1 Count number of elements under a threshold

1.1 Using the RDD interface

\[
R.\text{filter}(\lambda e: e<V).\text{count()}
\]

1.2 Using the DataFrame interface

\[
D.\text{where}(D["c"]<V).\text{count()}
\]

1.3 Using the Map-Reduce interface

```python
def mapper(l):
    if double(l)<V:
        return [(0,1)]
    return []

def combiner(k,l):
    return [(0,sum(l))]

def reducer(k,vals):
    return sum(vals)
```

2 Root cause

For this exercise we want to compute the set of attributes \( X \) such that:

\[
\frac{\text{number of objects having } X}{\text{number of objects}} < \frac{\text{number of objects having } X \text{ and } C}{\text{number of objects having } C}
\]

We will now compute the four parts separately and then assemble everything.

2.1 Using RDDs

The number of objects. For this we need to count the number of objects and not the number of pairs!

\[
\text{nbObjects} = D.\text{map}(\lambda x:x[0]).\text{distinct().count()}
\]

The number of objects having \( C \).

\[
\text{idWithC} = D.\text{filter}(\lambda x:x[1]==C)
\)

\[
\text{nbObjectsHavingC} = \text{idWithC}.\text{count()}
\]
An RDD containing pairs \((X, N_X)\) where \(N_X\) is the number of items having \(X\).

```python
def add(x, y):
    return x + y
```

\[\text{numberOfX} = D \text{.map(lambda x: (x[1], 1)).reduceByKey(add)}\]

An RDD containing pairs \((X, N_{XC})\) where \(N_{XC}\) is the number of items having both \(X\) and \(C\).

\[\text{numberOfXandC} = D \text{.join(idWithC)} \backslash
\text{.map(lambda x: (x[1][0], 1)) \backslash}
\text{.reduceByKey(add)}\]

Assembling everything.

\[\text{count} = \text{numberOfX} \text{.join(numberOfXandC)}\]

\# count contains triples \((X, (nX, nXC))\) so we have all the info we need

salient = \[
\text{count.filter(lambda x: x[1][0]/nbObjects < x[1][1]/nbObjectsHavingC)\backslash}
\text{.map(lambda x: x[0])}\]

Notice that count will no contain the attributes \(X\) where there are no objects having \(X\) and \(C\) but those are necessarily not in the output.

### 2.2 Using Dataframes

There are many ways to translate the above code in Dataframe or SQL. Here is one way to write using one SQL query:

```
DF.createOrReplaceTempView("D")
spark.sql(""
SELECT D2.attribute
FROM
    D D1,
    D D2,
    D D3,
    D D4,
    D D5
WHERE
    D1.identifier = D2.identifier
AND D1.attribute = '"""+C+"""'
AND D3.attribute = D2.attribute
AND D4.attribute = D1.attribute
GROUP BY D2.attribute
HAVING
```

2
\[
\frac{\text{COUNT(DISTINCT D3.identifier)}}{\text{COUNT(DISTINCT D5.identifier)}} < \frac{\text{COUNT(DISTINCT D1.identifier)}}{\text{COUNT(DISTINCT D3.identifier)}}
\]

The idea is that:

- \(D5\) will range over all tuples and thus \(\text{COUNT(DISTINCT D5.identifier)}\) computes the number of objects.

- \(D1\) and \(D2\) are pairs with the same identifier (thus the same object) that has both \(X\) and \(C\) and thus \(\text{COUNT(DISTINCT D1.identifier)}\) computes the number of objects having both \(X\) and \(C\).

- \(D3\) will refer to any object that has \(X\), and thus \(\text{COUNT(DISTINCT D3.identifier)}\) computes the number of objects having \(X\).

- \(D4\) will refer to any object that has \(C\) and thus \(\text{COUNT(DISTINCT D4.identifier)}\) computes the number of objects having \(C\).

### 3 Multiplying sparse matrices

We have a matrix \(A\) represented as a set of triples \((x, y, A_{x,y})\) and a matrix \(B\) represented as a set of triples \((y, z, B_{y,z})\) and we want to compute \(C\) which is the product. We now that \(C_{x,z} = \sum_y A_{x,y} \times B_{y,z}\) so we will compute the products in two steps, the first step will compute the set \((x, z), (A_{x,y} \times B_{y,z})\) where \(A_{x,y}\) and \(B_{y,z}\) are non zero. For this we will use a join on \(y\), computing an RDD containing all the tuples \(y, ((x, A_{x,y}), (z, B_{y,z}))\), then with a map \(\text{mult}\) will be transform this tuple into \((x, z), A_{x,y} \times B_{y,z}\). The second step will actually compute the product aggregating for a given pair \((x, z)\) all the \(y\) participating in the sum. For this simple reduce by key is used to sum all the different \(y\) components for a given pair \((x, z)\).

One optional last step is to remove the zeros in \(C\), for this we can simply use a filter.

```python
def mult(t):
    fromA = t[1][0]
    fromB = t[1][1]
    return ((fromA[0], fromB[0]), fromA[1] * fromB[1])

def add(x, y):
    return x + y

Am = A.map(lambda t: t[1], (t[0], t[2]))
Bm = B.map(lambda t: t[0], (t[1], t[2]))
C = Am.join(Bm).map(mult).reduceByKey(add)

CWithoutZeros = C.filter(lambda x: x[1] != 0)
```

3
Remark. This remark was not expected from students but notice that, at the join we will regroup all the $x$s and the $z$s attached to a given $y$. If we are really unlucky it might happen that there is only one $y$ and thus, because Spark is not that smart it might break even though we do not ask to materialize the matrix. We can reasonably suppose that this behavior does not happen but if it does we can force the join to be more parallel.
4 The page-rank algorithm in Spark RDD

```python
def pageRank(webpages, pointsTo, nbIter, alpha):
    N = webpages.count()  # number of webpages, used for initScore
    initScore = webpages.map(lambda x: (x, 1.0 / N))  # initScore is an RDD corresponding to the equalprobability score
    nbPointed = pointsTo.mapValues(lambda x: 1) .union(webpages.map(lambda x: (x, 0.))) .reduceByKey(add)  # number of pages each page points to. The union is needed to make sure # that pages without outgoing links appear in the RDD.

    def probTPOnePage(t):
        
        takes a triple t = (webpage, (score, nbPagesPointed)) and return the probability of a random TP from this webpage given the current probability of being in webpage 
        
        if t[1][1]==0:
            return t[1][0]
        else:
            return t[1][0]*alpha

    def probaTP(curScore):
        
        Returns the total probability of doing a random jump 
        
        return curScore.join(nbPointed).map(probTPOnePage).reduce(add)

    def oneIteration(curScore):
        
        # we compute the total probability of jumping
        pTP = probaTP(curScore)

        # then we compute an RDD containing for each page
        # how much score it receives from a TP
        scoreFromTP = initScore.mapValues(lambda x: x*pTP)

        # then we compute how much each page distribute to the pages it points to.
        # Takes a pair (proba, nbPointed)
        def distribute(t):
            
            if t[1] > 0:
                return t[0] *(1.-alpha) / t[1]
            probaDistribute = curScore.join(nbPointed).mapValues(distribute)

        # compute how much score a page receives from being pointed
        # note that this RDD will contain multiple entries for one page
        # but we will sum the various contributions later
        scoreFromPointed = pointsTo.join(probaDistribute).map(lambda x: x[1])

        # Return the new score
        return scoreFromPointed.union(scoreFromTP).reduceByKey(add)

    score = initScore
    for _ in range(nbIter):
        score = oneIteration(score)
    return score
```

5