ENGG 4030 Homework_2

I declare that the assignment submitted on Elearning system is original except for source material explicitly acknowledged, and that the same or related material has not been previously submitted for another course. I also acknowledge that I am aware of University policy and regulations on honesty in academic work, and of the disciplinary guidelines and procedures applicable to breaches of such policy and regulations, as contained in the website http://www.cuhk.edu.hk/policy/academichonesty/ .

Signed (Student 柳一村) Date: 18/03/2018

Name LIU Yicun SID 1155092202

Q1

Pre-processing:

Due to the fact that we have two datasets “shakespeare1” and “shakespeare2” in hand, I simply combine the two input files as “shakespeare3” and use it as input of the following task.

Note that in the second original dataset “shakespeare2”, there are some duplicate words in each line, here I choose to delete them in the mapper, using data structure named *Set* in python:

```
        words = line.split()
        words = list(set(words))
```

A: Find Frequent Item Sets

(a) As the requirement said, each line means a basket, and here we need to implement a non-distribute version of A-priori to find the top-40 frequent pairs.

In the first pass, we count the frequency of single item and find the frequent items:

```
with open(fname) as f:
    lines = [line.rstrip('\n') for line in open(fname)]
    for line in lines:
        num_lines += 1
        words = line.split()
        words = list(set(words))
        for word in words:
            if word not in wordfreq:
                wordfreq[word] = 1
            else:
                wordfreq[word] += 1

for word in wordfreq:
    if (wordfreq[word] >= threshold * num_lines):
        temp[word] = wordfreq[word]
```

In the second pass, I read baskets again and count in main memory only the pairs where both elements are frequent, and output the final pairs with top-40 frequencies:

```
# Second step of A-Priori
with open(fname) as f:
    lines = [line.rstrip('\n') for line in open(fname)]
    for line in lines:
        words = line.split()
        words = list(set(words))
        for j in range (0,len(words)-1):
            for k in range (j+1,len(words)):
                if(words[j] in wordfreq) and (words[k] in wordfreq):
                    if(words[j]<=words[k]):
                        pair = words[j]+","+words[k]
                    else:
                        pair = words[k]+","+words[j]
                    if pair not in pairfreq:
                        pairfreq[pair] = 1
                    else:
                        pairfreq[pair] += 1


if(len(pairfreq) > top_num):
    topkeys = sorted(pairfreq, key=pairfreq.get,reverse=True)[:top_num]
else:
    topkeys= sorted(pairfreq, key=pairfreq.get,reverse=False)

for topkey in topkeys:
    print(topkey,pairfreq[topkey])
```

Consider it is a non-distributed version, I simply run the code on my MacBook (late 2015, with core-m3), the result is:

```
('of,the', 352310)
('and,the', 174876)
('in,the', 146017)
('the,to', 145171)
('and,of', 106824)
('a,of', 100121)
('a,the', 84377)
('and,to', 74266)
('in,of', 73828)
('of,to', 68015)
('a,and', 66899)
('and,in', 62592)
('the,was', 62453)
('thou,thy', 61455)
('a,to', 57495)
('a,in', 54897)
('is,the', 54866)
('on,the', 53910)
('that,the', 51819)
('thee,thou',  50082)
('by,the', 47365)
('the,with', 44846)
('he,the', 44370)
('for,the', 43644)
('it,the', 41316)
('thee,thy', 41252)
('at,the', 40688)
('in,to', 40610)
('be,to', 40461)
('as,the', 39145)
('of,was', 36443)
('from,the', 34846)
('and,was', 34740)
('and,his', 34170)
('his,of', 33608)
('it,to', 33164)
('of,that', 33117)
('his,the', 33109)
('art,thou', 32978)
('i,the', 32738)
```

The running time is 4 mins 10 secs.

(b) Here I need to implement a distributed version of SON. Note that the first pass of SON is mainly A-Priori, so that I can utilize the code from part (a):

```python
p = 1
num_lines = 0
threshold = 0.005

for line in sys.stdin:
    num_lines += 1
    line = line.strip()

    basket.append(line)

    words = line.split()
    words = list(set(words))
    for word in words:
        if word not in wordfreq:
            wordfreq[word] = 1
        else:
            wordfreq[word] += 1

for word in wordfreq:
    if (wordfreq[word] >= threshold * num_lines / p):
        temp[word] = wordfreq[word]

wordfreq = copy.deepcopy(temp)
temp = {}

for i in range (0,len(basket)):
    line = basket[i]
    word_basket = line.strip().split()
    word_basket = list(set(word_basket))
    for j in range (0, len(word_basket)-1):
        for k in range(j+1,len(word_basket)):
            if(word_basket[j] in wordfreq):
                if(word_basket[k] in wordfreq):
                    if(word_basket[j]<= word_basket[k]):
                        pair = word_basket[j]+","+ word_basket[k]
                    else:
                        pair = word_basket[k]+","+ word_basket[j]
                    if pair not in pairfreq:
                        pairfreq[pair] = 1
                    else:
                        pairfreq[pair] += 1

for pair in pairfreq:
    if (pairfreq[pair] >= threshold * num_lines / p):
        print '%s\t%s' % (pair, str(pairfreq[pair]))
```

```python
import sys

current_pair = None
for line in sys.stdin:
    pair,count = line.strip().split('\t')
    if pair == current_pair:
        continue
    else:
        if current_pair:
            print current_pair
        current_pair = pair
if current_pair == pair:
    print current_pair
```

Mapper / Reducer in the first MapReduce job

In the first MapReduce job, I use A-Priori find the candidate pairs, which means that they should be frequent in at least one input file. The reducer in the first job stores the candidate pair into an intermediate file named “candidatepair_b.txt”.

```python
import sys

fname = 'candidatepair_b.txt'

threshold = 0.005

pairfreq = {}

with open(fname) as f:
    lines = [line.rstrip('\n') for line in open(fname)]
    for line in lines:
        pair = line.strip()
        pairfreq[pair] = 0

for line in sys.stdin:
    line = line.strip()
    words = line.split()
    words = list(set(words))
    for j in range (0,len(words)-1):
        for k in range(j+1,len(words)):
            if(words[j]<=words[k]):
                pair = words[j]+","+words[k]
            else:
                pair = words[k]+","+words[j]
            if pair in pairfreq:
                pairfreq[pair] += 1

for pair in pairfreq:
    print '%s\t%s' % (pair, str(pairfreq[pair]))
```

```python
import sys

pairfreq = {}

threshold = 0.005
p = 1
num_lines = 4340061

current_pair = None
for line in sys.stdin:
    pair,count = line.strip().split("\t")
    count = int(count)
    if current_pair == pair:
        total += count
    else:
        if current_pair:
            if total >= num_lines * threshold / p:
                print current_pair+"\t"+str(total)
        current_pair = pair
        total = count
```

Mapper / Reducer in the second MapReduce job

In the second MapReduce job, the mapper should read the candidate pairs from the intermediate file and counts only the candidate frequent pairs, while the reducer should combine and output the real frequent pairs. Note that when the reducer number is more than zero, the total threshold to determine the true frequent pairs should be divided by p (the number of reducers). I integrate all the commands into a bash file:

```
chmod +x mapper_b1.py mapper_b2.py reducer_b1.py reducer_b2.py

hdfs dfs -rm -r /user/ly116/output_b1
hdfs dfs -rm -r /user/ly116/output_b2

hdfs dfs -mkdir /user/ly116/shakespeare3
hdfs dfs -put ./shakespeare_basket/shakespeare_basket3 /user/ly116/shakespeare3

hadoop jar /usr/hdp/current/hadoop-mapreduce-client/hadoop-streaming.jar \
-D mapreduce.map.memory.mb=4096 \
-D mapreduce.reduce.memory.mb=4096 \
-D mapred.map.tasks=20 \
-D mapred.reduce.tasks=1 \
-input /user/ly116/shakespeare3/* \
-output output_b1 \
-file mapper_b1.py -mapper mapper_b1.py \
-file reducer_b1.py -reducer reducer_b1.py

hdfs dfs -getmerge output_b1 ~/candidatepair_b.txt

hadoop jar /usr/hdp/current/hadoop-mapreduce-client/hadoop-streaming.jar \
-D mapreduce.map.memory.mb=4096 \
-D mapreduce.reduce.memory.mb=4096 \
-D mapred.map.tasks=20 \
-D mapred.reduce.tasks=1 \
-input /user/ly116/shakespeare3/* \
-output output_b2 \
-file mapper_b2.py -mapper mapper_b2.py \
-file reducer_b2.py -reducer reducer_b2.py \
-file candidatepair_b.txt

hdfs dfs -cat /user/ly116/output_b2/* > out_b.txt

hdfs dfs -rm -r /user/ly116/output_b1
hdfs dfs -rm -r /user/ly116/output_b2
```

First MapReduce: 20 mappers, 1 reducer, avg execution time is 14s.

Job Name:	streamjob587191504991495096.jar
User Name:	ly116
Queue:	default
State:	SUCCEEDED
Uberized:	false
Submitted:	Mon Mar 19 01:45:00 HKT 2018
Started:	Mon Mar 19 01:45:05 HKT 2018
Finished:	Mon Mar 19 01:45:20 HKT 2018
Elapsed:	14sec
Diagnostics:	
Average Map Time	9sec
Average Shuffle Time	3sec
Average Merge Time	0sec
Average Reduce Time	0sec

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Mon Mar 19 01:45:03 HKT 2018	dic13.ie.cuhk.edu.hk:8042	logs

Task Type	Total	Complete
Map	20	20
Reduce	1	1

Second MapReduce: 20 mappers, 1 reducer, avg execution time is 15s.

Job Overview

Job Name:	streamjob8264891015266369938.jar
User Name:	ly116
Queue:	default
State:	SUCCEEDED
Uberized:	false
Submitted:	Mon Mar 19 01:45:26 HKT 2018
Started:	Mon Mar 19 01:45:31 HKT 2018
Finished:	Mon Mar 19 01:45:46 HKT 2018
Elapsed:	15sec
Diagnostics:	
Average Map Time	9sec
Average Shuffle Time	3sec
Average Merge Time	0sec
Average Reduce Time	0sec

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Mon Mar 19 01:45:29 HKT 2018	dic14.ie.cuhk.edu.hk:8042	logs

Task Type	Total	Complete
Map	20	20
Reduce	1	1

Benefited from distributed structure of multiple mappers and reducers, the overall execution time is only about 1/9 of the execution time of (a).
The final result (after sorting the top 40) is:

```
 1  of,the  352310
 2  and,the 174876
 3  in,the  146017
 4  the,to  145171
 5  and,of  106824
 6  a,of    100121
 7  a,the   84377
 8  and,to  74266
 9  in,of   73828
10  of,to   68015
11  a,and   66899
12  and,in  62592
13  the,was 62453
14  thou,thy    61455
15  a,to    57495
16  a,in    54897
17  is,the  54866
18  on,the  53910
19  that,the    51819
20  thee,thou   50082
21  by,the  47365
22  the,with    44846
23  he,the  44370
24  for,the 43644
25  it,the  41316
26  thee,thy    41252
27  at,the  40688
28  in,to   40610
29  be,to   40461
30  as,the  39145
31  of,was  36443
32  from,the    34846
33  and,was 34740
34  and,his 34170
35  his,of  33608
36  it,to   33164
37  of,that 33117
38  his,the 33109
39  art,thou    32978
40  i,the   32738
```

(c) For the SON of tuples, the main difference is the first mapper, which is expected to output the candidate tuples, here we add 'find candidate tuples' to the previous A-Priori code for pairs. With frequent pairs in hand, I read baskets for the third time and count in main memory only the tuples where six possible pairs of it are all frequent:

```
for i in range (0,len(basket)):
    line = basket[i]
    word_basket = line.strip().split()
    word_basket = list(set(word_basket))
    for j in range(0, len(word_basket) - 2):
        for k in range(j+1, len(word_basket) - 1):
            for l in range(k+1, len(word_basket)):
                if(word_basket[j]<= word_basket[k]):
                    pair_1 = word_basket[j]+","+ word_basket[k]
                else:
                    pair_1 = word_basket[k]+","+ word_basket[j]
                if(word_basket[j]<= word_basket[l]):
                    pair_2 = word_basket[j]+","+ word_basket[l]
                else:
                    pair_2 = word_basket[l]+","+ word_basket[j]
                if(word_basket[k]<= word_basket[l]):
                    pair_3 = word_basket[k]+","+ word_basket[l]
                else:
                    pair_3 = word_basket[l]+","+ word_basket[k]

                if(pair_1 in pairfreq) and (pair_2 in pairfreq) and (pair_3 in pairfreq):
                        if(word_basket[j] < word_basket[k]) and (word_basket[k] < word_basket[l]):
                            tupl = word_basket[j]+","+ word_basket[k]+","+ word_basket[l]
                        if(word_basket[j] < word_basket[l]) and (word_basket[l] < word_basket[k]):
                            tupl = word_basket[j]+","+ word_basket[l]+","+ word_basket[k]
                        if(word_basket[k] < word_basket[l]) and (word_basket[l] < word_basket[j]):
                            tupl = word_basket[k]+","+ word_basket[l]+","+ word_basket[j]
                        if(word_basket[k] < word_basket[j]) and (word_basket[j] < word_basket[l]):
                            tupl = word_basket[k]+","+ word_basket[j]+","+ word_basket[l]
                        if(word_basket[l] < word_basket[j]) and (word_basket[j] < word_basket[k]):
                            tupl = word_basket[l]+","+ word_basket[j]+","+ word_basket[k]
                        if(word_basket[l] < word_basket[k]) and (word_basket[k] < word_basket[j]):
                            tupl = word_basket[l]+","+ word_basket[k]+","+ word_basket[j]
                        if tupl not in tuplefreq:
                            tuplefreq[tupl] = 1
                        else:
                            tuplefreq[tupl] += 1
```

First MapReduce: 40 mappers, 10 reducers, avg execution time is 1min 14s.

Job Overview

Job Name:	streamjob2253674052302093918.jar
User Name:	ly116
Queue:	default
State:	SUCCEEDED
Uberized:	false
Submitted:	Mon Mar 19 01:46:38 HKT 2018
Started:	Mon Mar 19 01:46:43 HKT 2018
Finished:	Mon Mar 19 01:47:57 HKT 2018
Elapsed:	1mins, 14sec
Diagnostics:	
Average Map Time	33sec
Average Shuffle Time	57sec
Average Merge Time	0sec
Average Reduce Time	0sec

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Mon Mar 19 01:46:41 HKT 2018	dic8.ie.cuhk.edu.hk:8042	logs

Task Type	Total	Complete
Map	40	40
Reduce	10	10

Second MapReduce: 40 mappers, 1 reducer, avg execution time is 1min 30s.

Job Overview

Job Name:	streamjob7988286556542426190.jar
User Name:	ly116
Queue:	default
State:	SUCCEEDED
Uberized:	false
Submitted:	Mon Mar 19 01:48:04 HKT 2018
Started:	Mon Mar 19 01:48:08 HKT 2018
Finished:	Mon Mar 19 01:49:38 HKT 2018
Elapsed:	1mins, 30sec
Diagnostics:	
Average Map Time	38sec
Average Shuffle Time	1mins, 14sec
Average Merge Time	0sec
Average Reduce Time	0sec

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Mon Mar 19 01:48:06 HKT 2018	dic12.ie.cuhk.edu.hk:8042	logs

Task Type	Total	Complete
Map	40	40
Reduce	1	1

The final result (after sorting the top 20) is:

```
1   and,of,the  57234
2   in,of,the   47475
3   of,the,to   40526
4   a,of,the    38061
5   and,in,the  25216
6   and,the,to  24450
7   act,enter,scene 23816
8   thee,thou,thy   21170
9   of,the,was  20549
10  act,exeunt,scene    20404
11  of,on,the   18828
12  is,of,the   17794
13  enter,exeunt,scene  17087
14  act,enter,exeunt    17080
15  a,in,the    16892
16  in,the,to   16381
17  a,and,of    16350
18  of,that,the 15906
19  a,and,the   15521
20  by,of,the   15396
```

(d) To implement PCY on the basis of the previous SON construction, we need to maintain a hash table in the first mapper:

```
hashtable = [0 for x in range(100000)]
```

For each line of the first stdin, I hash every possible combination of pairs in that line into the hash table and add 1 to the count.

```
    for j in range(0, len(words)-1):
        for k in range(j+1, len(words)):
            hash_val = hash(words[j] + words[k]) % 100000
            hashtable[hash_val] += 1
```

By the assumption that “only pairs in frequent buckets are frequent”, I check the count in the hash table in the second pass of A-Priori and only take the qualified ones as candidate pairs:

```
for i in range (0,len(basket)):
    line = basket[i]
    word_basket = line.strip().split()
    word_basket = list(set(word_basket))
    for j in range (0, len(word_basket)-1):
        for k in range(j+1,len(word_basket)):
            if(word_basket[j] in wordfreq):
                if(word_basket[k] in wordfreq):
                    hash_val = hash(word_basket[j] + word_basket[k]) % 100000
                    if(hashtable[hash_val] >= threshold * num_lines / p):
                        if(word_basket[j]<= word_basket[k]):
                            pair = word_basket[j]+","+ word_basket[k]
                        else:
                            pair = word_basket[k]+","+ word_basket[j]
                        if pair not in pairfreq:
                            pairfreq[pair] = 1
                        else:
                            pairfreq[pair] += 1
```

The output is the same as (b), which ensure our implementation output the correct answer. The running time for (b), (c), (d) is listed below:

Algorithm	1st MapReduce	2th MapReduce	Total	Intermediate File Size
(a)	NA	NA	4min 10s	NA
(b)	14s	15s	29s	96.1 KB
(c)	1 min 14s	1 min 30s	2min 44s	1.4 MB
(d)	16s	12s	28s	70.3 KB

Comparison of (b) and (d): PCY takes longer time in the 1st MapReduce job to do the hashing but reduce the intermediate candidate pair file size by about 20%, which means that it only need to maintain a matrix/dictionary in the second MapReduce job and the execution time in the second job is shorter.

Comparison of (a) and (b): The distributed version (b) (20 mappers and 1 reducer) saves more than 80% time than the non-distributed version (a). Benefited from multiple mappers, it can output the candidate pairs in parallel and so is counting the real frequent pairs from the candidate pairs.

Comparison of (b) and (c): considering (b) need to find tuples but not pairs, which requires N times more search in each stdin line, another pass round of A-Priori in the first mapper and needs N times more RAM in the second MapReduce job, the execution time for both two jobs are significantly longer.

Q2 Locality-Sensitive Hashing

(a) $P1 = 1 - [1 - (T_1)^r]^B$ $P2 = 1 - [1 - (T_2)^r]^B$

(b) When T1=0.9, T2=0.7, P1=0.99 and P2=0.05, we can choose r = 20 while B = 40. to satisfy that circumstance.