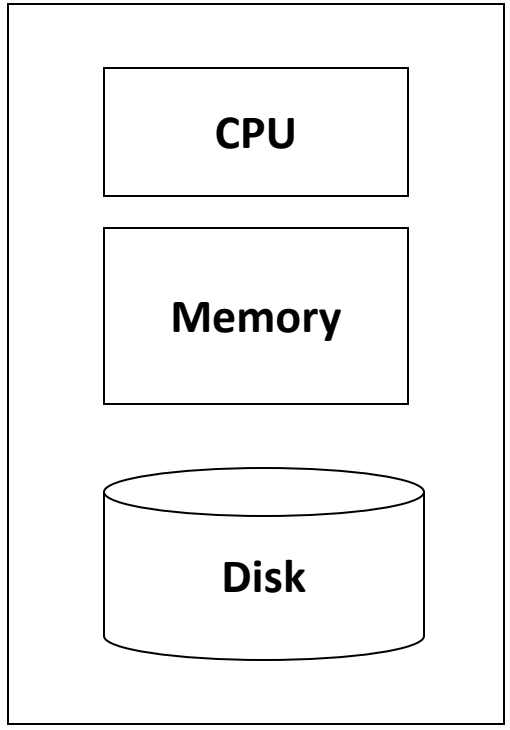
MapReduce

Adopted from slides by

Jure Leskovec, Anand Rajaraman, Jeff Ullman, http://www.mmds.org

Single Node Architecture



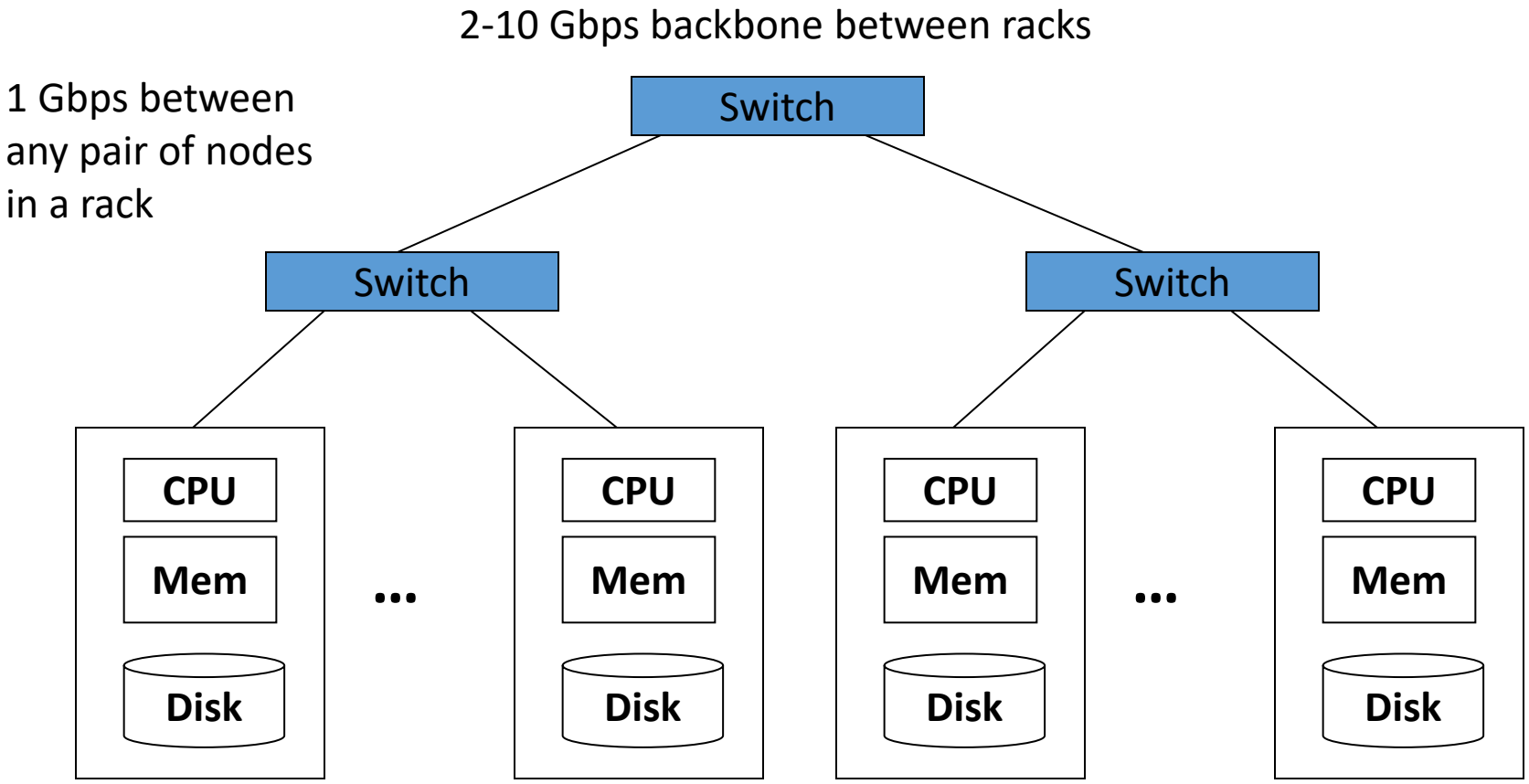
Machine Learning, Statistics

“Classical” Data Mining

Motivation: Google Example

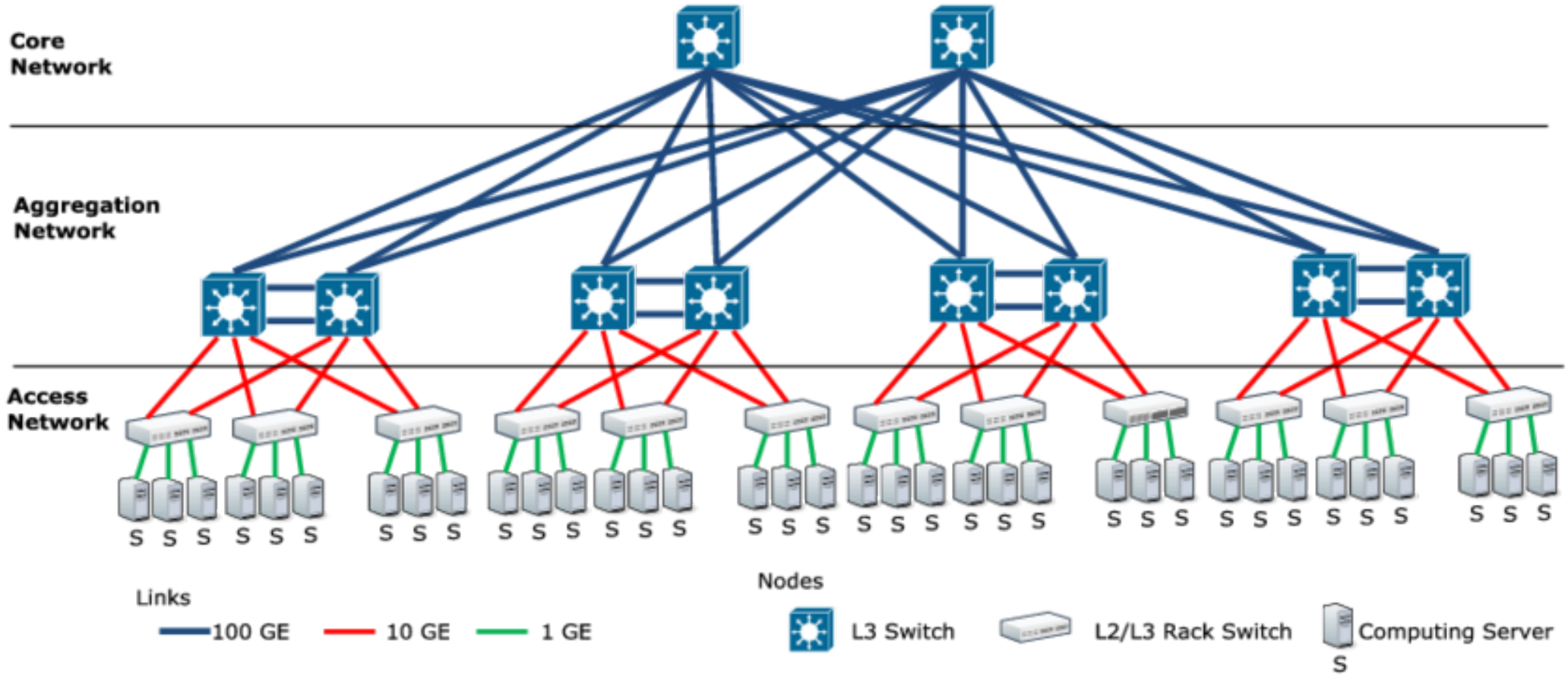
- 20+ billion web pages x 20KB = 400+ TB
- 1 computer reads 30-35 MB/sec from disk
  - ~4 months to read the web
- ~1,000 hard drives to store the web
- Takes even more to **do** something useful with the data!
- **Today, a standard architecture for such problems is emerging:**
  - Cluster of commodity computers
  - Commodity network (ethernet) to connect them

Cluster Architecture



Each rack contains 16-64 nodes

In 2011 it was estimated that Google had 1M machines, http://bit.ly/Shh0RO


Core Network
Aggregation Network
Access Network
S
Links
100 GE
10 GE
1 GE
Nodes
L3 Switch
L2/L3 Rack Switch
Computing Server
S

Large-scale Computing

- **Large-scale computing** for **data mining** problems on **commodity hardware**
- **Challenges:**
  1. **Copying data over a network takes time**
  2. **How can we make it easy to write distributed programs?**
  3. **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!

Idea and Solution

- **Idea:**
  1. Bring computation close to the data
  2. Design an abstract distributed computation model
  3. Store files multiple times for reliability, and also distribute the computation
- **Map-reduce** addresses these problems
  - Google's computational/data manipulation model
  - Elegant way to work with big data
  - **Storage Infrastructure – File system**
    - Google: GFS. Hadoop: HDFS
  - **Programming model**
    - Map-Reduce

Storage Infrastructure

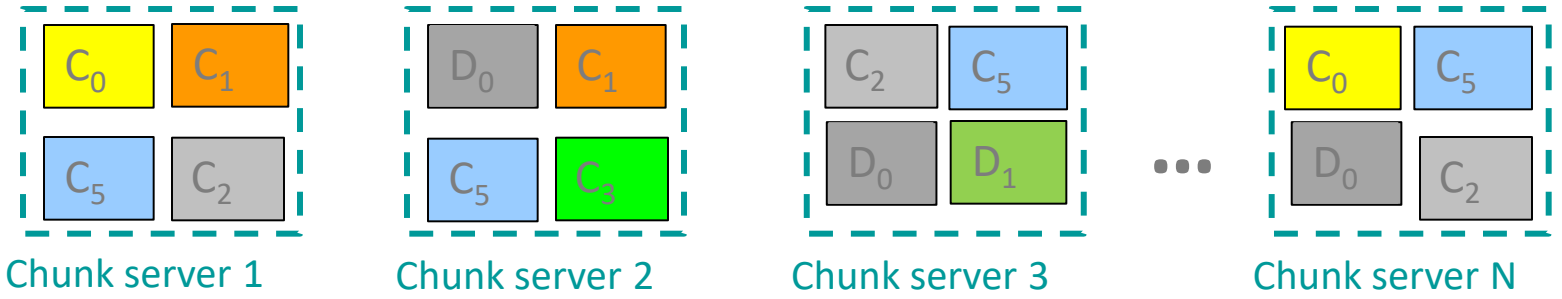
- **Problem:**
  - If nodes fail, how to store data persistently?
- **Answer:**
  - **Distributed File System:**
    - Provides global file namespace
    - Google GFS; Hadoop HDFS;
- **Typical usage pattern**
  - Huge files (100s of GB to TB)
  - Data is rarely updated in place
  - Reads and appends are common

Distributed File System

- **Chunk servers**
  - File is split into contiguous chunks
  - Typically each chunk is 16-64MB
  - 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

Distributed File System

- **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

Programming Model: MapReduce

Warm-up task:

- We have a huge text document
- Count the number of times each distinct word appears in the file
- **Sample application:**
  - Analyze web server logs to find popular URLs

Task: Word Count

Case 1:

- File too large for memory, but all <word, count> pairs fit in memory

Case 2:

- Even <word, count> pairs do not fit in memory

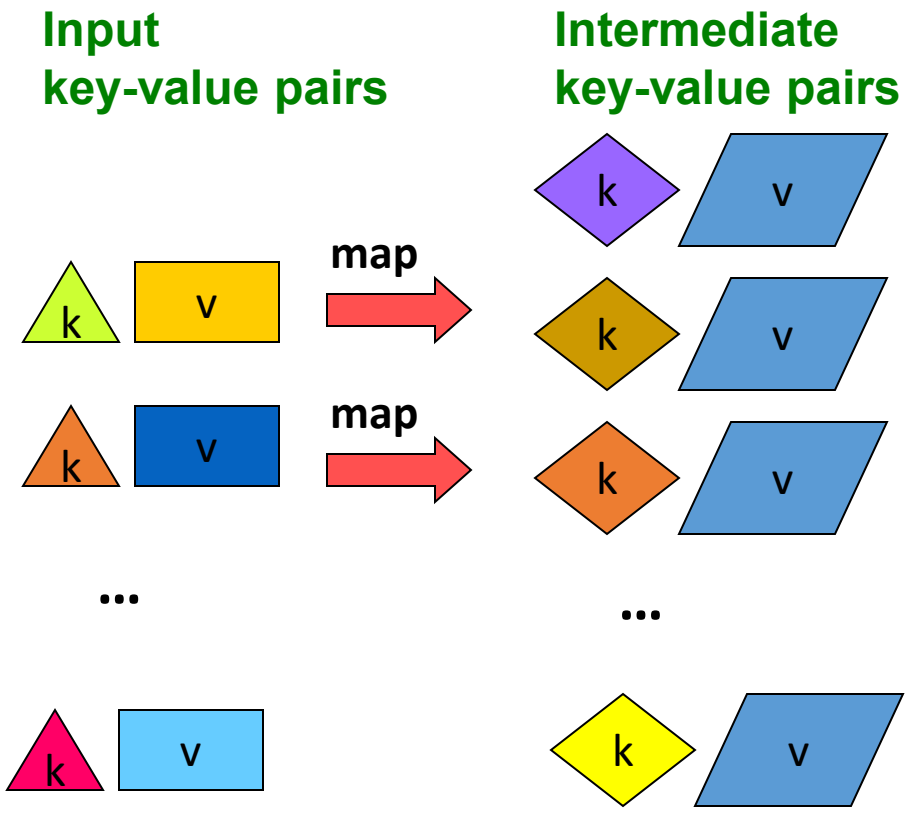
- Case 2 captures the essence of **MapReduce**
  - Great thing is that it is naturally parallelizable

MapReduce: Overview

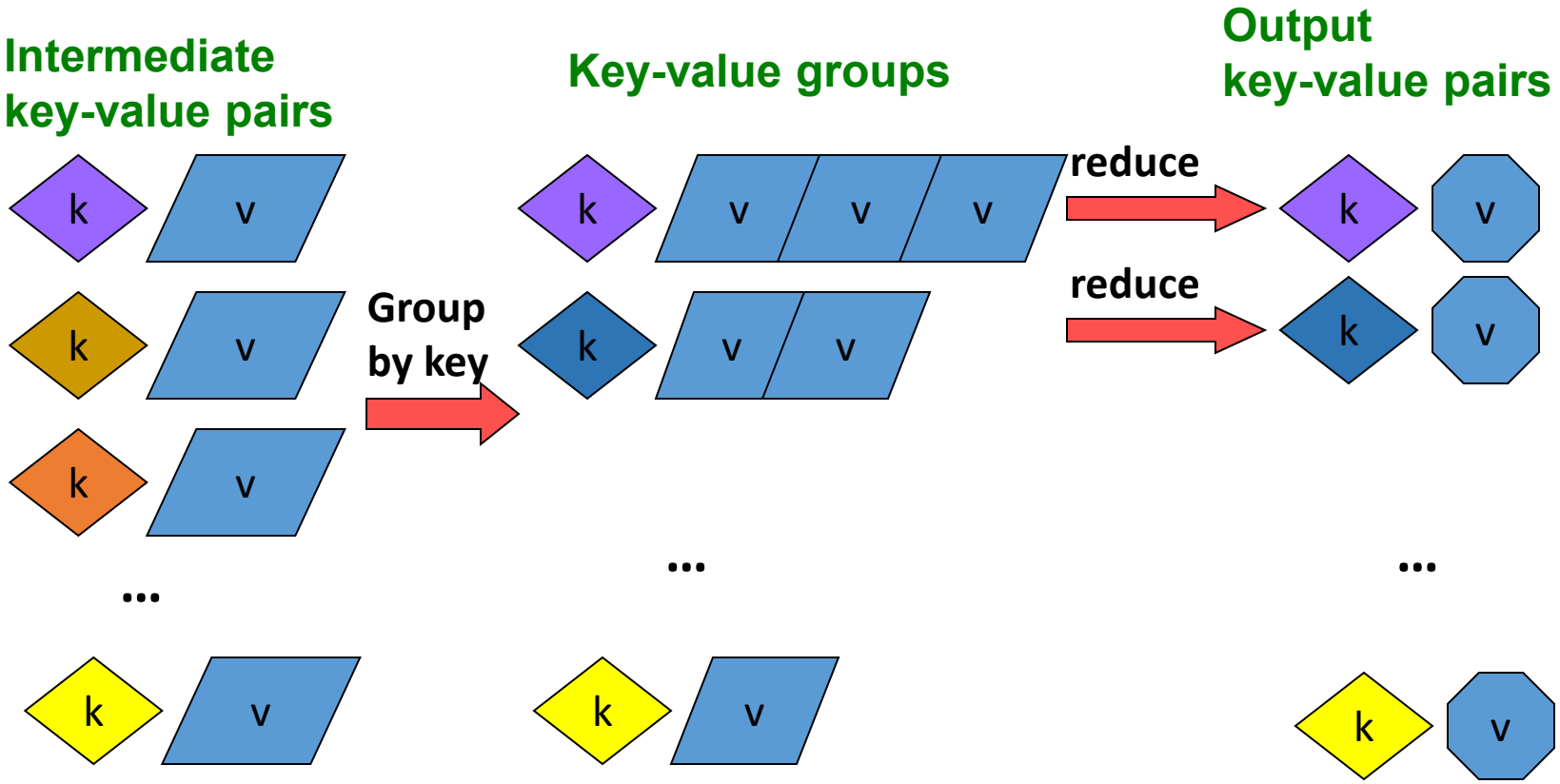
- Sequentially read a lot of data
- **Map:**
  - Extract something you care about
- **Group by key:** Sort and Shuffle
- **Reduce:**
  - Aggregate, summarize, filter or transform
- Write the result

Outline stays the same, **Map** and **Reduce** change to fit the problem

MapReduce: The Map Step



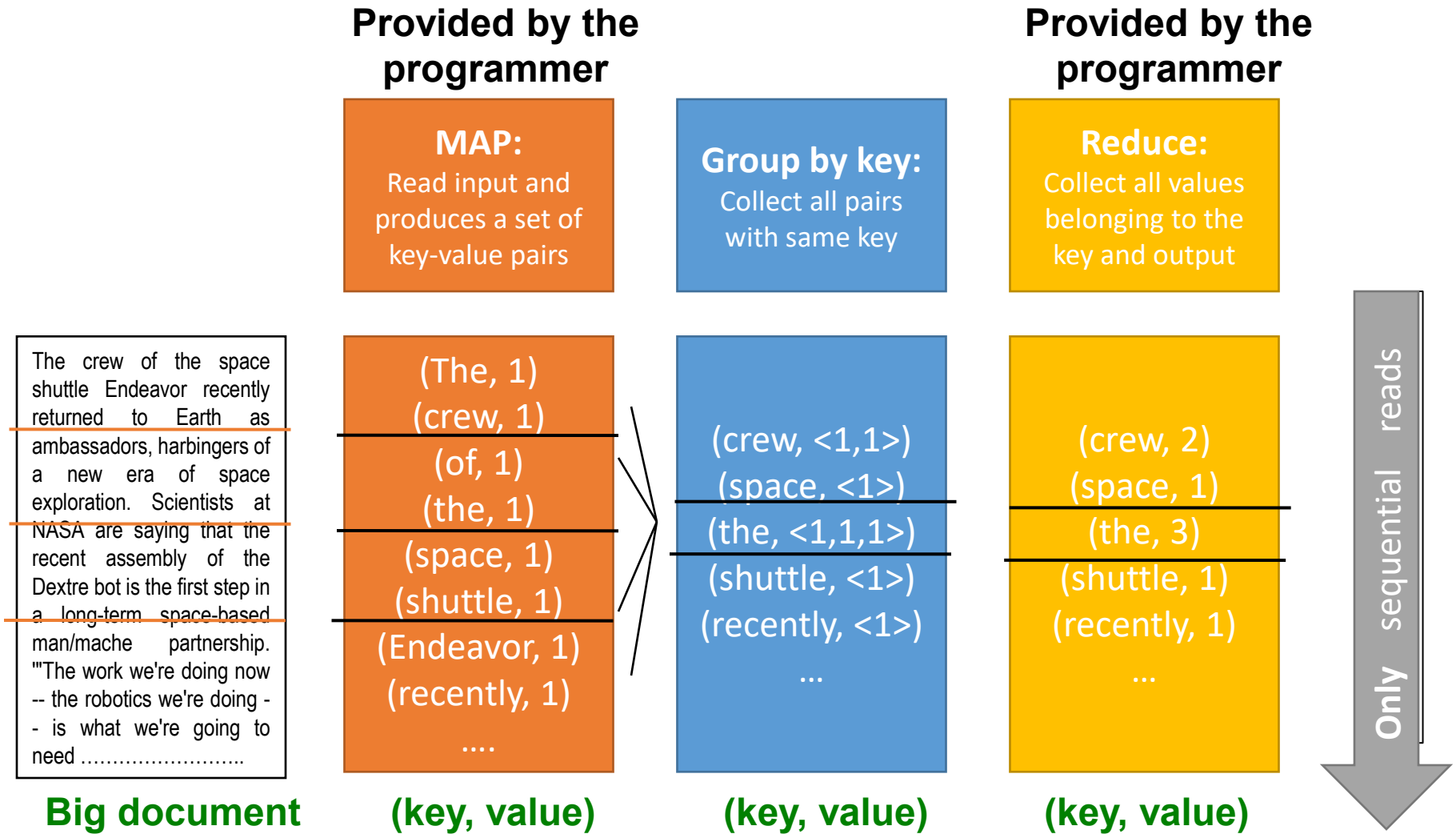
MapReduce: The Reduce Step



More Specifically

- **Input:** a set of key-value pairs
- Programmer specifies two methods:
  - **Map(k, v)** $\rightarrow$ <k', v'>*
    - Takes a key-value pair and outputs a set of key-value pairs
      - E.g., key is the filename, value is a single line in the file
    - There is one Map call for every *(k,v)* pair
  - **Reduce(k', <v'>*)** $\rightarrow$ <k'', v''>*
    - **All values *v'* with same key *k'* are reduced together and processed in *v'* order**
    - There is one Reduce function call per unique key *k'*

MapReduce: Word Counting

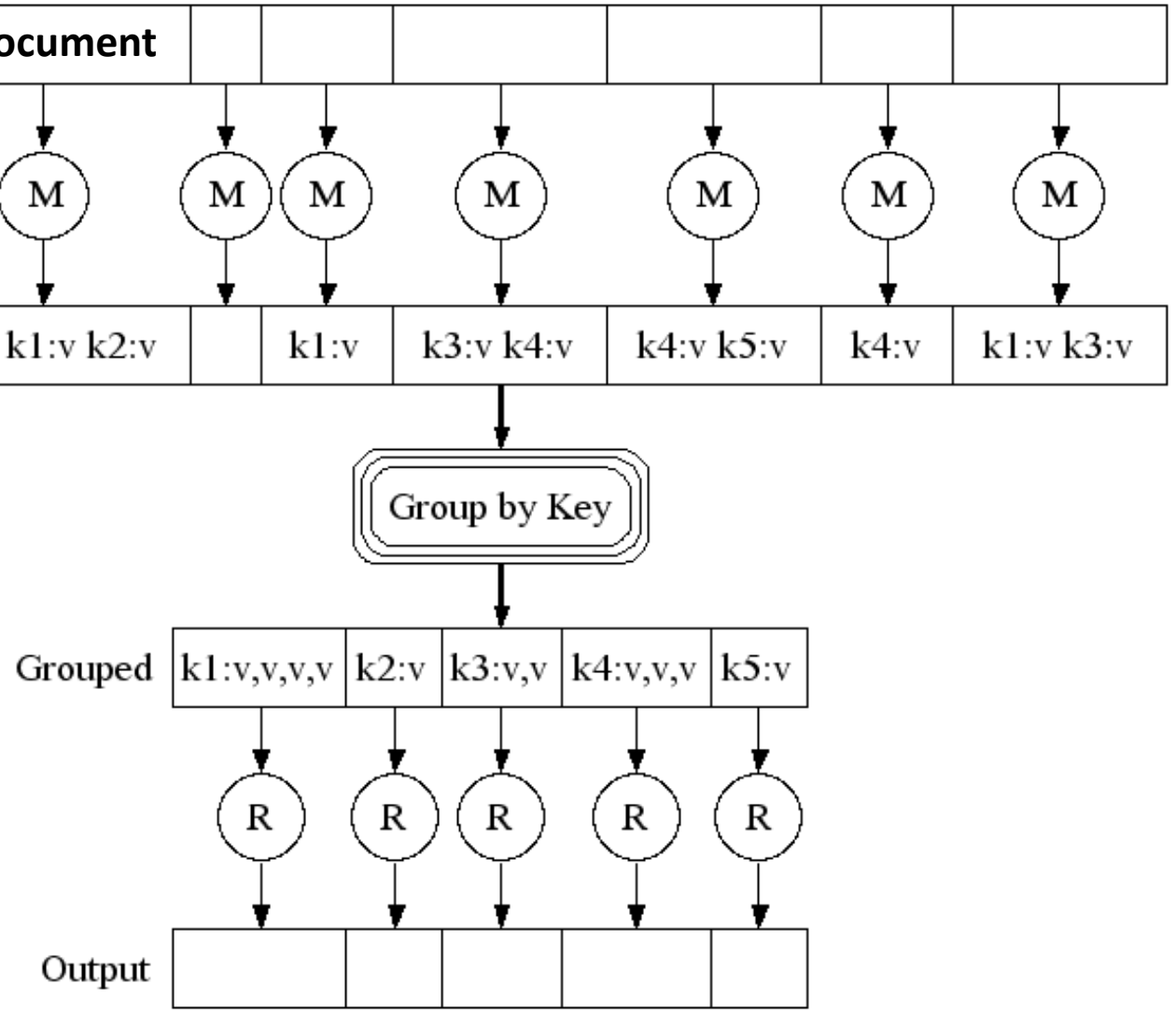


Map-Reduce: A diagram

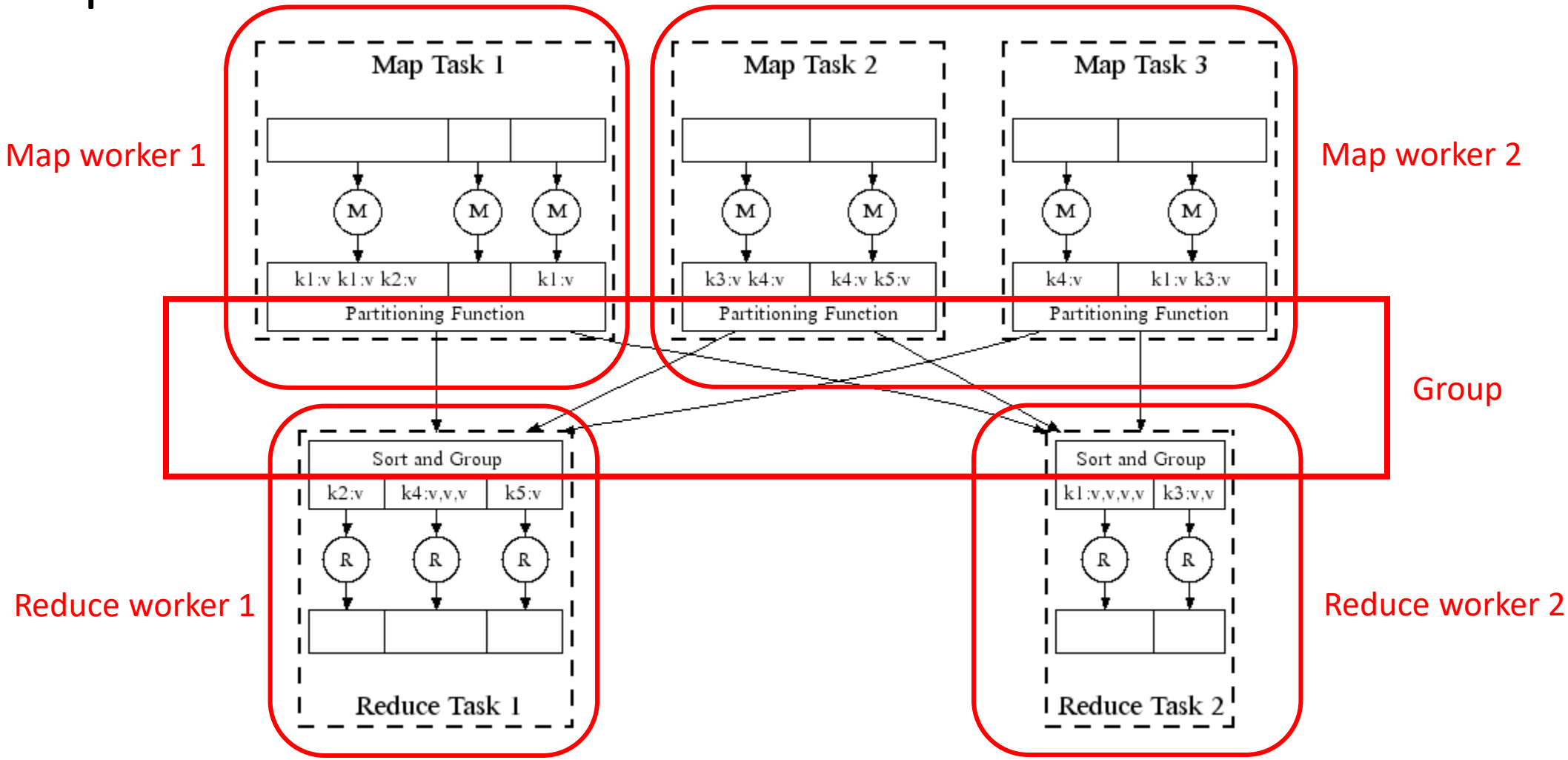
MAP:
Read input and produces a set of key-value pairs

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

Reduce:
Collect all values belonging to the key and output



Map-Reduce: In Parallel



All phases are distributed with many tasks doing the work

Word Count Using MapReduce

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

Example: Host size

- **Suppose we have a large web corpus**
- Look at the metadata file
  - Lines of the form: (URL, size, date, …)
- **For each host, find the total number of bytes**
  - That is, the sum of the page sizes for all URLs from that particular host

- **Map:**
  - Extract (hostname(URL), size) for each line
- **Reduce:**
  - Sum the sizes

Example: Language Model

- **Statistical machine translation:**
  - Need to count number of times every 5-word sequence occurs in a large corpus of documents

- **Very easy with MapReduce:**
  - **Map:**
    - Extract (5-word sequence, count) from document
  - **Reduce:**
    - Combine the counts

Map-Reduce: Environment

Map-Reduce environment takes care of:

- Partitioning the input data
- Scheduling the program's execution across a set of machines
- Performing the **group by key** step
- Handling machine failures
- Managing required inter-machine communication

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
- **Intermediate results are stored on local FS of Map and Reduce workers**
- **Output is often input to another MapReduce task**

Coordination: Master

- **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 info to reducers
- Master pings workers periodically to detect failures

Dealing with Failures

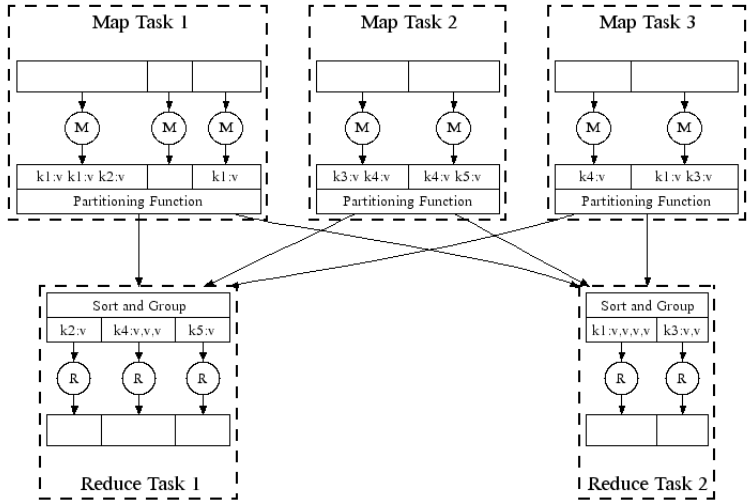
- **Map worker failure**
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- **Reduce worker failure**
  - Only in-progress tasks are reset to idle
  - Reduce task is restarted
- **Master failure**
  - MapReduce task is aborted and client is notified

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
  - 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

Refinement: Combiners

- Often a Map task will produce many pairs of the form $(k,v_1), (k,v_2), \ldots$ for the same key k
  - E.g., popular words in the word count example
- **Can save network time by pre-aggregating values in the mapper:**
  - combine(k, list(v_1)) → v_2
  - Combiner is usually same as the reduce function
- Works only if reduce function is commutative and associative

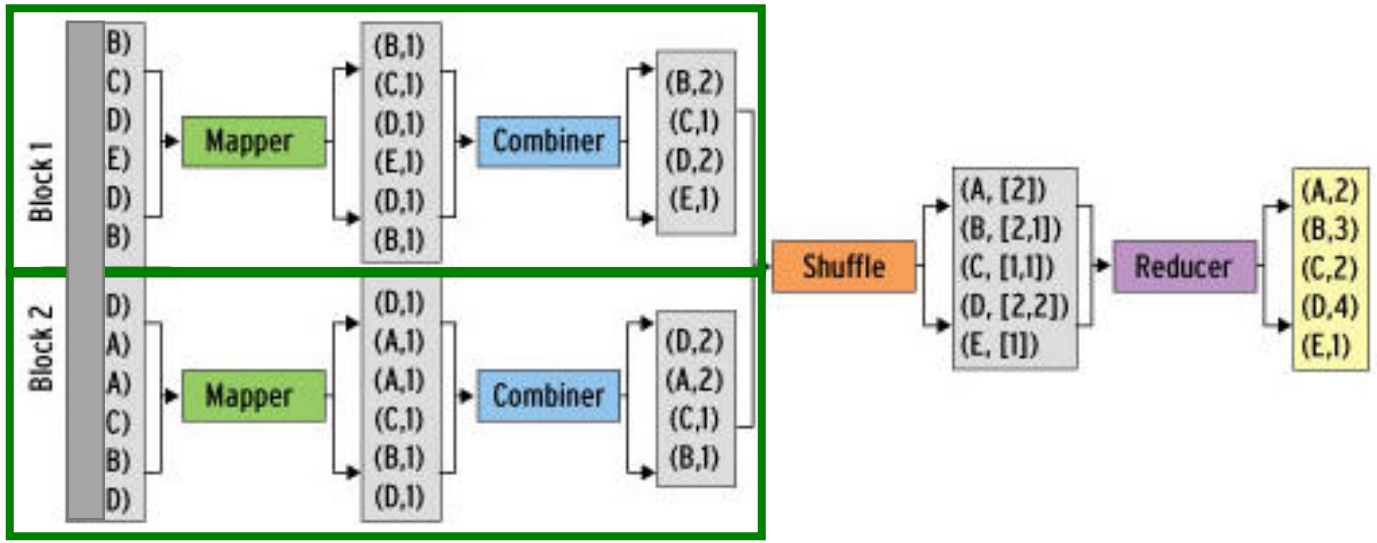


Refinement: Combiners

- Combiner trick works only if reduce function is
  - Commutative: $a + b = b + a$
  - Associative $(a + b) + c = a + (b + c)$
- Sum
- Average
- Median

Refinement: Combiners

- **Back to our word counting example:**
  - Combiner combines the values of all keys of a single mapper (single machine):



  - Much less data needs to be copied and shuffled!

Refinement: Partition Function

- **Want to control how keys get 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

Implementations

- Google
  - Not available outside Google
- Hadoop
  - An open-source implementation in Java
  - Uses HDFS for stable storage
  - Download: http://lucene.apache.org/hadoop/
- Spark
  - An open-source implementation in Scala
  - Supports Scala, Java, Python, and R
- Hive, Pig
  - Provide SQL-like abstractions on top of Hadoop MapReduce layer

Cloud Computing

- Ability to rent computing by the hour
  - Additional services e.g., persistent storage

- Amazon's "Elastic Compute Cloud" (EC2)
  - S3 (stable storage)
  - Elastic MapReduce (EMR)

How to design map-reduce programs

- What data are distributed?
- What computations are included in the program?
  - What computations can be conducted locally (so they can be put in map)?
  - What computations cannot be conducted locally (so they should be put in reduce)?
  - How do we aggregate computations (can be used to determine the key for intermediate key-value pairs)?

Matrix-Vector Multiplication by MapReduce

- $\boldsymbol{M} = \{m_{i,j}\}$ is a $n \times n$ matrix
- $\boldsymbol{v} = \{v_j\}$ is a vector of length n
- Compute $\boldsymbol{x} = \boldsymbol{M} \cdot \boldsymbol{v}$ where

$$x_i = \sum_{j=1}^{n} m_{i,j} \cdot v_j$$

- $\boldsymbol{M}$ cannot fit in main memory
- Assume $\boldsymbol{v}$ fits in main memory

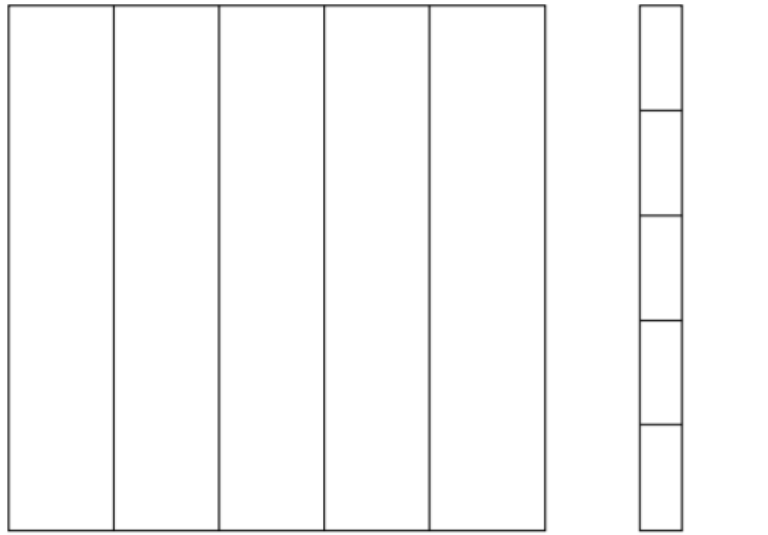
Matrix-Vector Multiplication by MapReduce

- Read $m_{i,j}$ and $\boldsymbol{v}$ into main memory
- **Map Function:**
  - For each element $m_{i,j}$, a mapper produces a key-value pair $(i, m_{i,j} \cdot v_j)$
- **Group:**
  - Each reducer receives $(i, < m_{i,1}v_1, m_{i,2}v_2, \cdots, m_{i,n}v_n >)$
- **Reduce Function:**
  - Sum all values and output $(i, x_i = \sum_{j=1}^{n} m_{i,j} \cdot v_j)$

Matrix-Vector Multiplication by MapReduce

- If $\boldsymbol{v}$ cannot fit in main memory
- Cause large number of disk accesses if move pieces of vector into main memory repeatedly

- Divide matrix into vertical stripes and divide the vector into an equal number of horizontal stripes
- Alternatively, divide matrix into squares



Relational-Algebra

- Basic SQL-like operations which can be used to describe more complex operations
- Relation $R(A_1, A_2, \cdots, A_n)$
- Relational algebra
  - Selection
  - Projection
  - Union, Insertion, Difference
  - Natural Join
  - Grouping and Aggregation

Difference by MapReduce

- Compute difference $R - S$
  - The tuples appear in R not in S

- **Map Function:**
  - For each tuple t in R, produce a key-value pair (t, R)
  - For each tuple t in S, produce a key-value pair (t, S)
- **Group:**
  - Each reducer receives $(t, < R >)$ or $(t, < S >)$ or $(t, < R, S >)$
- **Reduce Function:**
  - If receive $(t, < R >)$, produce (t, t); otherwise, produce nothing

Natural Join By MapReduce

- Compute the natural join $R(A, B) \bowtie S(B, C)$
- R and S are each stored in files
- Tuples are pairs (a, b) or (b, c)

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 By MapReduce

- **Map Function:**
  - For each tuple $R(a, b)$, produce a key-value pair $(b, (R, a))$
  - For each tuple $S(b, c)$, produce a key-value pair $(b, (S, c))$
- **Group:**
  - Each reducer receives $(b, < \cdots, (R, a), \cdots, (S, c), \cdots >)$
- **Reduce Function:**
  - For each pair of (R, a) and (S, c), output (a, c)

Grouping and Aggregation by MapReduce

- For relation $R(A, B, C)$, compute $\gamma_{A,\theta(B)}(R)$ where A is the grouping attributes and $\theta(B)$ is the aggregation
- Map performs the grouping, while Reduce performs the aggregation
- **Map Function:**
  - For each tuple (a, b, c), produce a key-value pair (a, b)
- **Group:**
  - Each reducer receives $(a, < b_1, b_2, \cdots >)$
- **Reduce Function:**
  - Compute $\theta(b_1, b_2, \cdots)$ and output (a, θ)

Matrix Multiplication by MapReduce

- $\boldsymbol{M} = \{m_{i,j}\}$ is a $|I| \times |J|$ matrix
- $\boldsymbol{N} = \{n_{j,k}\}$ is a $|J| \times |K|$ matrix
- Compute $\boldsymbol{P} = \boldsymbol{M} \cdot \boldsymbol{N}$ where

$$p_{i,k} = \sum_{j=1}^{n} m_{i,j} \cdot n_{j,k}$$

Matrix Multiplication by MapReduce

- View $\boldsymbol{M}$ as a relation $M(I, J, V)$ with tuples $(i, j, m_{i,j})$
- View $\boldsymbol{N}$ as a relation $N(J, K, W)$ with tuples $(j, k, n_{j,k})$

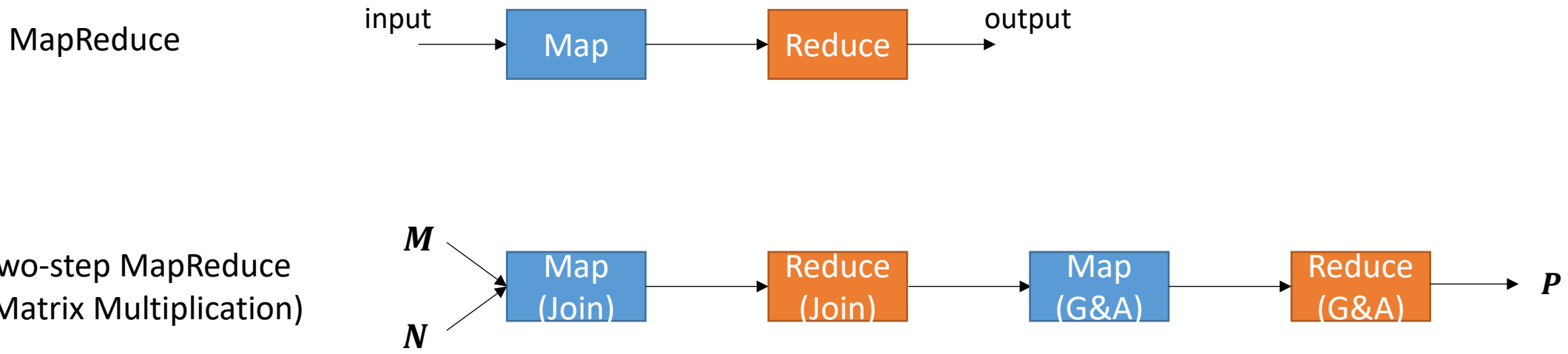
- Step 1: compute natural join $P(I, K, VW) = M(I, J, V) \bowtie N(J, K, W)$
- Step 2: compute grouping and aggregation for $P(I, K, VW)$ with I and K as grouping attributes and $\mathrm{sum}(VW)$ as the aggregation

Matrix Multiplication by MapReduce

- Step 1: natural join
  - **Map Function:** for each element $m_{i,j}$, produce key-value pair $(j, (\boldsymbol{M}, i, m_{i,j}))$; for each element $n_{j,k}$, produce key-value pair $(j, (\boldsymbol{N}, k, n_{j,k}))$
  - **Group:** each reducer receives $(j, < \cdots (\boldsymbol{M}, i, m_{i,j}), \cdots (\boldsymbol{N}, k, n_{j,k}), \cdots >)$
  - **Reduce Function:** for every pair of $(\boldsymbol{M}, i, m_{i,j})$ and $(\boldsymbol{N}, k, n_{j,k})$, produce key-value pair $((i, k), m_{i,j} n_{j,k})$
- Step 2: grouping and aggregation
  - **Map Function:** for each key-value pair $\left((i, k), m_{i,j} n_{j,k}\right)$, produce the exact pair
  - **Group:** each reducer receives $\left((i, k), < \cdots, m_{i,j} n_{j,k}, \cdots >\right)$
  - **Reduce Function:** compute $p_{i,k} = \sum_j m_{i,j} n_{j,k}$ and output $\left((i, k), p_{i,k}\right)$

Workflow of MapReduce

- The workflow among tasks can be represented by an acyclic flow graph $a \rightarrow b$, which represents that task a's output is input to task b



Cost Measures for Algorithms

- Computation cost
  - Asymptotic complexity, a function of input size, denoted as Big-O
  - The (average) complexity of bubble-sort is:
  - The (average) complexity of merge-sort is:

Cost Measures for Algorithms

- **In MapReduce we quantify the cost of an algorithm using**

1. *Communication cost* = total input size of all tasks
2. *Elapsed communication cost* = max of total input size of tasks along any path in flow graph
3. (*Elapsed*) *computation cost* = max of total running time of tasks along any path in flow graph

Space complexity is somewhat traded with communication cost

Communication Cost usually Dominates

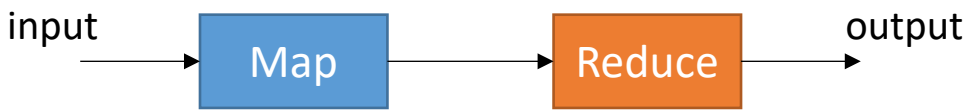
- The algorithm executed by each task tends to be very simple, often linear in the input size
- The interconnect speed for a computing cluster is much slower than the speed of execution by a processor
- The time taken to move data into main memory is also usually larger than the time of execution

Wall-Clock Time

- Or elapsed real time, considers both communication time and computation time (in parallel)
- Communication cost dominates only when computation workload is fairly distributed to multiple tasks (computing nodes)
  - Assigning all work to one task can minimize communication cost, but dramatically increase computation time

Measure for Communication Cost

- Communication cost counts only input size because:
  - If the output of one task is input to another task, then the output will be accounted for when measuring the input of the receiving task. There is no reason to count it twice
  - So we only miss the final output. In practice, the final output is rarely large in order to be interpretable



Example: Word Count

Word Count
Map Function:
For each word w, generate $(w, 1)$
Group:
Each reducer receives $(w, < 1, \cdots, 1 >)$
Reduce Function:
Compute the sum and output (w, sum)

Comm. cost: n n

Total comm. cost: $O(n)$

Example: Matrix Multiplication

- Compute $\boldsymbol{P} = \boldsymbol{M} \cdot \boldsymbol{N}$ where

$$p_{i,k} = \sum_{j=1}^{n} m_{i,j} \cdot n_{j,k}$$

- View $\boldsymbol{M}$ as a relation $M(I, J, V)$ with tuples $(i, j, m_{i,j})$
- View $\boldsymbol{N}$ as a relation $N(J, K, W)$ with tuples $(j, k, n_{j,k})$

- Step 1: compute natural join $P(I, K, VW) = M(I, J, V) \bowtie N(J, K, W)$
- Step 2: compute grouping and aggregation for $P(I, K, VW)$ with I and K as grouping attributes and $\mathrm{sum}(VW)$ as the aggregation

Two-Step Method

Step 1

Map Function:

For each element $m_{i,j}$, produce key-value pair $(j, (\boldsymbol{M}, i, m_{i,j}))$; for each element $n_{j,k}$, produce key-value pair $(j, (\boldsymbol{N}, k, n_{j,k}))$

Group:

Each reducer receives $(j, < \cdots (\boldsymbol{M}, i, m_{i,j}), \cdots (\boldsymbol{N}, k, n_{j,k}), \cdots >)$

Reduce Function:

For every pair of $(\boldsymbol{M}, i, m_{i,j})$ and $(\boldsymbol{N}, k, n_{j,k})$, produce key-value pair $((i,k), m_{i,j}n_{j,k})$

Step 2

Map Function:

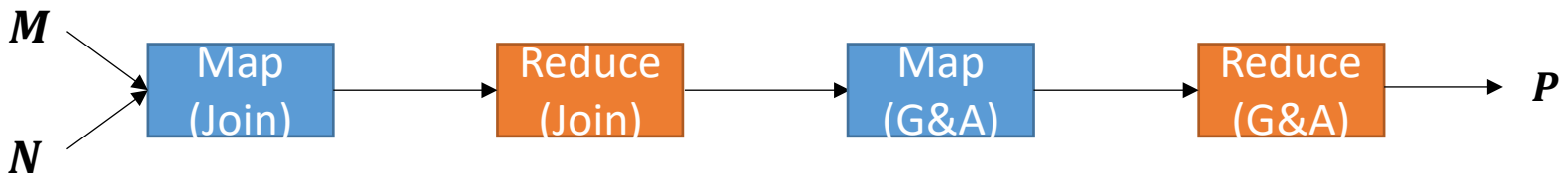
For each key-value pair $((i,k), m_{i,j}n_{j,k})$, produce the exact pair

Group:

Each reducer receives $((i,k), < \cdots, m_{i,j}n_{j,k}, \cdots >)$

Reduce Function:

Compute $p_{i,k} = \sum_j m_{i,j}n_{j,k}$ and output $((i,k), p_{i,k})$



Comm. cost $|I||J| + |J||K|$ $|I||J| + |J||K|$ $|I||J||K|$ $|I||J||K|$

Total comm. cost: $O(|I||J| + |J||K| + |I||J||K|)$

Example: Multiway Joins

- Compute the natural join of three relations

$$R(A,B) \bowtie S(B,C) \bowtie T(C,D)$$

- Two-step method: first compute the join of two relations and then compute the third join
- One-step method: directly compute the 3-way join

- Assuming R, S and T have sizes r, s and t
  - For simplicity, assume that the probability of R-tuple and S-tuple agree on B, and the probability of S-tuple and T-tuple agree on C, equal to p

Two-Step Method

- Step 1: compute $R(A,B) \bowtie S(B,C) = Z(A,C)$
- Step 2: compute $Z(A,C) \bowtie T(C,D)$

Natural Join

Map Function:

For each tuple $R(a,b)$, produce a key-value pair $(b,(R,a))$

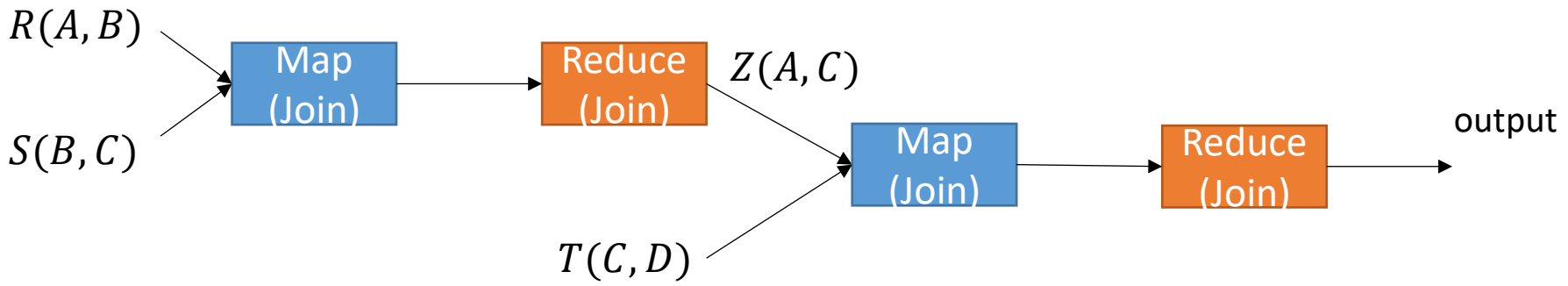
For each tuple $S(b,c)$, produce a key-value pair $(b,(S,c))$

Group:

Each reducer receives $(b, <\cdots,(R,a),\cdots,(S,c),\cdots>)$

Reduce Function:

For each pair of (R,a) and (S,c), output (a,c)



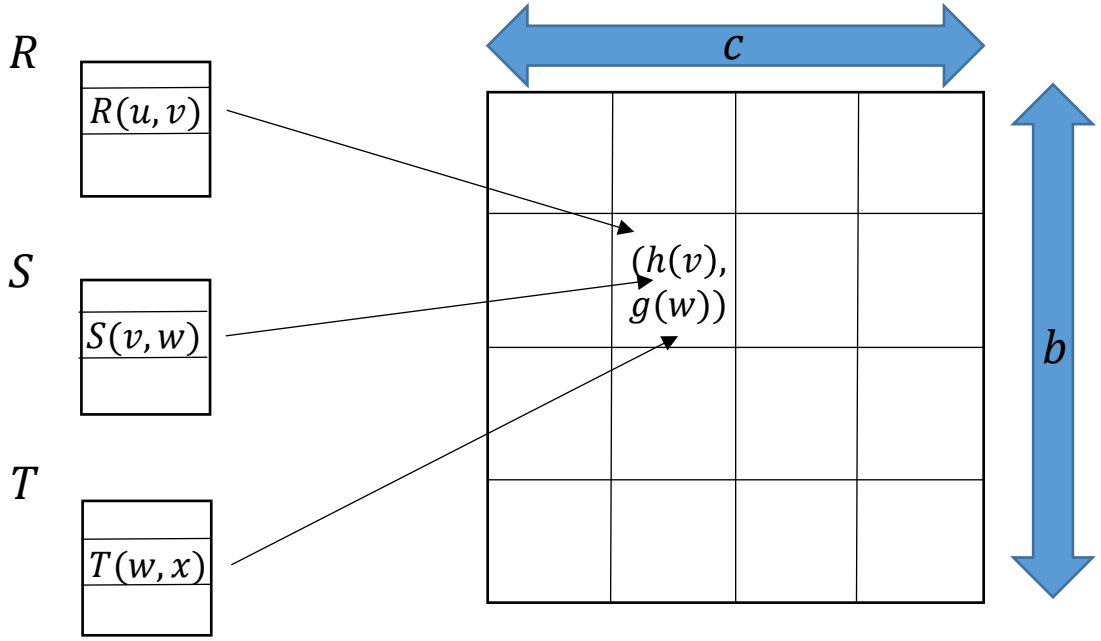
Comm. cost: $r+s$ $r+s$ $t+prs$ $t+prs$

Total comm. cost: $O(r+s+t+prs)$

One-Step Method

$R(A,B) \bowtie S(B,C) \bowtie T(C,D)$

- Use a hash function h to hash B-values into b buckets
- Use a hash function g to hash C-values into c buckets
- A reduce task corresponding to bucket pair (i,j) is responsible for joining tuples $R(u,v), S(v,w), T(w,x)$ where $h(v) = i$ and $g(w) = j$



One-Step Method

$R(A,B) \bowtie S(B,C) \bowtie T(C,D)$

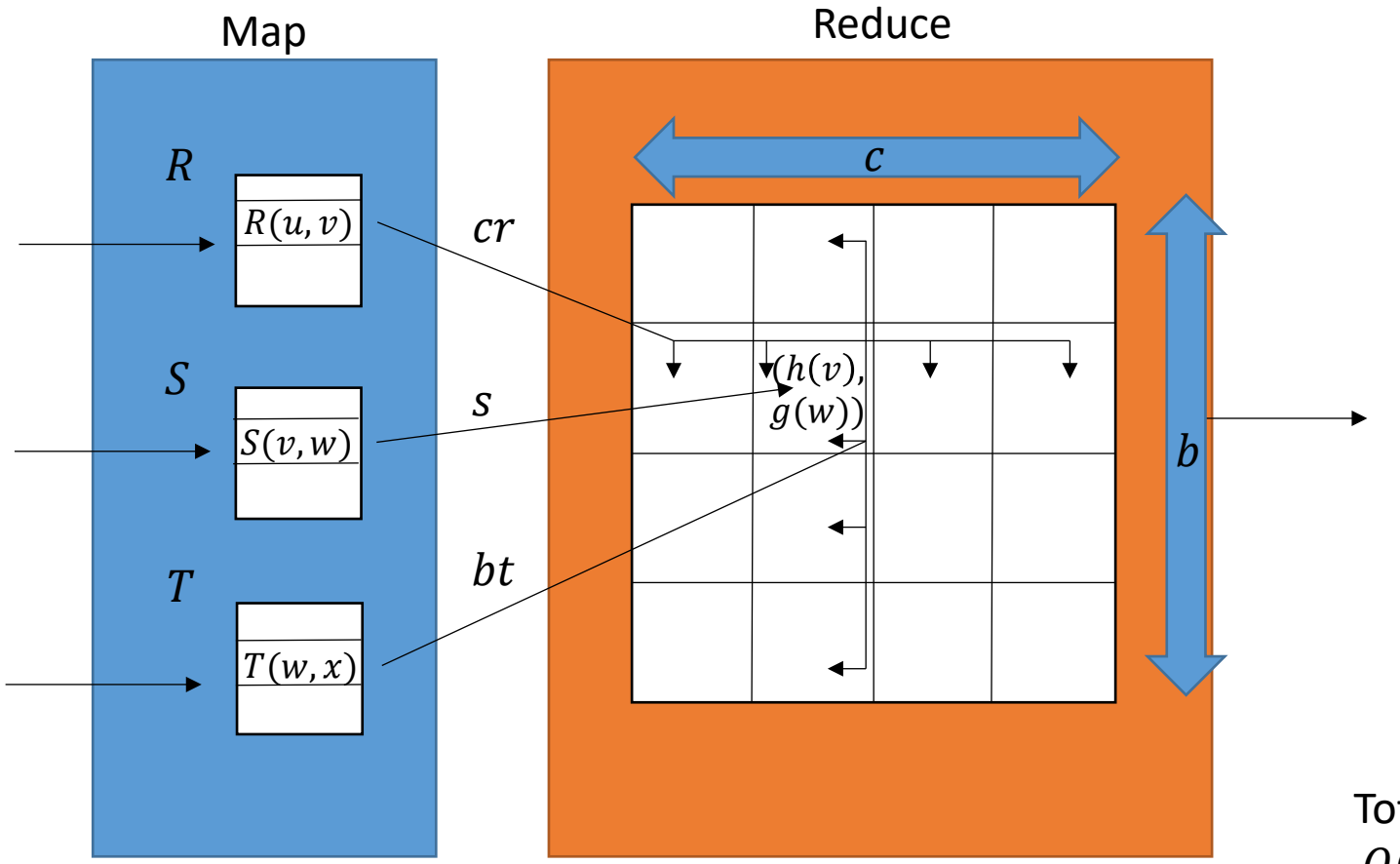
- **Map Function:**
  - For each tuple $R(u,v)$, produce kv-pairs $\big((h(v),j),(R,u,v)\big), j = 1,\cdots,c$
  - For each tuple $S(v,w)$, produce a kv-pair $\Big(\big(h(v),g(w)\big),(S,v,w)\Big)$
  - For each tuple $T(w,x)$, produce kv-pairs $\Big(\big(i,g(w)\big),(T,w,x)\Big), i = 1,\cdots,b$
- **Group:**
  - Each reducer receives $\big((i,j),<\cdots,(R,u,v),\cdots,(S,v,w),\cdots(T,w,x),\cdots>\big)$
- **Reduce Function:**
  - If receives at least one (S,v,w), for each pair of (R,u,v) and (T,w,x), output (u,x)

$R(u,v)$ if $h(v)=i$

$T(w,x)$ if $g(w)=j$

One-Step Method

$R(A,B) \bowtie S(B,C) \bowtie T(C,D)$



Comm. cost $r + t + s$ $cr + s + bt$

Total comm. cost:
$O(r + 2s + t + cr + bt)$

One-Step Method

- Total communication cost:

$$O(r + 2s + t + cr + bt)$$

$$cb = k$$

- Minimum total communication cost for a given k:

$$O(r + 2s + t + 2\sqrt{krt})$$

vs.

$$O(r + s + t + prt)$$

Summary

- Cluster Computing
- Distributed File Systems
- MapReduce
- The Map Function
- The Reduce Function
- Communication-Cost