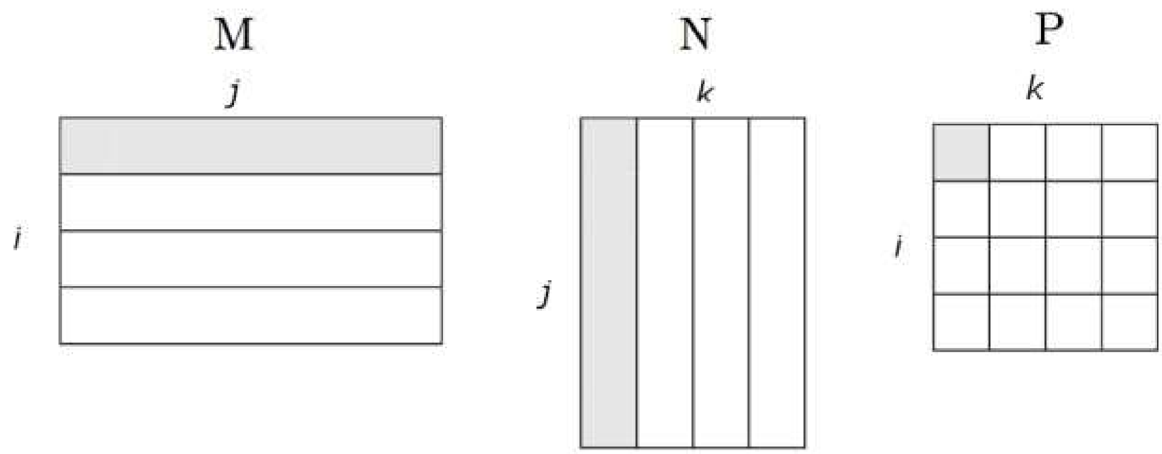
Experiment No.5
Implement simple sorting algorithm in Map reduce- Matrix Multiplications.
Date of Performance:
Date of Submission:

Aim: Implement simple sorting algorithm in Map reduce- Matrix Multiplications.

Theory:

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 pxq matrix M, whose element in row *i* and column *j* will be denoted m_{ij} and a qxr matrix N whose element in row j and column k is donated by n_{jk} then the product P = MN will be pxr matrix P whose element in row i and column k will be donated by p_{ik}, where $P(i,k) = m_{ij} * n_{jk}$.



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

for *each element* m_{ij} *of* M **do**
  produce $(key, value)$ pairs as $((i, k), (M, j, m_{ij}))$, for $k = 1, 2, 3, ..$ up to the number of columns of N
for *each element* n_{jk} *of* N **do**
  produce $(key, value)$ pairs as $((i, k), (N, j, n_{jk}))$, for $i = 1, 2, 3, ...$ up to the number of rows of M
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

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

Code:

- **Mapper.py**

```
import sys
m_r=2
m_c=3
n_r=3
n_c=2
i=0
```

```
for line in sys.stdin:
        el=map(int,line.split())
        if(i<m_r):
                for j in range(len(el)):
                        for k in range(n_c):
                                print('%d\t%d\t%d\t%d' %(i,k,j,el[j]))
    else:
                for j in range(len(el)):
                        for k in range(m_r):
                                print('%d\t%d\t%d\t%d' %(k,j,i+m_r,el[j]))
    i=i+1
```

- **Reducer.py**

```
import sys
m_r=2
m_c=3
n_r=3
n_c=2
matrix=[]
for row in range(m_r):
    r=[]
    for col in range(n_c):
        s=0
        for el in range(m_c):
            mul=1
            for num in range(2):
                line=sys.stdin.readline()
                n = map(int,line.split('\t'))[-1]
                mul*=n
            s+=mul
        r.append(s)
    matrix.append(r)
print('\n'.join([str(x) for x in matrix]))
```

- **Input:**
  - Matrix1

1	2	3
4	5	6

  - Matrix2.txt

7	8
9	10
11	12

Output:

```
[biadmin@bivm ~]$ cd matrix
[biadmin@bivm matrix]$ nano matrix1.txt
[biadmin@bivm matrix]$ nano matrix2.txt
[biadmin@bivm matrix]$ nano mapper.py
[biadmin@bivm matrix]$ cat *.txt |python mapper.py
0       0       0       1
0       1       0       1
0       0       1       2
0       1       1       2
0       0       2       3
0       1       2       3
1       0       0       4
1       1       0       4
1       0       1       5
1       1       1       5
1       0       2       6
1       1       2       6
0       0       4       7
1       0       4       7
0       1       4       8
1       1       4       8
0       0       5       5
1       0       5       5
0       1       5       6
1       1       5       6
0       0       6       10
1       0       6       10
0       1       6       11
1       1       6       11
[biadmin@bivm matrix]$ cat *.txt |python mapper.py|python reducer.py
[14, 77]
[138, 257]
[biadmin@bivm matrix]$ 
```

Conclusion:

Comment on Matrix Multiplication method

Matrix multiplication using MapReduce distributes the computational workload across multiple nodes by breaking down the multiplication process into smaller tasks. Mappers emit key-value pairs representing partial calculations, which are then grouped and processed by reducers to generate the final matrix. This approach is scalable and fault-tolerant, making it suitable for handling large matrices. However, it can incur significant communication overhead during data shuffling, and balancing the computational load across nodes can be challenging.