Example solutions for CW2

Task1

1

```
db.cl.find( { $and:[{ displayName: /^A/i}, {displayName: /es$/}, {friendsCount:{$lt: 25}}]},
{displayName:1, followersCount:1, friendsCount:1}).sort({displayName:-1})
```

Correct output 3, Correct conditions (^A/I, /es$/, friendsCount:$lt:25} 3, General syntax 4

2

```
db.cl.aggregate( [{$match: {friendsCount:{$gt:0}}}, { $group :{ _id:null, avgRatio: { $avg: {$divide:
["$followersCount","$friendsCount"]}}}} ] )
```

Correct output 3, Correct conditions (ratio definition) 3, General syntax 4

3

```
db.cl.find({body:{$regex:/Madrid/},verb:"post",friendsCount:{$gte:1000}}).count()
```

Task 2

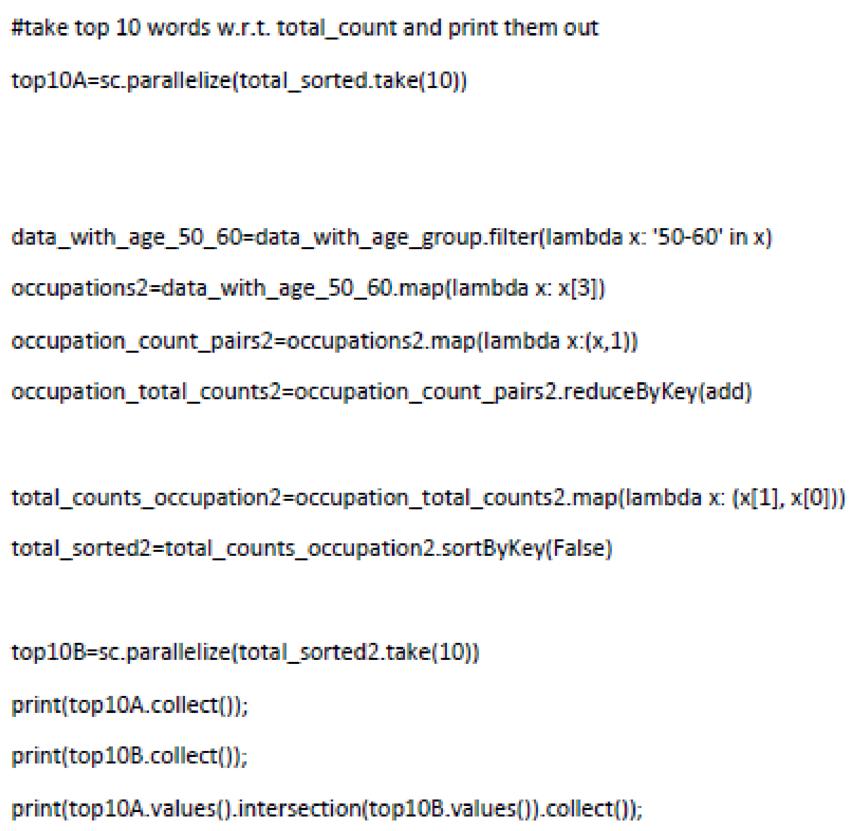
This is an example solution. There were other valid solutions by many of you. It is important to utilize the parallelization capabilities of Spark in the operations of sorting, finding the 10 most frequent values, and intersection.

For example, one could do everything (sort, top10, intersection) in Python after applying .collect() on the RDDs. But the program would not be parallel and could not handle data that is larger than the main memory of the computer where the driver runs. See https://umbertogriffo.gitbooks.io/apache-spark-best-practices-and-tuning/content/dont_collect_large_rdds.html to learn more.

Another could apply .collect() to the values of an RDD, then sort and top10 in Python (using dictionaries, for example) and then do intersection in Spark. This is better but not perfect, because still .collect() on an RDD limits the level of parallelization on the program.

Task 2 Code Screenshots:

```
from pyspark import SparkContext

from operator import add


sc = SparkContext( 'local', 'pyspark')


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+'

def parse_with_age_group(data):
    userid,age,gender,occupation,zip = data.split("|")
    return userid, age_group(int(age)),gender,occupation,zip,int(age)



fs=sc.textFile("file:///home/cloudera/Downloads/u.user")
data_with_age_group=fs.map(parse_with_age_group)
data_with_age_40_50=data_with_age_group.filter(lambda x: '40-50' in x)

#obtain the occupations of the age group [40,50)
occupations=data_with_age_40_50.map(lambda x: x[3])

#create (occupation, 1) pairs i.e., set count 1 per occupation
occupation_count_pairs=occupations.map(lambda x:(x,1))

#create (occupation, total_count) pair i.e., sum the counts for each
#occupation
occupation_total_counts=occupation_count_pairs.reduceByKey(add)

#create tuple (total_count, occupation) so that total_count is the key
#we need this to be able to sort by key in the next step
total_counts_occupation=occupation_total_counts.map(lambda x: (x[1], x[0]))
#sort by key (i.e., total_count) in descending order
total_sorted=total_counts_occupation.sortByKey(False)
```



```
#take top 10 words w.r.t. total_count and print them out
top10A=sc.parallelize(total_sorted.take(10))


data_with_age_50_60=data_with_age_group.filter(lambda x: '50-60' in x)
occupations2=data_with_age_50_60.map(lambda x: x[3])
occupation_count_pairs2=occupations2.map(lambda x:(x,1))
occupation_total_counts2=occupation_count_pairs2.reduceByKey(add)

total_counts_occupation2=occupation_total_counts2.map(lambda x: (x[1], x[0]))
total_sorted2=total_counts_occupation2.sortByKey(False)

top10B=sc.parallelize(total_sorted2.take(10))
print(top10A.collect());
print(top10B.collect());
print(top10A.values().intersection(top10B.values()).collect());
```

Task 2 Code Written:

```
from pyspark import SparkContext
from operator import add

sc = SparkContext('local', 'pyspark')

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+'

def parse_with_age_group(data):
    userid, age, gender, occupation, zip = data.split("|")
    return userid, age_group(int(age)),gender,occupation,zip,int(age)

fs = sc.textFile("file:///home/cloudera/Downloads/u.user")
data_with_age_group = fs.map(parse_with_age_group)
data_with_age_40_50=data_with_age_group.filter(lambda x: '40-50' in x)

# obtain the occupations of the age group [40, 50)
occupations=data_with_age_40_50.map(lambda x:x[3])

# create (occupation, 1) pairs i.e, set count 1 per occupation
occupation_count_pairs = occupations.map(lambda x:(x,1))

# create (occupation, total_count) pair i.e., sum the counts of each
occupation_total_counts=occupation_count_pairs.reduceByKey(add)
```

```
# create tuple(total_count, occupation) so that total_count is the key

# we need this to sort by key in the next step

total_counts_occupation=occupation_total_counts.map(lambda x: (x[1], x[0]))

# sort by key (ie. total count) in descending order

total_sorted = total_counts_occupation.sortByKey(False)


# take top 10 words w.r.t. total count and print them out

top10A = sc.parallelize(total_sorted.take(10))


data_with_age_50_60=data_with_age_group.filter(lambda x: '50-60' in x)

occupations2=data_with_age_50_60.map(lambda x: x[3])

occupation_count_pairs2=occupations2.map(lambda x: (x,1))

occupation_total_counts2=occupation_count_pairs2.reduceByKey(add)


total_counts_occupation2=occupation_total_counts2.map(lambda x: (x[1], x[0]))

total_sorted2=total_counts_occupation2.sortByKey(False)


top10B = sc.parallelize(total_sorted2.take(10))

print(top10A.collect())

print(top10B.collect())

print(top10A.values().intersection(top10B.values()).collect());
```

After running: *spark-submit task2.py --master local[2]*

Task2 Output (the below is not including a bunch of process lines printed):

```
[(25, u'educator'), (22, u'other'), (20, u'administrator'), (14, u'engineer'), (13, u'writer'), (13, u'librarian'),
(11, u'executive'), (10, u'programmer'), (8, u'scientist'), (7, u'healthcare')]

[(23, u'educator'), (12, u'administrator'), (11, u'librarian'), (11, u'other'), (6, u'writer'), (5, u'marketing'),
(5, u'engineer'), (3, u'lawyer'), (3, u'healthcare'), (3, u'programmer')]

[u'administrator', u'programmer', u'healthcare', u'writer', u'educator', u'librarian', u'other', u'engineer']
```

Note the print(top10A.collect()); and print(top10B.colect()); are not needed.

Task 3

1

SELECT MAX(count) as Max FROM (SELECT user,COUNT(time) as count FROM logs GROUP BY user) as tmp;

2

Here I provide two solutions.
SOL1: select * from logs where query like '%job%';
SOL2: select * from logs where instr(query,'job')!=0;

3

Here I provide two solutions.
SOL1: select count(distinct user) from logs where query <> '' and (time like '%21____' or time like '%22____');
SOL2:
select count(distinct user) from logs where hour(from_unixtime(unix_timestamp(time,'yyMMddHHmmss'),'yyyy-MM-dd HH:mm:ss'))='22' or hour(from_unixtime(unix_timestamp(time,'yyMMddHHmmss'),'yyyy-MM-dd HH:mm:ss'))='21' and query!='';