Chapter 2

MapReduce and the New Software Stack

Need for a New Software Stack (1/2)

- Modern data mining applications (i.e., Big data analysis) requires us to manage ***immense amount*** of data ***quickly***
  - (ex) the ranking of Web pages, searches in “friends” networks at SNS

- To deal with these applications, a ***new software stack*** has evolved
  - Gets parallelism from ***computing clusters***
    - Large collections of commodity hardware
    - Conventional processors connected by Ethernet cables or inexpensive switches
  - Begins with a form of a ***distributed file system (DFS)***
    - Uses much larger unit than the disk blocks in a conventional OS
    - Provide data replication to protect against the frequent media failures

Need for a New Software Stack (2/2)

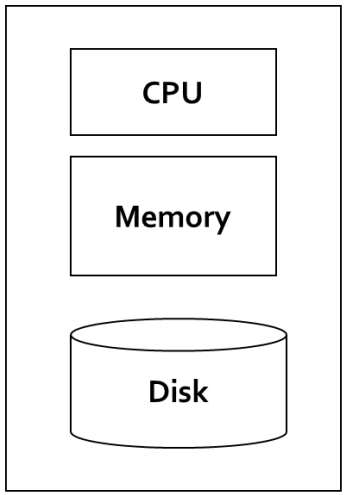
- Much of the course will be devoted to ***large scale computing*** for ***data mining***

- Challenges
  - How to distribute computation?
  - Distributed/parallel programming is hard

- ***MapReduce*** addresses all of the above
  - Google's computational/data manipulation model
  - Elegant way to work with big data

Motivation: Google Example

- 20+ billion web pages x 20KB = 400+ TB
- 1 computer reads 30-35 MB/sec from disk
  - ~4 months to read the web
- ~1,000 hard drives to store the web



Statistics
Machine Learning
"Classical" Data Mining

- Takes even more to ***do*** something useful with the data!
- Today, a standard architecture for such problems is emerging:
  - Cluster of commodity Linux nodes
  - Commodity network (ethernet) to connect them

MapReduce

- A high-level ***programming system*** on top of DFS
  - Central to the new software stack

- Enables many of common calculations on ***large-scale data*** to be performed on ***computing clusters*** efficiently
  - In a way that is ***tolerant*** of hardware failures during the computation

- MapReduce systems are evolving and extending rapidly
  - Today, it is common for MapReduce programs to be created from still higher-level programming systems (e.g., SQL systems)
  - Turns out to be a useful, but simple, case of more general and powerful ideas

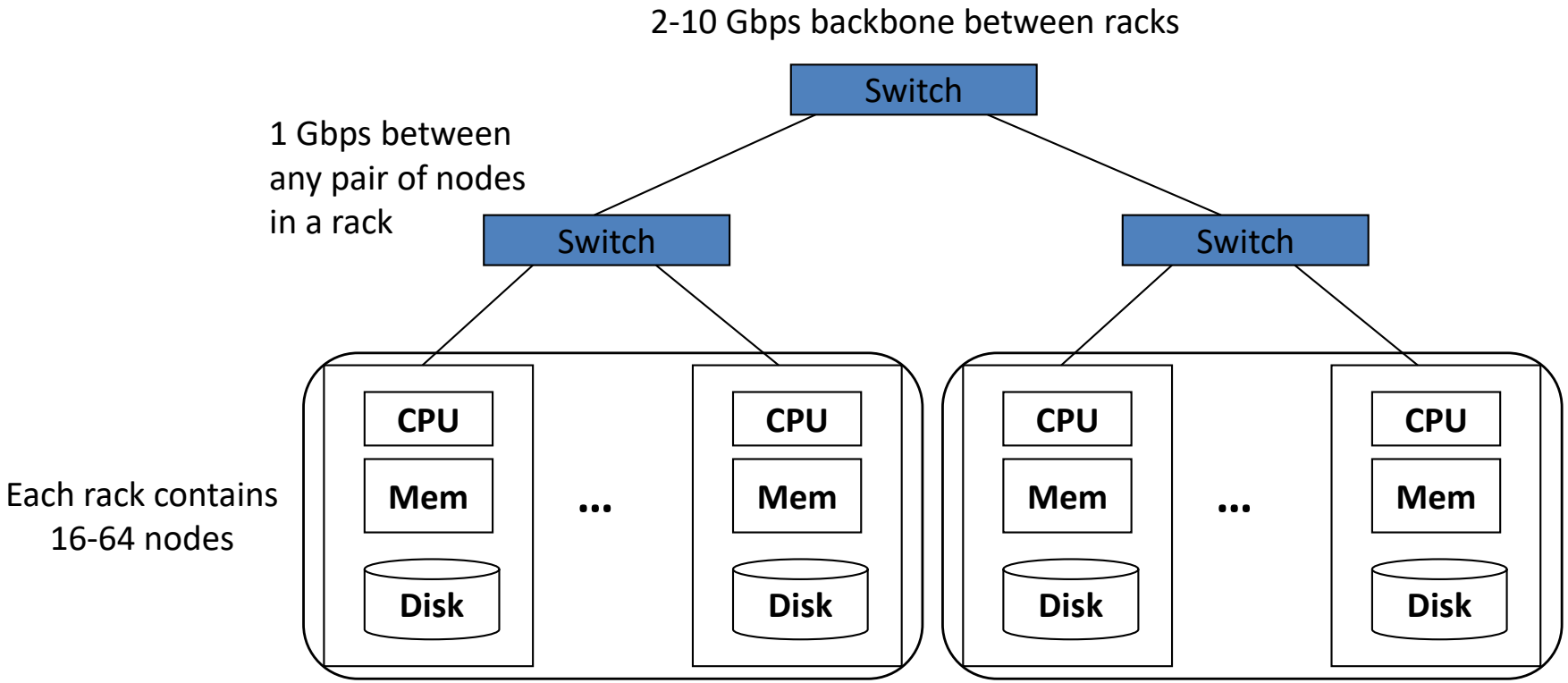
Distributed File Systems

- The prevalence of large-scale Web services has caused more and more computing to be done on ***thousands*** of compute nodes operating independently
  - Furthermore, the compute nodes are ***commodity*** hardware

- These new computing facilitates have given rise to a new generation of programming systems
  - Take advantage of the power of ***parallelism***
    - There are thousands of independent components
  - Avoid the ***reliability*** problems
    - Any of them could fail at any time

Physical Organization of Compute Nodes

- Compute nodes are stored on ***racks*** (perhaps 16 ~ 64 on a rack)
  - The nodes on a rack are connected by a network, typically gigabit Ethernet
- Racks are connected by another level of network or a ***switch***
  - Bandwidth: inter-rack communication > the intra-rack Ethernet



Large-Scale Computing

- Large-scale computing for data mining problems on ***commodity hardware***

- Challenges
  - How do you distribute computation?
  - How can we make it easy to write distributed programs?

- Machines ***fail***
  - One server may stay up 3 years (1,000 days)
  - If you have 1,000 servers, expect to loose 1/day
  - People estimated Google had ~1M machines in 2011
  - 1,000 machines fail every day!

Component Failure

- The more components, the more frequently something in the system will not be working at any given time
  - A disk, a single node, an entire rack, a switch, …
- Some calculations take minutes or even hours
  - If we had to abort and restart the computation every time one component failed, the computation might never complete successfully
- Solutions
  - Store files redundantly
    - If one node failed, its files would be available from other nodes
  - Divide computations into tasks
    - If any one task fails, it can be restarted without affecting other tasks

Large-Scale File System Organization

- Typical usage of a distributed file system (DFS)
  - Files can be enormous, possibly a terabyte in size
  - Files are rarely updated, and possibly data is appended to files sometimes

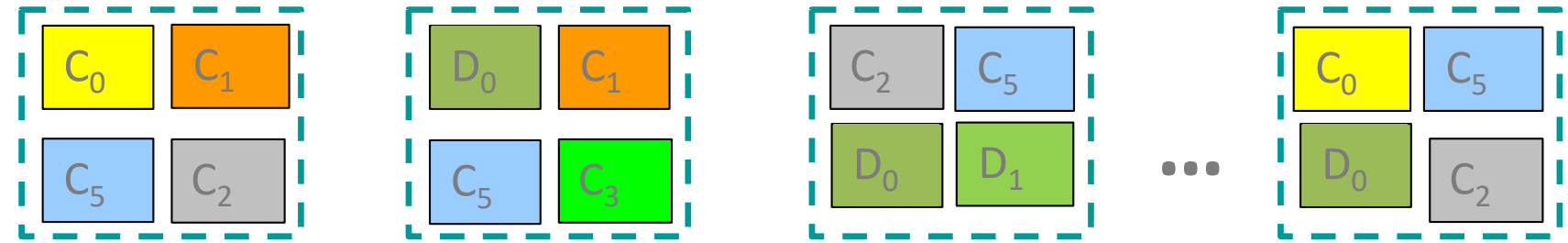
- Typical organization of DFS
  - Files are ***divided*** into chunks (typically 64 MB in size)
  - Chunks are ***replicated*** at three different compute nodes
    - The three nodes should be located on different racks
  - ***Master node*** (or name node)
    - Stores metadata about where the chunks of each file are stored
    - Used to locate the chunks of a file
    - Itself might be replicated

Distributed File System

- **Chunk servers**
  - File is split into contiguous chunks
  - Typically each chunk is 16-64 MB
  - Each chunk replicated (usually 2x or 3x)
  - Try to keep replicas in different racks
- **Master node** (a.k.a. Name Node in Hadoop's HDFS)
  - Stores metadata about where files are stored
  - Might be replicated
- **Client library for file access**
  - Talks to master to find chunk servers
  - Connects directly to chunk servers to access data

“Reliable” Distributed File System

- 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

DFS Implementations

- The Google File System (GFS)
  - The original of the class
- Hadoop Distributed File Systems (HDFS)
  - An open-source DFS used with Hadoop
  - An implementation of MapReduce
  - Distributed by the Apache Software Foundation
- CloudStore
  - An open-source DFS originally developed by Kosmix

MapReduce

- A style of programming that makes parallel computations easy
  - (ex) Google's MapReduce, Hadoop MapReduce

- Used to manage many ***large-scale*** computations in a way that is ***tolerant*** of hardware faults

- All you need is to write two functions: ***Map*** and ***Reduce***
  - The system will manage all the other aspects of the execution
    - Parallel execution
    - Coordination of tasks that execute Map or Reduce
    - Failure of these tasks

Execution of a MapReduce Computation

1. Some number of Map tasks each are given one or more chunks from a DFS

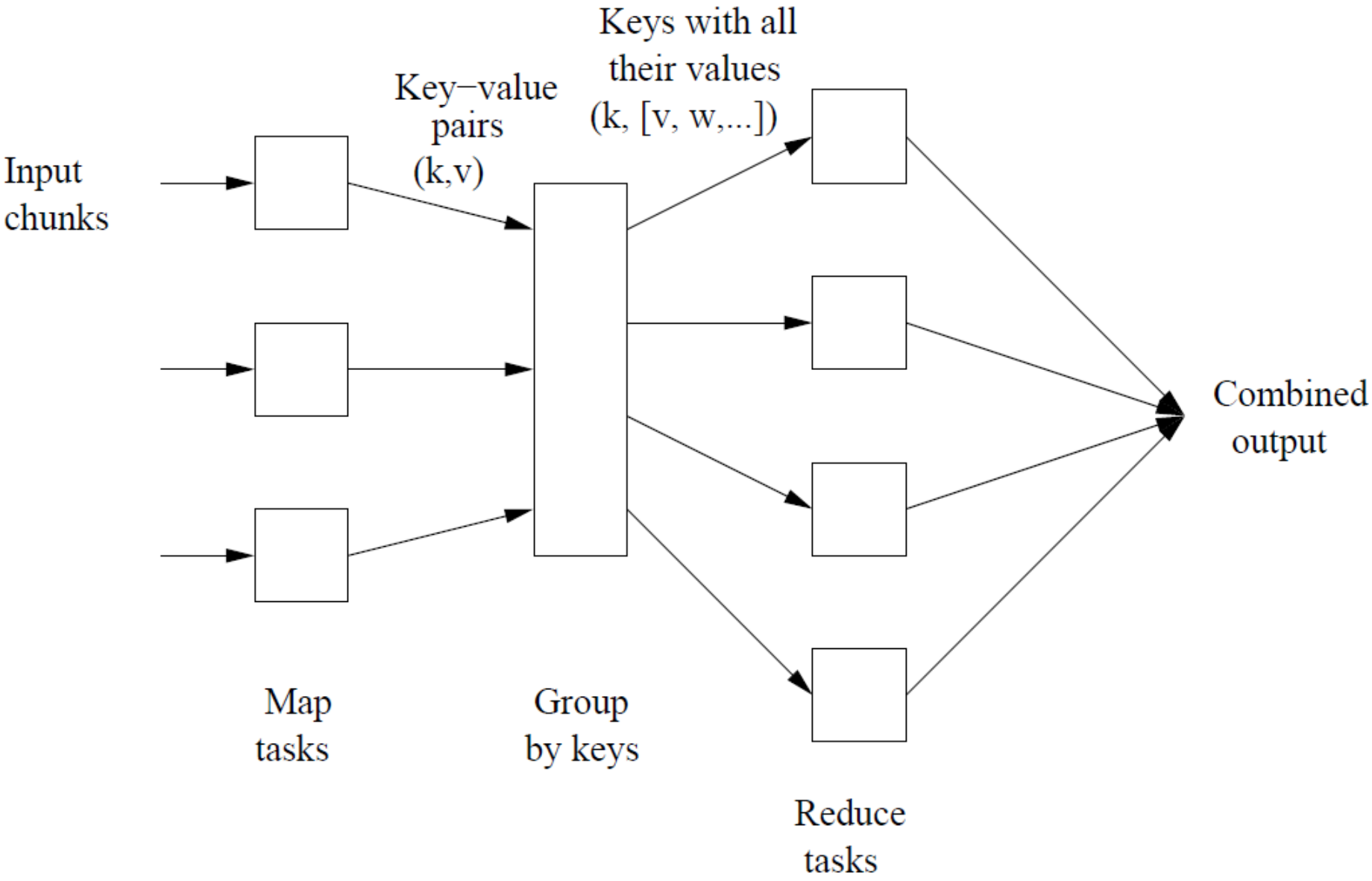
2. These Map tasks turn the chunk into a sequence of key-value pairs, according to the ***Map*** function

3. The key-value pairs from each Map task are collected by a master controller and sorted by key

4. The keys are divided among all the Reduce tasks, so all key-value pairs with the same key wind up at the same Reduce tasks

5. The Reduce tasks work on one key at a time, and combine all the values associated with that key according to the ***Reduce*** function

Schematic of a MapReduce Computation



Map Tasks

- Input files consist of ***elements*** (e.g., tuples or documents)
  - A chunk is a collection of elements
  - No element is stored across two chunks

- Map function
  - Input: an element
  - Output: zero or more key-value pairs
    - The types of keys and values are each arbitrary
    - Keys do not have to be unique
    - A Map task can produce several key-value pairs with the same key, even from the same element

Grouping by Key

- As soon as the Map tasks have all completed successfully, the key-values are grouped by key
  - The grouping is performed by the ***system***

- Basic steps (performed by the system)
  - Picks a hash function $h(k)$, where $h(k) = 0, 1, \ldots, r - 1$
    - r: the number of Reduce tasks (typically specified by the user)
  - For each key-value pair (k, v) output by a Map task, k is hashed and (k, v) is put in one of r local files
    - Each file is destined for one of the Reduce tasks
  - Merges the files from each Map task that are destined for a particular Reduce task and feeds the merged file to that process as $(k, [v_1, v_2, \ldots, v_n])$
    - Where $(k, v_1), (k, v_2), \ldots, (k, v_n)$ are all the key-value pairs with key k coming from all the Map tasks

Reduce Tasks

- Reduce function
  - Input: a pair consisting a key and its list of values, i.e., $(k, [v_1, v_2, \ldots, v_n])$
  - Output: a sequence of zero or more key-value pairs
    - These key-value pairs can be a different type from those from Map tasks

- The output from all the Reduce tasks are merged into a ***single*** file
  - Corresponds to the final result of the MapReduce computation

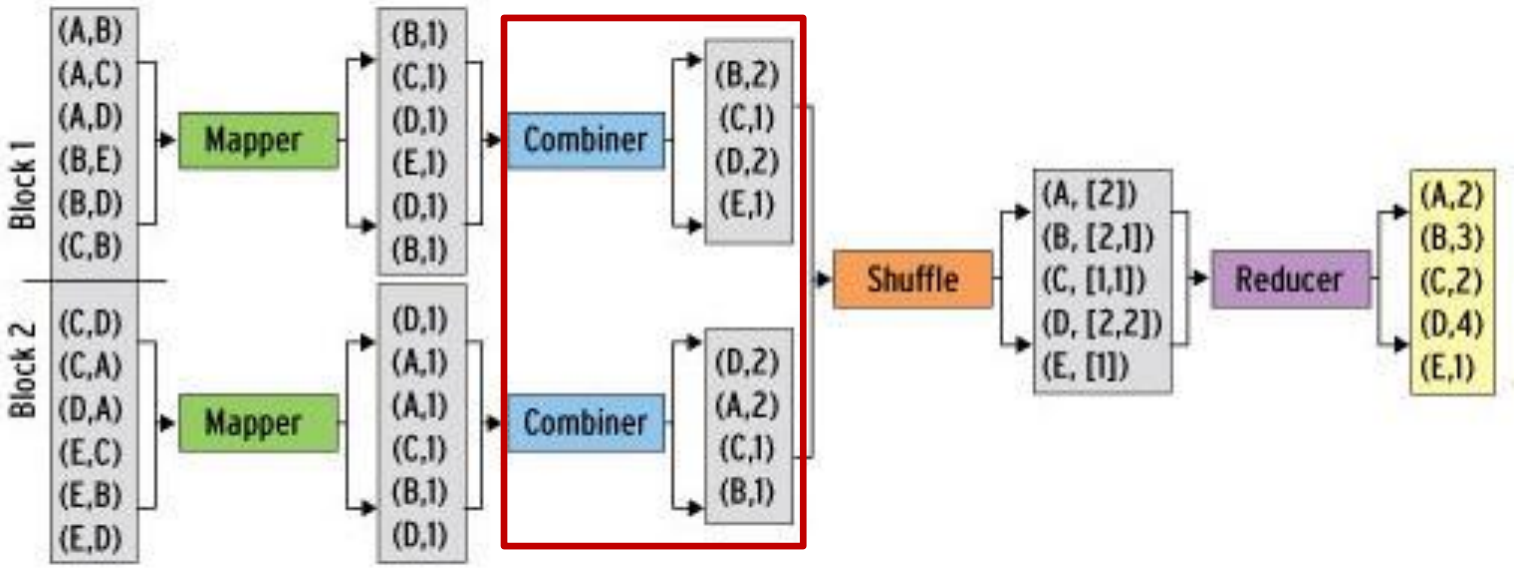
✓ [Note] the following terms have different meanings

- ***Reduce function***: the code written by the user
- ***Reducer***: the application of the Reduce function to a single key and its associate list of values
- ***Reduce task***: executes one or more reducers

Combiners

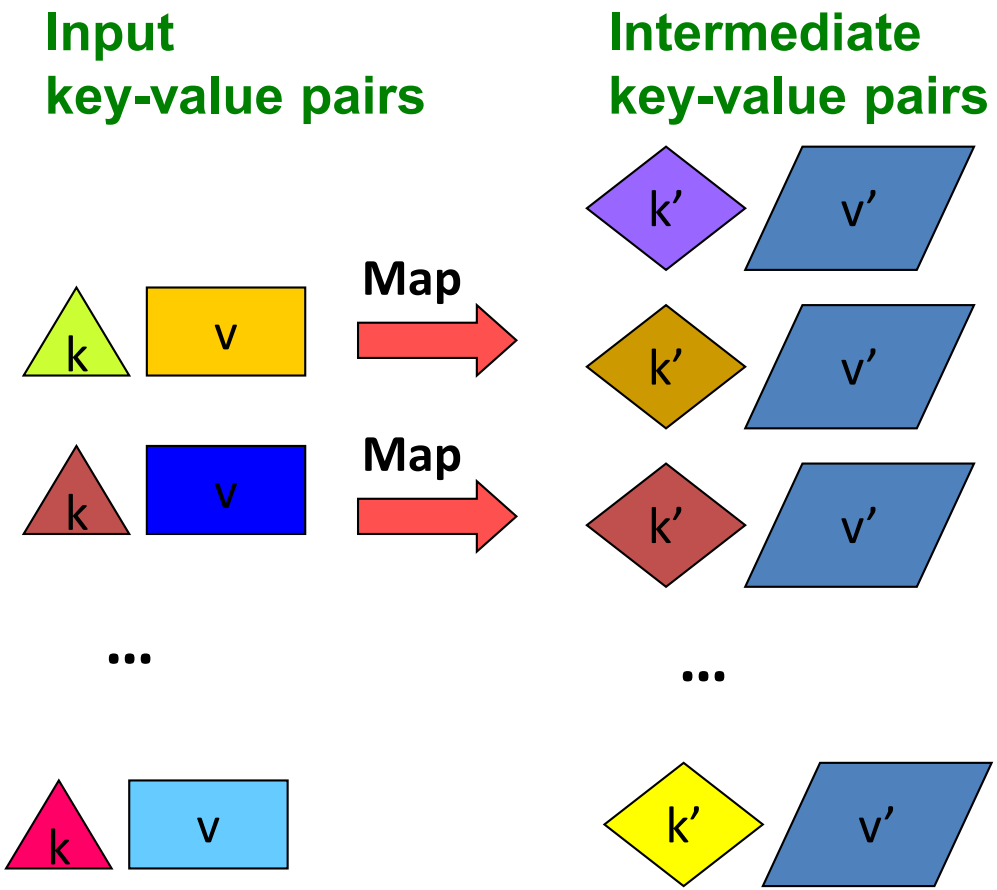
- Sometimes, the Reduce function is associative and communitive
  - That is, the values can be combined in any order, with the same result

- In this case, we can apply the Reduce function within the Map task, before the output of Map tasks is subjected to grouping



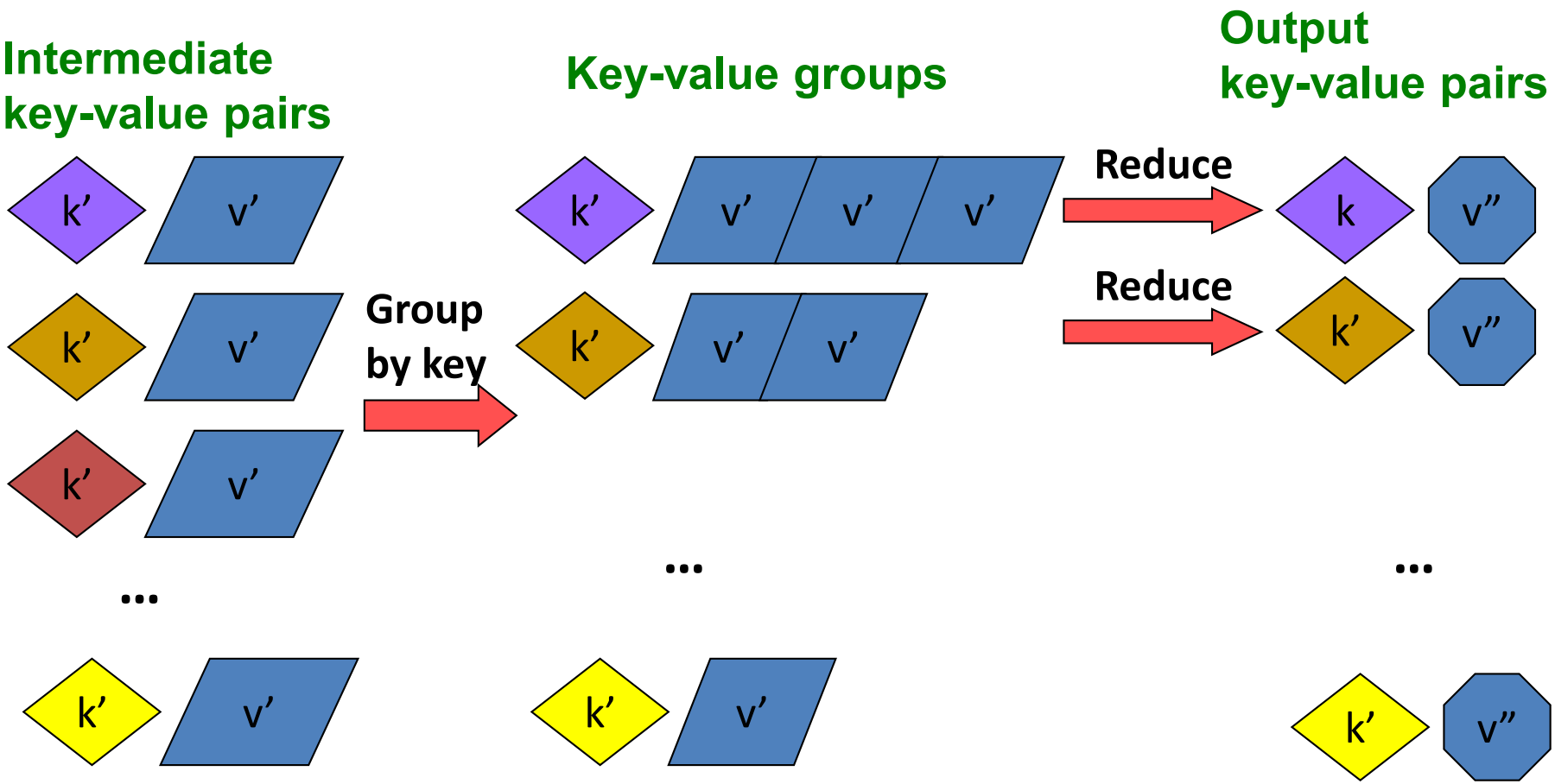
- ***Much less*** data needs to be copied and shuffled!

Summary: MapReduce (The Map Step)



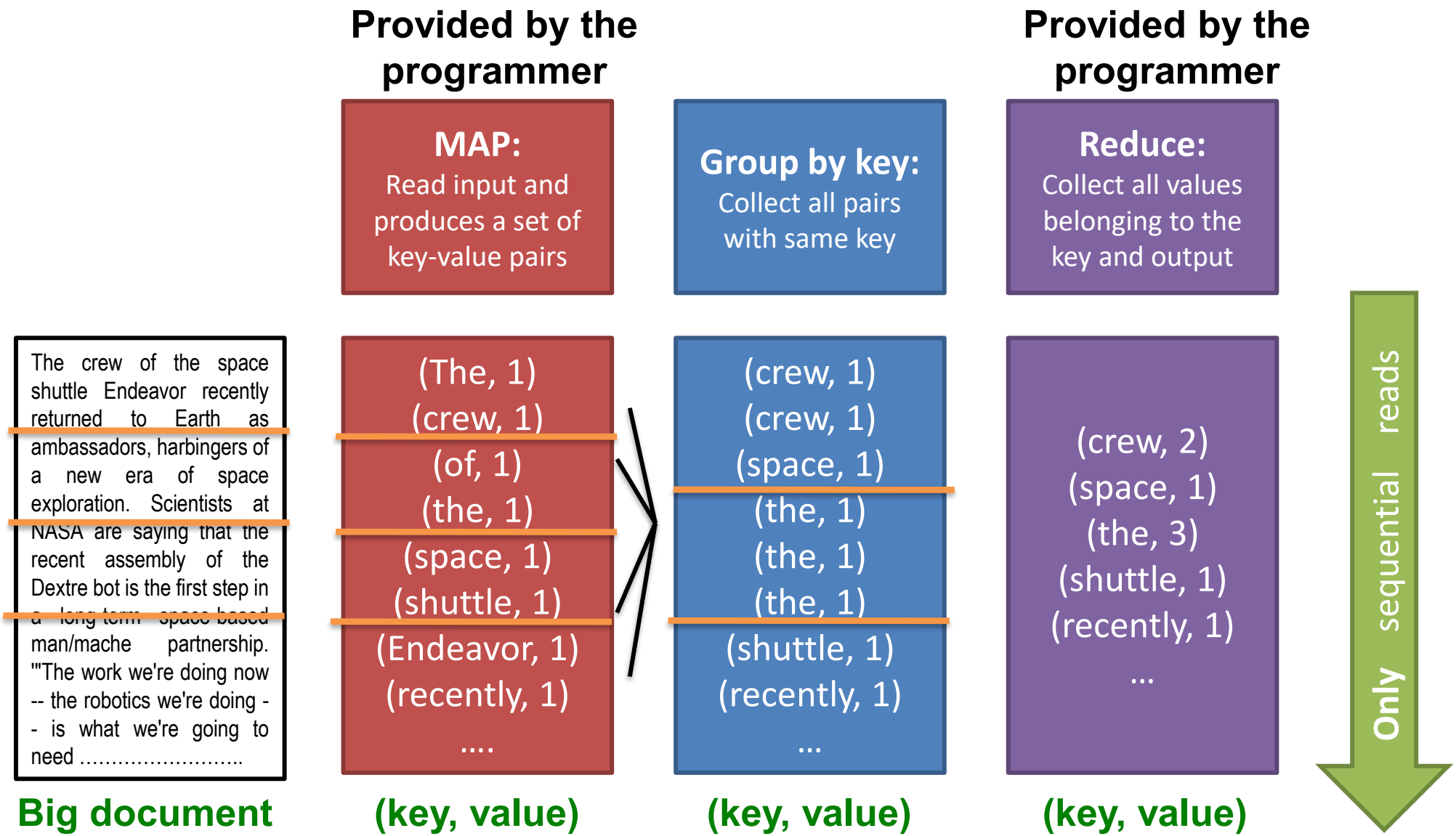
- Map(k, v) → <k', v'>*
  - Takes a key-value pair and outputs a set of key-value pairs

Summary: MapReduce (The Reduce Step)



- Reduce(k’, <v’>*) → <k’, v”>*
  - All values v’ with same key k’ are reduced together into v”

(Ex) Word Count Using MapReduce (1/2)



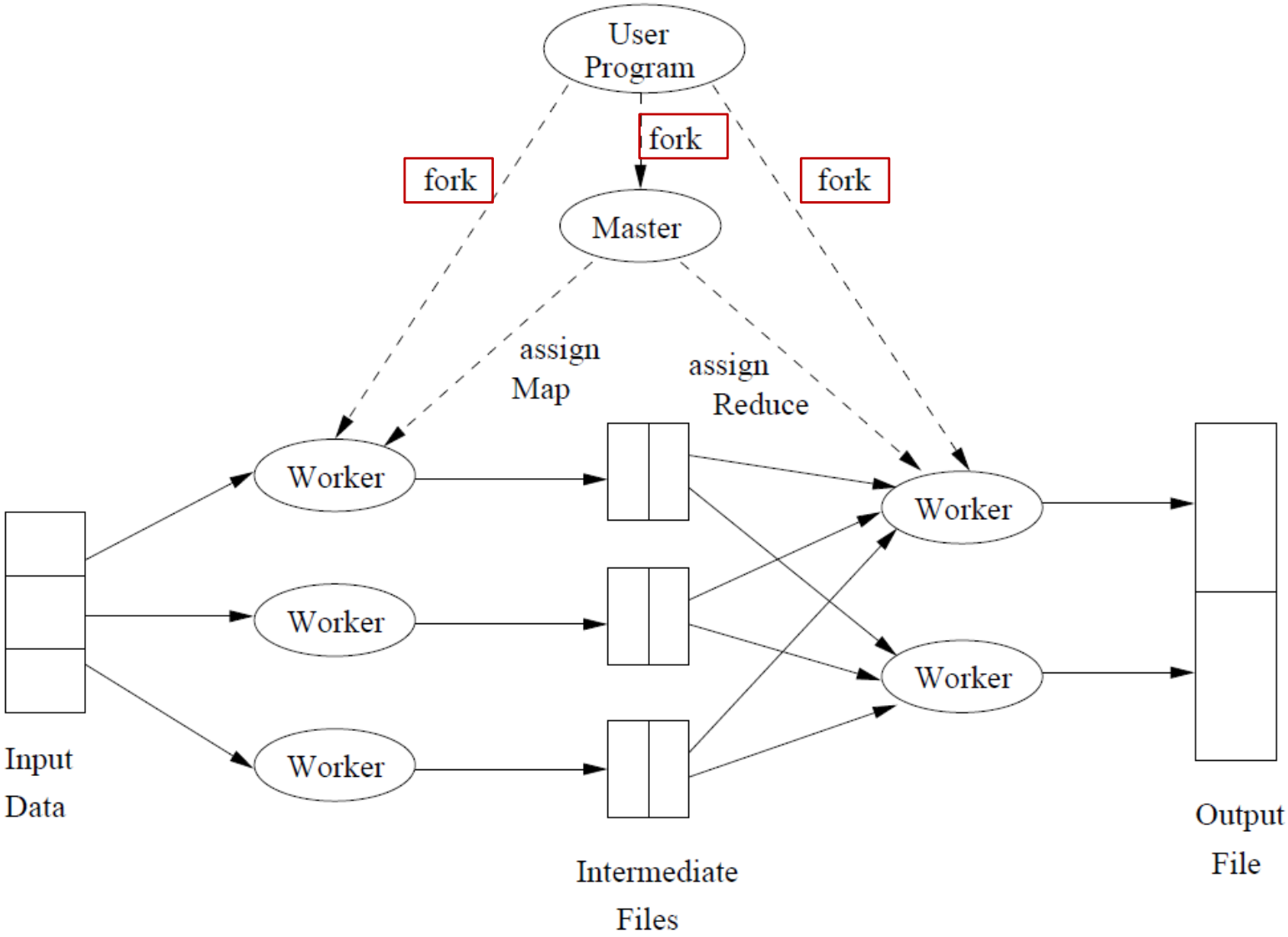
(Ex) Word Count Using MapReduce (2/2)

```
map(key, value):
// key: document name; value: text of the document
   for each word w in value:
       emit(w, 1)


reduce(key, values):
// key: a word; values: an iterator over counts
       result = 0
       for each count v in values:
             result += v
       emit(key, result)
```

Details of MapReduce Execution

- Overview of the execution of a MapReduce program

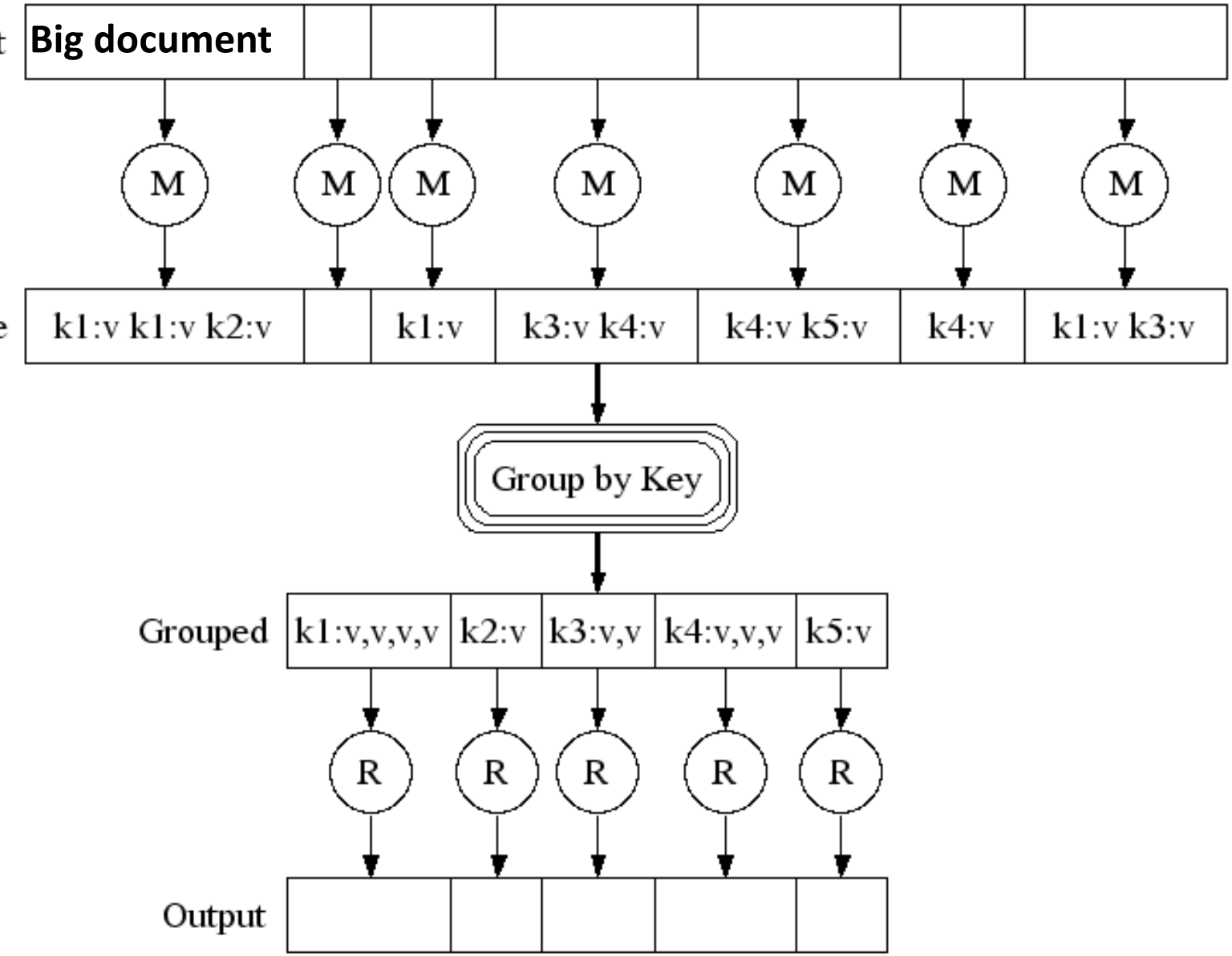


MapReduce: A Diagram

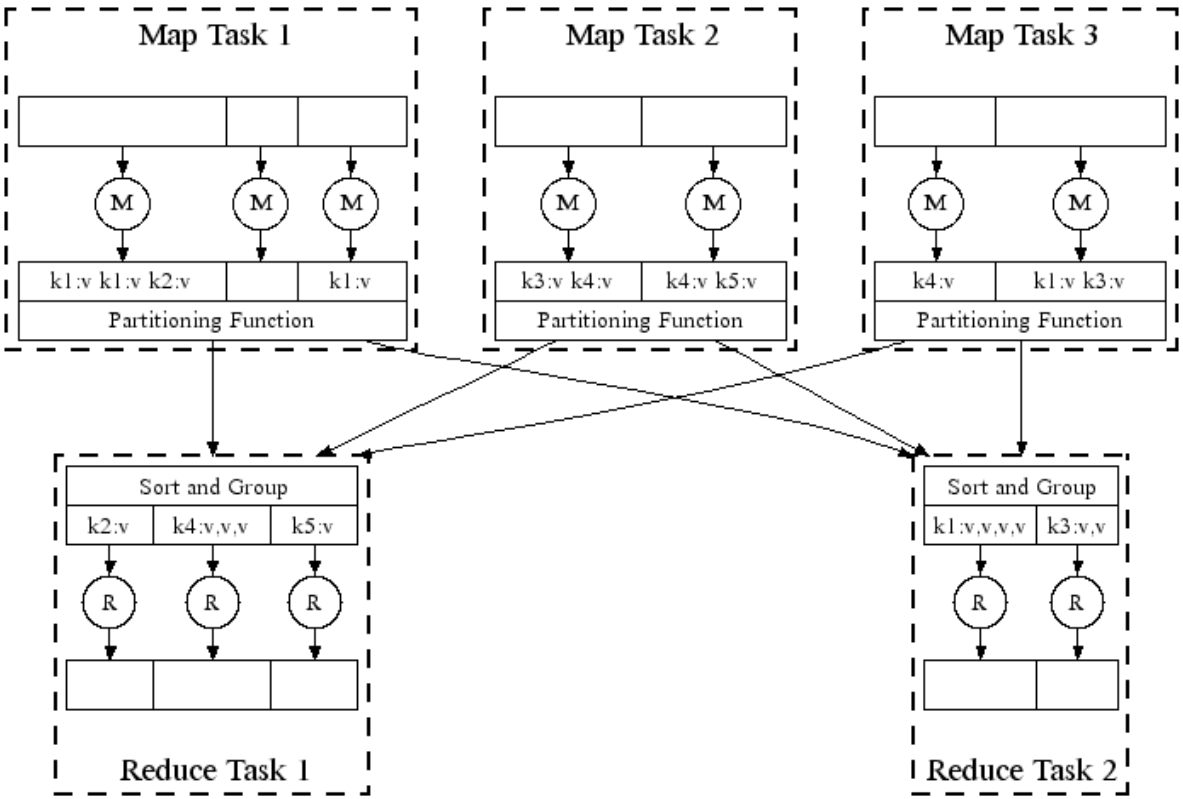
MAP:
Read input and produces a set of key-value pairs

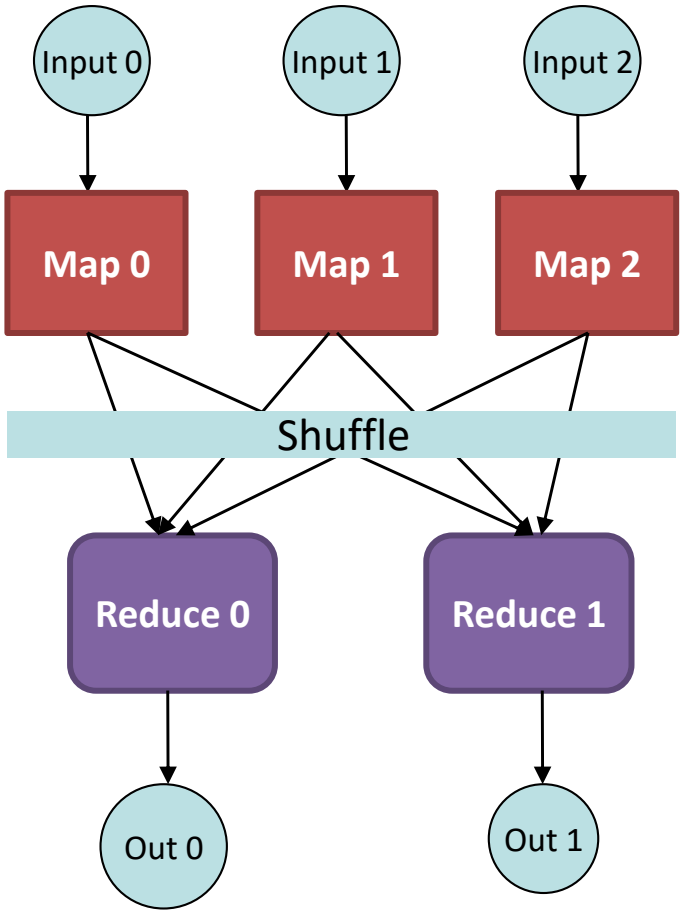
Group by key:
Collect all pairs with same key
(Hash merge, Shuffle, Sort, Partition)

Reduce:
Collect all values belonging to the key and output



MapReduce In Parallel





✓ All phases are distributed with many tasks doing the work

Master

- Creates some number of Map and Reduce tasks
- Assigns these tasks to Worker processes
- Keeps track of the status of each Map and Reduce task
  - Status: Idle, in-progress, or completed
  - Idle tasks get scheduled as workers become available
- When a Map task completed, it sends the Master the location and sizes of its R local files, and the Master pushes this information to Reduce tasks
- Pings workers periodically to detect failures

Worker

- Handles either Map tasks (a Map worker) or Reduce tasks (a Reduce worker), but not both
- Reports to the Master when it finishes a tasks, and a new task is scheduled by the Master for that Worker process
- Map tasks
  - Assigned one or more chunks of the input files(s)
  - Executes the Map function code written by the user
  - Creates a file for each Reduce task on its local disk
- Reduce tasks
  - Collects all its inputs from the files created by Map tasks
  - Executes the Reduce function code written by the user
  - Writes its output to a file in the distributed file system

Coping With Node Failures

- Master failure
  - The worse case that can happen
  - The entire MapReduce job is aborted and the client is notified
- Map worker failure
  - Map tasks completed or in-progress at the worker are reset to idle
    - Even if they had completed, their output is now unavailable
  - Reduce workers are notified when the Map task is rescheduled on another worker
- Reduce worker failure
  - Only in-progress tasks are reset to idle
  - These tasks are rescheduled on another reduce worker later

How Many Map and Reduce Tasks?

- M Map tasks, R Reduce tasks

- Rule of a thumb
  - Make M much larger than the number of nodes in the cluster
  - Creating one Map task for every chunk of the input file(s) 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
  - If there are too many Reduce tasks, the number of intermediate files each Map task has to create explodes

Algorithms Using MapReduce

- MapReduce is ***not*** a solution to every problem
  - (ex) online retail sales (i.e., the operations involve relatively little calculation and change the database frequently)

- MapReduce is useful for performing ***analytic*** queries on ***large*** amounts of data
  - (ex) finding for each user those users whose buying patterns were similar

- We consider the following algorithms
  - Matrix-vector multiplication (the original purpose for the Google's MR)
  - Relational-algebra operations (selection, projection, join, aggregation, …)
  - Matrix multiplication

Matrix-Vector Multiplication

- Notations
  - M: an $n \times n$ matrix, whose element in row i and column j is m_{ij}
  - $\mathbf{v}$: a vector of length n, whose jth element is v_j
  - $\mathbf{x}$: the matrix-vector product of length n, whose ith element is x_i

$$x_i = \sum_{j=1}^{n} m_{ij} v_j$$

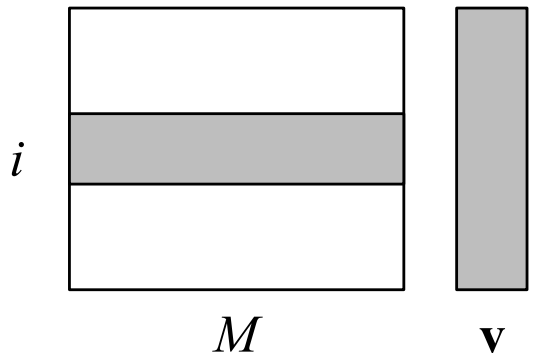
$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

- Assumptions
  - n is very large (e.g., the number of all Web pages (the tens of billions))
  - However, $\mathbf{v}$ fits in main memory
  - Each element of M is stored in a file of the DFS as (i, j, m_{ij})
  - Each element of $\mathbf{v}$ is stored in a file of the DFS as (j, v_j)

Matrix-Vector Multiplication by MapReduce

- Map function
  - If $\mathbf{v}$ is not already read into memory at the node, first read $\mathbf{v}$ into memory
    - Subsequently, $\mathbf{v}$ will be available to all applications of the Map function
  - Input: a matrix element (i, j, m_{ij})
  - Output: a key-value pair $(i, m_{ij}v_j)$
    - All terms of the sum that make up x_i will get the same key, i

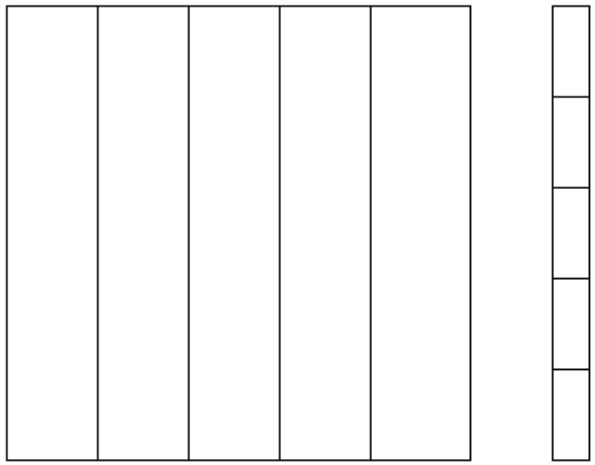
- Reduce function
  - Input: $(i, [m_{i1}v_1, m_{i2}v_2, \ldots, m_{in}v_n])$
  - Output: $(i, m_{i1}v_1 + m_{i2}v_2 + \ldots + m_{in}v_n)$
    - Note that $m_{i1}v_1 + m_{i2}v_2 + \ldots + m_{in}v_n = x_i$



If v Cannot Fit in Main Memory (1/2)

- If $\mathbf{v}$ is so large that it will not fit in its entirety in main memory
  - There will be a very large number of disk accesses to read pieces of $\mathbf{v}$ into memory

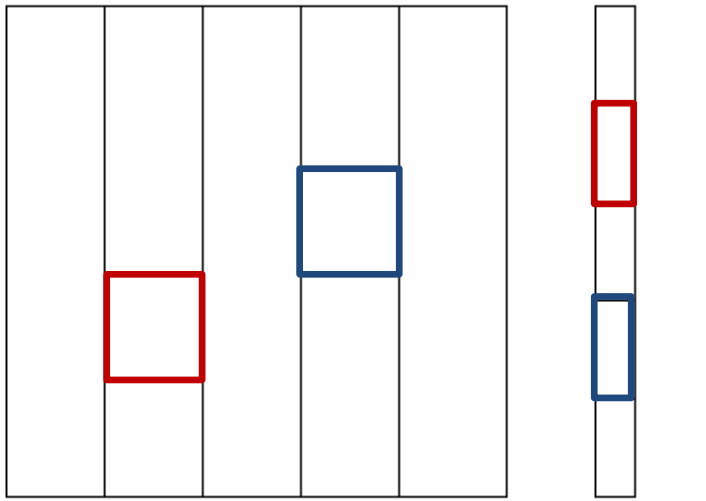
- Solution
  - Divide M into vertical stripes of equal width
  - Divide $\mathbf{v}$ into an equal number of horizontal stripes, of the same height
  - The ith stripe of M multiplies only components from the ith stripe of $\mathbf{v}$



Matrix M Vector $\mathbf{v}$

If v Cannot Fit in Main Memory (2/2)

- Solution (cont'd)



  - Each Map task
    - Assigned a chunk from one of the stripes of M
    - Gets the entire corresponding stripe of $\mathbf{v}$
  - The Map and Reduce tasks can then act exactly as the same as was described before

Relational-Algebra Operations (1/2)

- There are a number of operations on large-scale data that are used in ***database queries***

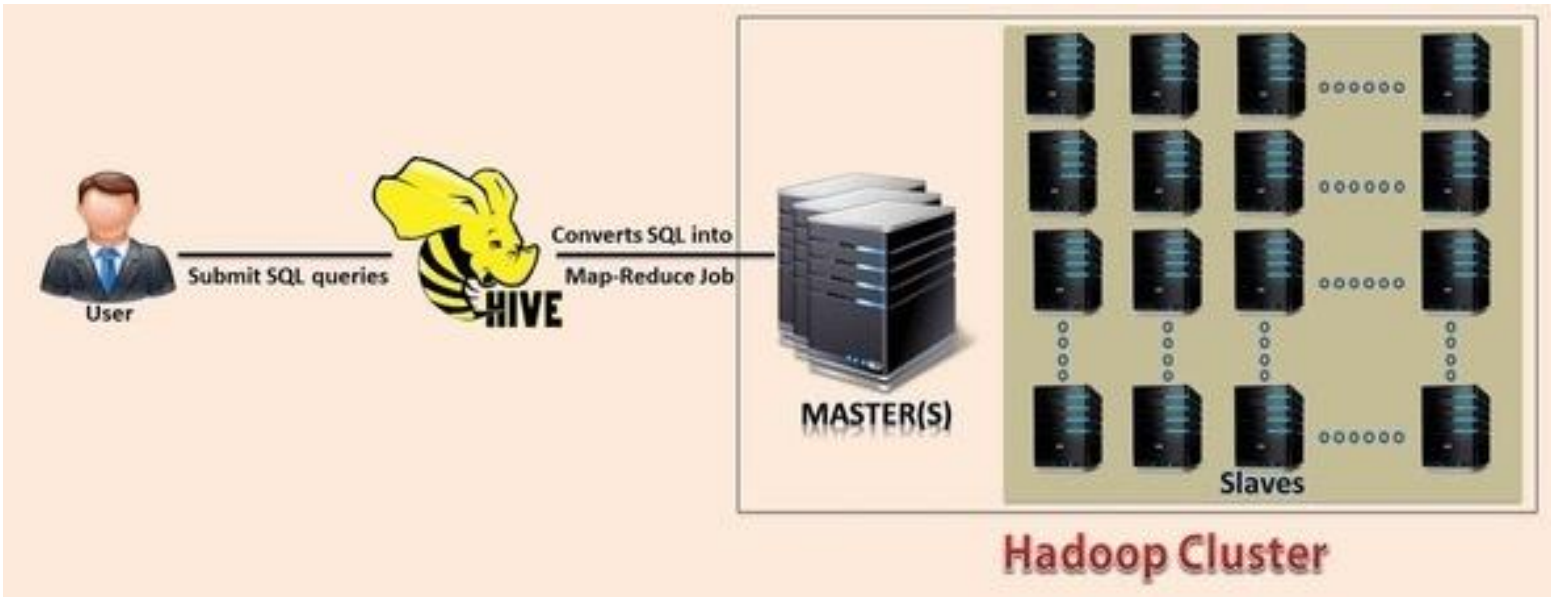
- Definitions
  - ***Relation***: a table with column headers called attributes
  - ***Tuples***: rows of the relations
  - ***Schema***: The set of attributes of a relation (e.g., $R(A_1, A_2, \ldots, A_n)$)

- Example
  - Relation $Links(From, To)$
    - Represents that the first URL has one or more links to the second URL

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

Relational-Algebra Operations (2/2)

- The relational-algebra operations discussed in the class
  - Selection
  - Projection
  - Union, Intersection, and Difference
  - Natural Join
  - Grouping and Aggregation



Selection

- $\sigma_C(R)$
  - Apply a condition C to each tuple in the relation and produce as output only those tuples that satisfy C

- Map function
  - For each tuple t in R, test if it satisfies C
  - If so, produce the key-value pair (t, t)

- Reduce function
  - Simply passes each key-value pair to the output

Projection

- $\pi_S(R)$
  - Produce from each tuple only the components for the attributes in S

- Map function
  - For each tuple t in R, construct a tuple t' by eliminating from t those components whose attributes are not in S
  - Output the key-value pair (t', t')

- Reduce function
  - Turns $(t', [t', t', \ldots, t'])$ into (t', t')

Union

- $R \cup S$
  - Produce the set of all distinct elements in R and S
- Map function
  - Turn each input tuple t into a key-value pair (t, t)
- Reduce function
  - Produce output (t, t)

Intersection

- $R \cap S$
  - Produce the set that contains all elements of R that also belong to S
- Map function
  - Turn each input tuple t into a key-value pair (t, t)
- Reduce function
  - If key t has value list $[t, t]$, then produce (t, t)
  - Otherwise(i.e., if key t has value list $[t]$), produce nothing

Difference

- $R - S$
  - Produce the set of elements in R but not in S

- Map function
  - For a tuple t in R, produce key-value pair (t, R)
  - For a tuple t in S, produce key-value pair (t, S)

- Reduce function
  - For each key t, if the associated value list is $[R]$, then produce (t, t)
  - Otherwise, produce nothing

Natural Join (1/2)

- $R(A, B) \bowtie S(B, C)$
  - For each pair of tuples, one from R and one from S, if the tuples agree on the common attributes, then produce a tuple that combine the tuples
  - Otherwise, produce nothing from this pair of tuples

A	B
a_1	b_1
a_2	b_1
a_3	b_2
a_4	b_3

R

$\bowtie$

B	C
b_2	c_1
b_2	c_2
b_3	c_3

S

=

A	C
a_3	c_1
a_3	c_2
a_4	c_3

Natural Join (2/2)

- Map function
  - For each tuple (a, b) of R, produce the key-value pair $(b, (R, a))$
  - For each tuple (b, c) of S, produce the key-value pair $(b, (S, c))$

- Reduce function
  - For each $(b, [(R, a), (S, c), \ldots])$, construct all pairs consisting of one with (R, a) and the other with (S, c)
  - For each pair $((R, a), (S, c))$, produce (a, b, c)

Grouping and Aggregation

- $\Upsilon_{A,\theta(B)}(R(A,B,C))$
  - Partition the tuples of R according to their values in attribute A
  - Then, for each group, aggregate the values in attribute B using function θ
    - θ: SUM, COUNT, AVG, MIN, MAX, …

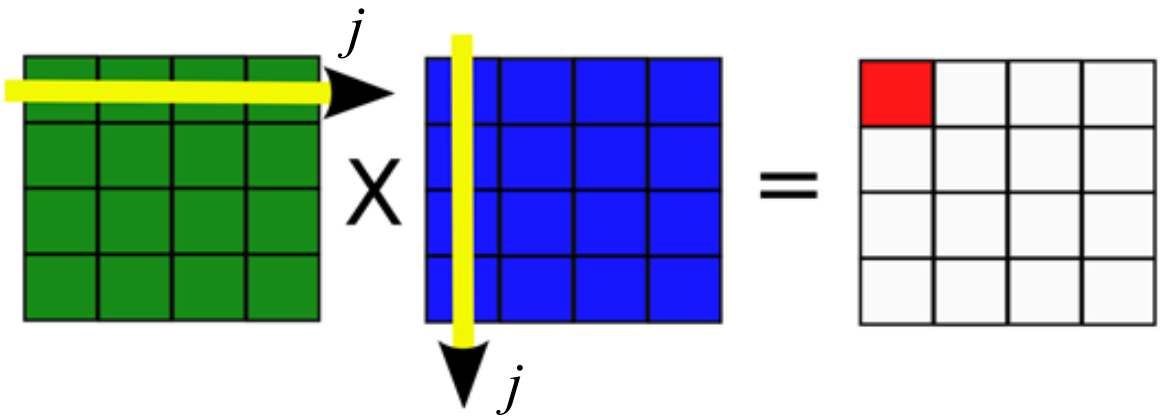
- Map function
  - For each tuple (a, b, c), produce the key-value pair (a, b)

- Reduce function
  - For each $(a, [b_1, b_2, \ldots])$, apply θ to the values $b_1, b_2, \ldots$
    - (ex) if θ is SUM, then output $(a, b_1 + b_2 + \ldots)$

Matrix Multiplication

- Notations
  - M: a matrix with element m_{ij} in row i and column j
  - N: a matrix with element n_{jk} in row j and column k
  - $P = MN$: the matrix with element p_{ik} in row i and column k

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



- Assumptions
  - Each element of M is stored in a file of the DFS as (i, j, m_{ij})
  - Each element of N is stored in a file of the DFS as (j, k, n_{jk})

Matrix Multiplication With 2 MR Jobs (1/2)

- First MapReduce job
  - Goal: Find m_{ij} and n_{jk} with the same j and obtain $m_{ij}n_{jk}$
- Map function
  - For each matrix element m_{ij}, produce the key-value pair $(j, (M, i, m_{ij}))$
  - For each matrix element n_{jk}, produce the key-value pair $(j, (N, k, n_{jk}))$
- Reduce function
  - For each $(j, [(M, i, m_{ij}), (N, k, n_{jk}), \ldots])$
    - For each pair $((M, i, m_{ij}), (N, k, n_{jk}))$, produce $((i, k), m_{ij}n_{jk})$

Matrix Multiplication With 2 MR Jobs (2/2)

- Second MapReduce job
  - Goal: Find $m_{ij}n_{jk}$ with the same i and k and obtain p_{ik}
- Map function
  - For each $((i, k), m_{ij}n_{jk})$, produce exactly this key-value pair, i.e., $((i, k), m_{ij}n_{jk})$
- 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 $v = p_{ik}$

Matrix Multiplication With 1 MR Job

- There often is more than one way to use MR to solve a problem

- 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, …, up to the number of columns of N
  - For each element n_{jk} of N, produce all the key-value pairs $((i, k), (N, j, n_{jk}))$ for i = 1, 2, …, up to the number of columns of M

- Reduce function
  - For each $((i, k), [(M, j, m_{ij}), (N, j, n_{jk}), \ldots])$
    - For each possible value of j, find (M, j, m_{ij}) and (N, j, n_{jk}), and compute $m_{ij}n_{jk}$,
    - Sum these products $v = m_{i1}n_{1k} + m_{i2}n_{2k} + \ldots$ and produce $((i, k), v)$

Measuring the Cost of an MR Algorithm

- We can estimate the ***efficiency*** of a MapReduce algorithm using a cost model

- For many applications, the dominant cost is the ***I/O cost*** rather than the CPU cost
  - (ex) transporting the outputs of Map tasks to their proper Reduce tasks
  - (ex) reading the input files from disk

- Therefore, we focus on the ***I/O cost*** of an algorithm
  - The algorithm executed by each task tend to be very simple
  - The interconnect speed for a cluster is slow compared with the CPU speed
  - The time taken to move the data into main memory may exceed the time needed to operate on the data in memory

The I/O Cost Model

- The I/O cost of a task
  - The size of the input to the task (measured in bytes)
  - ✓ We do not count output size because:
    - The output size of Map tasks will be accounted for when measuring the input size of the receiving Reduce tasks
    - The output size of Reduce tasks is rarely large
- The I/O cost of an algorithm
  - The sum of the I/O cost of all the tasks implementing the algorithm
- We shall use the big-O notation
  - (ex) $O(n)$, $O(n+m)$, ...

(Ex) The I/O Cost of the MR Join Algorithm

- Consider the natural join algorithm for $R \bowtie S$
  - Let $|R|$ and $|S|$ be the sizes of relations R and S, respectively
- The I/O cost for Map tasks: $O(|R| + |S|)$
  - The cost for reading their data from disk
  - Each Map task reads the chunk(s) to which it applies
- The I/O cost for Reduce tasks: $O(|R| + |S|)$
  - The cost for transferring the outputs of Map tasks to Reduce tasks
  - The sum of the outputs of the Map tasks is roughly as large as their inputs
- So, the I/O cost of the join algorithm: $O(|R| + |S|)$

Wall-Clock Time

- While I/O cost often influences our choice of algorithm to use, we must also be aware of the importance of ***wall-clock time***
  - The time it takes a parallel algorithm to finish
- One could minimize total I/O cost by assigning all the work to one task, thereby minimize total communication
  - However, the wall-clock time of such an algorithm would be quite high
- ***If the work is divided fairly among the tasks***, the wall-clock time would be approximately as small as it could be

Another Important Factor: Data Skew

- There is often significant variation in the lengths of the value lists for different keys
  - So different Reduce tasks take different amount of time

- Therefore, keys should be distributed to Reduce tasks carefully
  - The total execution time required by each Reduce task should be as evenly as possible