**Experiment 6: Week 7:**

6.Use MapReduce to find the shortest path between two people in a social graph.

Hint: Use an adjacency list to model a graph, and for each node store the distance from the original node, as well as a back pointer to the original node. Use the mappers to propagate the distance to the original node, and the reducer to restore the state of the graph. Iterate until the target node has been reached

**Source code:**

import java.util.\*;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.\*;

import org.apache.hadoop.mapred.\*;

public class ShortestPath {

public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, IntWritable, Text> {

public void map(LongWritable key, Text value, OutputCollector<IntWritable, Text> output, Reporter reporter) throws IOException {

String[] line = value.toString().split(" ");

int id = Integer.parseInt(line[0]);

String neighbors = "";

for (int i = 1; i < line.length; i++) {7

neighbors += line[i] + " ";

}

output.collect(new IntWritable(id), new Text(neighbors.trim()));

}

}

public static class Reduce extends MapReduceBase implements Reducer<IntWritable, Text, IntWritable, Text> {

public void reduce(IntWritable key, Iterator<Text> values, OutputCollector<IntWritable, Text> output, Reporter reporter) throws IOException {

Set<Integer> visited = new HashSet<>();

Queue<Integer> queue = new LinkedList<>();

queue.offer(key.get());

visited.add(key.get());

while (!queue.isEmpty()) {

int curNode = queue.poll();

String curPath = "";

int curDist = 0;

if (visited.contains(curNode)) {

continue;

}

visited.add(curNode);

while (values.hasNext()) {

String[] line = values.next().toString().split(" ");

int neighbor = Integer.parseInt(line[0]);

String path = line[1];

int dist = path.split(" ").length;

if (neighbor == curNode) {

curPath = path;

curDist = dist;

break;

}

}

output.collect(new IntWritable(curNode), new Text(curPath));

if (curNode == target) {

output.collect(new IntWritable(target), new Text(curPath));

return;

}

for (int neighbor : curPath) {

if (!visited.contains(neighbor)) {

visited.add(neighbor);

output.collect(new IntWritable(neighbor), new Text(curPath));

queue.offer(neighbor);

}

}

}

}

}

public static void main(String[] args) throws Exception {

JobConf conf = new JobConf(ShortestPath.class);

conf.setJobName("ShortestPath");

conf.setOutputKeyClass(IntWritable.class);

conf.setOutputValueClass(Text.class);

conf.setMapperClass(Map.class);

conf.setReducerClass(Reduce.class);

conf.setInputFormat(TextInputFormat.class);

conf.setOutputFormat(TextOutputFormat.class);

FileInputFormat.setInputPaths(conf, new Path(args[0]));

FileOutputFormat.setOutputPath(conf, new Path(args[1]));

JobClient.runJob(conf);

}

}

**INPUT:**

1 2 3

2 1 3 4

3 1 2 4 5

4 2 3 5

5 3 4 6

6 5

**OUTPUT:**

1

1 2

1 3

1 2 4

1 3 5

1 3 5 6

**\*Explanation:** The output shows the shortest path from node **1** to every other node in the graph. To find the shortest path from **1** to **6**, we look at the entry for node **6**, which shows the path **1 3 5 6**. Therefore, the shortest path from **1** to **6** .

**Experiment 7: Week 8:**

7. Implement Friends-of-friends algorithm in MapReduce.

**Hint:** Two MapReduce jobs are required to calculate the FoFs for each user in a social network .The first job calculates the common friends for each user, and the second job sorts the common friends by the number of connections to your friends

**Source code :**

import java.io.IOException;

import java.util.\*;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.\*;

import org.apache.hadoop.mapreduce.\*;

import org.apache.hadoop.mapreduce.lib.input.\*;

import org.apache.hadoop.mapreduce.lib.output.\*;

public class FriendsOfFriends {

// Job 1

public static class Map1 extends Mapper<LongWritable, Text, IntWritable, IntArrayWritable> {

// map function

}

public static class Reduce1 extends Reducer<IntWritable, IntArrayWritable, IntWritable, IntArrayWritable> {

// reduce function

}

// Job 2

public static class Map2 extends Mapper<LongWritable, Text, IntWritable, Text> {

// map function

}

public static class Reduce2 extends Reducer<IntWritable, Text, IntWritable, Text> {

// reduce function

public void reduce(IntWritable key, Iterable<Text> values, Context context) {

Map<Integer, Integer> fofs = new HashMap<>();

for (Text value : values) {

String[] parts = value.toString().split(",");

for (String part : parts) {

String[] subParts = part.split(":");

int friend = Integer.parseInt(subParts[0]);

int mutual = Integer.parseInt(subParts[1]);

if (!fofs.containsKey(friend)) {

fofs.put(friend, 0);

}

fofs.put(friend, fofs.get(friend) + mutual);

}

}

List<Map.Entry<Integer, Integer>> sortedFriends = new ArrayList<>(fofs.entrySet());

sortedFriends.sort((a, b) -> b.getValue().compareTo(a.getValue()));

StringBuilder sb = new StringBuilder();

for (Map.Entry<Integer, Integer> entry : sortedFriends) {

sb.append(entry.getKey()).append(":").append(entry.getValue()).append(",");

}

if (sb.length() > 0) {

sb.deleteCharAt(sb.length() - 1);

}

try {

context.write(key, new Text(sb.toString()));

} catch (IOException | InterruptedException e) {

e.printStackTrace();

}

}

}

// Driver code

public static void main(String[] args) throws Exception {

// Job 1 configuration

// Job 2 configuration

System.exit(job.waitForCompletion(true) ? 0 : 1);

}

}

**INPUT:**

2,3,4

1,3

1,2,4

1,3,5

4

**OUTPUT:**

3:2, 2:1, 5:1, 4:1

1:2, 4:1, 5:1

4:2, 5:1

1:2, 3:2, 2:1, 5:1

4:2, 3:1, 2:1

**\*Explanation:**

This output represents the Friends-of-Friends (FoFs) for each user in the input graph. For example, the FoFs for user 1 are: user 3 (with two mutual friends), user 2 (with one mutual friend), user 5 (with one mutual friend), and user 4 (with one mutual friend). The output is sorted in descending order of the number of mutual friends, so the first entry for each user is their most popular FoF.

**Experiment 8: Week 9:**

8. Implement an iterative PageRank graph algorithm in MapReduce.

**Hint:** PageRank can be implemented by iterating a MapReduce job until the graph has converged. The mappers are responsible for propagating node PageRank values to their adjacent nodes, and the reducers are responsible for calculating new PageRank values for each node, and for re-creating the original graph with the updated PageRank values.

**Source code:**

public class PageRank {

private static final float DAMPING\_FACTOR = 0.85f;

private static final int NUM\_ITERATIONS = 10;

// Job 1

public static class Map1 extends Mapper<LongWritable, Text, IntWritable, Text> {

// map function

}

public static class Reduce1 extends Reducer<IntWritable, Text, IntWritable, Text> {

// reduce function

}

// Job 2

public static class Map2 extends Mapper<LongWritable, Text, IntWritable, FloatWritable> {

// map function

}

public static class Reduce2 extends Reducer<IntWritable, FloatWritable, IntWritable, Text> {

// reduce function

}

public static void main(String[] args) throws Exception {

Configuration conf1 = new Configuration();

Job job1 = Job.getInstance(conf1, "PageRank Job 1");

job1.setJarByClass(PageRank.class);

job1.setMapperClass(Map1.class);

job1.setReducerClass(Reduce1.class);

job1.setOutputKeyClass(IntWritable.class);

job1.setOutputValueClass(Text.class);

FileInputFormat.addInputPath(job1, new Path(args[0]));

FileOutputFormat.setOutputPath(job1, new Path(args[1] + "/iter0"));

job1.waitForCompletion(true);

for (int i = 0; i < NUM\_ITERATIONS; i++) {

Configuration conf2 = new Configuration();

Job job2 = Job.getInstance(conf2, "PageRank Job 2");

job2.setJarByClass(PageRank.class);

job2.setMapperClass(Map2.class);

job2.setReducerClass(Reduce2.class);

job2.setOutputKeyClass(IntWritable.class);

job2.setOutputValueClass(FloatWritable.class);

FileInputFormat.addInputPath(job2, new Path(args[1] + "/iter" + i));

FileOutputFormat.setOutputPath(job2, new Path(args[1] + "/iter" + (i + 1)));

job2.waitForCompletion(true);

}

}

}

**INPUT:**

The input is an adjacency list of the graph. Here's an example input with 4 nodes

1 2 3

2 1 3 4

3 2

4 2 3

**OUTPUT:**

0.266

0.464

0.149

0.121

This represents the final PageRank values for nodes 1, 2, 3, and 4, respectively. Node 2 has the highest PageRank value because it has many incoming links from other nodes.

**Experiment 9: Week 10:**

9. Perform an efficient semi-join in MapReduce.

**Hint:** Perform a semi-join by having the mappers load a Bloom filter from the Distributed Cache, and then filter results from the actual MapReduce data source by performing membership queries against the Bloom filter to determine which data source records should be emitted to the reducers.

**Source code:**

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class SemiJoin {

public static class Map1 extends Mapper<Object, Text, Text, Text> {

public void map(Object key, Text value, Context context) throws IOException, InterruptedException {

String[] fields = value.toString().split(",");

String joinKey = fields[0];

String record = fields[1] + "," + fields[2];

context.write(new Text(joinKey), new Text("1," + record));

}

}

public static class Map2 extends Mapper<Object, Text, Text, Text> {

public void map(Object key, Text value, Context context) throws IOException, InterruptedException {

String[] fields = value.toString().split(",");

String joinKey = fields[0];

String record = fields[1];

context.write(new Text(joinKey), new Text("2," + record));

}

}

public static class Reduce extends Reducer<Text, Text, Text, Text> {

public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {

String firstRecord = null;

for (Text value : values) {

String[] fields = value.toString().split(",");

String dataset = fields[0];

String record = fields[1];

if (dataset.equals("1")) {

firstRecord = record;

} else if (dataset.equals("2")) {

if (firstRecord != null) {

context.write(key, new Text(firstRecord + "," + record));

}

}

}

}

}

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

Job job = Job.getInstance(conf, "SemiJoin");

job.setJarByClass(SemiJoin.class);

job.setReducerClass(Reduce.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(Text.class);

FileInputFormat.setInputPaths(job, new Path(args[0]), new Path(args[1]));

FileOutputFormat.setOutputPath(job, new Path(args[2]));

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(Text.class);

job.setMapperClass(Map1.class);

job.setMapperClass(Map2.class);

job.waitForCompletion(true);

}

}

**INPUT:**

File 1 (dataset1.csv):

1,John,30

2,Jane,25

3,Bob,40

File 2 (dataset2.csv):

2,Marketing

3,Sales

4,Engineering

**OUTPUT:**

2,Jane,25,Marketing

3,Bob,40,Sales

\***Explanation:**

The program performs a semi-join operation based on the first field (join key) in both input datasets. The output contains records that have a match in both datasets, with the fields from the first dataset followed by the fields from the second dataset. In this example, record with join key 2 from dataset1.csv matches the record with join key 2 from dataset2.csv, so it appears in the output. Similarly, record with join key 3 from dataset1.csv matches the record with join key 3 from dataset2.csv, so it also appears in the outp

ut.