CS44800 Project 5
Big Data Management Using Hadoop/Spark
Fall 2019

Due: **December 4, 2019, 11:59PM**
Total Points: **5 points**

Learning Objectives

1. Use Hadoop Distributed Filesystem (HDFS) to store input and output data files
2. Use Spark to perform basic query processing over data files stored in HDFS

Project Description

In this project, your goal is to use Hadoop and Spark to perform some data processing tasks. You are going to implement a given set of queries using Spark as your data processing framework. With this project, you will get hands-on experience with writing Spark applications and execute them on a real cluster.

Scholar Cluster

The scholar cluster provides an environment to perform Big Data processing using Hadoop and Spark frameworks. There are two ways to work with the Scholar cluster. The easiest way is to use the web-based Remote Desktop from any web browser using the following link:

https://desktop.scholar.rcac.purdue.edu/

Alternatively, you can use SSH to login to the cluster; e.g., replace `<username>` with your Purdue username, and run the following command from a terminal.

```
ssh <username>@scholar.rcac.purdue.edu
```

Note: every registered student in the class should have access to Scholar. If you have difficulties accessing Scholar, contact the TAs to resolve your issue. Try to contact us early to resolve your issue with the scholar cluster.

More information about Scholar can be found here:

https://www.rcac.purdue.edu/compute/scholar

Dataset

For this project, we will use the **MovieLens** dataset. More information about this dataset is found here and you should familiarize yourself with the data:

http://files.grouplens.org/datasets/movielens/ml-1m-README.txt

Before you start running queries, your data files need to be uploaded into HDFS. Download the 1M dataset (it has been pre-processed to have a valid UTF8 encoding to avoid any encoding related issues) from the Scholar cluster: `/home/bferdous/cs448` and the file name is `p5-data.zip`.

The table schema is provided below but more information is available in the README file link above. Make sure to familiarize yourself with the data encoding by reading the README file that comes with the data files.

Table Schema:

File name	Schema
users.dat	(UserID::Gender::Age::Occupation::Zipcode)
movies.dat	(MovieID::Title::Genres)
ratings.dat	(UserID::MovieID::Rating::Timestamp)

Part 1: Uploading data files to HDFS

Login into the Scholar cluster and download extract the data files into a temporary directory (say, "`~/tmp`") in your home directory. Create a subdirectory to store the data files on HDFS using the following command:

```
  hdfs dfs -mkdir /user/<username>/input
```

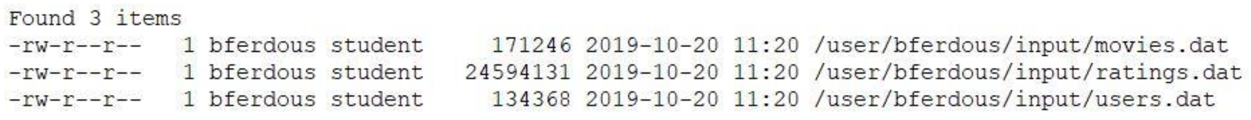
To upload data files into HDFS, use the following command:

```
  hdfs dfs -put ./tmp/* /user/<username>/input
```

Now, you should have your data files in HDFS. Verify that by listing the files in the "input" directory, using the following command:

```
  hdfs dfs -ls /user/<username>/input
```

The output of the above command should be similar to the following:



```
Found 3 items
-rw-r--r--   1 bferdous student     171246 2019-10-20 11:20 /user/bferdous/input/movies.dat
-rw-r--r--   1 bferdous student   24594131 2019-10-20 11:20 /user/bferdous/input/ratings.dat
-rw-r--r--   1 bferdous student     134368 2019-10-20 11:20 /user/bferdous/input/users.dat
```

Part 2: Warm-up Exercise

In this exercise, you will perform the following tasks:

1. Download skeleton code from the course website.
2. Verify that the environment settings are valid.
3. Use Maven to prepare your Spark application to submit to the cluster
4. Submit your application to the cluster using some example code in the skeleton Java project.

The skeleton code is available from the course website. Download and extract the Maven-based Java project into a directory of your choice.

Maven Configuration

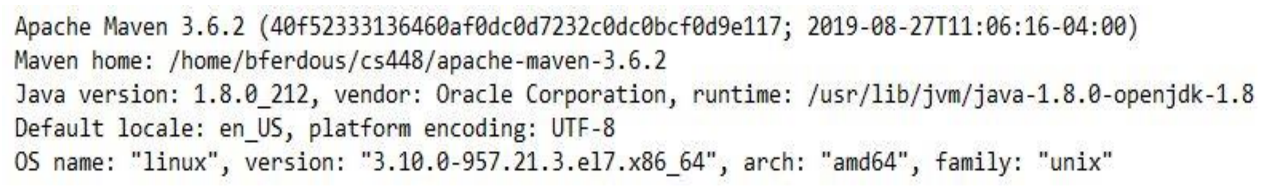
For this project, we use Maven to build Spark applications. Use the following command to activate Maven in your environment.

```
export PATH=/home/bferdous/cs448/apache-maven-3.6.2/bin:$PATH
```

After that, use following command to check Maven setup.

```
mvn --version
```

You should see an output like the following:



```
Apache Maven 3.6.2 (40f52333136460af0dc0d7232c0dc0bcf0d9e117; 2019-08-27T11:06:16-04:00)
Maven home: /home/bferdous/cs448/apache-maven-3.6.2
Java version: 1.8.0_212, vendor: Oracle Corporation, runtime: /usr/lib/jvm/java-1.8.0-openjdk-1.8
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "3.10.0-957.21.3.el7.x86_64", arch: "amd64", family: "unix"
```

Recall that Maven is a build tool which you use to compile your code. Re-running Maven to recompile your code is necessary to ensure that new changes in your code take effect after submitting the application to the Spark cluster.

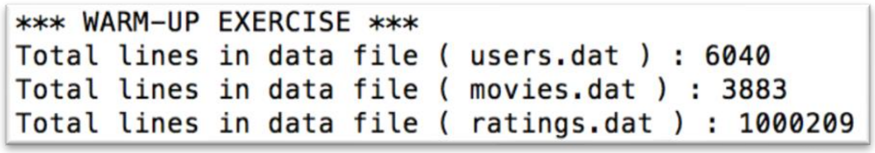
Now, try to build the current skeleton code. The skeleton code comes with a feature to check if your development environment is configured correctly and it is called the Warm-up mode. First, build and package your Spark application using the following command:

```
mvn clean package
```

Note that clean deletes all files related to a previous build process invocation. The purpose of the above command is to package your Spark application into a JAR file (which is basically a container of all your Java code). The JAR file is located in the `./target` directory of your maven-based project. This package is submitted to the Spark cluster using the following command:

```
spark-submit --class cs448.App target/cs448p5-1.0-SNAPSHOT.jar
-i "/user/<username>/input" -warmup
```

The `-i` argument is for specifying the input directory on HDFS. Notice the `-warmup` argument, which causes the application to run the built-in warm-up exercise. **Make sure to omit the –`warmup` argument when running your code. Otherwise, you will always run in the warm-up exercise.** The warm-up will read each of the data files, count the number of lines, and print the number. If you have completed Part 1 successfully, you should get exactly the following output.



```
*** WARM-UP EXERCISE ***
Total lines in data file ( users.dat ) : 6040
Total lines in data file ( movies.dat ) : 3883
Total lines in data file ( ratings.dat ) : 1000209
```

Part 3: Spark Application

Now, you are ready for your Spark application. **Your task is to implement two Spark applications of the following any two queries (out of four queries) described in this section. Each query must be implemented within a separate method. For the implementation of the query, you can choose any of the following APIs SparkSQL, SparkRDD or DataFrame.**

Query 1: Find all movie titles that are rated greater than or equal to `conf.q1Rating` and rated by users with occupation code equal to `conf.q1Occupation`.

/ example output **/**
```
Problem Child (1990)
```

Query 2: Find the movie titles that have not been rated with a rating equal to `Conf.q2Rating` by a user with an age of `conf.q2Age` or younger. The results should have no duplicate movie title.

/ example output: **/**
```
Mondo (1996)
```

Query 3: Find UserId of users that have rated with a rating equal to `conf.q3Rating` or lower more than `conf.q3Movies` movies that have `conf.q3Genre` as one of its genres.

/ example output: **/**
```
1339
```

Query 4: Find all movie titles and the number of ratings from a user having an age group equal to`conf.q4Age`.

/ example output: **/**
```
Reality Bites (1994)::114
```

For example, in the first query, you basically need to return distinct movie titles from the database that have a rating greater or equal to a supplied argument that is rated by a user having the given occupation code. Once you have implemented the above query inside the `runSparkApp1` method of the Project5 class, you can run the following command to execute the above query and store the output:

```
spark-submit --class cs448.App target/cs448p5-1.0-SNAPSHOT.jar -i
"/user/<username>/input" -o "/user/<username>/output" -q 1,12,3
```

Notice the additional parameters for specifying the query parameters. The argument `-q` has a comma-separated value which is parsed and passed to the Spark application to select which query is invoked and the respective arguments to use for the query. For example, `-q 1,12,3` indicates Query 1 is to be invoked with `u.occupation = 12`, and `r.rating >= 3`.

The provided skeleton code performs the parsing of the command line arguments. Therefore, you can focus on writing the logic that implements the queries. **You shall implement each query logic inside its respective method in Project5.java called `runSparkApp<queryNumber> (App.Conf conf).`**

Here `<queryNumber>` refers to the query associated with Spark application. For example, for the Spark application for query 1, place your logic inside `runSparkApp1` method of the `Project5` class.

An instance of `App.Conf` class is passed to each method. This instance contains all the application configuration and the query parameters parsed from the command line arguments. For completing Part 3, you have the following public data members:

`conf.inPath` : path to the directory on HDFS containing data input files

`conf.outPath` : path to the directory on HDFS used to save data output files

`conf.usersFName` : filename for the users table

`conf.moviesFName` : file name for the movies table

`conf.ratingsFName` : file name for the ratings table

Note that for these queries (in Part 3), you have the parameters available in the following format `conf.q<QueryNumber><ParameterName>` and they hold the values to be used in the queries. For example, for Query 1, you have the following data members available.

`conf.q1Occupation` : holds the integer value passed as the query parameter for occupation code.

`conf.q1Rating` : holds the integer value passed as the query parameter for rating.

Hint: You can utilize `resolveUri` method from the `CS448Utils` class to build a full path for the input or the output file.

For example, `CS448Utils.resolveUri(conf.inPath,conf.usersFName)` returns the full path to the user table input data file. See the warmup code example for more details.

Testing Your Code

The code base provided to you includes a simple testing framework that helps you test your distributed Spark application. To test your implementation, simply append -test to your Spark submission command. For example, the following command will run your implemented application and test it against a predefined test-case that uses Spark.

```
spark-submit --class cs448.App target/cs448p5-1.0-SNAPSHOT.jar -i
"/user/<username>/input" -o "/user/<username>/output" -q 1,12,3 -test
```

The output of each query should be saved in HDFS since it is used for the testing framework to validate your implementation. The result of each query should be stored in a separate directory using the following convention `query-<number>`. Here `<number>` is the query number. For example, assuming that the argument passed for the output directory `-o` is: "`/user/<username>/output`" (`<username>` is your username), then the output directory name for Query 1 is query-1, and the full path on HDFS is:

```
/user/<username>/output/query-1
```

Store your output as text files with one tuple per line and use ':: ' as field separator, if needed (i.e., similar to input data formatting). You need to consult Spark's documentation on how to store data from Spark as text files.

Suggested Strategy

We suggest the following strategy to tackle this project efficiently:

1. Make sure that you can access and work with the Scholar cluster.
2. Familiarize yourself with the data by reading the README file of the data files.
3. Run the **warm-up** exercise and make sure that you can submit and execute Spark applications.
4. Since you only need to submit 2 queries, carefully select the queries you want to implement.
5. Start solving the queries using Spark SQL. Test your code.
6. If possible, try implementing the queries using the other two APIs: DataFrame and SparkRDD.

What to Turn in

1. The **Project5.java** file, with the implementation of 2 queries from Part 3. If you need to create helper classes or any additional class, create those as inner lasses inside this file. No additional file should be submitted.

The file must be submitted on Blackboard.

FAQ

How to run testcases from my own HDFS directory?

You need the test-case results into your own HDFS directory. One way is by copying from the original test directory:

```
hdfs dfs -cp /user/bferdous/test  /user/<username>/test
```

Another way is to upload the files. The full path to testcases ZIP file on Scholar's own file systems (Not HDFS) is:

```
/home/bferdous/cs448/testcases.zip
```

You can also directly upload the files from:

```
/home/bferdous/cs448/testcases
```

The next step is to modify line#6 in `CS448Constants.java` to point to your own test directory on HDFS. For example, change it from (replace username with your own username):

```
public static String TESTCASE_PATH = "/user/bferdous/test/query-%d";
```

to:

```
public static String TESTCASE_PATH = "/user/<username>/test/query-%d";
```