CS6240
Soumyadeep Sinha

# Assignment -2Report

**PROBLEM ANALYSIS:**

**Cardinality:**

**Explanation:**

Find the number of incoming and outgoing edges for every node. This way we can determine if Y has m incoming edges—from followers x1,…, xm—and n outgoing edges—to followed users z1,…, zn—then there exactly m*n length-2 paths through Y—one for each combination (xi, Y, zj). This implies that if we can efficiently count the incoming and outgoing edges for each user.Now we can take the sum of this count to determine the cardinality.

**PseudoCode:**

```
class CardinalityMapper
        method Map(doc edge)
                for all users ∈ edge e do
                        user = users.split(",")
                        from =  user[0]
                        to = user[1]

                        If from < MAX_FILTER &&  to < MAX_FILTER
                                Emit(from, "O")
                                Emit(to, "I")




class CardinalityReducer
        method Reduce(user , counts [O, O, . I. ., I, O])
                For list of O and I for a user:
                        If O:
                            countO++
                        Else:
                            countI++
                emit(user, countO*countI)
```

```
Class ComMapper
        method Map(doc intermediate)
                for all count ∈ intermediate e do
                        c = count.split(“,”)
                        cnt = c[1]
                        Emit(”Sum”, cnt)

Class ComReducer
                method Reduce(Sum , counts [1, 3, . 320. .])
                        Sum <- 0
                        for all cnt ∈ intermediate e do
                                Sum = Sum+cnt
                        Emit(”Cardinality ”, Sum)
```

| MAX_VALUE | CARDINALITY |
|---|---|
| 1000 | 400016 |
| 10000 | 73597234 |
| 1000000 | 104405319023 |
| 11316811 | 953138453592 |

| | RS join input | RS join shuffled | RS join output | Rep join input | Rep join file cache | Rep join output |
|---|---|---|---|---|---|---|
| **Step 1 (join of Edges with itself)** | 950 B~ | 950 B~ | 950 B~ | 950 B~ | 950 B~ | 950 B~ |
| **Step 2 (join of Path2 with Edges)** | 950 B~ | 950 B~ | 950 B~ | N/A | N/A | N/A |

## JOIN IMPLEMENTATION:

### MAX FILTER

**Explanation:**

Read the file and in the map function emit only those edges where from and to values are less than the MAX_FILTER.

**Pseudo-Code:**

```
class MaxFilterMapper
        method Map(doc edge)
                for all users ∈ edge e do
                        user = users.split(“,”)
                        from =  user[0]
                        to = user[1]

                        If from < MAX_FILTER &&  to < MAX_FILTER
                                Emit(from, to)


class MaxFilterReducer
         method Reduce(from , counts [u1, u2, . . ., un])
                Emit(from, to)
```

### REDUCE SIDE JOIN

**Explanation:**

Input:

```
1,2
2,3
3,1
```

First Phase:
In the first Phase of the mapper we will split the user graph and emit the from user as key and (from user, to user) as value and to user as key and (from user and to user ) as value.
Mapper output:

1: (1,2)
1: (3,1)
2: (2,3)

2: (1,2)
3: (2,3)
3: (3,1)

In the first Reducer, for every from user create a hash table or lookup with from user as key and a list of to users as values.
So the hash map would look like for 1's iteration:
1 [2],  3[1]

Now we go through the list of users for various keys in the hashmap and see if we can reach the neighbouring node from that node.

Here, we will get no path for 2.
But for 1 we get:

3 -> 1 -> 2.

Now, lets see what happens when the key is 2.

Hashmap:  2[3] 1[2]

The result:  1 -> 2 -> 3

Similarly, when the key is 3.

Hashmap:  3[1] 2[3]

The result:  2 -> 3 -> 1

Thus, we get 3 paths from the reducer.

3 -> 1 -> 2
1 ->2 -> 3
2 -> 3 -> 1

Once we got all the Two- paths. We can get the original edges and the new Two path and combine them in the mapper.

So the output of the second mapper would look something like:

1 : (1 -> 2 -> 3)
1: (3 -> 1)
1: (2 -> 3 -> 1)

1: (1 -> 2)

2: (2 -> 3 -> 1)
2: (2 -> 3)
2: (3 -> 1 -> 2)
2: (1 -> 2)

3: (3 -> 1 -> 2)
3: (3 ->1)
3: (2 -> 3)
3: (1 -> 2 -> 3)

In reducer we make two hashmap

Threepath = 1[3]
Twopath = 3[1]

Now we iterate through the keys and try to try to see if the from key is present for the user in Twopath hashmap.

As we can see in the above example we have.

So we increase the triangle count by 1.

Similarly for 2's iteration

We will get

Threepath = 2[1]
Twopath = 1[2]

We increase the count by 1

For Similarly for 3's iteration

We will get

Threepath = 3[2]
Twopath = 2[3]

We increase the count by 1

So we have a total count of triangle as 3.

Since we have counted the same triangle thrice we will divide the result by 3 to obtain the number of triangle.

**Pseudo-Code:**

```
class PathTwoMapper
        method Map(doc edge)
                for all users ∈ edge e do
                        user = users.split(“,”)
                        from =  user[0]
                        to = user[1]
                        If from < MAX_FILTER &&  to < MAX_FILTER
                                Emit(from, to)


class PathTwoReducer
         method Reduce(from , counts [u1, u2, . . ., un])
                Hashmap  hm <-  (from, List(to))
                For all key from ∈ hashmap hm :
                        For all neighbour ∈ hm(key):
                                For all neighbors_neighbour ∈ hm(neighbour):
                                Emit(key, neighbour, neighbors_neighbour)


//Read the original File as well as the output of the PathTwoReducer.
class TriangleMapper
        method Map(doc originalPath_TwoPath_combined)
                for all users ∈  originalPath_TwoPath_combined e do
                        user = users.split(“,”)
                        from =  user[0]
                        to = user[user.length -1]
                        If from < MAX_FILTER &&  to < MAX_FILTER
                                Emit(from, value)
                                Emit(to, value)



class TriangleReducer
         method Reduce(from , counts [u1, u2, . . ., un])
                Hashmap  threePath<-  (from, List(to)) if user is from
                Hashmap  twoPath<-  (from, List(to)) if user is to
                For all key from ∈ hashmap  threePath:
                     For all neighbors_neighbour  ∈ hm(key):
                        If hashMap twoPath(neighbors_neighbour) contains key:
                                Emit(“Triangle”, “1”)
```

```
Class CounterMapper
        Method map(doc triangles)
                word1 = triangles.split(“,”)[0]
                word2 = triangles.split(“,”)[1]
                emit(word1, word2)

Class CounterReducer
        Method reduce(Triangle , counts [1, 1, . . ., 1])
                sum ← 0
                for all count c ∈ counts [1, 1, . . ., 1] do
                        sum ← sum + c
                Emit(“the number of triangles are”,  (count sum)/3)
```

**REPLICATED JOIN:**

**Explanation:**

First we will go through the edge file and create a adjacency list of every user.
Then in the mapper we will start reading the edge and try to find the “to users” neighbour’s neighbour and see if it is equal to the from user. If it is equal to the from user, then we will increase the count of the triangle by 1. Since we count the same triangle thrice , we will divide the count by 3 to obtain the result.

**Pseudo-Code:**

```
HashMap adjList //Distributed cache
method ReplicatedJoinMap(doc edge)
                for all users ∈ edge e do
                        user = users.split(“,”)
                        from =  user[0]
                        to = user[1]
                        For neighbors_neighbor ∈  adjList(to):
                                If adj( neighbors_neighbor ) contains from:
```

```
                    Emit("Triangle", "1")

Class ReplicatedJoinReducer
    Method reduce(Triangle , counts [1, 1, . . ., 1])
        sum ← 0
        for all count c ∈ counts [1, 1, . . ., 1] do
            sum ← sum + c
        Emit("the number of triangles are",  (count sum)/3)
```

| **Configuration** | **Small Cluster Result** | **Large Cluster Result** |
|---|---|---|
| **RS-join, MAX = 1000** | Running time: 98 sec<br>Triangle count: 1099 | Running time: 100 sec<br>Triangle count: 1099 |
| **Rep-join, MAX = 1000** | Running time: 190 sec<br>Triangle count: 1099 | Running time: 196 sec<br>Triangle count: 1099 |
| **RS-join, MAX = 10000** | Running time: 170 sec<br>Triangle count: 520296 | Running time: 164 sec<br>Triangle count: 520296 |
| **Rep-join, MAX = 10000** | Running time: 196 sec<br>Triangle count: 520296 | Running time: 198 sec<br>Triangle count: 520296 |
| **RS-join, MAX = 20000** | Running time: 330 sec<br>Triangle count: 2411611 | Running time: 324 sec<br>Triangle count: 2411611 |
| **Rep-join, MAX = 20000** | Running time: 430 sec<br>Triangle count: 2411611 | Running time: 320 sec<br>Triangle count: 2411611 |