Chapter 3

Distributed Algorithms based on MapReduce

Acknowledgements

- Hadoop: The Definitive Guide. Tome White. O'Reilly.
- Hadoop in Action. Chuck Lam, Manning Publications.
- MapReduce: Simplified Data Processing on Large Clusters. Jeff Dean, Sanjay Ghemawat, Google, Inc.
- MapReduce: Simplified Data Processing on Large Clusters. Slides from Dan Weld's class at U. Washington.
- MapReduce:
  Simplified Data Processing on Large Clusters. Conroy Whitney, 4th year CS – Web Development

Chapter Outline

- What is MapReduce
- Motivation
- Functions
- Work flow & data flow
- “Hello World” in MapReduce
- Details of your own MapReduce programs
- Implementation

What is MapReduce

- Origin from Google [OSDI'04]
  - MapReduce: Simplified Data Processing on Large Clusters
  - Jeffrey Dean and Sanjay Ghemawat
- Programming model for parallel data processing
- Hadoop can run MapReduce programs written in various languages, e.g., Java, Ruby, Python, C++
- For large-scale data processing
  - Exploits large set of commodity computers
  - Executes process in distributed manner
  - Offers high availability

Motivation

- Typical big data problem challenges:
  - How do we break up a large problem into smaller tasks that can be executed in parallel?
  - How do we assign tasks to workers distributed across a potentially large number of machines?
  - How do we ensure that the workers get the data they need?
  - How do we coordinate synchronization among the different workers?
  - How do we share partial results from one worker that is needed by another?
  - How do we accomplish all of the above in the face of software errors and hardware faults?

Typical Big Data Problem

- Iterate over a large number of records
- Extract something of interest from each record
- Shuffle and sort intermediate results

Map

- Aggregate intermediate results
- Generate final output

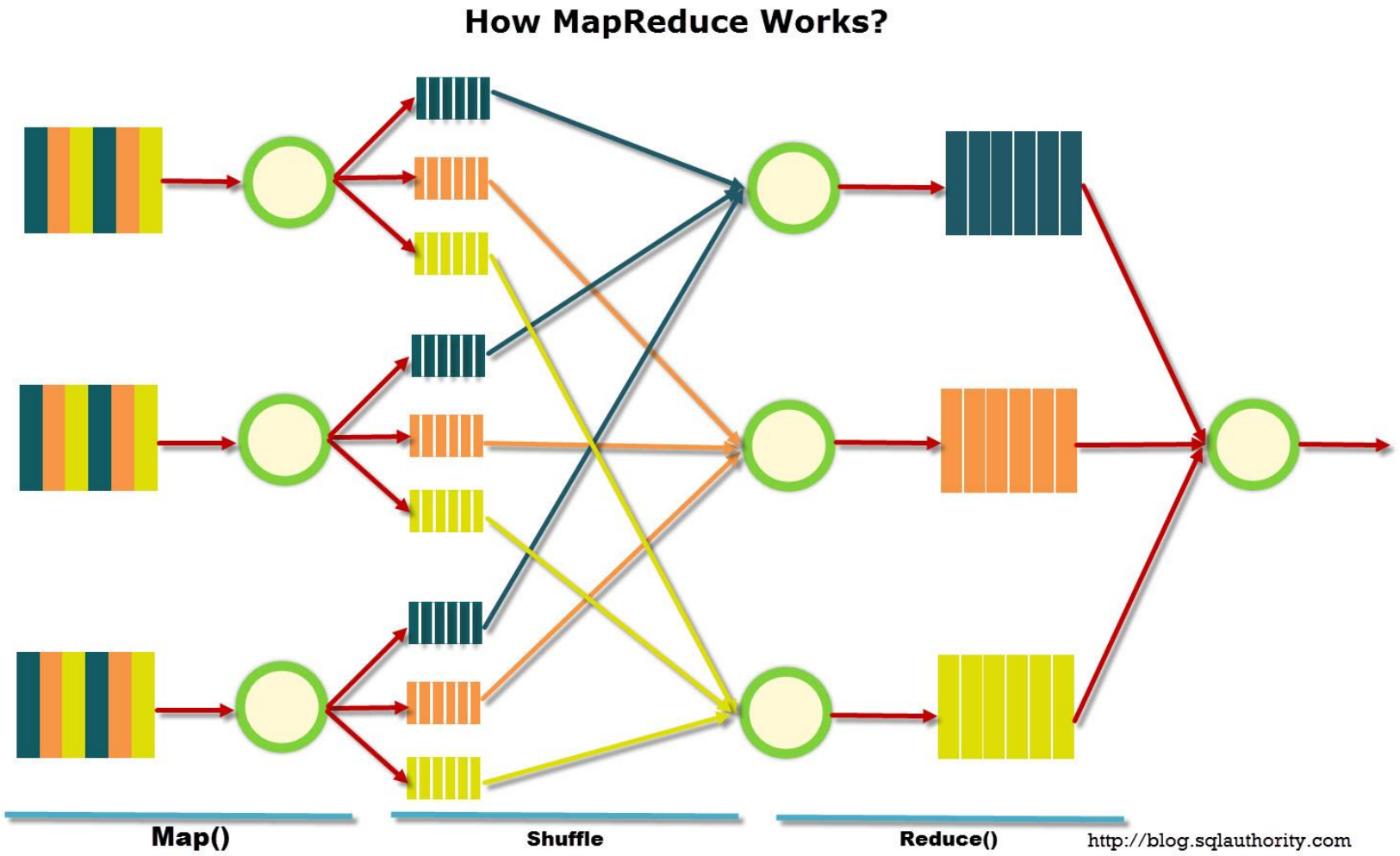
Reduce

Key idea:
provide a functional abstraction for these two operations

Data Structures

- ***<key, value>*** pairs are the basic data structure in MapReduce
  - Keys and values can be: integers, float, strings, raw bytes
  - They can also be arbitrary data structures
- The design of MapReduce algorithms involves:
  - Imposing the key-value structure on arbitrary datasets
    - E.g.: for a collection of Web pages, input keys may be URLs and values may be the HTML content
  - In some algorithms, input keys are not used (e.g., wordcount), in others they uniquely identify a record
  - Keys can be combined in complex ways to design various algorithms

A Brief View of MapReduce

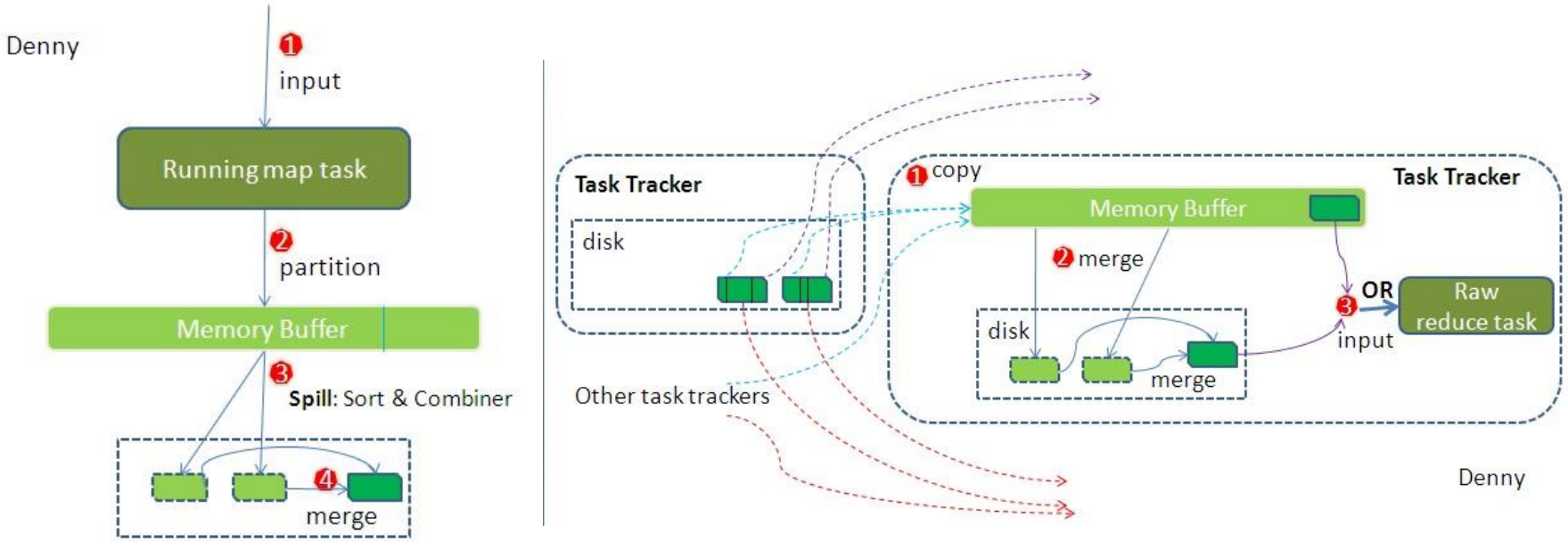


Map and Reduce Functions

- Programmers specify two functions:
  - **map (k1, v1) → list [<k2, v2>]**
    - Map transforms the input into key-value pairs to process
  - **reduce (k2, list [v2]) → [<k3, v3>]**
    - Reduce aggregates the list of values for each key
      - All values with the same key are sent to the same reducer
      - list [<k2, v2>] will be grouped according to key k2 as (k2, list [v2])
- The MapReduce environment takes in charge of everything else…
- A complex program can be decomposed as a succession of Map and Reduce tasks

Shuffle & Sort Functions

- Shuffle
  - Input to the Reducer is the sorted output of the mappers. In this phase the framework fetches the relevant partition of the output of all the mappers, via HTTP.
- Sort
  - The framework groups Reducer inputs by keys (since different Mappers may have output the same key) in this stage.



http://langyu.iteye.com/blog/992916

Brief Data Flow

- Data distribution:
  1. Input files are split into M pieces on **distributed file system** - 128 MB blocks
  2. Intermediate files created from map tasks are written to **local disk**
  3. Output files are written to **distributed file system**
- Assigning tasks
  - Many copies of user program are started
  - Tries to utilize data localization by running map tasks on machines with data
  - One instance becomes the Master
  - Master finds idle machines and assigns them tasks

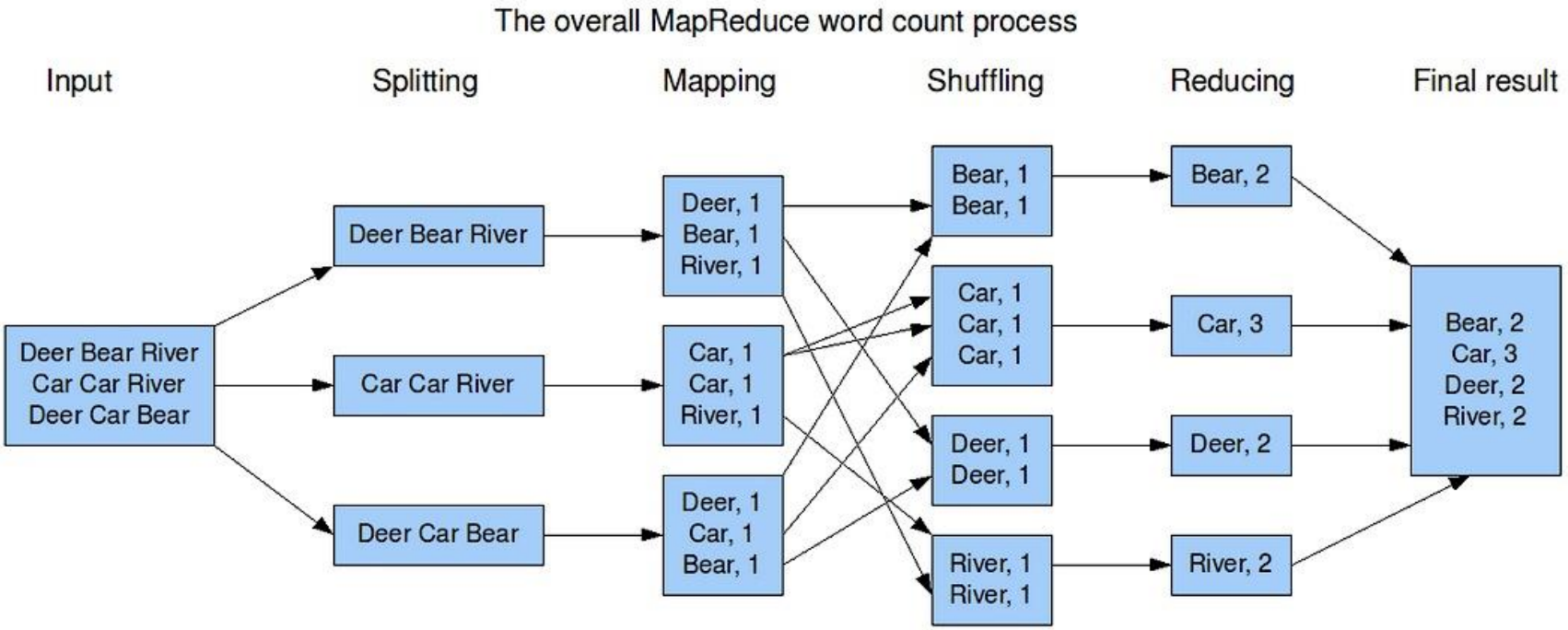
Brief Work Flow

- Application writer specifies
  - A pair of functions called Map and Reduce and a set of input files and submits the job
- Workflow
  - Input phase generates a number of FileSplits from input files (one per Map task)
  - The Map phase executes a user function to transform input kv-pairs into a new set of kv-pairs
  - The framework sorts & Shuffles the kv-pairs to output nodes
  - The Reduce phase combines all kv-pairs with the same key into new kv-pairs
  - The output phase writes the resulting pairs to files
- All phases are distributed with many tasks doing the work
  - Framework handles scheduling of tasks on cluster
  - Framework handles recovery when a node fails

Everything Else?

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles “data distribution”
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed file system (HDFS)
- You don’t know:
  - Where mappers and reducers run
  - When a mapper or reducer begins or finishes
  - Which input a particular mapper is processing
  - Which intermediate key a particular reducer is processing

"Hello World" in MapReduce



"Hello World" in MapReduce

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ doc d do
4:             Emit(term t, count 1)

1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count s)
```

"Hello World" in MapReduce

- Input:
  - Key-value pairs: (docid, doc) of a file stored on the distributed filesystem
  - docid : unique identifier of a document
  - doc: is the text of the document itself
- Mapper:
  - Takes an input key-value pair, tokenize the line
  - Emits intermediate key-value pairs: the word is the key and the integer is the value
- The framework:
  - Guarantees all values associated with the same key (the word) are brought to the same reducer
- The reducer:
  - Receives all values associated to some keys
  - Sums the values and writes output key-value pairs: the key is the word and the value is the number of occurrences

```
public class WordCount {

  public static class Map 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 tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken());
        output.collect(word, one);
}}}
```

Mapper

```
public static class Reduce 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));
}}
```

Reducer

```
  public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);
    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    JobClient.runJob(conf);
  }}
```

Run this program as a MapReduce job

Where the Magic Happens

- Implicit between the map and reduce phases is a parallel "group by" operation on intermediate keys
  - Intermediate data arrive at each reducer in order, sorted by the key
  - No ordering is guaranteed across reducers
- Output keys from reducers are written back to HDFS
  - The output may consist of *r* distinct files, where *r* is the number of reducers
  - Such output may be the input to a subsequent MapReduce phase
- Intermediate keys (used in shuffle and sort) are transient:
  - They are not stored on the distributed filesystem
  - They are "spilled" to the local disk of each machine in the cluster

Write your own WordCount in Java

Section Outline

- Program overview
- Mapper
- Reducer
- Main (The Driver)
- Combiner
- Partitioner
- Serialization and Writable
- Counters
- Input and Output
- Compression

Write your own WordCount in Java

Program Overview

MapReduce Program

- A MapReduce program consists of the following 3 parts:
  - Driver → main (would trigger the map and reduce methods)
  - Mapper
  - Reducer
- It is better to include the map reduce and main methods in 3 different classes

- Check detailed information of all classes at: https://hadoop.apache.org/docs/r2.7.2/api/allclasses-noframe.html

Write your own WordCount in Java

Mapper

Mapper

```
public static class TokenizerMapper
    extends Mapper<Object, Text, Text, IntWritable>{
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(Object key, Text value, Context context) throws
                IOException, InterruptedException {
            StringTokenizer itr = new
                    StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                    word.set(itr.nextToken());
                    context.write(word, one);
            }
        }
}
```

Mapper Explanation

- Maps input key/value pairs to a set of intermediate key/value pairs.

//Map class header

public static class TokenizerMapper

extends Mapper<Object, Text, Text, IntWritable>{

- Class Mapper<KEYIN,VALUEIN,KEYOUT,VALUEOUT>
  - KEYIN,VALUEIN -> (k1, v1) -> (docid, doc)
  - KEYOUT,VALUEOUT ->(k2, v2) -> (word, 1)

// IntWritable: A serializable and comparable object for integer

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

//Text: stores text using standard UTF8 encoding. It provides methods to serialize, deserialize, and compare texts at byte level

private Text word = new Text();

//hadoop supported data types for the key/value pairs, in package org.apache.hadoop

Mapper Explanation (Cont')

//Map method header

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

- Object key/Text value: Data type of the input Key and Value to the mapper
- Context: An inner class of Mapper, used to store the context of a running task. Here it is used to collect data output by either the Mapper or the Reducer, i.e. intermediate outputs or the output of the job
- Exceptions: IOException, InterruptedException
- This function is called once for each key/value pair in the input split. Your application should override this to do your job.

Mapper Explanation (Cont')

```
//Use a string tokenizer to split the document into words
StringTokenizer itr = new StringTokenizer(value.toString());
//Iterate through each word and a form key value pairs
while (itr.hasMoreTokens()) {
//Assign each work from the tokenizer(of String type) to a Text 'word'
        word.set(itr.nextToken());
//Form key value pairs for each word as <word, one> using context
        context.write(word, one);
}
```

- Map function produces Map.Context object
  - Map.context() takes *(k, v)* elements
- Any *(WritableComparable, Writable)* can be used

Write your own WordCount in Java

Reducer

Reducer

```
public static class IntSumReducer
    extends Reducer<Text,IntWritable,Text,IntWritable> {
        private IntWritable result = new IntWritable();

        public void reduce(Text key, Iterable<IntWritable> values,
        Context context) throws IOException, InterruptedException{
                int sum = 0;
                for (IntWritable val : values) {
                        sum += val.get();
                }
                result.set(sum);
                context.write(key, result);
        }
}
```

Reducer Explanation

```
//Reduce Header similar to the one in map with different key/value data type
public static class IntSumReducer
    extends Reducer<Text, IntWritable, Text, IntWritable>
//data from map will be <"word",{1,1,..}>, so we get it with an Iterator and thus we can go through the sets of values
public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException{
//Initaize a variable 'sum' as 0
        int sum = 0;
//Iterate through all the values with respect to a key and sum up all of them
        for (IntWritable val : values) {
            sum += val.get();
        }
// Form the final key/value pairs results for each word using context
        result.set(sum);
        context.write(key, result);
}
```

Write your own WordCount in Java

Main (The Driver)

Main (The Driver)

- Given the Mapper and Reducer code, the short main() starts the MapReduction running
- The Hadoop system picks up a bunch of values from the command line on its own
- Then the main() also specifies a few key parameters of the problem in the Job object
- Job is the primary interface for a user to describe a map-reduce job to the Hadoop framework for execution (such as what Map and Reduce classes to use and the format of the input and output files)
- Other parameters, i.e. the number of machines to use, are optional and the system will determine good values for them if not specified
- Then the framework tries to faithfully execute the job as-is described by Job

Main (The Driver)

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

Main Explanation

//Creating a Configuration object and a Job object, assigning a job name for identification purposes

Configuration conf = new Configuration();

Job job = Job.getInstance(conf, "word count");

- Job Class: It allows the user to configure the job, submit it, control its execution, and query the state. Normally the user creates the application, describes various facets of the job via Job and then submits the job and monitor its progress.

//Setting the job's jar file by finding the provided class location

job.setJarByClass(WordCount.class);

//Providing the mapper and reducer class names

job.setMapperClass(TokenizerMapper.class);

job.setReducerClass(IntSumReducer.class);

//Setting configuration object with the Data Type of output Key and Value for map and reduce

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

Main Explanation (Cont')

```
//The hdfs input and output directory to be fetched from the command line
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
//Submit the job to the cluster and wait for it to finish
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

Make It Running !

- Configure environment variables

export JAVA_HOME=…

export PATH=${JAVA_HOME}/bin:${PATH}

export HADOOP_CLASSPATH=${JAVA_HOME}/lib/tools.jar

- Compile WordCount.java and create a jar:

$ hadoop com.sun.tools.javac.Main WordCount.java

$ jar cf wc.jar WordCount*.class

- Put files to HDFS

$ hdfs dfs –put YOURFILES input

- Run the application

$ hadoop jar wc.jar WordCount input output

- Check the results

$ hdfs dfs –cat output/*

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

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

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
```

Make It Running !

- Given two files:
  - file1: Hello World Bye World
  - file2: Hello Hadoop Goodbye Hadoop
- The first map emits:
  - < Hello, 1> < World, 1> < Bye, 1> < World, 1>
- The second map emits:
  - < Hello, 1> < Hadoop, 1> < Goodbye, 1> < Hadoop, 1>
- The output of the job is:
  - < Bye, 1> < Goodbye, 1> < Hadoop, 2> < Hello, 2> < World, 2>

Write your own WordCount in Java

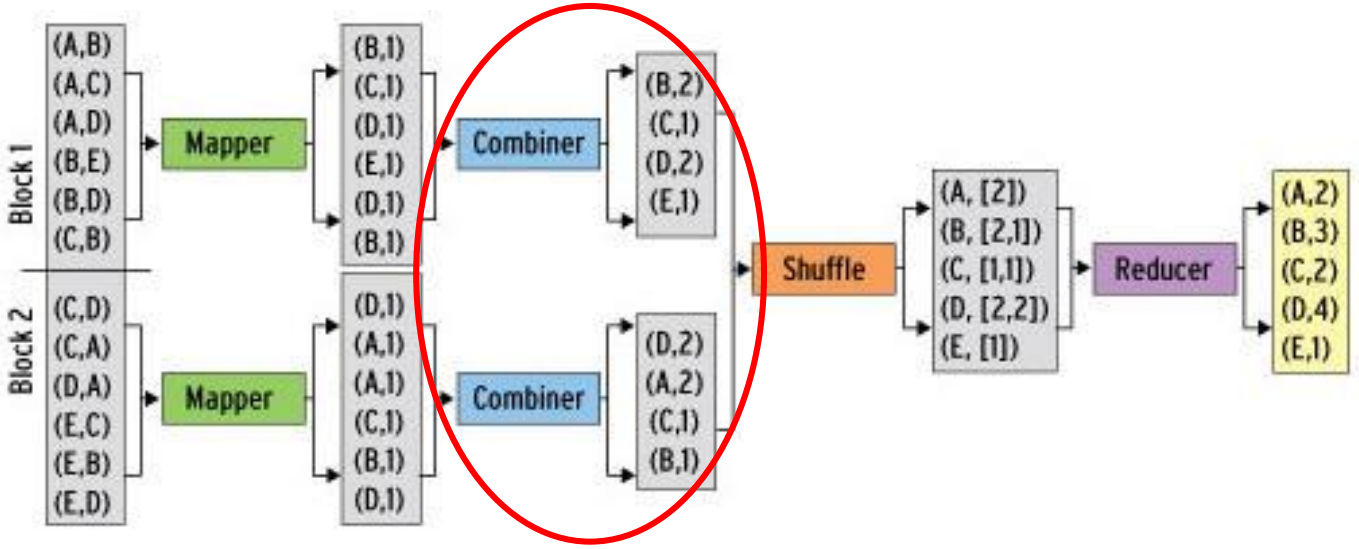
Combiner

Combiners

- Often a Map task will produce many pairs of the form (k,v_1), (k,v_2), … for the same key k
  - E.g., popular words in the word count example
- Combiners are a general mechanism to reduce the amount of intermediate data, thus saving network cost
  - They could be thought of as "mini-reducers"
- Warning!
  - The use of combiners must be thought carefully
    - Optional in Hadoop: the correctness of the algorithm cannot depend on computation (or even execution) of the combiners
    - A combiner operates on each map output key. It must have the same output key-value types as the Mapper class.
    - A combiner can produce summary information from a large dataset because it replaces the original Map output
  - Works only if reduce function is commutative and associative
    - In general, reducer and combiner are not interchangeable

Combiners in WordCount

- Combiner combines the values of all keys of a single mapper node (single machine):



- Much less data needs to be copied and shuffled!
- If combiners take advantage of all opportunities for local aggregation we have at most $m \times V$ intermediate key-value pairs
  - m: number of mappers
  - V: number of unique terms in the collection
- Note: not all mappers will see all terms

Combiners in WordCount

- In WordCount.java, you only need to add the follow line to Main:

  job.setCombinerClass(IntSumReducer.class);

  - This is because in this example, Reducer and Combiner do the same thing
  - Note: Most cases this is not true!
  - You need to write an extra combiner class
- Given two files:
  - file1: Hello World Bye World
  - file2: Hello Hadoop Goodbye Hadoop
- The first map emits:
  - < Hello, 1> < World, 2> < Bye, 1>
- The second map emits:
  - < Hello, 1> < Hadoop, 2> < Goodbye, 1>

Write your own WordCount in Java

Partitioner

Partitioner

- Partitioner controls the partitioning of the keys of the intermediate map-outputs.
  - The key (or a subset of the key) is used to derive the partition, typically by a *hash function*.
  - The total number of partitions is the same as the number of reduce tasks for the job.
    - This controls which of the m reduce tasks the intermediate key (and hence the record) is sent to for reduction.
- System uses HashPartitioner by default:
  - hash(key) mod R
- Sometimes useful to override the hash function:
  - E.g., ***hash(hostname(URL)) mod R*** ensures URLs from a host end up in the same output file
    - https://www.unsw.edu.au/faculties and https://www.unsw.edu.au/about-us will be stored in one file
- Job sets Partitioner implementation (in Main)

MapReduce: Recap

- Programmers must specify:
  - map $(k_1, v_1) \rightarrow [(k_2, v_2)]$
  - reduce $(k_2, [v_2]) \rightarrow [<k_3, v_3>]$
  - All values with the same key are reduced together
- Optionally, also:
  - combine $(k_2, [v_2]) \rightarrow [<k_3, v_3>]$
    - Mini-reducers that run in memory after the map phase
    - Used as an optimization to reduce network traffic
  - partition $(k_2$, number of partitions$) \rightarrow$ partition for k_2
    - Often a simple hash of the key, e.g., hash(k_2) mod n
    - Divides up key space for parallel reduce operations
- The execution framework handles everything else…

MapReduce: Recap (Cont')

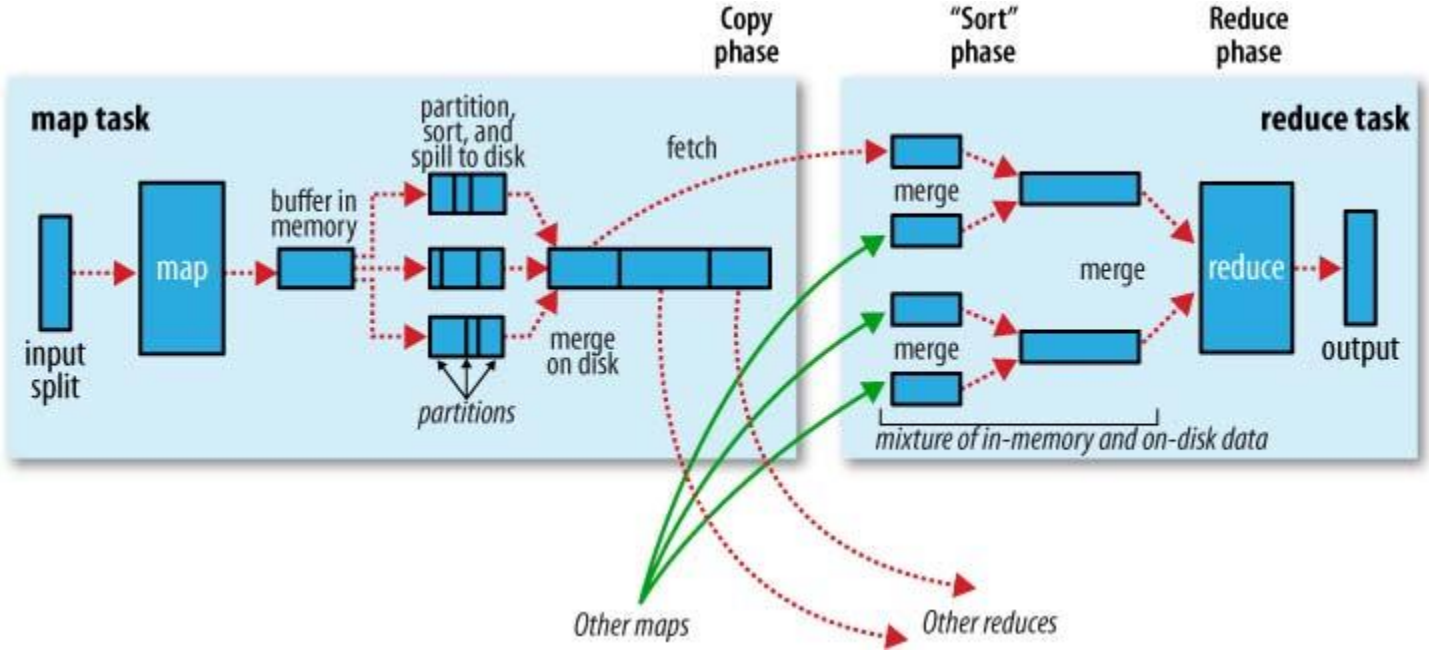
- Divides input into fixed-size pieces, ***input splits***
  - Hadoop creates one map task for each split
  - Map task runs the user-defined map function for each *record* in the split
- Size of splits
  - Small size is better for load-balancing: faster machine will be able to process more splits
  - But if splits are too small, the overhead of managing the splits dominate the total execution time
  - For most jobs, a good split size tends to be the size of a HDFS block, 64MB(default)
- Data locality optimization
  - Run the map task on a node where the input data resides in HDFS
  - This is the reason why the split size is the same as the block size

MapReduce: Recap (Cont')

- Map tasks write their output to local disk (not to HDFS)
  - Map output is intermediate output
  - Once the job is complete the map output can be thrown away
  - So storing it in HDFS with replication would be overkill
  - If the node of map task fails, Hadoop will automatically rerun the map task on another node
- Reduce tasks don't have the advantage of data locality
  - Input to a single reduce task is normally the output from all mappers
  - Output of the reduce is stored in HDFS for reliability
- The number of reduce tasks is not governed by the size of the input, but is specified independently

MapReduce: Recap (Cont')

- When there are multiple reducers, the map tasks partition their output:
  - One partition for each reduce task
  - The records for every key are all in a single partition
  - Partitioning can be controlled by a user-defined partitioning function



Write your own WordCount in Java

Serialization and Writable

Serialization

- Process of turning structured objects into a byte stream for transmission over a network or for writing to persistent storage
- Deserialization is the reverse process of serialization
- Requirements
  - Compact
    - To make efficient use of storage space
  - Fast
    - The overhead in reading and writing of data is minimal
  - Extensible
    - We can transparently read data written in an older format
  - Interoperable
    - We can read or write persistent data using different language

Writable Interface

- Hadoop defines its own “box” classes for strings (Text), integers (IntWritable), etc.
- Writable is a serializable object which implements a simple, efficient, serialization protocol

```
public interface Writable {
    void write(DataOutput out) throws IOException;
    void readFields(DataInput in) throws IOException;
}
```

- All values must implement interface Writable
- All keys must implement interface WritableComparable
- context.write(WritableComparable, Writable)
  - You cannot use java primitives here!!

Writable Wrappers

- There are **Writable** wrappers for all the Java primitive types except shot and char (both of which can be stored in an **IntWritable**)
- **get()** for retrieving and **set()** for storing the wrapped value
- Variable-length formats
  - If a value is between -122 and 127, use only a single byte
  - Otherwise, use first byte to indicate whether the value is positive or negative and how many bytes follow

Java Primitive	Writable Implementation	Serialized Size (bytes)
boolean	BooleanWritable	1
byte	ByteWritable	1
int	IntWritable	4
	VIntWritable	1~5
float	FloatWritable	4
long	LongWritable	8
	VLongWritable	1~9
double	DoubleWritable	8

Writable Examples

- Text
  - Writable for UTF-8 sequences
  - Can be thought of as the Writable equivalent of java.lang.String
  - Maximum size is 2GB
  - Use standard UTF-8
  - Text is mutable (like all Writable implementations, except NullWritable)
    - Different from java.lang.String
    - You can reuse a Text instance by calling one of the set() method
- NullWritable
  - Zero-length serialization
  - Used as a placeholder
  - A key or a value can be declared as a **NullWritable** when you don't need to use that position

Stripes Implementation

- A stripe key-value pair *a → { b: 1, c: 2, d: 5, e: 3, f: 2 }*:
  - Key: the term *a*
  - Value: the stripe *{ b: 1, c: 2, d: 5, e: 3, f: 2 }*
    - *In Java, easy, use map (hashmap)*
    - *How to represent this stripe in MapReduce?*
- MapWritable: the wrapper of Java map in MapReduce
  - put(Writable key, Writable value)
  - get(Object key)
  - containsKey(Object key)
  - containsValue(Object value)
  - entrySet()， returns Set<Map.Entry<Writable,Writable>>, used for iteration
- More details please refer to https://hadoop.apache.org/docs/r2.7.2/api/org/apache/hadoop/io/MapWritable.html

Pairs Implementation

- Key-value pair (a, b) → count
  - Value: count
  - Key: (a, b)
    - In Java, easy, implement a pair class
    - *How to store the key in MapReduce?*

- You must customize your own key, which must implement interface WritableComparable!

- First start from a easier task: when the value is a pair, which must implement interface Writable

Multiple Output Values

- If we are to output multiple values for each key
  - E.g., a pair of String objects, or a pair of int
- How do we do that?
- WordCount output a single number as the value
- Remember, our object containing the values needs to implement the Writable interface
- We could use Text
  - Value is a string of comma separated values
  - Have to convert the values to strings, build the full string
  - Have to parse the string on input (not hard) to get the values

Implement a Custom Writable

- Suppose we wanted to implement a custom class containing a pair of String objects. Call it StringPair.
- How would we implement this class?
  - Needs to implement the WritableComparable interface
  - Instance variables to hold the values
  - Construct functions
  - A method to set the values (two String objects)
  - A method to get the values (two String objects)
  - write() method： serialize the member variables (i.e., two String) objects in turn to the output stream
  - readFields() method: deserialize the member variables (i.e., two String) in turn from the input stream
  - As in Java: hashCode(), equals(), toString()
  - compareTo() method: specify how to compare two objects of the self-defind class

Implement a Custom Writable

- Implement the Writable interface

```
public class IntPair implements Writable {
```

- Instance variables to hold the values

```
    private int first, second;
```

- Construct functions

```
    public IntPair() {
    }

    public IntPair(int first, int second) {
        set(first, second);
    }
```

- set() method

```
    public void set(int left, int right) {
        first = left;
        second = right;
    }
```

Implement a Custom Writable

- get() method

```
public int getFirst() {
    return first;
}
public int getSecond() {
    return second;
}
```

- write() method

```
public void write(DataOutput out) throws IOException {
    out.writeInt(first);
    out.writeInt(second);
}
```

  - Write the two integers to the output stream in turn

- readFields() method

```
public void readFields(DataInput in) throws IOException {
    first = in.readInt();
    second = in.readInt();
}
```

  - Read the two integers from the input stream in turn

Implement a Custom WritableComparable

- In some cases such as secondary sort, we also need to override the hashCode() method.
  - Because we need to make sure that all key-value pairs associated with the first part of the key are sent to the same reducer!

```
public int hashCode()
    return first.hashCode();
}
```

  - By doing this, partitioner will only use the hashCode of the first part.
  - You can also write a paritioner to do this job

Implement a Custom Writable

- In some cases such as secondary sort, we also need to override the hashCode() method.
  - Because we need to make sure that all key-value pairs associated with the first part of the key are sent to the same reducer!

```
public int hashCode()
    return first.hashCode();
}
```

  - By doing this, partitioner will only use the hashCode of the first part.
  - You can also write a paritioner to do this job

Write your own WordCount in Java

Counters

MapReduce Counters

- Instrument Job's metrics
  - Gather statistics
    - Quality control – confirm what was expected.
      - E.g., count invalid records
    - Application level statistics.
  - Problem diagnostics
  - Try to use counters for gathering statistics instead of log files
- Framework provides a set of built-in metrics
  - For example bytes processed for input and output
- User can create new counters
  - Number of records consumed
  - Number of errors or warnings

Built-in Counters

- Hadoop maintains some built-in counters for every job.
- Several groups for built-in counters
  - File System Counters – number of bytes read and written
  - Job Counters – documents number of map and reduce tasks launched, number of failed tasks
  - Map-Reduce Task Counters– mapper, reducer, combiner input and output records counts, time and memory statistics

User-Defined Counters

- You can create your own counters
  - Counters are defined by a Java enum
    - serves to group related counters
    - E.g.,

```
enum Temperature {
        MISSING,
        MALFORMED
}
```

- Increment counters in Reducer and/or Mapper classes
  - Counters are global: Framework accurately sums up counts across all maps and reduces to produce a grand total at the end of the job

Implement User-Defined Counters

- Retrieve Counter from Context object
  - Framework injects Context object into map and reduce methods
- Increment Counter's value
  - Can increment by 1 or more

```
parser.parse(value);
if (parser.isValidTemperature()) {
  int airTemperature = parser.getAirTemperature();
  context.write(new Text(parser.getYear()),
      new IntWritable(airTemperature));
} else if (parser.isMalformedTemperature()) {
  System.err.println("Ignoring possibly corrupt input: " + value);
  context.getCounter(Temperature.MALFORMED).increment(1);
} else if (parser.isMissingTemperature()) {
  context.getCounter(Temperature.MISSING).increment(1);
}
```

Implement User-Defined Counters

- Get Counters from a finished job in Java
  - Counter counters = job.getCounters()
- Get the counter according to name
  - Counter c1 = counters.findCounter(Temperature.MISSING)
- Enumerate all counters after job is completed

```
for (CounterGroup group : counters) {
        System.out.println("* Counter Group: " + group.getDisplayName() + " (" +
        group.getName() + ")");
        System.out.println("  number of counters in this group: " + group.size());
        for (Counter counter : group) {
                System.out.println("  - " + counter.getDisplayName() + ": " +
        counter.getName() + ": "+counter.getValue());
        }
}
```

Write your own WordCount in Java

Input and Output

MapReduce SequenceFile

- File operations based on binary format rather than text format
- SequenceFile class prvoides a persistent data structure for binary key-value pairs, e.g.,
  - Key: timestamp represented by a LongWritable
  - Value: quantity being logged represented by a Writable
- Use SequenceFile in MapReduce:
  - job.setinputFormatClass(SequenceFileOutputFormat.class);
  - job.setOutputFormatClass(SequenceFileOutputFormat.class);
  - In Mapreduce by default *TextInputFormat*

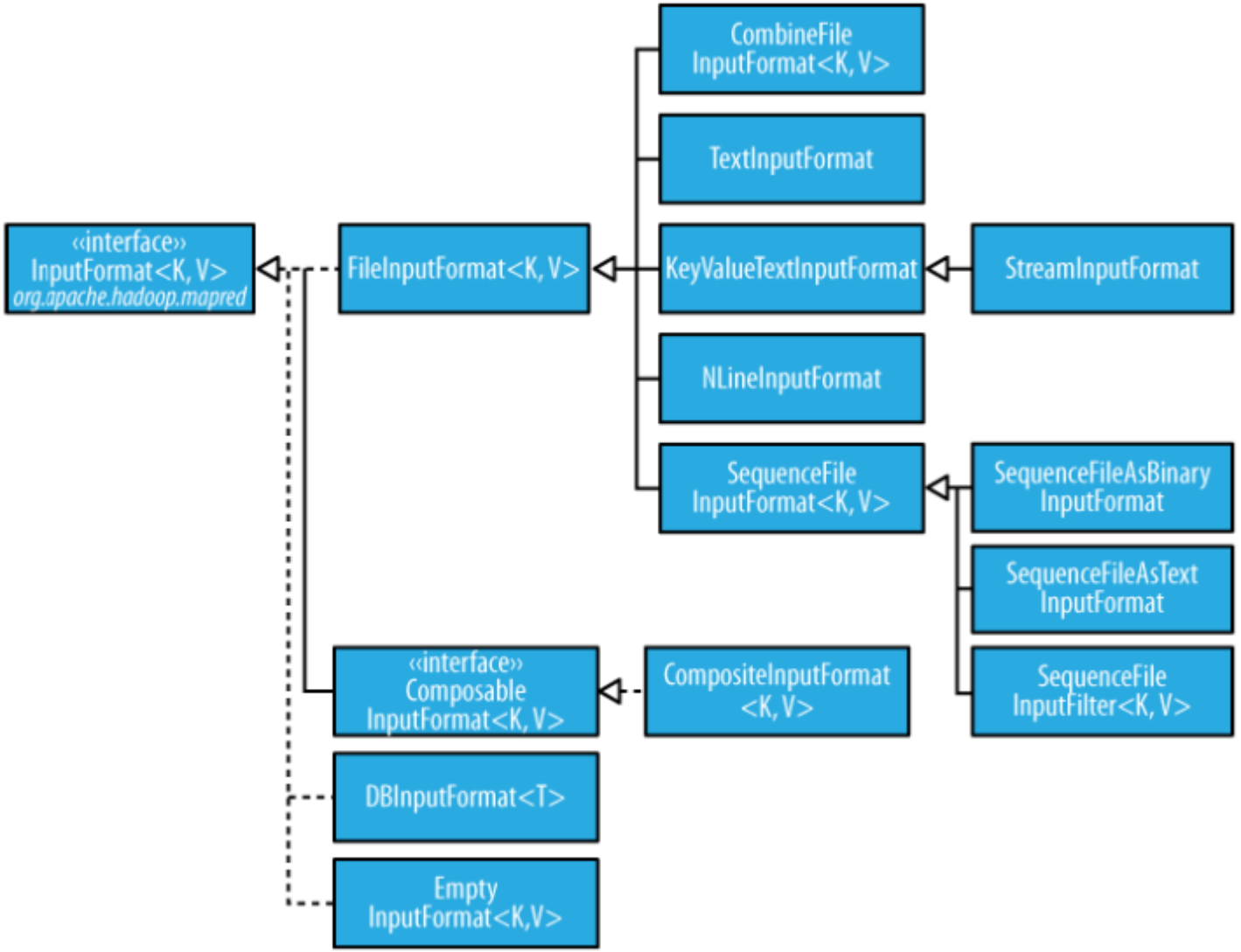
MapReduce Input Formats

- InputSplit
  - A **chunk** of the input processed by a single map
  - Each split is divided into records
  - Split is just a reference to the data (doesn't contain the input data)

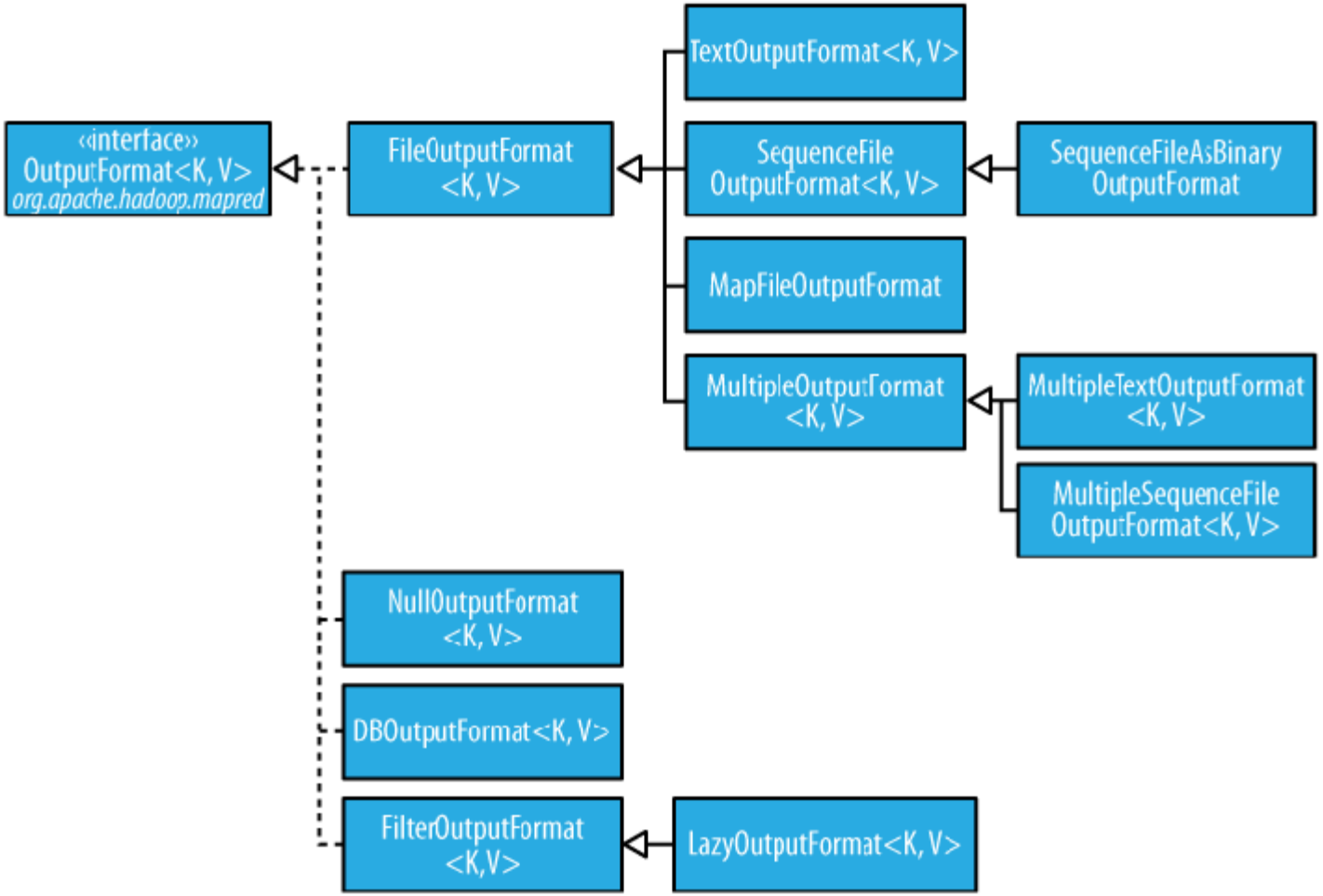
```
public interface InputSplit extends Writable {

  long getLength() throws IOException;

  String[] getLocations() throws IOException;

}
```

- RecordReader
  - Iterate over records
  - Used by the map task to generate record key-value pairs
- As a MapReduce application programmer, we do not need to deal with InputSplit directly, as they are created in InputFormat
- In MapReduce, by default TextInputFormat and LineRecordReader

MapReduce InputFormat



MapReduce OutputFormat



Write your own WordCount in Java

Compression

Compression

- Two major benefits of file compression
  - Reduce the space needed to store files
  - Speed up data transfer across the network

- When dealing with large volumes of data, both of these savings can be significant, so it pays to carefully consider how to use compression in Hadoop

Compression Formats

- Compression formats

Compression Format	Tool	Algorithm	Filename Extension	Multiple Files	Splittable
DEFLATE	N/A	DEFLATE	.deflate	NO	NO
gzip	gzip	DEFLATE	.gz	NO	NO
ZIP	zip	DEFLATE	.zip	YES	YES, at file boundaries
bzip2	bzip2	bzip2	.bz2	NO	YES
LZO	lzop	LZO	.lzo	NO	NO

- "Splittable" column
  - Indicates whether the compression format supports splitting
  - Whether you can seek to any point in the stream and start reading from some point further on
  - Splittable compression formats are especially suitable for MapReduce

Compression and Input Splits

- When considering how to compress data that will be processed by MapReduce, it is important to understand whether the compression format supports splitting
- Example of not-splitable compression problem
  - A file is a gzip-compressed file whose compressed size is 1 GB
  - Creating a split for each block won't work since it is impossible to start reading at an arbitrary point in the gzip stream, and therefore impossible for a map task to read its split independently of the others

Codecs

- A codec is the implementation of a compression-decompression algorithm

Compression Format	Hadoop Compression Codec
DEFLATE	org.apache.hadoop.io.compression.DefaultCodec
gzip	org.apache.hadoop.io.compression.GzipCodec
Bzip2	org.apache.hadoop.io.compression.BZip2Codec
LZO	com.hadoop.compression.lzo.LzopCodec

- CompressionCodec
  - createOutputStream(OutputStream out): create a CompressionOutputStream to which you write your uncompressed data to have it written in compressed form to the underlying stream
  - createInputStream(InputStream in): obtain a CompressionInputStream, which allows you to read uncompressed data from the underlying stream

Use Compression in MapReduce

- If the input files are compressed, they will be decompressed automatically as they are read by MapReduce, using the filename extension to determine which codec to use.
  - We do not need to do anything unless to guarantee that the filename extension is correct

- In order to compress the output of a MapReduce job, you need to:
  - Set the *mapreduce.output.fileoutputformat.compress* property to true
  - Set the *mapreduce.output.fileoutputformat.compress.codec* property to the classname of the compression codec
  - Use the static method of FileOutputFormat to set these properties

Use Compression in MapReduce

- In WordCount.java, you can do:

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    FileOutputFormat.setCompressOutput(job, true);
    FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

Implementations

Create Local Files

- Create folder “/home/hadoop/file” and 2 local files “file1.txt” and “file2.txt”

```
[hadoop@Master ~]$ ll
总用量 141372
-rw-r--r--. 1 hadoop hadoop 59468784  2月 27 03:31 hadoop-1.0.0.tar.gz
-rw-r--r--. 1 hadoop hadoop 85292206  2月 27 03:29 jdk-6u31-linux-i586.bin
[hadoop@Master ~]$ mkdir ~/file
[hadoop@Master ~]$ ll
总用量 141376
drwxrwxr-x. 2 hadoop hadoop     4096  3月  2 05:31 file
-rw-r--r--. 1 hadoop hadoop 59468784  2月 27 03:31 hadoop-1.0.0.tar.gz
-rw-r--r--. 1 hadoop hadoop 85292206  2月 27 03:29 jdk-6u31-linux-i586.bin
[hadoop@Master ~]$
```

```
[hadoop@Master ~]$ cd file
[hadoop@Master file]$ echo "Hello World" > file1.txt
[hadoop@Master file]$ echo "Hello Hadoop" > file2.txt
[hadoop@Master file]$ ll
总用量 8
-rw-rw-r--. 1 hadoop hadoop 12  3月  2 05:35 file1.txt
-rw-rw-r--. 1 hadoop hadoop 13  3月  2 05:36 file2.txt
[hadoop@Master file]$ more file1.txt
Hello World
[hadoop@Master file]$ more file2.txt
Hello Hadoop
[hadoop@Master file]$
```

Upload Local Files

- Create the input folder on HDFS and upload the local input files into the input folder

```
[hadoop@Master ~]$ hadoop fs -mkdir input
[hadoop@Master ~]$ hadoop fs -ls
Found 1 items
drwxr-xr-x   - hadoop supergroup          0 2012-03-02 05:41 /user/hadoop/input
[hadoop@Master ~]$
```

```
[hadoop@Master ~]$ hadoop fs -put ~/file/file*.txt input
[hadoop@Master ~]$ hadoop fs -ls input
Found 2 items
-rw-r--r--   1 hadoop supergroup         12 2012-03-02 05:45 /user/hadoop/input/file1.txt
-rw-r--r--   1 hadoop supergroup         13 2012-03-02 05:45 /user/hadoop/input/file2.txt
[hadoop@Master ~]$
```

Run Programs

- Compiled jar files "/usr/hadoop/hadoop-examples-1.0.0.jar“

- Runtime information

```
[hadoop@Master ~]$ hadoop jar /usr/hadoop/hadoop-examples-1.0.0.jar wordcount input output
12/03/02 06:07:36 INFO input.FileInputFormat: Total input paths to process : 2
12/03/02 06:07:36 INFO mapred.JobClient: Running job: job_201202292213_0002
12/03/02 06:07:37 INFO mapred.JobClient:  map 0% reduce 0%
12/03/02 06:07:51 INFO mapred.JobClient:  map 50% reduce 0%
12/03/02 06:07:52 INFO mapred.JobClient:  map 100% reduce 0%
12/03/02 06:08:06 INFO mapred.JobClient:  map 100% reduce 100%
12/03/02 06:08:11 INFO mapred.JobClient: Job complete: job_201202292213_0002
12/03/02 06:08:11 INFO mapred.JobClient: Counters: 29 ●●●
```

Check Results

- Check the output folder

```
[hadoop@Master ~]$ hadoop fs -ls output
Found 3 items
-rw-r--r--   1 hadoop supergroup          0 2012-03-02 06:08 /user/hadoop/output/_SUCCESS
drwxr-xr-x   - hadoop supergroup          0 2012-03-02 06:07 /user/hadoop/output/_logs
-rw-r--r--   1 hadoop supergroup         25 2012-03-02 06:07 /user/hadoop/output/part-r-00000
[hadoop@Master ~]$
```

- Check the output file

```
[hadoop@Master ~]$ hadoop fs -cat output/part-r-00000
Hadoop  1
Hello   2
World   1
[hadoop@Master ~]$
```

End of Chapter 3