Hadoop MapReduce

Sang Shin
JPassion.com
"Learn with Passion!"

Topics

- Hadoop MapReduce operational architecture
- MapReduce general concepts
- Hadoop MapReduce programming
- Hadoop MapReduce framework
- Hadoop MapReduce example: Word Count
- Executing Hadoop MapReduce application
- Input file and Input format

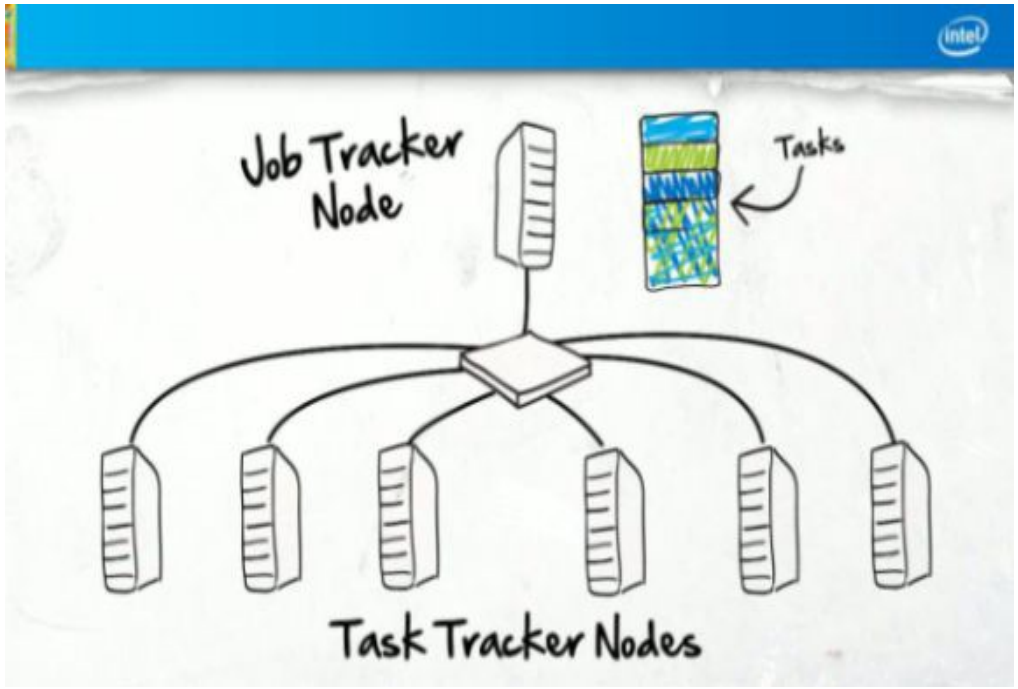
Acknowledgment

- Some contents of this presentation is created from “Hadoop Tutorial from Yahoo!" by Yahoo! Inc. is under a Creative Commons Attribution 3.0 Unported License
  - https://developer.yahoo.com/hadoop/tutorial/module4.html

Hadoop MapReduce Operational Architecture

Hadoop MapReduce Architecture

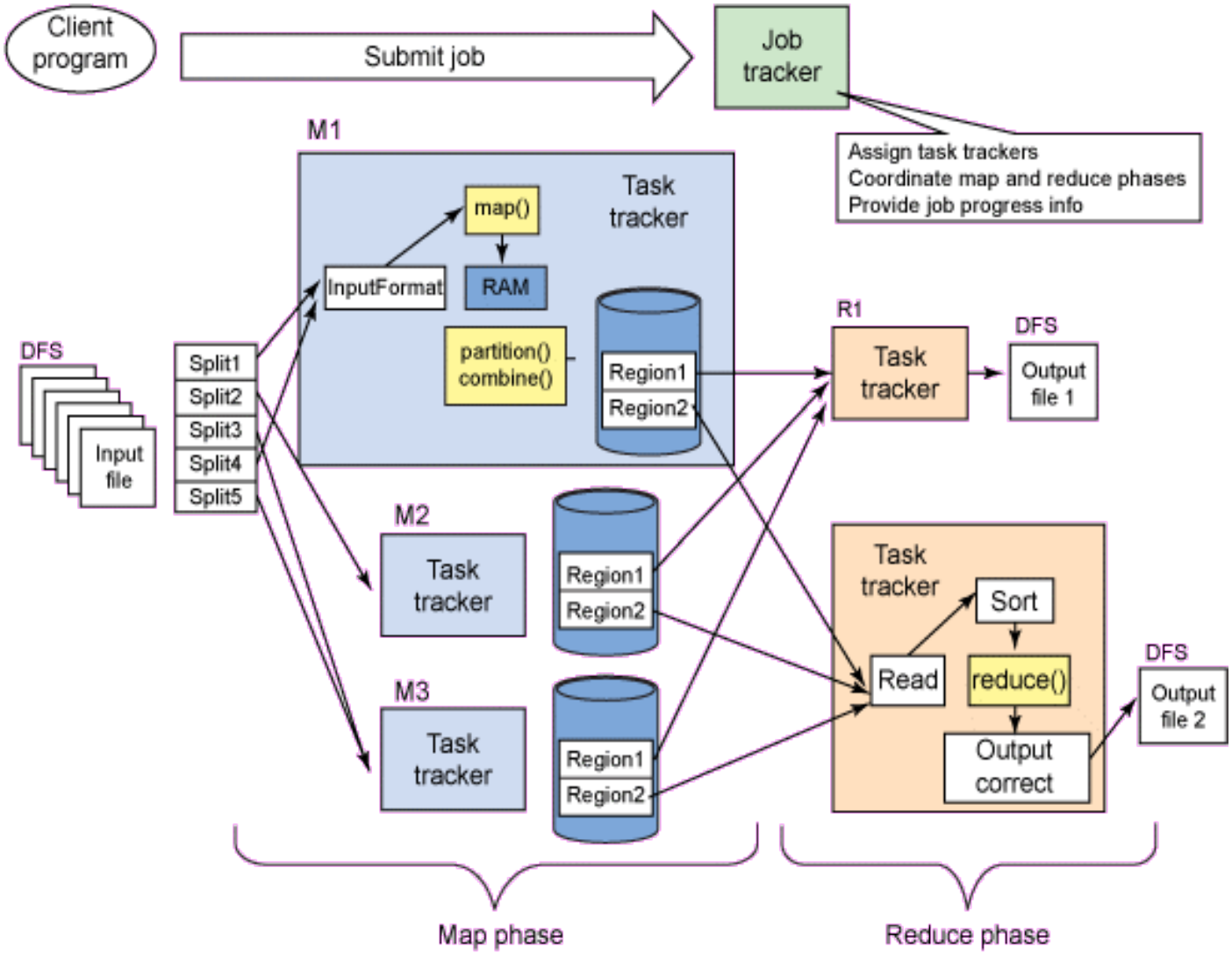
- Job submitter (client): submit jobs to job tracker
- One Job Tracker node: send tasks to task trackers and coordinate them
- Multiple Task Tracker nodes: execute job tasks



Job Execution Flow

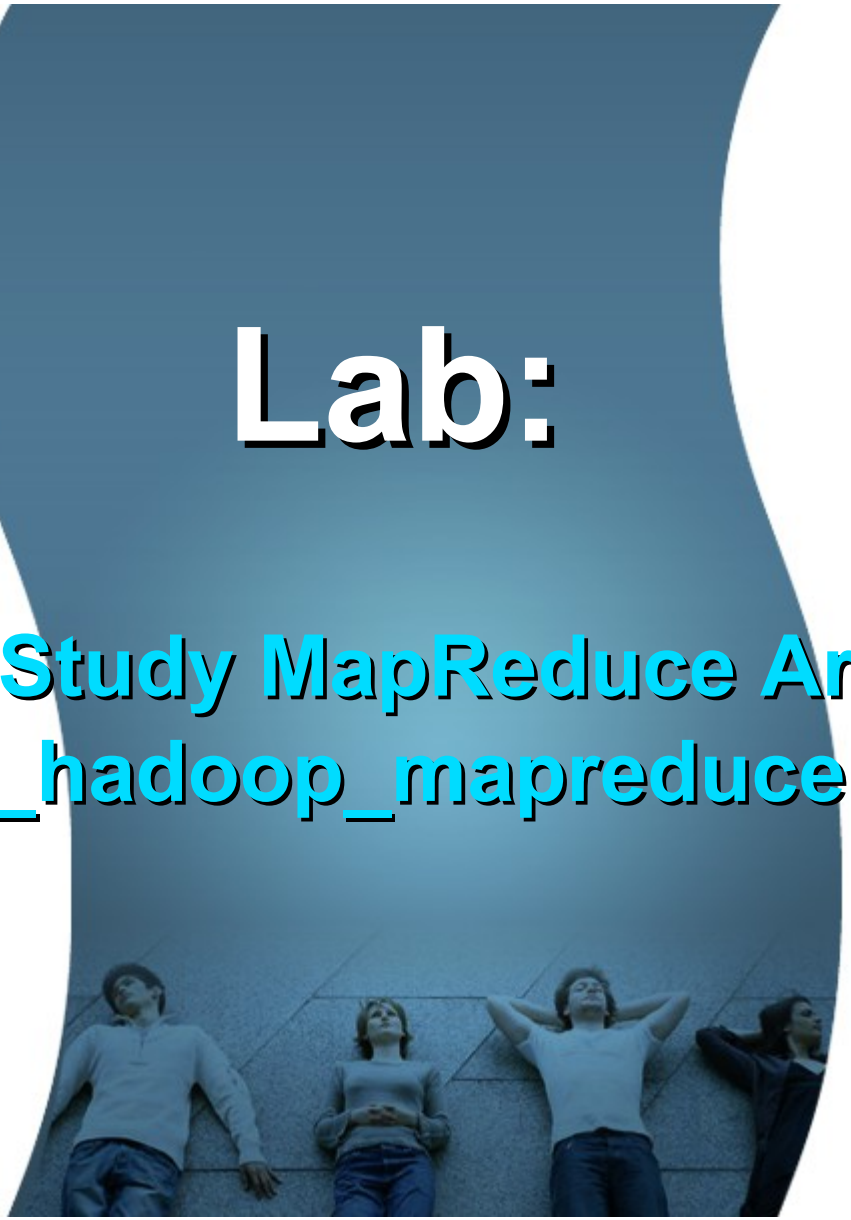
1. Job submitter (client) submits a job to Job Tracker
2. Job Tracker creates execution plan
3. Job Tracker sends tasks to Task Trackers
4. Task Trackers performs tasks and also report progress to Job Tracker via heartbeats
5. Job Tracker manages “map” and “reduce” phases
6. Job tracker updates states

MapReduce Architecture



Lab:

Exercise 0: Study MapReduce Architecture
5908_hadoop_mapreduce.zip



MapReduce General Concepts

Basic Concept of MapReduce

- MapReduce is a programming model designed for processing large volumes of data in parallel by dividing the work into a set of independent tasks
- MapReduce programs are written in a particular style influenced by functional programming constructs, specifically idioms for processing lists of data

MapReduce uses Functional Programming

- Computation of large volumes of data in parallel requires dividing the workload across a large number of machines
- Non-functional programming model, where components were allowed to share data arbitrarily, will not scale to large clusters (hundreds or thousands of nodes)
  - The communication overhead required to keep the data on the nodes synchronized at all times would be extremely high
- MapReduce uses functional programming, where all data elements in MapReduce are immutable, meaning that they cannot be updated
  - If, in a mapping task, you change an input (key, value) pair, it does not get reflected back in the input files; communication occurs only by generating new output (key, value) pairs, which are then forwarded by the system into the next phase of execution

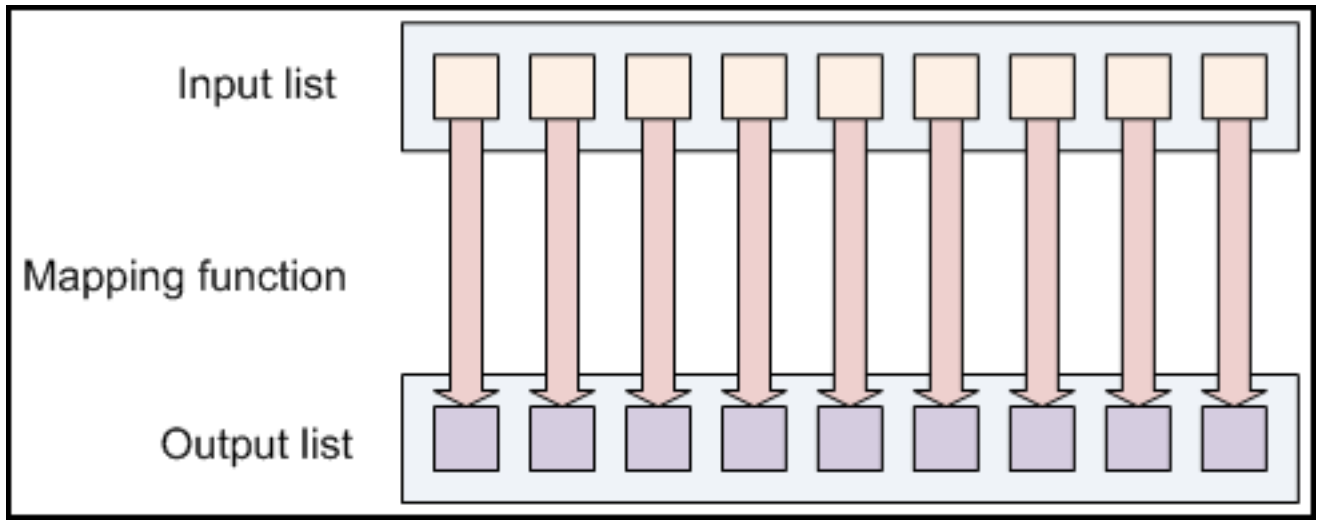
List Processing: Map and Reduce

- Conceptually, a MapReduce program performs the transformation
  - "lists of input data elements" -> "lists of output data elements"

- A MapReduce program will do this twice, using two different list processing idioms in two phases
  - Using "map" idiom in "mapping" phase
  - Using "reducing" idiom in "reducing" phase

“Mapping” Phase

- The first phase of a MapReduce program is called “Mapping” - A list of data elements are provided, one at a time, to a function called the “Mapper”, which transforms each element individually to an output data element – this is called “Mapping”

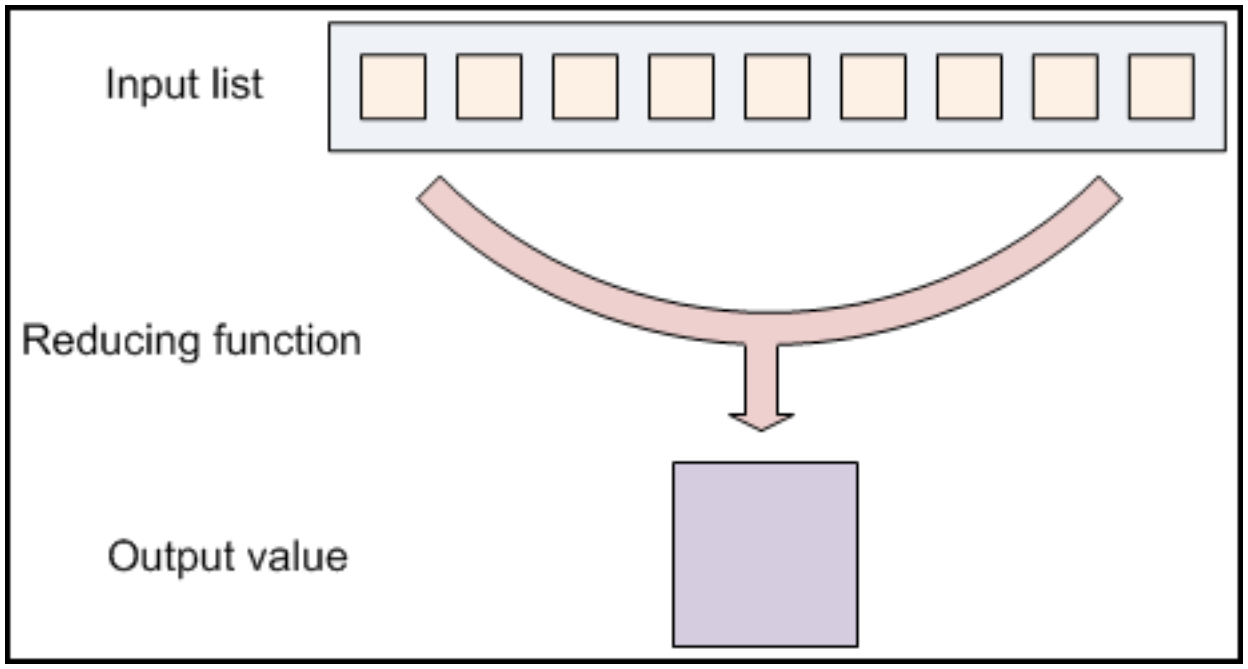


Mapping Lists Example

- A mapping function *toUpper(str)* which returns an uppercase version of its input string
  - You could use this function with map to turn “a list of strings” into “a list of uppercase strings”
  - Note that we are not modifying the input string here: we are returning a new string that will form part of a new output list.

"Reducing" Phase

- Reducing lets you aggregate values together
- A reducer function receives an iterator of input values from an input list. It then combines these values together, returning a single output value



Reducing Lists Examples

- Reducing is often used to produce "summary" data, turning a large volume of data into a smaller summary of itself
- For example, "+" can be used as a reducing function, to return the sum of a list of input values.

Hadoop MapReduce Programming

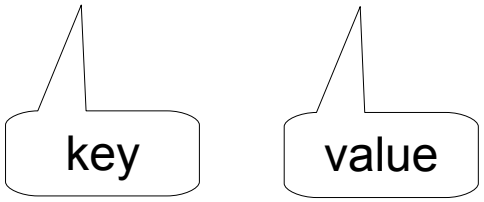
Hadoop MapReduce Framework

- The Hadoop MapReduce framework takes these concepts and uses them to process large volumes of data
- A MapReduce program has two components:
  - Mapper: one that implements the mapper
  - Reducer: one that implements the reducer
- Bother Mapper and Reducer takes input data and then generates output data

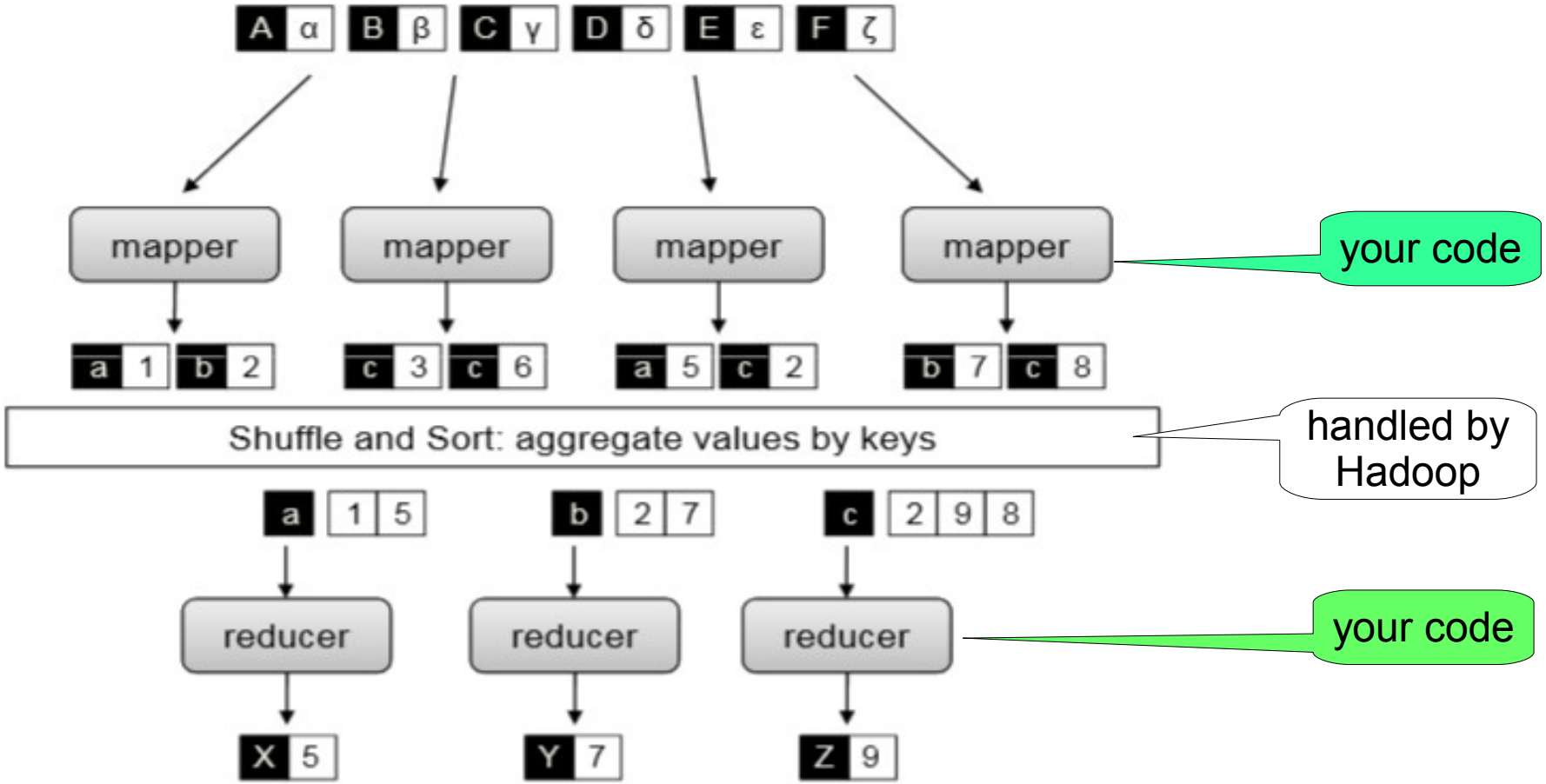
Keys and Values

- In Hadoop MapReduce, data element (in input and output) is always in key/value pair
  - Every value has a key associated with it
- For example, a log of time-coded speedometer readings from multiple cars could be keyed by license-plate number; it would look like:

```
AAA-123   65mph, 12:00pm
ZZZ-789   50mph, 12:02pm
AAA-123   40mph, 12:05pm
CCC-456   25mph, 12:15pm
```



MapReduce Data Flow



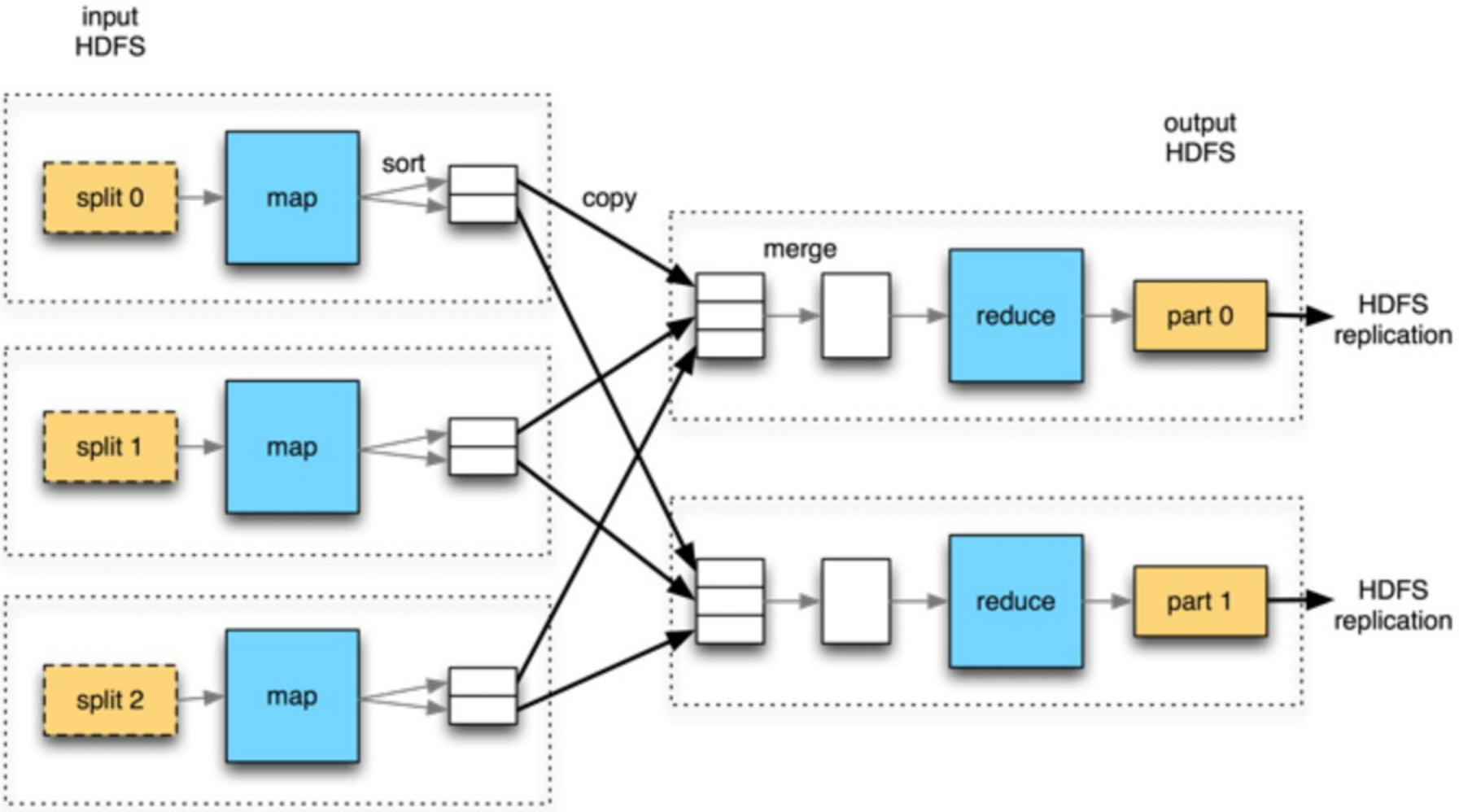
source: http://lintool.github.io/MapReduceAlgorithms/MapReduce-book-final.pdf

Hadoop MapReduce Framework

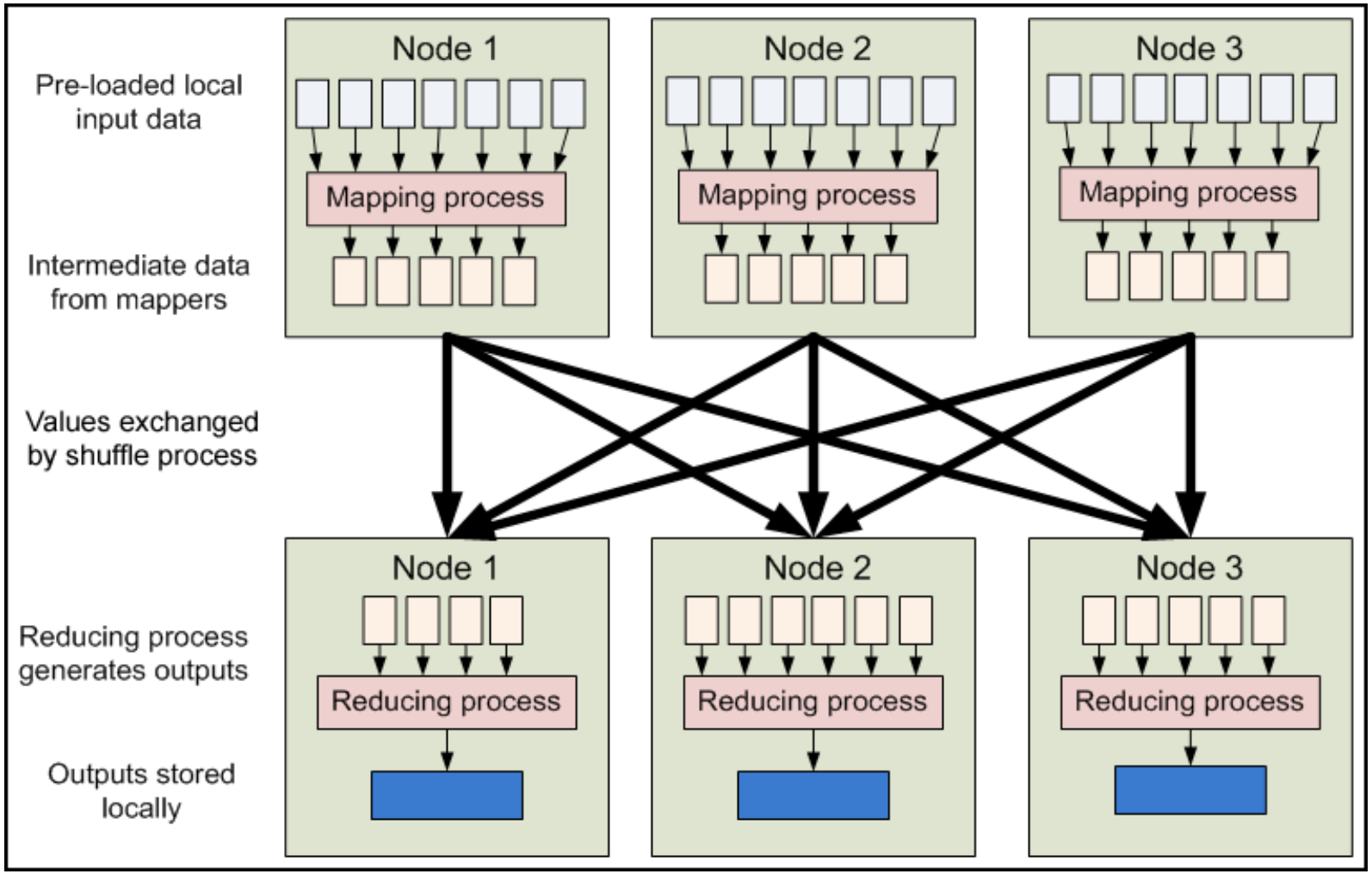
Hadoop MapReduce Framework

- Takes care of distributed processing and coordination
- Scheduling
  - Jobs are broken down into tasks, which are then scheduled
- Task localization with data
  - Hadoop framework sends the tasks (code) to nodes that host segment of data
- Error handling
  - Tasks are automatically retried on other machines when errors occur
- Data synchronization
  - Shuffles and sorts data
  - Moves data between nodes

Hadoop MapReduce Framework



MapReduce Data Flow



Hadoop MapReduce Example: Word Count

Example Application: Word Count

- We want to count how many times each word appears in a set of files
- For example, suppose there are two input files: “foo.txt” & “bar.txt”
  - *foo.txt: Sweet, this is the foo file*
  - *This is the second line in foo file*
  - *bar.txt: This is the bar file*
- We would expect the output
  - *Sweet, 1*
  - *This 2*
  - *bar 1*
  - *file 3*
  - *foo 2*
  - *in 1*
  - *...*

High-level Pseudo-code

```
mapper (file-contents):
  for each word in file-contents:
    emit (word, 1)       # word is key, and 1 is value

<Hadoop performs sorting and suffling>

reducer (word, values): # word, [1,1,1]
  sum = 0
  for each value in values:
    sum = sum + value
  emit (word, sum)    # for each word, sum is computed – word, 3
```

Mapper for Word Count Example

- Several instances of the mapper function are created on the different nodes in the cluster
  - Each instance receives a different input file (it is assumed that we have many such files)
  - The mappers emit (word, 1) pairs as output
- Input element to “word count” mapper
  - (offset from the file, content of a line)

  (0, *Sweet, this is the foo file)*

  (28, This is the second line of the foo file)
- Output elements from “word count” mapper
  - (word1,1) (word2,1) (word3 ,1), etc

  (Sweet,1) (this,1) (the,1) (foo,1) (file,1) (This,1) (is,1)(the,1) ...

Reducer for Word Count Example

- Several instances of the reducer method are also instantiated on the different nodes in the cluster
  - Each reducer is responsible for processing the list of values associated with a different word
  - The list of values will be a list of 1's; the reducer sums up those ones into a final count associated with a single word
  - The reducer then emits the final (word, count) output which is written to an output file
- Input element to “word count” reducer
  - (word1, (1,1)) (word2, (1,1,1)) (word3 , 1), etc

  (Sweet, 1) (this, 1) (the, (1,1,1)) (foo, (1,1)) (file, (1,1))
- Output elements from “word count” reducer
  - (word1, count)

  (Sweet,1) (the,3) (foo,2)..

Program Components

- Mapper class
  - Represents a Mapper
- Reducer class
  - Represents a Reducer
- Driver
  - Initializes the job and instructs the Hadoop platform to execute your code on a set of input files, and controls where the output files are placed

Word Count Mapper Class

```
// Mapper<KEYIN, VALUEIN, KEYOUT,VALUEOUT> interface
public class WordCountMapper extends Mapper<Object, Text, Text, IntWritable> {

    private Text word = new Text();
    private final static IntWritable one = new IntWritable(1);

    @Override
    public void map(Object key, Text value, Context context) throws IOException,
            InterruptedException {

        // Break line (represented by "value") into words for processing
        StringTokenizer wordList = new StringTokenizer(value.toString());

        while (wordList.hasMoreTokens()) {
            word.set(wordList.nextToken());
            context.write(word, one);
        }
    }
}
```

Word Count Reducer Class

```
// Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT>
public class WordCountReducer extends
        Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text text, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {

        int sum = 0;
        for (IntWritable value : values) {
            sum += value.get();
        }
        context.write(text, new IntWritable(sum));
    }
}
```

Driver Class

```
// Create job
Job job = new Job(conf, "WordCountDriver");
..

// Setup MapReduce classes
job.setMapperClass(WordCountMapper.class);
job.setReducerClass(WordCountReducer.class);
job.setNumReduceTasks(1);

...

// Execute job
int code = job.waitForCompletion(true) ? 0 : 1;
System.exit(code);
```

Executing MapReduce Application

Executing MapReduce Application

- Within Eclipse
  - Default file system is local file system
  - In order to use HDFS, you have to provide full path (as shown below)

  hdfs://localhost:8020/user/cloudera/my_data/input

- At the command line
  - Default file system is HDFS
  - You have to create jar file that contains the Java classes
  - Use “hadoop” command to run it

  hadoop jar target/wordcount-0.0.1-SNAPSHOT.jar com.jpassion.wordcount.WordCountDriver my_data/input output

Input and Output Directory

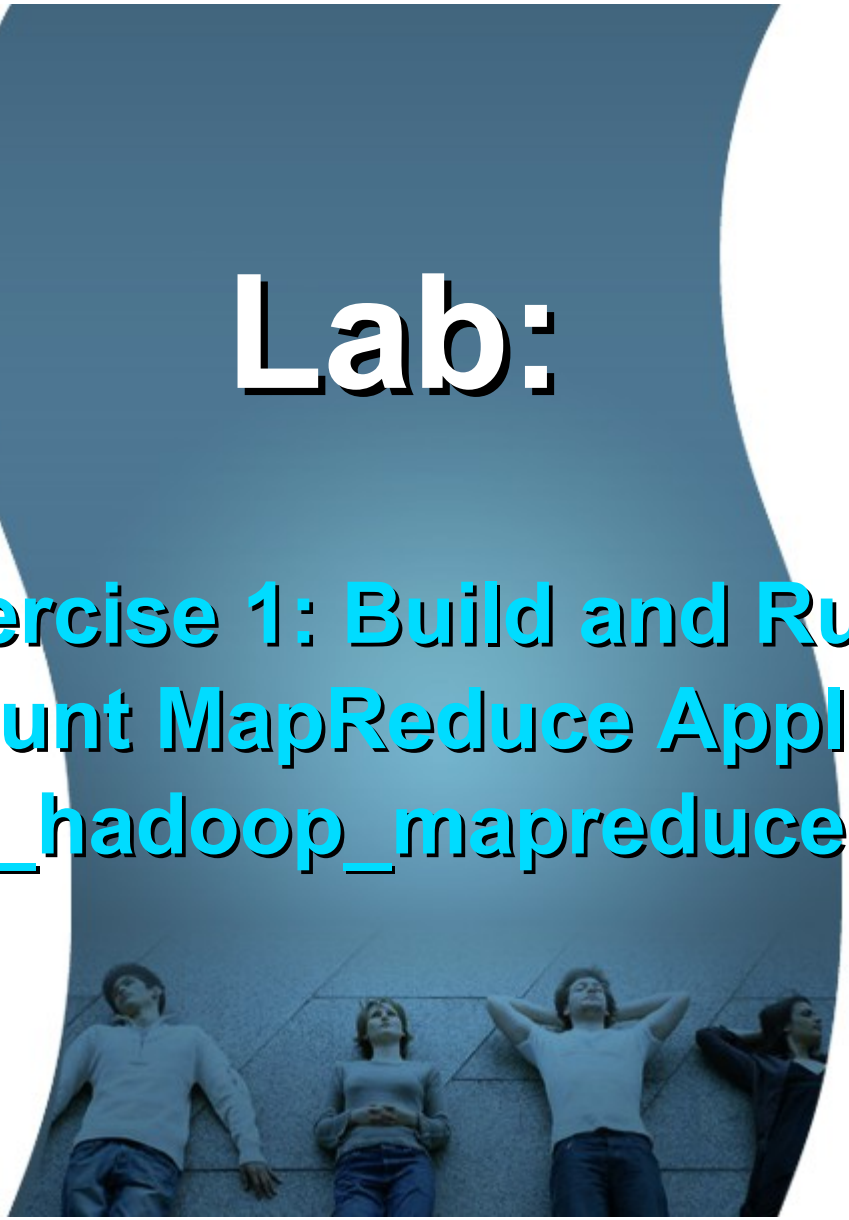
- An input can be either directory or a file
  - If it is a directory, all files under that directory are used as input files
- The output directory gets created by Hadoop
  - If the output directory already exists, Hadoop generates an error
  - You can delete the output directory first programmatically before submitting a job (we do this in our sample apps as shown below)

```
// Delete output if exists
FileSystem hdfs = FileSystem.get(conf);
if (hdfs.exists(outputDir))
        hdfs.delete(outputDir, true);

// Execute job
int code = job.waitForCompletion(true) ? 0 : 1;
System.exit(code);
```

Lab:

Exercise 1: Build and Run WordCount MapReduce Application 5908_hadoop_mapreduce.zip



Input Files and Input Format

Input Files

- This is where the data for a MapReduce task is initially stored
- The input files typically reside in HDFS for scalability and reliability reasons
- The format of these files is arbitrary
  - While line-based text files can be used, we could also use a binary format, multi-line input records, or something else entirely
- It is typical for these input files to be very large
  - Tens of gigabytes or more

Input Format

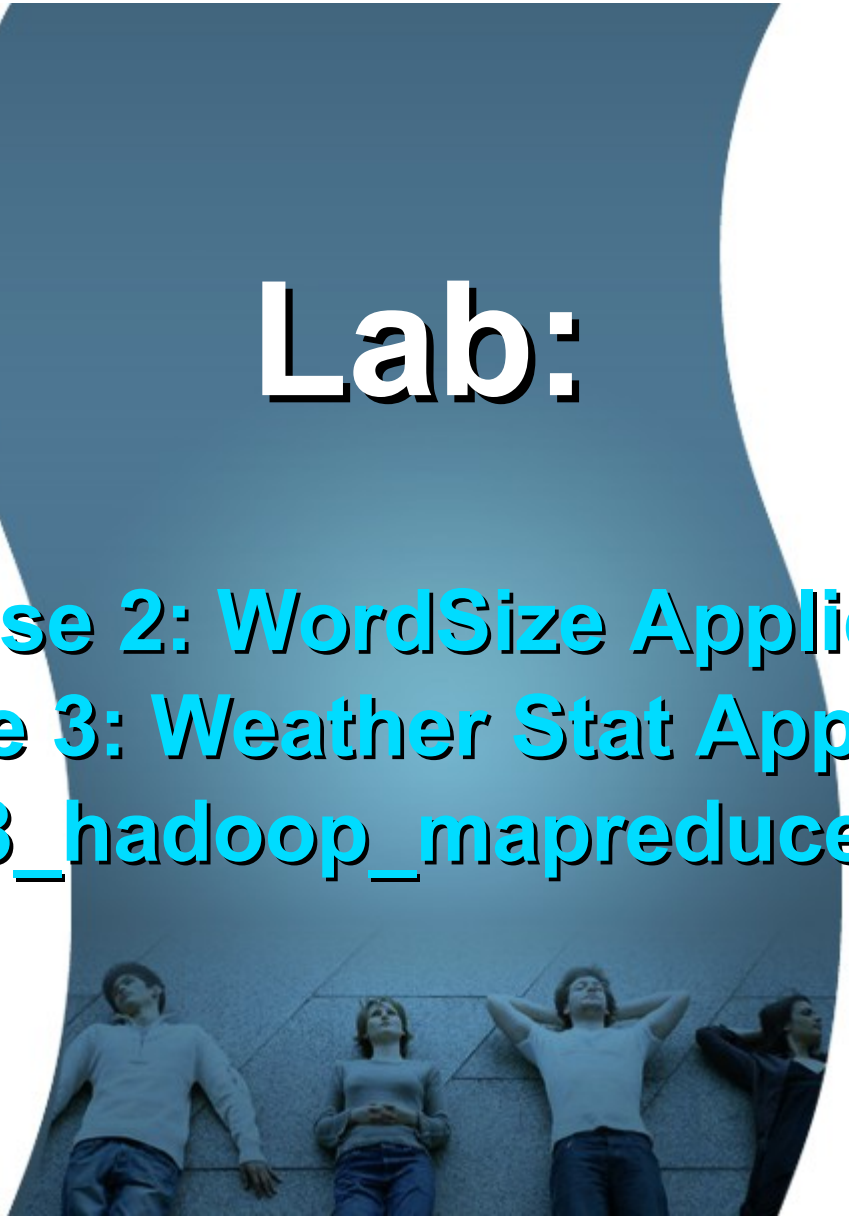
- How these input files are split up and read is defined by the *InputFormat*
- An *InputFormat* is a class that provides the following functionality:
  - Selects the files or other objects that should be used for input
  - Defines the *InputSplits* that break a file into tasks
  - Provides a factory for *RecordReader* objects that read the file
- Several InputFormats are provided with Hadoop
  - *TextInputFormat*
  - *KeyValueInputFormat*
  - *SequenceFileInputFormat*
- The default *InputFormat* is the *TextInputFormat*
  - This treats each line of each input file as a separate record, and performs no parsing

Input Formats Provided by MapReduce

InputFormat	Description	Key	Value
TextInputFormat	Default format; reads lines of text files	The byte offset of the line	The line contents
KeyValueInputFormat	Parses lines into key, val pairs	Everything up to the first tab character	The remainder of the line
SequenceFileInputFormat	A Hadoop-specific high-performance binary format	user-defined	user-defined

Lab:

Exercise 2: WordSize Application
Exercise 3: Weather Stat Application
5908_hadoop_mapreduce.zip



Learn with Passion!
JPassion.com