BIG DATA ANALYTICS AND REASONING

Project 05 - Big Data Social Network

Project report

Project team members

1.Gebreyowhans Hailekiros Bahre[MAT.233619]

2. Renier Perez Gallardo [MAT.232693]

Problem 1 (Sqoop - Hive)

Creating MySQL database, tables and loading the csv data into tables

1. **Create database;**

```
create database SocialNetwork;
```

2. **Create user table**

```
create table USER (email VARCHAR(255) NOT NULL PRIMARY KEY,fullname
VARCHAR(255) NOT NULL,age INT NOT NULL,followers_count INT NOT NULL);
```

3. **Create post table**

```
reate table POST (post_id INT NOT NULL PRIMARY KEY,category VARCHAR(255)
NOT  NULL,date DATE NOT NULL,user_email VARCHAR(255) NOT
NULL,FOREIGN KEY
     (user_email) REFERENCES USER(email));
```

4. **Create Likes table:**

```
Create table LIKES (user_email VARCHAR(255) NOT NULL,post_id INT NOT
NULL,PRIMARY KEY (user_email, post_id),FOREIGN KEY (user_email) REFERENCES
USER(email),FOREIGN KEY (post_id) REFERENCES POST(post_id)
   );
```

5. **load users.csv data into user table**

```
load data local infile 'users.csv' into table USER fields terminated by ',';
```

6. **load posts.csv into post table**

```
load data local infile 'posts.csv' into table POST fields terminated by ',';
```

7. **load likes.csv into likes table**

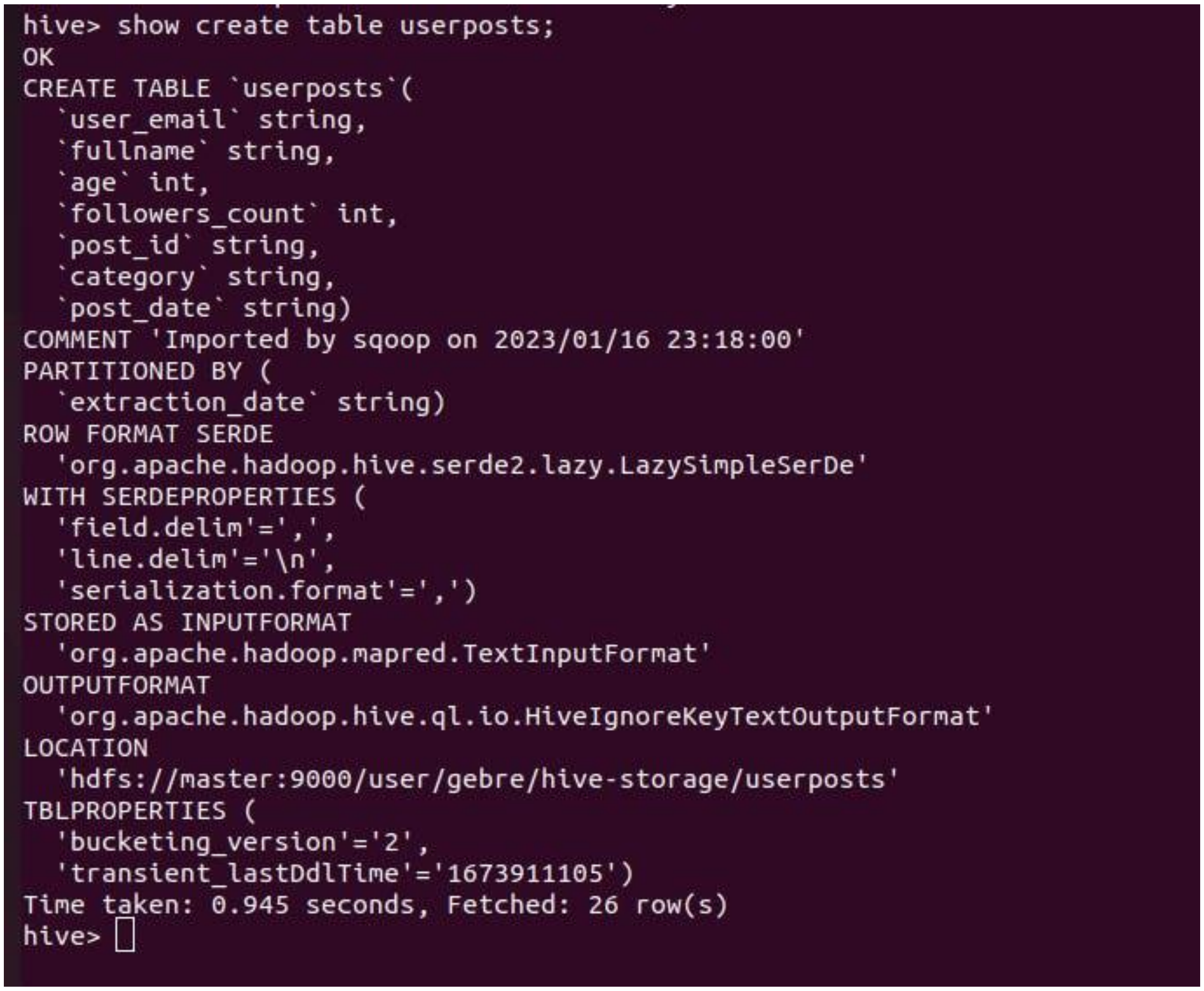
```
load data local infile 'likes.csv' into table LIKES fields terminated by ',';
```

I. Initial import of user posts from the MySQL database into hive-storage database using sqoop

```
sqoop-import --connect jdbc:mysql://master/SocialNetwork --username gebre -P  --target-dir hive-storage/socialnetwork.db/userposts/  --query  'select p.user_email,u.fullname,u.age,u.followers_count,p.post_id,p.category,p.date as post_date from USER as u join POST as p on u.email=p.user_email where $CONDITIONS'  --hive-import --hive-table userposts --hive-partition-key extraction_date --hive-partition-value '2023-01-22' --fields-terminated-by ',' -m 1
```

The above Sqoop command imports user posts data from MySQL database by joining the USER and POST tables by user email key and puts the result inside the users folder of the hive-storage directory and this data is partitioned based on extraction date (i.e. the date in which the data is imported from MySQL).

Note that when we run the above query for the first time it will automaticaly create a managed hive table named **userposts** which is partitioned by extraction date . The image in below shows the definition of the hive table created automatically.

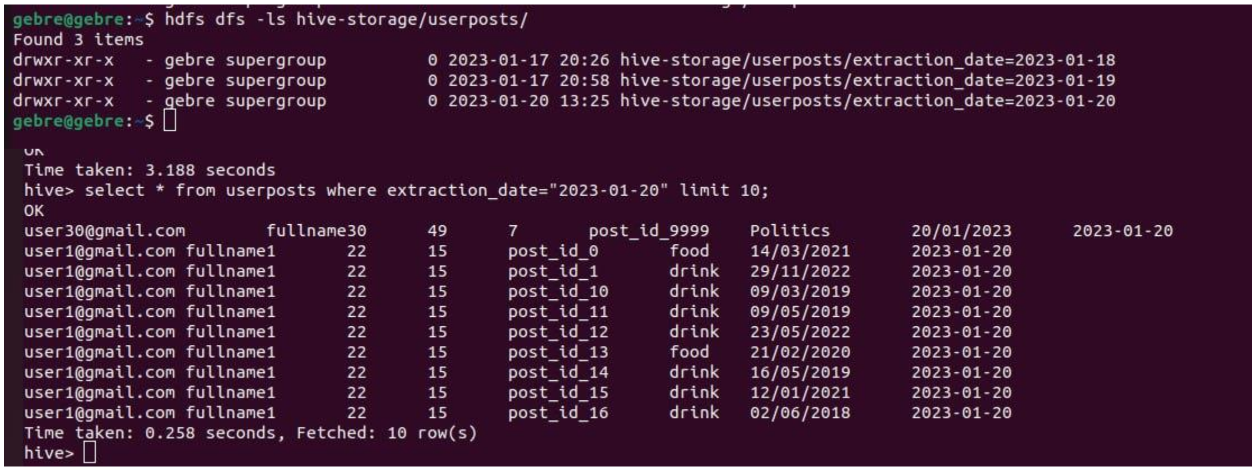


```
hive> show create table userposts;
OK
CREATE TABLE `userposts`(
  `user_email` string,
  `fullname` string,
  `age` int,
  `followers_count` int,
  `post_id` string,
  `category` string,
  `post_date` string)
COMMENT 'Imported by sqoop on 2023/01/16 23:18:00'
PARTITIONED BY (
  `extraction_date` string)
ROW FORMAT SERDE
  'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'
WITH SERDEPROPERTIES (
  'field.delim'=',',
  'line.delim'='\n',
  'serialization.format'=',')
STORED AS INPUTFORMAT
  'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT
  'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION
  'hdfs://master:9000/user/gebre/hive-storage/userposts'
TBLPROPERTIES (
  'bucketing_version'='2',
  'transient_lastDdlTime'='1673911105')
Time taken: 0.945 seconds, Fetched: 26 row(s)
hive>
```

To perform incremental import on this hive table we can use the post_date column of the POST table and imports it accordingly and puts it in separate partition based on extraction date like as in the below example.

sqoop-import --connect jdbc:mysql://master/SocialNetwork --username gebre -P --query 'select p.user_email,u.fullname,u.age,u.followers_count,p.post_id,p.category,p.date as post_date from USER as u join POST as p on u.email=p.user_email where $CONDITIONS and STR_TO_DATE(p.date,"%d/%m/%Y") >="2022-01-23"' --hive-table userposts --hive-import --hive-partition-key extraction_date --hive-partition-value '2023-01-24' --target-dir hive-storage/socialnetwork.db/userposts/ --fields-terminated-by ',' -m 1

As you can see in the above sqoop query , it imports the data starting from 2023-01-23 and puts it inside the userposts folder by creating another folder partition named extraction_date=2023-01-24. So every time we perform incremental import our Sqoop query will put the data in separate folders (i.e. depending the date in which the data is extracted) and appends it to the same hive table too. The following two images shows as how the incrementally imported files are stored inside our hdfs and some select query from the hive table.



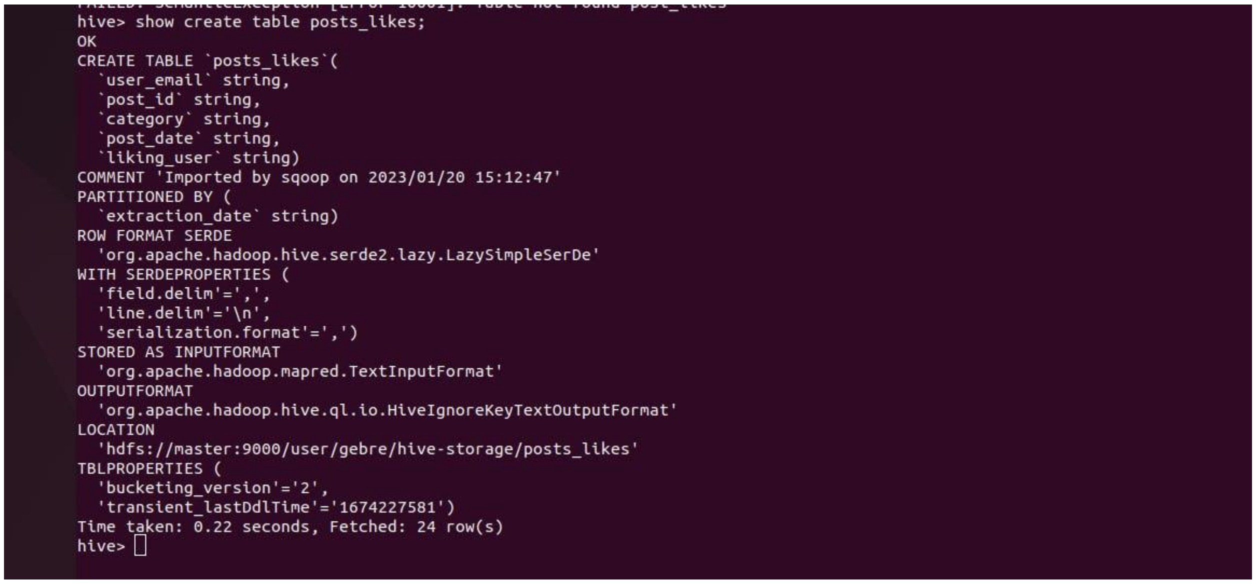
```
gebre@gebre:~$ hdfs dfs -ls hive-storage/userposts/
Found 3 items
drwxr-xr-x   - gebre supergroup          0 2023-01-17 20:26 hive-storage/userposts/extraction_date=2023-01-18
drwxr-xr-x   - gebre supergroup          0 2023-01-17 20:58 hive-storage/userposts/extraction_date=2023-01-19
drwxr-xr-x   - gebre supergroup          0 2023-01-20 13:25 hive-storage/userposts/extraction_date=2023-01-20
gebre@gebre:~$ 
```

```
 OK
 Time taken: 3.188 seconds
 hive> select * from userposts where extraction_date="2023-01-20" limit 10;
 OK
 user30@gmail.com        fullname30      49      7       post_id_9999    Politics        20/01/2023      2023-01-20
 user1@gmail.com fullname1       22      15      post_id_0       food    14/03/2021      2023-01-20
 user1@gmail.com fullname1       22      15      post_id_1       drink   29/11/2022      2023-01-20
 user1@gmail.com fullname1       22      15      post_id_10      drink   09/03/2019      2023-01-20
 user1@gmail.com fullname1       22      15      post_id_11      drink   09/05/2019      2023-01-20
 user1@gmail.com fullname1       22      15      post_id_12      drink   23/05/2022      2023-01-20
 user1@gmail.com fullname1       22      15      post_id_13      food    21/02/2020      2023-01-20
 user1@gmail.com fullname1       22      15      post_id_14      drink   16/05/2019      2023-01-20
 user1@gmail.com fullname1       22      15      post_id_15      drink   12/01/2021      2023-01-20
 user1@gmail.com fullname1       22      15      post_id_16      drink   02/06/2018      2023-01-20
 Time taken: 0.258 seconds, Fetched: 10 row(s)
 hive> 
```

II. Initial import of posts likes by joining the POST and LIKE tables of MySQL database into hive-storage.

    sqoop-import --connect jdbc:mysql://master/SocialNetwork --username gebre -P --query 'select p.user_email,p.post_id,p.category,p.date as post_date from POST as p join LIKES as l on p.post_id=l.post_id where $CONDITIONS' --target-dir hive-storage/socialnetwork.db/ --split-by p.post_id --hive-import --hive-partition-key extraction_date --hive-partition-value '2023-01-20' --fields-terminated-by ',' --hive-table posts_likes --direct -m 1

The above Sqoop query imports the posslikes data from MySQL data base by joining the POST and USER tables by joining them using post_id as key and puts the data inside hive-storage/posts_likes folder using extraction date as a partition too. The first time this query is executed it automatically created managed hive table named posts_likes and we can directly write an SQL query to access the post likes data that is imported in to the hive storage folder. The hive table definition looks like as in the below.

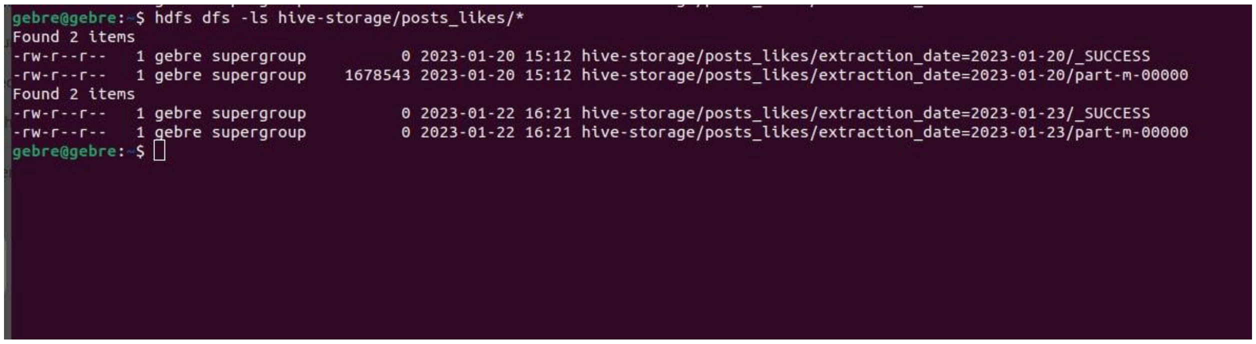


```
hive> show create table posts_likes;
OK
CREATE TABLE `posts_likes`(
  `user_email` string,
  `post_id` string,
  `category` string,
  `post_date` string,
  `liking_user` string)
COMMENT 'Imported by sqoop on 2023/01/20 15:12:47'
PARTITIONED BY (
  `extraction_date` string)
ROW FORMAT SERDE
  'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'
WITH SERDEPROPERTIES (
  'field.delim'=',',
  'line.delim'='\n',
  'serialization.format'=',')
STORED AS INPUTFORMAT
  'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT
  'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION
  'hdfs://master:9000/user/gebre/hive-storage/posts_likes'
TBLPROPERTIES (
  'bucketing_version'='2',
  'transient_lastDdlTime'='1674227581')
Time taken: 0.22 seconds, Fetched: 24 row(s)
hive>
```

To perform incremental import on this hive table we can use the post_date column of the POST table and imports it accordingly and puts it in separate partition based on extraction date like as in the below example.

sqoop-import --connect jdbc:mysql://master/SocialNetwork --username gebre -P --query 'select p.user_email,p.post_id,p.category,p.date as post_date from POST as p join LIKES as l on p.post_id=l.post_id where $CONDITIONS and STR_TO_DATE(p.date,"%d/%m/%Y") >="2023-01-23"' --target-dir hive-storage/socialnetwork.db/posts_likes/ --split-by p.post_id --hive-import --hive-partition-key extraction_date --hive-partition-value '2023-01-23' --fields-terminated-by ',' --hive-table posts_likes --direct -m 1

As you can see in the above sqoop query , it imports the data starting from 2023-01-23 and puts it inside the posts_likes folder by creating another folder partition named extraction_date=2023-01-23. So every time we perform incremental import our Sqoop query will put the data in separate folders (i.e. depending the date in which the data is extracted) and appends it to the same hive table too.



```
gebre@gebre:~$ hdfs dfs -ls hive-storage/posts_likes/*
Found 2 items
-rw-r--r--   1 gebre supergroup          0 2023-01-20 15:12 hive-storage/posts_likes/extraction_date=2023-01-20/_SUCCESS
-rw-r--r--   1 gebre supergroup    1678543 2023-01-20 15:12 hive-storage/posts_likes/extraction_date=2023-01-20/part-m-00000
Found 2 items
-rw-r--r--   1 gebre supergroup          0 2023-01-22 16:21 hive-storage/posts_likes/extraction_date=2023-01-23/_SUCCESS
-rw-r--r--   1 gebre supergroup          0 2023-01-22 16:21 hive-storage/posts_likes/extraction_date=2023-01-23/part-m-00000
gebre@gebre:~$
```

Starting from the above imported tables data we can do some calculations for example computing metrices asked in the problem 2 below .

Example 1: we can count the number of posts made by a user per category using the following hive query :
 select user_email,category,count(post_id) as postcount from posts_likes where extraction_date='2023-01-20' group by user_email,category;

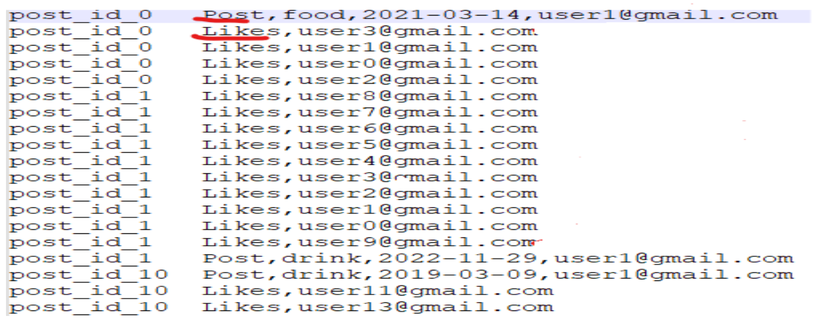
Problem 2 (HBase - Map Reduce)

Task 1 : Design a MapReduce job that joins post.csv and likes.csv into an HBase table (posts)

To solve this task we used two maper classes and one reducer class :

1. **PostsMapper** which is responsible for parsing **posts.csv** file and emit it into and hdfs file system using post id as key and category, date,useremail as values and here we also included a tag named **post** at the beginning of the values string which will further be used as an identifier to Post or likes in the reducer side.

2. **LikesMapper** which is responsible for parsing **likes.csv** file and emit into hdfs file system using post id as key and useremail(a user who likes that post id) as value and here we also included a tag named Likes at the beginning of the values string which will further be used as an identifier to post or likes in the reducer side. The image attached in the below shows the output of the two mappers classes(i.e. the data which is ready to be sent to the reducer).

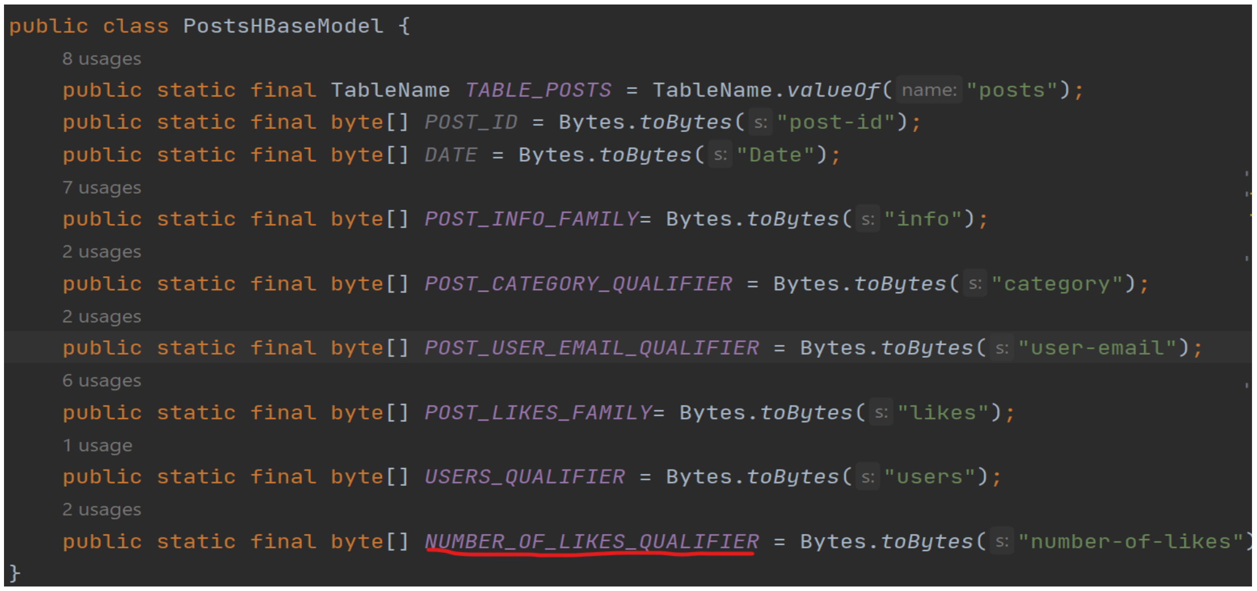


```
post_id_0    Post,food,2021-03-14,user1@gmail.com
post_id_0    Likes,user3@gmail.com
post_id_0    Likes,user1@gmail.com
post_id_0    Likes,user0@gmail.com
post_id_0    Likes,user2@gmail.com
post_id_1    Likes,user8@gmail.com
post_id_1    Likes,user7@gmail.com
post_id_1    Likes,user6@gmail.com
post_id_1    Likes,user5@gmail.com
post_id_1    Likes,user4@gmail.com
post_id_1    Likes,user3@gmail.com
post_id_1    Likes,user2@gmail.com
post_id_1    Likes,user1@gmail.com
post_id_1    Likes,user0@gmail.com
post_id_1    Likes,user9@gmail.com
post_id_1    Post,drink,2022-11-29,user1@gmail.com
post_id_10   Post,drink,2019-03-09,user1@gmail.com
post_id_10   Likes,user11@gmail.com
post_id_10   Likes,user13@gmail.com
```

3. The mappers will finally send list of PostId keys and list of users who liked the postid ,date the post is posted , the post category and the user who posts the post to reducer class.

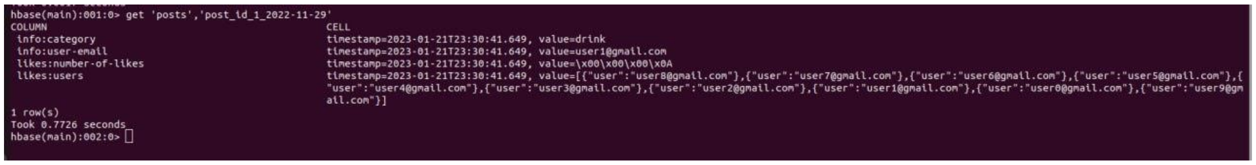
4. The **PostLikesReducer** then receives the data from the mappers and make some parsing to collect all the users who liked a particular postid into a json array object, uses postid_date as row key ,info as column family 1 with category and user email as column qualifier,likes column family with users qualifier (stores list of users who liked the post id in json array format),numberoflikes qualifier we created it by ourselves to store the number of users who liked the post because this column will help us in the second task and finally this reducer stores this all info into an hbase table named **posts**.

   The image attached in the below shows a code snippet of the posts hbase table.



```
public class PostsHBaseModel {
    8 usages
    public static final TableName TABLE_POSTS = TableName.valueOf( name: "posts");
    public static final byte[] POST_ID = Bytes.toBytes( s: "post-id");
    public static final byte[] DATE = Bytes.toBytes( s: "Date");
    7 usages
    public static final byte[] POST_INFO_FAMILY= Bytes.toBytes( s: "info");
    2 usages
    public static final byte[] POST_CATEGORY_QUALIFIER = Bytes.toBytes( s: "category");
    2 usages
    public static final byte[] POST_USER_EMAIL_QUALIFIER = Bytes.toBytes( s: "user-email");
    6 usages
    public static final byte[] POST_LIKES_FAMILY= Bytes.toBytes( s: "likes");
    1 usage
    public static final byte[] USERS_QUALIFIER = Bytes.toBytes( s: "users");
    2 usages
    public static final byte[] NUMBER_OF_LIKES_QUALIFIER = Bytes.toBytes( s: "number-of-likes")
}
```

The image in the below shows the result of how the data is stored in the posts hbase table.



```
hbase(main):001:0> get 'posts','post_id_1_2022-11-29'
COLUMN                                              CELL
 info:category                                      timestamp=2023-01-21T23:30:41.649, value=drink
 info:user-email                                    timestamp=2023-01-21T23:30:41.649, value=user1@gmail.com
 likes:number-of-likes                              timestamp=2023-01-21T23:30:41.649, value=\x00\x00\x00\x0A
 likes:users                                        timestamp=2023-01-21T23:30:41.649, value=[{"user":"user8@gmail.com"},{"user":"user7@gmail.com"},{"user":"user6@gmail.com"},{"user":"user5@gmail.com"},{
                                                    "user":"user4@gmail.com"},{"user":"user3@gmail.com"},{"user":"user2@gmail.com"},{"user":"user1@gmail.com"},{"user":"user0@gmail.com"},{"user":"user9@gm
                                                    ail.com"}]
1 row(s)
Took 0.7726 seconds
hbase(main):002:0> 
```

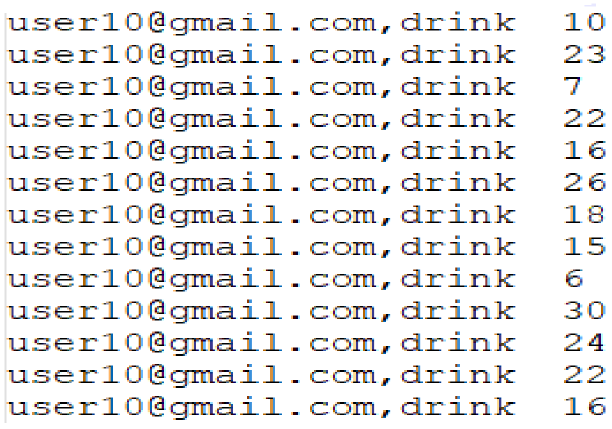
Task 2 : Design a MapReduce job that reads from the previous HBase table and computes:

1. Count of posts per category posted by a user

2. Average number of likes obtained by posts per category of each users

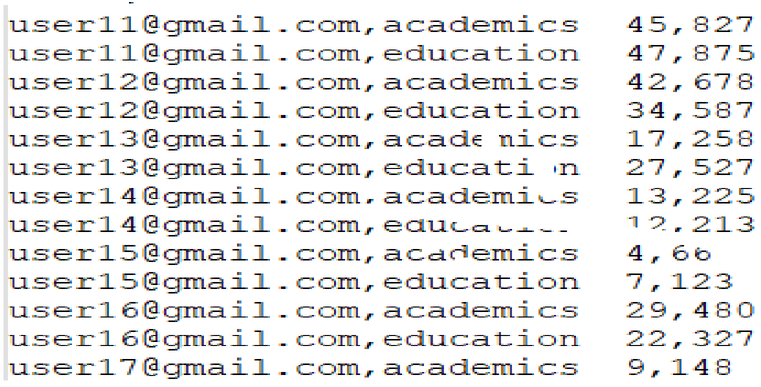
To solve this problem we used one maper, one combiner and one reducer classes where

1. The maper class named **PostCountAndAvgLikesMapper** reads data from hbase table and extracts the post category**,** user email who made post and number of likes. Then we Constructed a composite key using the user ID and category and we emit the composite key as a key and the number of likes as the value and send this data to the combiner class to perform map side aggregation. The result of this maper looks like as in the below image.



```
user10@gmail.com,drink  10
user10@gmail.com,drink  23
user10@gmail.com,drink  7
user10@gmail.com,drink  22
user10@gmail.com,drink  16
user10@gmail.com,drink  26
user10@gmail.com,drink  18
user10@gmail.com,drink  15
user10@gmail.com,drink  6
user10@gmail.com,drink  30
user10@gmail.com,drink  24
user10@gmail.com,drink  22
user10@gmail.com,drink  16
```

2. The combiner class named **PostCountAndAvgLikesCombiner** takes the key value pairs obtained from the previous mapper and groups all post categories which have the same composite key and sums the number of likes obtained for each category and produces a new key value pair that has count of all posts of the same category per user and sum of likes for each post category and the same composite key too. The result of the combiner class looks like in the below image.

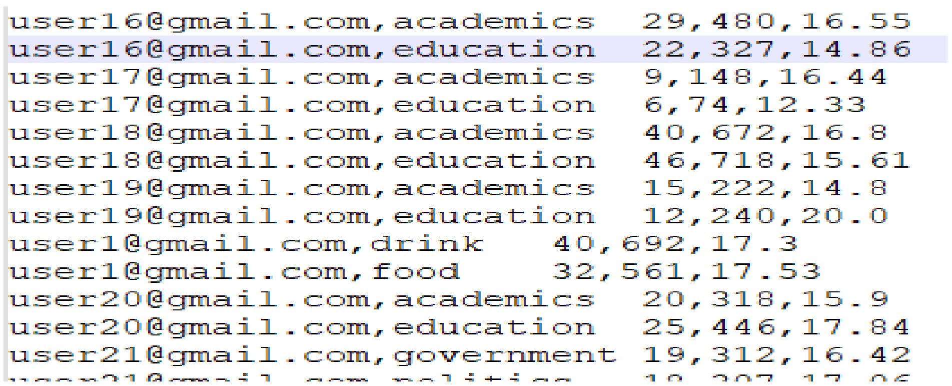


```
user11@gmail.com,academics  45,827
user11@gmail.com,education  47,875
user12@gmail.com,academics  42,678
user12@gmail.com,education  34,587
user13@gmail.com,academics  17,258
user13@gmail.com,education  27,527
user14@gmail.com,academics  13,225
user14@gmail.com,education  12,213
user15@gmail.com,academics  4,66
user15@gmail.com,education  7,123
user16@gmail.com,academics  29,480
user16@gmail.com,education  22,327
user17@gmail.com,academics  9,148
```

3. The Reducer class named **PostCountAndAvgLikesReducer** the receives the aggregate count of posts and total number of likes obtained per every user post and category and then computes the average number of likes obtained using

   $$\text{Average likes} = \frac{\textbf{total likes obtained per post category of user}}{\textbf{count of post category}}$$

   Then it will finally emits, the composite key as a key **,** count of posts ,number of likes ,Average likes obtained as values. The output of the reducer will look like as in the below image.



```
user16@gmail.com,academics   29,480,16.55
user16@gmail.com,education   22,327,14.86
user17@gmail.com,academics   9,148,16.44
user17@gmail.com,education   6,74,12.33
user18@gmail.com,academics   40,672,16.8
user18@gmail.com,education   46,718,15.61
user19@gmail.com,academics   15,222,14.8
user19@gmail.com,education   12,240,20.0
user1@gmail.com,drink    40,692,17.3
user1@gmail.com,food     32,561,17.53
user20@gmail.com,academics   20,318,15.9
user20@gmail.com,education   25,446,17.84
user21@gmail.com,government 19,312,16.42
```

Problem 3 (Spark)

Task 1 : The most frequent category among the posts of the user u

First, we start to obtain this information from the results obtained in question 2 exported in csv format. With this information, a dataset is created with the corresponding schema.

```
Dataset<Row> q2result = spark.read().schema("user_email string,category string,
post_count int, likes_count int, average float").csv("/social-
networkInput/q2result.csv");
```

Then using the group by and aggregation functions of the dataset, it is grouped by email and an aggregation is used to obtain the maximum value of the "post_count" column in the grouping, thus obtaining a new dataset with the expected result.

```
q2result.groupBy("user_email").agg(functions.max("post_count"));
```

Task 2 : The category with the highest average of likes among posts of the user u

To obtain this result, we proceeded in the same way with the "average" column.

```
q2result.groupBy("user_email").agg(functions.max("average"));
```

Task 3 : Create the USER schema

To create the schema, we first use the join function to join the dataset of the first two tasks with the dataset created from the user input file, thus obtaining a new dataset with the same rows as the original user file, but with new columns.

Example:

```
Dataset<Row> users = spark.read().schema("email string, fullname string, age int, followers_count int").csv("/social-networkInput/users.csv");

Dataset<Row> firstJoin = users.withColumnRenamed("email","user_email").join(frequent_categ,"user_email");
```

Then join this dataset with the dataset from the results of question 2, since it contains data that is needed for the schema.
The next step developed is to use the flatMap function to iterate through the dataset and for each row build a user object, of the User class implemented with the attributes required for the schema.

```
Dataset<User> user_dataset = thirdJoin.flatMap(new FlatMapFunction<Row,User>()
```

Within this function an algorithm is implemented, which since the aggregations are numeric and categorical values are needed, a comparison is made between the original values and those obtained in the aggregations to determine for each user which is the corresponding category.

```
String cat_max_post_count = null;
int max_post_count = Integer.parseInt(row.get(3).toString());
int post_count = Integer.parseInt(row.get(6).toString());
String category = row.get(5).toString();
if(max_post_count == post_count){
		cat_max_post_count = category;
}
```

Since the results of question two were obtained by repeating the user according to different categories, a dataset was obtained with the same format but only with the information required for the scheme.

Example:

email	age	followers count	cat_max_post_count	cat_max_avg_post_likes
User1	23	60	Drink	null
User1	23	60	null	Food

To unify the user information, it is grouped by common columns and an aggregation is applied to the category columns where the first category is taken, ignoring null values.

```
Dataset<Row> groupped =
user_dataset.groupBy("email","age","followersCount").agg(functions.first("catMaxPostCount",
true).alias("cat_max_post_count"),functions.first("catMaxAvgPostLikes", true).alias("cat_max_avg_post_likes"));
```

Task 4 : **Build a regression model to estimate the age of a user starting from statics about user posts. Choose the best model with the best hyperparmeters configuration**

To solve this task we use the Linear Regression, Generalized Linear Regression and Decision Tree Regression methods. In all of them, CrossValidator was used to evaluate a series of parameters in the models that are Introduced in a Grid Param with the metric, in this case, Root Mean Squared Error (RMSE). In addition, in all the methods a vectorization of the characteristics is carried out as expected for their processing.

Example:

```
StringIndexer indexer = new
StringIndexer().setInputCol("cat_max_post_count").setOutputCol("catMaxPostCountIndex");
Dataset<Row> indexed = indexer.fit(user_dataset).transform(user_dataset);
VectorAssembler assembler = new VectorAssembler().setInputCols(new String[]{"followersCount",
"catMaxPostCountIndex","catMaxAvgPostLikesIndex"}).setOutputCol("features");
Dataset<Row> output = assembler.transform(trainingData);

RegressionEvaluator evaluator = new
RegressionEvaluator().setLabelCol("age").setPredictionCol("prediction").setMetricName("rmse");

ParamMap[] paramGrid = new ParamGridBuilder().addGrid(dt.maxDepth(), new int[]{5, 7, 12}).build();
CrossValidator cv = new
CrossValidator().setEstimator(dt).setEvaluator(evaluator).setEstimatorParamMaps(paramGrid).setNumFolds(5);
CrossValidatorModel cvModel = cv.fit(output);
```

The methods were executed with different configurations and the results obtained are shown below.

Linear Regresion					
CrossValidator					
regParam	0,01	0,1	1		
elasticNetParam	0	0,5	1		
RMSE	9,32				
CrossValidator					
regParam	0,05	0,3	0,7		
elasticNetParam	0,2	0,6	0,9		
RMSE	11,83				
Individual					
regParam	0,1				
elasticNetParam	0,8				
Coefficients	0,01	0,35	-8,39		
Intercept	49,05				
numIterations	6				
objectiveHistory	0,5	0,42	0,21	0,21	0,21
residuals	2,35	-10,76	-9,52	-4,9	-1,96

regParam	0,6				
elasticNetParam	0,9				
Coefficients	-0,02	0,68	0		
Intercept	38,24				
numIterations	8				
objectiveHistory	0,5	0,5	0,5	0,5	0,5
residuals	-9,14	-7,13	-6,04	0,7	0,95

Generalized Linear Regression					
CrossValidator					
regParam	0,01	0,1	1		
RMSE	12,91				
CrossValidator					
regParam	0,05	0,3	0,7		
RMSE	21,99				
Individual					
regParam	0,1				
Coefficients	0,06	0,54	-2,19		
Intercept	45,21				
Coefficient Standard Errors	0,53	2,87	2,42	12,76	
T Values	0,12	0,19	-0,91	3,54	
P Values	0,9	0,85	0,38	0,002	
Dispersion	301,8				
residuals	-13,97	-13,22	-8,96	-12,41	-0,44
regParam	0,6				
Coefficients	0,5	-0,37	-5,53		
Intercept	39,62				
Coefficient Standard Errors	0,43	1,8	1,97	10,1	
T Values	1,16	-0,2	-2,81	3,92	
P Values	0,26	0,84	0,01	0	
Dispersion	194,6				
residuals	-8,42	-4,47	-5,89	12,24	1,29

Decision Tree Regression			
CrossValidator			
maxDepth	2	5	10
RMSE	7,24		

CrossValidator			
maxDepth	3	6	9
RMSE	5,39		
CrossValidator			
maxDepth	5	7	12
RMSE	5,42		

For the linear regression, as a result, the cross validator gives us a better rmse in the first configuration than in the second.
Without the cross validator with settings of regParam 0.6 and elasticNetParam 0.9 you get better results like the number of iterations to find the optimal coefficients for the model.
For the Generalized Linear Regression, the cross validator also returns the best rmse for the first configuration.
Without the cross validation, the configuration with regParam 0.6 seems to have better indicators, for example, although the dispersion is high in both configurations, it is lower in this one, as a sign of less variability than would be expected.
In the case of Decision Tree Regression, the results of the cross validation gave us that the second configuration presents a less rmse for the model.
When comparing the three methods, we decided that Decision Tree Regression can obtain better results to predict age.