
UNSW
AUSTRALIA

**Project1**

**COMP9313 Big Data Management**

**T2, 2020**

**z3290805 Pei Wang**

**University of New South Wales**

**Faculty of Computer Science and Engineering**

**July 2020**

**1. Implementation details of your c2lsh(). Explain how your major transform function works.**

In this implementation, c2lsh is a binary search wrapper of c2lsh_stride. This will be discussed in further details in the answer of question 3. The major transformation within c2lsh_stride() involves a map and a filter. A “count” action on the RDD is also involved.

In the beginning, the map transformation the calls count to generate a "c" number of matching hash values between the query hashes and the data hashes. This number is then added as the third item of the tuple in the data_hash.

Secondly, the filter transformation removes the data hashes having a “c” value less than alpha_m.

Lastly, the count action counts the number of records that is left in the filtered RDD against beta_n. The offset will be incremented if it is less than beta_n.

The function returns two variables offset and test. Offset is used for binary search. Test is the output of the filter transformation, which has a narrowed down scope for future iteration of the binary search.

**2. Show the evaluation result of your implementation using your own test cases.**

Two sets of tests were written first handedly. The first set was written to test count, which implicitly tests against the value set for alpha_m. The hashes in the query was kept static. The first 30 values are 10, followed by 30 values incremented by 1. Values of offset were changed and the count output was compared against expected values.

The second set was written to test c2lsh against larger hash values, which implicitly tests against the value set for beta_n. The value of offset has also been tested by using print. It has several corner cases including when the query hashes overlaps with a data hashes, where the hashes in the query (1s) are significantly smaller than the hashes in the data_hashes(over 10000).

```
# Set1
from submission import count
num = 100
myquery = [10]*30
for i in range(11,41):
        myquery.append(i)
mydata = [12]*60
c = count(mydata, myquery, 3)
assert(c == 35)

mydata = [10]*60
c = count(mydata, myquery, 3)
assert(c == 33)

mydata = [15]*60
c = count(mydata, myquery, 3)
```

```
assert(c == 7)

mydata = [40]*60
c = count(mydata, myquery, 5)
assert(c == 6)

mydata = [100]*60
c = count(mydata, myquery, 5)
assert(c == 0)

#Set2
def test_c2lsh(val, inc,  data, query_hashes):
        print('test with val', val)
        print(len(data))
        print(len(query_hashes))
        mydata = []
        num = 10000
        for i in range(len(data)):
        l = []
        for i in range(len(query_hashes)):
        l.append(num)
        mydata.append(l)
        num +=inc
        myquery = [val]* len(query_hashes)

        data = mydata
        query_hashes = myquery

        sc = createSC()
        data_hashes = sc.parallelize([(index, x) for index, x in enumerate(data)])
        res = submission.c2lsh(data_hashes, query_hashes, alpha_m, beta_n).collect()
        sc.stop()
        return res

res = test_c2lsh(10000, 100, data, query_hashes)
assert (set(res) == {0, 1, 2, 3, 4, 5, 6, 7, 8, 9})

res = test_c2lsh(15000, 100, data, query_hashes)
assert (set(res) == {45,46,47,48,49,50,51,52,53,54,55})

res = test_c2lsh(14950, 100, data, query_hashes)
assert (set(res) == {45,46,47,48,49,50,51,52,53,54})

res = test_c2lsh(1, 1, data, query_hashes)
assert (set(res) == {0, 1, 2, 3, 4, 5, 6, 7, 8, 9})
```

**3. What did you do to improve the efficiency of your implementation?**

My implementation of c2lsh is a binary search wrapper of c2lsh_stride. It improves the time complexity of the function from O(k*n) to O(k*logn), where k is the number of hashes in a query and n is the number of offset being tested.

Firstly, the target being searched for is the offset value. It is called "stride" because it modifies the offset value by a value more than 1. The original solution only increments the offset value by 1 which is massively inefficient. It will not work well against a hash value scaled up to millions. Hence, The algorithm needs to be smart enough to scale with the value of hashes. The stride value was initialized as one-fifth of the max hash value in a random sample from data_hashes or one-fifth of the max hash value of the query. The larger one would be chosen. Despite being mathematically imperfect, it is a successful attempt to allow stride values scaling with the value of hashes. Once the function c2lsh_stride finds a large offset value that will pass the alpha m and beta n requirement, it will start cutting the search space of offset in half as what a binary search would do.

Secondly, the size of the data hashes would also be reduced: Once a set of data hashes has been generated by an offset that is too large, it would be a super set including all the data hashes that would meet the requirement of a smaller offset value. This reduces the search space even progressively.

Lastly, since we are looking for a number of sets of hashes that meet the beta_n. The value of offset no longer matters if the amount is equal to beta_n. Hence an early exit was implemented: c2lsh_stride returns a third parameter to indicate whether there is an exact match to beta_n.

As the outcome, the toy test case could be done in 8.8 sec improved from > 10 seconds. It could also handle scenarios covered in 2.