CS 6240 : Large-Scale Parallel Processing Homework 2

Dharmish Shah

PROBLEM ANALYSIS

Pseudo Code to determine Cardinality -

Phase 1 – This phase finds all incoming and outgoing edges of every user and emits a key value pair of total outgoing and total incoming edges for each user.

```
class Mapper
    method Map(edges E)
        Initialize MAX_USER
        for all edge(UserId u,UserId v) ∈ E do
            Split edge(userId u, userId v) string by “,”
            if userId < MAX_USER and userId v < MAX_USER:
                Emit(UserId u  + “-Outgoing”, 1)
                Emit(UserId v + “-Incoming”, 1)

class Reducer
    method Reduce(UserId u , Count [c1, c2] )
            sum ← 0
            for all value t ∈ counts [c1 , c2 , . . .] do
                sum ← sum + 1
            Emit(u, 1)
```

Phase 2 – This phase finds total sum of incoming and outgoing edges of every user in twitter graph, giving us total number of Path-2 present in the graph.

```
class Mapper
    method setup()
        Initialize UserIdInfo
        for all edge(userId u, Count numberOfEdges) ∈ E do
            Split edge(userId u, Count numberOfEdges) string by “,”
            Add in map(UserId u Incoming , numberOfEdges)
            Add in map (UserId u Outgoing, numberOfEdges)

    method Map(edges E)
        for all edge(userId u, Count numberOfEdges) ∈ E do
            Split edge(userId u, Count numberOfEdges) string by “,”
            Get outgoing M and incoming N edges from UserIdInfo
            paths = M * N
            Emit(“Paths”, paths)
```

```
class Reducer
    method Reduce(Text Paths ,Count paths[p1, p2,...] )
            sum ← 0
            for all value p ∈ counts [p1 , p2 , . . .] do
                sum ← sum + p
            Emit("Total Paths", sum)
```

Outputs for Cardinality:

Number of Nodes – 11,316,811 users

MAX VALUE	PATH-2 cardinality
1000	400016
10000	73597234 ~ 73 Million
100K	799376231 ~ 799 Million
1 Million	95815384431 ~ 95 Billion
11 Million	~ 900 Billion

Hence, based on our assumption, the cardinality for Path-2 joins on whole dataset will be approximately ~900 Billion records.
So, total number of triangles will never exceed Path-2 joins cardinality and now we know, that Path-2 cardinality for whole data set will be our upper bound for total triangles both Rep Join and Reduce Join.

	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)	~900 Billion	~900 Billion	~900 Billion	~900 Billion	~900 Billion	~900 Billion
Step 2 (join of Path2 with Edges)	~900 Billion	~900 Billion	~900 Billion	N/A	N/A	N/A

JOIN IMPLEMENTATION

1. MAX FILTER

Twitter-Follower program takes *edges.csv* as input and returns all edges which are less than MAX_USER value. The input file have comma separated values (e,v) where UserId u is following UserId v. The output file results as a plain text, each line containing UserId and its respective follower count separated by space.

This program reads the input line by line. The map function parses a line to extract the edges. Each edge (e,v) states that UserId u is following UserId v. For each line, the map function splits the line by comma(“,”) and then checks whether both userId u and v are less than MAX_USER value. If yes, it outputs a key-value pair of UserId v with userId u denoting that UserId u is following userId v. So, the reducer now emits only the filtered edges which are less than MAX_USER value. We do this data reduction for finding approximation of smaller data set.

Following is the pseudo-code for Max filter program.

```
class Mapper
    method Map(edges E)
        Initialize MAX_USER
        for all edge(UserId u,UserId v) ∈ E do
            Split edge(userId u, userId v) string by “,”
            if userId < MAX_USER and userId v < MAX_USER:
                Emit(UserId u, userId v)

class Reducer
    method Reduce(UserId u , UserId v)
        Emit(UserId u, UserId v)
```

2. REDUCE JOIN

Now, on reduce side join, TwitterTriangleReduceJoin program takes filtered.*csv* as input and which are less than MAX_USER value and returns total number of distinct triangles in Twitter graph. The input file have comma separated values (e,v) where UserId u is following UserId v. The output file results as a plain text with one line giving final count of total number of distinct triangles.

Phase 1:

The purpose of this phase to find all the Path-2 joins from the edges. For example, if 1 follows 2 and 2 follows 3, then Path 2 output will be $1 \rightarrow 2 \rightarrow 3$, where we are joining on intermediate node, 2. This program reads the input line by line. The map function parses a line to extract the edges. For each line, the map function splits the line by comma(“,”) and then it emits two key-value pairs of follower and followee edge as key and edge itself as value. So, the reducer will now have list of all edges associated for each user whether its a follower or its following someone. First, we iterate through all values of every key received and will create a map with key as follower and value as list of followees. Then, reducer will iterate through list of values of every key in the hashmap finding whether that value is itself a key in the hashmap. If yes, then it will emit Path 2 join in a text file.

Following is the pseudo-code for Reduce Side Join program (Phase 1).

```
class Mapper
    method Map(edges E)
        for all edge(UserId u,UserId v) ∈ E do
            Split edge(userId u, userId v) string by “,”
            Emit(UserId u, edge(userId u, userId v))
            Emit(UserId v, edge(userId u, userId v))

class Reducer
    method Reduce(UserId key , Values values)
        Initialize Map<userFrom, List<Integer> userTo> 2PathMap
        for all values val ∈ values do:
            Split val(userId u, userId v) string by “,”
            Add userId v to list of userTo of key userFrom e in the 2PathMap
        for all key ∈ 2PathMap do:
            for all values val ∈ key:
                if  2PathMap.containsKey(val) do:
                    Add all values of key val in  2PathMap    // Add 2 path
Join
                    Emit(2-Path Edge)
```

Phase 2:

Now, that we have found all the 2-Path intermediate combinations of triangles, the purpose of this phase to find whether those 2-Path nodes is a part of complete traingle or not. For example, if 1 follows 2 and 2 follows 3, then is there a path from 3 following 1 to form triangle. This program reads the input line by line from two input files(one from original input file and one from intermediate 2-Path file). For each line, the map function splits the line by comma(“,”) and then it emits two key-value pairs of follower and followee edge as key and edge itself as value. But, it also makes sure that it emits correct key-value pairs of 2-Path intermediate edges emitting start and end nodes, ignoring mid node.So, the reducer will now have list of all edges associated for each user whether its a follower or its following someone. First, we iterate through all values of every key received and will create a two maps separately for 2-Path joins and original edges with key as follower and value as list of followees. Then, reducer will iterate through list of values of every key in the 2Path hashmap finding whether that value is itself a key in the original edges hashmap and vice versa. If yes, then it forms a triangle and it will emit 3 nodes forming the triangle in a text file.

Following is the pseudo-code for Reduce Side Join program (Phase 2).

```
class Mapper
    method Map(edges E)
        for all edge e ∈ E do
            val = Split edge e by “,”
```

```
                        // Iterate through both intermediate 2-path and original file
                        if val.length == 2:
                            Emit(UserId u, edge(userId u, userId v))
                            Emit(UserId v, edge(userId u, userId v))
                        if val.length == 3:
                            Emit(UserId u, edge(userId u, userId v, userId w))
                            Emit(UserId u, edge(userId u, userId v, userId w))

        class Reducer
            method Reduce(UserId key , Value values)
                Initialize Map<userFrom, List<Integer> userTo> 2PathMap
                Initialize Map<userFrom, List<Integer> userTo> OriginalEdgeMap
                for all values edge ∈ values do:
                    val = Split edge e by ","
                    // Iterate through both intermediate 2-path and original file
                    if val.length == 2:
                        Add userId v to list of userTo of key userFrom e in the
    OriginalEdgeMap
                    if val.length == 3:
                        Add userId u to list of userTo of key userFrom e in the 2PathMap

                for all key ∈ 2PathMap do:
                    for all values val ∈ key:
                        if  OriginalEdgeMap.containsKey(val) and
                                OriginalEdgeMap.get(val).containsValue(key) do:
                            Emit("Triangle", Triangle Nodes)     // Add triangle
```

Phase 3:

Now, that we have found all the combinations of triangles, the purpose of this phase to sum all the count of triangles and dividing it by 3 to return distinct triangles. For example, if 1 follows 2 and 2 follows 3, then possible triangles are 1→2→3 →1, 2→3→1 →2 and 3→1→2 →3. This program reads the input line by line. The map function parses a line to extract the each triangle we got from Phase 2. For each line, the map function splits the line by comma(",") and then it emits a key-value pair of "Triangle" as key and triangle users itself as value. So, the reducer will now have list of all triangles, and we count just count each of the triangle and then we will divide it by 3 to get distinct triangles. The, it will emit final count of distinct triangles in a final output text file.

Following is the pseudo-code for Reduce Side Join program (Phase 3)

```
    class Mapper
        method Map(Triangles T)
            for all triangle (UserId u,UserId v, UserId w) t ∈ T do
                Split edge("Triangle", t) string by ","
                Emit("Triangle", t)

    class Reducer
        method Reduce("Triangle", Edge(userId u, userId v,userId w))
```

```
sum ← 0
for all triangles t ∈ counts [c 1 , c 2 , . . .] do
    sum ← sum + 1
sum = sum/3
Emit(“No of Triangle”, sum)
```

3. REPLICATED JOIN

Phase 1:

The purpose of this phase to find all triangles from the edges. For example, if 1 follows 2 and 2 follows 3, then is there a path from 3 following 1 to form triangle. This program reads the input line by line. Before the map function, we use setup method to store the input file as a distributed cache file. The setup method reads line by line to extract edges and it emits two key-value pairs of follower and followee edge as key and edge itself as value, storing them in a hashmap “userIdInfo”.

Now, mapper method reads the input line by line from the input files. For each line, the map function splits the line by comma(“,”) and have list of all edges associated for each user. First, we iterate through all values of edge having start “s” and finding whether its a key in userIdInfo map or not, giving us 2-Path join. Then, it will iterate through list of values of every key in the 2-Path join in the userIdInfo map finding whether one of those values is itself key in the hashmap and vice versa. If yes, then it forms a triangle and it will emit 3 nodes forming the triangle in a text file.

Following is the pseudo-code for Max filter program.

```
class Mapper
    method Setup():
        Initiliaze UserIdInfo Map
        for all values edge ∈ values do:
            val = Split edge e by “,”
            // Iterate through both intermediate 2-path and original file
            if val.length == 2:
                Add userId v to list of userTo of key userFrom e in the
UserIdInfo

    method Map(edges E)
        val = Split each edge by (“,”)
        for all values edge ∈ values do:
            val = Split edge (userId u, userId v) by “,”
            u = val[0], v = val[1] // u = follower v= followee
            for all v ∈ key in UserIdInfo do:
                for all values 2pathVal ∈ v:
                    if UserIdInfo.contains(2pathVal) and and
                        UserIdInfo.get(2pathVal).containsValue(u) do:
                        Emit(“Triangle”, Triangle Nodes)     // Add triangle
```

Phase 2:

Now, that we have found all the combinations of triangles, the purpose of this phase to sum all the count of triangles and dividing it by 3 to return distinct triangles. For example, if 1 follows 2 and 2 follows 3, then possible triangles are 1→2→3 →1, 2→3→1 →2 and 3→1→2 →3. This program reads the input line by line. The map function parses a line to extract the each triangle we got from Phase 2. For each line, the map function splits the line by comma(“,”) and then it emits a key-value pair of “Triangle” as key and triangle users itself as value. So, the reducer will now have list of all triangles, and we count just count each of the triangle and then we will divide it by 3 to get distinct triangles. The, it will emit final count of distinct triangles in a final output text file.

Following is the pseudo-code for Replicated Side Join program (Phase 2)

```
class Mapper
    method Map(Triangles T)
        for all triangle (UserId u,UserId v, UserId w) t ∈ T do
            Split edge(“Triangle”, t) string by “,”
            Emit(“Triangle”, t)

class Reducer
    method Reduce(“Triangle”, Edge(userId u, userId v,userId w))
        sum ← 0
        for all triangles t ∈ counts [c 1 , c 2 , . . .] do
            sum ← sum + 1
        sum = sum/3
        Emit(“No of Triangle”, sum)
```

RUN TIME MEASUREMENTS

The Reduce and replicated join was ran on AWS 11 machines, 9 machines and 8 machines for both m4.xlarge and m5.xlarge. But, AWS never started the program in the cluster and was left in infinite state. The experiment for each run was tried for an average 25 minutes and each time the cluster was stuck in “Starting” state. So, it ran

Ran the program on AWS with configuration :

1) Small Cluster - 6 cheap machine instance 1 master and 5 workers on m5.xlarge instance
2) Large Cluster - 7 cheap machine instance 1 master and 6 workers on m5.xlarge instance

Configuration	Small Cluster Result	Large Cluster Result
RS-join, MAX = 10000	Running time: 268 secs , Triangle count: 520296	Running time: 274 secs, Triangle count: 520296
Rep-join, MAX = 10000	Running time: 80 secs, Triangle count: 520296	Running time: 78 secs, Triangle count: 520296
RS-join, MAX = 25000	Running time: 516 secs, Triangle count: 2628806	Running time: 476 secs, Triangle count: 2628806
Rep-join, MAX = 25000	Running time: 110 secs, Triangle count: 2628806	Running time: 106 secs, Triangle count: 2628806

REFERENCES

- http://lintool.github.io/MapReduceAlgorithms/
- https://storage.googleapis.com/pub-tools-public-publication-data/pdf/16cb30b4b92fd4989b8619a61752a2387c6dd474.pdf
- https://www.edureka.co/blog/mapreduce-example-reduce-side-join/