

7CCSMBDT Big Data Technologies

Coursework 2

Due: Monday 29 March 2021 (23:59 UK time)

OVERVIEW:

This coursework aims to make you familiar with the following techniques:

- i. **MongoDB:** source-available cross-platform document-oriented database program. It handles NoSQL databases of Document type, which stores documents comprised of tagged elements, like JSONs.
- ii. **Apache Spark:** open-source unified analytics engine for large-scale data processing. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. Apache Spark arises to solve MapReduce's limitation for interactive data mining.
- iii. **Apache Hive:** data software warehouse project built on top of Apache Hadoop for providing data query and analysis. Hive gives an SQL-like interface to query data stored in various databases and file systems that integrate with Hadoop. Hive emerges as a solution to the main limitation of MapReduce, its low-level programming model, which requires developers to write custom programs which are hard to maintain and reuse.

TASK 1: mongoDB queries [total marks 30]:

Load the file *championsleague_1.json* (available in KEATS) into a database *cw2* and a collection *cl*. Write a single query, for each subtask 1 to 3 below.

The first thing we have to do is to drop the *championsleague_1.json* file into our *cw2* database. In order to do so, first we have to create this database and run it. We will create a directory called *cw2* in the *data* folder of MongoDB. After that we start running MongoDB with the specified database path. This is how it looks like in the terminal:

[illegible]

This terminal must be left open for this part of the coursework.

Now that we have MongoDB running, we are in position of dropping the document (data we will work with) in this folder. As I have installed the version 4.4 of MongoDB, I have to call *mongoimport* in a folder called *Tools* ¹. Once we are in this folder, we can call the *mongoimport* function:

```
mongoimport <options> <connection-string> <file>
```

This is how it looks in the terminal:

```
Simbolo del sistema
Microsoft Windows [Versión 10.0.19042.870]
(c) 2020 Microsoft Corporation. Todos los derechos reservados.

C:\Users\Asus>cd "C:\Program Files\MongoDB\Tools\bin"

C:\Program Files\MongoDB\Tools\bin>mongoimport --db cw2 --collection cl --drop --file "C:\Users\Asus\Desktop\KCL\Semester 2\Big Data Technologies\Courseworks\Coursework 2\championsleague_1.json"
2021-03-25T17:32:34.844+0100   connected to: mongodb://localhost/
2021-03-25T17:32:34.879+0100   dropping: cw2.cl
2021-03-25T17:32:35.466+0100   23285 document(s) imported successfully. 0 document(s) failed to import.
```

As we can see, we have specified *cw2* as the database, *cl* as the collection, and the file where I have saved the *.json* document as the file.

Once we have the data loaded in our database, we have to start running MongoDB. We do so in the following manner:

```
Simbolo del sistema - mongo
2021-03-25T17:32:35.466+0100   23285 document(s) imported successfully. 0 document(s) failed to import.

C:\Program Files\MongoDB\Tools\bin>cd ..
C:\Program Files\MongoDB\Tools\bin>cd ..
C:\Program Files\MongoDB\Tools\bin>cd ..
C:\Program Files\MongoDB>dir
El volumen de la unidad C es OS
El número de serie del volumen es: 3018-587F

Directorio de C:\Program Files\MongoDB
04/03/2021  01:36    <DIR>          .
04/03/2021  01:36    <DIR>          ..
04/03/2021  00:38    <DIR>          Server
04/03/2021  01:36    <DIR>          Tools
               0 archivos             0 bytes
               4 dirs  378.870.908.320 bytes libres

C:\Program Files\MongoDB>cd Server\4.4\bin
C:\Program Files\MongoDB\Server\4.4\bin>mongo
MongoDB shell version v4.4
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id": "UUID('337f1719-ec81-490a-9018-ff0990725ca') " }
MongoDB server version: 4.4.4

The server generated these startup warnings when booting:
  2021-03-25T16:35:45.217+01:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted

  Enable MongoDB's free cloud-based monitoring service, which will then receive and display
  metrics about your deployment (disk utilization, CPU, operation statistics, etc.).

  The monitoring data will be available on a MongoDB website with a unique URL accessible to you
  and anyone you share the URL with. MongoDB may use this information to make product
  improvements and to suggest MongoDB products and deployment options to you.

  To enable free monitoring, run the following command: db.enableFreeMonitoring()
  To permanently disable this reminder, run the following command: db.disableFreeMonitoring()

> show dbs
admin      0.000GB
config     0.000GB
cw2        0.000GB
local      0.000GB
test       0.000GB
> use cw2
switched to db cw2
> show collections
cl
> db.cl.find()
[ { "_id" : ObjectId("605cb228c36060925ba948"), "displayName" : "Adrian Nikko", "friendsCount" : 262, "followersCount" : 2078, "statusesCount" : 28900, "verb" : "share", "postedTime" : "2015-05-06T02:30:49.000Z", "body" : "RT @Siaranbola
Live: Semifinal 2012: Ada Chelsea diremehkan, Semifinal MCL 2015: Ada Juventus. Akanah Moment Berulang? #hoknow http://t.co/AXkh", "verified" : "false" },
  { "_id" : ObjectId("605cb228c36060925ba949"), "displayName" : "Luis Mucano", "friendsCount" : 134, "followersCount" : 37, "statusesCount" : 184, "verb" : "share", "postedTime" : "2015-05-06T02:30:51.000Z", "body" : "RT @Juventus_VEN:
#COPAD ASQUOTUD El #Stadium ingresó 3.305.132 € por el #JuveReal, que recibió a 41.041 fanáticos MCL http://t.co/AXkh", "verified" : "false" },
  { "_id" : ObjectId("605cb228c36060925ba94a"), "displayName" : "meco", "friendsCount" : 638, "followersCount" : 622, "statusesCount" : 9200, "verb" : "share", "postedTime" : "2015-05-06T02:30:48.000Z", "body" : "RT @RobertoSuarez0k: #
Barcelona vs. #BayerMunichVoleta. Mariano Closs / Comenta. @diatorreVneChampionsFOX#Forneos#13.45 ColVdale RT htt", "verified" : "false" },
  { "_id" : ObjectId("605cb228c36060925ba94b"), "displayName" : "Aay", "friendsCount" : 356, "followersCount" : 657, "statusesCount" : 12006, "verb" : "share", "postedTime" : "2015-05-06T02:30:51.000Z", "body" : "RT @elpieroale: StillMay
always a great date to play on... Good luck boys @Juventusfcen @ChampionsLeague #J0006 MCL #2018", "verified" : "false" },
  { "_id" : ObjectId("605cb228c36060925ba94c"), "displayName" : "elio", "friendsCount" : 655, "followersCount" : 810, "statusesCount" : 8255, "verb" : "share", "postedTime" : "2015-05-06T02:30:49.000Z", "body" : "RT @i_soccer: This Car
```

As we can see, we have run *mongo* in the folder where it is saved, *bin*. Once we are in the mongo shell, we have switched to the database we are interested in (*cw2*) and we have opened the collection we want to run the queries on (*cl*). When we use *.find()* on our collection, we see all the documents in the *championsleague_1.json* file.

As we can see, there are different attributes for each entry: *_id*, *displayName*, *friendsCount*, *followersCount*, *statusesCount*, *verb*, *postedTime*, *body* and *verified*.

Now we are in position of performing the three subtasks of this first task.

¹ <https://docs.mongodb.com/database-tools/mongoimport/>

1.1 Write a query that returns the name, number of followers, and number of friends, of each user with fewer than 25 friends, whose name starts with “A” (case insensitive) and ends with “es” (case sensitive). The results must be sorted in decreasing order of displayName. [10]

To perform this subtask, we have to apply `.find()`, and pass some parameters to this method. I am going to put first the query we will apply and then explain it step by step:

```
1 db.cl.find(
2     { $and: [
3         { friendsCount: { $lt: 25 } },
4         { displayName: { $regex: /^A/i } },
5         { displayName: { $regex: /es$/ } }
6     ]
7 },
8     { _id:0, displayName:1, followersCount:1, friendsCount:1 }
9 ).sort( { displayName: -1 } )
```


As we can see in line 1, we apply the `.find()` method to the collection called `cl` from the database we are using, `cw2`. The question demands us to return some elements that satisfy multiple conditions, which explains the operator `$and` in line 2. Those three conditions are the following:

1. Users with fewer than 25 friends: this value is stored in the attribute `friendsCount`, and the `$lt` operator means lesser than, line 3.
2. Users whose name starts with “A” (case insensitive): this value is stored in the attribute `displayName`. To specify these elements we use the `®ex` evaluation operator. ‘^’ means that it starts with ‘A’, and the ‘i’ option is for the case insensitivity. This can be seen in line 4.
3. Users whose name ends with “es” (case sensitive): this value is also stored in the attribute `displayName`. To specify the corresponding elements we use `$regex` too, in this case we set ‘\$’ at the end, which means ends with ‘es’. There is no need to specify any option if we want the search to be case sensitive. This can be seen in line 5.

As the question description asks us, we have to return the name, number of followers, and number of friends. This can be specified through a second clause in the `.find()` method, it can be observed in line 8. `_id:0` so this is field not outputted.

Lastly, in line 9, we sort the results in descending order of `displayName`. This is specified by setting the value to `-1` in the `.sort()` method.

After executing the command shown above, we find out that there are three users that satisfy all the conditions: **“angie torres”**, **“Arizona Companies”**, and **“Adejies”**. The results are displayed below:



```
Symbol del sistema - mongo
> use cw2
> db.cl.find(
  { $and: [
    { friendsCount: { $lt: 25 } },
    { displayName: { $regex: /^A/i } },
    { displayName: { $regex: /es$/ } }
  ],
  { _id: 0, displayName: 1, followersCount: 1, friendsCount: 1 }
).sort( { displayName: -1 } )
{ "displayName": "angie torres", "friendsCount": 23, "followersCount": 33 }
{ "displayName": "angie torres", "friendsCount": 23, "followersCount": 23 }
{ "displayName": "Arizona Companies", "friendsCount": 0, "followersCount": 10 }
{ "displayName": "Adejies", "friendsCount": 13, "followersCount": 10 }
{ "displayName": "Adejies", "friendsCount": 13, "followersCount": 10 }
{ "displayName": "Adejies", "friendsCount": 13, "followersCount": 10 }
```

1.2 Write a query that returns the average ratio between number of followers and number of friends, over all documents. Note that the number of friends (denominator in the ratio) must be >0 for the ratio to be defined. [10]

This subtask asks us to calculate the ratio between `followersCount` and `friendsCount` of each document, and then average all the resulting ratios. To do this correctly, we have to exclude from the analysis the documents that have 0 `friendsCount` in order to avoid errors.

This subtask is a little more complex, as before, I will show the query used and then explain what it does step by step:

```

1  db.cl.aggregate( [
2      { $match: { friendsCount: { $gt: 0 } } },
3      { $addFields: {
4          ratio:
5              { $divide:
6                  ["$followersCount","$friendsCount"]
7              } } },
8      { $group: {
9          _id: null,
10         avgRatio: { $avg: "$ratio" }
11     } }
12 ] )

```

This time we have applied `.aggregate()`, which calculates aggregate values or the data in a collection, specifically in our case, in the collection `cl`. As we said in the first paragraph, we have to filter our collection to return the averages of the documents which have at least 1 friend. This is what is done with the `$match` operator in line 2.

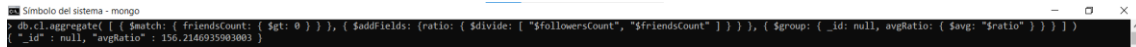
After that, we use `$addFields`, which adds new fields to documents. It outputs documents that contain all existing fields from the input documents, and the newly added fields. In our case, we are adding a new field called `ratio` (line 3), which is the division of `followersCount` by `friendsCount` (line 4), which is done due to the `$divide` operator (line 5). This will not throw an error because we have only matched the documents that have values greater than 0 in the `friendsCount` attribute.

The last step that has to be done to obtain the overall ratio, the average ratio of all documents, is done through the `$group` operator in line 8. This operator groups input documents by the specified `_id` expression and for each distinct groupings, outputs a document. As we can see in line 9, we are using `_id: null`, which is similar to saying “group by all elements”, meaning that we want the overall average of all input documents. The other parameter is the field we want to average, which is computed by the accumulator operator `$avg` (line 10). We are calculating the average of `ratio`, which was defined in the previous step.

The result given by this query is:

```
{ "_id" : null, "avgRatio" : 156.2146935903003 }
```

On average, each user has 156 followers for each friend. This has to be analysed carefully, as each user appears in more than one document. The results displayed by the terminal are shown like this:



```

Simbolo del sistema - mongo
> db.cl.aggregate([ { $match: { friendsCount: { $gt: 0 } } }, { $addFields: { ratio: { $divide: [ "$followersCount", "$friendsCount" ] } } }, { $group: { _id: null, avgRatio: { $avg: "$ratio" } } } ] )
{ "_id" : null, "avgRatio" : 156.2146935983803 }

```

1.3 Write a query that returns the number of users who have at least 1000 friends, posted a tweet (the field verb has the value “post”) and whose tweet (field body) contains the string “Madrid” (even as part of a word). [10]

This is quite similar to the first subtask. We have to perform the `.find()` method alongside with `.count()`, so the query returns the number of users. For clarification purposes, I have read in a discussion in KEATS that we can make the following assumption: “each document corresponds to a different user”. This way, we can find the documents which satisfy the corresponding conditions, and then count them.

Just like before, I will show the query applied to later explain it step by step:

```

1 db.cl.find( {
2     $and: [
3         { friendsCount: { $gte: 1000 } },
4         { verb: "post" },
5         { body: { $regex: /Madrid/ } }
6     ]
7 }
8 ).count()

```

As we can see in [line 1](#), we apply the `.find()` method to the collection called `cl` from the database we are using, `cw2`. The question demands us to return the count of documents that satisfy multiple conditions, which explains the operator `$and` in [line 2](#). Those three conditions are the following:

1. Users with at least 1000 friends: this value is stored in the attribute `friendsCount`, and the `$gte` operator is for greater than or equivalent, [line 3](#).
2. Users who have posted a tweet: this value is stored in the attribute `verb`. This attribute has two distinct values: “post” and “share”. We have to match the documents where the `verb` attribute is equal to “post”. As the question description does not say nothing about case sensitivity, we will assume it is a case-sensitive match. Anyway, the value in `verb` is stored literally like “post”. This be seen in [line 4](#).
3. Documents whose tweet (field body) contains the string “Madrid” (even as part of a word): this value is stored in the attribute `body`. To find the corresponding documents we use `$regex`. Just like before, we assume the search is case-sensitive, so there is no need to specify any option (seen in KEATS). This can be seen in [line 5](#).

Lastly, once we have matched all the documents that satisfy this criteria we are able to `.count()` ([line 8](#)) the documents that this search returns. What we see through the terminal is the following:

Manuel Ortiz Cachá (k20114995)

```

Simbolo del sistema - mongo
db.cl.find( {&id: { $friendsCount: { $gte: 1000 } }, (verb: "post"), (body: { $regex: /Madrid/ }) } ], { _id: 0, displayName: 1, friendsCount: 1, verb: 1 } )
{"_id": 1, "displayName": "Real Madrid Klan", "friendsCount": 1577, "verb": "post", "body": "\u00c9Madridistas! Cr\u00f3nica de #JuveVsReal \u201eNessus dorma\u201c http://t.co/42NgDYAVkK @unionmerengue por @missmarta_rn #realmadrid #CL #RealMadrid #UCL"}
{"_id": 2, "displayName": "Real Madrid Klan", "friendsCount": 1577, "verb": "post", "body": "\u00c9Madridistas! Cr\u00f3nica de #JuveVsReal \u201eNessus dorma\u201c http://t.co/42NgDYAVkK @unionmerengue por @missmarta_rn #realmadrid #CL #RealMadrid #UCL"}
{"_id": 3, "displayName": "T\u00edtular", "friendsCount": 11249, "verb": "post", "body": "\u00c9VECCHIA SIGMORA!\u00c9Mira las mejores fotos de #Juventus vs #RealMadrid, por la #ChampionsLeague!http://t.co/rktVR2WqH8 http://t.co/cdSvXyV6"}
{"_id": 4, "displayName": "Chilevision", "friendsCount": 1497, "verb": "post", "body": "\u00c9Juventus de @Kingarturo23 venc\u00edo al Real Madrid en la semifinal de la #ChampionsLeague. Revisa los goles + http://t.co/CsFTGwBz"}
{"_id": 5, "displayName": "@ElBarro Morata News", "friendsCount": 1867, "verb": "post", "body": "\u00c9Juve super\u00f3 2-1 a Real Madrid en la Semifinal #ChampionsLeague ... http://t.co/nqqQ0pfcdC"}
{"_id": 6, "displayName": "Real Madrid Klan", "friendsCount": 1576, "verb": "post", "body": "\u00c9Madridistas! Cr\u00f3nica de #JuveVsReal \u201eNessus dorma\u201c http://t.co/42NgDYAVkK @unionmerengue por @missmarta_rn #realmadrid #CL #RealMadrid #UCL"}
{"_id": 7, "displayName": "Chilevis\u00f3n Noticias", "friendsCount": 1495, "verb": "post", "body": "\u00c9Juventus de Arturo Vidal venc\u00edo al Real Madrid en la semifinal de la #ChampionsLeague + http://t.co/Aqu4BBg0B3 http://t.co/coByxsfGSPV"}
{"_id": 8, "displayName": "FUTBOL MUNDIAL", "friendsCount": 37810, "verb": "post", "body": "\u00c9La Juve derrot\u00f3 2-1 al Real y avanza en la ida de semifinales #Juventus #RealMadrid #UCL #ChampionLeague http://t.co/f0LgTob30r"}
{"_id": 9, "displayName": "DHE domains", "friendsCount": 2082, "verb": "post", "body": "\u00c9http://t.co/QcKzbew2 @ air! #Madrid #Barcelona London #Paris #Berlin #socialmedia http://t.co/FfmgG8KjW - 06.05.14:57"}
{"_id": 10, "displayName": "Real Madrid Klan", "friendsCount": 1576, "verb": "post", "body": "\u00c9Madridistas! Cr\u00f3nica de #JuveVsReal \u201eNessus dorma\u201c http://t.co/42NgDYAVkK @unionmerengue por @missmarta_rn #realmadrid #CL #RealMadrid #UCL"}
{"_id": 11, "displayName": "Real Madrid Klan", "friendsCount": 1576, "verb": "post", "body": "\u00c9Madridistas! Cr\u00f3nica de #JuveVsReal \u201eNessus dorma\u201c http://t.co/42NgDYAVkK @unionmerengue por @missmarta_rn #realmadrid #CL #RealMadrid #UCL"}
{"_id": 12, "displayName": "Football Freak", "friendsCount": 1024, "verb": "post", "body": "\u00c9Benzema: \u201cYes we can comeback. We are Real Madrid.\u201c #RealMadrid #LaLiga #UCL #Benzema"}
{"_id": 13, "displayName": "Judith", "friendsCount": 2002, "verb": "post", "body": "\u00c9http://t.co/SwPZZuM75 #EuroJuve #RealMadrid #inAllaLine grazie ragazzi... #amosas a Madrid http://t.co/FC9k4j31l"}
{"_id": 14, "displayName": "HD Xmas online", "friendsCount": 4133, "verb": "post", "body": "\u00c9RT @Twitterufes:el 'Swiss Express' quiere repetir en Madrid http://t.co/gdRGS3KA4 #JuveReal #UCL http://t.co/xDEZCY3g36"}
{"_id": 15, "displayName": "Livestream80", "friendsCount": 1987, "verb": "post", "body": "\u00c9Juventus 2-1 Real Madrid - All Goals & Highlights 2015 [VIDEO HD]#Juventus #RealMadrid #JuveReal #UCL #yn1nkl: http://t.co/xk2d0x0JU"}
{"_id": 16, "displayName": "Soy_Leyenda389", "friendsCount": 10673, "verb": "post", "body": "\u00c9Juventus de @Kingarturo23 venc\u00edo al Real Madrid en la semifinal de la #ChampionsLeague. Revisa los goles + http://t.co/RELE3Y0Y9"}
{"_id": 17, "displayName": "Real Madrid Klan", "friendsCount": 1576, "verb": "post", "body": "\u00c9Madridistas! El post-partido: Juventus 2 vs 1 Real Madrid. \u201eNessus dorma\u201c http://t.co/42NgDYAVkK @unionmerengue @ #realmadrid #CL #RealMadrid #UCL"}
{"_id": 18, "displayName": "Real Madrid Klan", "friendsCount": 1576, "verb": "post", "body": "\u00c9Madridistas! Cr\u00f3nica de #JuveVsReal \u201eNessus dorma\u201c http://t.co/42NgDYAVkK @unionmerengue por @missmarta_rn #realmadrid #CL #RealMadrid #UCL"}
{"_id": 19, "displayName": "Kevin Enrique Mu\u00f1ez", "friendsCount": 2165, "verb": "post", "body": "\u00c9En el Bernab\u00e9u, hoy se defini\u00f3 todo. Real Madrid vs. Juventus. #UEFAChampionsLeague"}
{"_id": 20, "displayName": "DHE domains", "friendsCount": 2002, "verb": "post", "body": "\u00c9http://t.co/QcKzbew2 @ air! #Madrid #Barcelona London #Paris #Berlin #socialmedia http://t.co/FogXOXI3b --- 06.05.14:57"}
Type "it" for more
db.cl.find( {&id: { $friendsCount: { $gte: 1000 } }, (verb: "post"), (body: { $regex: /Madrid/ }) } ),count()
185

```

As it can be seen in the screenshot, first we check if the `.find()` method returns the correct documents. Then, we count the documents returned from the first query. There are **185** documents that satisfy the previous conditions.

TASK 2: Apache Spark [total marks 40]:

Output all occupations that are performed by users in the age group[40, 50) and by users in the age group [50, 60) and are among the 10 most frequent occupations for the users in each age group.

This task aims to make use of Apache Spark commands to see Spark's data structures, like RDDs. RDD stands for Resilient Distributed Datasets, they are data structures that serve as the core unit of data for Apache Spark. From the user's side, he/she can manipulate it, control its partitioning, or make it persistent in memory, which avoids the need of writing data in disk, so it is much more efficient. They also enable efficient data reuse.

We will start by copying the code already given in the coursework description:

```
*task2.py X
# We import the requested libraries:
from pyspark import SparkContext
from operator import add

# SparkContext is the entry point to any spark functionality, to start the driver program we have to
# initialise a SparkContext object. We need to create SparkContext only in programs, pyspark shell does
# this automatically. In the following line Master is 'local' (URL of the cluster it connects to), and
# appName is 'pyspark' (name of our job):
sc = SparkContext('local', 'pyspark')

# This function, age_group, returns a string with the age range given a certain age:
def age_group(age):
    if age < 10:
        return '0-10'
    elif age < 20:
        return '10-20'
    elif age < 30:
        return '20-30'
    elif age < 40:
        return '30-40'
    elif age < 50:
        return '40-50'
    elif age < 60:
        return '50-60'
    elif age < 70:
        return '60-70'
    elif age < 80:
        return '70-80'
    else:
        return '+80'

# This function, parse_with_age_group, gets the different values of u.user (user id, age, gender,
# occupation and zipcode) and returns different variables (userid, age_group, gender, occupation,
# zip and age):
def parse_with_age_group(data):
    userid, age, gender, occupation, zip = data.split("|") # creates a list with the different attributes.
    return userid, age_group(int(age)), gender, occupation, zip, int(age) # returns the attributes in this order.
```

As we can see, after importing the requested libraries, we initialise *SparkContext* by creating an object called *sc*. We need to create *SparkContext* in the program, while the *pyspark* shell does this automatically.

After that, two functions are defined:

1. *age_group()*: this function returns a string describing the group age a certain age is from.
2. *parse_with_age_group()*: this function parses the data given in 'u.users.txt' and returns the values of the different attributes given in the data. It returns six distinct elements: *userid*, *age group*, *gender*, *occupation*, *zip code* and *age*.

Now we are in position of writing our program to find the top-10 most frequent occupations that overlap in the '40-50' and '50-60' groups. We will do this in five different steps:

1. Read and parse the data: we do so by helping ourselves with the functions created above, in the first screenshot.

```
# First of all, we will create an RDD (Resilient Distributed Datasets), which enables efficient data
# reuse. An RDD is a data structure that serves as the core unit of data for Apache Spark. From the
# user's side, he can manipulate it, control its partitioning, or make it persistent in memory, which
# avoids the need of writing data in disk.

# We populate the RDD with the built-in function 'sc.textFile', which creates an RDD based on a .txt
# file:
data = sc.textFile("file:///home/cloudera/Desktop/cw2/task2/u.user.txt") # as parameter you specify the path to the .txt file.

# Now we convert our RDD in a new one with the age_group in it:
users = data.map(parse_with_age_group)
```

As we can see, `.textFile()` creates an RDD based on a .txt file, which in our case is the text file which contains the different users, `u.user.txt`. After that, we map the RDD to create a new one with the information returned in the `parse_with_age_group` function. `.map()` is a transformation operation that creates a new RDD with the function passed by as a parameter.

2. Create an RDD for each age group ('40-50' and '50-60'): this is done with the `.filter()` transformation operation. It returns another RDD if it satisfies the condition passed as a parameter. In our case, if the string '40-50' is part of the entry, it passes that value to the new RDD, `forties`. The same is done for `fifties`.

```
# Next, we create a group for users in their forties and other with users in their fifties, we do so by
# using .filter():
forties = users.filter(lambda x: '40-50' in x) # each entry in 'users' is a list, so, if '40-50' or '50-60' is an
fifties = users.filter(lambda x: '50-60' in x) # element of that list, it creates a new RDD with those values.
```

3. Obtain the top-10 occupations of each group ('40-50' and '50-60'): this is done with the function `countByValue()`, which returns the count of each unique value in this RDD as a dictionary of (value, count) pairs. So, we will apply this function to the 4th element of the `forties` and `fifties` RDDs, the occupation.

```
# Now we have to compute their frequencies to get the most common occupations. The function .countByValue()
# returns a dictionary with the form KEY ('occupation')-VALUE (frequency). We only care about the 4th position,
# its occupation, that is why we map only that value.
occ_freqs_forties = forties.map(lambda x: x[3]).countByValue()
occ_freqs_fifties = fifties.map(lambda x: x[3]).countByValue()
```

4. Create two new RDDs with this new information: now that we have dictionaries with the counts of each occupation, we can store the top-10 most frequent occupations by sorting the dictionary by its value and getting the 10 first keys. This can be done easily with Python built-in functions like `sorted()`. We set the parameter `reverse` to `True` to obtain a list in descending order, having first the most frequent jobs. After that, we create a new RDD with the `.parallelize()` method, which creates an RDD of the array you pass to him as a parameter. As we can see, we only pass the 10 first values.

```
# Now, we create two new RDDs with the top-10 most frequent occupations in each age group. We will benefit from
# the 'sorted' function in Python. We will sort in descending order (reverse=True) the dictionary created in
# the previous step and get only the keys. After that, we select the 10 first elements:
top_10_forties = sc.parallelize(sorted(occ_freqs_forties, key = occ_freqs_forties.get, reverse = True)[:10])
top_10_fifties = sc.parallelize(sorted(occ_freqs_fifties, key = occ_freqs_fifties.get, reverse = True)[:10])
```

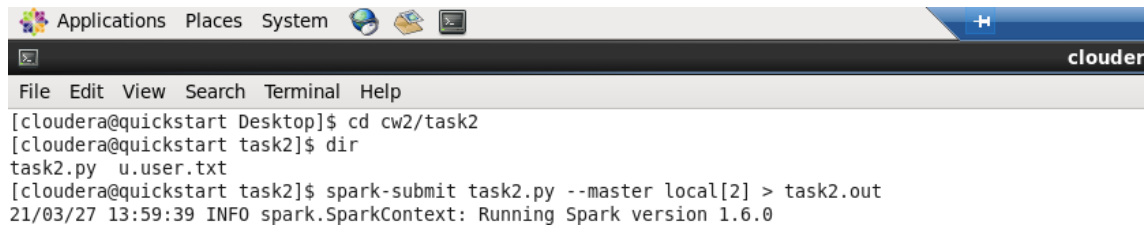
5. Intersect the two previous top-10 occupations per age group to obtain the common occupations: this can be easily handled with the `.intersection()` function, which returns the common elements from two different RDDs. The final result will be stored in an RDD called `common_top10_occs`.

```
# Once we have two RDDs, we only need their intersection, as the question description asks us to output all
# occupations that are performed by these two groups and are among the 10 most frequent occupations for the
# users in each group. This can be done easily with the function .intersection()
common_top10_occs = top_10_forties.intersection(top_10_fifties)
```


Once we have the final result stored in this new RDD, we are in position of printing it, which we will do with `.collect()`, which returns the values of an RDD. This step is straightforward:

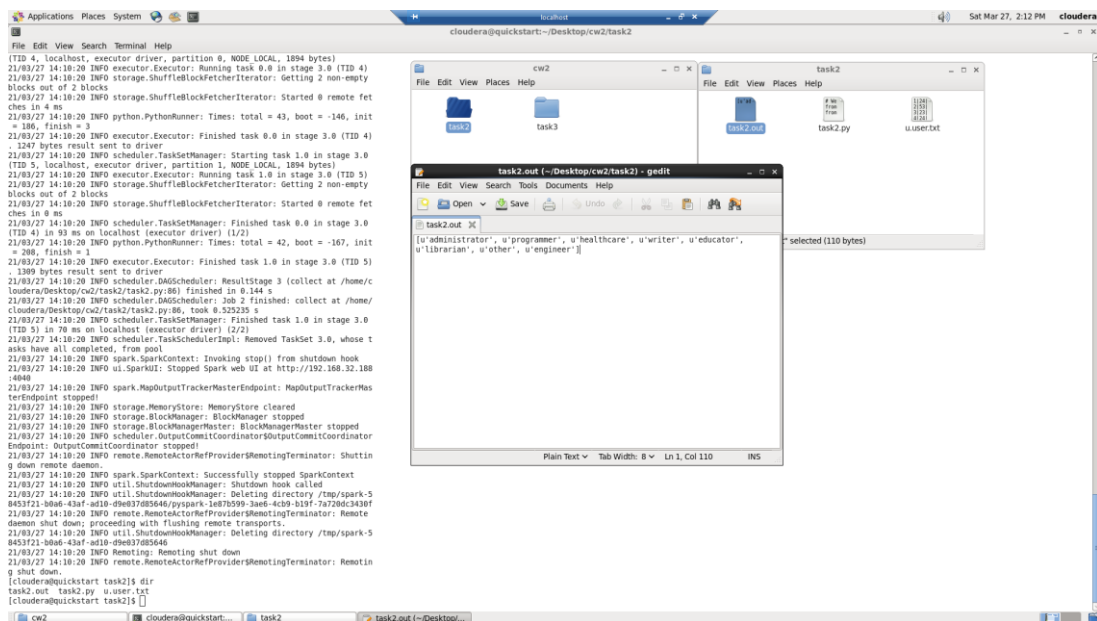
```
# Lastly, we print the final results.  
print common_top10_occs.collect()
```

Once we have our programme correctly coded, we can execute it through the terminal and store the output in another file, which we will call `task2.out`. This is done in the following manner: first we open the folder where we have the data (.txt) and the program (.py), then we execute the programme and store the output in another file, `task2.out`.



```
cloudera@quickstart Desktop]$ cd cw2/task2  
cloudera@quickstart task2]$ dir  
task2.py  u.user.txt  
cloudera@quickstart task2]$ spark-submit task2.py --master local[2] > task2.out  
21/03/27 13:59:39 INFO spark.SparkContext: Running Spark version 1.6.0
```

As we can see in the terminal, the program is being executed with the Spark version 1.6.0. We have specified the number of worker threads we want to start the shell with by passing 2 as a parameter of `master` local. Once the program has finished, we should have a new file called `task2.out` in our folder. Here we can see that this is correct:



As we can see at the last line of the terminal, a new file has been created, called `task2.out`, and the content of this file is displayed too. The common occupations from the top-10 most frequent jobs in the age groups '40-50' and '50-60' are the following:

- **Administrator**
- **Programmer**
- **Healthcare**
- **Writer**
- **Educator**
- **Librarian**
- **Other**
- **Engineer**

TASK 3: HIVE [total marks 30]:

Download the file *query_logs.txt* from KEATS. This file contains searches made by different users. Specifically, there are three tab-separated attributes *user*, *time* and *query*. Create a database *log_db* and a table *logs* that has the attributes *user*, *time*, *query*, all of which are of type *STRING*. Write a query, for each subtask 1 to 3 below.

The first thing we have to do is to create the database *log_db* and the table *logs*. The first step is to open a terminal and type *hive* to start this task of the coursework. I will leave the screenshot of my terminal and then explain what I have done:



```
Applications Places System cloudera@quickstart:~/Desktop
File Edit View Search Terminal Help
[cloudera@quickstart Desktop]$ hive
Logging initialized using configuration in jar:file:/usr/lib/hive/lib/hive-common-1.1.0-cdh5.13.0.jar!/hive-log4j.properties
WARNING: Hive CLI is deprecated and migration to Beeline is recommended.
hive>
> SHOW DATABASES;
OK
default
mydb
Time taken: 0.012 seconds, Fetched: 2 row(s)
hive>
> CREATE DATABASE log_db;
OK
Time taken: 0.009 seconds
hive>
> SHOW DATABASES;
OK
default
log_db
mydb
Time taken: 0.015 seconds, Fetched: 3 row(s)
hive>
> USE log_db;
OK
Time taken: 0.019 seconds
hive>
> SHOW TABLES;
OK
Time taken: 0.019 seconds
hive>
> CREATE TABLE logs (user STRING, time STRING, query STRING) COMMENT 'Query details' ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' LINES TERMINATED BY '\n' STORED AS TEXTFILE;
OK
Time taken: 0.138 seconds
hive>
> SHOW TABLES;
OK
logs
Time taken: 0.014 seconds, Fetched: 1 row(s)
hive>
> LOAD DATA LOCAL INPATH '/home/cloudera/Desktop/cw2/task3/query_logs.txt' OVERWRITE INTO TABLE logs;
Loading data to table log_db.logs
Table log_db.logs stats: [numFiles=1, numRows=0, totalSize=204599, rawDataSize=0]
OK
Time taken: 0.583 seconds
hive>
```

As we can see, there is no database called *log_db* before the *CREATE* statement. This statement is used to create a database in Hive. A database in Hive is a namespace or a collection of tables. We create a database with this name, and we select that database to carry on with the coursework: *USE log_db*. As we can see, now this database does exist.

After that, we create a table in this database, we have called the table *logs* and it has three attributes: *user*, *time* and *query*, all of them of type *STRING*. We also specify how are the rows and lines formatted. Each attribute is separated by a tab (*\t*), and each line is separated by a new line (*\n*). Lastly, we specify that the data will be loaded from a *.txt* file. As we can see, when we first execute *SHOW TABLES*, this collection does not appear, but later it does.

The last step we have to accomplish to have the data from *query_logs.txt* is to load it in the table we just created. This is done in the last command, *LOAD DATA LOCAL INPATH*. We specify the path to the text file with the data and we overwrite it in the table *logs*. We use the parameter *LOCAL* to load the data from the local filesystem.

Now we are in position of performing the three subtasks of this third task. Before starting with these subtasks, I am going to show some characteristics of our table. This is done with the *DESCRIBE FORMATTED* command, which shows the details of a specific table in a readable format:

```

hive> DESCRIBE EXTENDED log_db.logs;
OK
# col_name      data_type      comment
user            string
time            string
query           string

Detailed Table Information
Table(tableName=log_db, dbName=log_db, owner=cloudera, createTime=1616868997, lastAccessTime=0, retention=0, sd=StorageDescriptor(cols=[FieldSchema(name=user, type=string, comment=null), FieldSchema(name=time, type=string, comment=null), FieldSchema(name=query, type=string, comment=null)], location=hdfs://quickstart.cloudera:8020/user/hive/warehouse/log_db.db/logs, inputFormat=org.apache.hadoop.mapred.TextInputFormat, outputFormat=org.apache.hadoop.hive ql.io.HiveIgnoreKeyTextOutputFormat, compressed=false, numBuckets=1, serdeInfo=SerDeInfo(name=null, serializationLib=org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, parameters={serialization.format=1, line.delim=, field.delim=}), bucketCols=[], sortCols=[], parameters={skewedInfo=SkewedColumnInfo(skewedColNames=[], skewedColValues=[], skewedColLocationsMaps=[]), storedSubDirectories=false, partitionKeys=[], parameters={numFiles=1, transientLastDdlTime=1616868974, COLUMN_STATS_ACCURATE=true, totalSize=204599, numRows=0, comment=query details, rawDataSize=0}, viewOriginalText=null, viewExpandedText=null, tableType=MANAGED TABLE)
Time taken: 0.099 seconds, Fetched: 0 row(s)
hive> DESCRIBE FORMATTED log_db.logs;
OK
# col_name      data_type      comment
user            string
time            string
query           string

# Detailed Table Information
Database:       log_db
Owner:         cloudera
CreateTime:     Sat Mar 27 16:01:37 GMT 2021
LastAccessTime: UNKNOWN
Protect Mode:   None
Retention:      0
Location:       hdfs://quickstart.cloudera:8020/user/hive/warehouse/log_db.db/logs
Table Type:    MANAGED TABLE
Table Parameters:
  COLUMN_STATS_ACCURATE      true
  comment                    query details
  numFiles                   1
  numRows                    0
  rawDataSize                0
  totalSize                  204599
  transientLastDdlTime       1616868974

# Storage Information
SerDe Library:      org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe
InputFormat:        org.apache.hadoop.mapred.TextInputFormat
OutputFormat:       org.apache.hadoop.hive ql.io.HiveIgnoreKeyTextOutputFormat
Compressed:         No
Num Buckets:        1
Bucket Columns:     []
Sort Columns:       []
Storage Desc Params:
  field.delim          \t
  line.delim           \n
  serialization.format  1
Time taken: 0.091 seconds, Fetched: 36 row(s)

```

3.1 Write a query that returns the maximum number of visits of all users. Note that in each page visit a user may or may not issue a query. [7]

To perform this subtask, we need to find the user that has issued the greatest number of queries, and return the number of queries he/she issued. From the discussion forum in KEATS I have read about an example that is very explanatory: “if we have 3 users, one performs 3 visits, another performs 4, and the third user performs 5 visits, the query should return just the number 5”.

We will do this by executing an intermediate step that creates a table which will contain the number of visits each user does. I will show first the screenshot, and then explain what is done in each step:

```

hive>
> SHOW TABLES;
OK
logs
Time taken: 0.014 seconds, Fetched: 1 row(s)
hive>
> CREATE TABLE visits (user STRING, visits INT);
OK
Time taken: 0.113 seconds
hive>
> SHOW TABLES;
OK
logs
visits
Time taken: 0.014 seconds, Fetched: 2 row(s)
hive>
> INSERT INTO visits SELECT user, COUNT(time) FROM logs GROUP BY user;
Query ID = cloudera.20210329134242_069acel-9ae5-46a0-854c-c3caddce7b90
Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks not specified. Estimated from input data size: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapreduce.job.reducers=<number>
Starting Job = Job_1617019817826_0001, Tracking URL = http://quickstart.cloudera:8088/proxy/application_1617019817826_0001/
Kill Command = /usr/lib/hadoop/bin/hadoop job -kill job_1617019817826_0001
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 1
2021-03-29 13:42:49,652 Stage-1 map = 0%, reduce = 0%
2021-03-29 13:42:56,190 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 1.4 sec
2021-03-29 13:43:03,520 Stage-1 map = 100%, reduce = 100%, Cumulative CPU 3.55 sec
MapReduce Total cumulative CPU time: 3 seconds 550 msec
Ended Job = Job_1617019817826_0001
Loading data to table log_db.visits
Table log_db.visits stats: [numFiles=1, numRows=891, totalSize=17051, rawDataSize=16160]
MapReduce Jobs Launched:
Stage-1: Map: 1 Reduce: 1 Cumulative CPU: 3.55 sec HDFS Read: 212900 HDFS Write: 17124 SUCCESS
Total MapReduce CPU Time Spent: 3 seconds 550 msec
OK
Time taken: 26.884 seconds
hive>

```

As we can see at the beginning, there is only one collection called *logs*, which is the one we created from *query_logs.txt*. We create another table called *visits* with the *CREATE TABLE* command, just like before. After that, we populate this table with the unique users (*GROUP BY user*) and the number of times they have visited the webpage (*COUNT(time)*). Therefore, in the table *visits* we have a user id, a *STRING*, and an *INT* that describes the number of visits each user has made.

The last step is quite straightforward, we just have to find the maximum value in the column *visits*, which can be done with a *SELECT* command and the *MAX* function. The user with the most visits issued in this database made a total of **78 visits**. The output of hive can be seen here:

```

hive> SELECT MAX(visits) FROM visits;
Query ID = cloudera_20210327180404_ffcc5d4d-9d42-46d0-b59c-0248db46d9b7
Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks determined at compile time: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapreduce.job.reduces=<number>
Starting Job = job_1616859667761_0016, Tracking URL = http://quickstart.cloudera:8088/proxy/application_1616859667761_0016/
Kill Command = /usr/lib/hadoop/bin/hadoop job -kill job_1616859667761_0016
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 1
2021-03-27 18:04:53,924 Stage-1 map = 0%, reduce = 0%
2021-03-27 18:05:01,192 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 1.17 sec
2021-03-27 18:05:09,558 Stage-1 map = 100%, reduce = 100%, Cumulative CPU 2.86 sec
MapReduce Total cumulative CPU time: 2 seconds 860 msec
Ended Job = job_1616859667761_0016
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1 Reduce: 1 Cumulative CPU: 2.86 sec HDFS Read: 24375 HDFS Write: 3 SUCCESS
Total MapReduce CPU Time Spent: 2 seconds 860 msec
OK
78
Time taken: 24.004 seconds, Fetched: 1 row(s)
hive>

```

3.2 Write a query that returns all attributes of each user who issues a query that contains the string “job”. Note that “job” may appear within a word (e.g., “job” is part of “jobs”). [8]

This subtask is simpler, it can be obtained with a single query. We have to select all entries where a user has issued a query related to “job”. Hence, we need to find the entries where the column *query* contains the string “job”. This can be done effectively with the *SELECT* command, filtering the entries we want with the *WHERE* clause.

In the *WHERE* clause we indicate that we want the entries that are *LIKE* ‘%job%’. The percent sign (%) represents zero, one or multiple characters. Therefore, when we put the percent sign before and after the string we are looking for, we mean that we want the column to contain that word; regardless of it is at the beginning, in between or at the final of a value.

```

cloudera@quickstart:~
File Edit View Search Terminal Help
hive>
> SELECT * FROM logs WHERE query LIKE '%job%';
Query ID = cloudera_20210329140909_ace0579a-dd93-4ea3-a2de-c84e5f29b167
Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks is set to 0 since there's no reduce operator
Starting Job = job_1617019817826_0004, Tracking URL = http://quickstart.cloudera:8088/proxy/application_1617019817826_0004/
Kill Command = /usr/lib/hadoop/bin/hadoop job -kill job_1617019817826_0004
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 0
2021-03-29 14:09:37,838 Stage-1 map = 0%, reduce = 0%
2021-03-29 14:09:45,236 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 1.76 sec
MapReduce Total cumulative CPU time: 1 seconds 760 msec
Ended Job = job_1617019817826_0004
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1 Cumulative CPU: 1.76 sec HDFS Read: 208988 HDFS Write: 778 SUCCESS
Total MapReduce CPU Time Spent: 1 seconds 760 msec
OK
4077443B5801F0C3      970916182623      job openings
4077443B5801F0C3      970916182752      job openings listings
4077443B5801F0C3      970916182823      agricultural job listings
4077443B5801F0C3      970916182834      agricultural job listings employdogst
4077443B5801F0C3      970916182942      job listings
83607290B8BEAFC6      970916070440      jobs=hong kong
D5D8220D36969861      970916222708      job interview tips
D5D8220D36969861      970916224215      job interview tips
0E10D08EB5EEB192      970916134219      jobs at the university of minnesota
567854C718273984      970916021936      part time jobs
567854C718273984      970916022012      part time jobs
567854C718273984      970916022043      part time jobs
567854C718273984      970916022117      part time jobs
567854C718273984      970916022202      part time jobs
567854C718273984      970916022313      home jobs
567854C718273984      970916022444      home jobs
Time taken: 15.801 seconds, Fetched: 16 row(s)
hive>

```

The query returns 16 different entries, which are shown below:

4077443B5801F0C3	970916182623	job openings
4077443B5801F0C3	970916182752	job openings listings
4077443B5801F0C3	970916182823	agricultural job listings
4077443B5801F0C3	970916182834	agricultural job listings employdogst

4077443B5801F0C3	970916182942	job listings
83607290B8BEAFC6	970916070440	jobs=hong kong
D5D8220D36969861	970916222708	job interview tips
D5D8220D36969861	970916224215	job interview tips
0E10DD8EB5EEB192	970916134219	jobs at the university of minnesota
567854C718273984	970916021936	part time jobs
567854C718273984	970916022012	part time jobs
567854C718273984	970916022043	part time jobs
567854C718273984	970916022117	part time jobs
567854C718273984	970916022202	part time jobs
567854C718273984	970916022313	home jobs
567854C718273984	970916022444	home jobs

3.3 Write a query that returns the number of distinct users who issue a (non-empty) query between 21:00:00 (inclusive) and 22:59:59 (inclusive). Note that the second attribute in `query_logs.txt` is in the form `yyMMddHHmmss`. For example, in 970916105432 `yy=97`, `MM=09`, `dd=16`, `HH=10`, `mm=54`, `ss=32`, which means that the query was issued in 1997, September, 16th, and at time 10:54:32. [15]

This query is somewhat similar to the one in the previous subtask. We have to *SELECT* the users that satisfy a couple of conditions: they have visited the webpage at 21 or 22 hours, and they have issued a query (the query section is non-empty). This can be accomplished in a single query too.

When we *SELECT COUNT(DISTINCT(user))*, we are saying that we want the program to return the number of unique users that satisfy the conditions stated in the *WHERE* clause. We specify two conditions:

- `CAST(SUBSTRING(time, 7, 6) AS int) BETWEEN 210000 AND 225959`: this first part selects the 6 subsequent characters starting in position 7 ('HHmmss') of the column `time`, and converts it to an int. After that, it selects the entries if the value is between 210000 and 225959. The *BETWEEN* clause is inclusive, meaning that the begin and end values are included. Hence, any visit done between 21:00:00 and 22:59:59 hours will be included in this query. We could have also selected only the two subsequent characters starting in position 7 to be *BETWEEN* 21 and 22.
- `TRIM(query) <> '' OR query IS NOT NULL`: this second part of the *WHERE* clause is to satisfy the part where the users issued a query. We select the entries where the values of the column `query` is not null, or the entries where the column value in `query` is not an empty string. *TRIM* is a function that removes leading and trailing spaces from a string, so if a value is one or more spaces, it will not be selected.

This is how it looks when you do it in the terminal:

```
hive>
> SELECT COUNT(DISTINCT(user)) FROM logs WHERE (CAST(SUBSTRING(time, 7, 6) AS int) BETWEEN 210000 AND 225959) AND (TRIM(query) <> '' OR query IS NOT NULL);
Query ID = cloadera_20210329172626_0ba52877-2f35-4193-8e78-cc40c63f6df2
Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks determined at compile time: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapreduce.job.reduces=<number>
Starting Job = job_1617034564874_0003, Tracking URL = http://quickstart.cloudera:8088/proxy/application_1617034564874_0003/
Kill Command = /usr/lib/hadoop/bin/hadoop job -kill job_1617034564874_0003
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 1
2021-03-29 17:26:23,207 Stage-1 map = 0%, reduce = 0%
2021-03-29 17:26:30,579 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 2.08 sec
2021-03-29 17:26:38,986 Stage-1 map = 100%, reduce = 100%, Cumulative CPU 3.78 sec
MapReduce Total cumulative CPU time: 3 seconds 780 msec
Ended Job = job_1617034564874_0003
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1 Reduce: 1 Cumulative CPU: 3.78 sec HDFS Read: 214941 HDFS Write: 3 SUCCESS
Total MapReduce CPU Time Spent: 3 seconds 780 msec
OK
69
Time taken: 24.338 seconds, Fetched: 1 row(s)
hive>
```

As we can see, there are **69 unique users** that have issued a query in this webpage at 21 or at 22.