Name: Tania Rajabally

Branch: Comps

Batch: C

Roll no: 43

UID:2017130047

Date: 18/09/2020

**Experiment 4**

**Aim:** To implementSON Algorithm

**Theory:**

SON: (the Algorithm of Savasere, Omiecinski, and Navathe) algorithm is a simple way of paralleling:

* Each computing unit gets a portion of data that can be fit in its memory (need a data structure to store the local data)
* then use A-Priori or PCY or other methods to find the locally frequent itemsets,which forms the candidate set.
* During this process, the local threshold is adjusted to p\*s, (p is the portion of data you get)
* During the 2nd pass, you only count for the candidate itemsets. • Filter out infrequent candidates.
* If an itemset is not locally frequent from all computing units, then it also cannot be frequent globally
* This algorithm can deal with whatever length itemsets in its paralleling process

Here, PCY algorithm is used to draw out intermediate results from the samples

* This algorithm, which we call PCY after its authors, exploits the observation that there may be much unused space in main memory on the first pass.
* Think of the array as a hash table, whose buckets hold integers rather than sets of keys (as in an ordinary hash table) or bits (as in a Bloom filter)
* Pairs of items are hashed to buckets of this hash table. As we examine a basket during the first pass, we not only add 1 to the count for each item in the basket, but we generate all the pairs, using a double loop.
* We hash each pair, and we add 1 to the bucket into which that pair hashes. Note that the pair itself doesn’t go into the bucket; the pair only affects the single integer in the bucket.
* At the end of the first pass, each bucket has a count, which is the sum of the counts of all the pairs that hash to that bucket. If the count of a bucket is at least as great as the support threshold s, it is called a frequent bucket.
* We can say nothing about the pairs that hash to a frequent bucket; they could all be frequent pairs from the information available to us. But if the count of the bucket is less than s (an infrequent bucket), we know no pair that hashes to this bucket can be frequent, even if the pair consists of two frequent items.
* That fact gives us an advantage on the second pass. We can define the set of candidate pairs C2 to be those pairs {i, j} such that:
  + i and j are frequent items
  + {i, j} hashes to a frequent bucket.

**Code:**

import pandas as pd

from collections import defaultdict

from itertools import permutations, combinations

from google.colab import drive

drive.mount('/content/gdrive')

%cd "/content/gdrive/My Drive/BDA"

#Reading the data

full\_data = open("data.txt", "r")

orders = []

for line in full\_data:

line = line.split(",")

temp = []

for x in line:

temp.append(int(x))

orders.append(temp)

len(orders)

support = 6000

#Hash function = (i+j)%5

global\_list = []

#Splitting the data into samples of 20,000 records each

lines\_per\_file = 20000

smallfile = None

with open('data.txt') as bigfile:

for lineno, line in enumerate(bigfile):

if lineno % lines\_per\_file == 0:

if smallfile:

smallfile.close()

small\_filename = 'sample\_file\_{}.txt'.format(int(lineno / lines\_per\_file))

smallfile = open('samples/'+small\_filename, "w")

smallfile.write(line)

if smallfile:

smallfile.close()

#Function to return count of all the items and hash map to stored the hashed pairs

def findFI(orders):

dica = defaultdict(int)

hash = defaultdict(list)

for order in orders:

#print(order, "order")

for item in order:

dica[item] = dica[item] + 1

pairs = combinations(order, 2)

#print(list(pairs))

for pair in list(pairs):

hash[ (pair[0]+pair[1])%5 ].append(pair)

return dica , hash

#The two rules of PCY algorithm are applied here on the basis of Dictionary of item counts and Hash map of different pairs are returned from the above function

def applyPCY(orders):

final\_fi = []

item\_count, buckets = findFI(orders)

for key in buckets.keys():

for pair in buckets[key]:

if len(buckets[key]) >= support:

if item\_count[pair[0]] >= support and item\_count[pair[1]] >= support:

final\_fi.append(pair)

return list(set(final\_fi)), buckets

#Applying the above functions on each of the sample files

for i in range(6):

print("\n-----------------------------------------------------------\n")

print('FILE',i+1)

file = 'samples/sample\_file\_'+str(i)+'.txt'

f = open(file, "r")

orders = []

for line in f:

line = line.split(",")

temp = []

for x in line:

temp.append(int(x))

orders.append(temp)

li, buck = applyPCY(orders)

global\_list = global\_list + li

print("Number of frequent pairs",len(li))

print("Frequent pairs: ")

print(li)

# print(len(list(set(li))))

print("Hash map")

print("Number of keys", buck.keys())

for k in buck.keys():

print("Length of bucket",k, len(buck[k]))

# print(li)

#This list stores all the frequent pairs from all the samples

global\_list = list(set(global\_list))

len(global\_list)

#Pass2

#Reading the order records

full\_data = open("data.txt", "r")

orders = []

for line in full\_data:

line = line.split(",")

temp = []

for x in line:

temp.append(int(x))

orders.append(temp)

#Checking if the pair is frequent in the whole dataset as well

globd = defaultdict(int)

for p in global\_list:

for o in orders:

if p[0] in o and p[1] in o:

globd[p] = globd[p] + 1

final\_fi = []

for k,v in globd.items():

if v >= 36000:

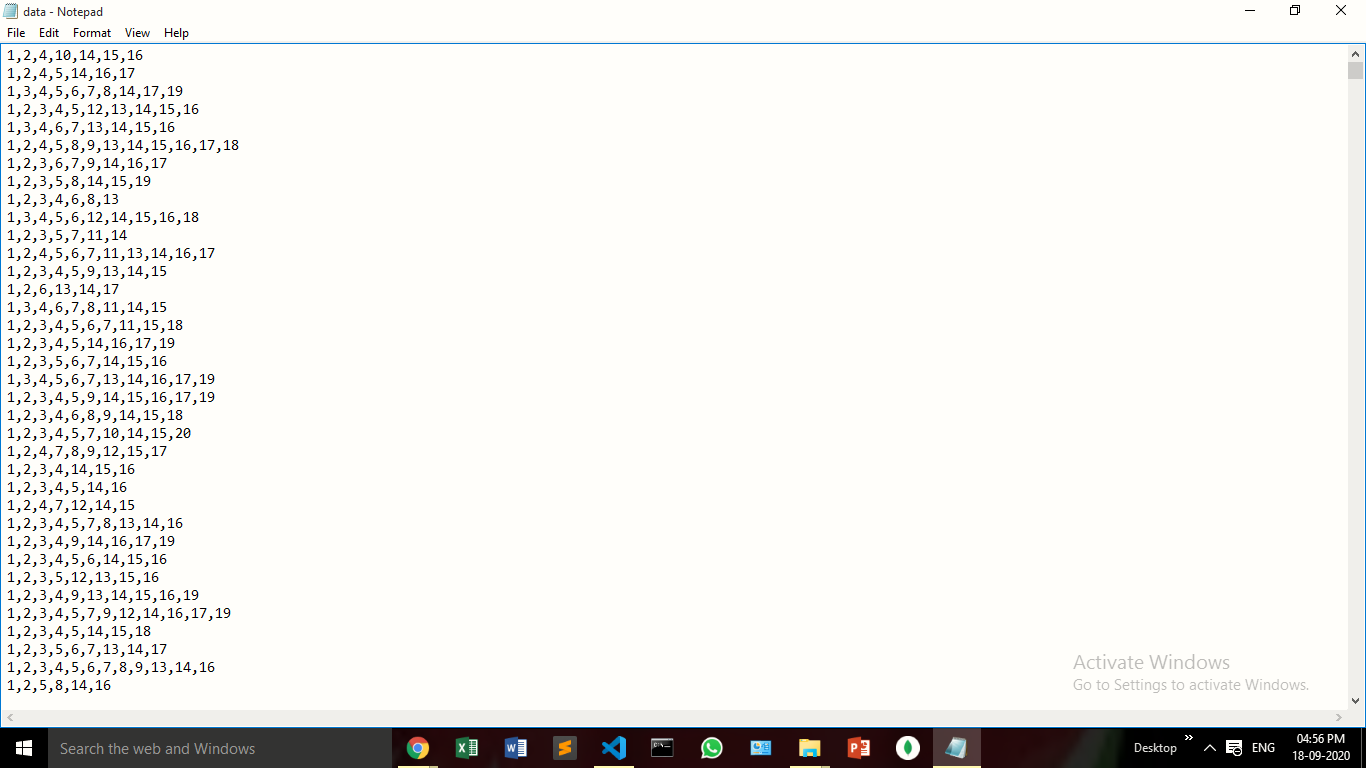
final\_fi.append(k)

print("Number of frequent pairs:", len(final\_fi))

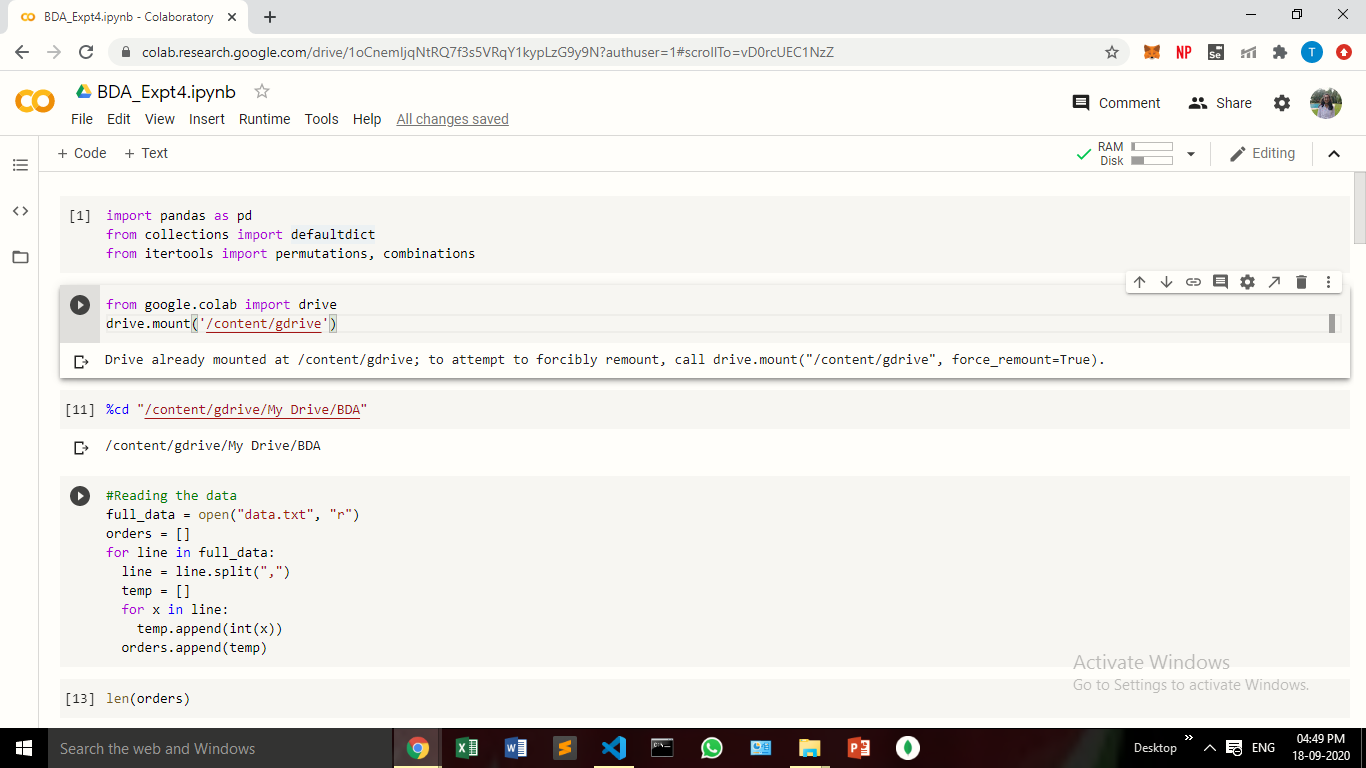
print("Frequent itemset pairs are:")

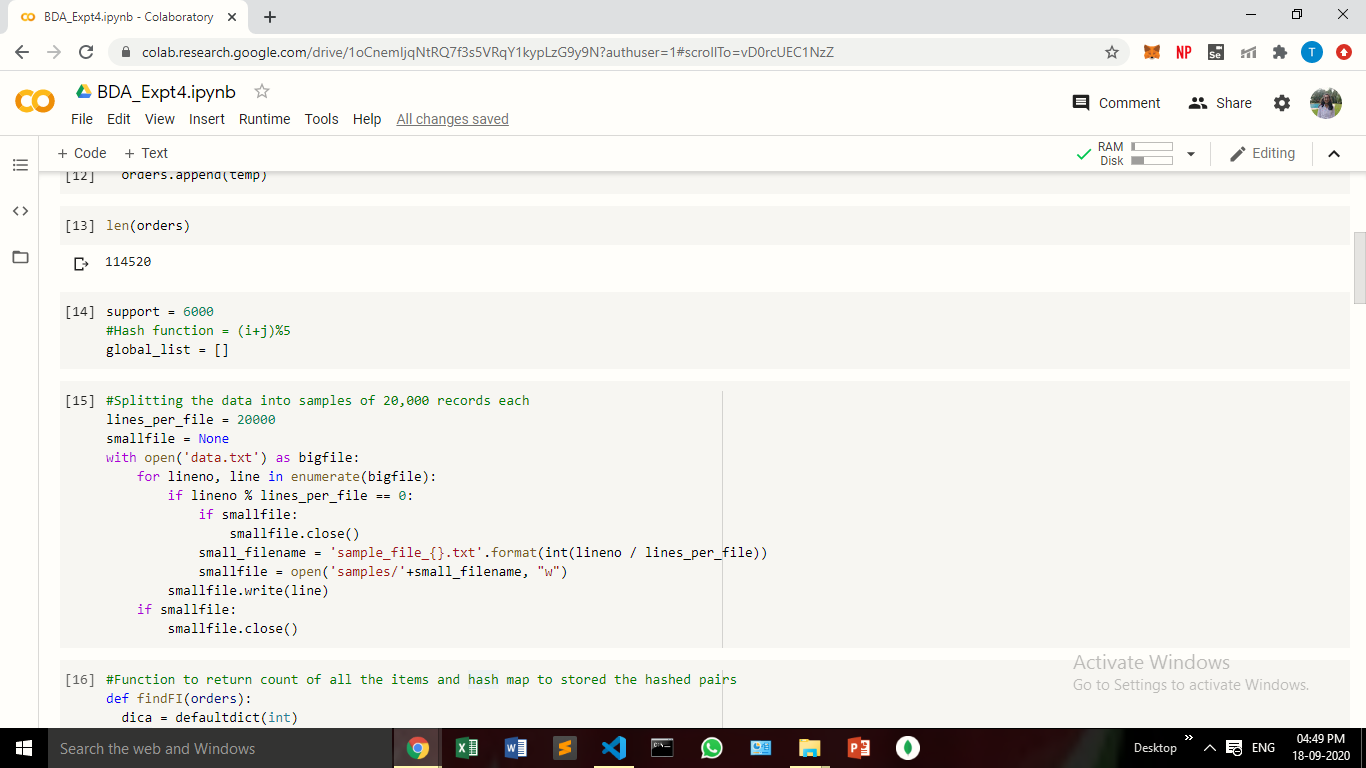
print(final\_fi)

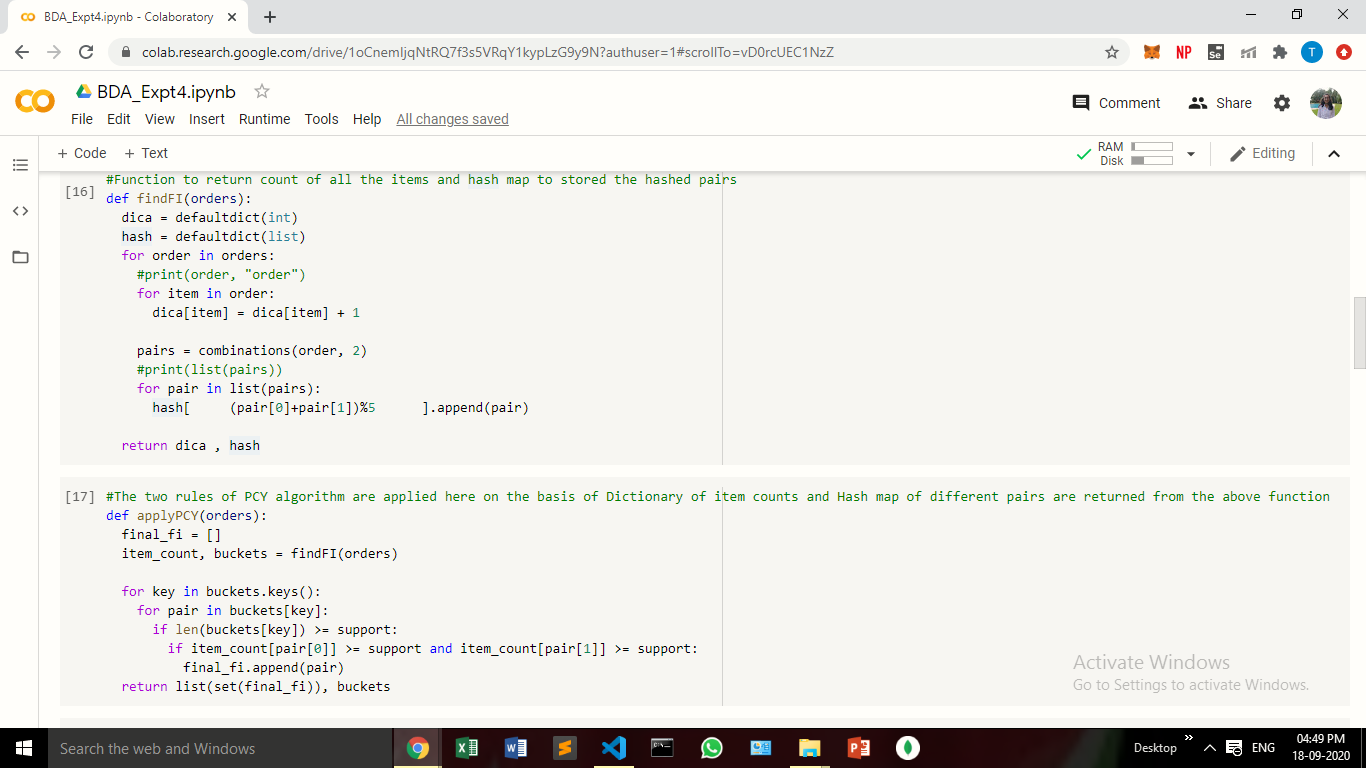
**Data.txt:**

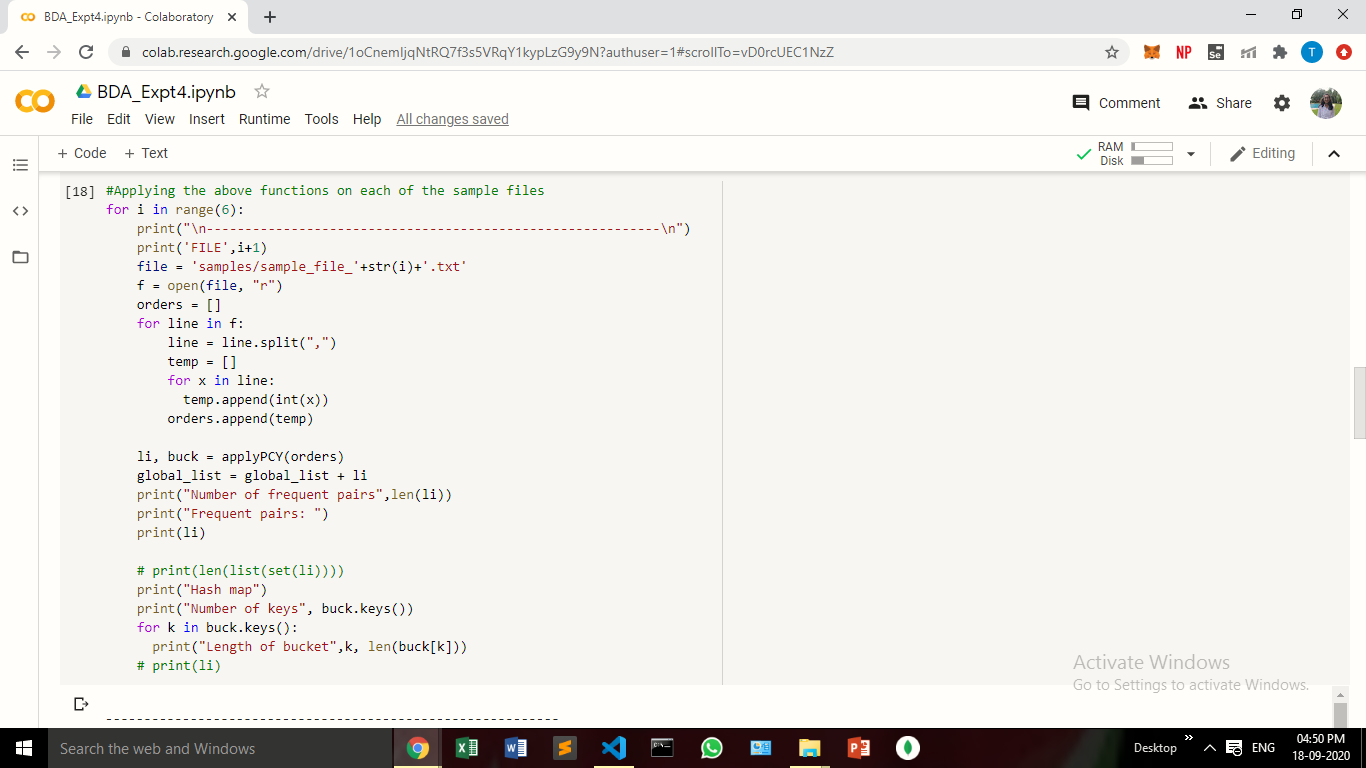
****

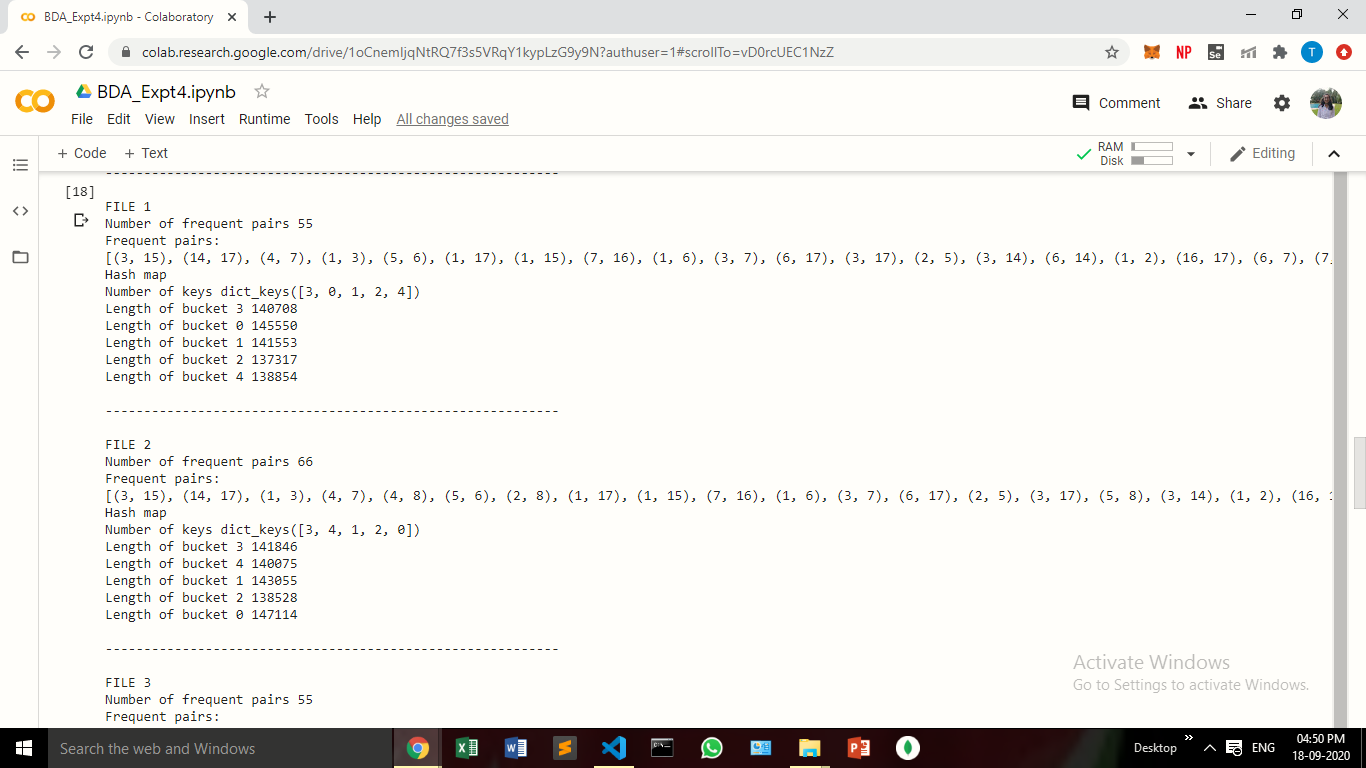
**Output:**

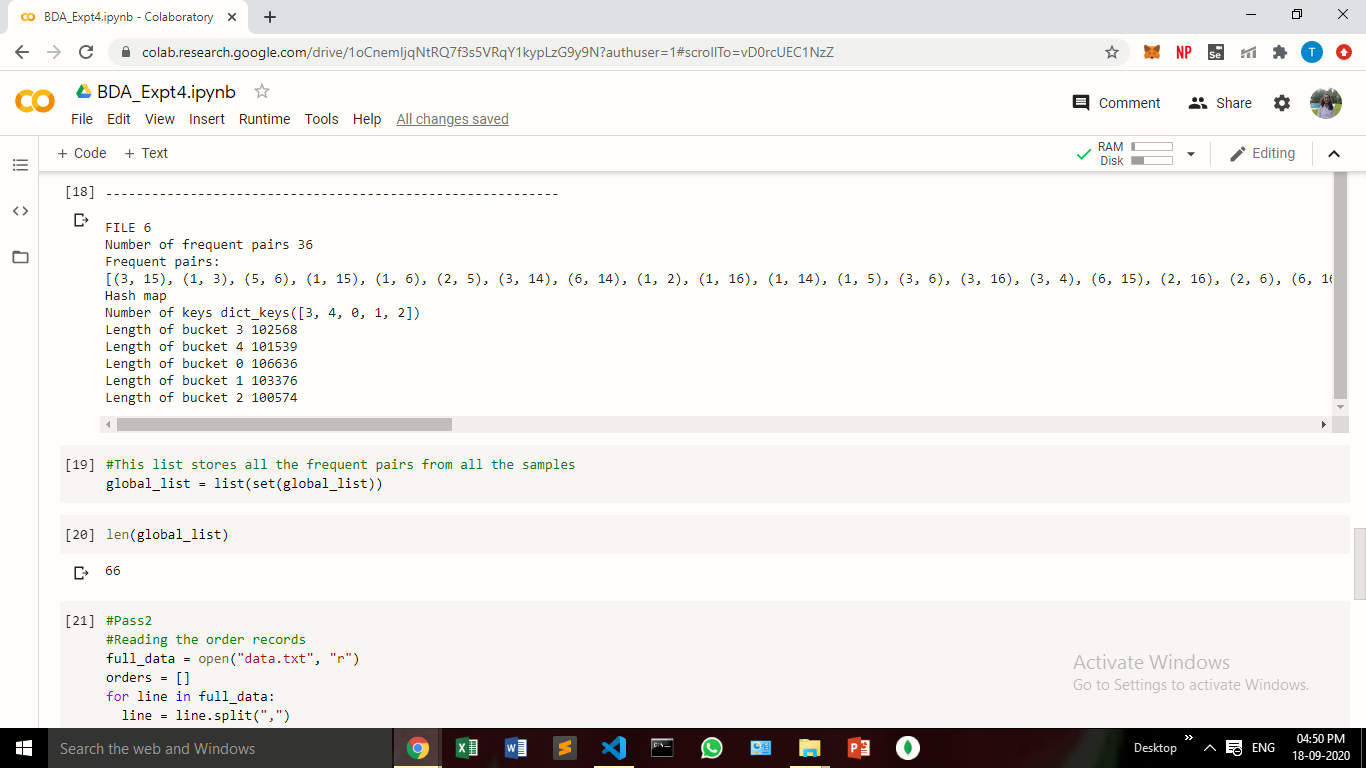
****

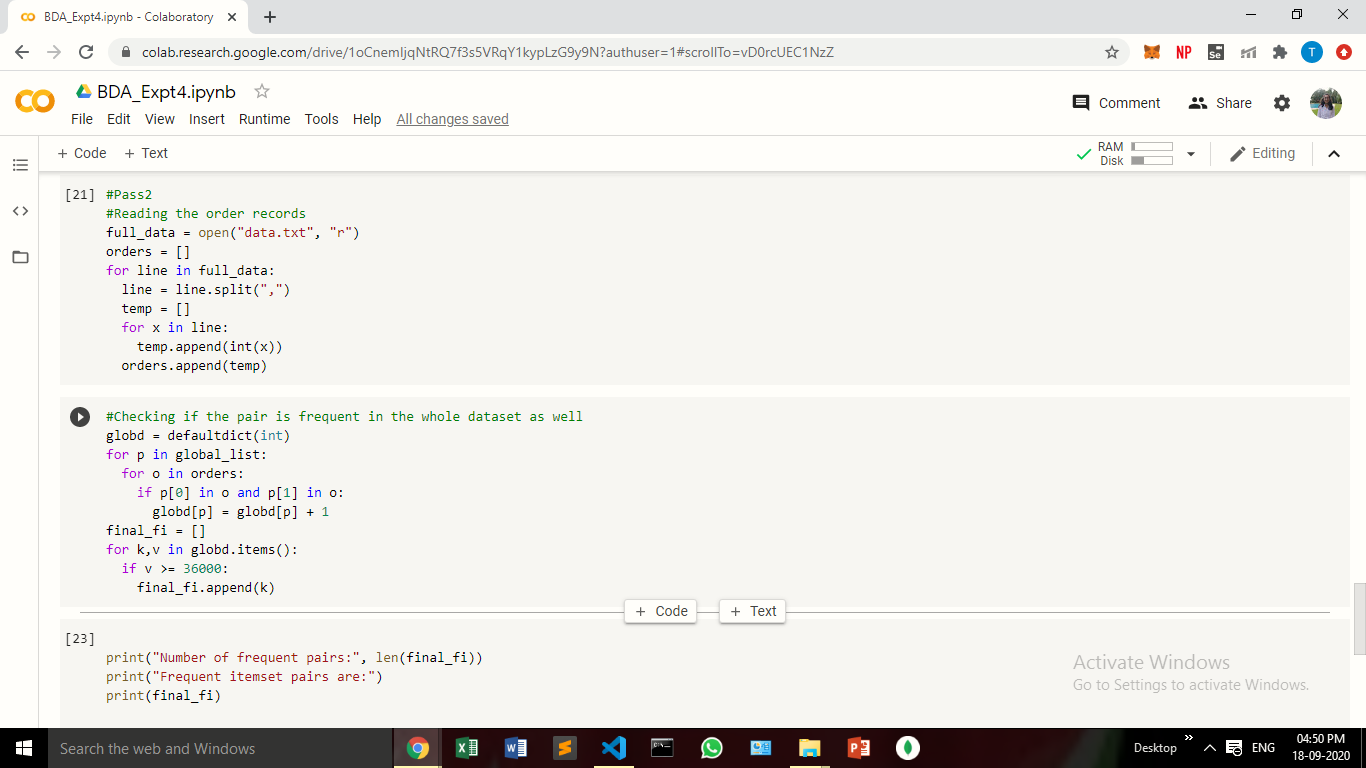
****

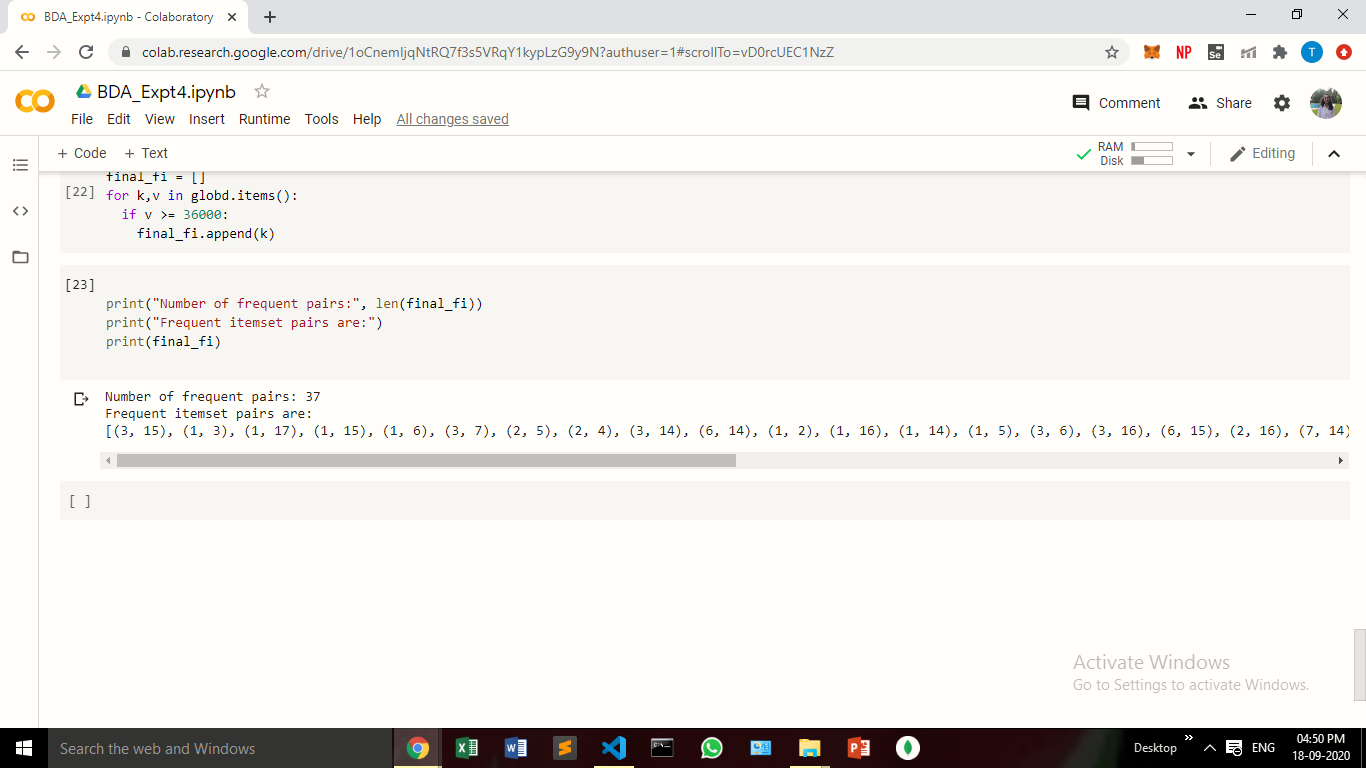
****

****

****

****

****

****

**Conclusion:**

In this experiment, we have understood the SON algorithm and have fully implemented the same using the PCY algorithm. The significance of the SON algorithm is realized here, when we deal with huge amounts of data / Big Data

.