ECEN 757
Spring 2022

Lecture 3: Mapreduce and Hadoop

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]

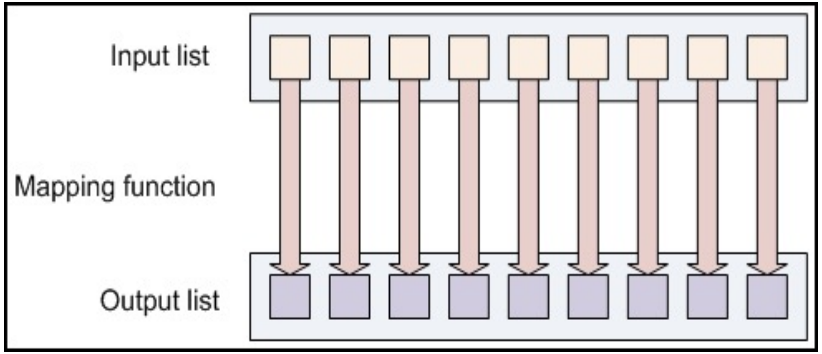
- (reduce + ‘(1 4 9 16))
  - (+ 16 (+ 9 (+ 4 1)))
  - Output: 30

  [processes set of all records in batches]

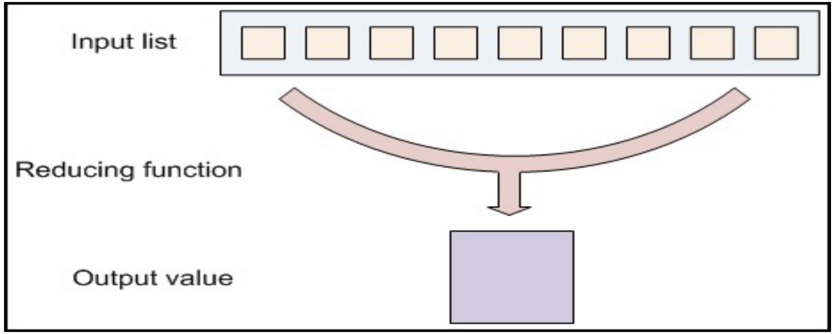
- Let’s consider a sample application: Wordcount
  - You are given a <u>huge</u> dataset (e.g., Wikipedia dump or all of Shakespeare’s works) and asked to list the count for each of the words in each of the documents therein

A High Level Overview

- MapReduce transforms a list of input into a list of output in two steps:
- Map: Transform each input element into an output element



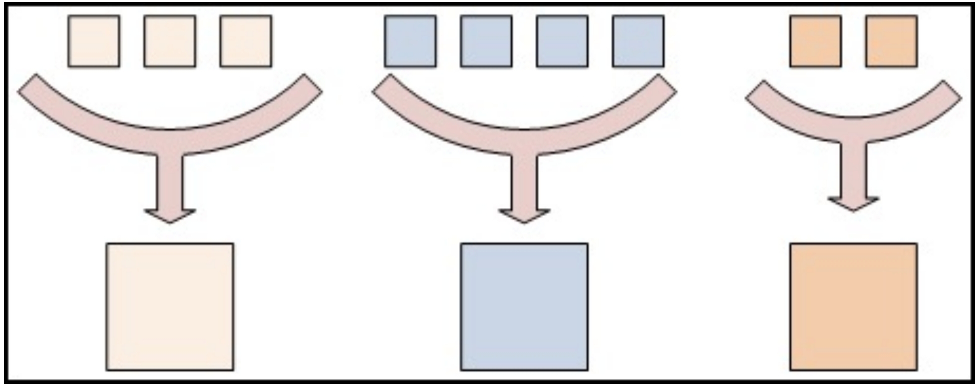
- Reduce: Transform an input list into an output element



Key/Value Specification

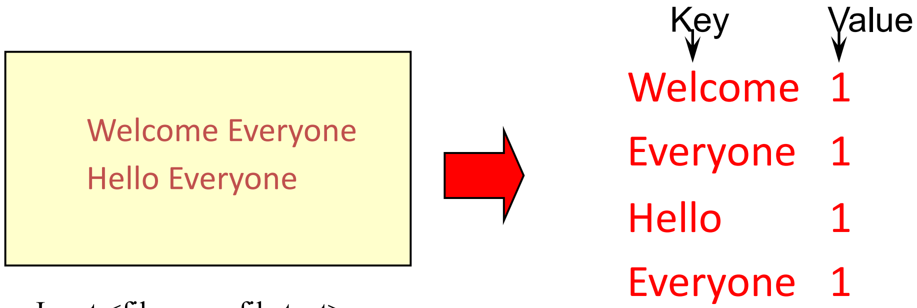
- All inputs/outputs are presented as a list of (key, value) pairs
  - In Wordcount, we can define "key" as a word, and "value" as the count of the word

- In the Reduce step, all pairs with the same key are aggregated at the same reducer



Map

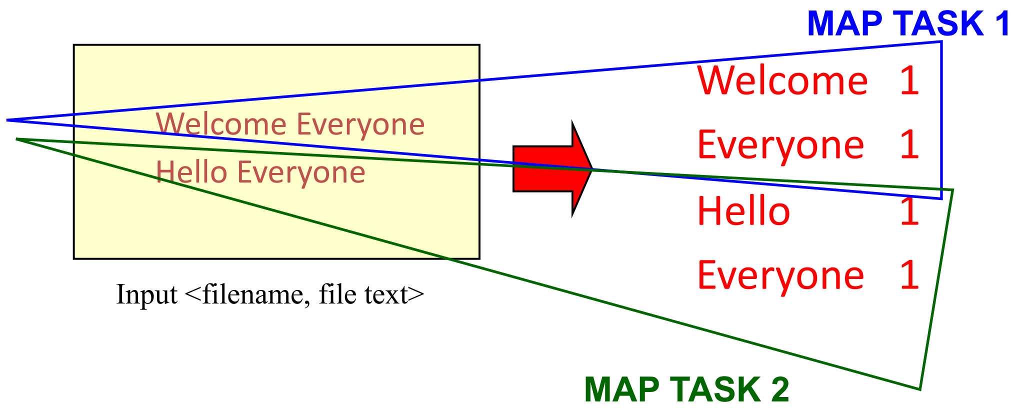
- Process individual records to generate intermediate key/value pairs.



Input <filename, file text>

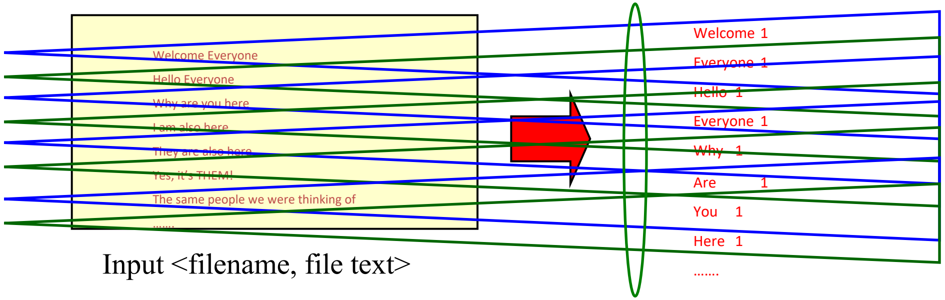
Map

- 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.



MAP TASKS

Reduce

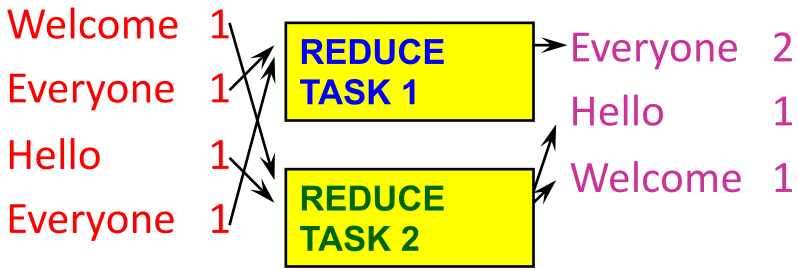
- Reduce processes and merges all intermediate values associated per key

Welcome	1
Everyone	1
Hello	1
Everyone	1

Key	Value
Everyone	2
Hello	1
Welcome	1

Reduce

- Each key assigned to one Reduce
- 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 servers

Pseudocode

```
mapper (filename, file-contents):
  for each word in file-contents:
    emit (word, 1)

reducer (word, values):
  sum = 0
  for each value in values:
    sum = sum + value
  emit (word, sum)
```

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)
    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 {
      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 → 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)>*

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 <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 – Sums up *URL_count's* to calculate overall_count.

  Emits multiple <URL, URL_count/overall_count>

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
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)

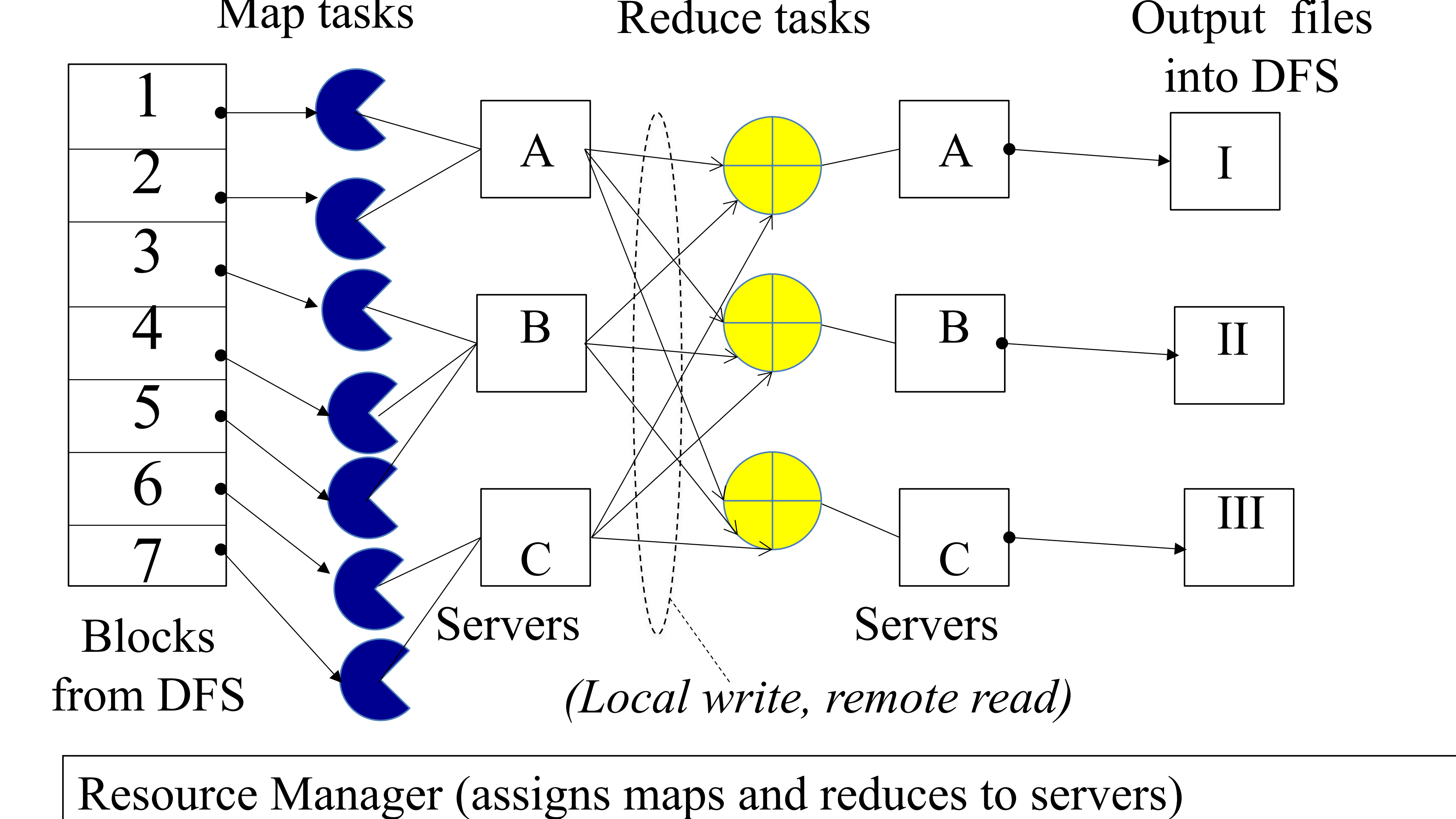
Inside MapReduce

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:
   - All Map output records with same key assigned to same Reduce task
   - Reduce cannot start until all Map tasks finish
   - 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)


Map tasks
Reduce tasks
Output files into DFS
1
2
3
4
5
6
7
A
B
C
A
B
C
I
II
III
Blocks from DFS
Servers
Servers
(Local write, remote read)
Resource Manager (assigns maps and reduces to servers)

The YARN Scheduler

- Used in 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

Fault Tolerance

- Server Failure
  - NM heartbeats to RM
    - If server fails, RM lets all affected AMs know, and AMs take 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

Fault Tolerance

- If failures happen at the Map stage:
- Restart ALL Map tasks assigned to the failed server
  - Map store its output locally
  - The output has not been read by others (because Reduce has not started)

- If failures happen at the Reduce stage
- Some completed tasks already write data to DFS
- Only uncompleted tasks need to be restarted
- RM needs to keep track of the progress

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 straggler task: task considered done when 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
    - a machine that contains a replica of corresponding input data, or failing that,
    - on the same rack as a machine containing the input, or failing that,
    - Anywhere

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://activewizards.com/blog/hadoop-3-comparison-with-hadoop-2-and-spark/ and https://hadoop.apache.org/docs/r3.0.0/)*

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