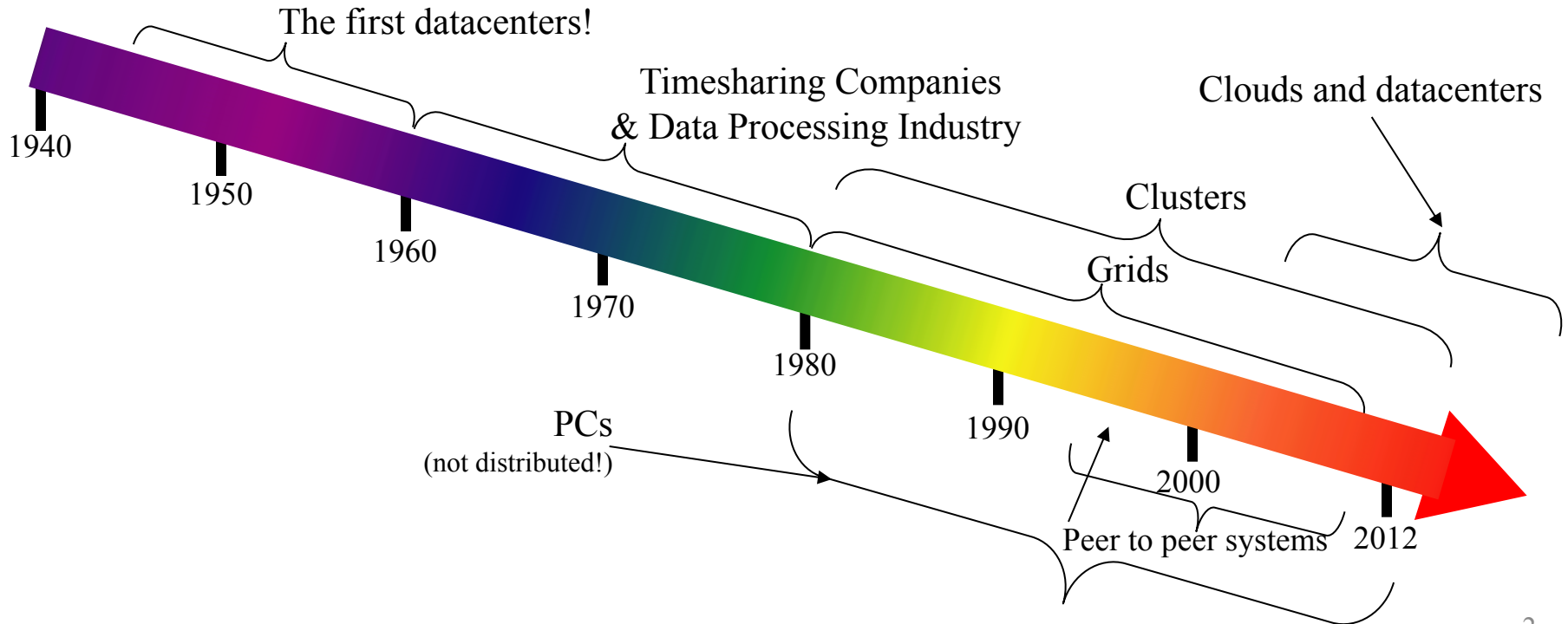


CS 425 / ECE 428  
Distributed Systems  
Fall 2018

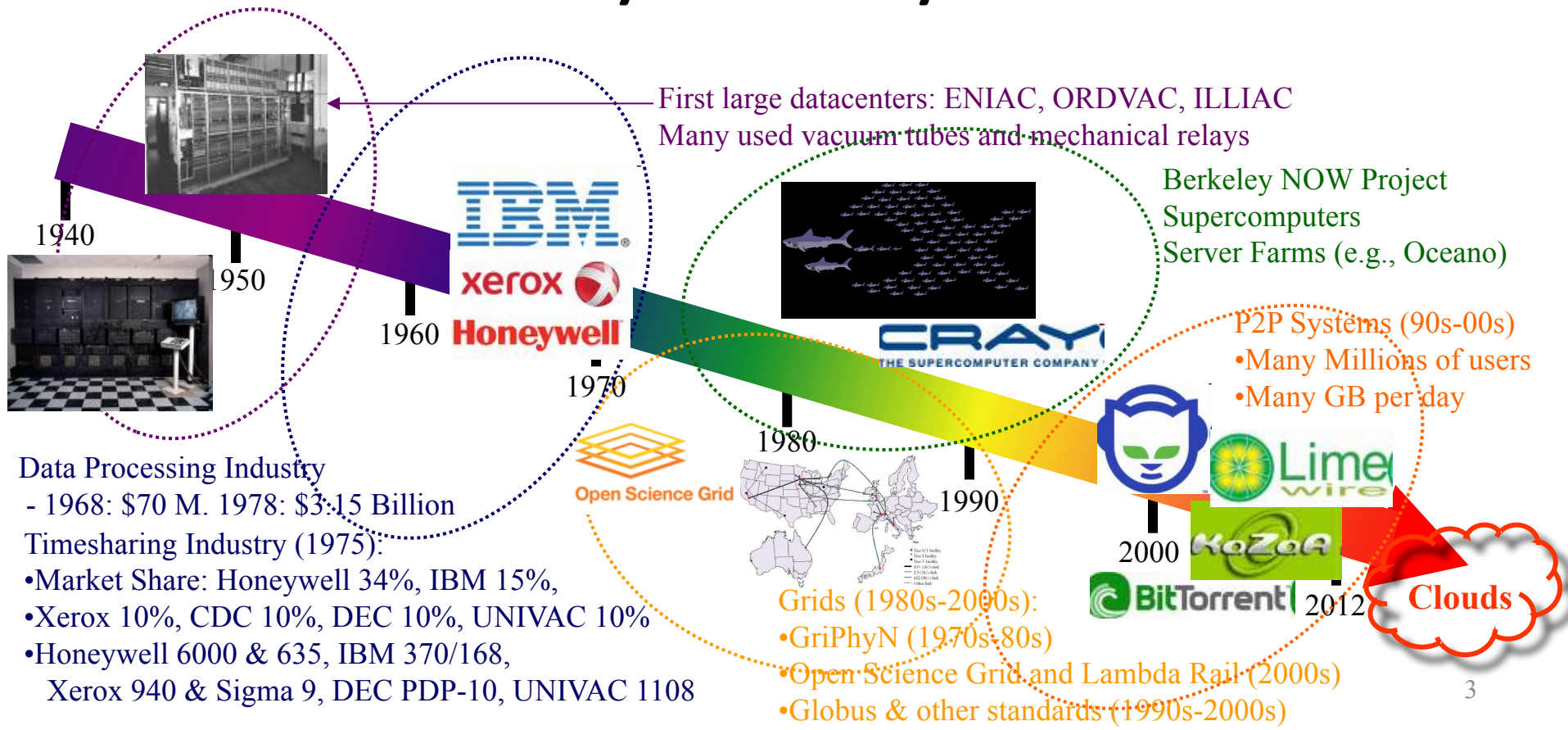
Indranil Gupta (Indy)

*Lecture 4: Mapreduce and Hadoop*

# “A Cloudy History of Time”



# “A Cloudy History of Time”



Data Processing Industry  
- 1968: \$70 M. 1978: \$3.15 Billion

Timesharing Industry (1975):  
 • Market Share: Honeywell 34%, IBM 15%,  
 • Xerox 10%, CDC 10%, DEC 10%, UNIVAC 10%  
 • Honeywell 6000 & 635, IBM 370/168,  
 Xerox 940 & Sigma 9, DEC PDP-10, UNIVAC 1108

# Four Features New in Today's Clouds

I. **Massive scale.**

II. **On-demand access:** Pay-as-you-go, no upfront commitment.

- And anyone can access it

III. **Data-intensive Nature:** What was MBs has now become TBs, PBs and XBs.

- Daily logs, forensics, Web data, etc.
- Humans have data numbness: Wikipedia (large) compressed is only about 10 GB!

IV. **New Cloud Programming Paradigms:** MapReduce/Hadoop, NoSQL/Cassandra/MongoDB and many others.

- High in accessibility and ease of programmability
- Lots of open-source

**Combination of one or more of these gives rise to novel and unsolved distributed computing problems in cloud computing.**

# What is MapReduce?

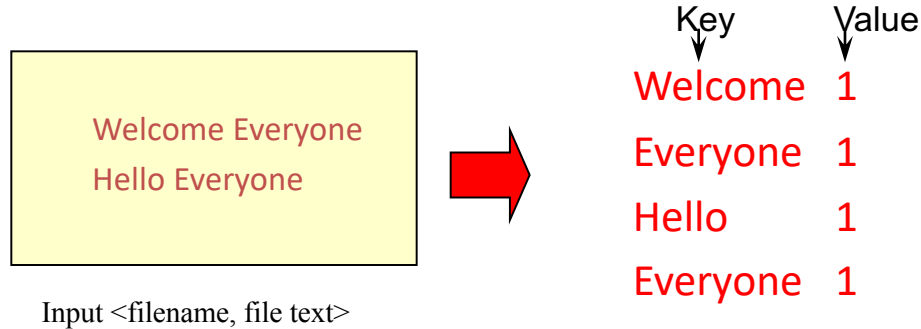
- Terms are borrowed from Functional Language (e.g., Lisp)

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

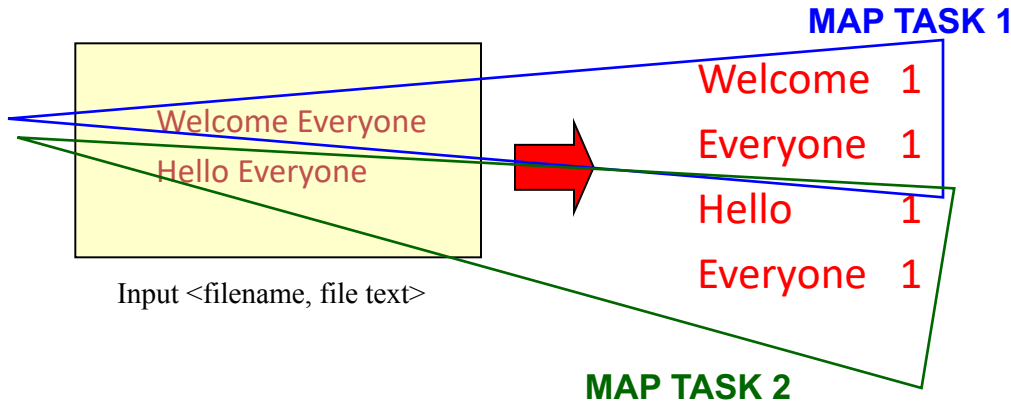
# Map

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



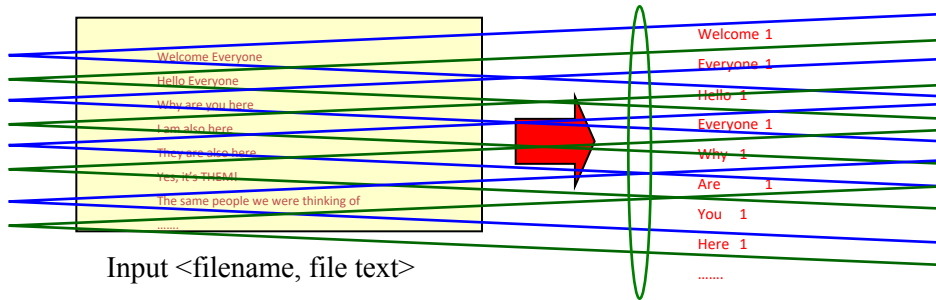
# Map

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



# Map

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

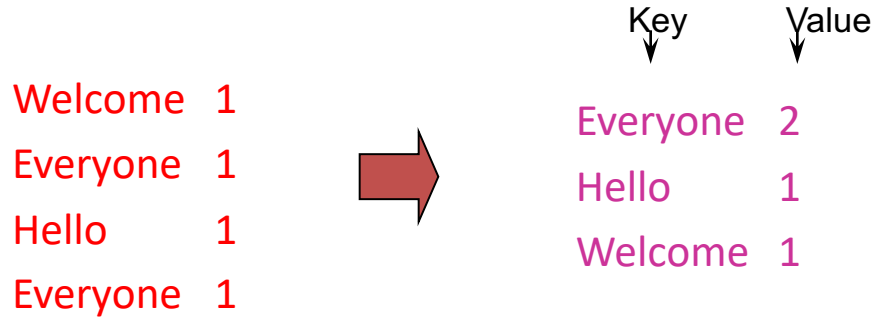


**MAP TASKS**



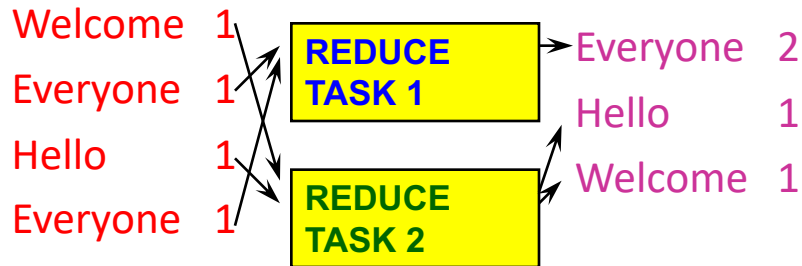
# Reduce

- Reduce processes and merges all intermediate values associated per key



# Reduce

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



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

# Hadoop Code - Map

```
public static class MapClass extends MapReduceBase implements
Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one =
        new IntWritable(1);
    private Text word = new Text();

    public void map( LongWritable key, Text value,
        OutputCollector<Text, IntWritable> output, Reporter reporter)
        // key is empty, value is the line
        throws IOException {
        String line = value.toString();
        StringTokenizer itr = new StringTokenizer(line);
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            output.collect(word, one);
        }
    }
}
// Source: http://developer.yahoo.com/hadoop/tutorial/module4.html#wordcount
```

# Hadoop Code - Reduce

```
public static class ReduceClass extends MapReduceBase implements
Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(
        Text key,
        Iterator<IntWritable> values,
        OutputCollector<Text, IntWritable> output,
        Reporter reporter)
        throws IOException {
            // key is word, values is a list of 1's
            int sum = 0;
            while (values.hasNext()) {
                sum += values.next().get();
            }
            output.collect(key, new IntWritable(sum));
        }
} // Source: http://developer.yahoo.com/hadoop/tutorial/module4.html#wordcount
```

# Hadoop Code - Driver

```
// Tells Hadoop how to run your Map-Reduce job
public void run (String inputPath, String outputPath)
    throws Exception {
    // The job. WordCount contains MapClass and Reduce.
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("mywordcount");
    // The keys are words
    (strings) conf.setOutputKeyClass(Text.class);
    // The values are counts (ints)
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(MapClass.class);
    conf.setReducerClass(ReduceClass.class);
    FileInputFormat.addInputPath(
        conf, newPath(inputPath));
    FileOutputFormat.setOutputPath(
        conf, new Path(outputPath));
    JobClient.runJob(conf);
} // Source: http://developer.yahoo.com/hadoop/tutorial/module4.html#wordcount
```

# Some Applications of MapReduce

## 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*

# Some Applications of MapReduce

## (2)

### Reverse Web-Link Graph

- Input: Web graph: tuples (a, b) where (page a  $\rightarrow$  page b)
- Output: For each page, list of pages that link *to* it
  
- Map – *process web log and for each input  $\langle source, target \rangle$ , it outputs  $\langle target, source \rangle$*
- Reduce - *emits  $\langle target, list(source) \rangle$*

# Some 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*  $\langle URL, 1 \rangle$
- Multiple Reducers - *Emits*  $\langle URL, URL\_count \rangle$

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

- Chain another MapReduce job after above one
- Map – *Processes*  $\langle URL, URL\_count \rangle$  and outputs  $\langle 1, (\langle URL, URL\_count \rangle) \rangle$
- 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*  $\langle URL, URL\_count/overall\_count \rangle$



# Some 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 –  $\langle key, value \rangle \rightarrow \langle value, _ \rangle$  (*identity*)
- Reducer –  $\langle key, value \rangle \rightarrow \langle key, value \rangle$  (*identity*)
- Partitioning function – partition keys across reducers based on **ranges** (can't use hashing!)
  - Take data distribution into account to balance reducer tasks

# 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)

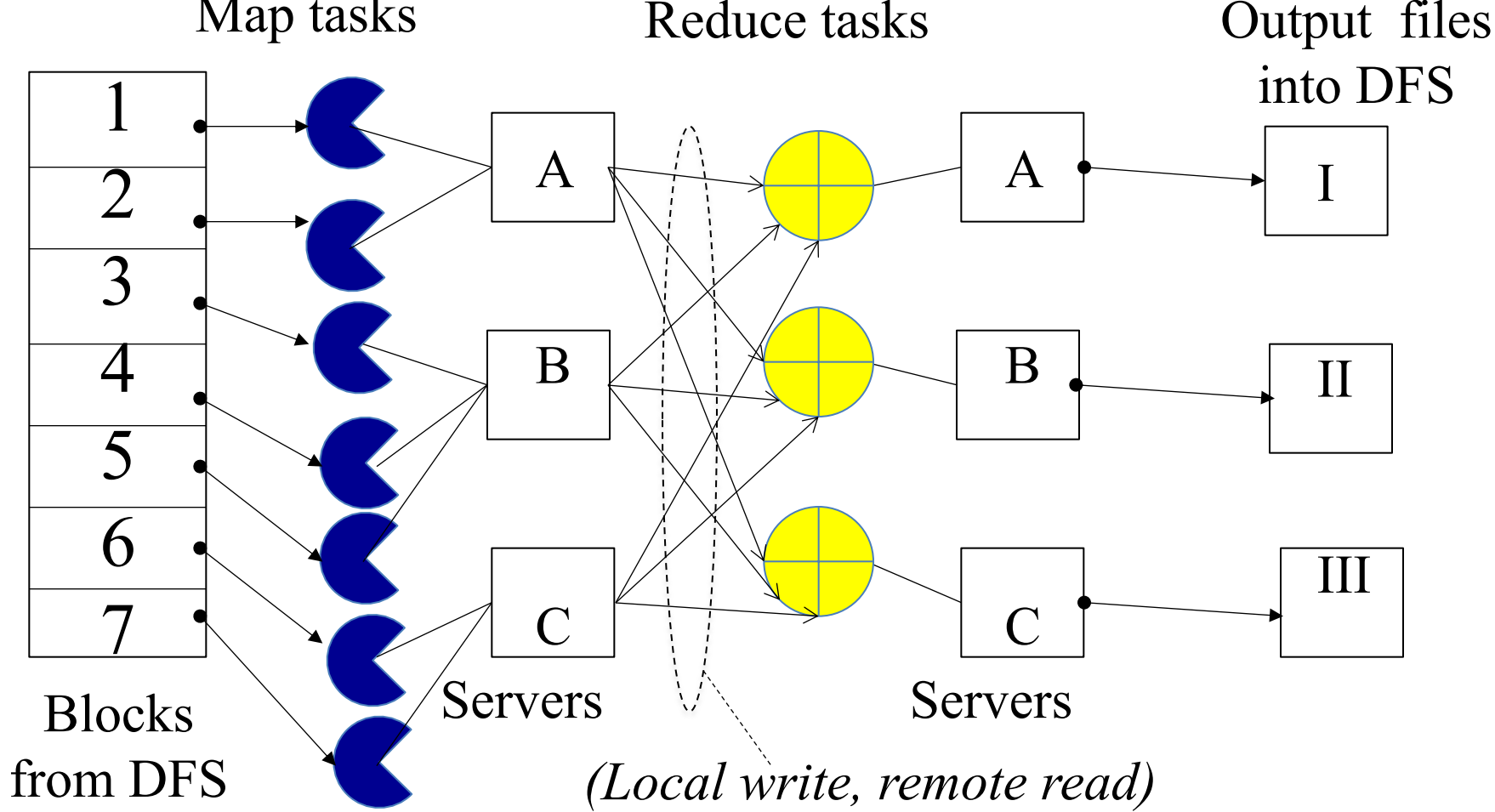
# Inside MapReduce

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.,  $\text{hash}(\text{key})\% \text{number of reducers}$**
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)



Resource Manager (assigns maps and reduces to servers)

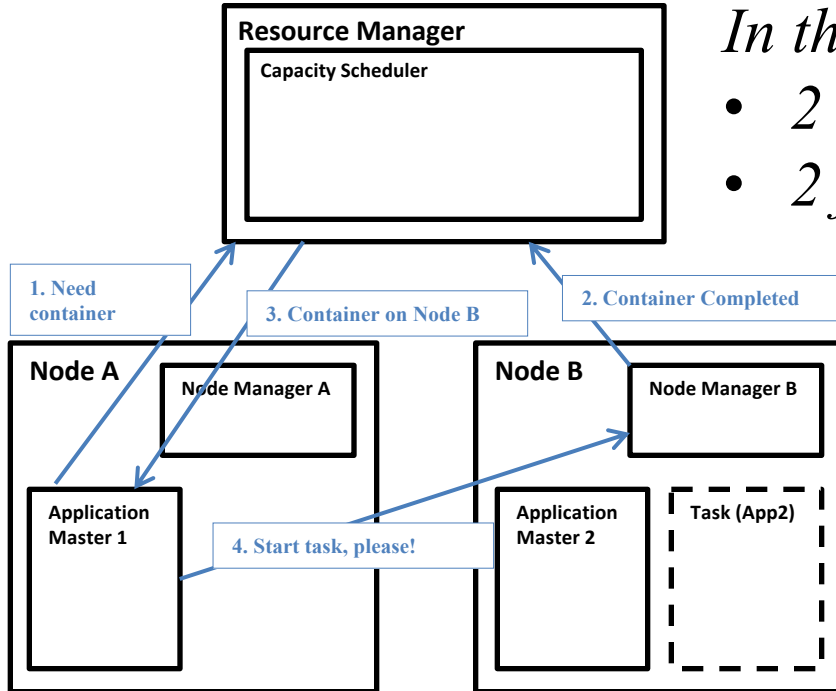
# The YARN Scheduler

- Used underneath Hadoop 2.x +
- YARN = Yet Another Resource Negotiator
- Treats each server as a collection of *containers*
  - Container = fixed CPU + fixed memory (think of Linux cgroups, but even more lightweight)
- 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)



# Fault Tolerance

- Server Failure
  - NM heartbeats to RM
    - If server fails: RM times out waiting for next heartbeat, 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 it 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 proactive backup (replicated) execution of some straggler tasks
  - A task considered done when its first replica complete (other replicas can then be killed)
  - Approach called **Speculative Execution**.

Barrier at the end  
of Map phase!



# Locality

- Locality
  - Since cloud has hierarchical topology (e.g., racks)
  - For server-fault-tolerance, GFS/HDFS stores 3 replicas of each of chunks (e.g., 64 MB in size)
    - For rack-fault-tolerance, 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
  - Note: The 2-1 split of replicas is intended to reduce bandwidth when writing file.
    - Using more racks does not affect overall Mapreduce scheduling performance

# That was Hadoop 2.x...

- Hadoop 3.x (new!) over Hadoop 2.x
  - Dockers instead of container
  - Erasure coding instead of 3-way replication
  - Multiple Namenodes instead of one (name resolution)
  - GPU support (for machine learning)
  - Intra-node disk balancing (for repurposed disks)
  - Intra-queue preemption in addition to inter-queue
  - (From <https://hortonworks.com/blog/hadoop-3-adds-value-hadoop-2/>)

# Mapreduce: Summary

- 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

# Announcements

- MP Groups **DUE TODAY 5 pm** (see course webpage).
  - **Hard deadline, as Engr-IT will create and assign VMs tomorrow!**
- Please fill out Student Survey by today (see course webpage).
- DO NOT
  - Change MP groups unless your partner has dropped
  - Leave your MP partner hanging: Both MP partners should contribute equally (we will ask!)
- MP1 due Sep 16<sup>th</sup>
  - VMs will be distributed soon (watch Piazza)
  - Demos will be Monday Sep 17<sup>th</sup> (schedule and details will be posted before that on Piazza)
- HW1 due Sep 25<sup>th</sup>
- Check Piazza often! It's where all the announcements are at!
- Waitlist form: open again as of today