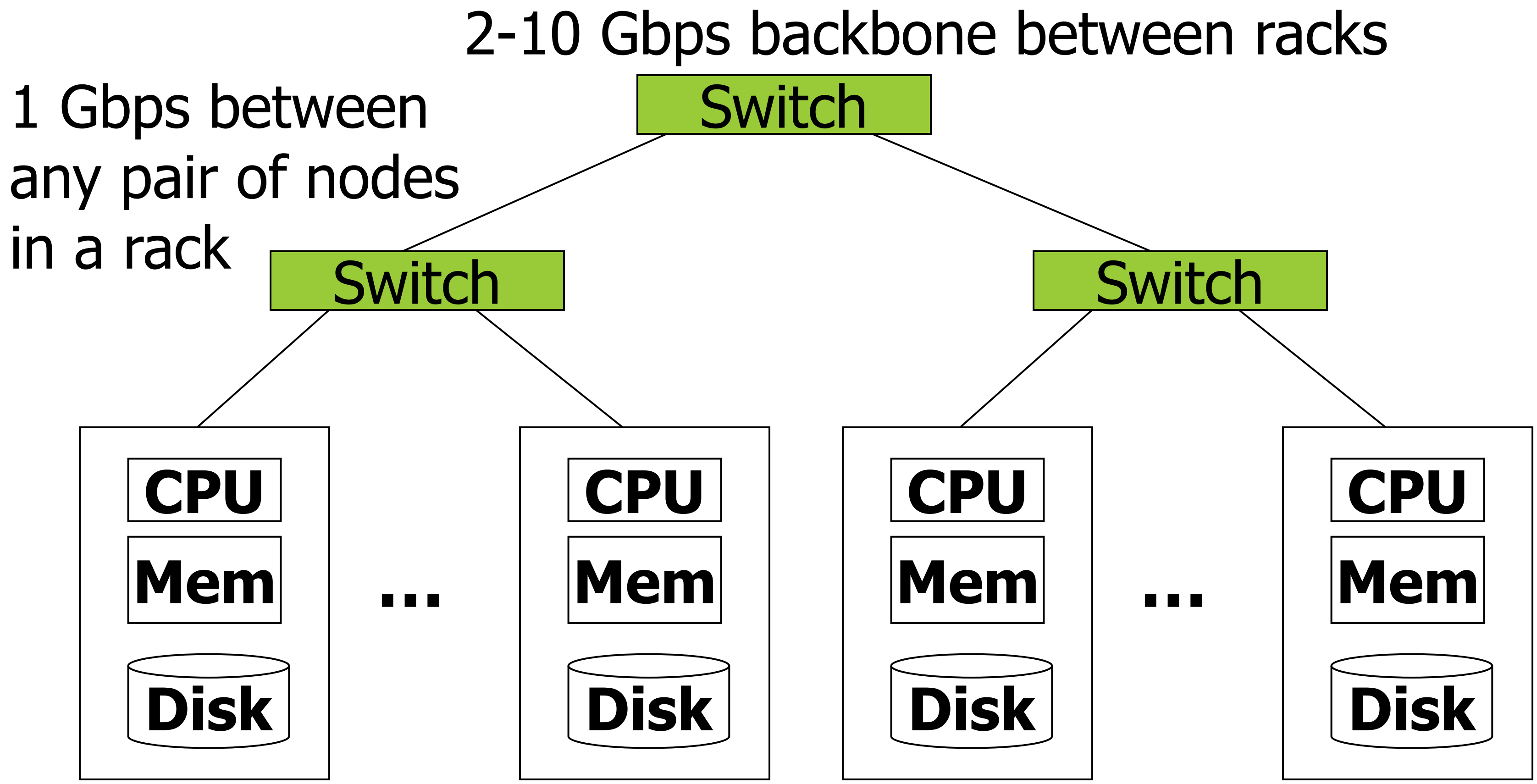
ENSF 612:
Lecture MapReduce Platform

Dr. Gias Uddin,
Department of Electrical and Software Engineering,
University of Calgary.
https://giasuddin.ca/

Topics

- Cluster architecture
- Key properties
- Distributed file system
- MapReduce programming model
- MapReduce implementation
- Performance refinements

Cluster Architecture



Each rack contains 16-64 nodes

In 2011 it was estimated that Google had 1M machines

http://bit.ly/Shh0RO

Key properties

- Store data on multiple nodes (computers)
  - Provides persistence and reliability
- Move computation close to data
  - Minimize data movement
- Simple programming model
  - Minimize all the above complexities
- **How are these properties satisfied?**

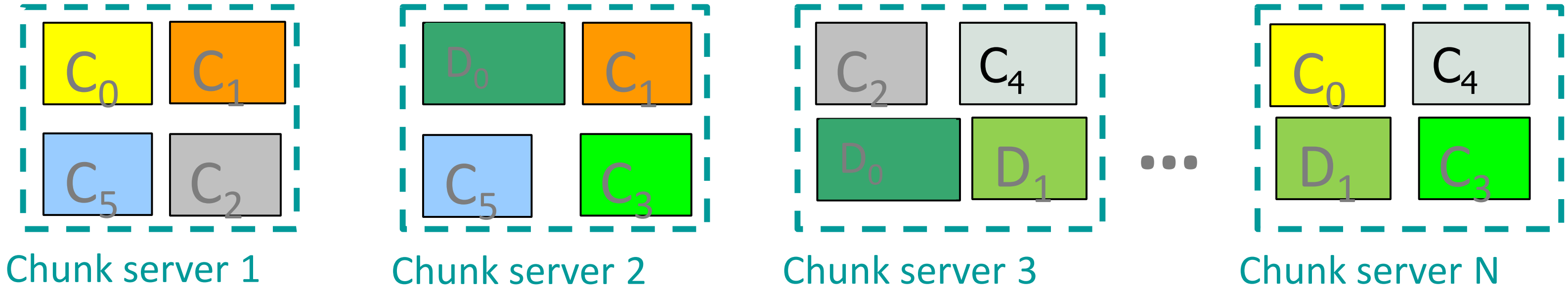
Distributed File System

- Data needs to be stored redundantly for reliability
  - Deal with nodes going down
- MapReduce platform uses distributed file system (DFS)
- DFS stores files across different nodes in a cluster
- DFS replicates a given piece of data on multiple nodes
  - Word count – node goes down with a partition
  - Start task on another node that has copy of that partition
- Examples: Google's GFS and Hadoop's HDFS

Distributed File System – cont'd

- Typical usage pattern of a DFS
  - Huge files (100s of GB to TB)
  - Data is rarely updated in place
  - Reads and appends are common
- First aspect of DFS - how is the data stored?
  - Data kept in chunks and spread across nodes
  - Each chunk is **replicated** on different nodes

Distributed File System – cont'd



- In above example, C0-C5 are chunks of file 1
- D0-D1 are chunks of file 2
- Each chunk replicated twice
- Replications don't share same server
- **Chunk servers are also where computations occur**!
  - Move computation to the data

Distributed File System – cont’d

- Summary of data storage
  - File is split into contiguous chunks (16-64 MB)
  - A chunk is replicated typically twice or thrice
  - We try to locate each replica in a different rack (why?)
- Second aspect of DFS is the **name/master node**
  - Keeps metadata about where files are stored
  - Needs to be replicated (why?)
- Third aspect of DFS is the client programming library
  - Contact name node – figure out chunk servers
  - Ask computation to be scheduled on those servers

MapReduce Computation Model

- Let's revisit the word count example in detail
- Can use UNIX commands and piping

  words(doc.txt) | sort | uniq -c

  where *words* takes a file and outputs the words in it, 1/line

  sort sorts the output of *words*

  uniq –c computes # of times each unique word appears

- This style of processing is naturally parallelizable
- MapReduce uses this style

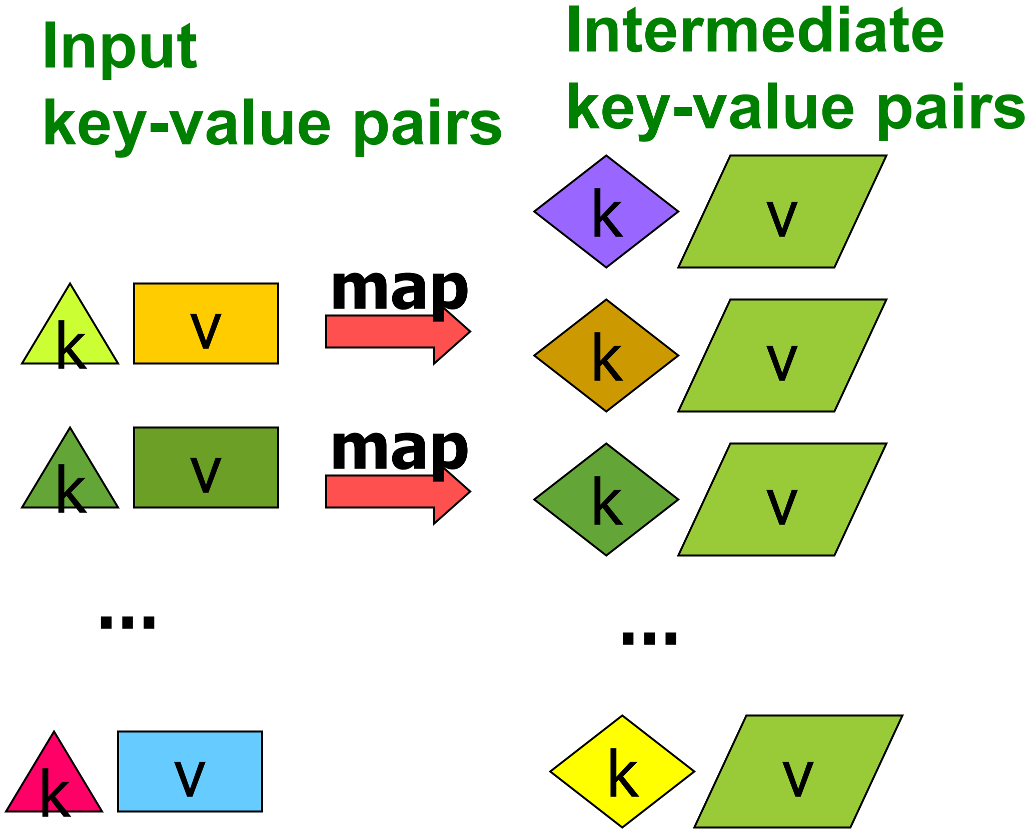
MapReduce Computation Model – cont'd

words(doc.txt)* | *sort* | *uniq -c

- **Map**
  - Scan input file one record at a time
  - Extract something useful from each record (key)
- **Group by key**
  - Sort and shuffle
- **Reduce**
  - Aggregate, summarize, filter, or transform
  - Output results
- Steps same – Map and Reduce change to fit problem

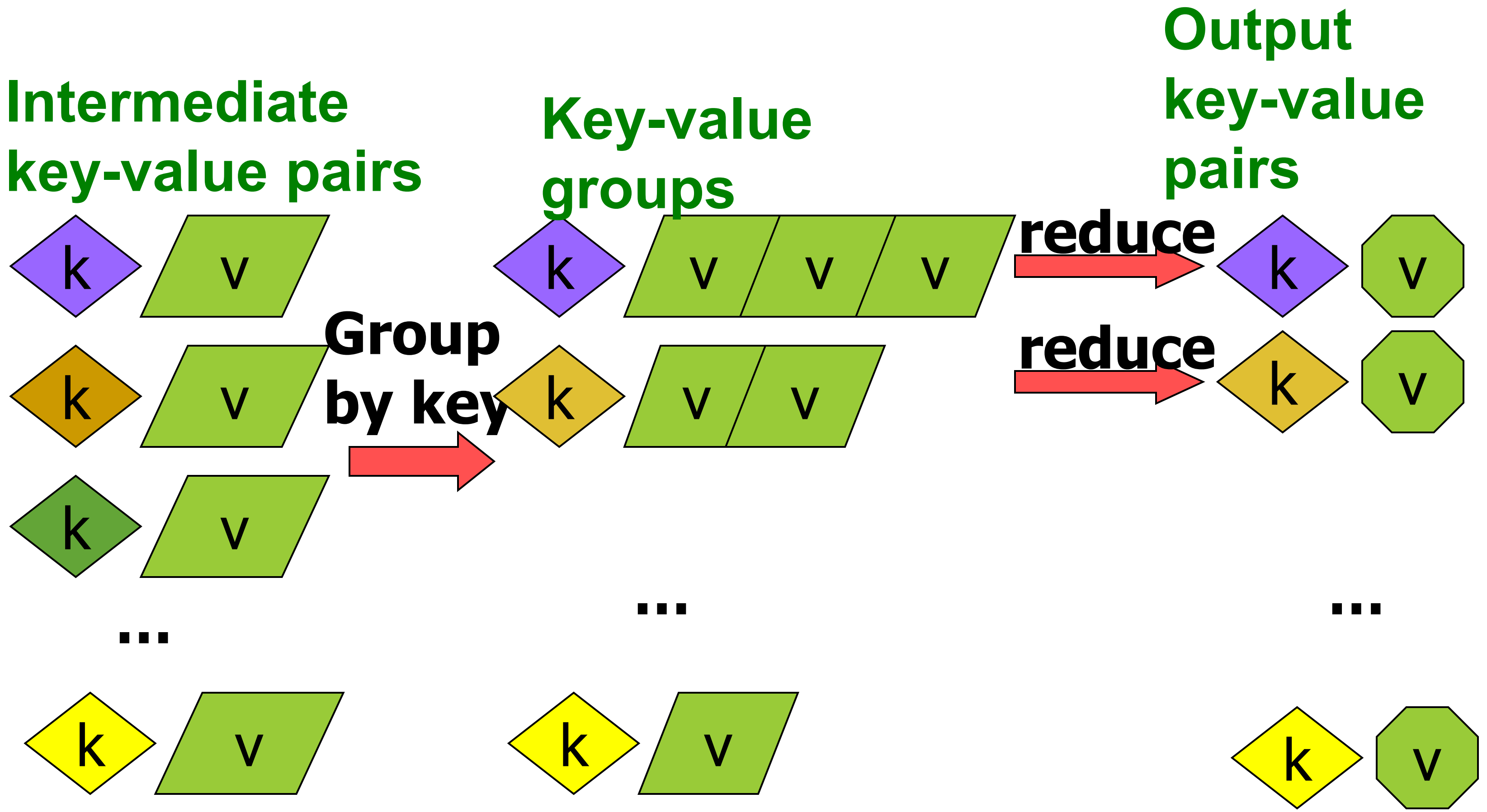
MapReduce Computation Model – cont'd

- MapReduce – series of transformations of key-value pairs
- **Map step**



MapReduce Computation Model – cont'd

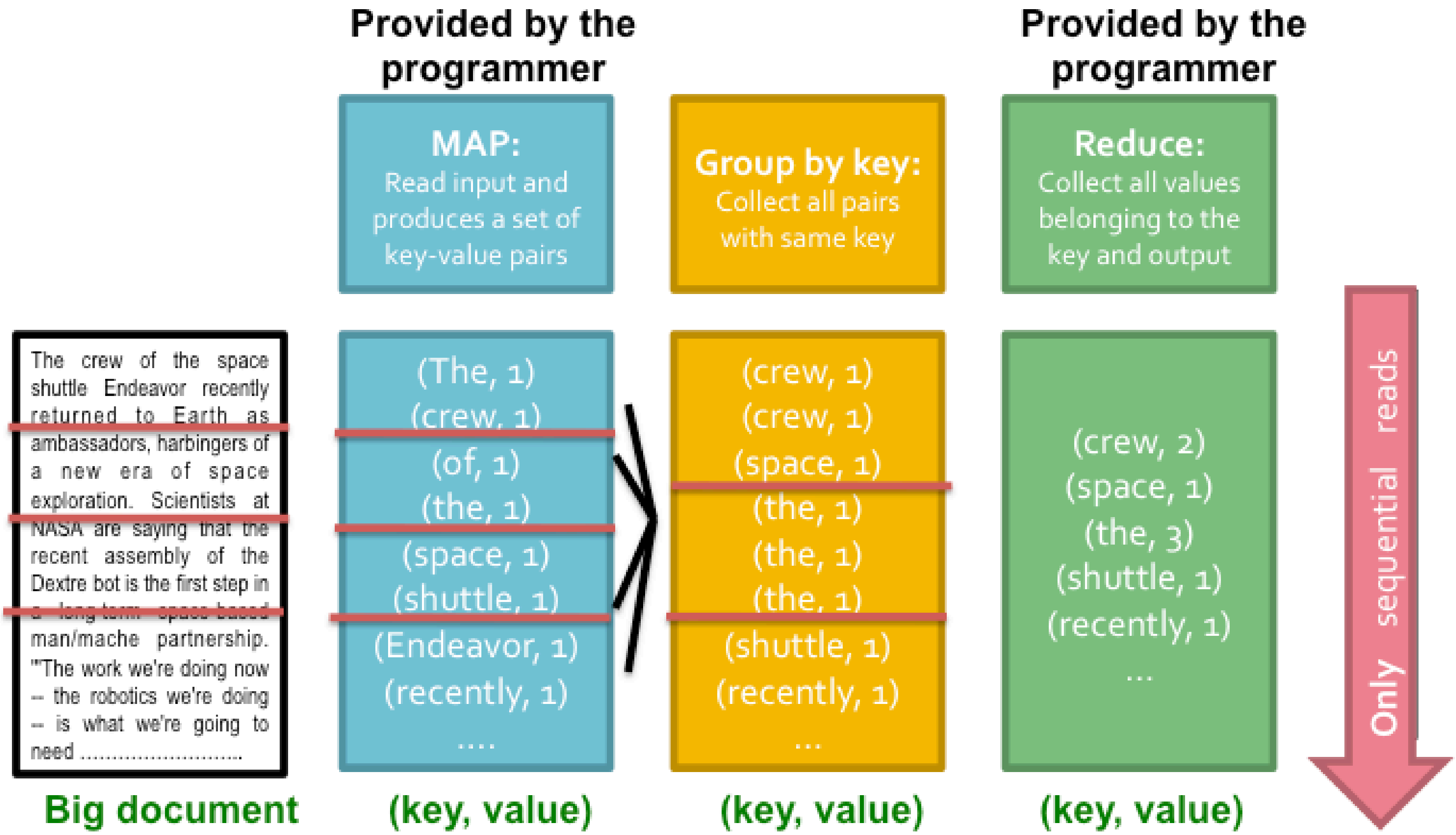
- **Reduce step**



MapReduce Computation Model – cont’d

- Formal description of MapReduce
- **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 Computation Model – cont'd



MapReduce Computation Model – cont'd

- Word count pseudo code

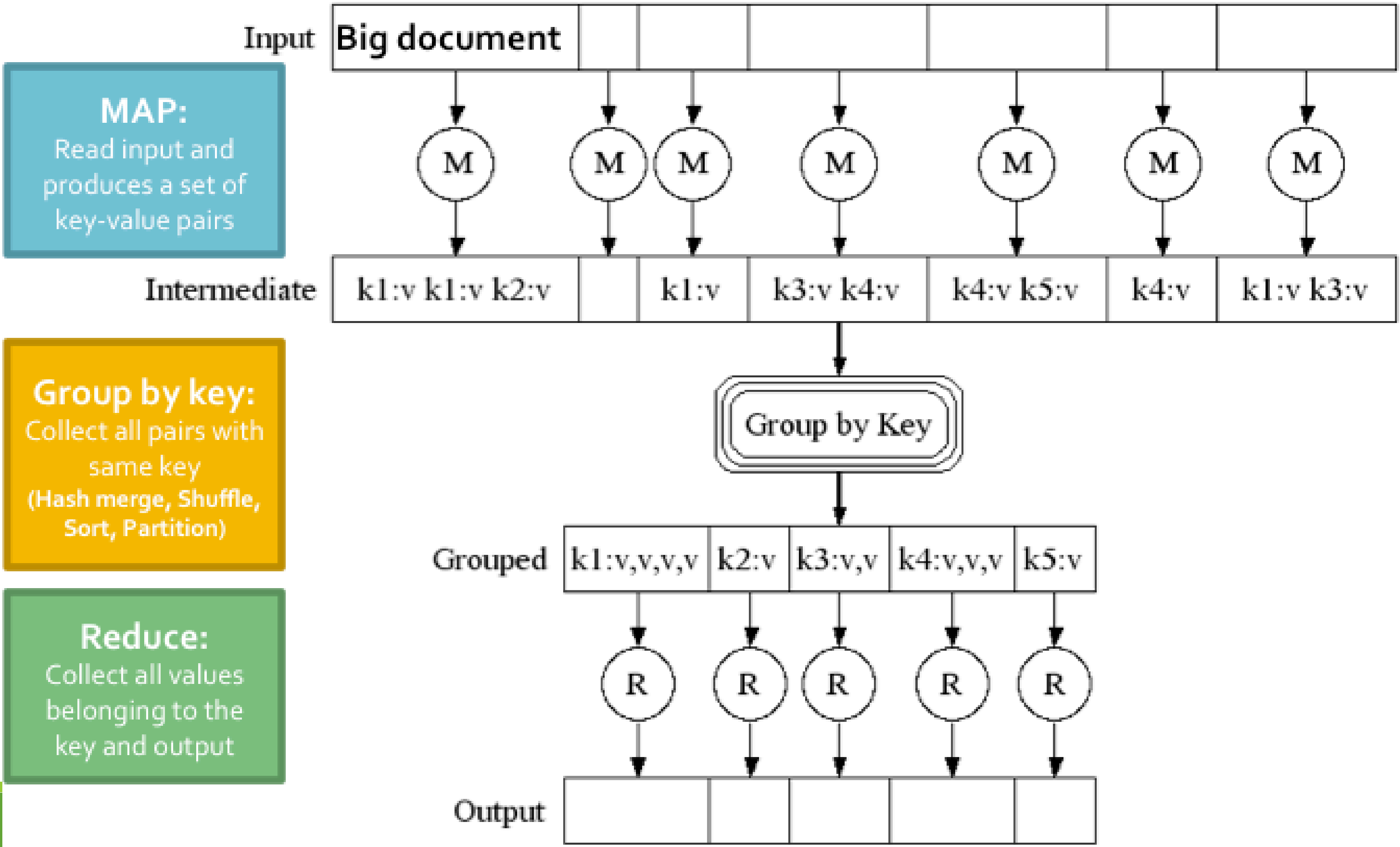
```
map(key, value):
// key: document name; value: text of the document
   for each word w in value:
    emit(w, 1)


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

MapReduce Implementation

- A MapReduce implementation should take care of
  - **Partitioning** the input data
  - **Scheduling** the program’s execution across a set of nodes
  - Performing the **group by key** step
  - Handling machine failures
  - Managing required inter-machine communication

MapReduce Implementation – cont'd

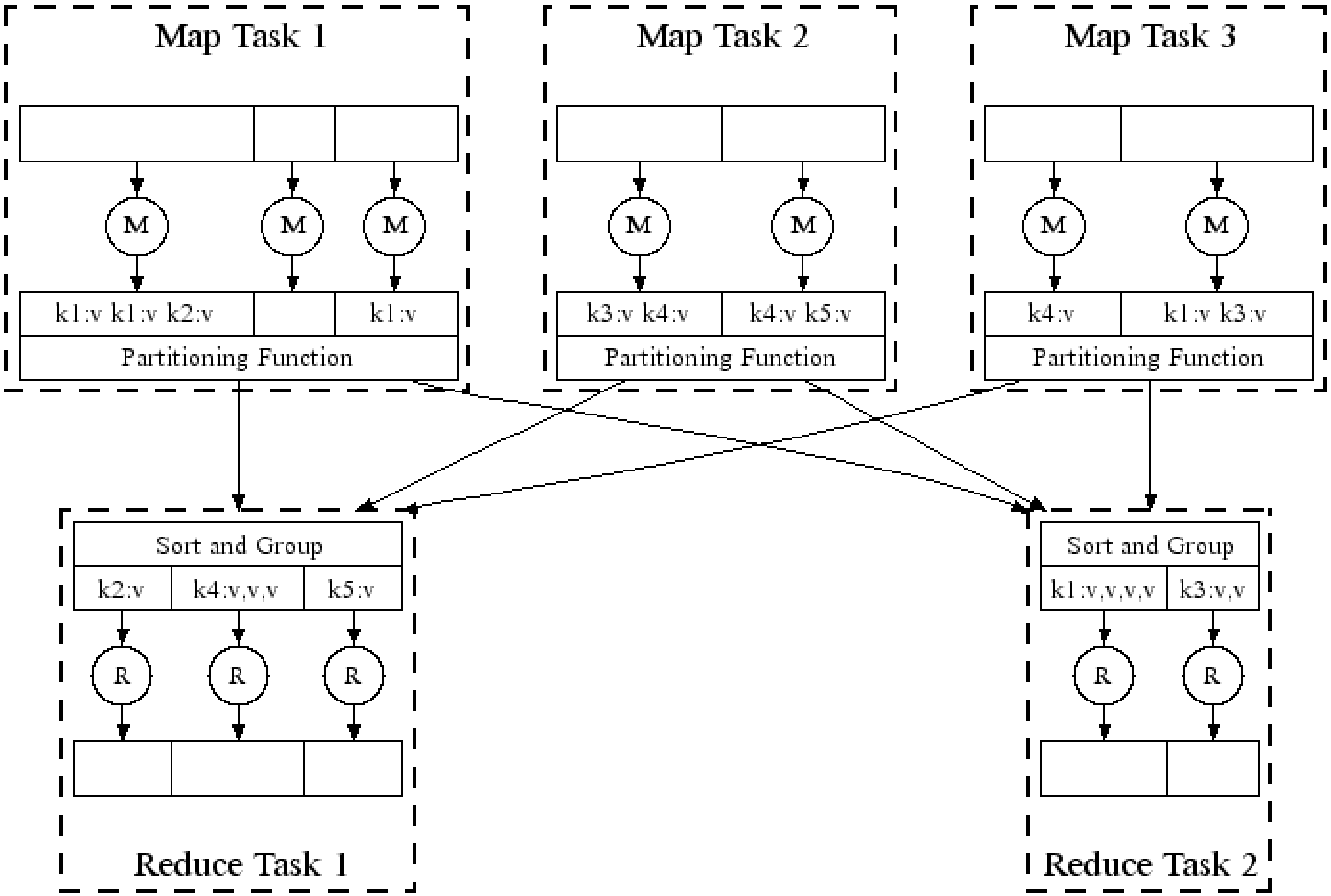
- Recap: conceptual execution of a MapReduce job



MapReduce Implementation – cont'd

- Recap: parallel execution of a MapReduce job

- 3 phase are distributed with many tasks doing them



MapReduce Implementation – cont'd

- Data flow
  - Input and final output are stored on DFS
    - Schedule map tasks "close" to storage of input data
  - Intermediate results stored on local FS of nodes
    - Why?
  - Output is often input to another MapReduce task

MapReduce Implementation – cont'd

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

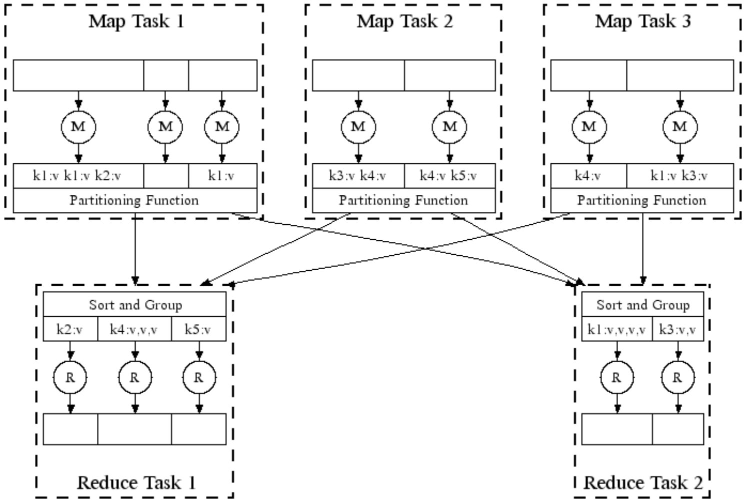
MapReduce Implementation – cont'd

- Dealing with failures
- Map worker failure
  - Tasks completed or in-progress at worker are reset to idle
  - Reduce workers 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

MapReduce Implementation – cont'd

- How many maps and how many reduces?
  - Need to specify # of Map/Reduce tasks –*M*/ *R*
  - Rule of thumb
    - **Make *M* much > the number of nodes in the cluster**
    - One DFS chunk per map is common
    - Improves dynamic load balancing
    - Speeds up recovery from worker failures
    - **Usually *R* is smaller than *M***
      - Because output is spread across *R* files

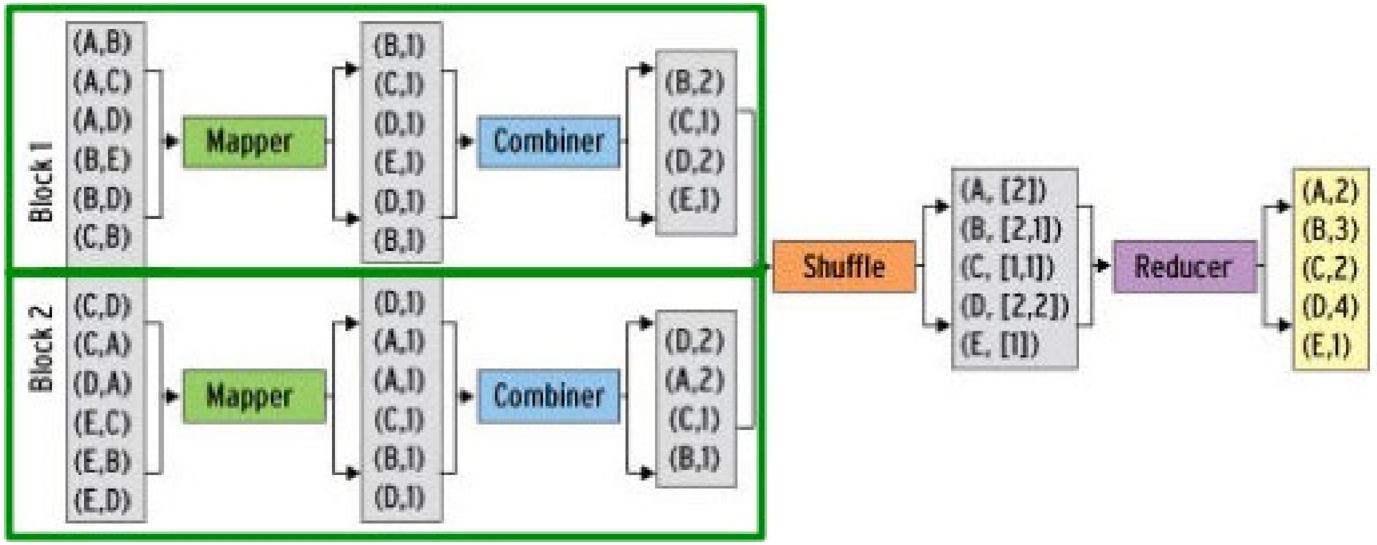
Performance Refinements



Combiners

- Often a Map task will produce many pairs of the form *(k,v_1), (k,v_2)*
  - E.g., popular words in the word count example
- Can save network time by pre-aggregating values in mapper
  - combine(k, list(v_1)) → v_2
  - Combiner is usually same as the reduce function

Performance Refinements – cont'd



Word count example

- Combiner combines values of all keys in a single mapper node
- Much less data needs to be copied and shuffled

Performance Refinements – cont'd

- Combiners work only if reduce function is commutative & associati
  - Example 1: *sum* function
  - Example 2: *average* function
  - Example 3: *median* function

Performance Refinements – cont'd

- Can control how keys get shipped to reducers
  - Set of keys that go to a singe reduce worker
- System uses a default partition function
  - ***hash(key) mod R***
- Sometimes useful to override hash function
  - E.g., **hash(hostname(URL)) mod *R***
  - Ensures URLs from a host end up in the same output file

MapReduce Platform Implementations

- Google MapReduce
  - Uses Google File System (GFS) for stable storage
  - Not available outside Google
- Hadoop
  - Open-source implementation in Java
  - Uses HDFS for stable storage
- Hive and Pig
  - Provide an SQL-like interface on top of MapReduce

Topics

- Cluster architecture
- Key properties
- Distributed file system
- MapReduce programming model
- MapReduce implementation
- Performance refinements

Acknowledgements

Portions of these slides were adapted with permission from Leskovec et al.

(http://www.mmds.org)