AKHIL
18CS10070

Question 1

a) Download the rating file, parse it and load it in an RDD named ratings.

LOGIC :

- we make a case class to map each input line to an object whose attributes indicate the field.
- we make a function that will convert each row of RDD (initially in string format) to the object of the above-mentioned class.

CODE :

```
case class Rating(user_ID: Integer, movie_ID: Integer, rating: Integer,
timestamp: String)
def parse_function( inp : String ): Rating = {
    val split_list = inp.split("::").map(_.trim).toList
    val obj = Rating( split_list( 0 ).toInt, split_list( 1 ).toInt,
split_list( 2 ).toInt, split_list( 3 ) )
    return obj
}


val rating_data_path = "/home/akhil/Desktop/sem-7/SDM/data/ratings.dat"

// makes RDD by reading(each row as a string) then
// replaces each row of RDD to Rating class objects
val ratings = sc.textFile(rating_data_path).map( temp =>
parse_function(temp) )
```

OUTPUT:

```
defined class Rating
parse_function: (inp: String)Rating
rating_data_path: String = /home/akhil/Desktop/sem-7/SDM/data/ratings.dat
ratings: org.apache.spark.rdd.RDD[Rating] = MapPartitionsRDD[895] at map at
/home/akhil/Desktop/sem-7/SDM/Q1.scala:183
```

b) How many lines does the ratings RDD contain? (20 points):

LOGIC:

- We use the .count function

CODE:

```
println("lines = "+ ratings.count)
```

OUTPUT:

```
lines = 1000209
```

c) Count how many unique movies have been rated.

LOGIC:

- we iterate on rows of rating RDD and take only movie_ID field and apply .distinct function and then .count() function

CODE:

```
println("unique movies = "+ ratings.map( iter =>
iter.movie_ID).distinct.count() )
```

OUTPUT:

```
unique movies = 3706
```

d) Which user gave the most ratings? Return the userID and number of ratings.

LOGIC:

- here we need to find users with max ratings.
- We map the unique users to 1 and take the sum by reduceByKey function and thus obtain the user_id and corresponding frequencies.
- Then we sort it based on the frequency and take the first element.

CODE:

```
var ans = ratings.
keyBy(x => x.user_ID)
.mapValues(x => 1)
.reduceByKey((x,y) => x + y)
.sortBy(_._2, false)
.take(1)
println("User :", ans)
```

OUTPUT:

```
res229: Array[(Integer, Int)] = Array((4169,2314))
(User :,MapPartitionsRDD[900] at keyBy at
/home/akhil/Desktop/sem-7/SDM/Q1.scala:181)
```

e) Which user gave the most '5' ratings? Return the userID and number of ratings.

LOGIC:

- first we filter out the rows having rating field value = 5
- then we order those rows by user_ID field
- then we make 2 columns and define the logic of 2nd column
- then sort by decreasing order in 2nd column and take the first entry after sorting

CODE:

```
ratings.
filter(x => x.rating == 5).keyBy(x => x.user_ID).
aggregateByKey(0)((acc, x) => acc + 1, (x, y) => x + y).
sortBy(_._2, false).take(1)
```

OUTPUT:

```
res231: Array[(Integer, Int)] = Array((4277,571))
```

Question 2

a) Read the movies and users files into RDDs. How many records are there in each RDD?

LOGIC :

- Here also we make 2 new case classes namely Movie and User
- and make 2 functions to take a string as input and parse it and return a new object of the corresponding class
- .map(x => parser(x)) will basically replace the string row in RDD to a parsed object which is returned by parser( ) function

CODE :

```
case class Movie(movie_ID: Integer, title: String, genre: String)
case class User(user_ID: Integer, gender: String, age: Integer, occupation: String,
zip_code: String)

def movie_parser(row: String): Movie = {
    val splitted = row.split("::").map(_.trim).toList
    return Movie(splitted(0).toInt, splitted(1), splitted(2))
}
def user_parser(row: String): User = {
    val splitted = row.split("::").map(_.trim).toList
    return User(splitted(0).toInt, splitted(1), splitted(2).toInt, splitted(3),
splitted(4))
}


val movies=sc.textFile("/home/akhil/Desktop/sem-7/SDM/data/movies.dat").map(element
=> movie_parser(element)).cache

val users = sc.textFile("/home/akhil/Desktop/sem-7/SDM/data/users.dat").map(element
=> user_parser(element)).cache

println("movies lines = " + movies.count)
println("users lines  = " + users.count)
```

OUTPUT:

```
defined class Movie
defined class User
parseMovies: (row: String)Movie
parseUsers: (row: String)User
movies: org.apache.spark.rdd.RDD[Movie] = MapPartitionsRDD[918] at map at
/home/akhil/Desktop/sem-7/SDM/Q2.scala:184
users: org.apache.spark.rdd.RDD[User] = MapPartitionsRDD[921] at map at
/home/akhil/Desktop/sem-7/SDM/Q2.scala:184
```

```
movies lines = 3883
users lines  = 6040
```

b) How many of the movies are comedies?

LOGIC :

- we apply a filter to check for the presence of “Comedy” as a substring in the genre field which is pipe separated

CODE :

```
var ans = movies.filter(x => x.genre.contains("Comedy")).count
println("comedy movies = " + ans)
```

OUTPUT:

```
ans: Long = 1200
comedy movies = 1200
```

c) Which comedy has the most ratings? Return the title and the number of rankings. Answer this question by joining two datasets.

LOGIC :

- we apply a filter for the “Comedy” genre
- then order them by movieID field
- join the movie RDD with ratings RDD

CODE :

```
var ans = movies.
filter(  x => x.genre.contains("Comedy")).
keyBy(   x => x.movie_ID).
join(    ratings.keyBy(x => x.movie_ID)).
map(     x => (x._2._1.title)).
groupBy( x => x).
map{case (k, v) => (k, v.size)}.
sortBy(  x => x._2, false).
take(1)
println("comedy having most ratings :" + ans(0) )
```

OUTPUT:

```
ans: Array[(String, Int)] = Array((American Beauty (1999),3428))
comedy having most ratings :(American Beauty (1999),3428)
```

e) Compute the number of unique users that rated the movies with movie_IDs 2858, 356, and 2329 without using an inverted index. Measure the time (in seconds) it takes to make this computation.

LOGIC :

- Here we again make the case class for Rating and the parser function, now we use filter with OR condition in it
- Then we group the rows by user_ID field then take the count of distinct keys in it
- for time calculation we use the inbuilt function `System.currentTimeMillis( )`

CODE :

```
val time1: Long = (System.currentTimeMillis)

println("unique users who rated movies[2858,356,2329] = " )
ratings.
filter( x=> x.movie_ID == 2858 || x.movie_ID == 356 || x.movie_ID == 2329 )
.groupBy( x=> x.user_ID).keys.distinct.count

println("Time taken = " + ((System.currentTimeMillis) - time1)/1000.0 + "
seconds" )
```

OUTPUT:

```
unique users who rated movies[2858,356,2329] =
res238: org.apache.spark.rdd.RDD[Rating] = MapPartitionsRDD[941] at filter
at /home/akhil/Desktop/sem-7/SDM/Q2.scala:185
res239: Long = 4213    //←  this is the answer

Time taken = 2.934 seconds
```

f) Create an inverted index on ratings, field movie_ID. Print the first item.

LOGIC :

- Used groupBy in order to compile all records with the key into an iterable list mapped to the key and returns this rdd.

CODE :

```
val inv_index = ratings.groupBy(r => r.movie_ID).cache
inv_index.lookup(1).take(1)
```

OUTPUT:

```
inv_index: org.apache.spark.rdd.RDD[(Integer, Iterable[Rating])] = ShuffledRDD[949] at groupBy at
/home/akhil/Desktop/sem-7/SDM/Q2.scala:183
res241: Seq[Iterable[Rating]] = ArrayBuffer(CompactBuffer(Rating(1,1,5,978824268),
Rating(6,1,4,978237008), Rating(8,1,4,978233496), Rating(9,1,5,978225952), Rating(10,1,5,978226474),
Rating(18,1,4,978154768), Rating(19,1,5,978555994), Rating(21,1,3,978139347), Rating(23,1,4,978463614),
Rating(26,1,3,978130703), Rating(28,1,3,978985309), Rating(34,1,5,978102970), Rating(36,1,5,978061285),
```

```
Rating(38,1,5,978046225), Rating(44,1,5,978019369), Rating(45,1,4,977990044), Rating(48,1,4,977975909),
Rating(49,1,5,977972501), Rating(51,1,5,977947828), Rating(56,1,5,977938855), Rating(60,1,4,977931983),
Rating(65,1,5,991368774), Rating(68,1,3,991376026), Rating(73,1,3,977867812), Rating(75,1,5,977851099),
Rating(76,1,5,977847069), Rating(78,1,4,978570648), Rating(80,1,3,9...
```

g) Compute the number of unique users that rated the movies with movie_IDs 2858, 356 and 2329 using the above-calculated index. Measure the time (in seconds) it takes to compute the same result using the index.

LOGIC :

- The inverted index is used to get individual users list given the movie_ID by filtering followed flatMap to get the ratings RDD for that MovieID
- Then we create a list with unique users individually for the 3 given MovieIDs,
- Then see for any intersection of the 3 items of lists similar to 4th part of this question

CODE :

```
val to_search = sc.parallelize(List[Integer](2858,356,2329)).keyBy(x => x)

val time2: Long = (System.currentTimeMillis)
inv_index.join(to_search).flatMap{x => x._2._1}.map(x =>
x.user_ID).distinct.count

println("Time taken = " + ((System.currentTimeMillis) - time2)/1000.0 + "
seconds" )
```

OUTPUT:

```
to_search: org.apache.spark.rdd.RDD[(Integer, Integer)] =
MapPartitionsRDD[951] at keyBy at
/home/akhil/Desktop/sem-7/SDM/Q2.scala:182
time2: Long = 1630433851095
res242: Long = 4213
Time taken = 16.591 seconds
```

Question 3

CODE TO TAKE INPUT & PARSE IT :

```
case class my_class( debug_level : String, timestamp : Option[Timestamp]
,download_id : String , retrieval_stage : String , rest : String)

var timeFormat = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ssX")

// PARSING FUNCTION
def parse_function = {inp: String => {
        try{
            var temp = inp.split("--").toList
            val result = (temp(0).split(",").toList.map(_.trim()) ++
temp(1).split(":",2).toList.map(_.trim()))
            if(result.length == 5)    Some(result)
            else    None
        }catch{
            case e1 : java.lang.IndexOutOfBoundsException => None
        }
    }
}

// TIME CONVERSION FUNCTION
def time_function (inp : String) : Option[Timestamp] = {
    try{ Some(new Timestamp(format.parse(inp).getTime()))}
    catch{ case e: Exception => None}
}

// FUNCTION TO TAKE INPUT AND RETURN A RDD
def take_input(inp : String) : org.apache.spark.rdd.RDD[my_class]={
       var temp = sc.textFile(inp)
       var parsedRDD = temp.filter( x => x.size > 0 ).map( line => line.split(", |
-- ",4) )
       parsedRDD = parsedRDD.map( line => if( line.size == 4 )
Array(line(0),line(1),line(2))++line(3).split(": ",2) else line)
       var myRDD = parsedRDD.map( x => try{
             my_class( x(0), timeFormat.parse(x(1)), x(2), x(3), x(4) )
             } catch {
                    case e : Exception => my_schema(null, null, null, null, null)
             }
       )
       return myRDD
}
```

a. Create a function that given an RDD and a field (e.g. download_id), it computes an inverted index on the RDD for efficiently searching the records of the RDD using values of the field as keys.

LOGIC:

- The function extracts the field that is passed as parameter, sets it as key and then groups by key to form an inverted index.

CODE :

```
def get_inverted_index(rdd : RDD[my_class], field_idx: Int) = {
    rdd.map(x => (x.productElement(field_idx), x)).
        groupByKey()
}
```

OUTPUT :

```
get_inverted_index: (rdd: org.apache.spark.rdd.RDD[log_file_schema],
field_idx: Int)org.apache.spark.rdd.RDD[(Any, Iterable[log_file_schema])]
```

b. Compute the number of different repositories accessed by the client 'ghtorrent-22' (without using the inverted index).

LOGIC:

- Filter by download_id ghtorrent-22
- Count number of distinct repository names

CODE:

```
var ans = rdd.filter(_.download_id ==
"ghtorrent-22").map(get_repo_name).distinct.count
print(ans)
```

OUTPUT:

```
4571
```

c. Compute the number of different repositories accessed by the client 'ghtorrent-22' using the inverted index calculated above.

LOGIC:

- Compute an inverted index using the function above and store as an RDD
- Get the list of values for ghtorrent-22 and count the number of distinct repositories in the list

CODE:

```
val index = get_inverted_index(rdd, 2)
val filter22 = index.filter(x => x._1 == "ghtorrent-22")
val filtered = filter22.mapValues(x => {
```

```
    var result = List[my_class]();
    x.iterator.toList
}).flatMap(x => x._2).map(x => get_repo_name(x))

var ans = filtered.distinct.count
print("ans="+ans)
```

OUTPUT:

```
index: org.apache.spark.rdd.RDD[(Any, Iterable[log_file_schema])] =
ShuffledRDD[11] at groupByKey at /home/akhil/Desktop/sem-7/SDM/Q3.scala:30
filter22: org.apache.spark.rdd.RDD[(Any, Iterable[log_file_schema])] =
MapPartitionsRDD[12] at filter at /home/akhil/Desktop/sem-7/SDM/Q3.scala:29
filtered: org.apache.spark.rdd.RDD[Option[String]] = MapPartitionsRDD[15]
at map at /home/akhil/Desktop/sem-7/SDM/Q3.scala:36

ans=4571
```