COMP9313 Big Data Management

Project 1

Report

z5240221 Jatin Gupta

Table of Contents

1 Ques 1

1.1 Implementation details of your c2lsh()

My implementation of c2lsh function was written with a lot of iterations. I started working on the problem as soon as the project was released. During the complete implementation I have iterated from simple python script for doing the complete function to the final submission using pyspark functions. Initially I approached the code with the same approach as lab 2, by mapping through all the hashes and building a set and adding new candidates to the set but it was quite slow as it was taking 500 sec on my system.

I switch the implementation with filter method. In this approach, I iterated through all the hashes and mapped through all of them, if they satisfy as the candidate, my code will return the id of the hash else, it will return None. Later, I would filter on the values which are not None using filter function. Both approaches used simple for loop in python to iterate through all indexes of query and current hash value simultaneously and count the number of matches if it is less or equal to offset. The return of count function would return a count which can be compared with alpha_m value and hence decide if it to be used as a candidate or not. This implementation would take O(2N*M) for each offset where N is the number of hashes and M is the dimension of the query and hash values and N >> M.

Then for the next implementation I reduced the time by using while loop and breaking the loop as soon as alpha_m is satisfied in the count function. Also, I switch from using map and then filtering to filtering and in the final step when I am about to return to the caller, I use map for only getting the ids of the hashes.

For the final implementation, I used binary search to find the offset which satisfies the query. I used one pass where I increment the offset by the power of 2 and reach a point where the candidates are equal or more than beta_n. And then using binary search back track to the correct offset value by reducing the search space by half for each iteration.

1.2 Explain how your major transform function works.

My major transformation is filter method which filters the candidate for output. It takes in hash values query, offset, alpha_m. Parameter alpha_m is used as a stopping condition for the while loop which searches correct candidate for the query using offset. The count collision function returns true if the candidate is accepted as the match for the query for the provided offset else it sends false which deletes the candidate from the resultant rdd.

At last as the calling function is only interested in keys of the hashes, I use map to get only the ids of the candidate hashes.

2 Ques 2

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

Though I have tested this program with numerous test cases, but I am only including two cases for simplicity. I have also debugged the code with pycharm debugger to see the internal working which helped me to understand how spark works.

```
def generate_test_case(dim, count, seed, start=-1000, end=1000):
    random.seed(seed)

    data = [
        [
            random.randint(start, end)
            for _ in range(dim)
        ]
        for i in range(count)
    ]

    query = [random.randint(start, end) for _ in range(dim)]

    return data, query

alpha_m, beta_n = 10, 10

# data for output 1
data, query = generate_test_case(15, 20000, 7, -1000, 1000)
# data for output 2
data2, query2 = generate_test_case(20, 20000, 7, -10000, 10000)
```

Output 1

```
running time: 46.34946060180664
Number of candidate:  10
set of candidate:  {11360, 11745, 8642, 14688, 8516, 12452, 16169, 6703, 14868, 1078}
```

Output 2

```
running time: 56.23536252975464
Number of candidate:  10
set of candidate:  {1664, 11712, 12096, 7429, 1799, 16786, 13108, 10805, 15991, 9176}
```

3 Ques 3

3.1 What did you do to improve the efficiency of your implementation?

My implementation improved efficiency in 3 aspects. They are

1. By using alpha_m to stop the searching for matches. In the implementation given in the notes, the count function searches all the items in query even if it has counted beyond alpha_m. Hence, I have used alpha_m as the stopping condition to improve the efficiency.
2. I filter the dataset first and then map the dataset.
3. I used to increase the offset with the power of 2 and then backtrack to find the correct offset value. I divide the search space into half by using binary search which was the most significant change to make the code faster.