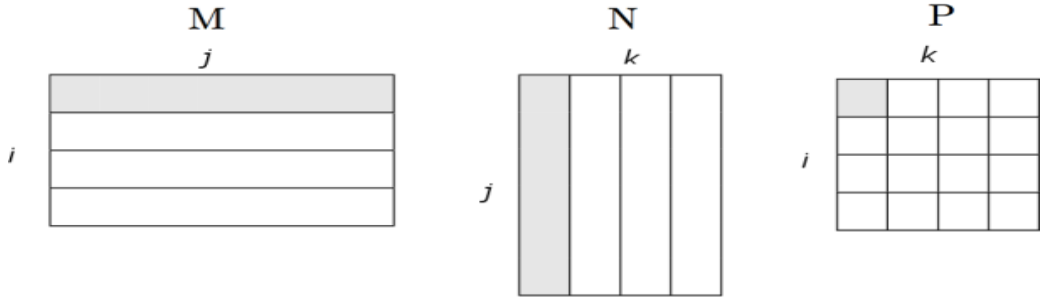
Aim: Implement simple sorting algorithm in Map reduce- Matrix Multiplications.

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

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

Code:

```
from collections import defaultdict

def mapper(matrix_name, i, j, value, N, P):
    if matrix_name == 'A':
        for k in range(P):
            yield (i, k), ('A', j, value)
    elif matrix_name == 'B':
        for i in range(N):
```

```
        yield (i, j), ('B', i, value)

def reducer(key, values):
    A_values = {}
    B_values = {}

    for value in values:
        if value[0] == 'A':
            A_values[value[1]] = value[2]
        elif value[0] == 'B':
            B_values[value[1]] = value[2]

    result = 0
    for k in A_values:
        if k in B_values:
            result += A_values[k] * B_values[k]

    return key, result

A = [[1, 2, 3], [4, 5, 6]]
B = [[7, 8], [9, 10], [11, 12]]

M = len(A)
N = len(A[0])
```

```
P = len(B[0])

intermediate = defaultdict(list)

for i in range(M):
    for j in range(N):
        for key, value in mapper('A', i, j, A[i][j], N, P):
            intermediate[key].append(value)

for i in range(N):
    for j in range(P):
        for key, value in mapper('B', i, j, B[i][j], N, P):
            intermediate[key].append(value)

results = []
for key, values in intermediate.items():
    result = reducer(key, values)
    results.append(result)

sorted_results = sorted(results)
for result in sorted_results:
    print(f"Element {result[0]}: {result[1]}")
```

Output:

```
Element (0, 0): 11
Element (0, 1): 12
Element (1, 0): 55
Element (1, 1): 60
Element (2, 0): 0
Element (2, 1): 0
```

Conclusion :

The MapReduce method for matrix multiplication distributes the workload across multiple machines, enhancing scalability and efficiency. It breaks down the multiplication process into mappers and reducers, handling the computation in a parallelized manner. This approach is particularly useful for multiplying large datasets in a distributed environment, such as Hadoop. However, it may introduce overhead due to communication between nodes