

Lab 6) Implement Matrix Multiplication with Hadoop Map Reduce.

We can represent a matrix as a relation (table) in RDBMS where each cell in the matrix can be represented as a record (I, j, value). As an example, let us consider the following matrix and its representation. It is important to understand that this relation is a very inefficient relation if the matrix is dense. Let us say we have 5 Rows and 6 Columns, then we need to store only 30 values but if you consider above relation we are storing 30 rowid, 30 col_id and 30 values in other sense we are tripling the data. So, a natural question arises why we need to store in this format? In practice most of the matrices are sparse matrices. In sparse matrices not, all cells used to have any values, so we don't have to store those cells in DB. So, this turns out to be very efficient in storing such matrices.

MapReduce Logic

Logic is to send the calculation part of each output cell of the result matrix to reducer. So, in matrix multiplication the first cell of output (0,0) has multiplication and summation of elements from row 0 of the matrix A and elements from col 0 of matrix B. To do the computation of value in the output cell (0,0) of resultant matrix in a separate reducer we need to use (0,0) as output key of MapPhase and value should have array of values from row 0 of matrix A and column 0 of matrix B. Hopefully this picture will explain the point. So in this algorithm output from map phase should be having a <key.value>, where key represents the output cell location (0,0), (0,1) etc. and value will be list of all values required for reducer to do computation. Let us take an example for calculating value at output cell (0,0). Here we need to collect values from row 0 of matrix A and col 0 of matrix B in the map phase and pass (0,0) as key. So, a single reducer can do the calculation.

ALGORITHM

We assume that the input files for A and B are streams of (key,value) pairs in sparse matrix format, where each key is a pair of indices (i,j) and each value is the corresponding matrix element value. The output files for matrix $C=A*B$ are in the same format.

We have the following input parameters:

The path of the input file or directory for matrix A.

The path of the input file or directory for matrix B.

The path of the directory for the output files for matrix C. strategy = 1, 2, 3 or 4.

R = the number of reducers.

I = the number of rows in A and C.

K = the number of columns in

A and rows in B. J = the number of columns in B and

C.

IB = the number of rows per A block and C block.

KB = the number of columns per A block and rows per B block. JB = the number of columns per B block and C block.

In the pseudo-code for the individual strategies below, we have intentionally avoided factoring common code for the purposes of clarity.

Note that in all the strategies the memory footprint of both the mappers and the reducers is flat at scale.

Note that the strategies all work reasonably well with both dense and sparse matrices. For sparse matrices we do not emit zero elements. That said, the simple pseudo-code for multiplying the individual blocks shown here is certainly not optimal for sparse matrices. As a learning exercise, our focus here is on mastering the MapReduce complexities, not on optimizing the sequential matrix multiplication algorithm for the individual blocks.

Steps

1. setup ()
2. var NIB = (I-1)/IB+1
3. var NKB = (K-1)/KB+1
4. var NJB = (J-1)/JB+1
5. map (key, value)
6. if from matrix A with key=(i,k) and value=a(i,k)
7. for $0 \leq j_b < N_{JB}$
8. emit (i/IB, k/KB, j_b, 0), (i mod IB, k mod KB, a(i,k))
9. if from matrix B with key=(k,j) and value=b(k,j)
10. for $0 \leq i_b < N_{IB}$
 emit (i_b, k/KB, j/JB, 1), (k mod KB, j mod JB, b(k,j))

Intermediate keys (i_b, k_b, j_b, m) sort in increasing order first by i_b, then by k_b, then by j_b, then by m. Note that m = 0 for A data and m = 1 for B data.

The partitioner maps intermediate key (i_b, k_b, j_b, m) to a reducer r as follows:

11. $r = ((i_b * J_B + j_b) * K_B + k_b) \bmod R$
12. These definitions for the sorting order and partitioner guarantee that each reducer $R[i_b, k_b, j_b]$ receives the data it needs for blocks $A[i_b, k_b]$ and $B[k_b, j_b]$, with the data for the A block immediately preceding the data for the B block.

13. var A = new matrix of dimension IBxKB
14. var B = new matrix of dimension KBxJB
15. var sib = -1
16. var skb = -1

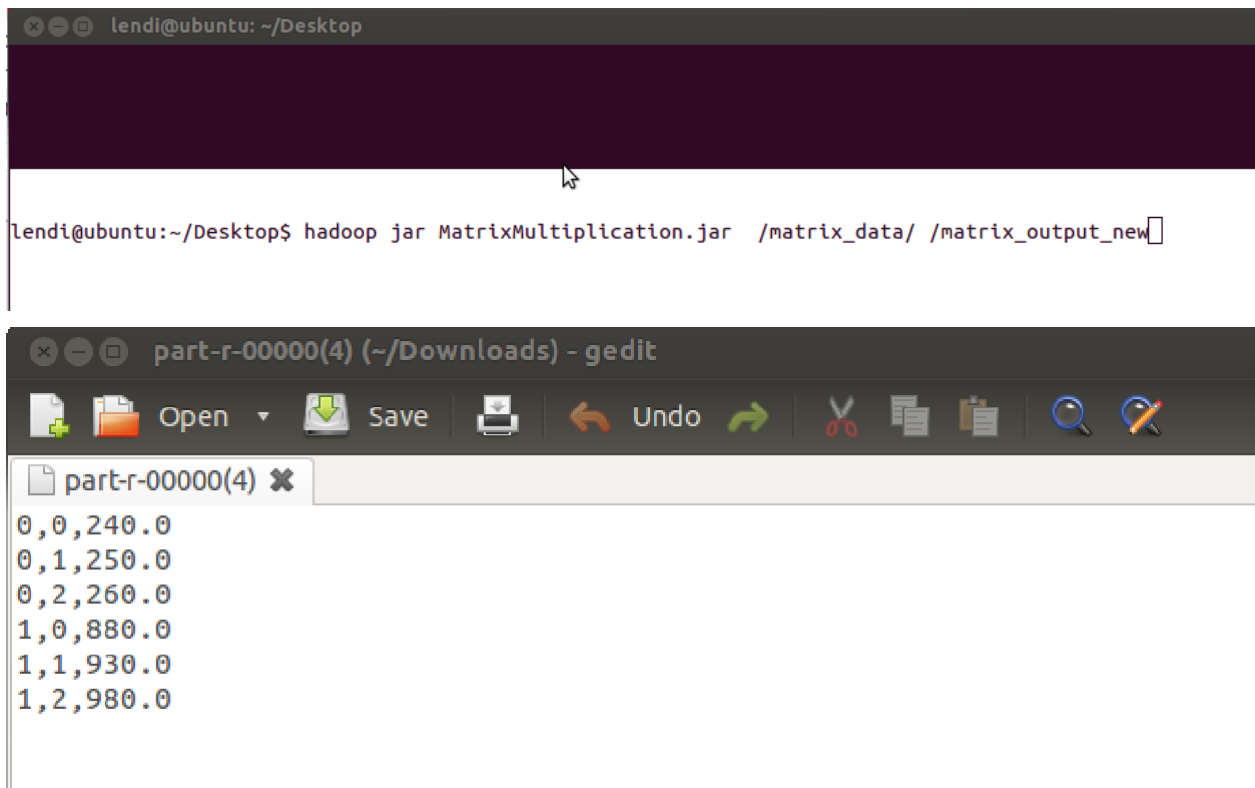
Reduce (key, valueList)

17. if key is (ib, kb, jb, 0)
18. // Save the A block.
19. sib = ib
20. skb = kb
21. Zero matrix A
22. for each value = (i, k, v) in valueList A(i,k) = v
23. if key is (ib, kb, jb, 1)
24. if ib != sib or kb != skb return // A[ib,kb] must be zero!
25. // Build the B block.
26. Zero matrix B
27. for each value = (k, j, v) in valueList B(k,j) = v
28. // Multiply the blocks and emit the result.
29. ibase = ib*IB
30. jbase = jb*JB
31. for 0 <= i < row dimension of A
32. for 0 <= j < column dimension of B
33. sum = 0
34. for 0 <= k < column dimension of A = row dimension of B
 - a. sum += A(i,k)*B(k,j)
35. if sum != 0 emit (ibase+i, jbase+j), sum

INPUT:-

Set of Data sets over different Clusters are taken as Rows and Columns

OUTPUT:-



The image displays two screenshots from a Linux environment. The top screenshot is a terminal window titled 'lendi@ubuntu: ~/Desktop'. It shows a command being entered: 'lendi@ubuntu:~/Desktop\$ hadoop jar MatrixMultiplication.jar /matrix_data/ /matrix_output_new'. The bottom screenshot is a gedit text editor window titled 'part-r-00000(4) (~/.Downloads) - gedit'. It shows the output of the Hadoop command, which consists of seven lines of comma-separated values representing matrix multiplication results.

```
lendi@ubuntu:~/Desktop$ hadoop jar MatrixMultiplication.jar /matrix_data/ /matrix_output_new
```

```
0,0,240.0
0,1,250.0
0,2,260.0
1,0,880.0
1,1,930.0
1,2,980.0
```