COMP9313 20T2 Project 1 Report

Student Name: Raymond Lu
Student Number: z5277884

1. Configuration

- Code Environment: MacBook Pro (13-inch, 2017, Two Thunderbolt 3 ports)
- Python 3.6.5
- PySpark 2.4.6
- Spark 2.4.6
- Hadoop 2.7
- Jdk 1.8

2. Implementation Details of *c2lsh()*

1) **Update the Structure of Data Hashes**

In the algorithm given in the lecture, it could be seen that every time we get a new offset, we need to check whether the difference between an individual data hash and the query hash is less than or equal to the offset. Therefore, if we can get the difference before the collision check starts, it can save time from calculating the same differences for multiple times. Moreover, it would be a waste of time if we check a data hash that has been confirmed to be a collision in a previous loop when we get a new offset value. Hence, we introduce a flag that shows whether a data hash has been confirmed collided or not in the previous loops.

To update the feature, we use the *map()* function in *PySpark* to apply updates to each element in *"data_hashes"*. After this, the elements in data hashes will transform from $(id, hashes)$ to $(id,\ hash\ differences,\ flag)$.

2) **Update the Flag of Collision**

We define a function *update()* to update the *"data_hashes"* variable in each loop. For a new offset, if the flag is still $False$, we check whether the data hash satisfies the count of αm, namely, the number of differences between hash values is greater or equal to αm this time. If the flag is already $True$, we do nothing to the data instance.

3) **Using A Loop to Find Enough Candidates**

To reach the candidate number requirement βn, we use a while loop to find out the minimum offset that can satisfy this demand. Offset starts from 0, and in each loop, its value increases by one. In each loop, *"data_hashes"* updates itself by using the *update()* function in *map()*. Afterward, we apply the *flatMap()* function to extract the data elements which have *True* flags, and map them into a list then store the list as a Resilient Distributed Dataset (RDD). We check the number of elements in the new RDD is greater than or equal to βn or not. If yes, we return the new RDD named *"candidate"*. Otherwise, we start a new loop.

4) **Extreme Cases**

At the beginning of the function, we check whether βn is greater than the count of *"data_hashes"* or αm is greater than the length of query hashes. If one of them is true, then the function will get into a dead loop. We try to avoid this by raising exceptions.

5) **Define Inner Functions**

Inner functions are functions which are utilized in PySpark RDD functions. If we define them out of the "c2lsh()" function, each slave will get a copy of them and there is no memory sharing between slaves. This will result in an error output.

3. Evaluation Results

1) **Test Cases Generation**

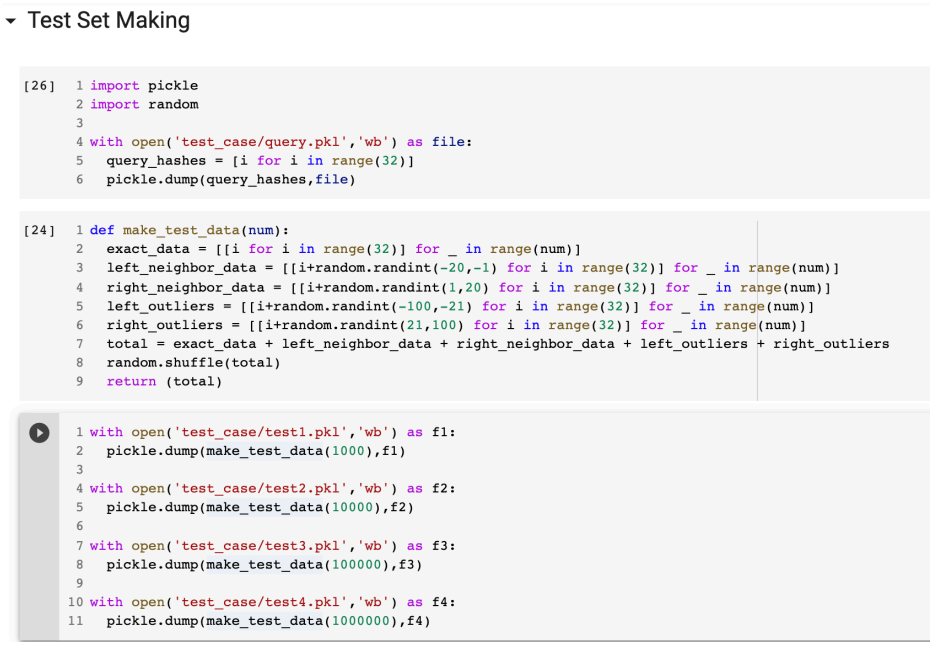

Figure 1. Test set generation code on Google Colab

In this part, we try to generate test cases with patterns like the "toys" dataset. We generate a 32-number list as the query hashes. We define a function to make test data cases. In the *make_test_data(num)* function, we generate *num* exact data hashes which are exactly the same as the query hashes. After this, left neighbor data and right neighbor data hashes are generated to examine the effect of αm later. Eventually, outlier data hashes are generated, for augmenting the size of test case data.

2) **Toy Dataset Experiment**

$\alpha m = 10, \beta n = 10$
running time: 1.3142969608306885 seconds
Number of candidates: 10
set of candidates: {0, 70, 40, 10, 80, 50, 20, 90, 60, 30}

$\alpha m = 20, \beta n = 10$
running time: 1.811398983001709 seconds
Number of candidates: 100
set of candidates: {0, 1, 2, 3, 4, …, 97, 98, 99}

$\alpha m = 30, \beta n = 10$
running time: 1.879776954650879 seconds
2.591852903366089
Number of candidates: 10
set of candidates: {0, 70, 40, 10, 80, 50, 20, 90, 60, 30}

$\boldsymbol{\alpha m = 10, \beta n = 20}$
running time: 1.8128139972686768 seconds
Number of candidates: 100
set of candidates: {0, 1, 2, 3, 4, …, 97, 98, 99}

$\boldsymbol{\alpha m = 10, \beta n = 30}$
running time: 1.9896540641784668 seconds
Number of candidates: 100
set of candidates: {0, 1, 2, 3, 4, …, 97, 98, 99}

From the results above, it could be seen that when αm increases, the running time also grows. And the number of candidates changes along with αm. If we print out the values of αm, current offset and the number of candidates when we have $\#\, number\ of\ candidates \geq \beta n$ and $\beta n = 10$, we have the table below.

$\boldsymbol{\alpha m}$	**Offset**	**Number of Candidates**
10	10	10
20	21	100
30	21	10

Table 1. The relationship between αm, offset and number of candidates

According to the statistics in Table 1, it could be seen that the bigger αm we have, the more loops are likely to be required to retrieve sufficient candidates. By increasing the value of offset, we broaden the range of possible candidates.

On the other hand, when βn increases, the running time also rises. Because more loops are demanded to find out a proper value of offset to expand the candidate range. As for $\boldsymbol{\alpha m = 10, \beta n = 20}$ and $\boldsymbol{\alpha m = 10, \beta n = 30}$, they both collect enough candidates when offset is 21. Why they have different running times could lie in the temporal CPU performance's difference.

3) **Test Case 1 Experiment**

In this dataset, there are **5,000** data hashes.

$\boldsymbol{\alpha m = 10, \beta n = 10}$
running time: 1.1136586666107178 seconds
Number of candidates: 1000
set of candidates: {2050, 3, 4, 2052, 6, …, 4089, 2042, 4092}

$\boldsymbol{\alpha m = 20, \beta n = 10}$
running time: 1.3588688373565674 seconds
Number of candidates: 1000
set of candidates: {2050, 3, 4, 2052, 6, …, 4089, 2042, 4092}

$\boldsymbol{\alpha m = 30, \beta n = 10}$
running time: 1.1530859470367432 seconds
Number of candidates: 1000
set of candidates: {2050, 3, 4, 2052, …, 4089, 2042, 4092}

$\boldsymbol{\alpha m = 10, \beta n = 2000}$
running time: 1.9336678981781006 seconds
Number of candidates: 2470

set of candidates: {3, 4, 7, 8, …, 4996, 4998, 4999}

$\alpha m = 10, \beta n = 3000$
running time: 2.8485209941864014 seconds
Number of candidates: 3000
set of candidates: {3, 4, 7, 8, …, 4996, 4998, 4999}

$\alpha m = 10, \beta n = 4000$
running time: 2.2914249897003174 seconds
Number of candidates: 4129
set of candidates: {2, 3, 4, 6, …, 4996, 4998, 4999}

$\alpha m = 10, \beta n = 5000$
running time: 11.017954111099243 seconds
Number of candidates: 5000
set of candidates: {0, 1, 2, 3, …, 4997, 4998, 4999}

In test case 1, when βn is set to be 10, if we print out the offset value and the number of candidates when $\#\ number\ of\ candidates \geq \beta n$, we can see the offset is 1 and the number of candidates is 1000. Hence, for the first three records, the time difference could consist in the temporal CPU performance.

When βn increases, the offset loop number also grows, which leads to longer execution time.

4) **Test Case 2 Experiment**

In this dataset, there are 50,000 data hashes.

$\alpha m = 10, \beta n = 10$
running time: 1.4284422397613525 seconds
Number of candidates: 10000
set of candidates: {2, 4, 32773, 8, 10, …}

$\alpha m = 20, \beta n = 10$
running time: 1.6152188777923584 seconds
Number of candidates: 10000
set of candidates: {2, 4, 32773, 8, 10, …}

$\alpha m = 30, \beta n = 10$
running time: 1.8253111839294434 seconds
Number of candidates: 10000
set of candidates: {2, 4, 32773, 8, 10, …}

$\alpha m = 10, \beta n = 20000$
running time: 4.335103988647461 seconds
Number of candidates: 24610
set of candidates: {1, 2, 3, 4, 7, 8, 9, 10, 12, 13, …}

$\alpha m = 10, \beta n = 30000$
running time: 9.196204900741577 seconds
Number of candidates: 30000
set of candidates: {1, 2, 3, 4, 7, 8, 9, 10, 12, 13, …}

$\alpha m = 10, \beta n = 40000$
running time: 44.59819197654724 seconds
Number of candidates: 40000
set of candidates: {1, 2, 3, 4, 7, 8, 9, 10, 12, 13, …}

$\alpha m = 10, \beta n = 50000$
running time: 65.13952922821045 seconds
Number of candidates: 50000
set of candidates: {1, 2, 3, 4, 7, 8, 9, 10, 12, 13, …}

The pattern in test case 2 is similar to test case 1. But as the size of dataset is 10 times larger than test case 1, when the value of βn increases, we can see the execution time boosts up dramatically.

5) **Test Case 3 Experiment**

In this dataset, there are 500,000 data hashes.

$\alpha m = 10, \beta n = 10$
running time: 5.611429929733276 seconds
Number of candidates: 100000
set of candidates: {262145, 3, 6, 262156, 13, 21, …}

$\alpha m = 20, \beta n = 10$
running time: 5.614091634750366 seconds
Number of candidates: 100000
set of candidates: {262145, 3, 6, 262156, 13, 21, …}

$\alpha m = 30, \beta n = 10$
running time: 6.006803274154663 seconds
Number of candidates: 100000
set of candidates: {262145, 3, 6, 262156, 13, 21, …}

$\alpha m = 10, \beta n = 200000$
running time: 41.52479887008667 seconds
Number of candidates: 246240
set of candidates: {0, 3, 5, 6, 8, 12, 13, 17, …}

$\alpha m = 10, \beta n = 300000$
running time: 90.55144596099854 seconds
Number of candidates: 300000
set of candidates: {0, 3, 5, 6, 8, 12, 13, 16, …}

$\alpha m = 10, \beta n = 400000$
running time: 404.8549349308014 seconds
Number of candidates: 413037
set of candidates: {0, 2, 3, 4, 5, 6, 8, 9, 10, 12, …}

$\alpha m = 10, \beta n = 500000$
running time: 718.1928441524506 seconds
Number of candidates: 500000
set of candidates: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, …}

The pattern is similar to test cases 1 and 2.

6) **Test Case 4 Experiment**

In this dataset, there are 5,000,000 data hashes.

In this test case, because its size is comparatively big and there is not enough computer memory for its calculation, the experiment cannot be conducted.

4. Improvement in Algorithm

In normal cases, we apply *filter()* function to find out data instances whose absolute value of its data hash minus query hashes is smaller than the given offset. Then we use *map()* function to get the IDs of candidates. After that, we check the count of candidates is greater or equal to the βn or not and decide whether to increase the offset by one and start a new loop or return the candidate IDs as a RDD.

It could be easily noticed that the intuitive algorithm calculates the difference between data hashes and query hashes for multiple times which sacrifices the efficiency, and this algorithm would also repeatedly check each data instance's hashes even an instance is classified as "collided" in previous loops. To reduce the calculation times, the idea of getting a difference first, and introducing a flag denoting successful collision is established.

Before starting the loop to find out enough candidates, we calculate the absolute differences first, so that we do not need to repeat this calculation again. Also, in the loop of finding candidates, we update the flag to be True if an instance's hash value satisfies collision under the current offset. By doing this, we do not check instances with True flags again to see whether it collides with the query hashes or not.

This algorithm works well on large size RDDs. If the target RDD has a small size, it would be better to apply the combination of *filter()* and *map()*. Because the overhead of updating data hashes consumes more time than the saved time from repeat collision checking.

Attempts on using *mapPartitions()* function have been tried, but it does not work well on the toy dataset as well the test cases. The running time is given below. It reveals that *mapPartitions(),* though can take the whole RDD once and apply iterator to each element in the RDD, reducing the time of reading, is not suitable for this project.

Dataset	Running Time
toy	3.45 seconds
test 1	2.86 seconds
test 2	13.76 seconds
test 3	75.05 seconds

Table 2. Trials of using mapPartitions() function