MapReduce

Dr. Rajiv Misra
Associate Professor
Dept. of Computer Science & Engg.
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Preface

Content of this Lecture:

- In this lecture, we will discuss the '**MapReduce paradigm'** and its internal working and implementation overview.

- We will also see many examples and different applications of MapReduce being used, and look into how the '**scheduling and fault tolerance'** works inside MapReduce.

Introduction

- **MapReduce** is a programming model and an associated implementation for **processing and generating large data sets.**

- Users specify a ***map*** function that processes a key/value pair to generate a set of intermediate key/value pairs, and a ***reduce*** function that merges all intermediate values associated with the same intermediate key.

- Many real world tasks are expressible in this model.

Contd...

- Programs written in this functional style **are automatically parallelized and executed** on a large cluster of commodity machines.
- The **run-time system** takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication.
- This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.
- A **typical MapReduce computation processes** many terabytes of data on thousands of machines. Hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

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

- Also known as Name Nodes in HDFS
- Stores metadata
- Might be replicated

Client library for file access

- Talks to master to find chunk servers
- Connects directly to chunkservers to access data

Motivation for Map Reduce (Why)

- **Large-Scale Data Processing**
  - Want to use 1000s of CPUs
  - But don’t want hassle of managing things
- **MapReduce Architecture provides**
  - Automatic parallelization & distribution
  - Fault tolerance
  - I/O scheduling
  - Monitoring & status updates

MapReduce Paradigm

What is MapReduce?

- Terms are borrowed from Functional Language (e.g., Lisp)

Sum of squares:

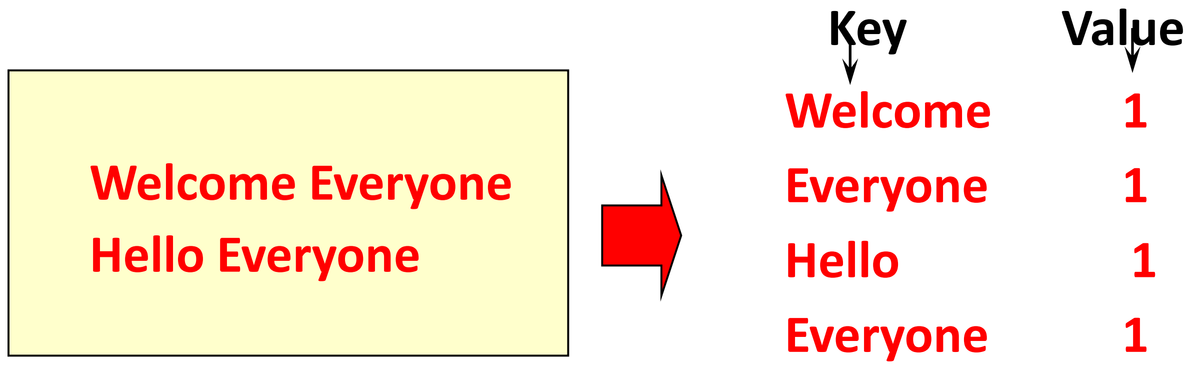
- **(map square ‘(1 2 3 4))**
  - **Output: (1 4 9 16)**

  [processes each record sequentially and independently]
- **(reduce + ‘(1 4 9 16))**
  - **(+ 16 (+ 9 (+ 4 1)))**
  - **Output: 30**

  [processes set of all records in batches]
- Let’s consider a sample application: **Wordcount**
  - You are given a **huge** dataset (e.g., Wikipedia dump or all of Shakespeare’s works) and asked to list the count for each of the words in each of the documents therein

Map

- Process individual records to generate intermediate key/value pairs.

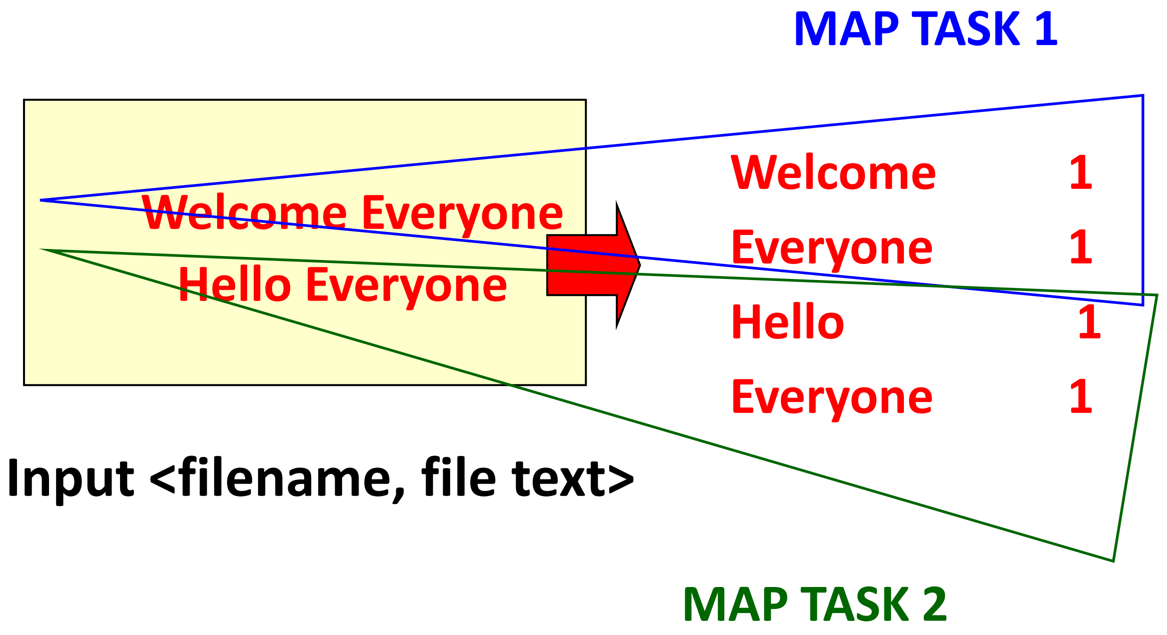


Key	Value
Welcome	1
Everyone	1
Hello	1
Everyone	1

Input <filename, file text>

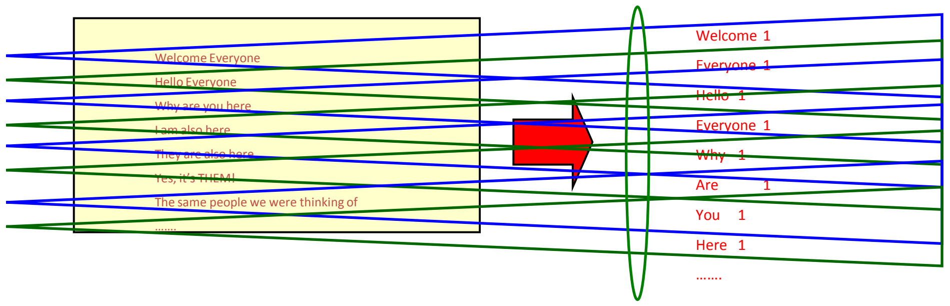
Map

- **Parallelly** Process individual records to generate intermediate key/value pairs.



Map

- **Parallelly** Process **a large number** of individual records to generate intermediate key/value pairs.



Input <filename, file text>

MAP TASKS

Reduce

- Reduce processes and merges all intermediate values associated per **key**

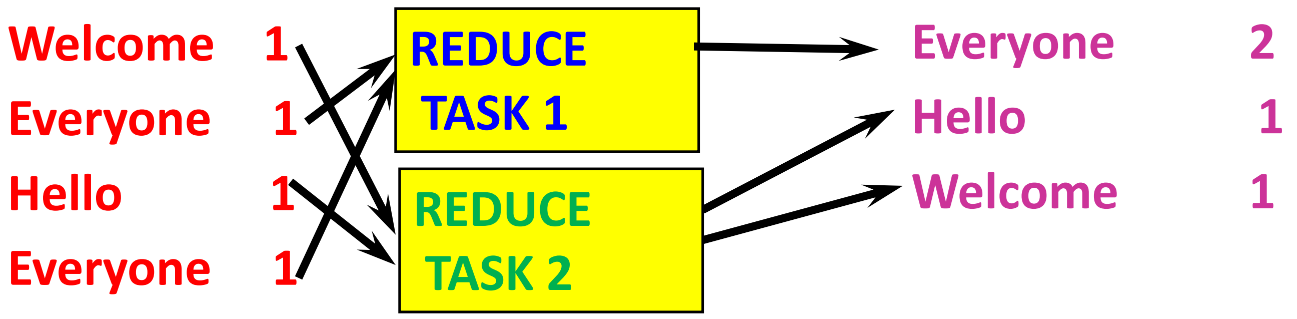
Welcome	1
Everyone	1
Hello	1
Everyone	1

$\Rightarrow$

Key	Value
Everyone	2
Hello	1
Welcome	1

Reduce

- Each key assigned to one Reduce
- Parallelly Processes and merges all intermediate **values by partitioning keys**



- Popular: ***Hash partitioning***, *i.e.*, key is assigned to
  – reduce # = hash(key)%number of reduce tasks

Programming Model

- The computation takes a set of **input key/value pairs**, and produces a set of **output key/value pairs.**

- The user of the MapReduce library expresses the computation as two functions:

(i) The Map

(ii) The Reduce

(i) Map Abstraction

- Map, written by the user, takes an input pair and produces a set of **intermediate key/value pairs.**
- The MapReduce library groups together all intermediate values associated with the same **intermediate key *'I'*** and passes them to the **Reduce function.**

(ii) Reduce Abstraction

- **The Reduce function,** also written by the user, accepts an **intermediate key *'I'*** and a set of values for that key.

- It merges together these values to form a **possibly smaller set of values.**

- Typically just **zero or one output value** is produced per Reduce invocation. The **intermediate values** are supplied to the user's reduce function via **an iterator.**

- This allows us to **handle lists of values** that are too large to fit in memory.

Map-Reduce Functions for Word Count

```
map(key, value):
// key: document name; value: text of 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)
```

Map-Reduce Functions

- **Input:** a set of key/value pairs
- User supplies two functions:

  map(k,v) → list(k1,v1)

  reduce(k1, list(v1)) → v2
- (k1,v1) is an intermediate key/value pair
- **Output** is the set of (k1,v2) pairs

MapReduce Applications

Applications

- Here are a few simple applications of interesting programs that can be easily expressed as **MapReduce computations.**
- **Distributed Grep:** The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.
- **Count of URL Access Frequency:** The map function processes logs of web page requests and outputs (URL; 1). The reduce function adds together all values for the same URL and emits a (URL; total count) pair.
- **ReverseWeb-Link Graph:** The map function outputs (target; source) pairs for each link to a target URL found in a page named source. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: (target; list(source))

Contd...

- **Term-Vector per Host:** A term vector summarizes the most important words that occur in a document or a set of documents as a list of (word; frequency) pairs.

- The map function emits a (hostname; term vector) pair for each input document (where the hostname is extracted from the URL of the document).

- The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final (hostname; term vector) pair

Contd...

- **Inverted Index:** The map function parses each document, and emits a sequence of (word; document ID) pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a (word; list(document ID)) pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

- **Distributed Sort:** The map function extracts the key from each record, and emits a (key; record) pair. The reduce function emits all pairs unchanged.

Applications of MapReduce

(1) Distributed Grep:

- Input: large set of files
- Output: lines that match pattern

- Map – ***Emits a line if it matches the supplied pattern***

- Reduce – ***Copies the intermediate data to output***

Applications of MapReduce

(2) Reverse Web-Link Graph:

- **Input:** Web graph: tuples (a, b) where (page a → page b)

- **Output:** For each page, list of pages that link *to* it

- Map – ***process web log and for each input <source, target>, it outputs <target, source>***
- Reduce - ***emits <target, list(source)>***

Applications of MapReduce

(3) Count of URL access frequency:

- Input: Log of accessed URLs, e.g., from proxy server
- Output: For each URL, % of total accesses for that URL

- Map – ***Process web log and outputs <URL, 1>***
- Multiple Reducers - ***Emits <URL, URL_count>***

(So far, like Wordcount. But still need %)

- Chain another MapReduce job after above one
- Map – ***Processes <URL, URL_count>*** and outputs ***<1, (<URL, URL_count>)>***
- 1 Reducer – Does two passes. In first pass, sums up all ***URL_count's*** to calculate overall_count. In second pass calculates %'s

Emits multiple <URL, URL_count/overall_count>

Applications of MapReduce

(4) Map task's output is sorted (e.g., quicksort)
Reduce task's input is sorted (e.g., mergesort)

Sort

- Input: Series of (key, value) pairs
- Output: Sorted <value>s

- Map – ***<key, value> → <value, _> (identity)***
- Reducer – ***<key, value> → <key, value> (identity)***
- Partitioning function – partition keys across reducers based on **ranges** (can't use hashing!)
  - Take data distribution into account to balance reducer tasks

MapReduce Scheduling

Programming MapReduce

Externally: For **user**

1. Write a Map program (short), write a Reduce program (short)
2. Specify number of Maps and Reduces (parallelism level)
3. Submit job; wait for result
4. Need to know very little about parallel/distributed programming!

Internally: For the Paradigm and Scheduler

1. Parallelize Map
2. Transfer data from Map to Reduce (**shuffle data**)
3. Parallelize Reduce
4. Implement Storage for Map input, Map output, Reduce input, and Reduce output

(Ensure that no Reduce starts before all Maps are finished. That is, ensure **the *barrier*** between the Map phase and Reduce phase)

Inside MapReduce

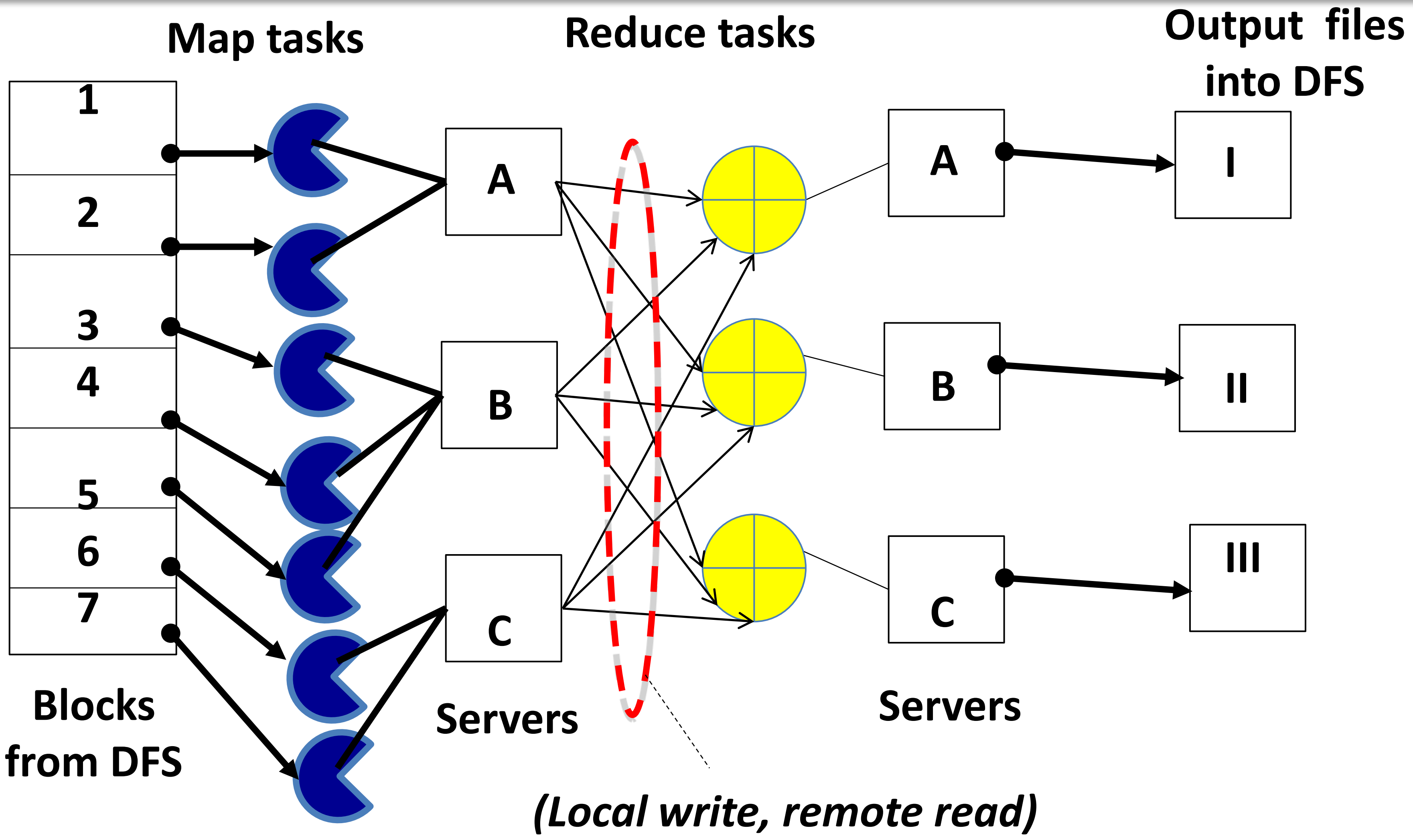
For the cloud:

1. Parallelize Map: **easy!** each map task is independent of the other!
   - All Map output records with same key assigned to same Reduce
2. Transfer data from Map to Reduce:
   - Called Shuffle data
   - All Map output records with same key assigned to same Reduce task
   - use **partitioning function, e.g., hash(key)%number of reducers**
3. Parallelize Reduce: **easy!** each reduce task is independent of the other!
4. Implement Storage for Map input, Map output, Reduce input, and Reduce output
   - Map input: from **distributed file system**
   - Map output: to local disk (at Map node); uses **local file system**
   - Reduce input: from (multiple) remote disks; uses local file systems
   - Reduce output: to distributed file system

local file system = Linux FS, etc.

distributed file system = GFS (Google File System), HDFS (Hadoop Distributed File System)

Internal Workings of MapReduce



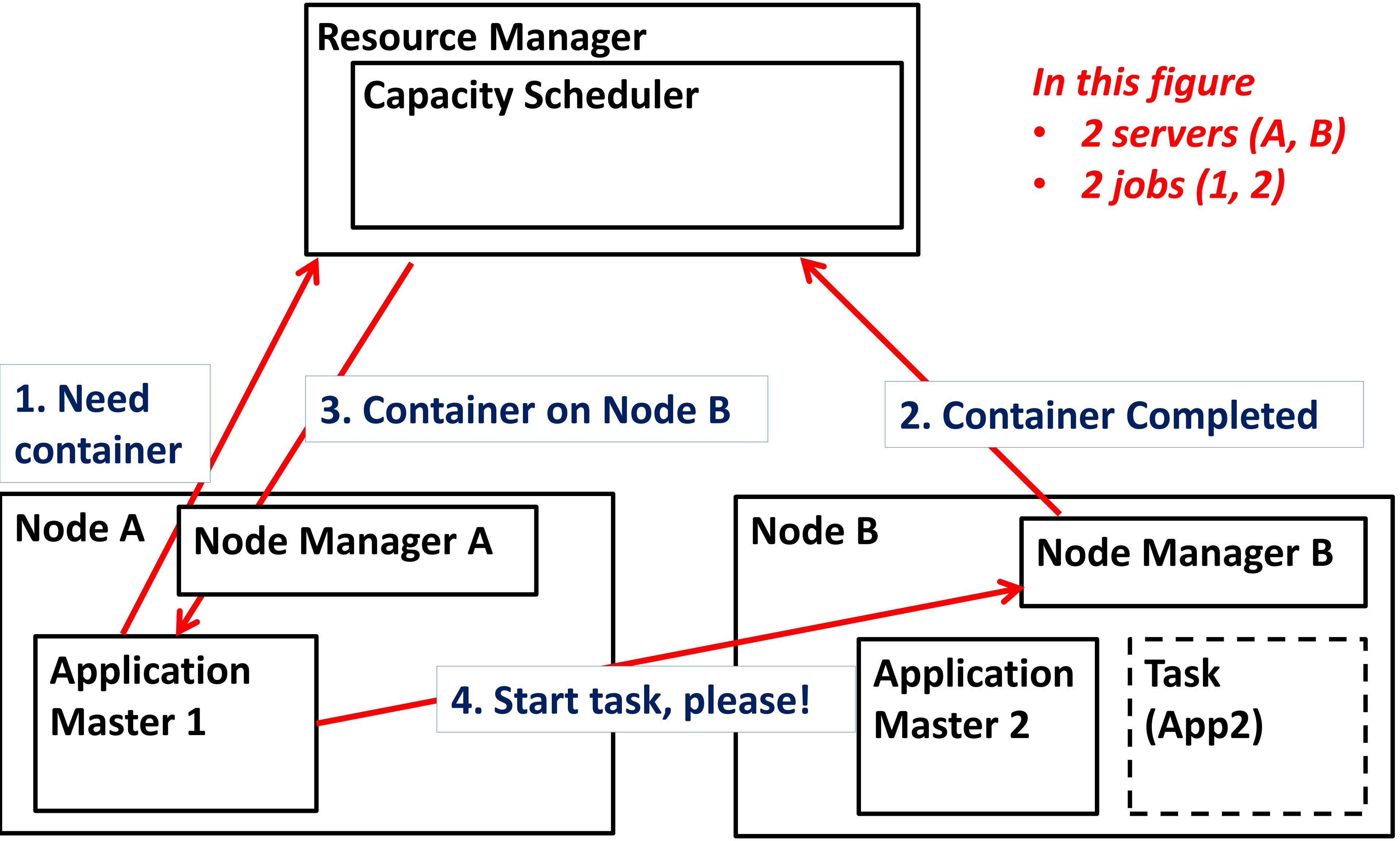
Resource Manager (assigns maps and reduces to servers)

The YARN Scheduler

- Used underneath Hadoop 2.x +
- YARN = Yet Another Resource Negotiator
- Treats each server as a collection of ***containers***
  - Container = fixed CPU + fixed memory

- Has 3 main components
  - **Global *Resource Manager (RM)***
    - Scheduling
  - **Per-server *Node Manager (NM)***
    - Daemon and server-specific functions
  - **Per-application (job) *Application Master (AM)***
    - Container negotiation with RM and NMs
    - Detecting task failures of that job

YARN: How a job gets a container



In this figure
- *2 servers (A, B)*
- *2 jobs (1, 2)*

MapReduce Fault-Tolerance

Fault Tolerance

- Server Failure
  - NM heartbeats to RM
    - If server fails, RM lets all affected AMs know, and AMs take appropriate action
  - NM keeps track of each task running at its server
    - If task fails while in-progress, mark the task as idle and restart it
  - AM heartbeats to RM
    - On failure, RM restarts AM, which then syncs up with its running tasks
- RM Failure
  - Use old checkpoints and bring up secondary RM
- Heartbeats also used to piggyback container requests
  - Avoids extra messages

Slow Servers

Slow tasks are called **Stragglers**

- The slowest task slows the entire job down (why?)
- Due to Bad Disk, Network Bandwidth, CPU, or Memory
- Keep track of "progress" of each task (% done)
- Perform backup **(replicated)** execution of straggler tasks
  - A task considered done when its first replica complete called **Speculative Execution.**

Locality

- **Locality**
  - Since cloud has hierarchical topology **(e.g., racks)**
  - GFS/HDFS stores 3 replicas of each of chunks (e.g., 64 MB in size)
    - Maybe on different racks, e.g., 2 on a rack, 1 on a different rack
  - **Mapreduce attempts to schedule a map task on**
    1. a machine that contains a replica of corresponding input data, or failing that,
    2. on the same rack as a machine containing the input, or failing that,
    3. Anywhere

Implementation Overview

Implementation Overview

- Many different implementations of the MapReduce interface are possible. The right choice depends on the environment.

- **For example,** one implementation may be suitable for a small shared-memory machine, another for a large NUMA multi-processor, and yet another for an even larger collection of networked machines.

- Here we describes an implementation targeted to the computing environment in wide use at Google: large clusters of commodity PCs connected together with switched Ethernet.

Contd...

(1) Machines are typically dual-processor x86 processor running Linux, with 2-4 GB of memory per machine.

(2) Commodity networking hardware is used . Typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in overall bisection bandwidth.

(3) A cluster consists of hundreds or thousands of machines, and therefore machine failures are common.

(4) Storage is provided by inexpensive IDE disks attached directly to individual machines.

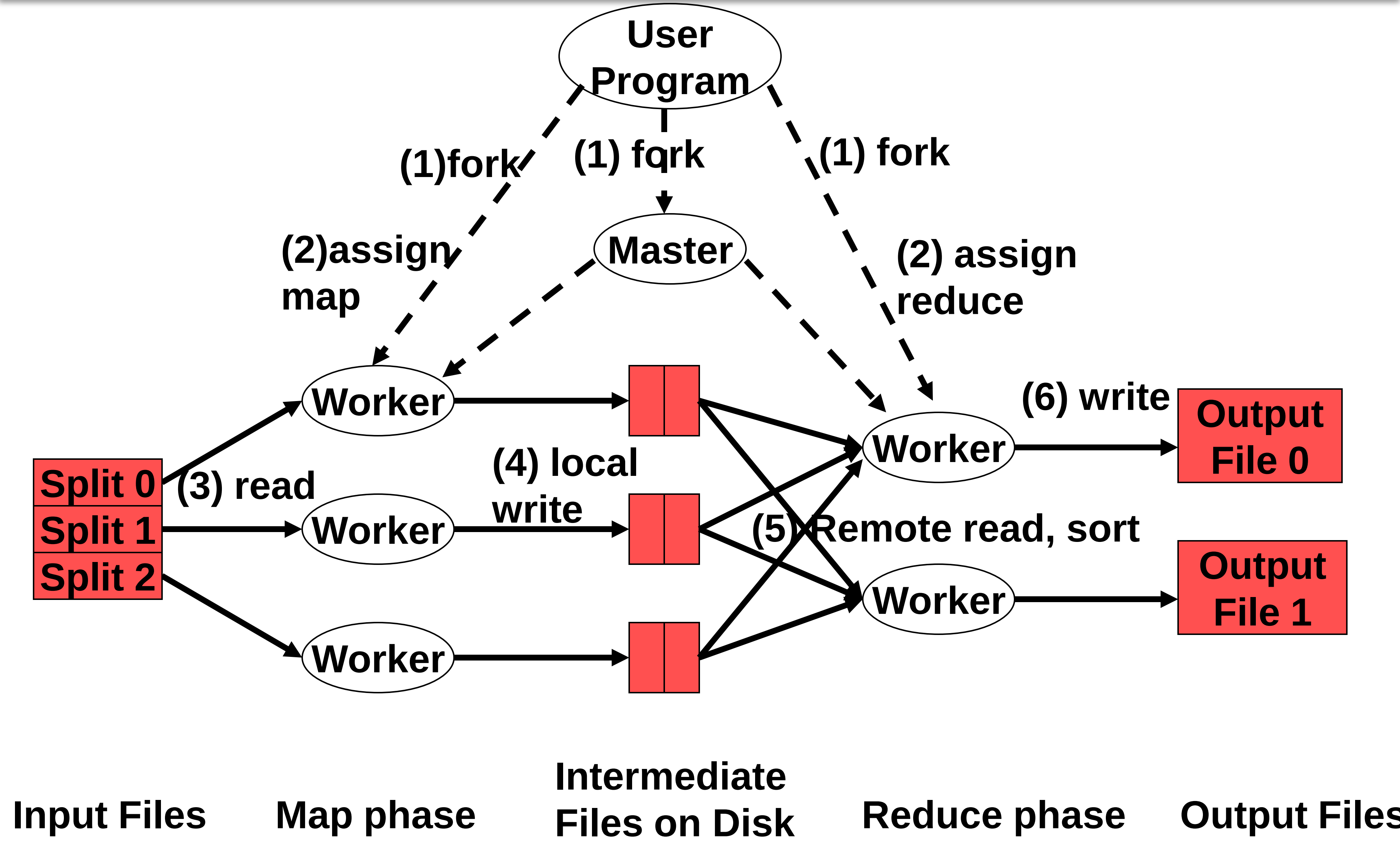
(5) Users submit jobs to a scheduling system. Each job consists of a set of tasks, and is mapped by the scheduler to a set of available machines within a cluster.

Distributed Execution Overview

- The **Map invocations** are distributed across multiple machines by automatically partitioning the input data into a set of M splits.
- The input splits can be processed in parallel by different machines.

- **Reduce invocations** are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., hash(key) mod R).
- The number of partitions (R) and the partitioning function are specified by the user.
- Figure 1 shows the overall flow of a MapReduce operation.

Distributed Execution Overview



Sequence of Actions

When the user **program calls the MapReduce function**, the following **sequence of actions** occurs:

1. The MapReduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece. It then starts up many copies of the program on a cluster of machines.
2. One of 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 a map task or a reduce task.
3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.

Contd...

4. 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 the master, who is responsible for forwarding these locations to the reduce workers.

5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together.

- The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.

Contd...

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the 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.

7. 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.

Contd...

- After successful completion, the output of the mapreduce execution is available in the R output files (one per reduce task, with file names as specified by the user).

- Typically, users do not need to combine these R output files into one file- they often pass these files as input to another MapReduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

Master Data Structures

- The master keeps several data structures. For each map task and reduce task, it stores the **state (idle, in-progress, or completed),** and the identity of the worker machine **(for non-idle tasks).**

- The master is the conduit through which the location of intermediate le regions is propagated from map tasks to reduce tasks. Therefore, for each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task.

- Updates to this location and size information are received as map tasks are completed. The information is pushed incrementally to workers that have in-progress reduce tasks.

Fault Tolerance

- Since the MapReduce library is designed to help process very large amounts of data using hundreds or thousands of machines, the library must tolerate machine failures gracefully.
- **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
- **Master failure**
  - MapReduce task is aborted and client is notified

Locality

- Network bandwidth is a relatively scarce resource in the computing environment. We can conserve network bandwidth by taking advantage of the fact that the input data (managed by GFS) is stored on the local disks of the machines that make up our cluster.
- GFS divides each file into 64 MB blocks, and stores several copies of each block (typically 3 copies) on different machines.
- The MapReduce master takes the location information of the input les into account and attempts to schedule a map task on a machine that contains a replica of the corresponding input data. Failing that, it attempts to schedule a map task near a replica of that task's input data (e.g., on a worker machine that is on the same network switch as the machine containing the data).
- When running large MapReduce operations on a significant fraction of the workers in a cluster, most input data is read locally and consumes no network bandwidth.

Task Granularity

- The **Map phase is subdivided into M pieces and the reduce phase into R pieces.**
- Ideally, M and R should be much larger than the number of worker machines.
- Having each worker perform many different tasks improves dynamic load balancing, and also speeds up recovery when a worker fails: the many map tasks it has completed can be spread out across all the other worker machines.
- There are practical bounds on how large M and R can be, since the master must make **O(M + R) scheduling decisions** and keeps **O(M * R) state in memory.**
- Furthermore, R is often constrained by users because the output of each reduce task ends up in a separate output file.

Partition Function

- Inputs to map tasks are created by contiguous splits of input file
- For reduce, we need to ensure that records with the same intermediate key end up at the same worker
- System uses a default partition function e.g., **hash(key) mod R**
- Sometimes useful to override
  - E.g., **hash(hostname(URL)) mod R** ensures URLs from a host end up in the same output file

Ordering Guarantees

- **It is guaranteed that within a given partition, the intermediate key/value pairs are processed in increasing key order.**

- **This ordering guarantee makes it easy to generate a sorted output file per partition, which is useful when the output file format needs to support efficient random access lookups by key, or users of the output and it convenient to have the data sorted.**

Combiners Function (1)

- In some cases, there is significant repetition in the intermediate keys produced by each map task, and the user specified Reduce function is commutative and associative.

- **A good example of this is the word counting example. Since word frequencies tend to follow a Zipf distribution, each map task will produce hundreds or thousands of records of the form <the, 1>.**

- All of these counts will be sent over the network to a single reduce task and then added together by the Reduce function to produce one number. We allow the user to specify an optional Combiner function that does partial merging of this data before it is sent over the network.

Combiners Function (2)

- The Combiner function is executed on each machine that performs a map task.
- Typically the same code is used to implement both the combiner and the reduce functions.
- The only difference between a reduce function and a combiner function is how the MapReduce library handles the output of the function.
- The output of a reduce function is written to the final output file. The output of a combiner function is written to an intermediate le that will be sent to a reduce task.
- Partial combining significantly speeds up certain classes of MapReduce operations.

MapReduce Examples

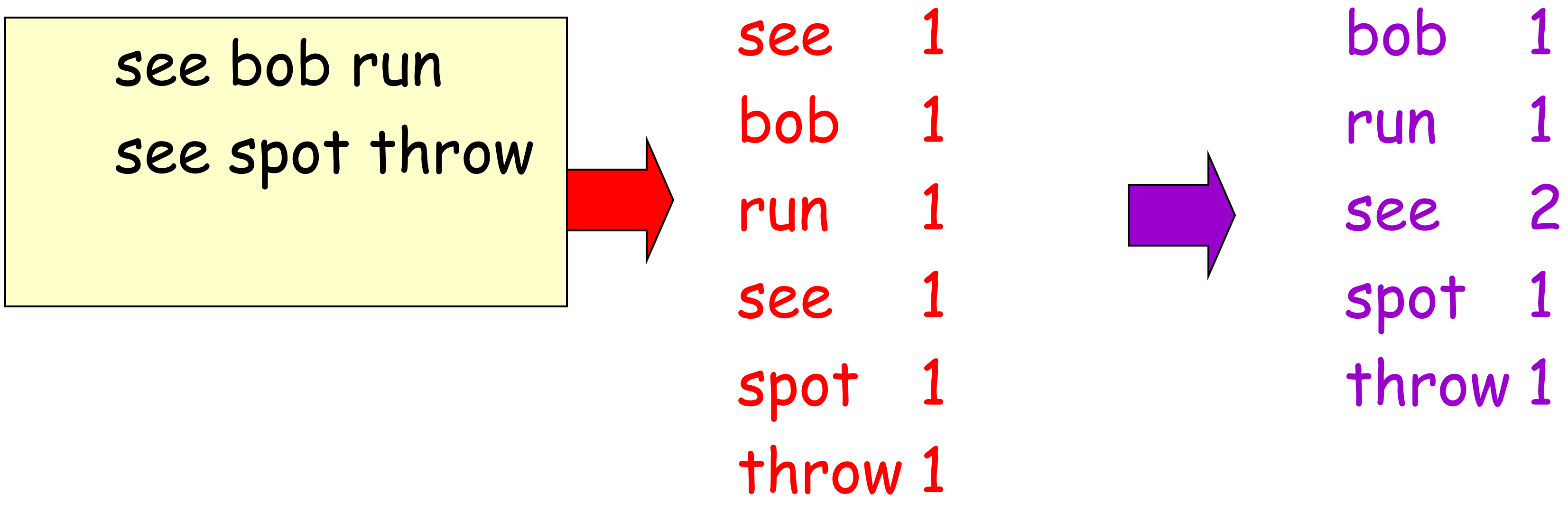
Example: 1 Word Count using MapReduce

```
map(key, value):
// key: document name; value: text of 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)
```

Count Illustrated

```
map(key=url, val=contents):
 For each word w in contents, emit (w, "1")
reduce(key=word, values=uniq_counts):
    Sum all "1"s in values list
    Emit result "(word, sum)"
```



see bob run
see spot throw

see	1
bob	1
run	1
see	1
spot	1
throw	1

bob	1
run	1
see	2
spot	1
throw	1

Example 2: Counting words of different lengths

- **The map function takes a value and outputs key:value pairs.**

- For instance, if we define a map function that takes a string and outputs the length of the word as the key and the word itself as the value then

- map(steve) would return 5:steve and
- map(savannah) would return 8:savannah.

This allows us to run the map function against values in parallel and provides a huge advantage.

Example 2: Counting words of different lengths

Before we get to the reduce function, the mapreduce framework groups all of the values together by key, so if the map functions output the following **key:value pairs:**

3 : the
3 : and
3 : you
4 : then
4 : what
4 : when
5 : steve
5 : where
8 : savannah
8 : research

They get grouped as:

3 : [the, and, you]
4 : [then, what, when]
5 : [steve, where]
8 : [savannah, research]

Example 2: Counting words of different lengths

- Each of these lines would then be passed as an argument to the reduce function, which accepts a key and a list of values.
- In this instance, we might be trying to figure out how many words of certain lengths exist, so our reduce function will just count the number of items in the list and output the key with the size of the list, like:

3 : 3

4 : 3

5 : 2

8 : 2

Example 2: Counting words of different lengths

- The reductions can also be done in parallel, again providing a huge advantage. We can then look at these final results and see that there were only two words of length 5 in the corpus, etc...

- **The most common example of mapreduce is for counting the number of times words occur in a corpus.**

Example 3: Finding Friends

- Facebook has a list of friends (note that friends are a bi-directional thing on Facebook. If I'm your friend, you're mine).
- They also have lots of disk space and they serve hundreds of millions of requests everyday. They've decided to pre-compute calculations when they can to reduce the processing time of requests. **One common processing request is the "You and Joe have 230 friends in common" feature.**
- When you visit someone's profile, you see a list of friends that you have in common. This list doesn't change frequently so it'd be wasteful to recalculate it every time you visited the profile (sure you could use a decent caching strategy, but then we wouldn't be able to continue writing about mapreduce for this problem).
- We're going to use mapreduce so that we can calculate everyone's common friends once a day and store those results. Later on it's just a quick lookup. We've got lots of disk, it's cheap.

Example 3: Finding Friends

- Assume the friends are stored as **Person->[List of Friends],** our friends list is then:

- A -> B C D
- B -> A C D E
- C -> A B D E
- D -> A B C E
- E -> B C D

Example 3: Finding Friends

For map(A -> B C D) :

(A B) -> B C D

(A C) -> B C D

(A D) -> B C D

For map(B -> A C D E) : (Note that A comes before B in the key)

(A B) -> A C D E

(B C) -> A C D E

(B D) -> A C D E

(B E) -> A C D E

Example 3: Finding Friends

For map(C -> A B D E) :

(A C) -> A B D E

(B C) -> A B D E

(C D) -> A B D E

(C E) -> A B D E

For map(D -> A B C E) :

(A D) -> A B C E

(B D) -> A B C E

(C D) -> A B C E

(D E) -> A B C E

And finally for map(E -> B C D):

(B E) -> B C D

(C E) -> B C D

(D E) -> B C D

Example 3: Finding Friends

- Before we send these key-value pairs to the reducers, we group them by their keys and get:

(A B) -> (A C D E) (B C D)
(A C) -> (A B D E) (B C D)
(A D) -> (A B C E) (B C D)
(B C) -> (A B D E) (A C D E)
(B D) -> (A B C E) (A C D E)
(B E) -> (A C D E) (B C D)
(C D) -> (A B C E) (A B D E)
(C E) -> (A B D E) (B C D)
(D E) -> (A B C E) (B C D)

Example 3: Finding Friends

- Each line will be passed as an argument to a reducer.
- The **reduce function will simply intersect the lists of values** and output the same key with the result of the intersection.
- For example, **reduce((A B) -> (A C D E) (B C D))**

  will **output (A B) : (C D)**
- **and means that friends A and B have C and D as common friends.**

Example 3: Finding Friends

- The result after reduction is:
- (A B) -> (C D)
- (A C) -> (B D)
- (A D) -> (B C)
- (B C) -> (A D E)
- (B D) -> (A C E)
- (B E) -> (C D)
- (C D) -> (A B E)
- (C E) -> (B D)
- (D E) -> (B C)

Now when D visits B's profile, we can quickly look up (B D) and see that they have three friends in common, (A C E).

Reading

Jeffrey Dean and Sanjay Ghemawat,

“MapReduce: Simplified Data Processing on Large Clusters”

http://labs.google.com/papers/mapreduce.html

Conclusion

- **The MapReduce programming model has been successfully used at Google for many different purposes.**

- The model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality optimization, and load balancing.

- A large variety of problems are easily expressible as MapReduce computations.

- **For example, MapReduce is used for the generation of data for Google's production web search service, for sorting, for data mining, for machine learning, and many other systems.**

Conclusion

- Mapreduce uses **parallelization + aggregation** to schedule applications across clusters

- **Need to deal with failure**

- Plenty of ongoing research work in **scheduling and fault-tolerance for Mapreduce and Hadoop.**