K. J. Somaiya College of Engineering, Mumbai-77

(Autonomous College Affiliated to University of Mumbai)

Batch: BDA_2 Roll No.: 1211061

Experiment / assignment / tutorial No.__3__

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of the Staff In-charge with date

TITLE : Implementation of simple algorithms in MapReduce – Matrix Multiplication etc.

AIM : Implementation of simple algorithms in MapReduce – Matrix Multiplication etc.

Expected Outcome of Experiment :

CO :

CO2:
Understand the fundamental enabling techniques like Hadoop, MapReduce and NO SQL in achieving Big data analytics

Books/ Journals/ Websites referred :

1. Anand Rajaraman and Jeff Ullman "Mining of Massive Datasets", Cambridge University Press,
2. Alex Holmes "Hadoop in Practice", Manning Press, Dreamtech Press.
3. Big data analytics by Radha Shankarmani, M. Vijayalakshmi. Wiley publication

Pre Lab/ Prior Concepts :
Map Reduce:

- Hadoop MapReduce is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.
- A MapReduce *job* usually splits the input data-set into independent chunks which are processed by the *map tasks* in a completely parallel manner. The framework

sorts the outputs of the maps, which are then input to the *reduce tasks*. Typically both the input and the output of the job are stored in a file-system. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks.

Matrix multiplication as two map reduce job

The input file has one line of the following format for each non-zero element of a matrix :
<M><i><j><m_ij>

Map and Reduce function of Job1:

Step1Mapper

```
public void map(LongWritable ikey, Text ivalue, Context
                context) throws IOException,
                InterruptedException {
        String line=ivalue.toString();
        String[]
        tokens=StringUtils.split(line,',';
        if(tokens[0].equals("M")){
                key.set(Integer.parseInt(tokens[2]));
     value.set(tokens[0],Integer.parseInt(tokens[1]),Integer.parseInt(tokens[3]));
        }
        else
        {
                key.set(Integer.parseInt(tokens[1]));
     value.set(tokens[0],Integer.parseInt(tokens[2]),Integer.parseInt(tokens[3]));
        }
        context.write(key,value);
}
```

Step2Mapper

```
public void map(LongWritable ikey, Text ivalue, Context
                context) throws IOException,
                InterruptedException {
        String line=ivalue.toString();
        String[] tokens=StringUtils.split(line,'\t');
        String[]  subtokens=StringUtils.split(tokens[0],',');

        key.set(Integer.parseInt(subtokens[0]),
        Integer.parseInt(subtokens[1]));
        value.set(Integer.parseInt(tokens[1]));
        context.write(key, value);
}
```

Map and Reduce function of Job2:

Step1Reducer

```
            public void reduce(IntWritable _key, Iterable<Relation>
            values, Context context) throws IOException,
            InterruptedException {

      ArrayList<Relation> mRels=new ArrayList<>();
      ArrayList<Relation> nRels=new ArrayList<>();


      //separating mrelation and nrelation

      for(Relation value: values){

            //for every relation create a new object at the
            reducer side Relation temp=new Relation();

            //transfer the data from old object to new object

       temp.set(value.getFromMatrix(), value.getIorK(), value.getmijOrnik());

            if(value.getFromMatrix().equals("M")){

                  mRels.add(temp);
            }
            else
            {
                  nRels.add(temp);

            }
      }
      for (Iterator iterator = mRels.iterator(); iterator.hasNext();)
      {
            Relation mrelation = (Relation) iterator.next();
            for (Iterator iterator2 = nRels.iterator(); iterator2.hasNext();)
                  {
                  Relation nrelation = (Relation) iterator2.next();
                  key.set(mrelation.getIorK(), nrelation.getIorK());
                  value.set(mrelation.getmijOrnik()*
                  nrelation.getmijOrnik()); context.write(key, value);

                  }
      }
```

Step2Reducer

```
public class Step2Reducer extends
        Reducer<IntPair, IntWritable, IntPair,
IntWritable> { private IntWritable value=new
IntWritable();

public void reduce(IntPair _key, Iterable<IntWritable> values,
                Context context) throws IOException,
                InterruptedException {
        int sum=0;
        for(IntWritable
                value:values)
                {
                sum+=value.
                get();

        }
        value.set(sum);
        context.write(_ke
        y, value);
}
```

Conclusion:

<u>Post lab questions:</u>

1. **Write the Map and Reduce function for computing matrix multiplication as single map reduce job.**

```
map(key, value):
  // value is ("A", i, j, a_ij) or ("B", j, k, b_jk)
  if value[0] == "A":
    i = value[1]
    j = value[2]
    a_ij = value[3]
    for k = 1 to p:
      emit((i, k), (A, j, a_ij))
  else:
    j = value[1]
```

```
        k = value[2]
        b_jk = value[3]
        for i = 1 to m:
            emit((i, k), (B, j, b_jk))

reduce(key, values):
    // key is (i, k)
    // values is a list of ("A", j, a_ij) and ("B", j, b_jk)
    hash_A = {j: a_ij for (x, j, a_ij) in values if x == A}
    hash_B = {j: b_jk for (x, j, b_jk) in values if x == B}
    result = 0
    for j = 1 to n:
        result += hash_A[j] * hash_B[j]
    emit(key, result)
```

Function

Mapper:-

```
public void map(LongWritable key, Text value, Context context) throws
IOException,
InterruptedException {
            Configuration conf = context.getConfiguration();
            int m = Integer.parseInt(conf.get("m"));
            int p = Integer.parseInt(conf.get("p"));
            String line = value.toString();
            String[] indicesAndValue = line.split(",");
            Text outputKey = new Text();
            Text outputValue = new Text();
            if (indicesAndValue[0].equals("A")) {
                for (int k = 0; k < p; k++) {
                    outputKey.set(indicesAndValue[1] + "," + k);
                    outputValue.set("A," + indicesAndValue[2] + "," +
indicesAndValue[3]);
```

```
                    context.write(outputKey, outputValue);
                }
            } else {
                for (int i = 0; i < m; i++) {
                    outputKey.set(i + "," + indicesAndValue[2]);
                    outputValue.set("B," + indicesAndValue[1] + "," +
indicesAndValue[3]);
                    context.write(outputKey, outputValue);
                }
            }
        }
```

Reducer:-

```
public void reduce(Text key, Iterable<Text> values, Context context) throws
IOException, InterruptedException {
            String[] value;
           HashMap<Integer, Float> hashA = new HashMap<Integer, Float>();
            HashMap<Integer, Float> hashB = new HashMap<Integer, Float>();
            for (Text val : values) {
                value = val.toString().split(",");
                if (value[0].equals("A")) {
hashA.put(Integer.parseInt(value[1]), Float.parseFloat(value[2]));
               } else {
                    hashB.put(Integer.parseInt(value[1]), Float.parseFloat(value[2]));
            int n = Integer.parseInt(context.getConfiguration().get("n"));
           float result = 0.0f;
            float a_ij;
            float b_jk;
            for (int j = 0; j < n; j++) {
                a_ij = hashA.containsKey(j) ? hashA.get(j) : 0.0f;
                b_jk = hashB.containsKey(j) ? hashB.get(j) : 0.0f;
                result += a_ij * b_jk;
            if (result != 0.0f) {
                context.write(null, new Text(key.toString() + "," +
            Float.toString(result)));
```