Purdue University
Fall 2016
CS 448: Introduction to Relational Database Systems
Project 2: SQL via MapReduce

/**********
* Overview
**********/

This project will demonstrate how to convert a SQL query into a sequence of MapReduce jobs which can be run on files stored in HDFS.

/**********
* Environment
**********/

We will use Purdue's OpenStack cluster for this assignment. The master node for this cluster is openstack-vm-11-251.rcac.purdue.edu. You can SSH into the master node with credentials username:[PurdueID]_ostack, password:[PurdueID]_ostackpwd. (By PurdueID, we mean the name that you use to login to MyPurdue, Blackboard, etc.) If you are connecting from off campus, you will need to set up a VPN connection first. Instructions for setting up a VPN to Purdue's network are here: https://www.itap.purdue.edu/connections/vpn/ .

Be sure to change your password after you log in.

```
$ passwd
```

To see a list of all the nodes in the cluster, run

```
$ cat /etc/hosts
```

CAUTION: This cluster is temporary. It will be wiped after the project is graded. If you have any code or results that you wish to save, move them to permanent storage on another system.

NOTE: This cluster does not mount the CS Department's NFS shared file system, so your CS home directory is not available.

We are running the Cloudera Cluster Manager on the cluster. You can access the Cluster Manager web UI at http://openstack-vm-11-251.rcac.purdue.edu:7180 . Log in with username:student, password:studentpwd. Again, if you are off campus, you must set up a VPN for this to work. You can use the web UI to explore HDFS, check on the status of jobs, check the cluster workload, etc. Note that you will not need to access the web UI to complete the project: The UI is available simply for you to explore.

```
/**********
* Basic commands for HDFS and MapReduce
**********/
```

Before you begin the project, you should know how to:

- Move files from the cluster master node to HDFS and back.
- Compile and run a MapReduce job.

If you have used Hadoop before, you may be familiar with these concepts. If you have not, follow the steps in this section. This section will not be graded.

After logging on to the cluster master node with SSH, populate a simple text file:

```
$ printf "aaa\nbbb\nccc\nddd\naaa\nbbb\nccc\nddd\n" > tmp.txt
$ cat tmp.txt
```

Create a directory in your personal HDFS directory to store the file:

```
$ hdfs dfs -mkdir /user/[PurdueID]_ostack/in
```

Copy the text file from the master node to the new directory:

```
$ hdfs dfs -put ./tmp.txt /user/[PurdueID]_ostack/in
```

List the directory contents:

```
$ hdfs dfs -ls /user/[PurdueID]_ostack/in
```

Now that we have loaded a file into HDFS, we would like to run a MapReduce job over it. Copy the file Select.java from your personal machine to the cluster master node. On the master node, navigate to the directory which holds Select.java and run the following commands:

```
$ mkdir select
$ CP=$(hadoop classpath)
$ javac -classpath $CP -d select/ Select.java
$ jar -cvf select.jar -C select .
$ hadoop jar select.jar org.myorg.Select /user/[PurdueID]_ostack/in
/user/[PurdueID]_ostack/out
```

Notice the last two arguments to the final command. Our MapReduce job will take these arguments to be the source directory (which already exists) and the destination directory (which will be created by our MapReduce job).

Finally, check the output:

```
$ hdfs dfs -ls /user/[PurdueID]_ostack/out
$ hdfs dfs -cat /user/[PurdueID]_ostack/out/*
```

Move the output from HDFS back to your directory on the master node:

```
$ hdfs dfs -getmerge /user/[PurdueID]_ostack/out ./output.txt
$ cat output.txt
```

Finally, remove the output directory so that you can run your job again. (Hadoop will complain if you try to write to an HDFS directory that already exists.)

```
$ hdfs dfs -rm -r /user/[PurdueID]_ostack/out
```

/**********
* Dataset
**********/

Movie Lens 1M dataset, taken from GroupLens Research:
http://grouplens.org/datasets/movielens/

Tables:

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

CAUTION: There are several datasets listed on the Movie Lens webpage. Be sure you get the 1M dataset.

NOTE: Some of the titles in movies.dat contain characters that are not UTF-8. These characters may compromise your results. So, as is often the case in the real world, we must put our data through a "cleaning" phase before we query it. Use the following bash command to remove all non-UTF-8 characters from the movies.dat file before you load it into HDFS:

```
$ iconv -f utf-8 -t utf-8 -c movies.dat > movies_utf8.dat
```

/**********
* Example
**********/

We will provide you with the code to complete an example query on MapReduce. Below we show the query and a single line from the final output.

Example Query (Query 1):

```
SELECT DISTINCT m.title
FROM Movies m, Ratings r, Users u
WHERE
     m.MovieID = r.MovieID
     AND r.UserID = u.UserID
     AND u.Occupation = 12 --(programmer)
     AND r.Rating >= 3
;
/* example output: */
Vertigo (1958)  [tab]  MoviesUsersRatingsTable
```

We will perform this query in six stages. Each stage will consist of one MapReduce job. However, some jobs will not have a reduce phase, so they will simply output the results of the map phase.

1. Perform a selection operation on the Users table: we will use a map function to output only those pairs where Occupation equals 12.

2. Perform a selection operation on the Ratings table: use a map function to output only those pairs where Rating >= 3. "Bring along" MovieID.

3. Take the outputs from Step 1 and Step 2 and join them on UserID. Whenever we find a match, output the MovieID from the Ratings table.

4. Perform a selection operation on the Movies table: we will not eliminate any rows, but this step will transform the Movies table into a format that is easier to use in a join.

5. Take the outputs from Step 3 and Step 4 and join them on UserID. Output the movie titles.

6. Take the output from Step 5 and output only the unique movie titles.

Here we show the six jobs written in another way. For each job, we show the format of the input file and the format of the output file.

Job1 (select):

```
Input                                 -->   Key        Value
[UserID]::[M/F]::[Age]::[Occup]::[Zip]           [UserID]   UsersTable
```

Job2 (select):

```
Input                                 -->   Key        Value
[UserID]::[MovieID]::[Rating]::[Time]       [UserID]   RatingsTable::[MovieID]
```

Job3 (join):

```
Input                                 -->   Key        Value
```

```
[UserID]   UsersTable                       [MovieID]  UsersRatingsTable
[UserID]   RatingsTable::[MovieID]

Job4 (select):
Input                                 -->   Key        Value
[MovieID]::[Title]::[Genre]                      [MovieID]  MoviesTable::[Title]

Job5 (join):
Input                                 -->   Key        Value
[MovieID]  UsersRatingsTable                [Title]    MoviesUsersRatingsTable
[MovieID]  MoviesTable::[Title]

Job6 (distinct):
Input                                 -->   Key        Value
[Title]    MoviesUsersRatingsTable          [Title]    MoviesUsersRatingsTable
```

Lastly, we show some made-up example input and output below:
(Note that dates have been removed from the movie titles for brevity.)

Job	**Mapper Input**	**Reducer Input**	**Output**
Job1	Users 44::M::20::12::47906 45::F::22::10::47906 46::M::19::12::47906 47::F::25::18::47906 ...	[no reduce phase]	Out1 44 UsersTable 46 UsersTable ...
Job2	Ratings 44::101::4::[time] 44::103::5::[time] 44::107::2::[time] 45::108::5::[time] 46::115::3::[time] ...	[no reduce phase]	Out2 44 RatingsTable::101 44 RatingsTable::103 45 RatingsTable::108 46 RatingsTable::115 ...
Job3	Out1 + Out2 44 UsersTable 46 UsersTable 44 RatingsTable::101 44 RatingsTable::103 45 RatingsTable::108 46 RatingsTable::115 ...	[reduce() 1] 44 UsersTable 44 RatingsTable::101 44 RatingsTable::103 [reduce() 2] 45 RatingsTable::108 [reduce() 3] 46 UsersTable 46 RatingsTable::115 ...	Out3 101 UsersRatingsTable 103 UsersRatingsTable 115 UsersRatingsTable ...
Job4	Movies 101::Taken::Action 102::Heat::Action 103::Matrix::Action ...	[no reduce phase]	Out4 101 MoviesTable::Taken 102 MoviesTable::Heat 103 MoviesTable::Matrix ...
Job5	Out3 + Out4 101 UsersRatingsTable 103 UsersRatingsTable 115 UsersRatingsTable 101 MoviesTable::Taken 102 MoviesTable::Heat 103 MoviesTable::Matrix ...	[reduce() 1] 101 UsersRatingsTable 101 MoviesTable::Taken [reduce() 2] 102 MoviesTable::Heat [reduce() 3] 103 UsersRatingsTable 103 MoviesTable::Matrix ...	Out5 Taken Movies...Table Matrix Movies...Table ...
Job6	Out5 Taken Movies...Table Matrix Movies...Table Taken Movies...Table Matrix Movies...Table ...	[reduce() 1] Taken Movies...Table Taken Movies...Table [reduce() 2] Matrix Movies...Table Matrix Movies...Table ...	Out6 Taken Movies...Table Matrix Movies...Table ...

```
/**********
* Notes
**********/
```

The following discussion will be easier to understand if you first execute the example code and observe the output at each stage.

Let us consider how to do the join in Job 3. Taking a closer look at this join will help us to see why we have formatted the output of Jobs 1 and 2 as we have.

Recall how to do a standard nested loops join. We require an outer table and an inner table. Let us take Users to be the outer table and Ratings to be the inner table. For each row of the outer table, we loop over each row of the inner table and check whether the join key of the outer row (Users.UserID) matches the join key of the inner row (Ratings.UserID). For each match, we produce an output row.

In MapReduce, the data is distributed and very large, so we cannot execute nested loops using two tables on a single machine. What we can do instead is identify a subset of rows from Users and subset of rows from Ratings that match on the join column and collect these two subsets on a single machine to join. Of course, we should carry out this process for every unique join column key.

Recall that the shuffle phase of MapReduce groups key-value pairs by key. Since we want rows from Users to end up with rows from Ratings that have the same UserID, we will choose UserID to be our key, and any remaining information to be the value.

When the shuffle phase completes and the reduce phase begins, each reduce() call will have access to a set of matching key-value pairs from both tables. All of the key-value pairs that are fed into a single reduce() will match on UserID, so to perform a nested loop join we only need to separate the input rows into their respective tables and then loop over them.

Notice that the Users-Ratings relationship is one-to-many. That is, a single user can give multiple ratings, but a single rating never belongs to more than one user. This means that when we process matching rows from both tables in a reduce() call, reduce() will only see one row from table Users. In the code, we still use double nested loops to preserve generality.

```
/**********
* Task
**********/
```

We ask that you implement the following three queries as sequences of Hadoop jobs. For each query, we demonstrate what a single line of output from your final job should look like. After completing each query, use $ hdfs dfs -getmerge to consolidate the results of your last job into a single file on the cluster master node. You should turn in your code and results for the example query above and the three queries below.

```
/* Query 2: */
SELECT DISTINCT zip
FROM users u
WHERE
     u.occupation = 15
     OR u.occupation = 17
;
/* example output: */
01453  [tab]  UsersTable
```

```
/* Query 3: */
SELECT DISTINCT r.movieid
FROM ratings r, users u
WHERE
     r.userid = u.userid
     AND r.rating = 5
     AND u.occupation = 6
;
/* example output: */
1014  [tab]  UsersRatingsTable
```

```
/* Query 4: */
SELECT m.title, ROUND(AVG(r.rating), 1)
FROM users u, ratings r, movies m
WHERE
     u.userid = r.userid
     AND r.movieid = m.movieid
     AND u.age = 35
GROUP BY r.movieid
;
/* example output: */
Extreme Measures (1996)::2.9  [tab]  MoviesUsersRatingsTable
```

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.

NOTE: When you collect your results with -getmerge, the rows of your output may not be in order. This is not a problem.

/**********
* Deliverables
**********/

Please follow these instructions exactly. Take care to name your files as indicated.

Create a directory with your PurdueID as the directory name. In this directory, create four subdirectories, one for each query. Within each query directory, place the code for your MapReduce jobs as well as a single file with your merged query results (retrieved from HDFS).

```
[PurdueID]/
      q1/
            MR1.java
            ...
            MR6.java
            q1_results.txt
      q2/
            MR1.java
            ...
            q2_results.txt
      ...
```

Navigate to the parent directory of your [username] directory and run this command:

```
$ tar -cvf [PurdueID].tar [PurdueID]
```

Submit the file [PurdueID].tar using turnin:

```
$ turnin -c cs448 -p proj2 [PurdueID].tar
```