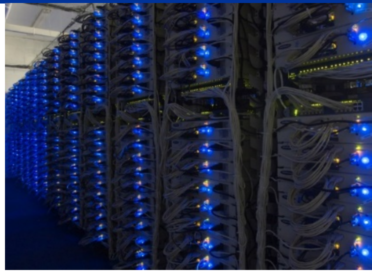
Recap: Cluster Architecture

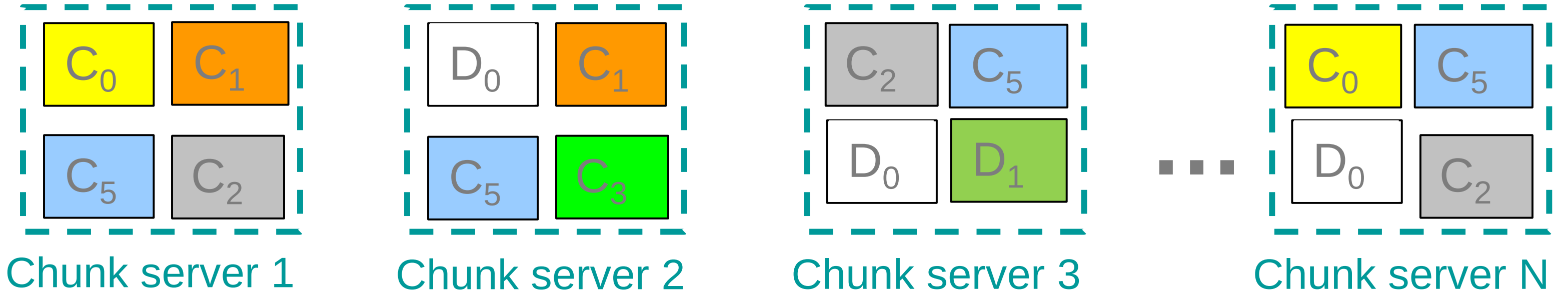


- **Limitations of single node architecture to ingest, store, analyze and process data in a timely manner**
- **Connect low-cost nodes through high-speed networks for high-performance distributed computing**
  - Parallelization and distribution of large-scale computations
- **Build fault-tolerance into the storage infrastructure**
  - Distributed File System
  - Replicate files → Data availability
- **Bring computation close to the data**
  - Data-parallel Programming Model (MapReduce)
  - Divide computations into tasks → Restart failed task without affecting other tasks

Recap: Distributed File System

Reliability and Availability

- **Data kept in "chunks" spread across machines**
- **Each chunk replicated on different machines**
- **Seamless recovery from disk or machine failure**



Bring computation directly to the data

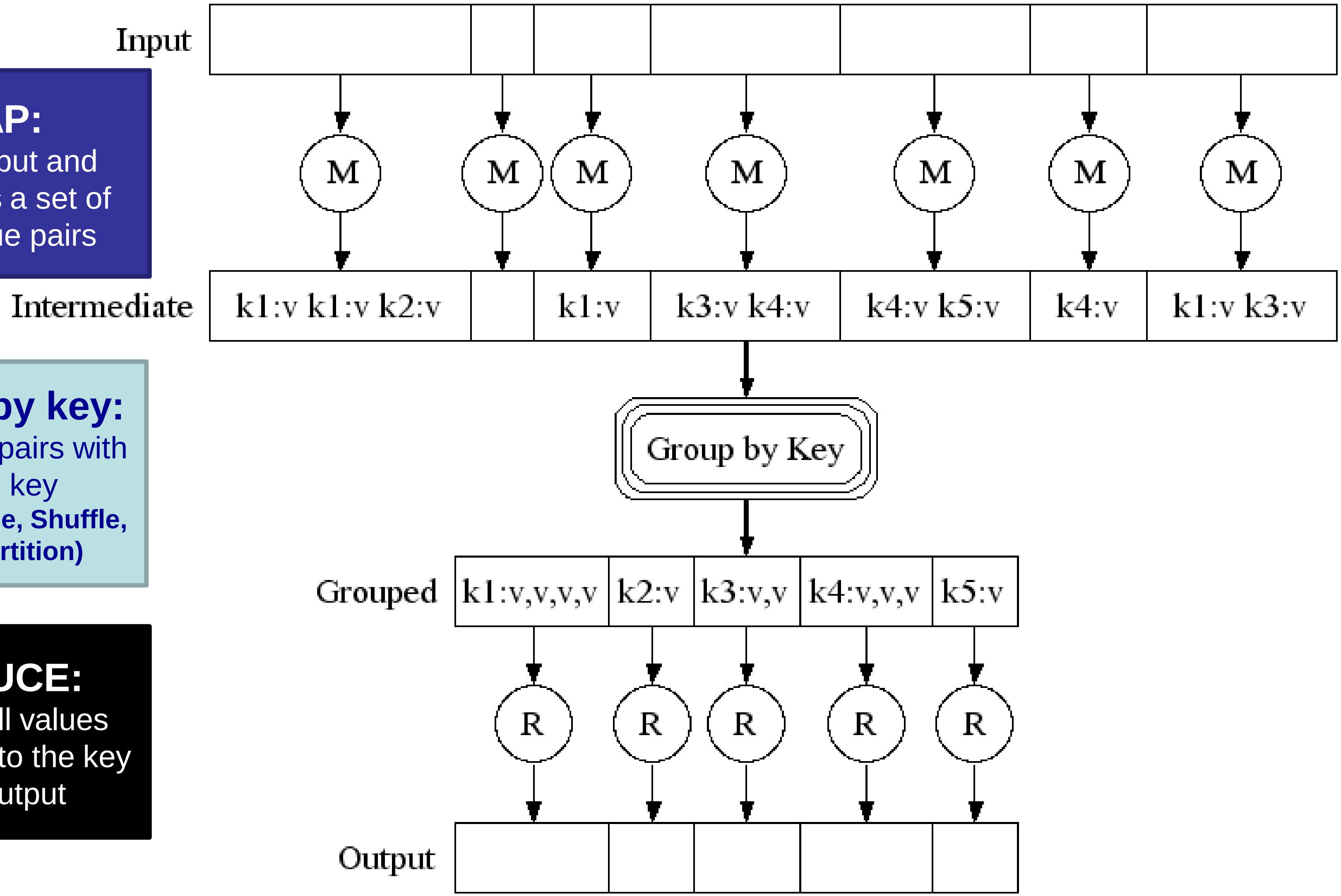
Chunk servers also serve as compute servers

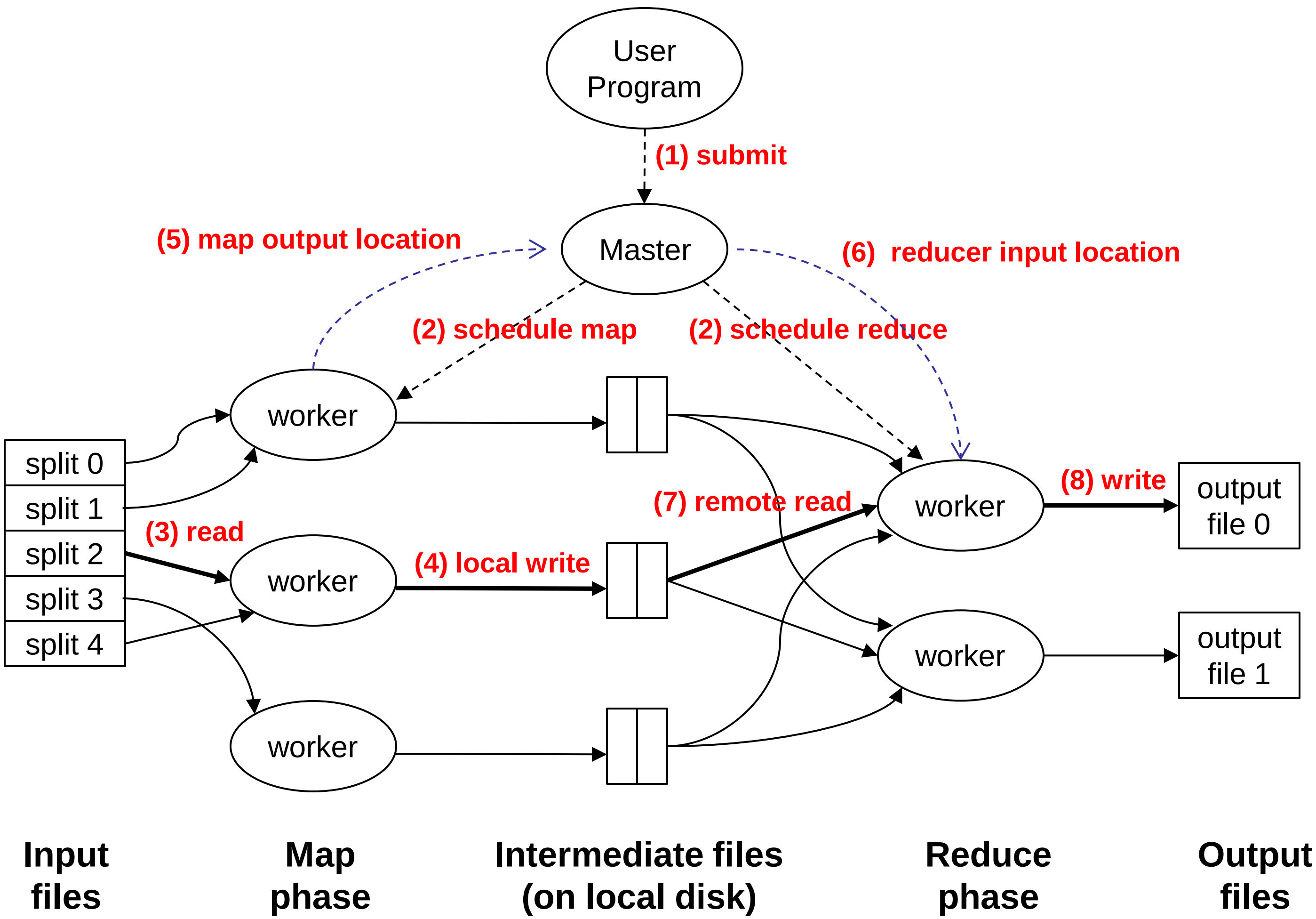
Recap: MapReduce

MAP:
Read input and produces a set of key-value pairs

Group by key:
Collect all pairs with same key
(Hash merge, Shuffle, Sort, Partition)

REDUCE:
Collect all values belonging to the key and output





Output of Reduce task is stored in HDFS for reliability

Grouping by Key

- **User define number of Reduce tasks** *R*
- **System apply hash function to keys, produce a bucket number from** *0* **to** *R – 1*
- **Hash each key output by Map task, put key-value pair in one of the** *R* **local files**
- **Master controller merge files from Map tasks destined for the same Reduce task as a sequence of (key, list of values) pairs**

MapReduce Data Flow

- **Input and final output are stored on a distributed file system (FS)**
  - Scheduler tries to schedule map tasks “close” to physical storage location of input data
  - Push computation to data, minimize network use
- **Intermediate results are stored on local FS of Map workers rather than pushing it directly to Reducers**
  - Allows recovery if a reducer crashes
- **Output can be input to another MapReduce task**

MapReduce Coordination

- **Master node takes care of coordination**
  - **Task status:** (idle, in-progress, completed)
  - **Idle tasks** get scheduled as workers become available
- **When a map task completes, it sends the master the location and sizes of its *R* intermediate files, one for each reducer**
- **Master pushes this information to reducers**
- **Master pings workers periodically to detect failures**

Dealing with Failures

- **Master node fail - Restart entire MapReduce job**
- **Compute node of a Map worker fail**
  - Reset completed or in-progress map tasks at worker to idle
  - Restart **all** the map tasks assigned to this node
  - Inform Reduce workers when task is rescheduled on another worker, location of input from that map task has changed
- **Compute node of a Reduce worker fail**
  - Only reset in-progress tasks to idle
  - Restart these reduce tasks at another node

How many Map and Reduce jobs?

- ***M* map tasks, *R* reduce tasks**
- **Make *M* much larger than the number of nodes in the cluster**
  - One DFS chunk per map is common
  - Improves dynamic load balancing and speeds up recovery from worker failures
- **Usually *R* is smaller than *M***
  - Because output is spread across *R* files

Why Map outputs to Local Disk?

- **Map output is intermediate**
  - To be processed by reduce task to produce final output
  - Can be discarded after job is complete
  - Storing in DFS with replication is overkill
- **Automatically rerun map task on another node to recreate the map output if the node running map task fails before the map output is consumed by reduce task**

Refinement: Backup Tasks

- **Slow workers significantly lengthen job completion time**
    - Other jobs on the machine
    - Bad disks
    - Weird things
- **Solution**
  - Near end of phase, spawn backup copies of tasks
    - Whichever one finishes first “wins”
- **Effect**
  - Dramatically shortens job completion time

Refinement: Combiners

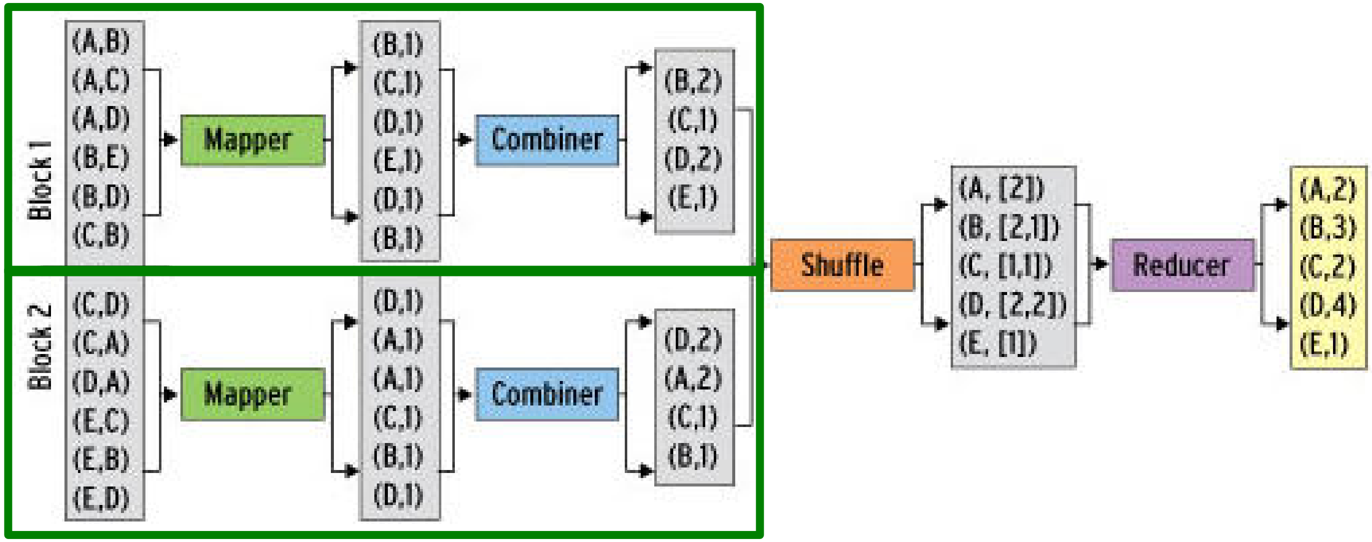
- **A Map task often produce many pairs of the form *(k,v_1), (k,v_2), …* for the same key *k***
  - e.g., popular words in the word count example
- **A Combiner is a local aggregation function for repeated keys produced by the same map**
  - For associative operations such as sum, count, max
  - Decreases size of intermediate data
  - Save network use
- **Example** local counting for Word Count

```
def combiner (key, values): output (key, sum(values))
```

Refinement: Combiners

Word Count Example

Combiner combines the values of same keys of a single mapper (single machine)



Much less data needs to be copied and shuffled!

Refinement: Partition Function

- **Want to control how keys are partitioned**
  - Inputs to map tasks are created by contiguous splits of input file
  - Reduce needs to ensure that records with the same intermediate key end up at the same worker
    - System uses a default partition function: *hash(key) mod R*
    - Sometimes useful to override the hash function:

      e.g., hash(hostname(URL)) mod R ensures URLs from a host end up in the same output file

Exercise

Suppose the input to a Map-Reduce operation is a set of integer values.
The map function takes an integer i and produces the list of pairs (p,i) such that p is a prime divisor of i.
For example, map(12) = [(2,12), (3,12)].
The reduce function is addition.
That is, reduce(p, $[i_1, i_2, \ldots, i_k]$) is $(p, i_1+i_2+\ldots+i_k)$.

What is the output if the input is the set of integers 15, 21, 24, 30, 49?

Algorithms using MapReduce

Matrix-Vector Multiplication

m_{11}	m_{12}	m_{13}
m_{21}	m_{22}	m_{23}
m_{31}	m_{32}	m_{33}

v_1	v_2	v_3

Matrix *M*

- **dimension *n x n***
- **m_{ij} denotes element at row *i* and column *j***

x_1
x_2
x_3

Vector *v*

- **length *n***
- **v_j denotes j^{th} element**

Matrix-vector product is a vector *x* of length *n*

$$x_i = \Sigma\, m_{ij} v_j$$

Matrix-Vector Multiplication

- **Matrix *M* and vector *v* stored in separate files in DFS**
- **Compute node executing map task read vector v**
- **Each map task operate on a chunk of matrix M**
- **Map function**
  - Apply to one element of M
  - Produce key-value pair (i, m_{ij} v_j)
  - All terms of the sum that make up component x_i of the matrix-vector product will get the same key i

m_{11}	m_{12}	m_{13}
m_{21}	m_{22}	m_{23}
m_{31}	m_{32}	m_{33}

v_1	v_2	v_3

- **Reduce function**
  - Sum up all the values associated with a given key i.
  - Result is a pair (i, x_i)

x_1
x_2
x_3

Relational Algebra Operations

- **Relation Links describe Web structure**
- **Two attributes From and To**
- **Row or tuple is a pair of URLs such that there is at least one link from the first URL to the second**
  - Billions of tuples

From	***To***
url1	**url2**
url1	**url3**
url2	**url3**
url2	**url4**

Relational Algebra Operations

- **Relation stored as a file in the DFS**
- **Elements are tuples of the relation**
- **Operations:**
  - Selection
  - Projection
  - Union, Intersection, Difference
  - Natural Join
  - Grouping and Aggregation

Selection

- **Apply condition C to each tuple in relation and output only those tuples that satisfy C**
- **Map function**
  - For each tuple t in R, test if it satisfies C.
  - If yes, output key-value pair (t, t)
- **Reduce function**
  - Identity → Pass each key-value pair to output

Projection

- **Given subset S of attributes, produce from each tuple only the values for the attributes in S**
- **Map function**
  - For each tuple t in R, construct tuple t’ by eliminating values of attributes that are not in S.
  - Output key-value pair (t’, t’)
- **Reduce function**
  - Eliminate duplicates
  - (t’, [t’, t’, …, t’]) → (t’, t’)

Natural Join

- **Join R(A, B) and S(B, C)**
  - Find tuples that agree on the B attributes
  - Use the B-value of tuples as key, and value will be the other attributes and relation name
- **Map function**
  - Each tuple (a, b) of R → key-value pair (b, (R, a))
  - Each tuple (b, c) of S → key-value pair (b, (S, c))
- **Reduce function**
  - Each key value b will be associated with a list of pairs that are either (b, (R, a)) or (b, (S, c))
  - Match all the pairs (b, (a, R)) with all (b, (c, S)) and outputs (a, b, c)

Natural Join

- Use Relation Links to find paths of length 2 in Web
- Triples of URLs (u, v, w) such that there is a link from u to v and from v to w
- Natural join of Links with itself

From	*To*
url1	url2
url1	url3
url2	url3
url2	url4

From	*To*
url1	url2
url1	url3
url2	url3
url2	url4

- May not want entire path but pairs (u, w)

Grouping and Aggregation

- **Let R (A, B, C), apply operator** $\gamma_{A,\ \theta(B)}\ (R)$
- **Map function perform grouping**
  - Each tuple (a, b, c) → key-value pair (a, b)
- **Reduce function perform aggregation**
  - Each key *a* represents a group
  - Apply aggregate operator θ to the list [b_1, b_2, …, b_n] of B-values associated with key *a*
  - Output is a pair (a, x) where x is the result of applying θ to the list
    - If operator θ is SUM, then $x = b_1 + b_2 + \ldots + b_n$
    - If operator θ is MAX, then x is the largest of b_1, b_2, …, b_n

Grouping and Aggregation

- **Social network site has relation Friends(User, Friend)**
- **Tuples are pairs (*a, b*) such that *b* is a friend of *a***
- **Statistics on number of friends members have**
  - Compute a count of the number of friends of each user
- **Done by grouping and aggregation**

$$\gamma_{\textit{User, COUNT (Friend)}}\,\textit{(Friends)}$$

  - Group all the tuples by user → one group for each user
  - For each group count the number of friends
  - One tuple for each group, e.g. (Sally, 300)

Matrix Multiplication

m_{11}	m_{12}	m_{13}
m_{21}	m_{22}	m_{23}
m_{31}	m_{32}	m_{33}
m_{41}	m_{42}	m_{43}

n_{11}	n_{12}	n_{13}
n_{21}	m_{22}	n_{23}
n_{31}	n_{32}	n_{33}

- **Matrix *M* with element m_{ij} , Matrix *N* with element n_{jk}**
- **Product P = MN is the matrix P with element p_{ik}**

$$p_{ik} = \Sigma_j \, m_{ij} \, n_{jk}$$

- **Matrix as a relation with 3 attributes: row number, column number and value in that row and column**

  Relation M (I, J, V) with tuples (i, j, m_{ij})

  Relation N (J, K, W) with tuples (j, k, n_{jk})

  Sparse matrix, omit tuples for zero elements

- **Product MN is a natural join (common attribute j) followed by grouping and aggregation**
  - Implemented as the cascade of 2 MapReduce operations

Matrix Multiplication

- **Join M (I, J, V) and N (J, K, W)**
  - Find tuples that agree on attribute J
  - Produce tuples (i, j, k, v, w)
- **Five-component tuple represents the pair of matrix elements (m_{ij}, n_{jk})**

  → four-component tuple (i, j, k, v *x* w) represents the product $m_{ij}\, n_{jk}$
  - Perform grouping and aggregation
  - Use I and K as grouping attribute and sum of V x W as aggregation

Matrix Multiplication

- **Map function**
  - Each matrix element m_{ij} → key-value pair (j, (M, i, m_{ij}))
  - Each matrix element n_{jk} → key-value pair (j, (N, k, n_{jk}))
- **Reduce function**
  - Each key j will be associated with a list of values that are either (M, i, m_{ij}) or (N, k, n_{jk})
  - Match all the pairs (M, i, m_{ij}) and (N, k, n_{jk}) to produce a key-value pair with key (i, k) and value equals to product of m_{ij} and n_{jk}
- **Perform grouping and aggregate with another MR op**
  - Map function is an identity
  - Reduce function sum the list of values associated with key (i, k)
  - Result is a pair ((i, k), v) where v is the value of the element in row i and column k of matrix P = MN

Summary

- **Cluster Computing**
  - Cluster of compute nodes for large scale applications
- **Distributed File Systems**
  - Architecture for very large scale file systems (chunks, replication)
- **MapReduce**
  - Data-parallel programming system (robust to hardware failure)
  - **Map and Reduce functions** (written by user, key-value pairs)
- **Hadoop**
  - Open-source implementation of a DFS (HDFS) and MapReduce
- **Applications of MapReduce**
  - Matrix-vector and matrix-matrix multiplication
  - Relational algebra operators e.g. join, grouping and aggregation