, implement a query that for each part and amount *finds an average of amount ordered by year, month and day.*

```
select part, amount,
             AVG(amount) over (partition by part
                               order by oyear, omonth, oday)
from orders;
```

```
bolt     200.00     200.000000 AVG(200)
bolt     100.00     150.000000 AVG(200+100)
bolt     100.00     150.000000 AVG(200+100)
bolt     200.00     150.000000 AVG(200+100+100+200)
```

```
bolt     300.00     180.000000 AVG(200+100+100+200+300)
nut      23.00      23.000000
screw    20.00      20.000000
screw    55.00      37.500000
washer   45.00      45.000000
washer   100.00     72.500000
```

Processing of aggregation (average) is performed over an expanding frame. At the beginning a *frame* includes the first row, next first row an second row, next first 3 rows, etc.

It is possible to make a frame fixed size and smaller than a window. Process the following statement.

```
select part, amount,
             AVG(amount) over (partition by part
                               order by oyear, omonth, oday
                               rows 1 preceding)
from orders;
```

The statement finds for each part and amount an average amount of the current and previous one amount when the amounts are sorted in time.

```
bolt     200.00     200.000000 AVG(200)
bolt     100.00     150.000000 AVG(200+100)
bolt     100.00     100.000000 AVG(100+100)
bolt     200.00     150.000000 AVG(100+200)
bolt     300.00     250.000000 AVG(200+300)
nut      23.00      23.000000
screw    20.00      20.000000
screw    55.00      37.500000
washer   45.00      45.000000
washer   100.00     72.500000
```

Processing of a statement

```
select part, amount,
             AVG(amount) over (partition by part
                               order by oyear, omonth, oday
                               rows unbounded preceding)
from orders;
```

returns the same results as without rows *frame* (the first select statement in this section).

The options of *window framing* are the following.

```
(ROWS | RANGE) BETWEEN (UNBOUNDED | [num]) PRECEDING AND ([num]
PRECEDING | CURRENT ROW | (UNBOUNDED | [num]) FOLLOWING)
```

For example:

```
ROWS BETWEEN 3 PRECEDING AND CURRENT ROW,
ROWS BETWEEN UNBOUNDED PRECEDING AND 2 FOLLOWING
```

```
(ROWS | RANGE) BETWEEN CURRENT ROW AND (CURRENT ROW | (UNBOUNDED |
[num]) FOLLOWING)
```

For example:

```
ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
```

```
(ROWS | RANGE) BETWEEN [num] FOLLOWING AND (UNBOUNDED | [num]) FOLLOWING
```

For example:

```
ROWS BETWEEN 2 FOLLOWING AND UNBOUNDED FOLLOWING
```

(9) How to use `window` clause ?

It is possible to simplify syntax a bit with `window` clause (definition). Implement the following query.

```
select part, SUM(amount) over w
from orders
window w as (partition by part);
```

(10) How to use `LEAD` and `LAG` functions ?

`LEAD` and `LAG` functions allow to access the previous and the next values in a column. For example, we would like to *find the current and the next amount for each `part` ordered by year, month, day*.

```
select part, amount,
             LEAD(amount) over (partition by part
                                order by oyear, omonth,oday)
from orders;
```

```
bolt     200.00      100.00
bolt     100.00      100.00
bolt     100.00      200.00
bolt     200.00      300.00
bolt     300.00
nut      23.00
screw    20.00       55.00
screw    55.00
washer   45.00       100.00
washer   100.00
```

Next, we would like to *find the current and the previous amount for each `part` ordered by year, month, day*.

```
select part, amount,
             LAG(amount) over (partition by part
                               order by oyear, omonth, oday)
from orders;
```

```
bolt       200.00
bolt       100.00    200.00
bolt       100.00    100.00
bolt       200.00    100.00
```

```
bolt        300.00     200.00
nut         23.00
screw       20.00
screw       55.00      20.00
washer      45.00
washer      100.00      45.00
```

Next we subtract the previous row value from the current row value.

```
select part, amount,
             amount-LAG(amount) over(partition by part
                                     order by oyear, omonth, oday)
from orders;
```

```
bolt        200.00
bolt        100.00     -100.00
bolt        100.00     0.00
bolt        200.00     100.00
bolt        300.00     100.00
nut         23.00
screw       20.00
screw       55.00      35.00
washer      45.00
washer      100.00      55.00
```

Empty places (NULLs) can be eliminated with a parameter `0` in `LAG` function.

```
select part, amount,
             amount+lag(amount,1,0) over(partition by part
                                       order by oyear, omonth, oday)
from orders;
```

```
bolt        200.00     200.00
bolt        100.00     300.00
bolt        100.00     200.00
bolt        200.00     300.00
bolt        300.00     500.00
nut         23.00      23.00
screw       20.00      20.00
screw       55.00      75.00
washer      45.00      45.00
washer      100.00     145.00
```

(11) How to use analytics functions ?

Finally, we implement windowing with the analytics functions `RANK()`, `DENSE_RANK()`, `CUM_DIST()`,

A function `RANK()` assigns a rank to row such that the rows with the same value of `amount` are ranked with the same number and rank is increased by the total number of rows with the same value.

```
select part, amount,
             RANK() over (partition by part order by amount)
from orders;
```

```
bolt        100.00     1 RANK=1
bolt        100.00     1 RANK=1
bolt        200.00     3 RANK=1+2
bolt        200.00     3 RANK=1+2
bolt        300.00     5 RANK=3+2
nut         23.00      1
screw       20.00      1
screw       55.00      2
washer      45.00      1
washer      100.00     2
```

A function `DENSE_RANK()` assigns a rank to row such that the rows with the same value of `amount` are ranked with the same number and rank is increased by 1 for each group of rows with the same value of `amount`.

```
select part, amount,
             DENSE_RANK() over (partition by part
                                order by amount)
from orders;
```

```
bolt        100.00     1
bolt        100.00     1
bolt        200.00     2
bolt        200.00     2
bolt        300.00     3
nut         23.00      1
screw       20.00      1
screw       55.00      2
washer      45.00      1
washer      100.00     2
```

A function `CUME_DIST()` computes the relative position of a specified value in a group of values. For a given row r, the `CUME_DIST()` the number of rows with values lower than or equal to the value of r, divided by the number of rows being evaluated, i.e. entire window.

```
select part, amount,
             CUME_DIST() over (partition by part
```

```
                                    order by amount)
from orders;
```

```
bolt        100.00    0.4 2 rows/5
bolt        100.00    0.4 2 rows/5
bolt        200.00    0.8 4 rows/5
bolt        200.00    0.8 4 rows/5
bolt        300.00    1.0 5 rows/5
nut         23.00     1.0
screw       20.00     0.5
screw       55.00     1.0
washer      45.00     0.5
washer      100.00    1.0
```

A function `PERCENT_RANK()` is similar to a function `CUME_DIST()`. For a row `r`, `PERCENT_RANK()` calculates the rank of `r` minus 1, divided by the number of rows being evaluated -1 , i.e. entire window-1.

```
select part, amount,
       PERCENT_RANK() over (partition by part
                            order by amount)
from orders;
```

```
bolt        100.00    0.0
bolt        100.00    0.0
bolt        200.00    0.5
bolt        200.00    0.5
bolt        300.00    1.0
nut         23.00     0.0
screw       20.00     0.0
screw       55.00     1.0
washer      45.00     0.0
washer      100.00    1.0
```

A function `NTILE(k)` divides a window into a number of buckets indicated by `k` and assigns the appropriate bucket number to each row. The buckets are numbered from `1` to `k`.

```
select part, amount,
       NTILE(2) over (partition by part
                                    order by amount)
from orders;
```

```
bolt        100.00    1
bolt        100.00    1
bolt        200.00    1
bolt        200.00    2
bolt        300.00    2
nut         23.00     1
screw       20.00     1
screw       55.00     2
washer      45.00     1
washer      100.00    2
```

```
select part, amount,
       NTILE(5) over (partition by part
                                 order by amount)
from orders;
```

```
bolt        100.00     1
bolt        100.00     2
bolt        200.00     3
bolt        200.00     4
bolt        300.00     5
nut         23.00      1
screw       20.00      1
screw       55.00      2
washer      45.00      1
washer      100.00     2
```

Finally, a function `ROW_NUMBER` does not need any explanations.

```
select part, amount,
       ROW_NUMBER() over (partition by part
                                  order by amount)
from orders;
```

```
bolt        100.00     1
bolt        100.00     2
bolt        200.00     3
bolt        200.00     4
bolt        300.00     5
nut         23.00      1
screw       20.00      1
screw       55.00      2
washer      45.00      1
washer      100.00     2
```

End of Exercise 5

ISIT312 Big Data Management

Session 2, 2021

Exercise 6

Introduction to HBase

In this exercise, you will learn how to create, how to load data, how to perform data manipulations, and how to search HBase tables using Hbase shell commands. You will also learn how to use API to process HBase tables.

Be careful when copying the Linux commands in this document to your working Terminal, because it is error-prone. Maybe you should type those commands by yourself.

Prologue
Login to your system and start VirtualBox.

When ready start a virtual machine `ISIT312-BigDataVM-07-SEP-2020`.

(1) How to start Hadoop and HBase server ?

Open a new Terminal window and start Hadoop in the following way.

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

When ready navigate to a folder where you plan to keep HBase scripts from this lab (you may have to create such folder now) and start HBase server in the following way.

```
$HBASE_HOME/bin/start-hbase.sh
```

(2) How to start HBase command shell ?

To start command line interface to HBase process the following command in a Terminal window.

```
$HBASE_HOME/bin/hbase shell
```

For a good start use a command `help` to get a pretty comprehensive help from HBase command Line Interface (CLI). A complete printout of help is listed at the end of this document.

(3) How to use HBase CLI ?

HBase CLI provide several commands to be processed at `hbase main: ...:0>` command prompt. For example, processing of a command `whoami` at the prompt returns the following message.

```
bigdata (auth:SIMPLE)
```

```
    groups: bigdata, adm, cdrom, sudo, dip, plugdev, lpadmin,
sambashare
```

Processing of a command `version` returns the following messages.

```
1.2.6, rUnknown, Mon May 29 02:25:32 CDT 2017
```

Processing of a command `table_help` returns information about table-reference commands (see Appendix B in this document).

Command `status` returns the following message.

```
1 active master, 0 backup masters, 1 servers, 0 dead, 2.0000
average load
```

(4) How to create a script file and how to process a script file ?

To create HBase CLI script start `gedit` editor, type in the lines

```
status
whoami
```

and save a file as `script.hb`.

To process a script file type the following command at `hbase` prompt:

```
source('script.hb')
```

and press Enter key.

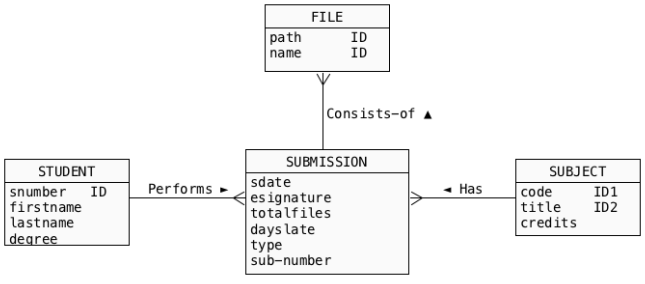
If you would like to process a script and save the results in a file then exit HBase CLI and in a Terminal window process the following command.

```
$HBASE_HOME/bin/hbase shell < script.hb > report.rpt
```

The command starts HBase shall that reads input from a file `script.hb` and outputs the results into a file `report.rpt`.

(5) How to design a HBase table ?

Consider the following conceptual schema.



We would like to implement a conceptual schema given above as a single HBase table. We create four column families: `STUDENT`, `SUBJECT`, `FILE`, and `SUBMISSION` and we distribute the rows over the column families in a way schematically presented below. I would called as "a relational implementation style" ;). We concatenate the names of entity types with the values of identifiers of entity instances and we use it as the keys of rows in HBase table. The submitted files will be represented as column qualifiers. it means that a column family `FILE` will have a variable number of column qualifiers depending on the total number of submitted files.

```
                    STUDENT      SUBJECT        SUBMISSION  FILE
student:snumber1    xxxxxxxxxxx
student:snumber2    xxxxxxxxxxx
…
subject:code1                    xxxxxxxxxxx
subject:code2                    xxxxxxxxxxx
…
submission:skey1    xxx          xxx            xxxxxxxxxxx xxxxxx
submission:skey2    xxx          xxx            xxxxxxxxxxx xxx
…
```

where a submission `skey-n` consists of `snumber-n|code-n|sub-number-n`. The rows that describe submissions spread over all column families. The total number of overlaps determines what we call in the relational model as "denormalization" or simply speaking "redundancies" in HBase table.

(6) How to create a HBase table ?

To create a HBase table we must provide a name of a table and a name of at least one column family. Process the following command to create HBase table `COURSEWORK` with a column family `STUDENT`.

```
create 'COURSEWORK', 'STUDENT'
```

To verify a structure of HBase table use `describe` command.

```
describe 'COURSEWORK'
```

(7) How to add the new column families to an existing HBase table ?

Next, we use `alter` command to add a column family `SUBJECT` to HBase table `COURSEWORK`.

```
alter 'COURSEWORK', {NAME=>'SUBJECT', VERSIONS=>'1'}
```

In our design we plan to represent each entity type as a separate column family. Process the following commands to add the column families `FILE` and `SUBMISSION`.

```
alter 'COURSEWORK', {NAME=>'FILE', VERSIONS=>'2'}
alter 'COURSEWORK', {NAME=>'SUBMISSION', VERSIONS=>'1'}
```

Use `describe` command to verify the structures of HBase table `COURSEWORK`.

```
describe COURSEWORK
```

(8) How to enter data into HBase table ?

First, we shall add few rows to a column family `STUDENT`. We shall use the values of attribute `snumber` prefixed with a string `'student'` as a key of rows in HBase table. The following commands add to a column family a row with key `'student:007'`, and the column qualifiers `snumber`, `fname`, `lname`, `degree`, and `put` the values into cells of HBase table.

```
put 'COURSEWORK','student:007','STUDENT:snumber','007'
put 'COURSEWORK','student:007','STUDENT:first-name','James'
put 'COURSEWORK','student:007','STUDENT:last-name','Bond'
put 'COURSEWORK','student:007','STUDENT:degree','MIT'
```

To verify the insertions use `scan` command.

```
scan 'COURSEWORK'
```

A sequence of `put` commands implements an insertion of a single row with a key `'student:007'` into HBase table `COURSEWORK`.

Now, add all remaining data to a table `COURSEWORK`. The next sequence of put commands inserts the next row into column family `STUDENT`.

```
put 'COURSEWORK','student:666','STUDENT:snumber','666'
put 'COURSEWORK','student:666','STUDENT:firstname','Harry'
put 'COURSEWORK','student:666','STUDENT:lastname','Potter'
put 'COURSEWORK','student:666','STUDENT:degree','BCS'
```

Next, we insert two rows into a column family `SUBJECT`.

```
put 'COURSEWORK','subject:312','SUBJECT:code','312'
put 'COURSEWORK','subject:312','SUBJECT:title','Big Data'
put 'COURSEWORK','subject:312','SUBJECT:credits','6'
put 'COURSEWORK','subject:313','SUBJECT:code','313'
put 'COURSEWORK','subject:313','SUBJECT:title','Very Big Data'
put 'COURSEWORK','subject:313','SUBJECT:credits','12'
```

Next, we insert information about a submission performed by one of the students enrolled in one of the subjects.

```
put 'COURSEWORK','submission:007|312|1','SUBMISSION:sdate','01-APR-2017'
put 'COURSEWORK','submission:007|312|1','SUBMISSION:esignature','jb'
put 'COURSEWORK','submission:007|312|1','SUBMISSION:totalfiles','2'
put 'COURSEWORK','submission:007|312|1','SUBMISSION:dayslate','0'
put 'COURSEWORK','submission:007|312|1','STUDENT:snumbner','007'
put 'COURSEWORK','submission:007|312|1','SUBJECT:code','312'
put 'COURSEWORK','submission:007|312|1','FILE:fnumber1','path/file-name1-1'
put 'COURSEWORK','submission:007|312|1','FILE:fnumber2','path/file-name1-1'
```

Note, that a row with a key `'submission:007|312|1'` contributes to column families `SUBMISSION`, `STUDENT`, `SUBJECT`, `FILE`.

It is possible to "denormalize" the row by inserting the values into the cells labelled with the column qualifiers in `STUDENT` and `SUBJECT` to the rows that describe submissions. For example, the following commands add the first and the last name of a student who performed a submission.

```
put 'COURSEWORK','submission:007|312|1','STUDENT:firstname','James'
put 'COURSEWORK','submission:007|312|1','STUDENT:lastname','Bond'
```

Verify a structure of HBase table after all additions.

```
describe 'COURSEWORK'
```

Add all remaining data.

```
put 'COURSEWORK','submission:007|313|1','SUBMISSION:sdate','02-APR-2017'
put 'COURSEWORK','submission:007|313|1','SUBMISSION:esignature','jb'
put 'COURSEWORK','submission:007|313|1','SUBMISSION:totalfiles','2'
put 'COURSEWORK','submission:007|313|1','SUBMISSION:dayslate','0'
put 'COURSEWORK','submission:007|313|1','SUBMISSION:type','project'
put 'COURSEWORK','submission:007|313|1','STUDENT:snumbner','007'
put 'COURSEWORK','submission:007|313|1','SUBJECT:code','313'
put 'COURSEWORK','submission:007|313|1','FILE:fnumber1','path/file-name3-1'

put 'COURSEWORK','submission:666|312|3','SUBMISSION:sdate','01-APR-2017'
put 'COURSEWORK','submission:666|312|3','SUBMISSION:esignature','hp'
put 'COURSEWORK','submission:666|312|3','SUBMISSION:totalfiles','2'
put 'COURSEWORK','submission:666|312|3','SUBMISSION:dayslate','0'
put 'COURSEWORK','submission:666|312|3','SUBMISSION:type','assignment'
put 'COURSEWORK','submission:666|312|3','STUDENT:snumbner','666'
put 'COURSEWORK','submission:666|312|3','SUBJECT:code','312'
put 'COURSEWORK','submission:666|312|3','FILE:fnumber1','path/file-name1-1'
put 'COURSEWORK','submission:666|312|3','FILE:fnumber2','path/file-name1-1'

put 'COURSEWORK','submission:666|312|4','SUBMISSION:sdate','02-APR-2017'
put 'COURSEWORK','submission:666|312|4','SUBMISSION:esignature','hp'
put 'COURSEWORK','submission:666|312|4','SUBMISSION:totalfiles','2'
put 'COURSEWORK','submission:666|312|4','SUBMISSION:dayslate','0'
put 'COURSEWORK','submission:666|312|4','SUBMISSION:type','assignment'
put 'COURSEWORK','submission:666|312|4','STUDENT:snumbner','666'
put 'COURSEWORK','submission:666|312|4','SUBJECT:code','312'
put 'COURSEWORK','submission:666|312|4','FILE:fnumber1','path/file-name2-1'
put 'COURSEWORK','submission:666|312|4','FILE:fnumber2','path/file-name2-2'

put 'COURSEWORK','submission:666|313|2','SUBMISSION:sdate','02-APR-2017'
put 'COURSEWORK','submission:666|313|2','SUBMISSION:esignature','hp'
put 'COURSEWORK','submission:666|313|2','SUBMISSION:totalfiles','2'
put 'COURSEWORK','submission:666|313|2','SUBMISSION:dayslate','0'
put 'COURSEWORK','submission:666|313|2','SUBMISSION:type','project'
put 'COURSEWORK','submission:666|313|2','STUDENT:snumbner','666'
put 'COURSEWORK','submission:666|313|2','SUBJECT:code','313'
put 'COURSEWORK','submission:666|313|2','FILE:fnumber1','path/file-name3-1'
```

(9) How to replace a row ?

To replace a row use `put` command in the following way. Note, that the replacements happen only for the cells with one version allowed. If some cells allow for more than one version then instead of the replacement a new version of a value in a cell is created.

```
put 'COURSEWORK','student:007','STUDENT:snumber','008'
scan 'COURSEWORK'
```

(10) How to retrieve a row and a cell ?

To retrieve a row we use a row key and a command `get` in the following way.

```
get 'COURSEWORK','student:007'
```

To retrieve the contents of a cell we have to provide a full address of a cell that consists of `row key, column family: column qualifier`.

```
get 'COURSEWORK','student:007','STUDENT:snumber'
```

In the following way we can list up to 5 versions in a cell.

```
get 'COURSEWORK','student:007',{COLUMN=>'STUDENT:snumber',VERSIONS=>5}
```

(11) How to create a new version of a value in a cell ?

Assume that in one of existing submissions a student would like to submit a new version of a file `filenumber1`. A new version can be created because a column family `FILE` was created with value of parameter `VERSIONS` equal to `2` (see below). A new version is created by processing of the following `put` command.

```
alter 'COURSEWORK', {NAME=>'FILE', VERSIONS=>'2'}
```

A new version is created by processing the following `put` command.

```
put 'COURSEWORK','submission:007|313|1','FILE:fnumber1','path/file-name3-3'
```

The results can be verified with `get` command.

```
get 'COURSEWORK','submission:007|313|1',{COLUMN=>'FILE:fnumber1',VERSIONS=>2}
```

(12) How to retrieve many rows from a given column family ?

A `scan` command is equivalent to an operation of projection in the relational data model. It can be used to list the contents of entire HBase table, the contents of select column family, and the contents of cells in a given column family and in a given column qualifier (see below).

```
scan 'COURSEWORK',{COLUMN=>'STUDENT'}
scan 'COURSEWORK',{COLUMN=>'STUDENT:first-name'}
```

(13) How to list the names of created Hbase tables ?

A column list displays the names of Hbase tables created so far.

```
list
```

(14) How to drop Hbase table ?

To drop HBase table we have disable it first with a command disable

```
disable 'COURSEWORK'
```

and then we can drop it with `drop` command.

```
drop 'COURSEWORK'
```

(15) How to truncate Hbase table ?

A command `truncate` can be used to delete the contents of HBASE table without changing its internal structure. The command preserves all column families and column qualifiers within the column families.

```
truncate 'COURSEWORK'
```

(16) How to exit Hbase CLI ?

To exit HBase CLI process at `hbase main: ...:0>` prompt a command `exit`.

```
exit
```

(17) How to use Java API to process HBase tables ?

First we have to set `HBASE_CLASSPATH` variable. Process the following shell commands in a terminal window.

```
HBASE_CLASSPATH="."
for jar in `ls /usr/share/hbase/lib/*.jar`; do
HBASE_CLASSPATH=${HBASE_CLASSPATH}:$jar
done
export HBASE_CLASSPATH
```

Next use `gedit` editor and create a file `ListTables.java` with the following contents.

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.HBaseAdmin;

public class ListTables {
    public static void main(String[] args) throws Exception {
        Configuration conf = HBaseConfiguration.create();
        HBaseAdmin admin = new HBaseAdmin(conf);
// Get all the list of tables using HBaseAdmin object
      HTableDescriptor[] tableDescriptor = admin.listTables();

// Display the names of all tables
       System.out.println("HBase tables:");
       for (int i=0; i<tableDescriptor.length;i++ ){
      System.out.println(tableDescriptor[i].getNameAsString());}

    }
}
```

To compile `ListTables.java` use the following command.

```
javac -classpath $HADOOP_CLASSPATH:$HBASE_CLASSPATH ListTables.java
```

To process `ListTables` use the following command.

```
java -cp $HADOOP_CLASSPATH:$HBASE_CLASSPATH ListTables
```

(15) How to use the remaining example of HBase Java API ?

The following simple examples of HBase Java API are available on Moodle.

Creating HBase table	`CreateTable.java`
Inserting a row (cells)	`PutRow.java`
Retrieving a row (cells)	`GetRow.java`
Retrieving entire row with versions	`GetEntireRow.java`
Scanning a table	`ScanTable.java`
Deleting a row	`DelRow.java`
Dropping a table	`DropTable.java`

To compile and to process the remaining example of HBase Java API use `javac` and `java` commands in the same way as for `ListTables.java` example (see yellow highlighted fragments that should be replaced with the appropriate names).

Appendix A

hbase(main):001:0> help
HBase Shell, version 1.2.6, rUnknown, Mon May 29 02:25:32 CDT 2017
Type 'help "COMMAND"', (e.g. 'help "get"' -- the quotes are necessary) for help on a specific command.
Commands are grouped. Type 'help "COMMAND_GROUP"', (e.g. 'help "general"') for help on a command group.

COMMAND GROUPS:
Group name: general
Commands: status, table_help, version, whoami

Group name: ddl
Commands: alter, alter_async, alter_status, create, describe, disable, disable_all, drop, drop_all, enable, enable_all, exists, get_table, is_disabled, is_enabled, list, locate_region, show_filters

Group name: namespace
Commands: alter_namespace, create_namespace, describe_namespace, drop_namespace, list_namespace, list_namespace_tables

Group name: dml
Commands: append, count, delete, deleteall, get, get_counter, get_splits, incr, put, scan, truncate, truncate_preserve

Group name: tools
Commands: assign, balance_switch, balancer, balancer_enabled, catalogjanitor_enabled, catalogjanitor_run, catalogjanitor_switch, close_region, compact, compact_rs, flush, major_compact, merge_region, move, normalize, normalizer_enabled, normalizer_switch, split, trace, unassign, wal_roll, zk_dump

Group name: replication

Commands: add_peer, append_peer_tableCFs, disable_peer, disable_table_replication, enable_peer, enable_table_replication, list_peers, list_replicated_tables, remove_peer, remove_peer_tableCFs, set_peer_tableCFs, show_peer_tableCFs

Group name: snapshots
Commands: clone_snapshot, delete_all_snapshot, delete_snapshot, list_snapshots, restore_snapshot, snapshot

Group name: configuration
Commands: update_all_config, update_config

Group name: quotas
Commands: list_quotas, set_quota

Group name: security
Commands: grant, list_security_capabilities, revoke, user_permission

Group name: procedures
Commands: abort_procedure, list_procedures

Group name: visibility labels
Commands: add_labels, clear_auths, get_auths, list_labels, set_auths, set_visibility

SHELL USAGE:
Quote all names in HBase Shell such as table and column names. Commas delimit
command parameters. Type <RETURN> after entering a command to run it.
Dictionaries of configuration used in the creation and alteration of tables are
Ruby Hashes. They look like this:

{'key1' => 'value1', 'key2' => 'value2', ...}

and are opened and closed with curley-braces. Key/values are delimited by the
'=>' character combination. Usually keys are predefined constants such as
NAME, VERSIONS, COMPRESSION, etc. Constants do not need to be quoted. Type
'Object.constants' to see a (messy) list of all constants in the environment.

If you are using binary keys or values and need to enter them in the shell, use
double-quote'd hexadecimal representation. For example:

hbase> get 't1', "key\x03\x3f\xcd"
hbase> get 't1', "key\003\023\011"
hbase> put 't1', "test\xef\xff", 'f1:', "\x01\x33\x40"

The HBase shell is the (J)Ruby IRB with the above HBase-specific commands added.
For more on the HBase Shell, see http://hbase.apache.org/book.html

Appendix B

hbase(main):003:0> table_help
Help for table-reference commands.

You can either create a table via 'create' and then manipulate the table via commands like 'put', 'get', etc.
See the standard help information for how to use each of these commands.

However, as of 0.96, you can also get a reference to a table, on which you can invoke commands.
For instance, you can get create a table and keep around a reference to it via:

```
hbase> t = create 't', 'cf'
```

Or, if you have already created the table, you can get a reference to it:

```
hbase> t = get_table 't'
```

You can do things like call 'put' on the table:

```
hbase> t.put 'r', 'cf:q', 'v'
```

which puts a row 'r' with column family 'cf', qualifier 'q' and value 'v' into table t.

To read the data out, you can scan the table:

```
hbase> t.scan
```

which will read all the rows in table 't'.

Essentially, any command that takes a table name can also be done via table reference.
Other commands include things like: get, delete, deleteall,
get_all_columns, get_counter, count, incr. These functions, along with
the standard JRuby object methods are also available via tab completion.

For more information on how to use each of these commands, you can also just type:

```
hbase> t.help 'scan'
```

which will output more information on how to use that command.

You can also perform the general admin actions directly on a table; things like enable, disable,
flush and drop just by typing:

```
hbase> t.enable
hbase> t.flush
hbase> t.disable
hbase> t.drop
```

Note that after dropping a table, your reference to it becomes useless and further usage
is undefined (and not recommended).

End of exercise 6

ISIT312 Big Data Management

Session 2, 2021

Exercise 7

Using Apache Pig Latin

In this exercise you will learn how to use Apache Pig Latin for processing of the data files stored in HDFS.

Be careful when copying the Linux commands in this document to your working Terminal, because it is error-prone. Maybe you should type those commands by yourself.

Prologue

Login to your system and start VirtualBox.

When ready start a virtual machine `ISIT312-BigDataVM-07-SEP-2020`.

(1) How to start Hadoop?

Open a new Terminal window and start Hadoop in the following way.

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

When using Pig we have to additionally start History Server. Use the following command in a Terminal Window to start History Server.

```
$HADOOP_HOME/sbin/m*sh start historyserver
```

Create a text file `orders.txt` with the following contents:

```
bolt,James,200,2016,01,01
bolt,Peter,100,2017,01,30
bolt,Bob,300,2018,05,23
screw,James,20,2017,05,11
screw,Alice,55,2018,01,01
nut,Alice,23,2018,03,16
washer,James,45,2016,04,24
washer,Peter,100,2016,05,12
bolt,James,200,2018,01,05
bolt,Peter,100,2018,01,05
bolt,James,,2018,01,01
```

And load the file into HDFS into a location `/user/bigdata` .

(2) How to start Grunt Shell ?

To start Grunt Shell process the following command in a Terminal window.

```
$PIG_HOME/bin/pig
```

Apache Pig command line interface (Grunt) should open a new line with a prompt `grunt>`.

As usual, we start from asking about help. Process `help` command at `grunt>` prompt. The outcomes are listed in Appendix A at the end of this document.

It is possible to type and to process Pig commands directly at `grunt>` prompt. For few simple commands it is probably all right to do so. However, the longer sequences to commands require application of Pig script.

(3) How to process a sequence of Pig commands?

It is possible to process a sequence of Pig commands separated with `';'` as a value of `-e` switch of `pig` command. For example, it is possible to load into Pig container orders the contents of a file `orders.txt` located in HDFS and list the file with `dump` command as below.

```
$PIG_HOME/bin/pig -e "orders = load '/user/bigdata/orders.txt'
using PigStorage(','); dump orders;"
```

Of course, the best solution is to create Pig script file and process the file. Create a text file `script.pig` put into it the following lines.

```
orders = /user/bigdata/orders.txt' using PigStorage(',');
dump orders;
```

Then, save the file and process the following command at `$` prompt in a Terminal window.

```
$PIG_HOME/bin/pig -f script.pig
```

Note, that a clause `PigStorage(',')` determines a separator between the values in the rows of input data file. By default a separator is `TAB`. In our case the values are separated with a comma.

(4) How to assign the names and types to columns in Pig data container ?

Process the following command at `grunt>` prompt. A clause `as ( … )` determines the names of columns and the types of columns.

```
orders = load '/user/bigdata/orders.txt' using PigStorage(',')
as
(item:chararray,customer:chararray,quantity:int,year:int,month:i
nt,day:int);
```

To verify the results, process a command `describe`.

```
describe orders;
```

(5) How to load a map into Pig storage ?

Create a text file `keyvalue.txt` that contains the following lines.

```
[first-name#James,last-name#Bond,city#London,country#UK]
[first-name#Harry,last-name#Potter,city#Avondale,country#UK]
```

Next, load a file `keyvalue.txt` into HDFS folder `/user/bigdata`.

Then, process the following commands at `grunt>` prompt.

```
keyvalue = load '/user/bigdata/keyvalue.txt'as
(department:chararray,personal:map[]);
describe keyvalue;
dump keyvalue;
```

A data container `keyvalue` contains two maps earlier loaded into HDFS.

(6) How to load a bag with tuples into Pig storage ?

Create a text file `hobbies.txt` with the following rows.

```
James,({painting},{swimming})
Harry,({cooking})
Robin,({})
```

The file contains information about the first names of people and their hobbies.

Next load a file `hobbies.txt` into HDFS folder `/user/bigdata`.

Then process the following commands at `grunt>` prompt.

```
hobbies = load '/user/bigdata/hobbies.txt' as

(firstname:chararray,hobbies:bag{t:(hobby:chararray)});
dump hobbies;
describe hobbies;
```

Note, that a name of a bag `t` must be included in a specification of bag structure. It is also possible to nest within a bag not only sets of values but also sets of tuples.

Create a text file `nested.txt` with the following contents.

```
James,({Ferrari,xyz123}{Honda,pkr856})
Harry,({Rolls Royce,xxx666})
```

Next load a file `nested.txt` into HDFS folder `/user/bigdata`.

Then process the following commands at `grunt>` prompt.

```
nested = load '/user/bigdata/nested.txt' as
(firstname:chararray,cars:bag{t:(manufacturer:chararray, rego:chararray)});
dump nested;
describe nested;
```

(7) How to compute projections of a data container ?

We start from `describe` command to refresh our memory on a structure of o`rders` container create in step 3.

```
describe orders;
```

Assume that we would like to create a container `items` with single column `item` extracted from a container `orders`. Process the following commands at `grunt>` prompt.

```
items = foreach orders generate item;
dump items;
```

It is possible to remove duplicates with Remove duplications with distinct operation.

```
distinctitems = distinct items;
dump distinctitems;
describe distinctitems;
```

Projection on many columns

```
dates = foreach orders generate day, month, year;
dump dates;
```

(8) How to perform selections from data container ?

A `filter` command can be used to select all orders where quantity is greater than 100;

```
biggerorders = filter orders by quantity > 100;
dump biggerorders;
```

A useful example is to check the rows for nulls.

```
nulls = filter orders by quantity is null;
dump nulls;
```

(9) How to split data containers ?

Sometimes we would like to save the rows filtered from a data container into several other containers. For example, we split a container `orders` into a container `orders2018` that contains orders submitted in 2018 and a container `olderorders` that contains other orders.

```
split orders into
  orders2018 if year==2018,
  olderorders otherwise;
dump orders2018;
```

(10) How to compute an inner join ?

Create a data set `items.txt` with the following contents

```
bolt,2.23
screw,3.5
```

```
nut,1.25
washer,2.5
nail,0.4
fastener,4.1
pin,10.05
coupler,9.95
```

and load it into HDFS into a folder `/user/bigdata` .

Next, create a new data container `items` in the following way.

```
items = load '/user/bigdata/items.txt' using PigStorage(',') as
             (item:chararray,price:float);
dump items;
```

Now we implement an inner join operation to find the prices of all ordered products.

```
orders_join_items = join orders by item, items by item;
describe orders_join_items;
dump orders_join_items;
```

(11) How to implement left outer join ?

A left outer join operation joins all `items` with `orders` and includes into the result the items that have not been ordered so far extended with nulls in all columns from a container `orders`. Process the following commands.

```
leftouter = join items by item left outer, orders by item;
dump leftouter;
describe leftouter;
```

The results saved in `leftouter` data container can be used to find the names of items that have not been ordered yet, i.e. it is nothing else but implementation of *antijoin* operation. *Antijoin* operation, in contrast to join operation, finds all rows from the left argument of the operation that cannot be joined with the rows from the right argument of the operation. Process the following commands to find all items that have not been ordered yet.

```
notordered = filter leftouter by orders::item is null;
dump notordered;
```

(12) How to implement non-equi join ?

An inner join operation computed in a step 10 assumes that the rows are connect only when a value in a column `item` in a container `items` is the same as a value in a column `item` in a container `orders`. It is so called equi join. It is possible to implement a join operation that joins the rows from two containers that satisfy any condition different from equality condition. For example, find all pairs of items such that the first element in each pair has a price higher than the second element. Process the following sequence of commands and verify the results after each step.

```
newitems = load '/user/bigdata/items.txt' using PigStorage(',')
           as (item:chararray,price:float);
```

```
crossjoin = cross items, newitems;
describe crossjoin;
result = filter crossjoin by items::price > newitems::price;
```

(13) How to perform groupings and how to compute aggregation functions ?

An operation `group by` can be used to restructure a container with flat tuples into a container with tuple nested with bags. Assume that we would like to aggregate all orders on the same items into the bags and assign these bags with the ordered item. This can be achieved in the following way.

```
ordergrp = group orders by item;
```

To find an internal structure of a data container `ordergrp` process a command

```
describe ordergrp;
```

and later on a command

```
dump ordergrp;
```

Now, it is possible to count the total number of orders in each bag to get the result identical to `SELECT` with `GROUP BY` clause of SQL.

```
itemscnt = foreach ordergrp generate group, COUNT(orders.item);
dump itemsscnt;
```

(14) How to process `CUBE` and `ROLLUP` operators ?

`CUBE` operator performs grouping over all subsets of a give set of values and saves the results in a bag.

Process the following commands.

```
ordcube = cube orders by CUBE(item,name);
describe ordcube;
dump ordcube;
```

Like with `group` operation it is possible to apply an aggregation function to the results.

```
cntcube = foreach ordcube generate group, COUNT(cube.item);
dump cntcube;
```

In the same way it is possible to process `ROLLUP` operator.

```
ordrollup = cube orders by ROLLUP(item,name);
dump ordrollup;
describe ordrollup;

cntrollup = foreach ordrollup generate group, COUNT(cube.item);
dump cntrollup;
```

(15) How to unnest (flatten) data bags ?

Consider a data container `ordergrp` created in a step (13).

```
describe ordergrp;

ordergrp: {group: chararray,orders: {(item: chararray,
    name: chararray,quantity: int,year: int,month: int,day: int)}}
```

An operation `flatten` can be used to "ungroup" a nested structure. Process the following commands.

```
orderunnest = foreach ordergrp generate flatten (orders);
describe orderunnest;
dump orderunnest;
```

(16) How to save the results in HDFS ?

To save the contents of a data container created by processing Pig Latin commands use a command `store`.

```
store ordergrp into '/user/bigdata/orders' using
PigStorage(',');
store ordcube into '/user/bigdata/ordcube' using
PigStorage('|');
```

Next, in a new terminal window use the following command to list the contents of HDFS.

```
$HADOOP_HOME/bin/Hadoop fs -ls /user/bigdata/orders
$HADOOP_HOME/bin/Hadoop fs -cat /user/bigdata/orders/part-m-00000
```

(17) How to process Hadoop commands in front of `grunt>` prompt ?

To list the contents of HDFs process the following `fs` command at `grunt>` prompt.

```
fs -cat /user/bigdata/orders/part-m-00000
```

Appendix A

The outcomes of `help` command.

```
<pig latin statement>; - See the PigLatin manual for details:
http://hadoop.apache.org/pig
File system commands:
    fs <fs arguments> - Equivalent to Hadoop dfs command:
http://hadoop.apache.org/common/docs/current/hdfs_shell.html
Diagnostic commands:
    describe <alias>[::<alias] - Show the schema for the alias. Inner
aliases can be described as A::B.
    explain [-script <pigscript>] [-out <path>] [-brief] [-dot|-xml] [-
param <param_name>=<param_value>]
        [-param_file <file_name>] [<alias>] - Show the execution plan to
compute the alias or for entire script.
        -script - Explain the entire script.
```

```
        -out - Store the output into directory rather than print to
stdout.
        -brief - Don't expand nested plans (presenting a smaller graph
for overview).
        -dot - Generate the output in .dot format. Default is text
format.
        -xml - Generate the output in .xml format. Default is text
format.
        -param <param_name - See parameter substitution for details.
        -param_file <file_name> - See parameter substitution for details.
        alias - Alias to explain.
    dump <alias> - Compute the alias and writes the results to stdout.
Utility Commands:
    exec [-param <param_name>=param_value] [-param_file <file_name>]
<script> -
        Execute the script with access to grunt environment including
aliases.
        -param <param_name - See parameter substitution for details.
        -param_file <file_name> - See parameter substitution for details.
        script - Script to be executed.
    run [-param <param_name>=param_value] [-param_file <file_name>]
<script> -
        Execute the script with access to grunt environment.
        -param <param_name - See parameter substitution for details.
        -param_file <file_name> - See parameter substitution for details.
        script - Script to be executed.
    sh  <shell command> - Invoke a shell command.
    kill <job_id> - Kill the hadoop job specified by the hadoop job id.
    set <key> <value> - Provide execution parameters to Pig. Keys and
values are case sensitive.
        The following keys are supported:
        default_parallel - Script-level reduce parallelism. Basic input
size heuristics used by default.
        debug - Set debug on or off. Default is off.
        job.name - Single-quoted name for jobs. Default is
PigLatin:<script name>
        job.priority - Priority for jobs. Values: very_low, low, normal,
high, very_high. Default is normal
        stream.skippath - String that contains the path. This is used by
streaming.
        any hadoop property.
    help - Display this message.
    history [-n] - Display the list statements in cache.
        -n Hide line numbers.
    quit - Quit the grunt shell.For a good start use a command help to
get a pretty comprehensive help from HBase command Line Interface (CLI).
A complete printout of help is listed at the end of this document.
```

End of exercise 7

ISIT312 Big Data Management

Session 2, 2021

Exercise 8

Spark Fundamentals

In this exercise, you will learn basic operations in Spark's Shell and development of simple Spark applications. This will help you gain proper understanding on Spark and prepare to work at the tasks included in the Assignments.

Be careful when copying the Linux commands in this document to your working Terminal, because it is error-prone. Maybe you should type those commands by yourself.

Prologue
Login to your system and start VirtualBox.

When ready start a virtual machine `ISIT312-BigDataVM-07-SEP-2020`.

(1) How to start Hadoop?

Open a new Terminal window and start Hadoop in the following way.

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

(2) Spark-shell quick start

In the Big Data VM, Spark-shell can be started in two modes: pseudo-distributed mode and local mode. The following command starts Spark-shell in the pseudo-distributed mode.

```
$SPARK_HOME/bin/spark-shell --master yarn
```

Besides integrating with Hadoop YARN, Spark comes with its own cluster manager, e.g. standalone cluster manager, see the lecture notes for more information.

The following command starts Spark-shell in the local mode.

```
$SPARK_HOME/bin/spark-shell --master local[*]
```

The `*` symbol means using multiple threads in the VM to process a Spark job. It is recommended you use the "local" master for efficiency reasons.

Spark-shell runs on top of the Scala REPL. To quit Scala REPL, type

```
:quit
```

When Spark-shell is started, a SparkSession instance named `spark`, which is the entry points to a Spark application. To view it, simply type:

```
spark
```

With this SparkSession instance, we can create DataFrames.

```
val myRange0 = spark.range(20).toDF("number")
myRange0.show()
val myRange1 = spark.range(18).toDF("numbers")
myRange1.show()
myRange0.except(myRange1).show()
```

Open another Terminal window and upload the file `README.md` located in `$SPARK_HOME` to HDFS.

```
cd $SPARK_HOME
$HADOOP_HOME/bin/hadoop fs -put README.md README.md
```

Next, read it into a DataFrame.

```
val YOUR_HDFS_PATH="."
val textFile = spark.read.textFile("./README.md")
```

Next, count the total number of lines and display the first line in the file.

```
textFile.count()
textFile.first()
```

Next, count how many lines contain the word "Spark".

```
textFile.filter(line => line.contains("Spark")).count()
```

Next, find frequencies of each word in the document.

```
textFile.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b)
val wordCounts = textFile.flatMap(line => line.split(" ")).groupByKey(identity).count()
wordCounts.show()
wordCounts.collect()
```

(3) Run Scala script in Spark-shell

Process a command

```
:quit
```

to quit Spark-shell.

Open a plain document in Text Editor (`gedit`).

Insert the following Scala commands into the document and save it as `myScalaScript.txt` in the working directory.

```
val YOUR_HDFS_PATH="."
val textFile = spark.read.textFile("./README.md")
textFile.count()
textFile.first()
textFile.filter(line => line.contains("Spark")).count()
textFile.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b)
```

```
val wordCounts = textFile.flatMap(line => line.split(" ")).groupByKey(identity).count()
wordCounts.show()
wordCounts.collect()
```

Restart Spark-shell in the local mode (see step (2)).

Process script `myScalaScript.txt` in Spark-shell in the following way.

```
:paste myScalaScript.txt
```

Note, that only the results of `wordCounts.show()` and `wordCounts.collect()` are displayed.

(4) DataFrame/Dataset transformations and actions

Download the files `people.json`, `people.txt` and `employees.json` to your virtual machine.

Then, in Terminal window, upload the files to HDFS in the following way.

```
cd ~
$HADOOP_HOME/bin/hadoop fs -put people.json people.json
$HADOOP_HOME/bin/hadoop fs -put people.txt people.txt
$HADOOP_HOME/bin/hadoop fs -put employees.json employees.json
$HADOOP_HOME/bin/hadoop fs -ls
```

Next, display the contents of a file `people.json`.

```
$HADOOP_HOME/bin/hadoop fs -cat people.json
```

Type the following DataFrame/Dataset operations in Spark-shell window to read `people.json` file into a dataframe and to display its contents and structure.

```
val df = spark.read.json("./people.json")
df.show()
df.printSchema()
```

Next, perform few basic operations on the contents of dataframe.

```
df.select($"name", $"age" +1 ).show()
df.filter($"age" >21).show()
df.groupBy("age").count().show()
df.createOrReplaceTempView("people")
val sqlDF = spark.sql("select * from people")
sqlDF.show()
```

Does it remind you SQL ? Yes, it is "the same chicken but in a bit different gravy".

Next, create a Dataset in the following way.

```
case class Person(name: String, age: Long)
val ccDS = Seq(Person("Andy", 32)).toDS()
ccDS.show()
```

```
ccDS.select($"name").show()
```

Next, practice another way to create DataFrame. This time we shall use a text file `people.txt` already uploaded to HDFS.

```
val peopleDF =
spark.sparkContext.textFile("./people.txt").map(_.split(",")).map(
attributes=>Person(attributes(0),attributes(1).trim.toInt)).toDF()
```

And verify its contents with

```
peopleDF.show()
```

Next, we save DataFrame as DataSet.

```
case class Employee(name: String, salary: Long)
val ds = spark.read.json("./employees.json").as[Employee]
```

And verify it contents with

```
ds.show()
```

(5) Self-contained application

In the following, we implement a self-contained application and submit it as a Spark job.

Open a new document in Text Editor, input the following code and save it as a file `SimpleApp.scala`.

```
import org.apache.spark.sql.SparkSession
object SimpleApp
{
  def main(args: Array[String])
  {
    val text = "./README.md"
    val spark = SparkSession.builder.appName("Simple
Application").config("spark.master", "local[*]").getOrCreate()
    val data = spark.read.textFile(text).cache()
    val numAs = data.filter(line => line.contains("a")).count()
    val numBs = data.filter(line => line.contains("b")).count()
    println(s"Lines with a: $numAs, Lines with b: $numBs")
    spark.stop()
  }
}
```

Compile `SimpleApp.scala` by the following command in the Terminal.

```
scalac -classpath "$SPARK_HOME/jars/*" SimpleApp.scala
```

Then create a jar file in the following way.

```
jar cvf app.jar SimpleApp*.class
```

Finally, process it with Spark-shell in the following way.

```
$SPARK_HOME/bin/spark-submit --master local[*] --class SimpleApp
app.jar
```

When ready, retrieve the following line from a "jungle" of messages generated by Spark-shell.

```
Lines with a: 62, Lines with b: 30
```

(6) Shakespeare wordcount exercise

Complete the following exercise:
Objective: Count the frequent words used by William Shakespeare, but remove the known English stops words (such as “the”, “and” and “a”) in stop-words-list.csv. Return top 20 most frequent non-stop words in Shakespeare’s works.

Data sets: `shakespeare.txt`, `stop-words-list.csv`

For a good start the first few lines of code are provided below.

```
val shakes = spark.read.textFile(".../shakespeare.txt")
val swlist = spark.read.textFile(".../stop-word-list.csv")
val shakeswords = shakes.flatMap(x =>
x.split("\\W+")).map(_.toLowerCase.trim).filter(_.length>0)shake
swords.createOrReplaceTempView("shakeswords")
val stopwords = swlist.flatMap(x=>x.split(",")).map(_.trim)
stopwords.createOrReplaceTempView("stopwords")
```

The final output should be as follows:

```
result.show(20)
+-----+-----+
|value|count|
+-----+-----+
| d   | 8608|
| s   | 7264|
| thou| 5443|
| thy | 3812|
|shall| 3608|
| thee| 3104|
| o   | 3050|
| good| 2888|
| now | 2805|
| lord| 2747|
| come| 2567|
| sir | 2543|
| ll  | 2480|
| here| 2366|
| more| 2293|
| well| 2280|
| love| 2010|
| man | 1987|
| hath| 1917|
| know| 1763|
+-----+-----+
only showing top 20 rows
```

End of exercise 8

ISIT312 Big Data Management

Session 2, 2021

Exercise 9

Advanced Operations in Spark

In this exercise, you will learn some advanced operations in Spark, including integration of Spark with Hive and HBase as well as Spark Streaming.

Be careful when copying the Linux commands in this document to your working Terminal, because it is error-prone. Maybe you should type those commands by yourself.

Prologue

Login to your system and start VirtualBox.

When ready start a virtual machine `ISIT312-BigDataVM-07-SEP-2020`.

In this lab, we use *Terminal* but not Zeppelin.

(1) How to start Hadoop, Hive, HBase and Spark services?

Start the five essential Hadoop services: resourcemanager, nodemanager, namenode, datanode and historyserver in a Terminal window (see a previous lab).

First, start the Hive's metastore service. In a Terminal window, process:

```
$HIVE_HOME/bin/hive --service metastore
```

A message shows that metastore is up and running.

```
SLF4J: Actual binding is of type
[org.apache.logging.slf4j.Log4jLoggerFactory]
```

Next, start hiveserver2. Open a ***new*** Terminal window and process:

```
$HIVE_HOME/bin/hiveserver2
```

Still next, start the HBase service. Open another ***new*** Terminal window and process:

```
$HBASE_HOME/bin/start-hbase.sh
```

[Troubleshooting] Some HBase services many exit due to idle time. Use the JPS command to check. If they exit, re-start the HBase service by repeating the above step.

Finally, configurate and start Spark. Process the following in your terminal window:

```
export SPARK_CONF_DIR=$SPARK_HOME/remote-hive-metastore-conf
```

The above shell command tells Spark to use the configuration file in the folder `remote-hive-metastore-conf` in `$SPARK_HOME` instead of using the default one. Then, **in the same terminal window**, process:

```
$SPARK_HOME/bin/spark-shell --master local[*] --packages
com.hortonworks:shc-core:1.1.0-2.1-s_2.11 --repositories
http://repo.hortonworks.com/content/groups/public
```

or

```
$SPARK_HOME/bin/spark-shell --master local[*] --packages
com.hortonworks:shc-core:1.1.0-2.1-s_2.11 --repositories
https://repo.hortonworks.com/content/repositories/releases
```

The last two options in both the above commands indicate that we use a third-party connector, namely the *Hortonworks Spark-HBase-Connector (shc)* to connect Spark and HBase.

If you don't use HBase in your operations, just process:

```
$SPARK_HOME/bin/spark-shell --master local[*]
```

(2) Spark DataFrame and Dataset operations

Download a flight-dataset `2015-summary.json` and load it to HDFS.

```
$HADOOP_HOME/bin/hadoop fs -put 2015-summary.json 2015-summary.json
$HADOOP_HOME/bin/hadoop fs -ls
```

Input the following DataFrame/Dataset operations.

```
> val flightData = "<your hdfs path>/2015-summary.json"
> val df = spark.read.format("json").load(flightData)
// Or, equivalently,
> val df = spark.read.json(flightData)

> df.createOrReplaceTempView("dfTable")
> df.printSchema()

> df.select("DEST_COUNTRY_NAME", "ORIGIN_COUNTRY_NAME").show(2)

> import org.apache.spark.sql.functions.{expr, col}
> df.select(col("DEST_COUNTRY_NAME"),
    expr("ORIGIN_COUNTRY_NAME")).show(2)

> df.select(col("DEST_COUNTRY_NAME").alias("Destination"),
  expr("ORIGIN_COUNTRY_NAME").alias("Origin")).show(2)
> df.filter(col("count") < 2).show(2)

> import org.apache.spark.sql.functions.{desc, asc}
> df.orderBy(desc("count"), asc("DEST_COUNTRY_NAME")).show(2)
> df.limit(5).count()
```

```
> import org.apache.spark.sql.functions.{count,sum}

> val df1 = df.groupBy(col("DEST_COUNTRY_NAME"))
.agg(
      count(col("ORIGIN_COUNTRY_NAME")).
      alias("#CountriesWithFlightsTo"),
  sum(col("count")).alias("#AllFlightsTo")
).orderBy(desc("#AllFlightsTo"))
> df1.show(2)

> import spark.implicits._

> val person = Seq(
  (0, "Bill Chambers", 0, Seq(100)),
  (1, "Matei Zaharia", 1, Seq(500, 250, 100)),
  (2, "Michael Armbrust", 1, Seq(250, 100)))
  .toDF("id", "name", "graduate_program", "spark_status")

> val graduateProgram = Seq(
  (0, "Masters", "School of Information", "UC Berkeley"),
  (2, "Masters", "EECS", "UC Berkeley"),
  (1, "Ph.D.", "EECS", "UC Berkeley"))
  .toDF("id", "degree", "department", "school")

> val joinExpression = person
  .col("graduate_program") === graduateProgram.col("id")

> person.join(graduateProgram, joinExpression, "inner").show()
```

You can save DataFrames as different data formats (e.g., JSON, Parquet, CSV, JDBC, etc.). For example:

```
> df.write.json("<your_file_name>")
```

(3) The `:paste` command and Scala script execution

The `:paste` command allows you to paste multiple lines of Scala codes into the Spark shell. Process:

```
> :paste
// Entering paste mode (ctrl-D to finish)
```

With the `:paste` command, you can also run a script of Scala code. For example, open a plain document in Text Editor (`gedit`) and copy some lines to it, such as:

```
import spark.implicits._
val person = Seq(
  (0, "Bill Chambers", 0, Seq(100)),
  (1, "Matei Zaharia", 1, Seq(500, 250, 100)),
  (2, "Michael Armbrust", 1, Seq(250, 100)))
  .toDF("id", "name", "graduate_program", "spark_status")
```

Save the document in `Desktop` with a name `myscript.sc`. Then, process in the Spark shell:

```
> :paste /home/bigdata/Desktop/myscript.sc
```

(4) Spark and Hive

Spark has native support to Hive. In particular, Spark can use Hive's metastore service to manage data shared with multiple users currently.

We can retrieve the Hive tables (if some tables are saved in Hive):

```
> import spark.sql
> sql("show tables").show()
 // you will also see all of your Hive tables here.
```

We can also save a table to the database of Hive.

```
> import org.apache.spark.sql.{SaveMode}

> val df = spark.read.format("json").load(flightData).limit(20)
> df.write
    .mode(SaveMode.Overwrite)
    .saveAsTable("flight_data")
> sql("select * from flight_data").show()
```

You should be able to query the same table in the Hive's Beeline shell:

```
$HIVE_HOME/bin/beeline
> !connect jdbc:hive2://localhost:10000
> show tables;
// You should see the table you created before.
```

You can also develop a self-contained application to connect to Hive. See the lecture note for details.

(5) Spark and HBase

Start the HBase shell:

```
$HBASE_HOME/bin/hbase shell
```

Create a `Contacts` table with the column families `Personal` and `Office` in HBase shell:

```
> create 'Contacts', 'Personal', 'Office'
```

Load a few rows of data into HBase:

```
> put 'Contacts', '1000', 'Personal:Name', 'John Dole'
> put 'Contacts', '1000', 'Personal:Phone', '1-425-000-0001'
> put 'Contacts', '1000', 'Office:Phone', '1-425-000-0002'
> put 'Contacts', '1000', 'Office:Address', '1111 San Gabriel Dr.'
```

Then in Spark-shell, process:

```
> def catalog = s"""{
    |"table":{"namespace":"default", "name":"Contacts"},
    |"rowkey":"key",
    |"columns":{
    |"recordID":{"cf":"rowkey", "col":"key", "type":"string"},
```

```
    |"officeAddress":{"cf":"Office", "col":"Address", "type":"string"},
    |"officePhone":{"cf":"Office", "col":"Phone", "type":"string"},
    |"personalName":{"cf":"Personal", "col":"Name", "type":"string"},
    |"personalPhone":{"cf":"Personal", "col":"Phone", "type":"string"}
    |}
    |}""".stripMargin
```

The `catalog` function defined above corresponds to the structure of the HBbase table `Contacts` we have created. We can make HBase queries by Spark's Structured API:

```
> import org.apache.spark.sql.execution.datasources.hbase._
> def withCatalog(cat: String) = {
  spark.sqlContext
    .read
    .options(Map(HBaseTableCatalog.tableCatalog->cat))
    .format(
    "org.apache.spark.sql.execution.datasources.hbase"
    )
    .load()
  }

// Query
> def dfHbase = withCatalog(catalog)
> dfHbase.show()
```

Besides reading, we also can write data into HBase tables. To this end, we first create a Dataset:

```
> case class ContactRecord(
    recordID: String, // rowkey
    officeAddress: String,
    officePhone: String,
    personalName: String,
    personalPhone: String )

> import spark.implicits._
  // Insert records
> val newContact1 = ContactRecord("16891", "40 Ellis St.",
     "674-555-0110", "John Jackson","230-555-0194")
> val newContact2 = ContactRecord("2234", "8 Church St.",
     "234-325-23", "Ale Kole","")
> val newDataDS = Seq(newContact1, newContact2).toDS()
```

We can save the Dataset object to HBase:

```
> newDataDS.write
  .options(Map(
   HBaseTableCatalog.tableCatalog -> catalog,
    HBaseTableCatalog.newTable -> "5" //this param must be set
  )).format(
   "org.apache.spark.sql.execution.datasources.hbase"
   ).save()
> dfHbase.show()
```

Stop the HBase service when you finish with it:

```
$HBASE_HOME/bin/stop-hbase.sh
```

(5) Spark Structured Streaming

In the following, we stimulate a streaming version of the wordcount application. We use the Unix NetCat utility to generate the input strings. Open a new terminal window and process:

```
nc -lk 9999
```

Input the following code in Spark-shell, which tells Spark to listen to Port 9999 in the localhost.

```
> val lines = spark.readStream
  .format("socket") // socket source
  .option("host", "localhost") // listen to the localhost
  .option("port", 9999) // and port 9999
  .load()
```

Then, develop the wordcount application and start the streaming.

```
> import spark.implicits._
> val words = lines.as[String].flatMap(_.split(" "))
> val wordCounts = words.groupBy("value").count()
> val query = wordCounts.writeStream
  .outputMode("complete") // accumulate counting result of the
stream
  .format("console") // use the console as the sink
  .start()
```

There is no output in Spark-shell yet, because there is no input generated on Port 9999. Input some words in your NetCat terminal window, such as:

```
apache spark
apache hadoop
spark streaming
...
```

You should see return batches in Spark-shell.

End of exercise 9