Overview:

This project will demonstrate how to convert a SQL query into a series of MapReduce jobs that can be run on distributed table files. We will walk through an example query and then present the project requirements. Before you begin, make sure that you are able to complete the Hadoop walkthrough available through RCAC:

https://www.rcac.purdue.edu/compute/hathi/guide/#run_hadoop

Completing this walkthrough will ensure that you have the basic knowledge necessary to load files into HDFS and run a MapReduce job on the Hathi cluster. It will also give you a chance to configure your work environment on Hathi.

Dataset:

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

Tables:

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

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 > tmp.txt; mv tmp.txt movies.dat

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
;

Approach:

We will perform this query in six stages. Each stage will consist of one MapReduce job. However, some jobs will not use a reduce function, so they will simply output the results of the map step.
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 print out only the unique movie titles.

---

Job1 (select):
Input --> Key Value
UserID::Gender::Age::Occup::Zip-code UserID 'UsersTable'

Job2 (select):
Input --> Key Value
UserID::MovieID::Rating::Timestamp UserID 'RatingsTable::'+MovieID

Job3 (join):
Input --> Key Value
UserID 'UsersTable'
UserID 'RatingsTable::'+MovieID

Job4 (select):
Input --> Key Value
MovieID::Title::Genre MovieID 'MoviesTable::'+Title

Job5 (join):
Input --> Key Value
MovieID 'UsersRatingsTable'
MovieID 'MoviesTable::'+Title

Job6 (distinct):
Input --> Key Value
Title 'MoviesUsersRatingsTable' Title 'MoviesUsersRatingsTable'
<table>
<thead>
<tr>
<th>Job</th>
<th>Mapper Input</th>
<th>Reducer Input</th>
<th>Output</th>
</tr>
</thead>
</table>
| 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  
... |
[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  
[reducer 1] | Out3 44 UsersTable 44 RatingsTable::101 44 RatingsTable::103 45 RatingsTable::108 46 RatingsTable::115  
... |
[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  
[reducer 1] | Out5 101 UsersRatingsTable 101 MoviesTable::Taken 101 MoviesTable::Taken  
... |
| Job6 | Out5 Taken Movies...Table Matrix Movies...Table Taken Movies...Table Matrix Movies...Table  
[reducer 2] | Out6 Taken Movies...Table Matrix Movies...Table Matrix Movies...Table  
... |
Notes:

Let us consider how to do the join in Job 3. Taking a closer look at this process will help us to understand 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.

There are some differences between the standard nested loop join described above and the join that we will implement in Hadoop. First, when we perform a standard nested loop join we run loops over two separate tables. However, our Hadoop design will feed both the inner and outer table into the same MapReduce job. This means we will have only one large table that contains rows from both source tables. We will run both our inner and our outer loop over this single table and use the Value field of each row to determine which table the row came from. (You might try to think of ways to revise or optimize this step.)

Second, a MapReduce job groups all of its input rows by key and sends all rows in a given group to a single reduce job. In our case, the key is UserID, so when we write our reduce job, we already know that all input rows to a single job will have matching UserID values. In other words, there is no need to explicitly check whether Users.UserID matches Ratings.UserID. Thus, to perform the join, we loop through every row of input, and when we find an outer table (Users) entry, we then begin another loop over every row of input to find rows from the inner table (Ratings). When we find such a row, we can immediately produce an output row (without checking for a key match). In our current query, there will only be one row from the Users table (because users are unique), but in other queries, joins may find multiple rows from both tables. The reduce job should output one row for each Outer-Inner pair that we find.

To recapitulate, the Value field of each input row tells us whether that row is part of the inner or outer table. We have decided to use the Users table as our outer table. When we look through the rows of input, if we find a row where Value = 'UsersTable,' then we know that we need to loop through all of the rows again to search for rows where Value = 'RatingsTable::' + MovieId.

Assignment:

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. You should turn in your code and your query results.
/* 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.
Useful Commands:

# recursively remove a directory
hdfs dfs -rm -r /user/[username]/mlens/out3

# view the first few lines of output from a finished MapReduce job
hdfs dfs -cat /user/[username]/mlens/out2/part-00000 | head

# list HDFS contents
hdfs dfs -ls -R /user/[username]

# load a file into HDFS
hdfs dfs -copyFromLocal movies.dat /user/[username]/mlens/input

It can be difficult to visualize the contents of your hdfs scratch space. Consider adding this to your .bashrc:

```
alias hdst="echo "/user/[username]/";hdfs dfs -ls -R /user/[username] | awk ' {print $8}' | sed -e 's/[^-][^/]*\|--/-/g' -e 's/[^/]/ / -e 's/-/1/''
```