Yuan Gao z5239220 Report

1. Implementation details of the C2LSH algorithm

> In this project, C2LSH algorithm is divided three function to implement `absDiff(),isSatisfy(),c2lsh()`.
> The main algorithm is **binary search**

> **a) In the function `absDiff()`**

```
def absDiff(data_hash, query_hash):
    length = len(data_hash)
    res_arr=[]
    for i in range(length):
        res_arr.append(abs(data_hash[i] - query_hash[i]))
    return res_arr
```

> This function generates the list that stroe the Absolute difference of `data_hash` and `query_hash`
>
> - Firstly, use `map()` modify the element of `data_hashes` to the Absolute difference

> **b) In the function `isSatisfy()`**

```
def isSatisfy(difference, alpha_m, offset):
    num = 0
    for value in difference:
        if value <= offset:
            num += 1
        if num >= alpha_m:
            return True
    return False
```

> Determine whether the current dataCode satisfies the the number of `alpha_m`
> This function is mainly to determine whether current data_hash satisfied the offset(number of hashcodes) conditions.

> **c) In the function `c2lsh()`**

```
def c2lsh(data_hashes, query_hashes, alpha_m, beta_n):
    numCandidates = 0
    data_hashes = data_hashes.map(lambda x : (x[0], absDiff(x[1],
query_hashes)))
    data_Maxcode = data_hashes.flatMap(lambda x : [max(x[1])]).max()
    left = 0
    right = data_Maxcode
    while left < right:
```

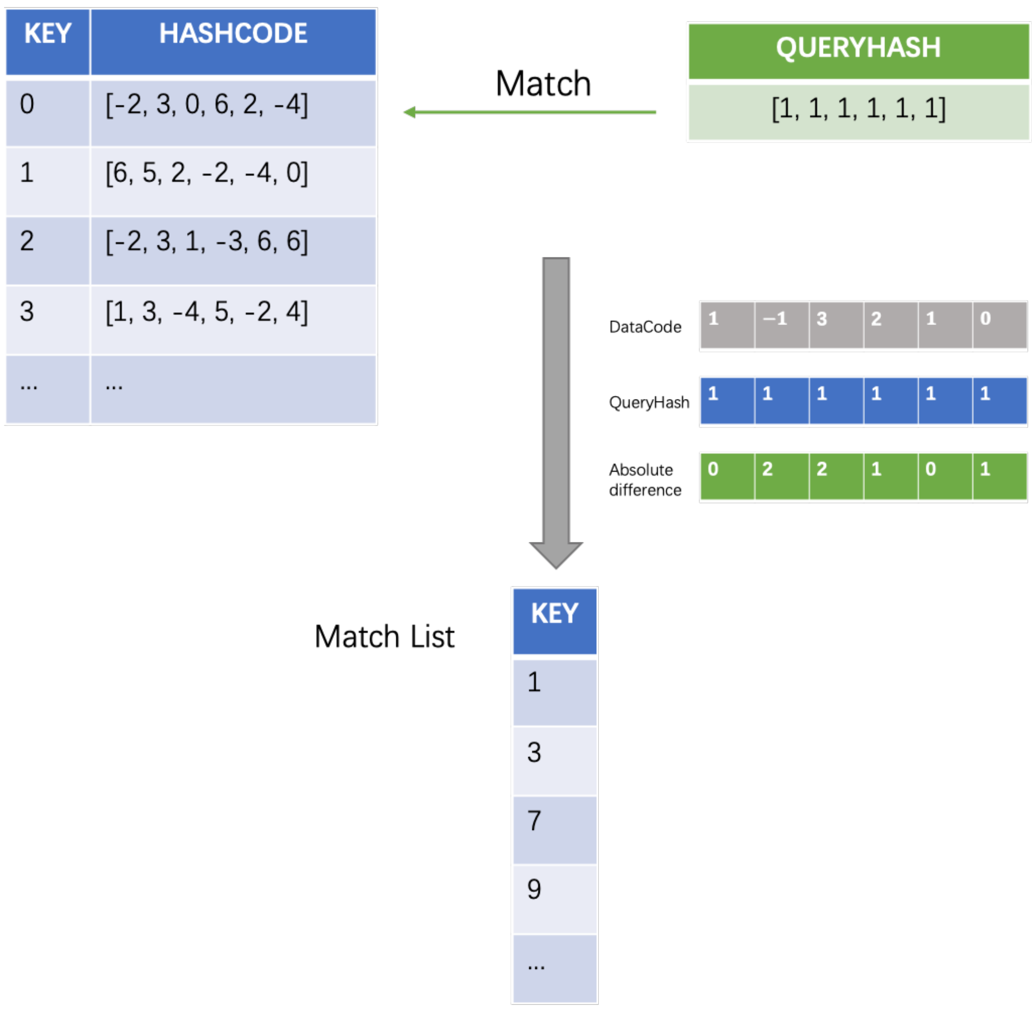
```
        offset = (left + right) // 2
        candidatesRDD = data_hashes.flatMap( lambda x : [x[0]] if
isSatisfy(x[1], alpha_m, offset) else [])
        numCandidates = candidatesRDD.count()

        if numCandidates == beta_n:
            break

        if numCandidates <= beta_n:
            left = offset + 1
        else:
            right = offset

    if numCandidates < beta_n:
        offset += 1
        candidatesRDD = data_hashes.flatMap( lambda x : [x[0]] if
isSatisfy(x[1], alpha_m, offset) else [])
        numCandidates = candidatesRDD.count()

    return candidatesRDD
```



This is the main function:

- Create `data_hashes` to store the absolute value of the difference between

> "HashCode" and "QueryHash"
> - Get the left boundary and right boundary(0 and the maximum value of HashCode)
> - Iterate, use `flatMap()` selects the KeyNum of `data_hashes` which satisfied the `isSatisfy()` and use binary search to get `offset`
> - Count elements of `candidatesRDD`. If the number of element equal to `beta_n`, exit loop.

2. The evaluation result of implementation using own test cases

> **Generate test cases:**

```
import pickle
import random

one_arr = [[1 for _ in range(16)] for _ in range(100000)] #32个0, 5w个
mfive_five = [[random.randint(-4, 6) for _ in range(16)] for _ in
range(100000)]
otRand = [[random.randint(10, 20) for _ in range(16)] for _ in range(100000)]
arr = one_arr + mfive_five + otRand
random.shuffle(arr)

with open("testQuery.pkl", 'wb') as f:
    pickle.dump([1 for _ in range(16)], f)

with open("testCase.pkl", 'wb') as f:
    pickle.dump(arr, f)
```

> There are two test file:
>
> - "testQuery.pkl"
>
> > -Stored 16 digits HashCode(query)
> > -It consists of sixteen "1"
>
> - "testCase.pkl"
>
> > -Stored 16 digits HashCode
> > -Total number of cases are 300,000
> > -100,000 every digit is "1"
> > -100,000 every digit random between -4 to 6 (abs(1 - 6) && abs(1 - (-4)))
> > -100,000 every digit random between 10 to 20

> **Result:**
> Set `alpha_m  = 10`, `beta_n = 200000`
> Add `print("Offset: ", offset,  "NumCandidates: ", numCandidates)` in the `c2lsh()`

> ```
> Stage 0 contains a task of very large size (1954 KB). The maximum
> recommended task size is 100 KB.
> Stage 1 contains a task of very large size (1954 KB). The maximum
> recommended task size is 100 KB.
> offset:  19 numCandidates:  200000
> Stage 2 contains a task of very large size (1954 KB). The maximum
> recommended task size is 100 KB.
> running time: 2.160277843475342
> Number of candidate:  200000
> ```
>
> The result matched *100,000 every digit is "1"* and *100,000 every digit random between -4 to 6*

3. Efficient improvement of implementation

> **a) `filter` version**
> This version use the `filter()` function

```python
def c2lsh(data_hashes, query_hashes, alpha_m, beta_n):
    offset = 0
    numCandidates = 0
    while numCandidates < beta_n:
        candidateRDD = data_hashes.filter(lambda x : isQualified(x[1],
query_hashes, alpha_m, offset)).map(lambda x : x[0])
        numCandidates = candidateRDD.count()
        offset += 1
    return candidateRDD
```

> This method results in subtraction calculations for each iteration

> **b) The second prioritization scheme:**
> This method create extra bool element in `data_hashes` which recorded the access status (accessed = TRUE / not accessed = FALSE)

```python
data_hashes = data_hashes.map(lambda x: (tuple(x[1]),
x[0])).groupByKey().map(lambda x: (x[0], x[1], False))
```

> **c) The version without binary search:**

```python
def isSatisfy(dataHash, queryHash, alpha_m, offset):
    num = 0
    length = len(dataHash)
    for i in range(length):
        if (abs(dataHash[i] - queryHash[i]) <= offset):
            num += 1
    return num >= alpha_m

def c2lsh(data_hashes, query_hashes, alpha_m, beta_n):
    offset = 0
```

```
    numCandidates = 0
    while numCandidates < beta_n:
        candidatesRDD = data_hashes.flatMap( lambda x : [x[0]] if 
isSatisfy(x[1], query_hashes, alpha_m, offset) else [])
        numCandidates = candidatesRDD.count()
        print("offset: ", offset,  "numCandidates: ", numCandidates)
        offset += 1
    return candidatesRDD
```

> This function just use `flatMap` and iteration

> **Test case:**
> In addition to the above two test cases:
> c) Add a 16 digits and 1,000,000 HashCode as an addtional test case
> d) Add an extreme example(32 digits 100,000 HashCode):
>
> ```
> one_arr = [[1 for _ in range(32)] for _ in range(5000)]
> mfive_five = [[random.randint(-999, 1001) for _ in range(32)] for _ in 
> range(5000)]
> otRand = [[random.randint(-10000, 10000) for _ in range(32)] for _ in 
> range(90000)]
> ```

> **Compare:**
>
> | | a) Toy Runing Time(s) | b) 300,000 Test Runing Time(s) | c) 1,000,000 Test Runing Time(s) | d) 100,000 Test Runing Time(s)(32 digits) |
> |---|---|---|---|---|
> | `filter()` | 0.51 | 3.66 | 10.65 | 175.32 |
> | Has Bool Marked | 0.87 | 7.34 | 18.41 | 203.23 |
> | `flatMap()` and binary search | 0.82 | 2.16 | 5.51 | 4.71 |
> | `flatMap()` and iteration | 0.47 | 3.58 | 7.01 | 173.79 |
>
> The most efficinet C2LSH algorithm is "`flatMap()` and binary search".