Distributed Systems

CS425/ECE428

April 23 2021

Instructor: Radhika Mittal

Acknowledgements for some of the materials: Indy Gupta, Nikita Borisov

Our agenda for the next 3-4 classes

- Brief overview of key-value stores
- Distributed Hash Tables
    - Peer-to-peer protocol for efficient insertion and retrieval of key-value pairs.
- Key-value stores in the cloud
    - How to run large-scale distributed computations over key-value stores?
        - Map-Reduce Programming Abstraction
    - How to design a large-scale distributed key-value store?
        - Case-study: Facebook's Cassandra

Features of cloud

I. Massive scale.
   - Tens of thousands of servers and cloud tenants, and hundreds of thousands of VMs.

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

III. Data-intensive nature:
   - What was MBs has now become TBs, PBs and XBs.
     - Daily logs, forensics, Web data, etc.

Must deal with immense complexity!

- Fault-tolerance and failure-handling
- Replication and consensus
- Cluster scheduling

- How would a cloud user deal with such complexity?
  - **Powerful abstractions and frameworks**
  - Provide **easy-to-use** API to users.
  - Deal with the complexity of distributed computing under the hood.

MapReduce
is one such powerful
abstraction.

MapReduce Abstraction

- Map/Reduce
  - Programming model inspired from LISP (and other functional languages).
- Expressive: many problems can be phrased as map/reduce.
- Easy to distribute across nodes.
  - High-level job divided into multiple independent "map" tasks, followed by multiple independent "reduce" tasks.
- Nice retry/failure semantics.

MapReduce Architecture

- *MapReduce programming abstraction:*
  - Easy to program distributed computing tasks.

- MapReduce programming abstraction offered by multiple open-source *application frameworks*:
  - Handle creation of "map" and "reduce" tasks.
  - *e.g. Hadoop: one of the earliest map-reduce frameworks.*
  - *e.g. Spark: easier API and performance optimizations.*

- Application frameworks use *resource managers*.
  - Deal with the hassle of distributed cluster management.
  - *e.g. Kubernetes, YARN, Mesos, etc.*

MapReduce Architecture

- *Map/Reduce abstraction:*
  - Easy to program distributed computing tasks.

- MapRe … le open-s…
  - Cre…
  - e.g. … s.
  - e.g.

- Automatic parallelization & distribution
- Fault tolerance
- Scheduling
- Monitoring & status updates

- Application frameworks use *resource managers*.
  - Deal with the hassle of distributed cluster management.
  - e.g. *Kubernetes, YARN, Mesos, etc.*

Map/Reduce in LISP

Sum of squares:

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

- (reduce + 0'(1 4 9 16))
  - (+ 16 (+ 9 (+ 4 (+1 + 0))))
  - Output: 30

Map/Reduce in LISP

Sum of squares:

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

  [processes each record sequentially and independently]

Unary operator

- (reduce + 0‘(1 4 9 16))
  - (+ 16 (+ 9 (+ 4 (+1 + 0))))
  - Output: 30

Map/Reduce in LISP

Sum of squares:

Unary operator

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

  [processes each record sequentially and independently]

Binary operator

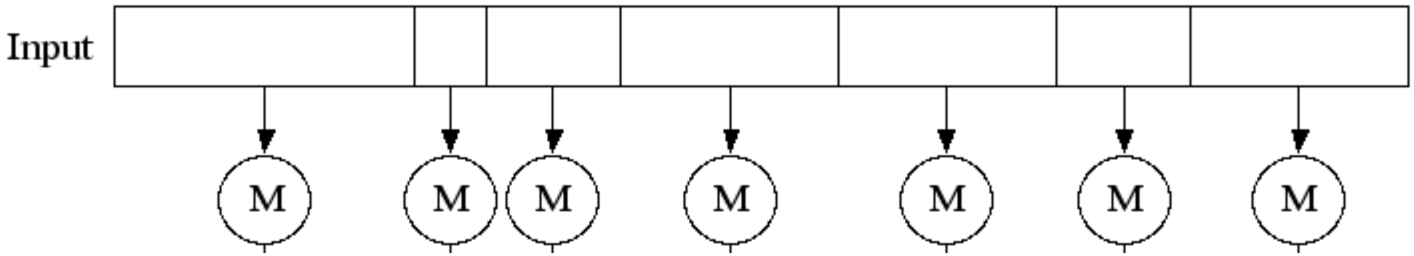
- (reduce **+** 0‘(1 4 9 16))
  - (+ 16 (+ 9 (+ 4 (+1 + 0))))
  - Output: 30

  [processes set of *all records* in batches]

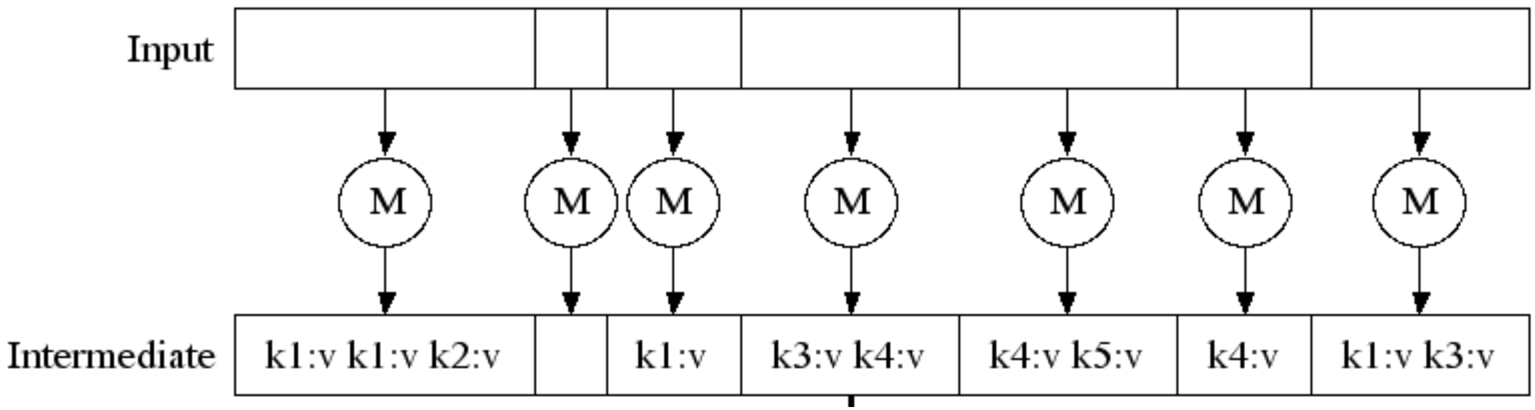
MapReduce Overview

- 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.

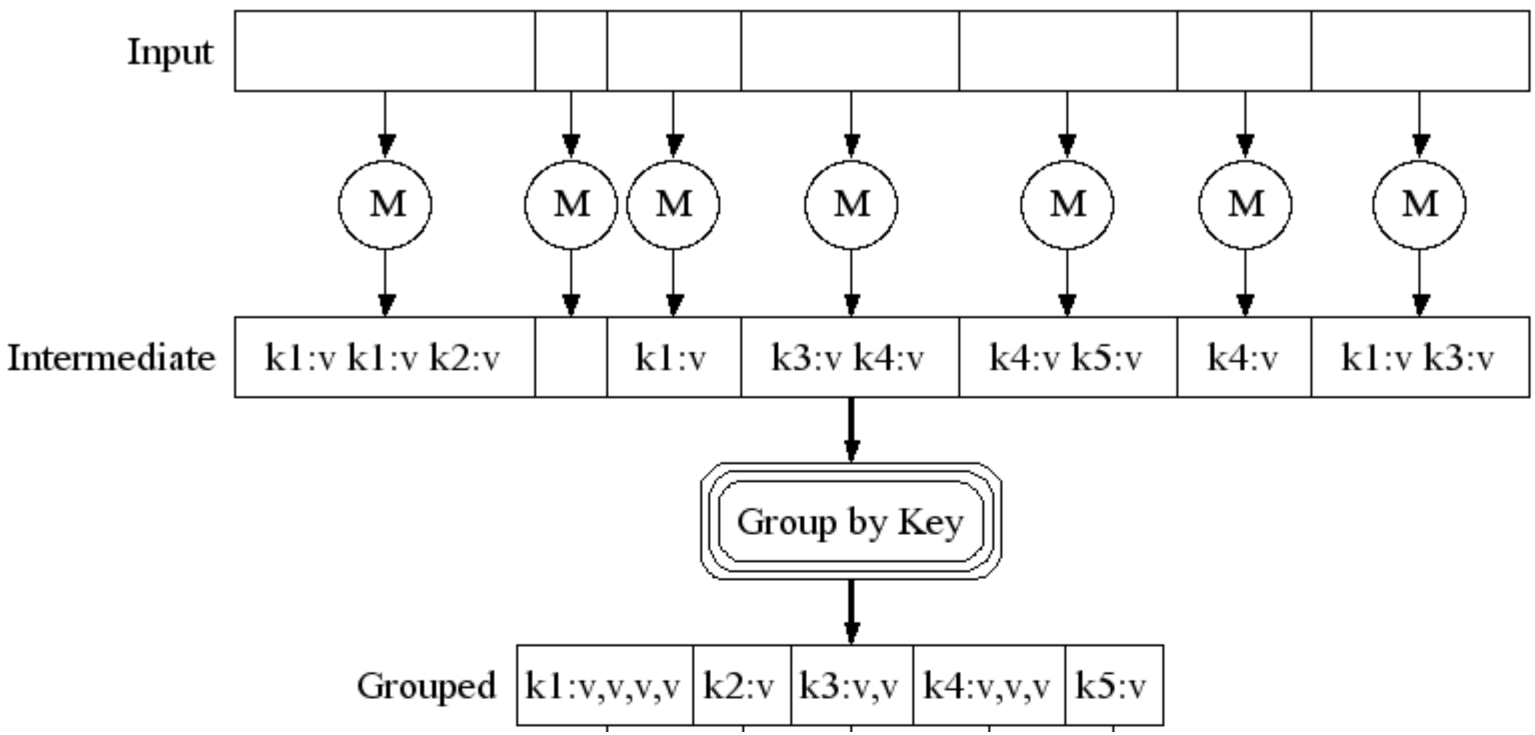
MapReduce Overview



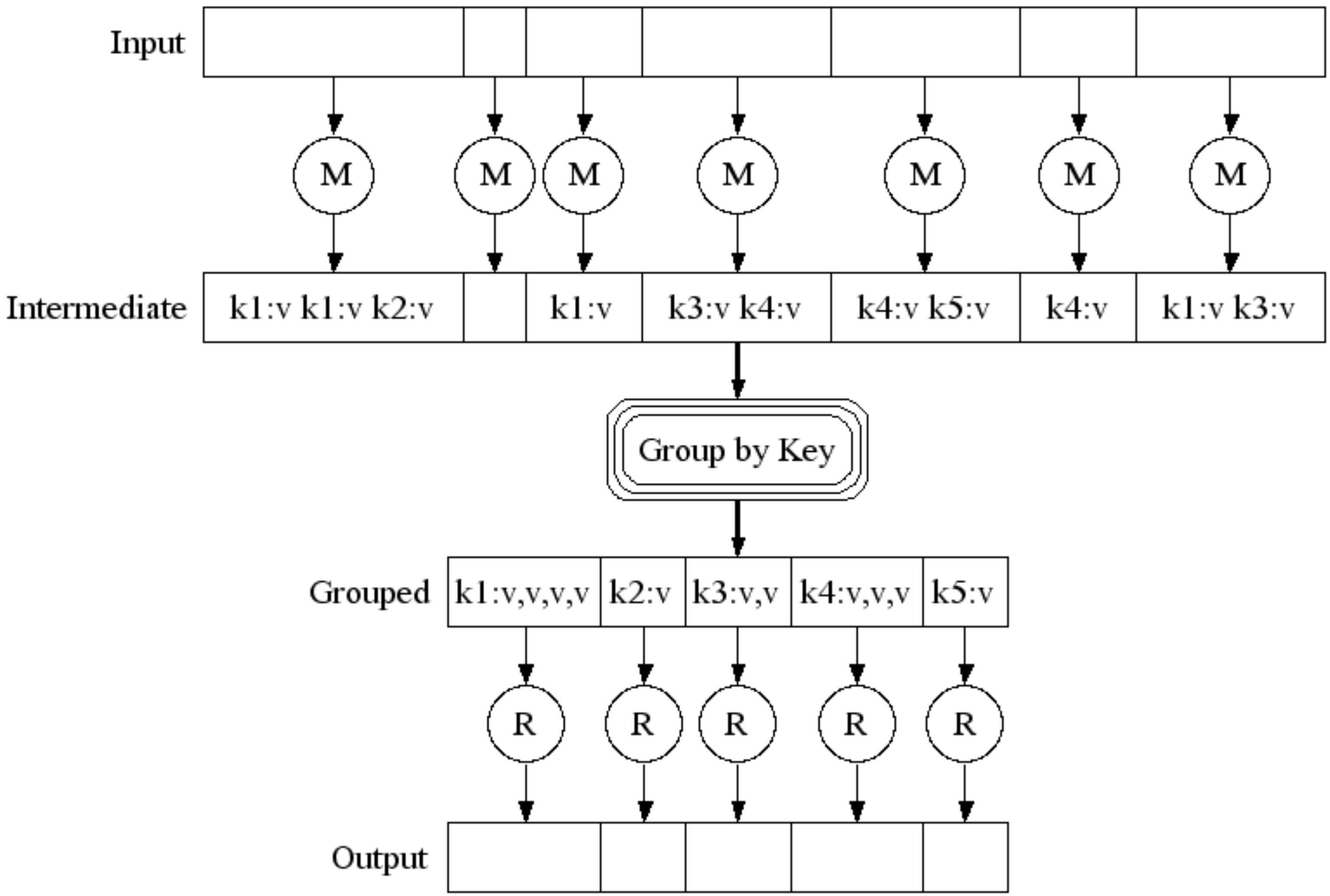
MapReduce Overview



MapReduce Overview



MapReduce Overview



Typical Example: Word Count

- We have a large file of words containing multiple lines (or records).
- Count the number of times each distinct word appears in the file.

- *Sample application*: analyze web server logs to find popular URLs.

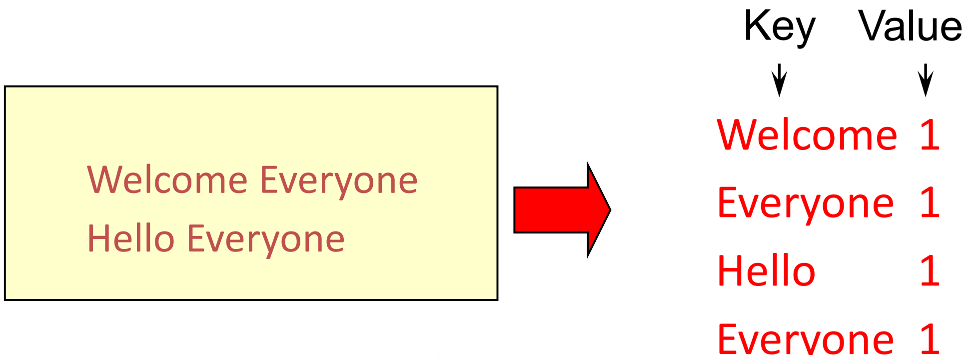
Map

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

Input <record, text per record>

Map

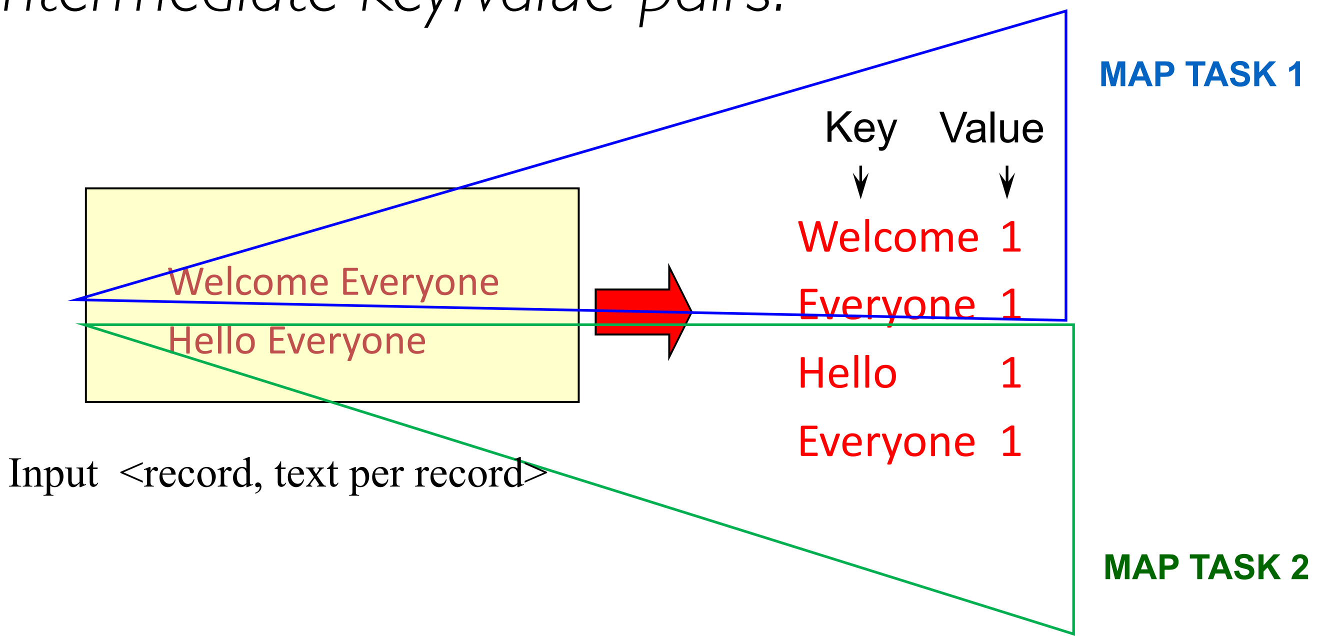
- Process individual records to generate *intermediate key/value pairs.*



Input <record, text per record>

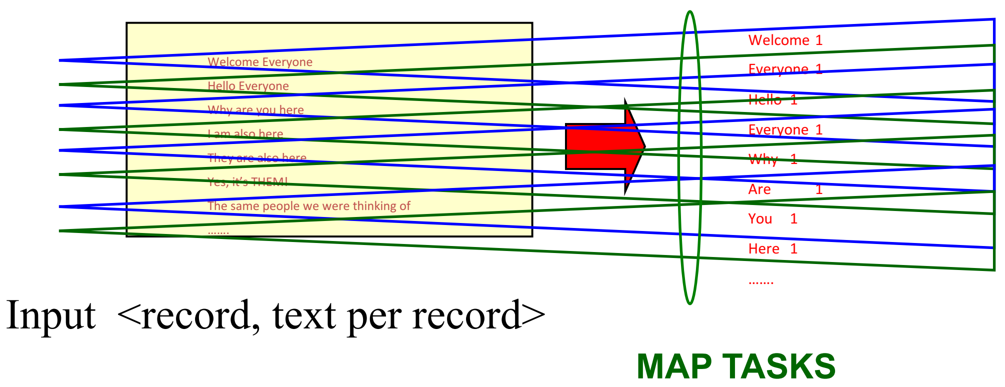
Map

- 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.



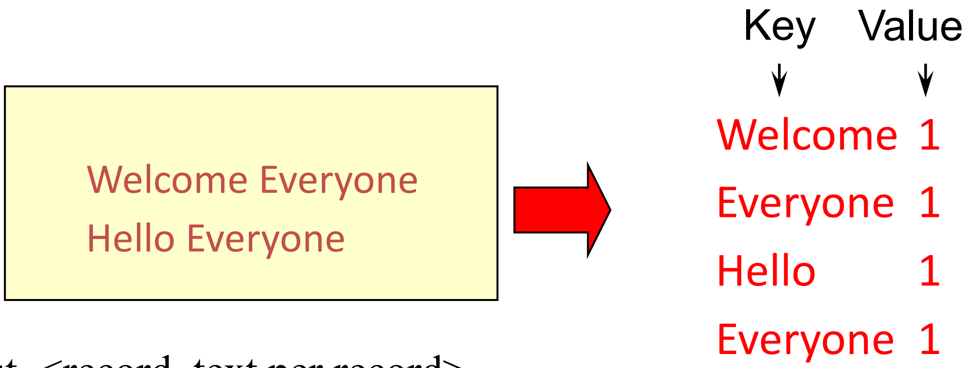
Reduce

- Processes and merges all intermediate values associated <u>per key.</u>

Welcome	1
Everyone	1
Hello	1
Everyone	1

Map

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



Input <record, text per record>

Reduce

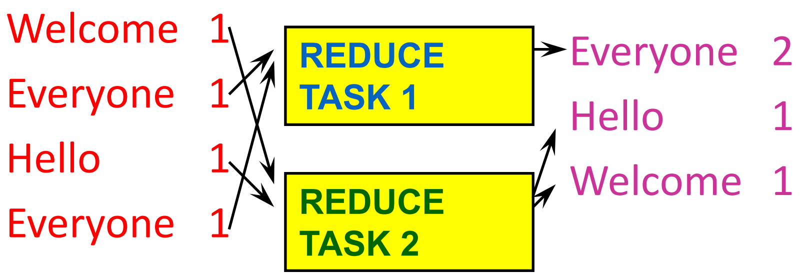
- Processes and merges all intermediate values associated per key.

Welcome	1
Everyone	1
Hello	1
Everyone	1

Key	Value
Everyone	2
Hello	1
Welcome	1

Reduce

- Each key assigned to one Reduce task.
- Parallelly processes and merges all intermediate values <u>partitioned</u> per key.



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

MapReduce Overview

- 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 Overview

- Input: a set of key/value pairs (record, list of words)
- User supplies two functions:
  - map(k,v) → list(k1,v1)
  - reduce(k1, list(v1)) → v2
- (k1,v1) is an intermediate key/value pair. (word, 1)
- Output is the set of (k1,v2) pairs. (word, count)

Word Count using MapReduce

```
map(key, value):
// key: record (line no.); value: list of words in the record
  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)
```

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
```

Spark Code

Python:

```
text_file = sc.textFile("hdfs://...")
counts = text_file.flatMap \
            (lambda line: line.split(" ")) \
            .map(lambda word: (word, 1)) \
            .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

// Source: http://spark.apache.org/examples.html

More examples: Host size

- Suppose we have a large web corpus
- Metadata file
  - Lines of the form (URL, size, date, …)
- For each host, find the total number of bytes
  - i.e., the sum of the sizes for all pages from a given host/URL

```
map(key, value):
// key: metadata record#;
//value: (URL, size, …) :
   for each (URL, size) in value:
            emit(URL, size)
```

```
reduce(key, values):
// key: URL, values: iterator over sizes:
   result = 0
   for each size s in values:
           result += s
   emit(key, result)
```

More examples: Distributed Grep

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

```
map(key, value):
// key: file, value: list of lines
   for each line in value:
      if "pattern" in line:
         emit(line, 1)
```

```
reduce(key, values):
// key: line that matches pattern; values: 1's
   emit(key, 1)
```

More examples: Graph reversal

- Input: Web graph: tuples (a, b) where (page a → page b)
- Output: For each page, list of pages that link to it

```
map(key, value):
   // key: source page,
   //value: target page
   emit(value, key)
```

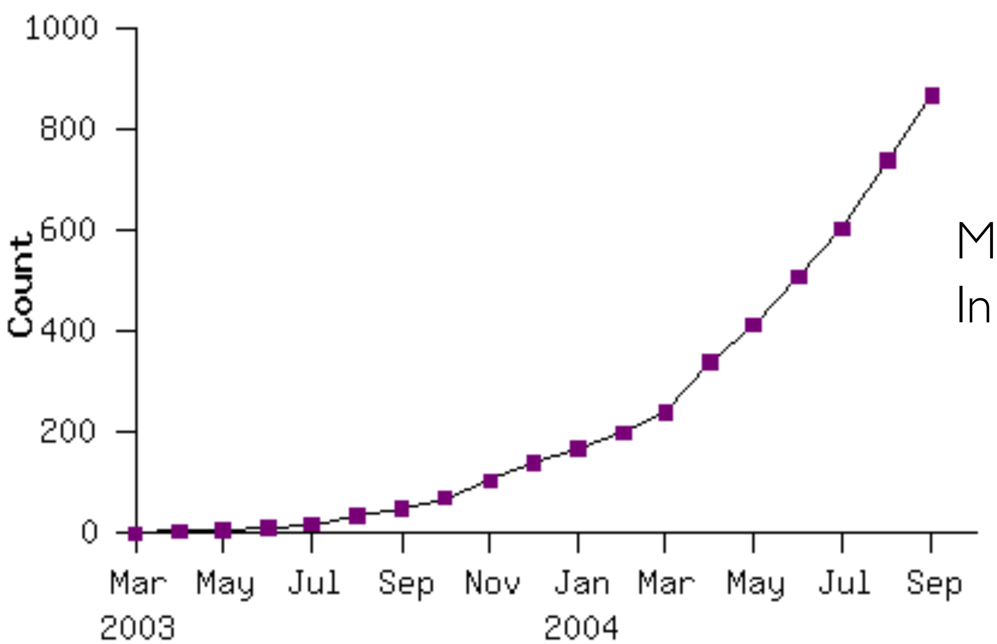
```
reduce(key, values):
   // key: target; values: list of pages that link to it.
   result = concetanate(values)
   emit(key, result)
```

MapReduce Chains

- map1 -> reduce1 -> map2 -> reduce2

- E.g., output the most common words by frequency
  - Map1: emit (“word”, 1)
  - Reduce1: emit (“word”, count)
  - Map2: emit (count, “word”)
  - Reduce2: identity, i.e. emit(count, list of words)

MapReduce is popular and widely applicable



MapReduce Programs
In Google Source Tree

Example uses:

distributed grep	distributed sort	web link-graph reversal
term-vector / host	web access log stats	inverted index construction
document clustering	machine learning	statistical machine translation
...	...	...

MapReduce Execution

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!

MapReduce Execution

Internally: For the framework and resource manager in the cloud

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)

MapReduce Execution

Internally: For the framework and resource manager in the cloud

1. Parallelize Map (easy!)
   - Each map task is independent of the other!
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)

MapReduce Execution

Internally: For the framework and resource manager in the cloud

1. Parallelize Map (easy!)
2. Transfer data from Map to Reduce (**shuffle data**)
   - All Map output records with same key assigned to same Reduce
   - Use partitioning function, e.g., hash(key)%number of reducers
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)

MapReduce Execution

Internally: For the framework and resource manager in the cloud

1. Parallelize Map (easy!)
2. Transfer data from Map to Reduce (**shuffle data**)
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

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

MapReduce Execution

Internally: For the framework and resource manager in the cloud

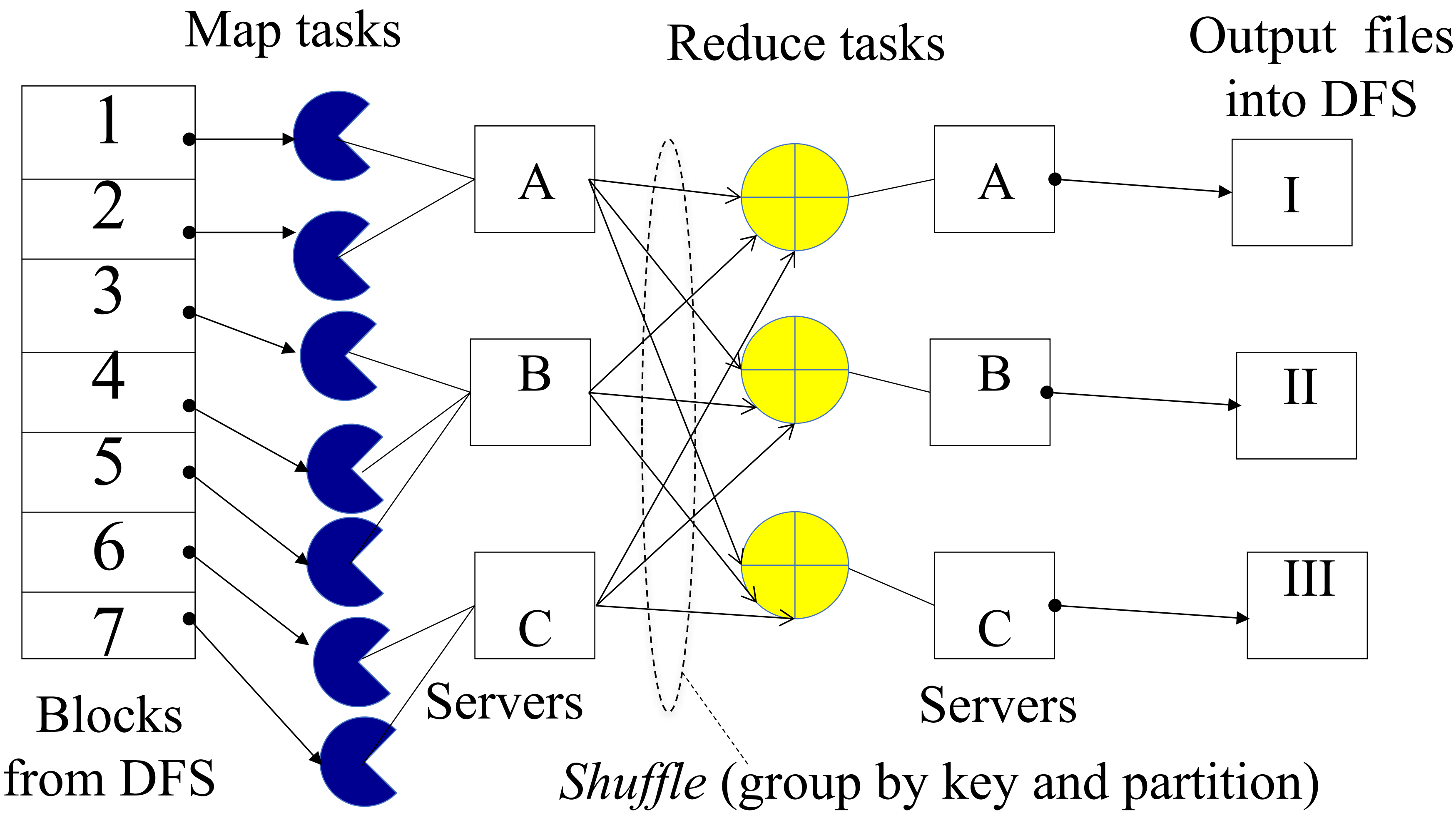
1. Parallelize Map
2. Transfer data from Map to Reduce (**shuffle data**)
3. Parallelize Reduce
4. Implement Storage for Map input, Map output, Reduce input, and Reduce output
   - Map input: from distributed file system/data store
   - Map output: to local disk (at Map node); uses local file system
   - Reduce input: from (multiple) remote disks; uses local file systems
   - Reduce output: to distributed file system/data store

   local file system (e.g. Linux FS)

   distributed file system (e.g. Google File System, Hadoop Distributed File System)

   distributed data store (e.g. Cassandra, BigTable, Spanner, DynamoDB)

MapReduce Execution



Resource Manager (assigns map and reduce tasks to servers)

Resource Manager

- Examples:
  - *YARN* (Yet Another Resource Negotiator), used underneath Hadoop 2.x +
  - *Kubernetes*, *Borg*, *Mesos*, etc.
- Treats each server as a collection of *containers*
  - Container = fixed CPU + fixed memory (e.g. *Docker)*
  - Each tasks runs in a container.
- Has 3 main components
  - Global *Resource Manager (RM):* Cluster Scheduling
  - Per-server *Node Manager (NM):* Daemon and server-specific functions
  - Per-application (job) *Application Master (AM)*
    - Container negotiation with RM and NMs.
    - Handling task failures of that job.

Fault Tolerance

- 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.

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.
- Straggler mitigation has been a very active area of research.

Barrier at the end of Map phase!

Task Scheduling

- Favour data locality:
  - attempts to schedule a map task on a machine that contains a replica of corresponding input data.
  - *if that's not possible*, on the same rack as a machine containing the input.
  - *if that's not possible*, anywhere.
- What does "*if that's not possible*" mean?
  - No more resources available on the machine.
  - Might be worth waiting a while for resources to become available.
    - Delay scheduling in Spark!
- Cluster scheduling is also a very active area of research.

Summary

- Cloud provides distributed computing infrastructure as a service.
- Running a distributed job on the cloud cluster can be very complex:
    - Must deal with parallelization, scheduling, fault-tolerance, etc.
- MapReduce is a powerful abstraction to hide this complexity.
    - User programming via easy-to-use API.
    - Distributed computing complexity handled by underlying frameworks and resource managers
- Plenty of ongoing research work in scheduling, fault-tolerance, and straggler mitigation for MapReduce.