CS44800 Project 4
Big Data Management Using Hadoop/Spark
Spring 2019

Due: **April 22, 2019, 11:59PM**
Total Points: **7 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 Java 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 from the course website (it has been pre-processed to have a valid UTF8 encoding to avoid any encoding related issues). The data files are available on the Scholar cluster at: `/home/tqadah/cs448_public` and the file name is `p4-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
```

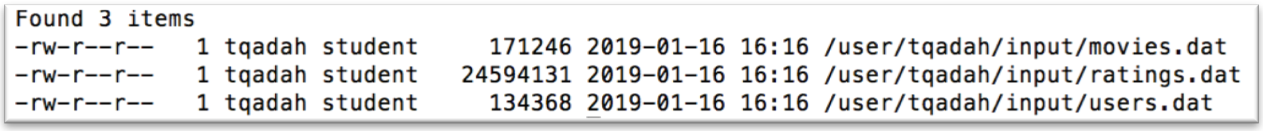
To upload data files into HDFS, use the following command:

```
$ 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 tqadah student     171246 2019-01-16 16:16 /user/tqadah/input/movies.dat
-rw-r--r--   1 tqadah student   24594131 2019-01-16 16:16 /user/tqadah/input/ratings.dat
-rw-r--r--   1 tqadah student     134368 2019-01-16 16:16 /user/tqadah/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

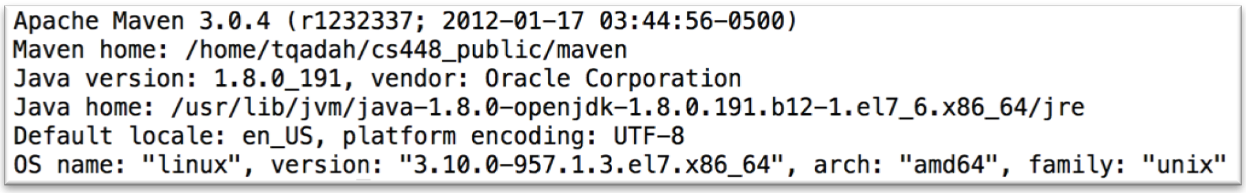
For this project, we use Maven to build Spark applications. Use the following command to activate Maven in your environment.

```
$ export PATH=/home/tqadah/cs448_public/maven/bin:$PATH
```

After that, use following command to check Maven setup.

```
$ mvn --version
```

You should see an output similar to the following:



```
Apache Maven 3.0.4 (r1232337; 2012-01-17 03:44:56-0500)
Maven home: /home/tqadah/cs448_public/maven
Java version: 1.8.0_191, vendor: Oracle Corporation
Java home: /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.191.b12-1.el7_6.x86_64/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "3.10.0-957.1.3.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.

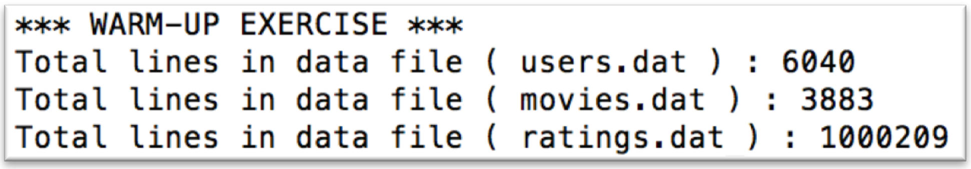
Now, try to build and current skeleton code. The skeleton code comes with a feature to check if your development environment is configured correctly and it is called the Warm-up mode. First, build and package your Spark application using the following command:

```
$ mvn clean package
```

Note that `clean` deletes all files related to a previous build process invocation. You can omit that word if want to. The purpose of the above command is to package your Spark application into a JAR file (which is basically 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/p4-1.0-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: Your First Spark Application

Now, you are ready for your first Spark application. In this section, we will guide you through your first application. Your task in this part of the project is to implement the following query over the data files using Spark. The SQL specification is given below.

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`.

```
SELECT DISTINCT m.title
FROM Movie m, Rating r, User u
WHERE
     m.movieId = r.movieId
     AND r.userId = u.userId
     AND u.occupation = ?
     AND r.rating >= ?

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

You basically need to return distinct movie titles from the database that has a rating grater or equal to a supplied argument that is rated by a user having the given occupation code.

For example, once you have implemented the above query inside the runSparkApp1 method of the Project4 class. You can run the following command to execute the above query and store the output:

```
$ spark-submit --class cs448.App target/p4-1.0-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. For example, `-q 1,12,3` indicates Query 1 is to be invoked with `u.occupation = 12`, and `r.rating >= 3` (i.e., respectively replacing the question marks in the above SQL query specification with the passed arguments at run-time).

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 `Projec4.java` called `runSparkApp<queryNumber>(App.Conf conf)`

Here `<queryNumber>` refers to the query associated with Spark application. For your first Spark application, place your logic inside `runSparkApp1` method of the `Project4` 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 and Part 4, 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:` file name for the users table

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

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

Note that for this query and other queries (in Part 4), 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.

Part 4: Additional Queries

Furthermore, we ask that you implement the following additional three queries as three Spark applications where each is contained within a separate method. Therefore, you need a create a separate Spark session for each query.

For each query, we demonstrate how a single line of output from your Spark application output file is formatted.You should turn in your code and results for the query above and the three queries below. 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 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. Below are the specifications of the three additional queries along with an example output of one line in the output file.

Query 2: Find zip-codes of users having **either** `conf.q2Occupation1` **or** `conf.q2Occupation2` as their occupation. The results should have no duplicate zip-codes.

```
/** example output: **/
01453
```

Query 3: Find the MovieID of all movies that have a rating equal to `conf.q3Rating` and are rated by users with occupation `conf.q3Occupation`. The results should have no duplicate MovieIDs.

```
/** example output: **/
1014
```

Query 4: Find all movie titles and their average ratings that are rated by users having age group equal to `conf.q4Age`.

```
/** example output: **/
Extreme Measures (1996)::2.9
```

Note: Query 4 involves taking averages. Depending on how you compute the averages, there may be slight (plus or minus 0.1) discrepancies between your results and our results. No points will be taken off for these rounding differences.

Testing Your Code

The codebase 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/p4-1.0-SNAPSHOT.jar -i
"/user/<username>/input" -o "/user/<username>/output" -q 1,12,3
-test
```

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 Spark by going over the programming guide.
3. Familiarize yourself with the data by reading the README file of the data files.

4. Run the warm-up exercise and make sure that you can submit and execute Spark applications.
5. Start solving queries using the SparkSQL/DataFrame API. Test your code.
6. Re-write two of the queries using the SparkRDD API. Test your code.

What to Turn in

A zipped file with the following naming convention: CS448p4_<purdue-username>.zip and contains the following:

1. The source code for your Spark applications implementing the 4 queries (Project4.java).
2. A README file that includes answers to only two of the following open-ended questions:
   a. Give an example of a novel application that can be built using Spark. Discuss why that application is important and why Spark is a good framework for the application that you are proposing.
   b. Choose an existing feature of Spark that you would have liked to practice with more. Discuss why the feature is important and why you would like to learn more about it
   c. Propose a feature that currently does not exist in Spark and discuss why such a feature is important and how it can improve the Spark as a framework.

The zipped file should be submitted on Blackboard.

Grading

Refer to the rubric below to understand how this project assignment is graded. Note that, you are free to use any suitable Spark API (i.e., SparkRDD or SparkSQL/DataFrame) but at least two queries (out of the total of 4) must be implemented using SparkRDD API. Otherwise, points will be deducted from your grade.

CRITERIA	**PERFECT (1 POINT)**	**ACCEPTABLE (0.5 POINTS)**	**UNSATISFACTORY (0 POINTS)**
CODE STATUS	Submitted compiles and execute successfully with the Spark cluster.	Submitted code compiles but there are some issues at run-time.	Submitted code does not compile .
QUERY 1 IMPLEMENTATION	Query is implemented correctly. Output is stored in the correct directory and in the correct format. Passes all test cases.	Output is stored correctly. Passes some of the test cases.	Query is not implemented correctly or no implementation is provided.
QUERY 2 IMPLEMENTATION	Query is implemented correctly. Output is stored in the correct directory and in the correct format. Passes all test cases.	Output is stored correctly. Passes some of the test cases.	Query is not implemented correctly or no implementation is provided.
QUERY 3 IMPLEMENTATION	Query is implemented correctly. Output is stored in the correct directory and in the correct format. Passes all test cases.	Output is stored correctly. Passes some of the test cases.	Query is not implemented correctly or no implementation is provided.

QUERY 4 IMPLEMENTATION	Query is implemented correctly. Output is stored in the correct directory and in the correct format. Passes all test cases.	Output is stored correctly. Passes some of the test cases	Query is not implemented correctly or no implementation is provided.
USAGE OF SPARK RDD API	At least two queries are implemented with Spark RDD API.	One query is implemented with Spark RDD API.	No query is implemented with Spark RDD API.
README FILE	At least two questions are addressed concisely and clearly.	Provided answers are not clear or does not answer the questions completely or less than two questions are addressed properly.	No README file is provided or no answers are provided at all.

References

https://spark.apache.org/docs/2.2.0/