COMP4434 Big Data Analytics

Lecture 11 MapReduce

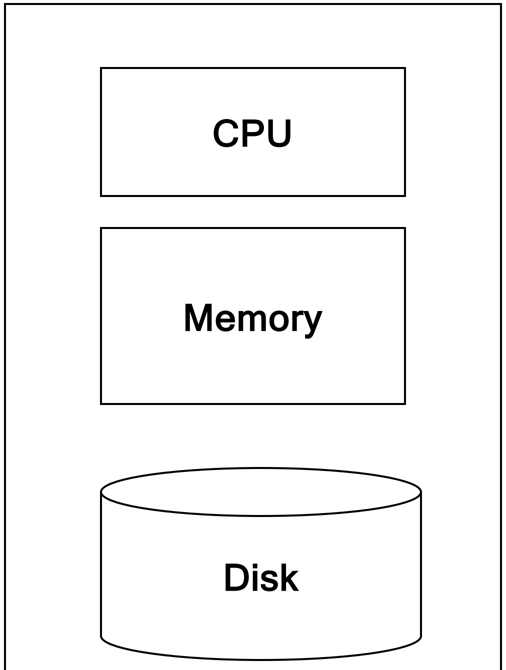
HUANG Xiao

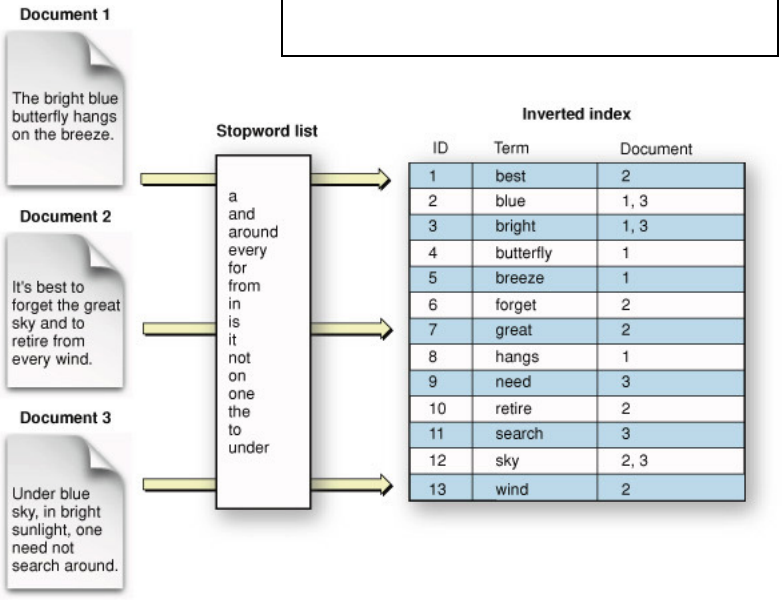
xiaohuang@comp.polyu.edu.hk

Motivation: Google Example

- Google
  - TB of Web Data stored by using Inverted index
  - TB to PB of Logs
    - Analysis: find the most popular keywords
- Processing cannot be done by one machine efficiently
  - Needs Parallel Machines

1 Terabyte = 10^12 byte
1 Petabyte = 10^15 byte




Document 1

The bright blue butterfly hangs on the breeze.

Document 2

It's best to forget the great sky and to retire from every wind.

Document 3

Under blue sky, in bright sunlight, one need not search around.

Stopword list

a
and
around
every
for
from
in
is
it
not
on
one
the
to
under

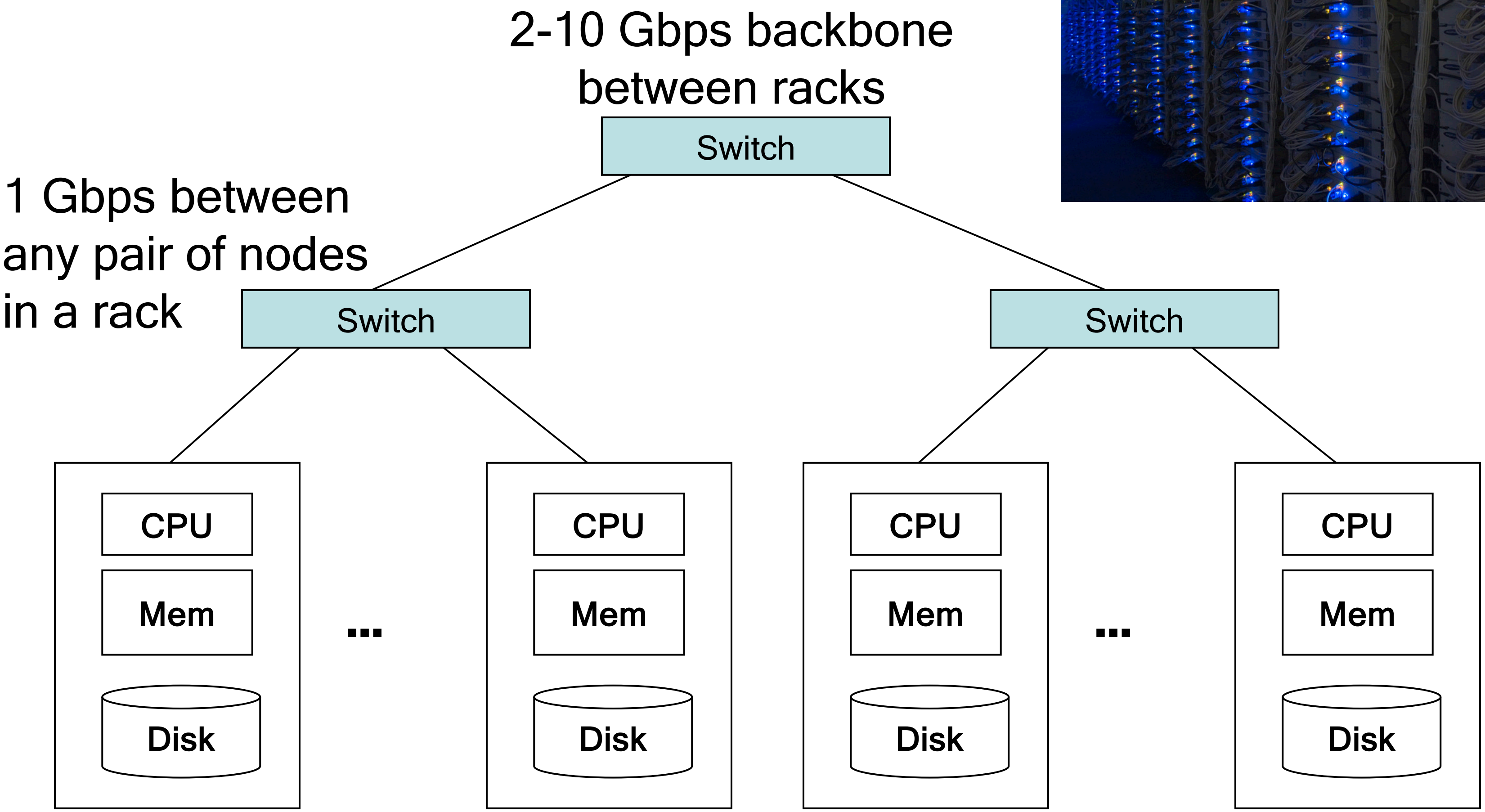
Inverted index

ID	Term	Document
1	best	2
2	blue	1, 3
3	bright	1, 3
4	butterfly	1
5	breeze	1
6	forget	2
7	great	2
8	hangs	1
9	need	3
10	retire	2
11	search	3
12	sky	2, 3
13	wind	2

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 Linux nodes
  - Commodity network (ethernet) to connect them

Cluster Architecture


2-10 Gbps backbone between racks
Switch
1 Gbps between any pair of nodes in a rack
Switch
Switch
CPU
Mem
Disk
...
CPU
Mem
Disk
CPU
Mem
Disk
...
CPU
Mem
Disk

Each rack contains 16-64 nodes

Challenges in Large-scale Computing

- 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!

Origin of MapReduce

- MapReduce is a programming model Google has used successfully for processing its “big-data” sets (~ 20000 peta bytes per day)
  - Users specify the computation in terms of a **map** and a **reduce** function
  - Underlying runtime system automatically parallelizes the computation across large-scale clusters of machines
  - Underlying system also handles machine failures, efficient communications, and performance issues
    - Store files multiple times for reliability

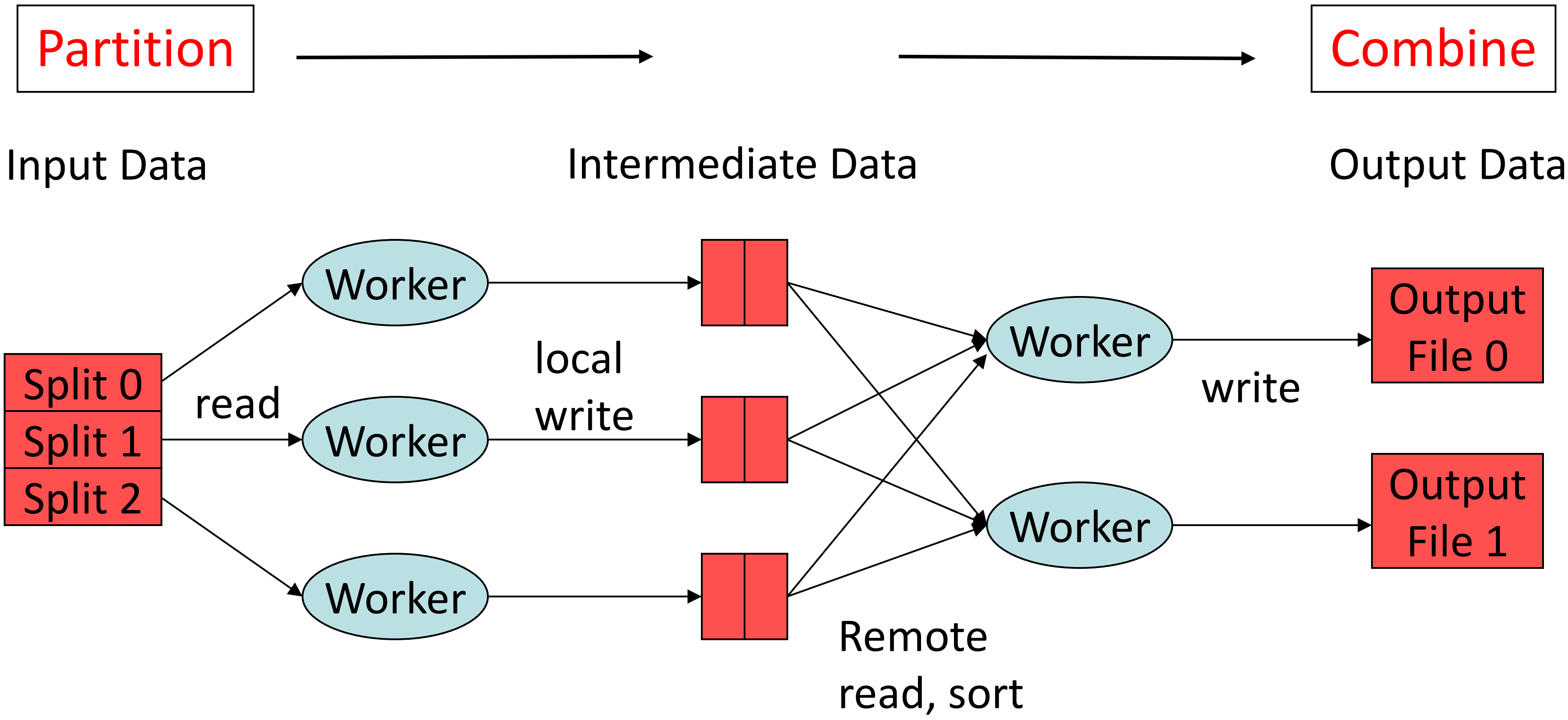
Example Problem

- Given: a massive data collection of documents (100 Terabytes)
- Problem: **Count** the frequency of each term in this document collection
- Idea: **distributed processing**, i.e., solve this problem by using multiple computers

MapReduce Workflow

- **Map**: a mapping that is responsible for dividing the data and transforming the original data into key-value pairs
- **Shuffle**: the process of further organizing and delivering the Map output to the Reduce
  - the output of the Map must be sorted and segmented
  - then passed to the corresponding Reduce
- **Reduce**: a merge that processes the values with the same key and then outputs to the final result

Parallelization: Divide and Conquer



Partition

Combine

Input Data

Intermediate Data

Output Data

Split 0

Split 1

Split 2

read

Worker

Worker

Worker

local write

Worker

Worker

Remote read, sort

write

Output File 0

Output File 1

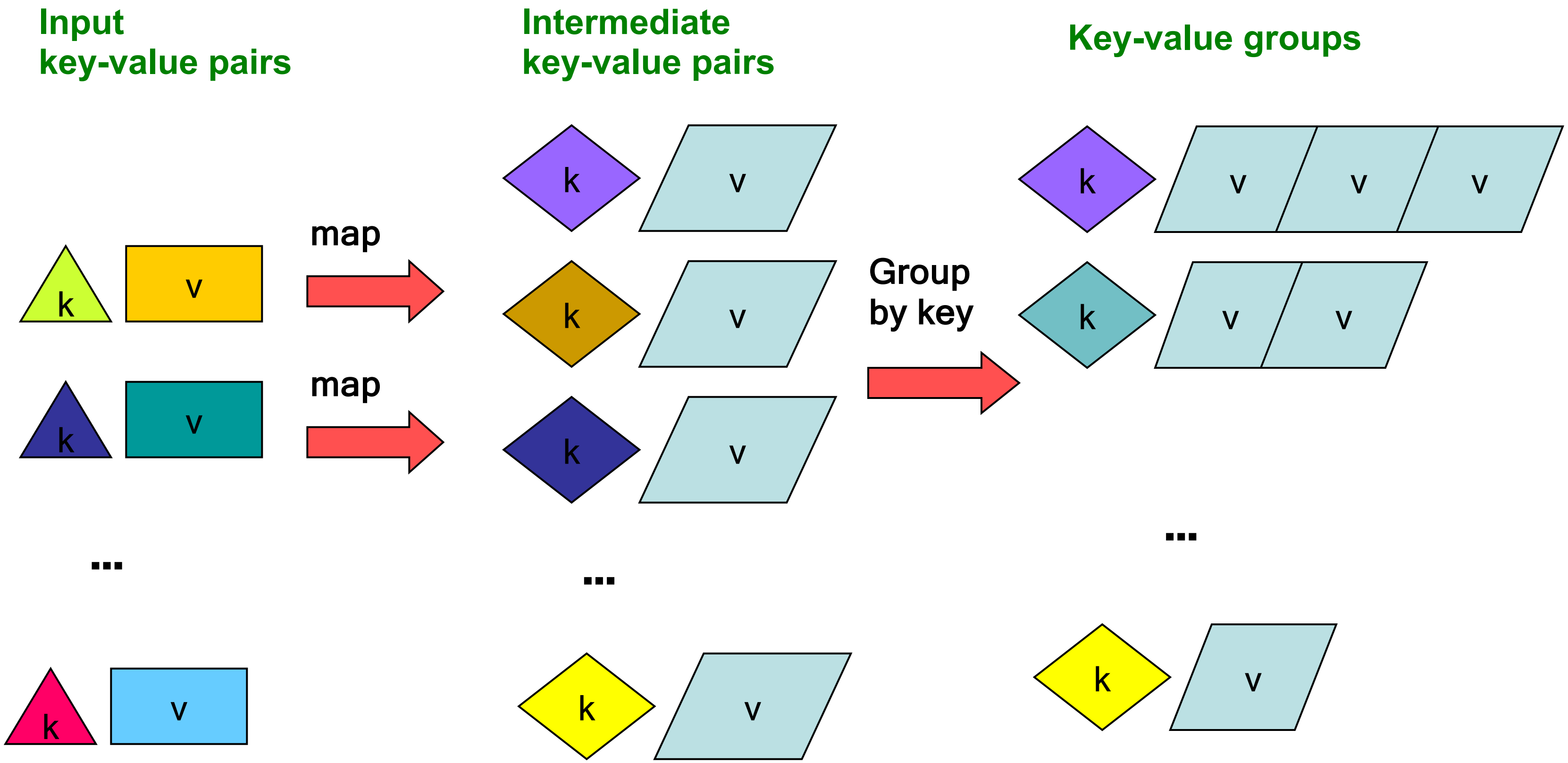
Map

extract something you care about from each record

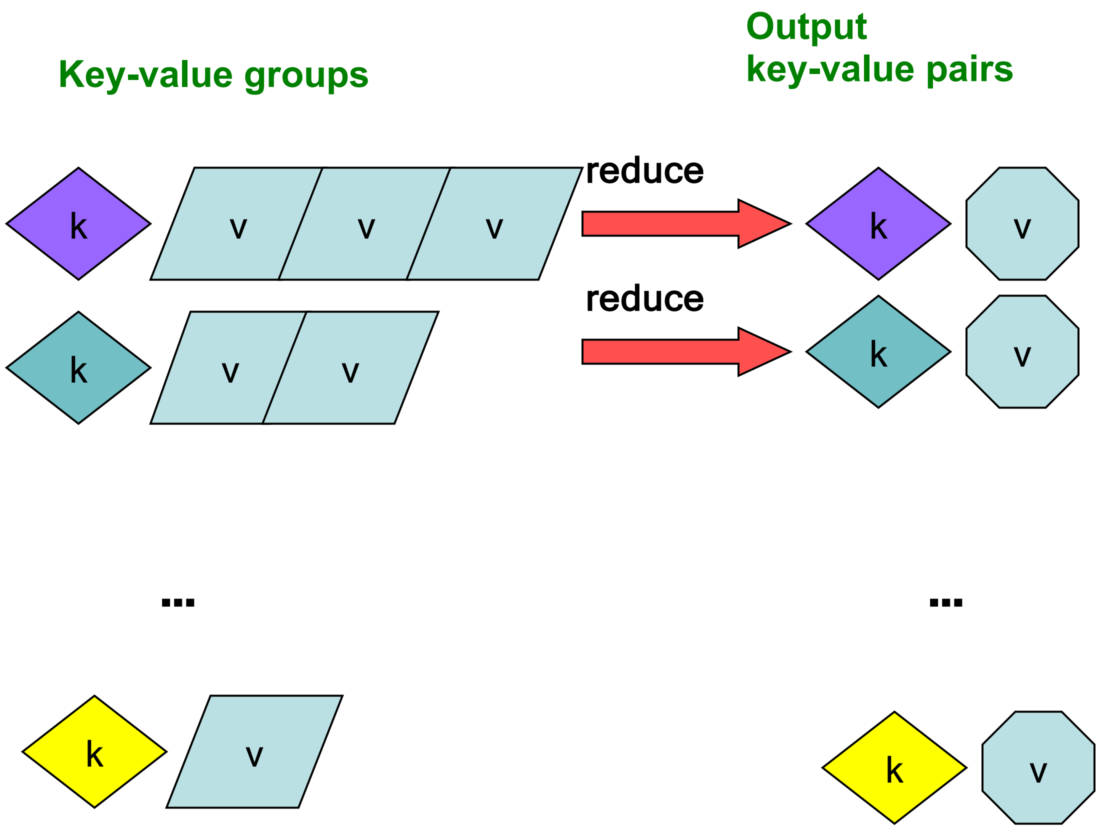
Reduce

aggregate, summarize, filter, or transform

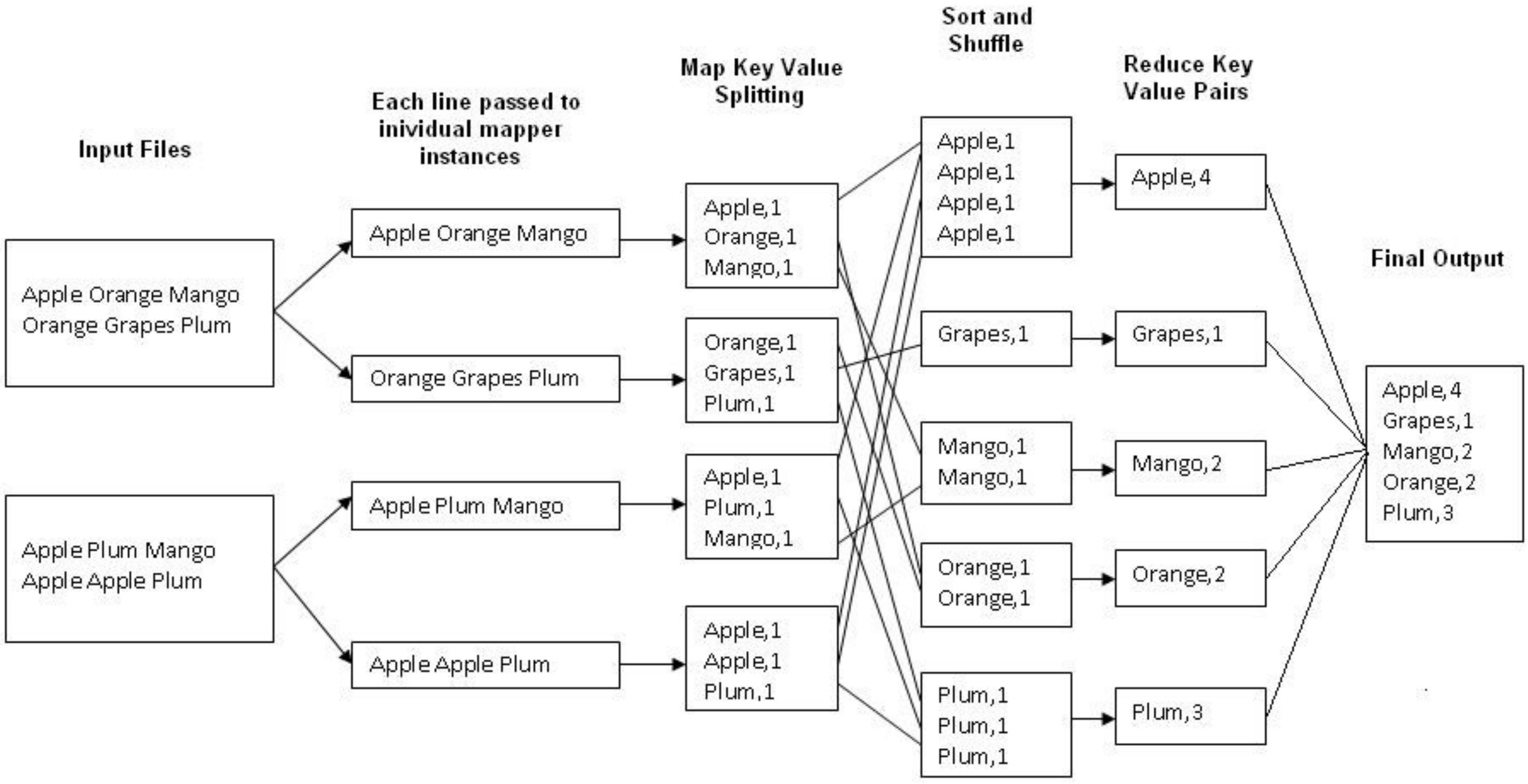
MapReduce: The Map Step



MapReduce: The Reduce Step



Example Problem



http://kickstarthadoop.blogspot.ca/2011/04/word-count-hadoop-map-reduce-example.html

Word Count Using MapReduce

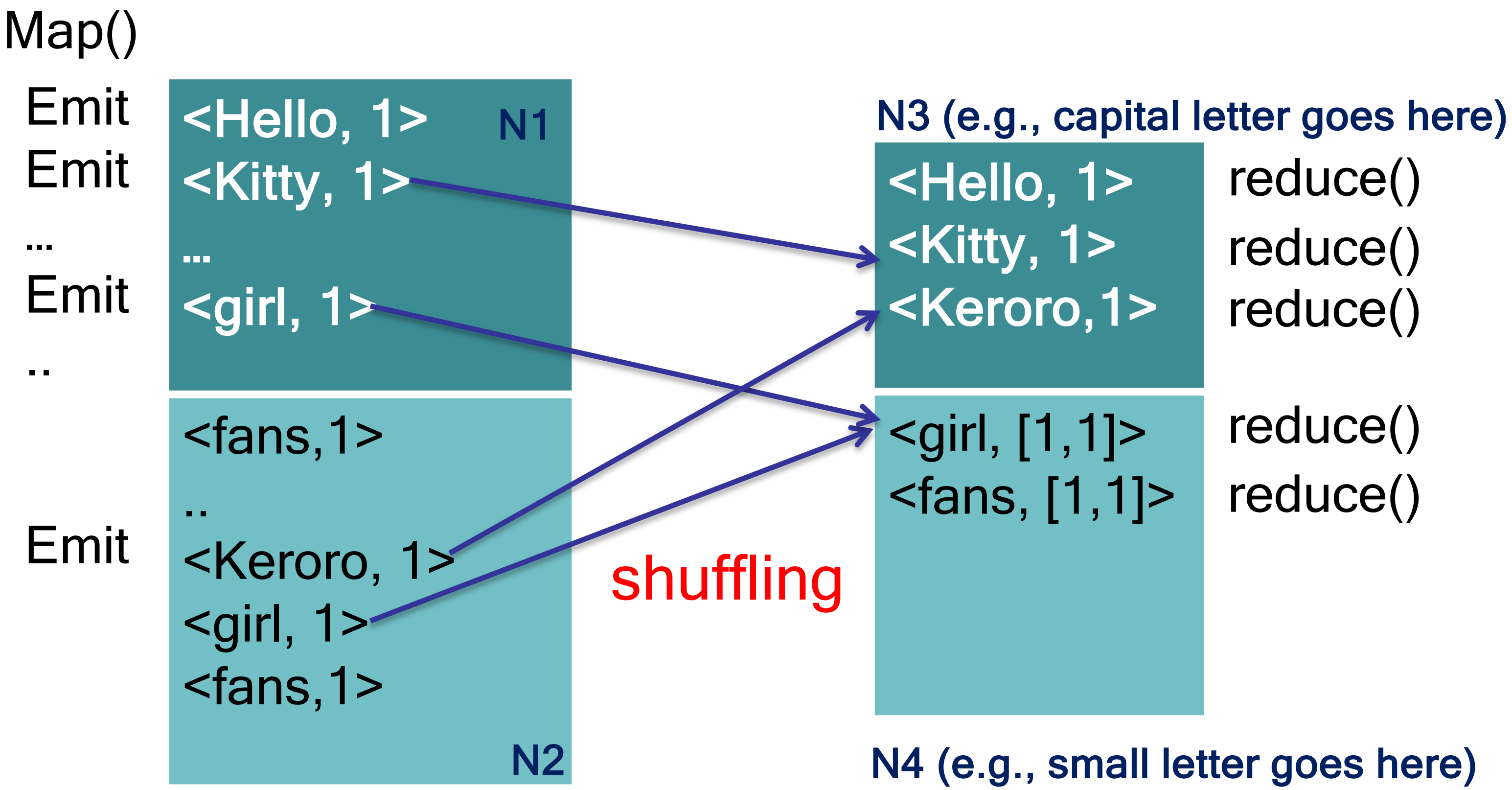
```
map(key, value):
// key: document name; value: text of the document
   for each word w in value:
      emit(w, 1)


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

MapReduce Program

```
Map() {    //to emit, do pre-processing
     Open File F.txt;
     while has lines left {
       L = Read a line;
       S = tokenize(L);  // S is a list of tokens
       for each token in S {
            emit(S[i], 1); // key value pair
       }
     }
}
```

```
Reduce(k, value-list) {
// <key, value> with same key will shuffle to the same machine
// this Reduce function is invoked once per unique k
    append(k, size(value-list)) to file WC.txt;
}
```

System View



Map()

Emit <Hello, 1>

Emit <Kitty, 1>

… …

Emit <girl, 1>

..

N1

<fans,1>

..

Emit <Keroro, 1>

<girl, 1>

<fans,1>

N2

shuffling

N3 (e.g., capital letter goes here)

<Hello, 1> reduce()

<Kitty, 1> reduce()

<Keroro,1> reduce()

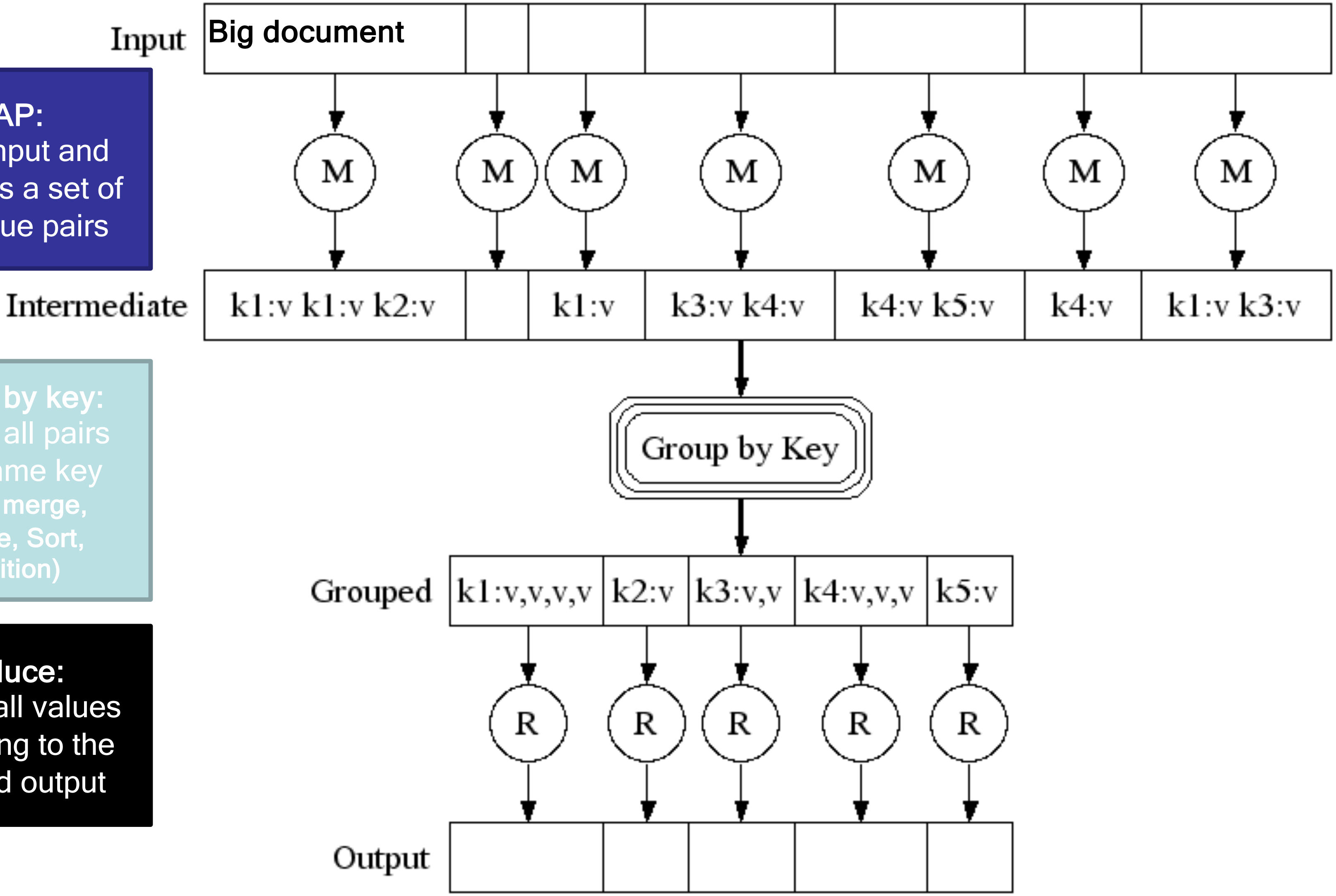
<girl, [1,1]> reduce()

<fans, [1,1]> reduce()

N4 (e.g., small letter goes here)

Hello Kitty has a lot of girl fans
Keroro has two girl fans

Map-Reduce: A diagram



Input: Big document

Intermediate: k1:v k1:v k2:v | | k1:v | k3:v k4:v | k4:v k5:v | k4:v | k1:v k3:v

Group by Key

Grouped: k1:v,v,v,v | k2:v | k3:v,v | k4:v,v,v | k5:v

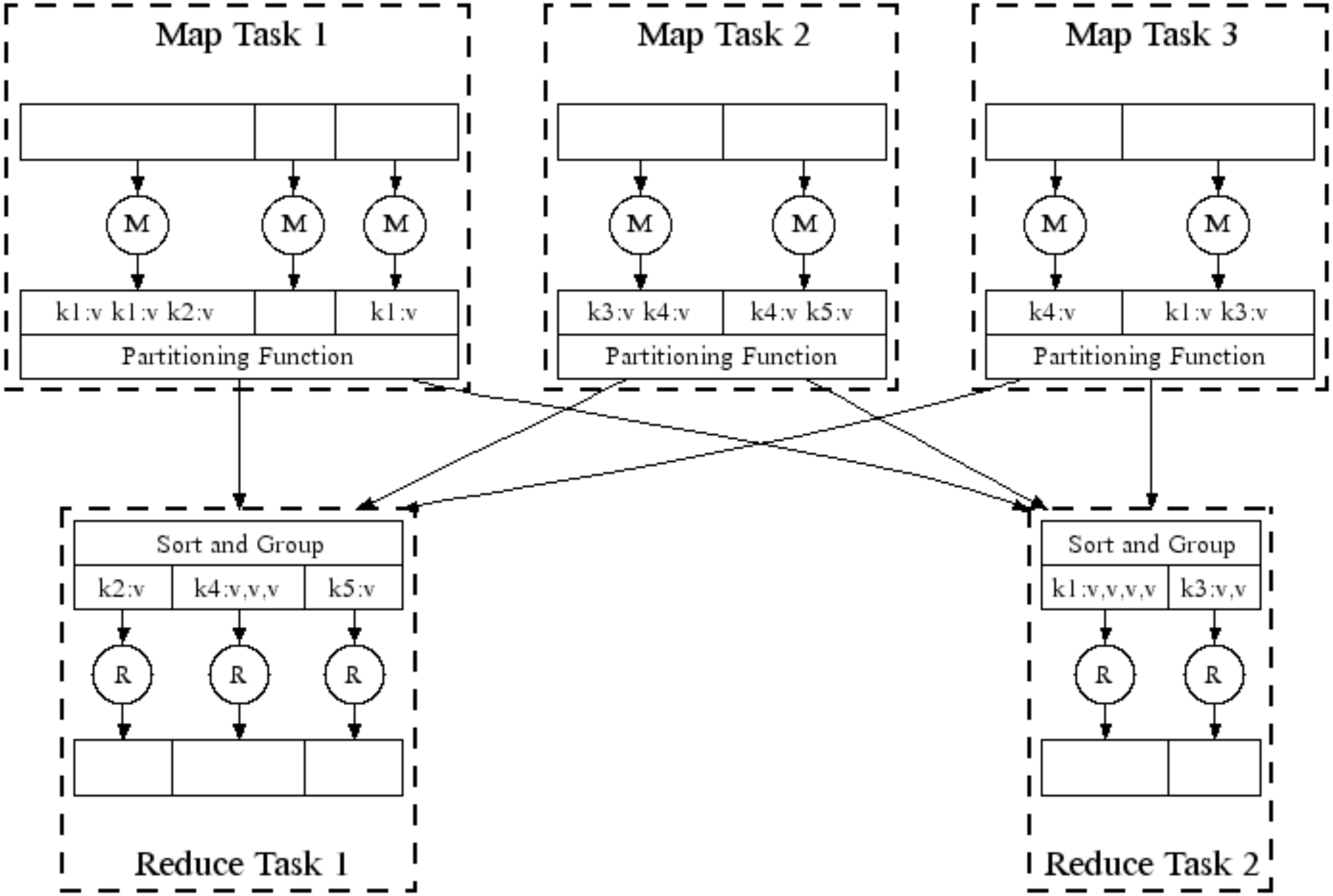
Output

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

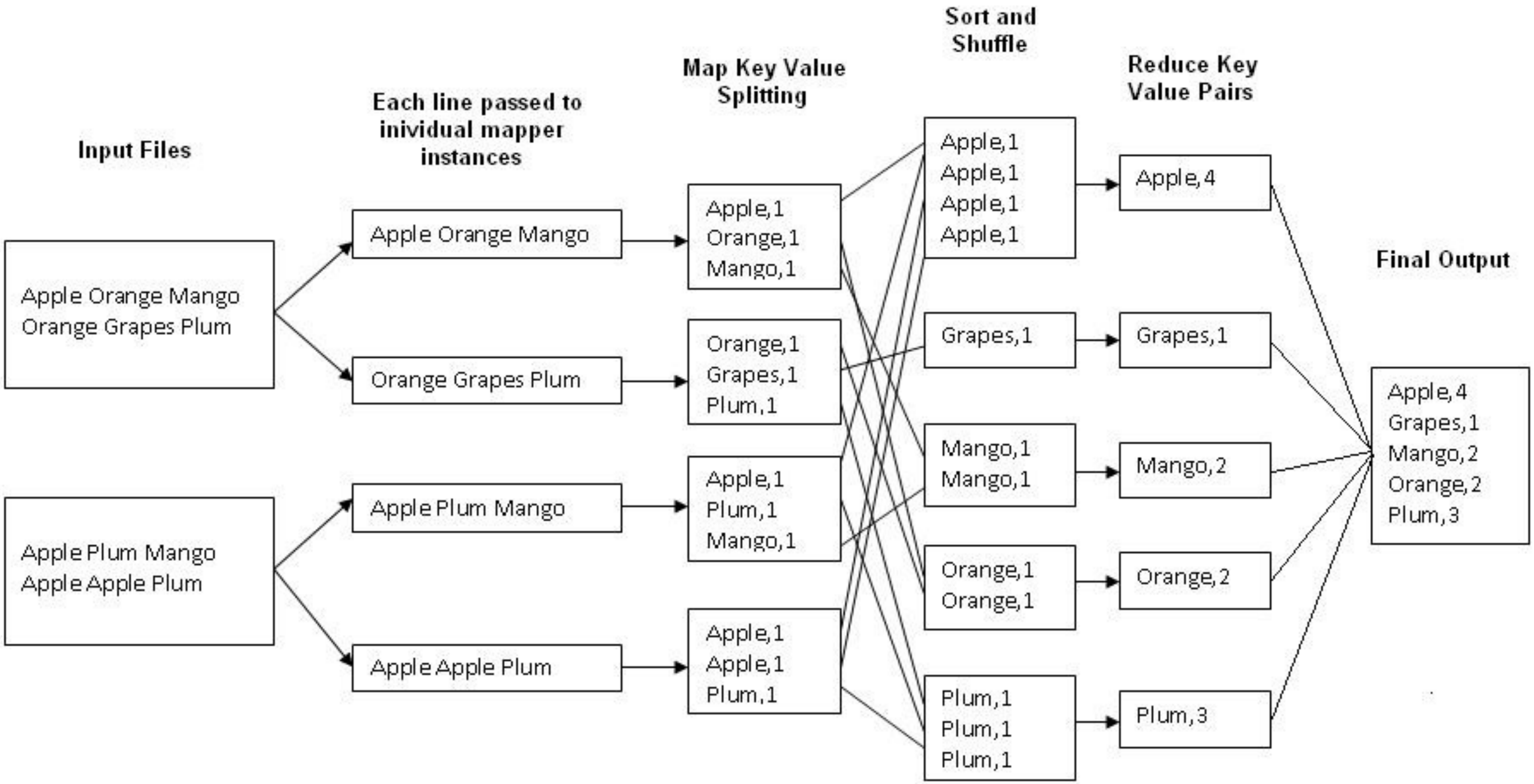


All phases are distributed with many workers doing the tasks

Programmer Specifies

- Map, Reduce, input files
- Workflow:
  - Read inputs as a set of key-value-pairs
  - **Map** transforms input kv-pairs into a new set of k'v'-pairs
  - Sorts & Shuffles the k'v'-pairs to output nodes
  - All k'v'- pairs with a given k' are sent to the same **reduce**
  - **Reduce** processes all k'v'-pairs grouped by key into new k''v''-pairs
  - Write the resulting pairs to files
- All phases are distributed with many workers doing the tasks

Example Problem Again



http://kickstarthadoop.blogspot.ca/2011/04/word-count-hadoop-map-reduce-example.html

Case 1: SQL Execution

- Given a 10TB table in HDFS: `Student(id,name,year,gpa,gender)`
- Write a MapReduce program to compute the following query

```
SELECT   year, AVG(gpa)
FROM     Student
WHERE    gender = ‘Male’
GROUP BY year
```

Query to look at the average gpa for the male students in each year from table Student.

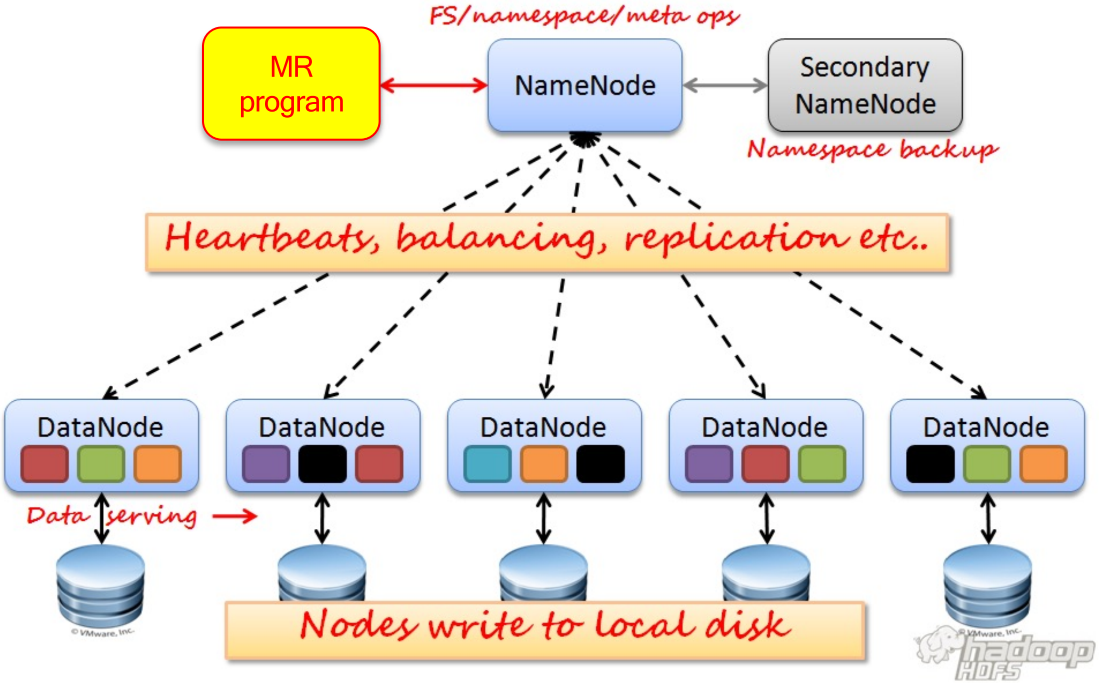
MapReduce Program

```
Map(String key, Record value) {
    // key:        table name
    // value:      table content

    for each row in value {
        if (row.gender = ‘Male’)
            emit(row.year, row.gpa); // intermediate key value pairs
    }
}
```

```
Reduce(String key, Iterator values) {
    // key:        year
    // values:     a list of gpa

    sumGPA = 0;
    for each gpa in values
        sumGPA += gpa;
    emit(key, sumGPA/sizeof(values));
}
```

MapReduce Programming

- Just write the content for Map() and Reduce() function
- Don't even know how the splits are distributed
- Hadoop system will find the right block for each split



- Be aware that the large file is split to multiple machines
- No need to take care of which machines have the block of the file

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

Manage Multiple Workers

- Challenges:
  - Workers may run in any order
  - Workers may interrupt each other
  - Don’t know when workers need to communicate partial results
  - Workers may access shared data in any order
- Solutions:
  - Do not allow workers to access shared data immediately
  - Do not allow workers to interrupt each other
  - When workers finish, perform **batch updates**
  - Scheduler tries to schedule map tasks “close” to physical storage location of input data
    - MapReduce assumes an architecture where processors and storage are co-located

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
  - Completed map tasks are re-executed when failure occurs because their output is stored on the local disk(s) of the failed machine and therefore inaccessible
- Reduce worker failure
  - Only in-progress tasks are reset to idle
  - Reduce task is restarted
  - Completed reduce tasks do not need to be re-executed since their output is stored in a global file system
- Master failure
  - MapReduce task is aborted and client is notified

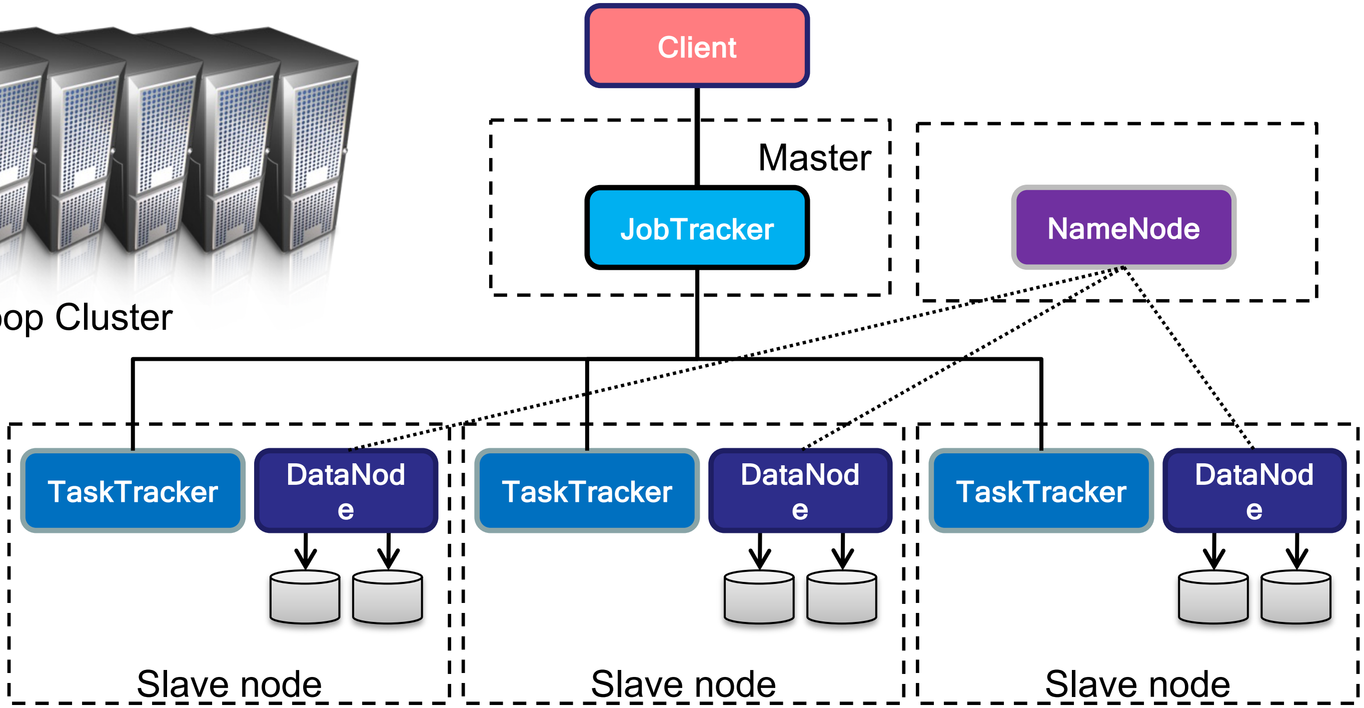
Who are using MapReduce?

- Google
  - Original proprietary implementation
- Apache Hadoop MapReduce
  - Most common (open-source) implementation
  - Built to specs defined by Google
- Amazon Elastic MapReduce
  - Uses Hadoop MapReduce running on Amazon EC2
- Microsoft Azure HDInsight
- or Google Cloud MapReduce for App Engine

Hadoop Cluster Architecture

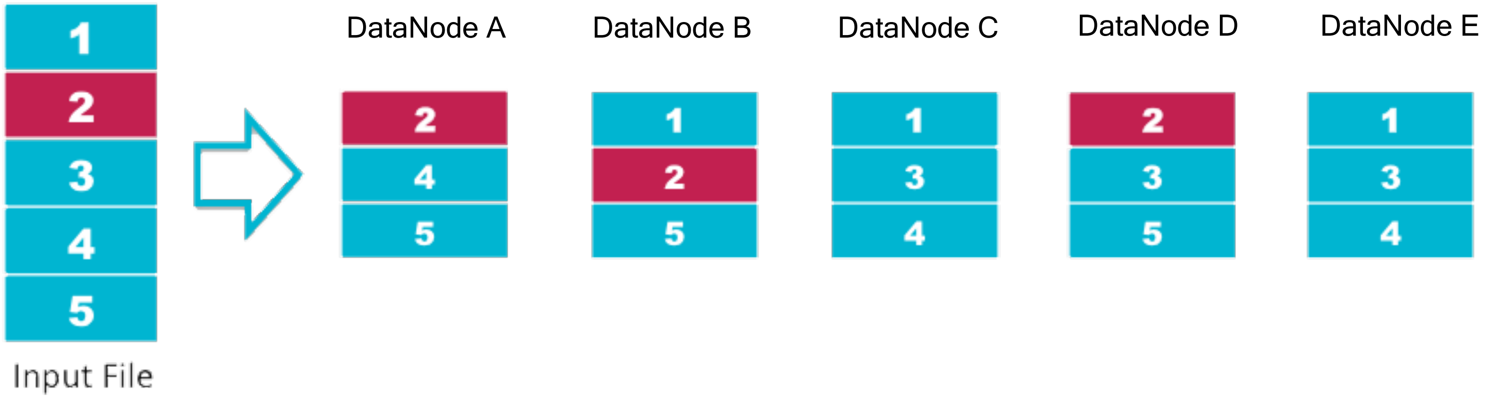
- A Hadoop Cluster includes a single master and multiple slave nodes.
  - Namenode: Central manager for the file system namespace
  - DataNode: Provide storage for objects
  - JobTracker: Central manager for running MapReduce jobs
  - TaskTracker: accept and runs map, reduce and shuffle

Hadoop Cluster



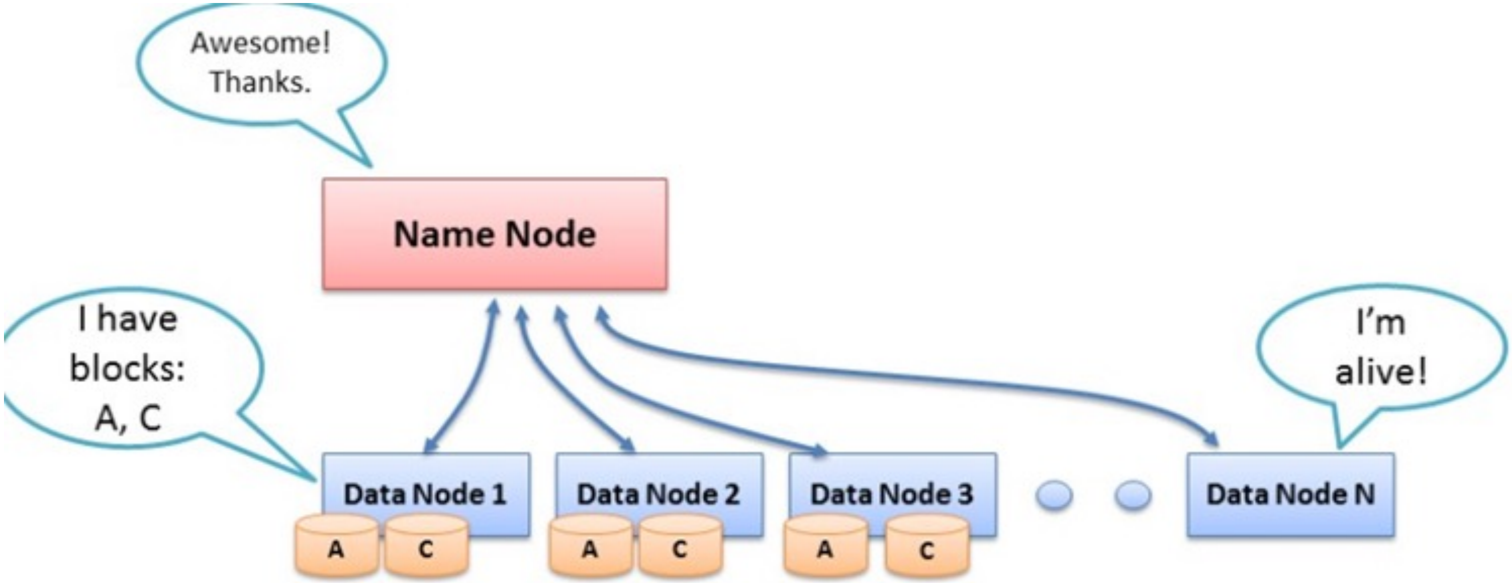
Hadoop Distributed File System (HDFS)

- HDFS is a block-structured file system: each file will be separated into blocks (typically 64MiB)
- Each block of a file is replicated across a number of data nodes to prevent loss of data



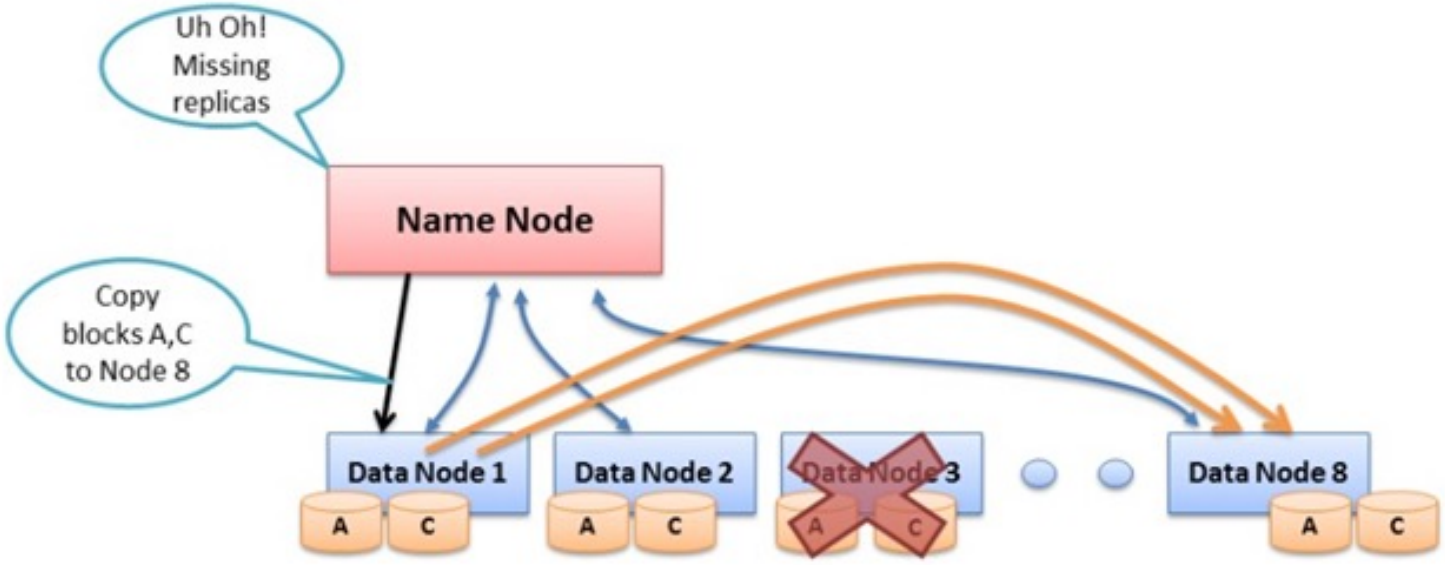
Heartbeat

- Each DataNode sends a Heartbeat message to the NameNode periodically (every 3 seconds via a TCP handshake).
- The NameNode can detect a dead DataNode by the absence of Heartbeat messages for a period of time (10 minutes by default).

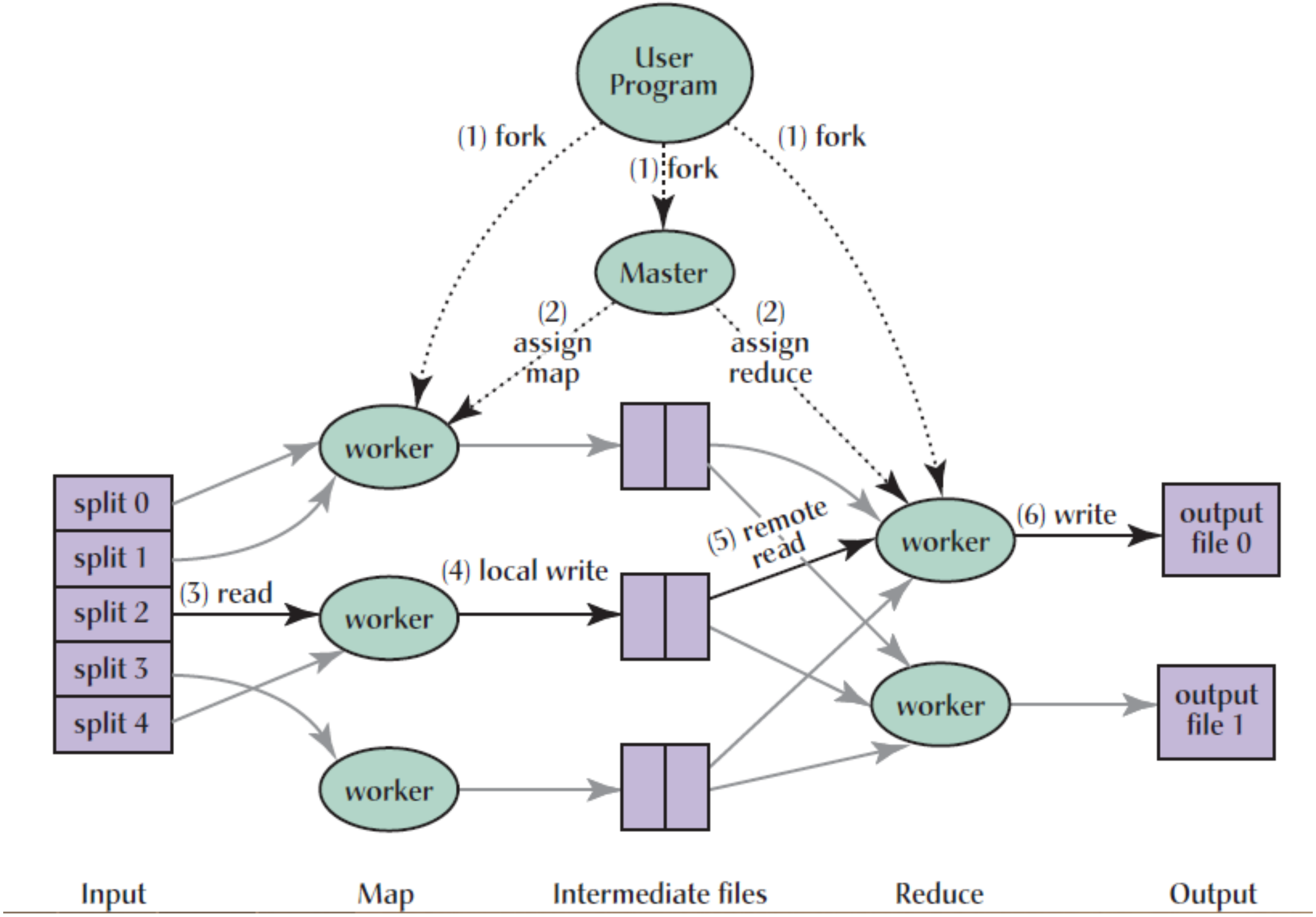


Re-replicating Missing Replicas

- Missing Heartbeat Messages signify dead DataNode
- NameNode will notice and instruct other DataNode to replicate data to new DataNode

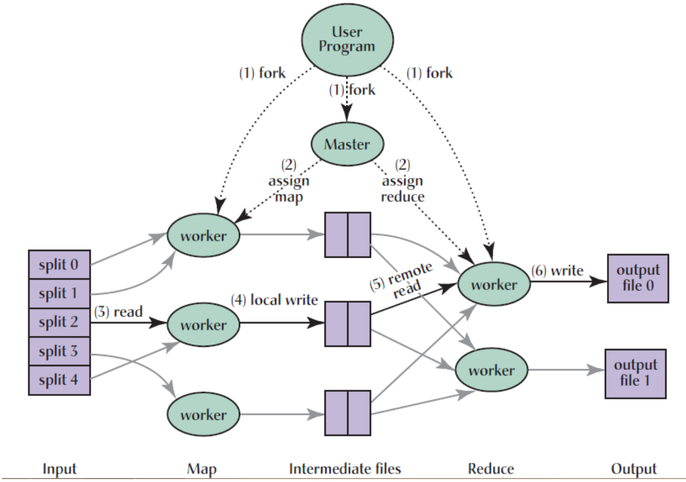


MapReduce Execution Overview



MapReduce Execution Overview

- The MapReduce library in the user program first splits the input files into M pieces of typically 16MB-64MB/piece. It then starts up many copies of the program on a cluster of machines.



- One of the the copies of the program is special: the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one either an M or R task.

- A worker who is assigned a map task reads the content of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the users-defined Map function. The intermediate K/V pairs produced are **buffered in memory**.

MapReduce Execution Overview

- Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to master, who is responsible for forwarding these locations to the reduce workers.
- When a reduce worker gets the location from master, it uses remote calls to read the buffered data from the disks. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so all of the same occurrences are grouped together. (note: if the amount of intermediate data is too large to fit in memory, an external sort is used)

MapReduce Execution Overview

- The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and corresponding set of intermediate values to the user’s Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.

- When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.

- After successful completion, the output of the MapReduce execution is available in the R output files.

Exercise

- Assume that we have two groups of files, i.e., group A and group B. In each group, each file contains many lines of IDs. We assume that all IDs in group A are different from each other. IDs in group B may have repetition.
- Our goal is to calculate the set difference between IDs in group A and IDs in group B.
- Write brief pseudo-code for Map and Reduce workers, including the (key, value) pairs of the input and output.

- Hints: (1) The set difference between two sets A and B means the set that consists of the elements of A which are not elements of B. (2) E.g., group A contains files a1 = {ID1, ID2, ID3} and a2 = {ID4, ID5, ID6}. Group B contains files b1 = {ID1, ID5, ID9} and b2 = {ID2, ID4, ID8}. Then, the set difference between A and B is {ID3, ID6}.

Solution 1

```
Map(key, value) {
// key:     file name in group A or B
// value: file content
for each row in file_name:
      emit(ID, A); // or emit(ID, B); intermediate KV pairs
}

Reduce(key, values) {
// key: ID
If values contain and only contain A:
    emit(ID);
}
```

Solution 2: Map

```
Map(key, value) {
// key:    file name in group B and file name in group A
// value: content in files
repetition = {};
for each row in file_name_groupB {
    if ID == any IDs in set(file_name_groupA):
        repetition.append(ID);
}
emit(file_name_groupA, repetition); // intermediate KV pairs
}
```

Solution 2: Reduce

```
Reduce(key, values) {
// key: file_name_groupA
// values: remain number for each file in group A
repetition_set = {};
for ID in values:
    if ID not in repetition_set:
          repetition_set.append(ID);
emit( setdiff(file_name_groupA, repetition_set) );
}
```