Problem 1: Matrix Multiplication (with multiple mapper and reducer)

If M is a matrix with element m_{ij} in row i and column j, and N is a matrix with element n_{jk} in row j and column k, then the product $P = MN$ is the matrix P with element p_{ik} in row i and column k, where

$$p_{ik} = \sum_j m_{ij} n_{jk}$$

It is required that the number of columns of M equals the number of rows of N, so the sum over j makes sense.

We can think of a matrix as a relation with three attributes: the row number, the column number, and the value in that row and column. Thus, we could view matrix M as a relation $M(I,J,V)$, with tuples (i,j,m_{ij}), and we could view matrix N as a relation $N(J,K,W)$, with tuples (j,k,n_{jk}). As large matrices are often sparse (mostly 0's), and since we can omit the tuples for matrix elements that are 0, this relational representation is often a very good one for a large matrix. However, it is possible that i, j, and k are implicit in the position of a matrix element in the file that represents it, rather than written explicitly with the element itself. In that case, the Map function will have to be designed to construct the I, J, and K components of tuples from the position of the data.

The product MN is almost a natural join followed by grouping and aggregation. That is, the natural join of $M(I,J,V)$ and $N(J,K,W)$, having only attribute J in common, would produce tuples (i,j,k,v,w) from each tuple (i,j,v) in M and tuple (j,k,w) in N. This five-component tuple represents the pair of matrix elements (m_{ij},n_{jk}). What we want instead is the product of these elements, that is, the four-component tuple $(i,j,k,v \times w)$, because that represents the product $m_{ij}n_{jk}$. Once we have this relation as the result of one MapReduce operation, we can perform grouping and aggregation, with I and K as the grouping attributes and the sum of $V \times W$ as the aggregation. That is, we can implement matrix multiplication as the cascade of two MapReduce operations, as follows. First:

The Map Function: For each matrix element m_{ij}, produce the key value pair $j,(M, i, m_{ij}))$. Likewise, for each matrix element n_{jk}, produce the key value pair $j,(N, k, n_{jk}))$. Note that M and N in the values are not the matrices themselves. Rather they are names of the matrices or, more precisely, a single bit that indicates whether the element comes from M or N (as we mentioned regarding the similar Map function we used for the natural join):.

The Reduce Function: For each key j, examine its list of associated values. For each value that comes from M, say (M,i,m_{ij}), and each value that comes from N, say (N,k,n_{jk}), produce a key-value pair with key equal to (i,k) and value equal to the product of these elements, $m_{ij}n_{jk}$.

Now, we perform a grouping and aggregation by another MapReduce operation applied to the output of the first MapReduce operation.

The Map Function: This function is just the identity. That is, for every input element with key (i,k) and value v, produce exactly this key-value pair.

The Reduce Function: For each key (i,k), produce the sum of the list of values associated with this key. The result is a pair $(i, k), v)$, where v is the value of the element in row i and column k of the matrix $P = MN$.

Implement the above stated functions for the given dataset (mm.json)

Problem 2: Matrix Multiplication (with One MapReduce Step)

There often is more than one way to use MapReduce to solve a problem. You may wish to use only a single MapReduce pass to perform matrix multiplication $P = MN$.[1] It is possible to do so if we put more work into the two functions. Start by using the Map function to create the sets of matrix elements that are needed to compute each element of the answer P. Notice that an element of M or N contributes to many elements of the result, so one input element will be turned into many key-value pairs. The keys will be pairs (*i,k*), where *i* is a row of *M* and *k* is a column of *N*. Here is a synopsis of the Map and Reduce functions.

The Map Function: For each element m_{ij} of *M*, produce all the key-value pairs $(i,k),\ \ (M,j,m_{ij}))$ for $k = 1,2,\ldots$ up to the number of columns of *N*. Similarly, for each element n_{jk} of *N*, produce all the key-value pairs $(i,k),\ \ (N,j,n_{jk}))$ for $i = 1,2,\ldots$ up to the number of rows of *M*. As before, *M* and *N* are really bits to tell which of the two matrices a value comes from.

The Reduce Function: Each key (*i,k*) will have an associated list with all the values (M,j,m_{ij}) and (N,j,n_{jk}), for all possible values of *j*. The Reduce function needs to connect the two values on the list that have the same value of *j*, for each *j*. An easy way to do this step is to sort by *j* the values that begin with *M* and sort by *j* the values that begin with *N*, in separate lists. The *j*th values on each list must have their third components, m_{ij} and n_{jk} extracted and multiplied. Then, these products are summed and the result is paired with (*i,k*) in the output of the Reduce function.

You may notice that if a row of the matrix *M* or a column of the matrix *N* is so large that it will not fit in main memory, then the Reduce tasks will be forced to use an external sort to order the values associated with a given key (*i,k*). However, in that case, the matrices themselves are so large, perhaps 10^{20} elements, that it is unlikely we would attempt this calculation if the matrices were dense. If they are sparse, then we would expect many fewer values to be associated with any one key, and it would be feasible to do the sum of products in main memory.

Implement the above stated functions for the given dataset (mm.json)