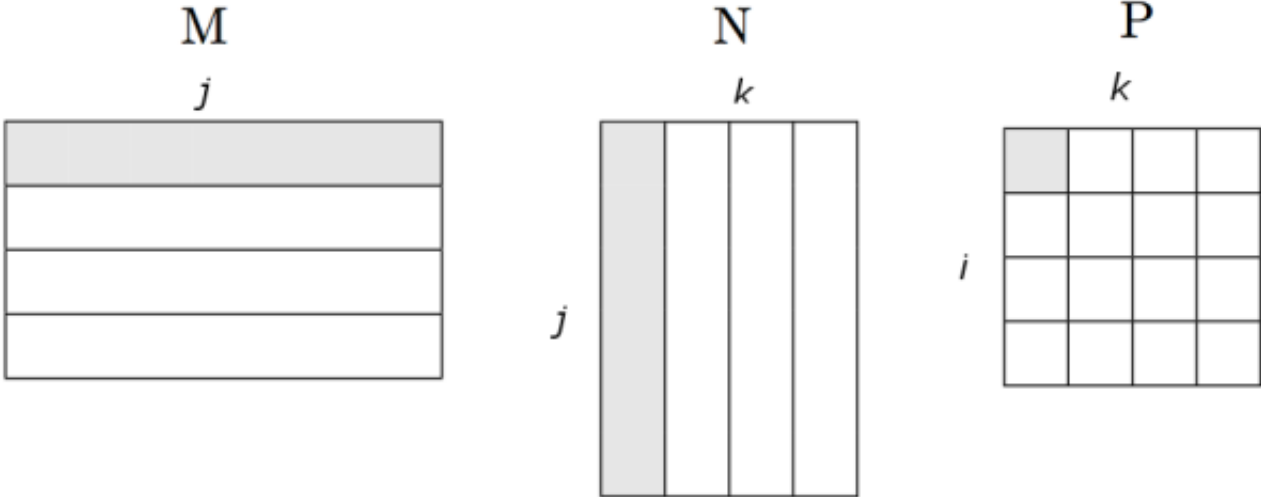


Matrix Multiplication with MapReduce

[Maruf Aytekin](#)

Matrix-vector and matrix-matrix calculations fit nicely into the MapReduce style of computing. In this post I will only examine matrix-matrix calculation as described in [1, ch.2].

Suppose we have a $p \times q$ matrix M , whose element in row i and column j will be denoted m_{ij} and a $q \times r$ matrix N whose element in row j and column k is denoted by n_{jk} then the product $P = MN$ will be $p \times r$ matrix P whose element in row i and column k will be denoted by p_{ik} , where

$$P(i, k) = \sum_j m_{ij} * n_{jk}$$


The diagram shows three matrices: M, N, and P. Matrix M is a 4x4 grid with the first row shaded. Matrix N is a 4x4 grid with the first column shaded. Matrix P is a 4x4 grid with the first row shaded. The shaded areas represent the elements involved in the calculation of P(i, k).

Matrix Data Model for MapReduce

We represent matrix M as a relation

$$M(I, J, V)$$

, with tuples

$$(i, j, m_{ij})$$

, and matrix N as a relation

$$N(J, K, W)$$

, with tuples

$$(j, k, n_{jk})$$

. Most matrices are sparse so large amount of cells have value zero.

When we represent matrices in this form, we do not need to keep entries for the cells that have values of zero to save large amount of disk space.

As input data files, we store matrix M and N on HDFS in following format:

$$M, i, j, m_{ij}$$

```
M,0,0,10.0
M,0,2,9.0
M,0,3,9.0
M,1,0,1.0
M,1,1,3.0
M,1,2,18.0
M,1,3,25.2
....
```

$$N, j, k, n_{jk}$$

```
N,0,0,1.0
N,0,2,3.0
N,0,4,2.0
N,1,0,2.0
N,3,2,-1.0
N,3,6,4.0
N,4,6,5.0
N,4,0,-1.0
....
```

MapReduce

We will write Map and Reduce functions to process input files. Map function will produce

$$key, value$$

pairs from the input data as it is described in Algorithm 1. Reduce

function uses the output of the Map function and performs the calculations and produces

key, value

pairs as described in Algorithm 2. All outputs are written to HDFS.

Algorithm 1: The Map Function

```

1 for each element  $m_{ij}$  of  $M$  do
2   produce  $(key, value)$  pairs as  $((i, k), (M, j, m_{ij}))$ , for  $k = 1, 2, 3, ..$  up
   to the number of columns of  $N$ 
3 for each element  $n_{jk}$  of  $N$  do
4   produce  $(key, value)$  pairs as  $((i, k), (N, j, n_{jk}))$ , for  $i = 1, 2, 3, ...$  up
   to the number of rows of  $M$ 
5 return Set of  $(key, value)$  pairs that each key,  $(i, k)$ , has a list with
   values  $(M, j, m_{ij})$  and  $(N, j, n_{jk})$  for all possible values of  $j$ 

```

Algorithm 2: The Reduce Function

```

1 for each key  $(i, k)$  do
2   sort values begin with  $M$  by  $j$  in  $list_M$ 
3   sort values begin with  $N$  by  $j$  in  $list_N$ 
4   multiply  $m_{ij}$  and  $n_{jk}$  for  $j_{th}$  value of each list
5   sum up  $m_{ij} * n_{jk}$ 
6 return  $(i, k), \sum_{j=1} m_{ij} * n_{jk}$ 

```

The value in row i and column k of product matrix P will be:

$$P_{(i,k)} = \sum_{j=1} m_{ij} * n_{jk}$$

.

Let me examine the algorithms on an example to explain the algorithms better. Suppose we have two matrices, M , 2×3 matrix, and N , 3×2 matrix as follows:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}$$

The product P of MN will be as follows:

$$\begin{bmatrix} 1a + 2c + 3e & 1b + 2d + 3f \\ 4a + 5c + 6e & 4b + 5d + 6f \end{bmatrix}$$

The Map Task

For matrix M , map task (Algorithm 1) will produce

key, value

pairs as follows:

$$\begin{aligned} & (i, k), (M, j, m_{ij}) \\ & m_{11} = 1 \\ & (1, 1), (M, 1, 1)k = 1 \\ & (1, 2), (M, 1, 1)k = 2 \\ & m_{12} = 2 \\ & (1, 1), (M, 2, 2)k = 1 \\ & (1, 2), (M, 2, 2)k = 2 \\ & \dots\dots\dots \\ & m_{23} = 6 \\ & (2, 1), (M, 3, 6)k = 1 \\ & (2, 2), (M, 3, 6)k = 2 \end{aligned}$$

For matrix N , map task (Algorithm 2) will produce

key, value

pairs as follows:

$$(i, k), (N, j, n_{jk})$$

$$n_{11} = a$$

$$(1, 1), (N, 1, a)_{i=1}$$

$$(2, 1), (N, 1, a)_{i=2}$$

$$n_{21} = c$$

$$(1, 1), (N, 2, c)_{i=1}$$

$$(2, 1), (N, 2, c)_{i=2}$$

$$n_{31} = e$$

$$(1, 1), (N, 3, e)_{i=1}$$

$$(2, 1), (N, 3, e)_{i=2}$$

$$\dots\dots$$

$$n_{32} = f$$

$$(1, 2), (N, 3, f)_{i=1}$$

$$(2, 2), (N, 3, f)_{i=2}$$

After combine operation the map task will return

key, value

pairs as follows:

$$((i, k), [(M, j, m_{ij}), (M, j, m_{ij}), \dots, (N, j, n_{jk}), (N, j, n_{jk}), \dots])$$

$$(1, 1), [(M, 1, 1), (M, 2, 2), (M, 3, 3), (N, 1, a), (N, 2, c), (N, 3, e)]$$

$$(1, 2), [(M, 1, 1), (M, 2, 2), (M, 3, 3), (N, 1, b), (N, 2, d), (N, 3, f)]$$

$$(2, 1), [(M, 1, 4), (M, 2, 5), (M, 3, 6), (N, 1, a), (N, 2, c), (N, 3, e)]$$

$$(2, 2), [(M, 1, 4), (M, 2, 5), (M, 3, 6), (N, 1, b), (N, 2, d), (N, 3, f)]$$

Note that the entries for the same key are grouped in the same list, which is performed by the framework. This output will be stored in HDFS and feed the reduce task as input.

The Reduce Task

Reduce task takes the \$key,value\$ pairs as the input and process one key at a time. For each key it divides the values in two separate lists for M and N . For key

$(1, 1)$

, the value is the list of

$$[(M, 1, 1), (M, 2, 2), (M, 3, 3), (N, 1, a), (N, 2, c), (N, 3, e)]$$

Reduce task sorts values begin with M in one list and values begin with N in another list as follows:

$$\begin{aligned} list_M &= [(M, 1, 1), (M, 2, 2), (M, 3, 3)] \\ list_N &= [(N, 1, a), (N, 2, c), (N, 3, e)] \end{aligned}$$

,

then sums up the multiplication of m_{ij} and n_{jk} for each j as follows:

$$P(1, 1) = 1a + 2c + 3e$$

The same computation applied to all input entries of reduce task.

$$P_{(i,k)} = \sum_{j=1} m_{ij} * n_{jk}$$

for all i and k is then calculated as follows:

$$\begin{aligned} P(1, 1) &= 1a + 2c + 3e \\ P(1, 2) &= 1b + 2d + 3f \\ P(2, 1) &= 4a + 5c + 6e \\ P(2, 2) &= 4b + 5d + 6f \end{aligned}$$

The product matrix P of MN is then generated as:

$$\begin{bmatrix} 1a + 2c + 3e & 1b + 2d + 3f \\ 4a + 5c + 6e & 4b + 5d + 6f \end{bmatrix}$$

Experiments

Data

I have setup a single node Hadoop installation with HDFS and run the matrix

calculation experiment on this installation. We used 1000 x 100 matrix M and

100 x 1000 matrix N with sparsity level of 0.3. This means each matrix has about

30K entries. The matrix les M and N stored in input directory on HDFS and

the output of the computation is stored in output directory on HDFS.

Source Code

I developed mapper and reducer classes as Map.java and Reduce.java as well as the main application class called MatrixMultiply.java. As it seen in MatrixMultiply.java code below, in main method the configuration parameters are being set as well as the input/output directories of MapReduce job [2].

```
public class MatrixMultiply {  
  
    public static void main(String[] args) throws Exception {  
        if (args.length != 2) {  
            System.err.println("Usage: MatrixMultiply <in_dir> <out_dir>");  
            System.exit(2);  
        }  
        Configuration conf = new Configuration();  
        // M is an m-by-n matrix; N is an n-by-p matrix.  
        conf.set("m", "1000");  
        conf.set("n", "100");  
        conf.set("p", "1000");  
        @SuppressWarnings("deprecation")  
        Job job = new Job(conf, "MatrixMultiply");  
        job.setJarByClass(MatrixMultiply.class);  
        job.setOutputKeyClass(Text.class);  
        job.setOutputValueClass(Text.class);  
  
        job.setMapperClass(Map.class);  
        job.setReducerClass(Reduce.class);  
  
        job.setInputFormatClass(TextInputFormat.class);  
        job.setOutputFormatClass(TextOutputFormat.class);  
  
        FileInputFormat.addInputPath(job, new Path(args[0]));  
        FileOutputFormat.setOutputPath(job, new Path(args[1]));  
  
        job.waitForCompletion(true);  
    }  
}
```

Mapper class extends org.apache.hadoop.mapreduce.Mapper class and implements the map task described in Algorithm 1 and creates the

key, value

pairs from the input files as it shown in the code below:

```

public class Map
    extends org.apache.hadoop.mapreduce.Mapper<LongWritable, Text, Text, Text> {
    @Override
    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();
        // (M, i, j, Mij);
        String[] indicesAndValue = line.split(",");
        Text outputKey = new Text();
        Text outputValue = new Text();
        if (indicesAndValue[0].equals("M")) {
            for (int k = 0; k < p; k++) {
                outputKey.set(indicesAndValue[1] + "," + k);
                // outputKey.set(i,k);
                outputValue.set(indicesAndValue[0] + "," + indicesAndValue[2]
                    + "," + indicesAndValue[3]);
                // outputValue.set(M,j,Mij);
                context.write(outputKey, outputValue);
            }
        } else {
            // (N, j, k, Njk);
            for (int i = 0; i < m; i++) {
                outputKey.set(i + "," + indicesAndValue[2]);
                outputValue.set("N," + indicesAndValue[1] + "," +
                    indicesAndValue[3]);
                context.write(outputKey, outputValue);
            }
        }
    }
}

```

Reducer class, extends org.apache.hadoop.mapreduce.Reducer class and implements the reduce task described in Algorithm 2 and creates the

key, value

pairs for the product matrix then writes its output on HDFS as it shown in the code below:


```

public class Reduce
    extends org.apache.hadoop.mapreduce.Reducer<Text, Text, Text, Text> {
    @Override
    public void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {
        String[] value;
        //key, Values
        //(i,k), [(M/N,j,V/W),...]
        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("M")) {
                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 m_ij;
        float n_jk;
        for (int j = 0; j < n; j++) {
            m_ij = hashA.containsKey(j) ? hashA.get(j) : 0.0f;
            n_jk = hashB.containsKey(j) ? hashB.get(j) : 0.0f;
            result += m_ij * n_jk;
        }
        if (result != 0.0f) {
            context.write(null,
                new Text(key.toString() + "," + Float.toString(result)));
        }
    }
}

```

Complete source code is [here](#).

References

[1] Anand Rajaraman and Jerrey David Ullman. Mining of Massive Datasets.

Cambridge University Press, New York, NY, USA, 2011.

[2] One-step matrix multiplication with hadoop, [[Online](#)], 2014

Advertisements