2020 T2 COMP9313 Project 1

My Pseudocode of c2lsh()

Def c2lsh():

> *Find the **smallest offset** from data hashes that returns enough candidate (> beta_n)*
>
> *Use that **offset** to get all the candidates that meets the offest*
>
> ***Return** all of those candidates*

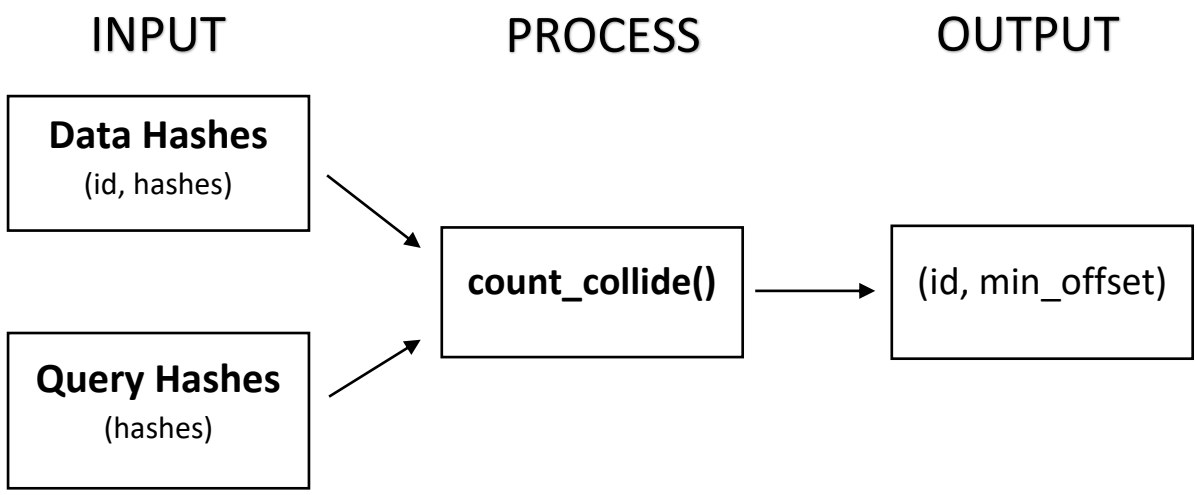
So, the goal is simple. **Find the minimum offset that returns enough candidates more than beta_n.**

My Implementation of c2lsh()

There are 3 key parts in my interpretation of **c2lsh().**

The 1st part is using map transformation to compute new value for each data hashes via my custom function **count_collide().** The main objective of that function is to find the most minimum offset for each data hashes such that it can be a candidate by meeting **alpha_n** which represents the minimum amount of collision between data hashes and query hashes.

Function **count_collide()** returns **(ID, minimum offset)** for each data hashes.



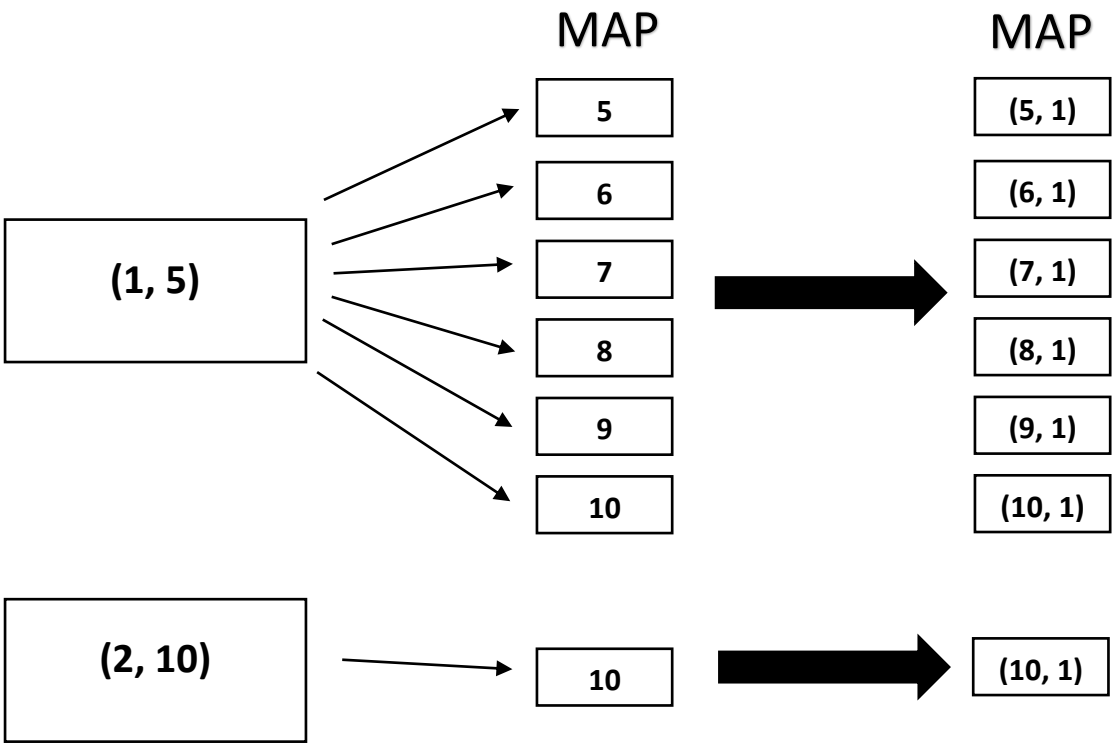
Then I use **action max()** to get the maximum offset from all the data hashes which is then stored in the variable **max_offset**.

The 2nd part is using **map reduce technique** to find how many data hashes fits under each offset. Using variable **max_offset**, we use a **flat map transformation** which expands to all acceptable offsets for each data hashes, ranging from its minimum offset to **max_offset**.

Then I use another **map transformation** to give each offset a value of 1.

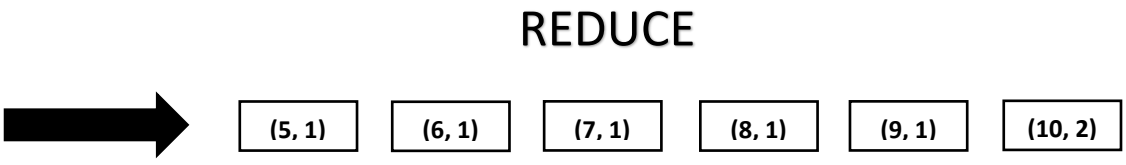
*Example – ((1, 5), (2, 10)) from **collide_count()***

max_offset = 10



The next part is **reducing**. I use **reduceByKey**() **transformation** to combine similar key by accumulating its value and reduce data storage.

This also shows how many data hashes are candidate within each offset.



I then use filter function where I filter out offsets that does not meet the required minimum number of candidates (less than **beta_n**) and then get the minimum key via **action min()** which should be our minimum offset to find enough candidates (**beta_n**), which will be stored as variable **offset**.

We then use another **map transformation** to get all ID that has offset equal/lower than **offset**.

In this example, assume the beta_n is 2 or we need 2 candidates.

From there, we discard offset 5, 6, 7, 8, 9 as they do not include 2 candidates.
However, 10 does so 10 is the minimum offset to get all required minimum candidates.

Evaluation result of my implementation

The base of all of those cases are provided in Piazza. But each test cases except the toy sample are manipulated to test each key condition.

Link – https://piazza.com/class/kamb1tqxe9h6np?cid=149

Toy Sample (Provided)

```
running time: 20.73741102218628
Number of candidate:  10
set of candidate:  {0, 70, 40, 10, 80, 50, 20, 90, 60, 30}
```

2nd Test Sample – All Same Data Hashes

- Number of Collision = 10, Number of Candidate = 50
- All data hashes are exactly the same (100)
- Dimension per each data hashes – 20

```
running time: 26.08004093170166
Number of candidate:  100
set of candidate:  {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 2
8, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 5
9, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 9
0, 91, 92, 93, 94, 95, 96, 97, 98, 99}
```

3rd Test Sample – 1000 Random Data Hashes

- Number of Collision = 5, Number of Candidate = 100
- 1000 Data Hashes (All randomised via seeds)
- Dimension per each data hashes – 30

```
running time: 21.359761476516724
Number of candidate:  50
set of candidate:  {130, 643, 772, 261, 773, 10, 782, 534, 663, 24, 792, 154, 27, 284, 285, 540, 667, 926, 291, 932, 549, 93
3, 307, 825, 698, 62, 708, 198, 455, 202, 208, 337, 722, 211, 723, 724, 980, 87, 92, 221, 989, 353, 745, 106, 363, 876, 245,
249, 507, 639}
```

4th Test Sample – Big Dimension

- Number of Collision = 5, Number of Candidate = 50
- 1000 Data Hashes (All randomised via seeds)
- Dimension per each data hashes – 200

```
running time: 21.788774013519287
Number of candidate:  61
set of candidate:  {256, 1, 898, 131, 387, 899, 774, 392, 908, 397, 274, 531, 790, 665, 410, 538, 669, 159, 927, 33, 291, 16
4, 549, 931, 937, 556, 818, 692, 821, 956, 61, 829, 321, 834, 584, 712, 335, 849, 981, 216, 859, 989, 350, 607, 480, 353, 86
9, 486, 359, 489, 362, 238, 113, 114, 371, 501, 629, 375, 886, 891, 892}
```

5th Test Sample – Small Dataset

- Number of Collision = 10, Number of Candidate = 20
- 100 Data Hashes (All randomised via seeds)
- Dimension per each data hashes – 10

```
running time: 21.383513927459717
Number of candidate:  20
set of candidate:  {0, 1, 2, 32, 34, 5, 35, 36, 37, 41, 13, 47, 16, 17, 48, 20, 21, 25, 27, 31}
```

6th Test Sample – Huge Dataset with Huge Volume

- Number of Collision = 10, Number of Candidate = 100
- 20,000 Data Hashes (All randomised via seeds)
- Dimension per each data hashes – 100

```
running time: 27.180132627487183
Number of candidate:  120
set of candidate:  {4610, 10755, 14341, 8204, 11285, 6679, 12826, 4129, 547, 12835, 9765, 562, 6196, 1077, 9780, 6711, 1089,
9793, 17992, 17482, 7243, 7778, 5229, 7792, 19064, 9857, 8852, 16532, 14486, 3740, 3229, 3234, 5794, 9388, 15536, 2228, 376
4, 184, 13507, 12484, 10951, 17095, 10953, 14025, 1227, 15573, 19669, 14040, 6361, 8421, 9448, 10472, 18161, 19189, 4855, 74
17, 250, 18683, 19200, 6920, 19735, 280, 1305, 15649, 12071, 1324, 18732, 815, 10547, 7476, 3893, 1852, 14654, 5441, 12099,
4420, 2376, 10569, 12617, 13640, 5974, 16221, 11616, 3425, 11631, 881, 8051, 7031, 8571, 2442, 10638, 16784, 14232, 13730, 6
057, 5547, 13746, 14259, 12218, 14780, 18876, 959, 14798, 14799, 7644, 9697, 482, 12264, 5609, 10217, 11753, 2028, 13806, 15
344, 11249, 11251, 18425, 7675, 8190, 2559}
```

How do I improve my implementation?

Based on the pseudocode from lecture note, I test and found that the greedy algorithm provided by lecturer takes around 60 seconds to find candidates nearest to the data query in the toy data, a very slow and unreasonable time.

From the pseudocode, I know that if I find the minimum offset, it will be easy from then on. I found it to be a big issue because it involves a series of map and reduce transformation to find the minimum offset. That way, we do not keep doing action for every offset from 1 to infinity till they found enough candidates. And the constant usage of action seriously impacts the running time and it shows.

So, the use of map and reduce transformation and not use for loop and any greedy methods in my algorithm cut down the running time by a half but I believe that I could done it better as explained more below.

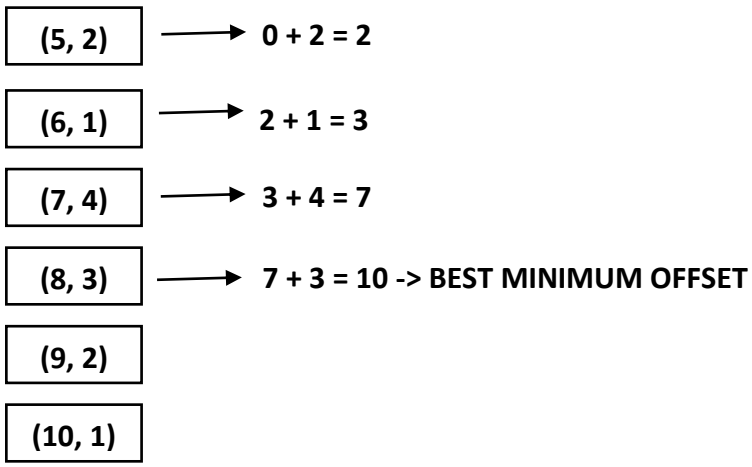
The main performance issue is the **flat map function** where we expand each offset to all acceptable offsets for each data hashes, ranging from its minimum offset to the variable **max_offset.** While it works, it is a naïve solution as the space complexity **O(n^2)** where **n** is the maximum offset in the data hashes.

This issue is prevalent in 1st and 5th test sample where with small data, my implementation needlessly creates a lot of unwanted trouble and went into over 20 seconds.

The best solution is to sort key (offset) in ascending order. Using a variable that stored the current amount of data hashes which satisfies the current offset, we keep iterating till there is enough candidates under the current offset. That way, we will save storage from unnecessary data and hence should be quicker to process and access data.

However, due to my limited knowledge of PySpark and blocked function list, I struggled to find an effective way to implement that solution and stuck with the old and current solution.

Example of my preferred method – get at least 10 candidates



Overall, I don't think it affects the running time a lot, but it will save a lot of data space and processor. Other processes don't need to be updated to be faster and I'm happy with it.