COMP9313: Big Data Management

Lecturer: Xin Cao

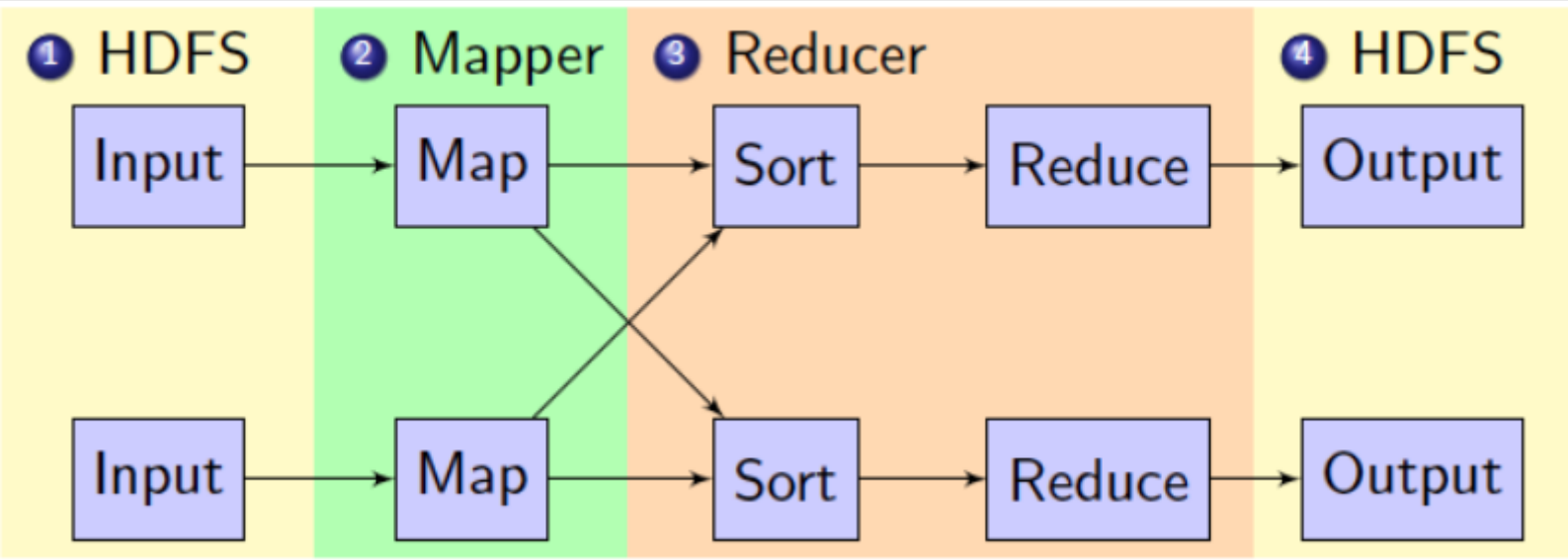
Course web site: http://www.cse.unsw.edu.au/~cs9313/

Chapter 2: MapReduce I

Hadoop MapReduce Brief Data Flow

1. Mappers read from HDFS
2. Map output is partitioned by key and sent to Reducers
3. Reducers sort input by key
4. Reduce output is written to HDFS

Intermediate results are stored on local FS of Map and Reduce workers



Data Structures in MapReduce

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

Map and Reduce Functions

Programmers specify two functions:

map $(k_1, v_1) \rightarrow$ list $[<k_2, v_2>]$

- Map transforms the input into key-value pairs to process

reduce $(k_2,$ list $[v_2]) \rightarrow [<k_3, v_3>]$

- Reduce aggregates the list of values for each key
- All values with the same key are sent to the same reducer

list $[<k_2, v_2>]$ will be grouped according to key k_2 as $(k_2,$ list $[v_2])$

The MapReduce environment takes in charge of everything else…

A complex program can be decomposed as a succession of Map and Reduce tasks

Understanding MapReduce

Map>>

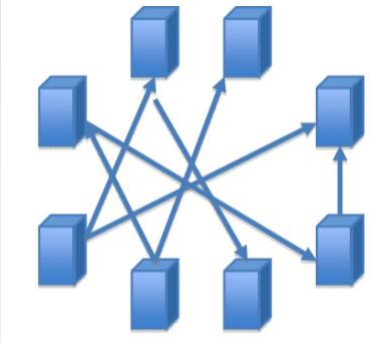
(K1, V1) →

- Info in
- Input Split

list (K2, V2)

- Key / Value out (intermediate values)
- One list per local node
- Can implement local Reducer (or Combiner)

Shuffle/Sort>>



Reduce

(K2, list(V2)) →

- Shuffle / Sort phase precedes Reduce phase
- Combines Map output into a list

list (K3, V3)

- Usually aggregates intermediate values

(input) <k1, v1> → **map** → <k2, v2> → **combine** → <k2, list(V2)> → **reduce** → <k3, v3> *(output)*

WordCount - Mapper

Let's count number of each word in documents (e.g., Tweets/Blogs)

Reads input pair <k1,v1>

- The input to the mapper is in format of <docID, docText>:

<D1,"Hello World" >,<D2,"Hello Hadoop Bye Hadoop" >

Outputs pairs <k2, v2>

- The output of the mapper is in format of <term, 1>:

<Hello, 1><World, 1><Hello, 1><Hadoop, 1><Bye, 1><Hadoop, 1>

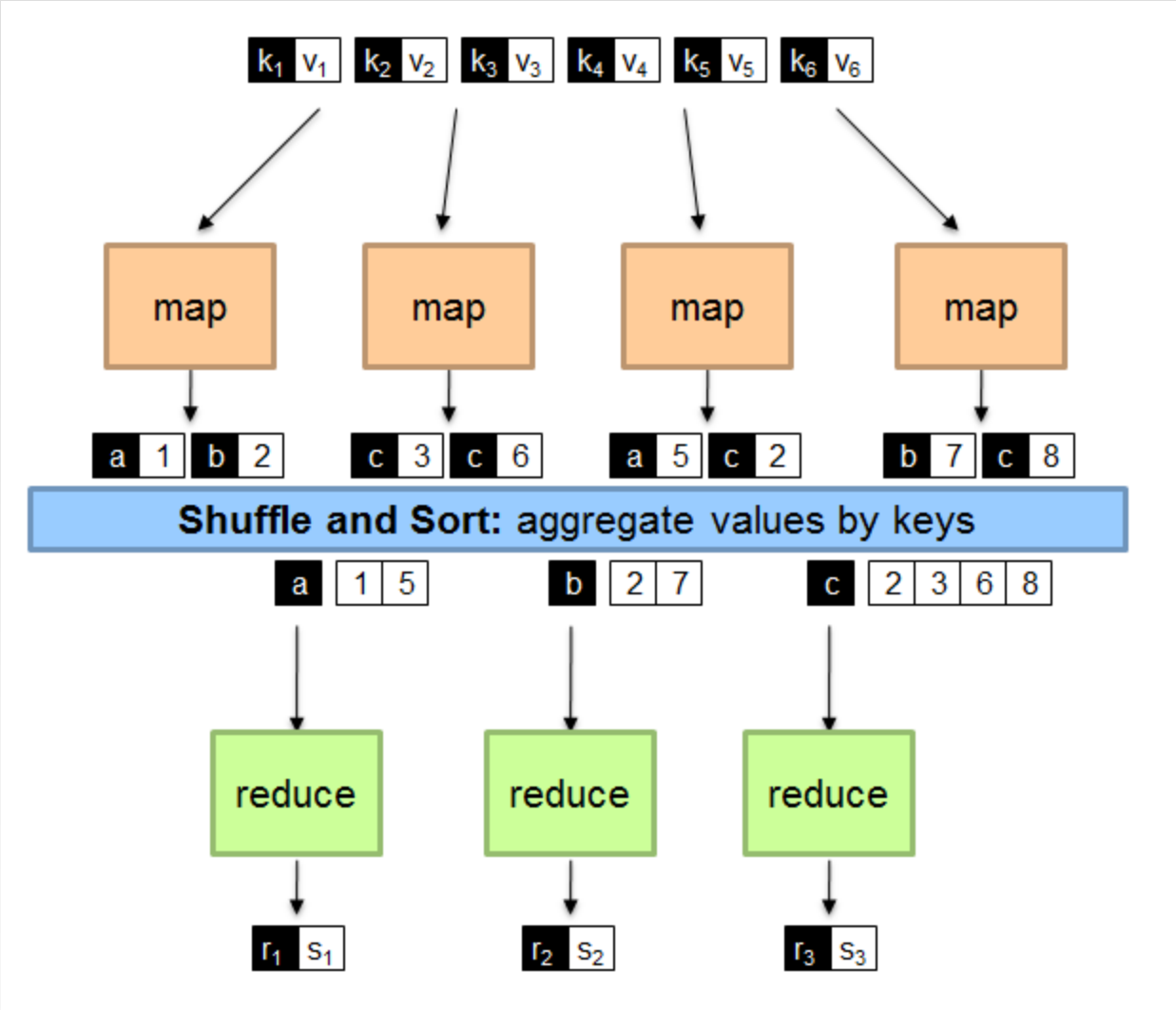
After shuffling and sort, reducer receives <k2, list(v2)>

<Hello, {1, 1}><World, {1}><Hadoop, {1, 1}><Bye, {1}>

The output is in format of <k3, v3>:

<Hello, 2><World, 1><Hadoop, 2><Bye, 1>

A Brief View of MapReduce



Shuffle and Sort

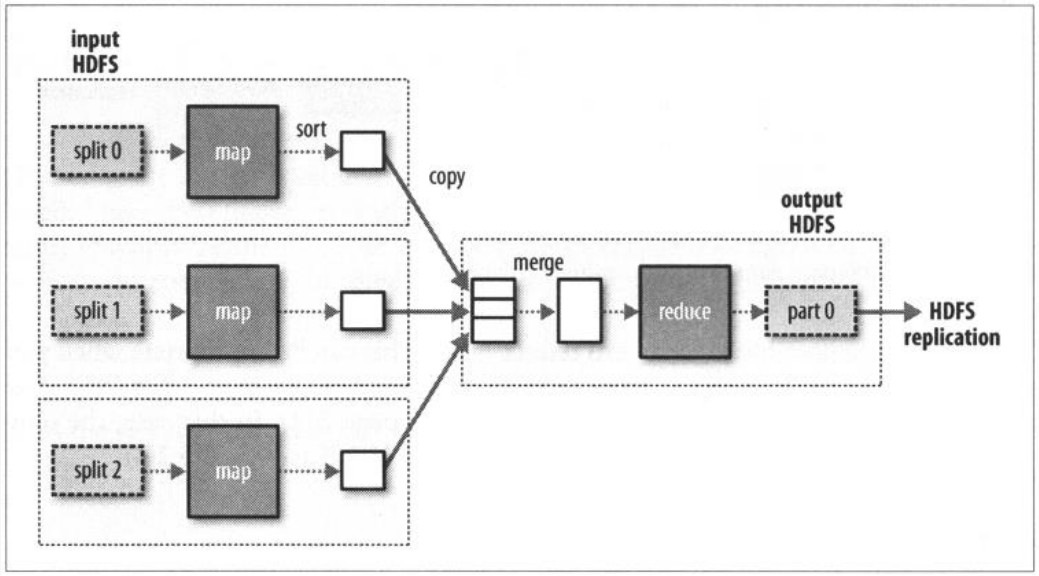
Shuffle

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.

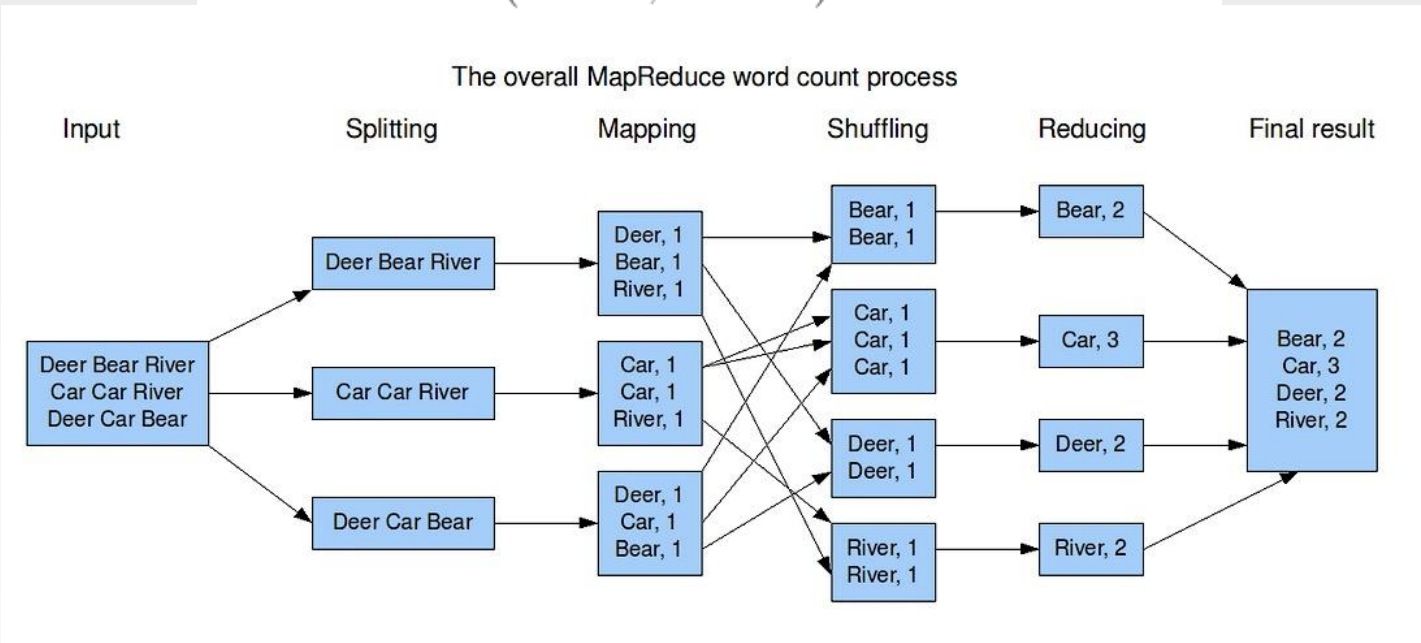
Hadoop framework handles the Shuffle and Sort step .



“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

Write Your Own WordCount in Java?

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

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
```

What is Writable?

Hadoop defines its own “box” classes for strings (Text), integers (IntWritable), etc.

All values must implement interface Writable

All keys must implement interface WritableComparable

Writable is a serializable object which implements a simple, efficient, serialization protocol

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

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);
```

Main (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(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 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);
  }
```

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 time

They could be thought of as “mini-reducers”

Warning!

The use of combiners must be thought carefully

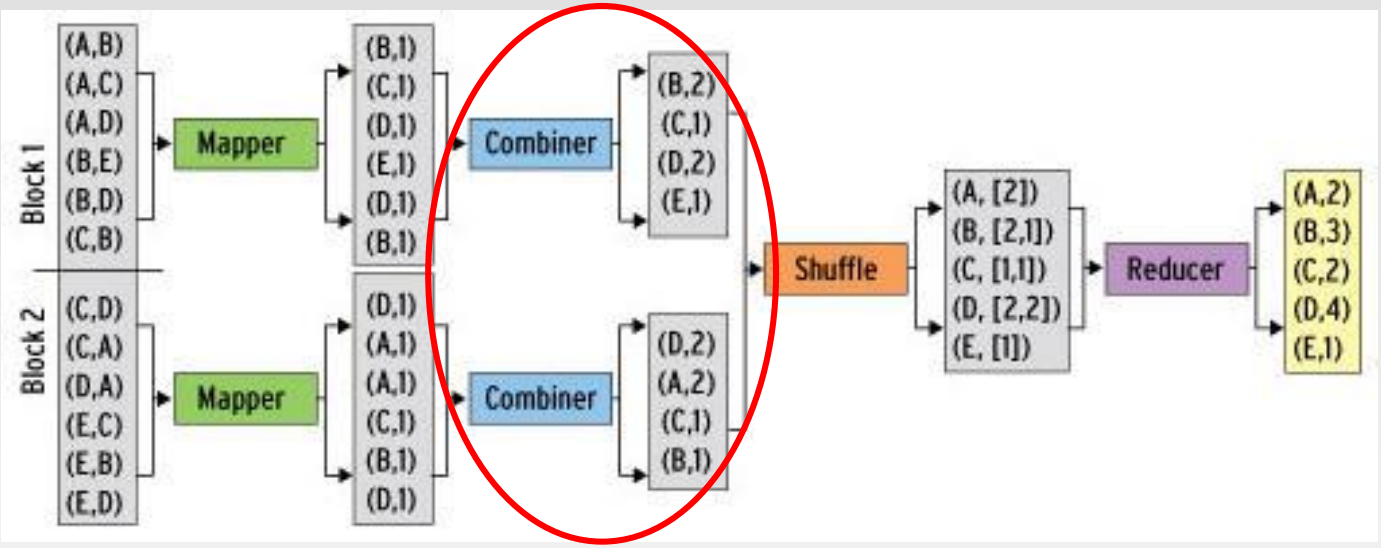
- Optional in Hadoop: the correctness of the algorithm cannot depend on computation (or even execution) of the combiners
- A combiner operates on each map output key. It must have the same output key-value types as the Mapper class.
- A combiner can produce summary information from a large dataset because 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 Bye Hadoop

The first map emits:

< Hello, 1> < World, 2> < Bye, 1>

The second map emits:

< Hello, 1> < Hadoop, 2> < Bye, 1>

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) → 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)

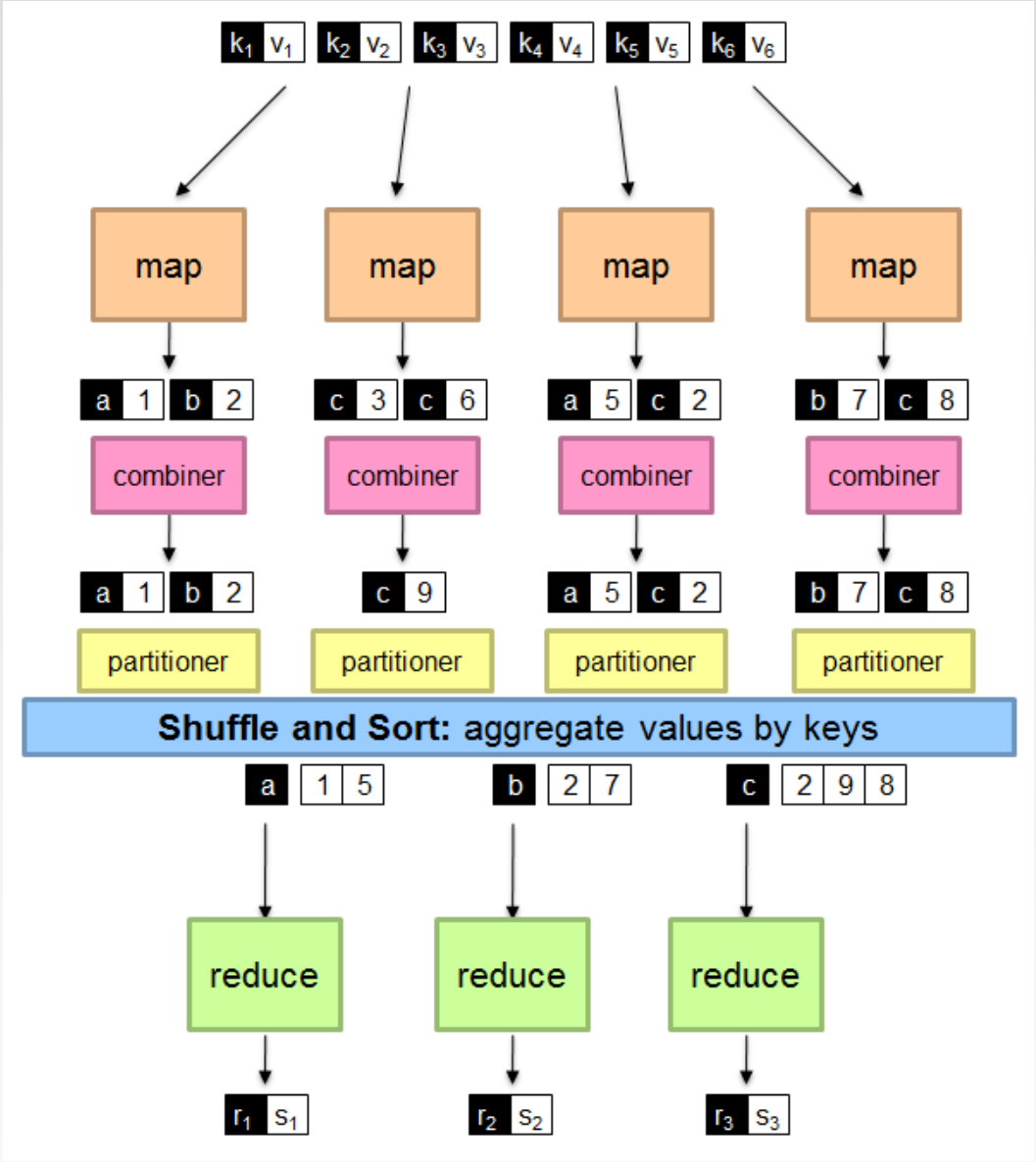
- Map output is intermediate output
- Once the job is complete the map output can be thrown away
- So storing it in HDFS with replication would be overkill
- If the node of map task fails, Hadoop will automatically rerun the map task on another node

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



Another Example: Analysis of Weather Dataset

Data from NCDC(National Climatic Data Center)

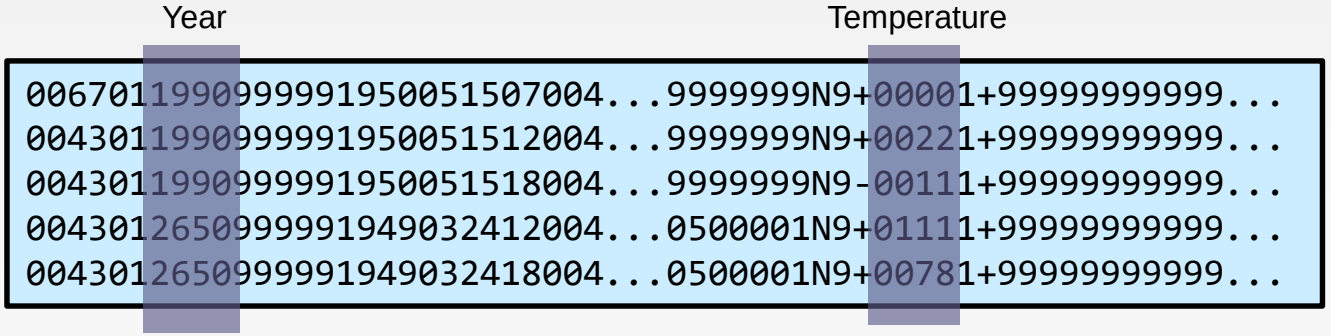
- A large volume of log data collected by weather sensors: e.g. temperature

Data format

- Line-oriented ASCII format
- Each record has many elements
- We focus on the temperature element
- Data files are organized by date and weather station
- There is a directory for each year from 1901 to 2001, each containing a gzipped file for each weather station with its readings for that year

Query

- What's the highest recorded global temperature for each year in the dataset?



Contents of data files

```
% ls raw/1990 | head
010010-99999-1990.gz
010014-99999-1990.gz
010015-99999-1990.gz
010016-99999-1990.gz
010017-99999-1990.gz
010030-99999-1990.gz
010040-99999-1990.gz
010080-99999-1990.gz
010100-99999-1990.gz
010150-99999-1990.gz
```

List of data files

Analyzing the Data with Unix Tools

To provide a performance baseline

Use *awk* for processing line-oriented data

Complete run for the century took **42 minutes** on a single EC2 High-CPU Extra Large Instance

```
#!/usr/bin/env bash
for year in all/*
do
  echo -ne `basename $year .gz`"\t"
  gunzip -c $year | \
    awk '{ temp = substr($0, 88, 5) + 0;
           q = substr($0, 93, 1);
           if (temp !=9999 && q ~ /[01459]/ && temp > max) max = temp }
         END { print max }'
done
```

```
% ./max_temperature.sh
1901    317
1902    244
1903    289
1904    256
1905    283
...
```

How Can We Parallelize This Work?

To speed up the processing, we need to run parts of the program in **parallel**

Challenges?

Divide the work into even distribution is not easy

- File size for different years varies

Combining the results is complicated

- Get the result from the maximum temperature for each chunk

We are still limited by the processing capacity of a single machine

- Some datasets grow beyond the capacity of a single machine

To use **multiple machines**, we need to consider a variety of complex problems

Coordination: Who runs the overall job?

Reliability: How do we deal with failed processes?

Hadoop can take care of these issues

MapReduce Design

We need to answer these questions:

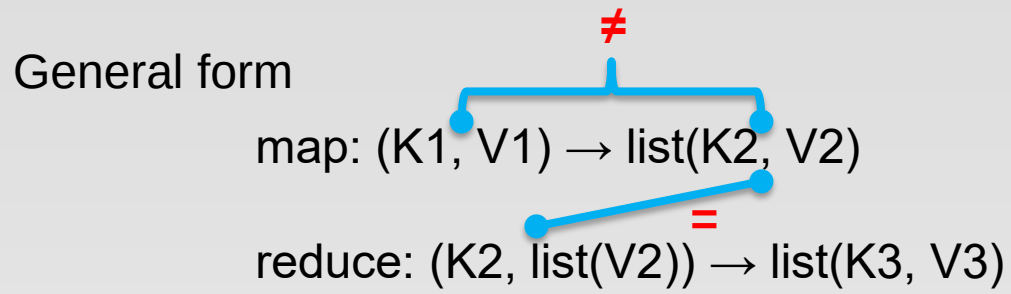
- What are the map input key and value types?
- What does the mapper do?
- What are the map output key and value types?
- Can we use a combiner?
- Is a partitioner required?
- What does the reducer do?
- What are the reduce output key and value types?

And: What are the file formats?

- For now we are using text files
- We may use binary files

MapReduce Types

General form



Combine function

```
map: (K1, V1) → list(K2, V2)
combine: (K2, list(V2)) → list(K2, V2)
reduce: (K2, list(V2)) → list(K3, V3)
```

The same form as the reduce function, except its output types

Output type is the same as Map

The combine and reduce functions may be the same

Partition function

```
partition: (K2, V2) → integer
```

Input intermediate key and value types

Returns the partition index

MapReduce Design

Identify the input and output of the problem

Text input format of the dataset files (input of mapper)

- Key: offset of the line (unnecessary)
- Value: each line of the files (string)

Output (output of reducer)

- Key: year (string or integer)
- Value: maximum temperature (integer)

Decide the MapReduce data types

Hadoop provides its own set of basic types

- optimized for network serialization
- org.apache.hadoop.io package

In WordCount, we have used Text and IntWritable

Key must implement interface WritableComparable

Value must implement interface Writable

Writable Wrappers

Java primitive	Writable implementation
boolean	BooleanWritable
byte	ByteWritable
short	ShortWritable
int	IntWritable VIntWritable
float	FloatWritable
long	LongWritable VLongWritable
double	DoubleWritable

Java class	Writable implementation
String	Text
byte[]	BytesWritable
Object	ObjectWritable
null	NullWritable

Java collection	Writable implementation
array	ArrayWritable ArrayPrimitiveWritable TwoDArrayWritable
Map	MapWritable
SortedMap	SortedMapWritable
enum	EnumSetWritable

What does the Mapper Do?

Pull out the year and the temperature

Indeed in this example, the map phase is simply data preparation phase

Drop bad records(filtering)

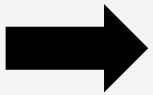
Input File

```
0067011990999991950051507004...9999999N9+00001+99999999999...
0043011990999991950051512004...9999999N9+00221+99999999999...
0043011990999991950051518004...9999999N9-00111+99999999999...
0043012650999991949032412004...0500001N9+01111+99999999999...
0043012650999991949032418004...0500001N9+00781+99999999999...
```

Input of Map Function (key, value)

```
(0, 0067011990999991950051507004...9999999N9+00001+99999999999...)
(106, 0043011990999991950051512004...9999999N9+00221+99999999999...)
(212, 0043011990999991950051518004...9999999N9-00111+99999999999...)
(318, 0043012650999991949032412004...0500001N9+01111+99999999999...)
(424, 0043012650999991949032418004...0500001N9+00781+99999999999...)
```

Map



Output of Map Function (key, value)

```
(1950, 0)
(1950, 22)
(1950, -11)
(1949, 111)
(1949, 78)
```

Map Input and Output

Input

- Key: offset of the line (unnecessary)
  - The dataset is quite large and contains a huge number of lines
  - LongWritable
- Value: each line of the files (string)
  - Text

Output

- Key: year
  - Both string or integer format
  - Text/IntWritable
- Value: temperature
  - Integer is already enough to store it
  - IntWritable

Combiner and Partitioner?

What does the Reducer Do?

Reducer input

(year, [temperature1, temperature2, temperature3, …])

Scan all values received for the key, and find out the maximum one

Reducer output

Key: year

- String/IntWritable

Value: maximum temperature

- IntWritable

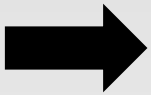
MapReduce Design of NCDC Example

The output from the map function is processed by MapReduce framework

Sorts and groups the key-value pairs by key

```
(1950, 0)
(1950, 22)
(1950, -11)
(1949, 111)
(1949, 78)
```

Sort and Group By



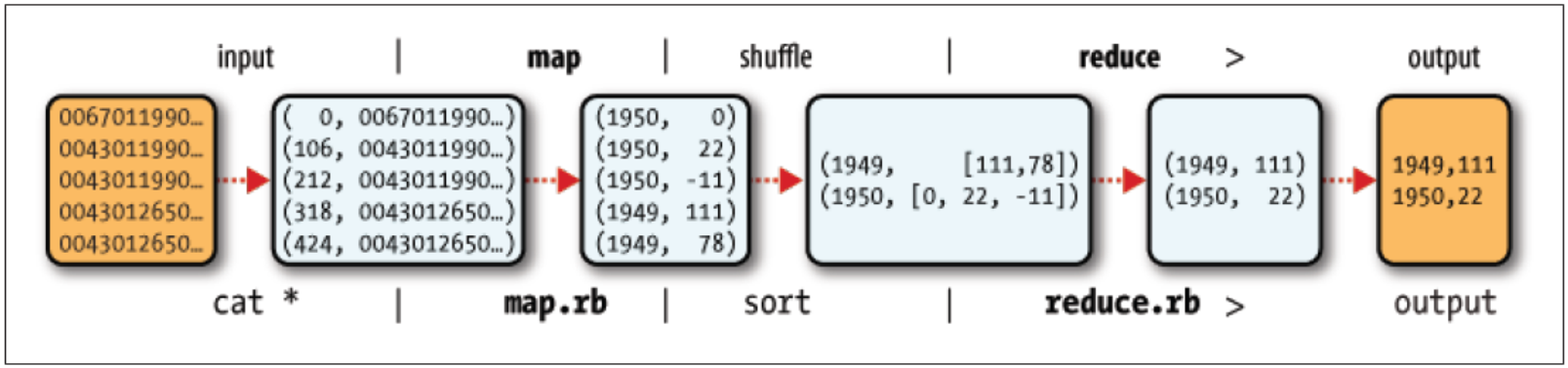
```
(1949, [111, 78])
(1950, [0, 22, -11])
```

- Reduce function iterates through the list and pick up the maximum value

```
(1949, [111, 78])
(1950, [0, 22, -11])
```

Reduce

```
(1949, 111)
(1950, 22)
```



Java Implementation of the Example

```
public class MaxTemperatureMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
  private static final int MISSING = 9999;

  @Override
  public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
    String line = value.toString();
    String year = line.substring(15, 19);
    int airTemperature;
    if (line.charAt(87) == '+') {
      airTemperature = Integer.parseInt(line.substring(88, 92));
    } else {
      airTemperature = Integer.parseInt(line.substring(87, 92));
    }
    String quality = line.substring(92, 93);
    if (airTemperature != MISSING && quality.matches("[01459]")) {
      context.write(new Text(year), new IntWritable(airTemperature));
    }
  }
}
```

Java Implementation of the Example

```
public class MaxTemperatureReducer
             extends Reducer<Text, IntWritable, Text, IntWritable> {
  @Override
  public void reduce(Text key, Iterable<IntWritable> values,
      Context context) throws IOException, InterruptedException {
    int maxValue = Integer.MIN_VALUE;
    for (IntWritable value : values) {
      maxValue = Math.max(maxValue, value.get());
    }
    context.write(key, new IntWritable(maxValue));
  }
}
```

Java Implementation of the Example

```
public class MaxTemperatureWithCombiner {
//specify the usage of the job
  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperatureWithCombiner <input path> " + "<output path>");
      System.exit(-1);
    }
//Construct a job object to configure, control and run the job
    Job job = new Job();
    job.setJarByClass(MaxTemperatureWithCombiner.class);
    job.setJobName("Max temperature");
//Specify input and output paths
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
//Specify map and reduce classes, also a combiner
    job.setMapperClass(MaxTemperatureMapper.class);
    job.setCombinerClass(MaxTemperatureReducer.class);
    job.setReducerClass(MaxTemperatureReducer.class);
//Specify output type
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
//submit the job and wait for completion
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Codes can be found here:
http://hadoopbook.com/code.html

For Large Datasets

Data stored in HDFS (organized as blocks)

Hadoop MapReduce 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 a split is normally the size of a HDFS block (e.g., 64Mb)

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
  - The largest size of the input that can be guaranteed to be stored on a single node
  - If the split spanned two blocks, it would be unlikely that any HDFS node stored both blocks

For Large Datasets

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
- 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

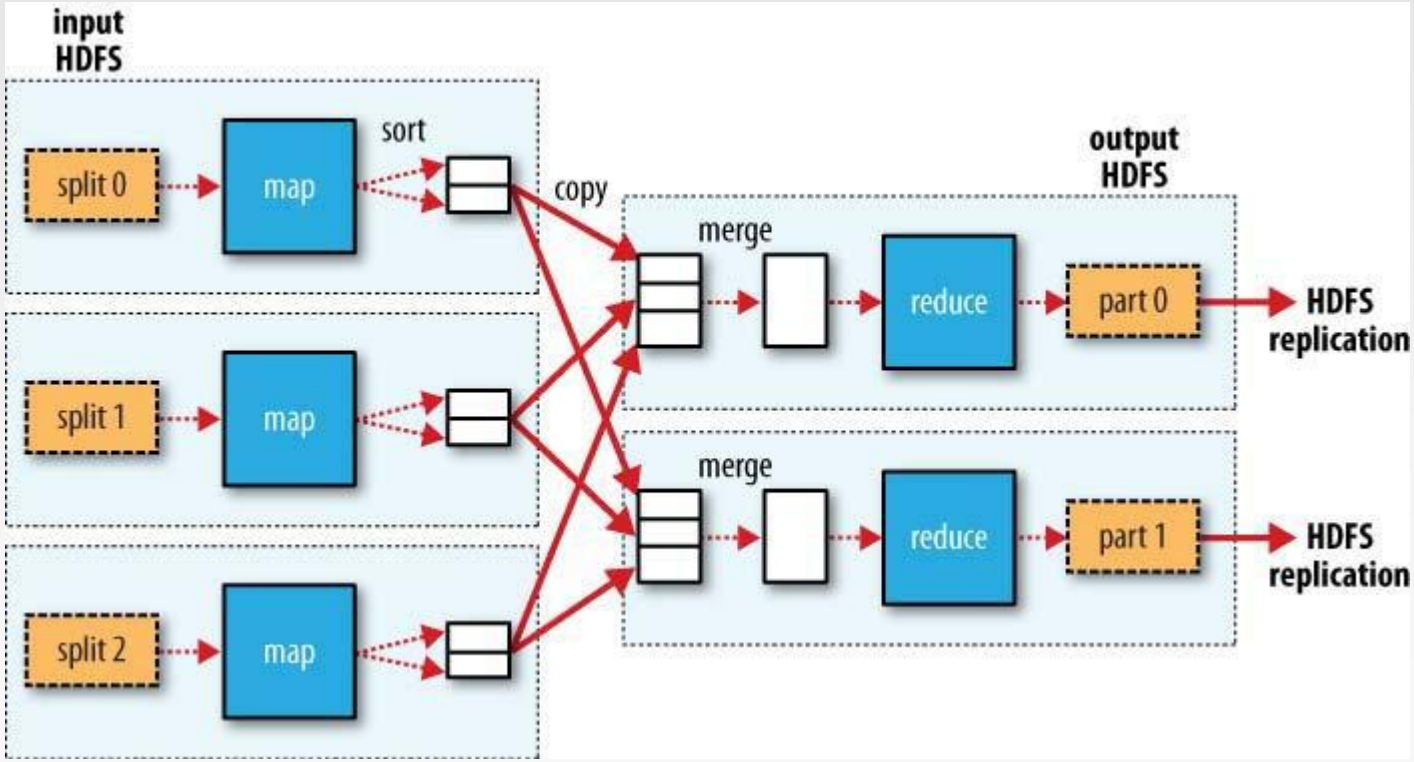
Reduce tasks don’t have the advantage of data locality

- 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

More Detailed MapReduce Dataflow

When there are multiple reducers, the map tasks partition their output:

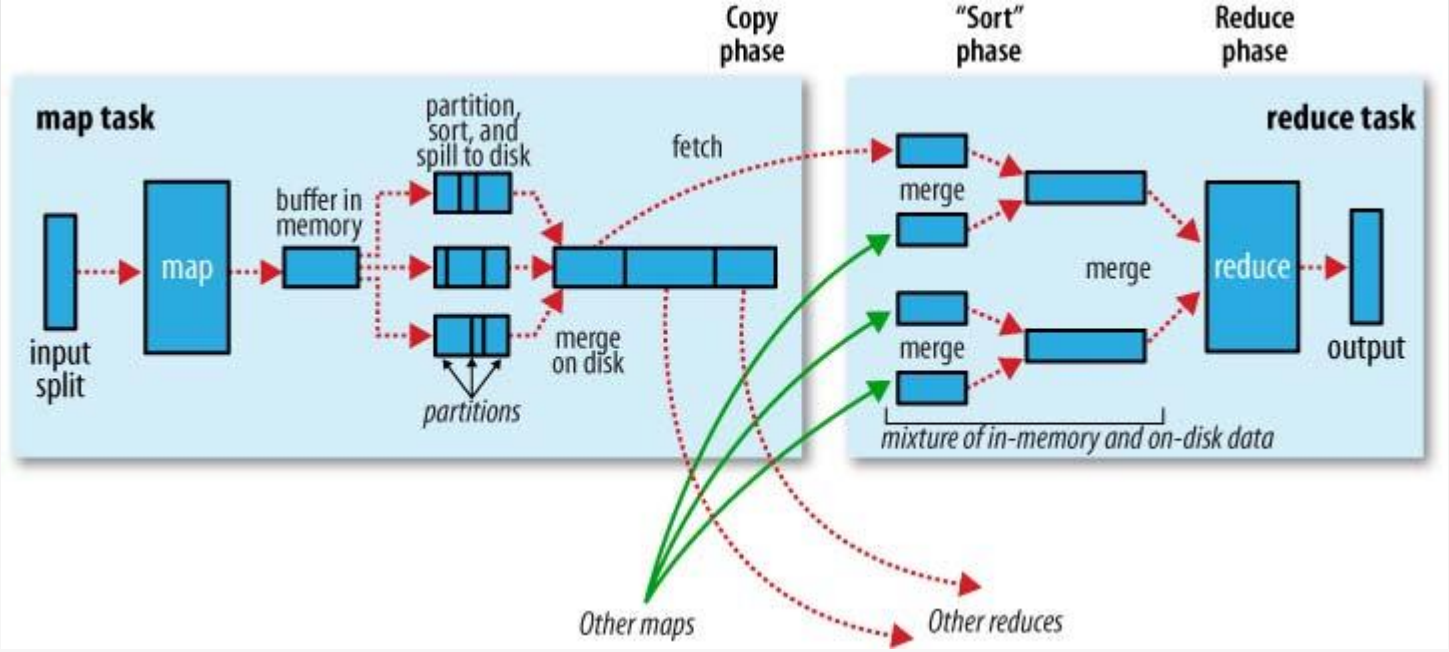
- One partition for each reduce task
- The records for every key are all in a single partition
- Partitioning can be controlled by a user-defined partitioning function



More Detailed MapReduce Dataflow

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



MapReduce Algorithm Design Patterns

Design Pattern 1: In-mapper Combining

Importance of Local Aggregation

Ideal scaling characteristics:

- Twice the data, twice the running time
- Twice the resources, half the running time

Why can't we achieve this?

- Data synchronization requires communication
- Communication kills performance

Thus… avoid communication!

- Reduce intermediate data via local aggregation
- Combiners can help

WordCount Baseline

```
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)
```

What's the impact of combiners?

Word Count: Version 1

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1                ▷ Tally counts for entire document
6:         for all term t ∈ H do
7:             Emit(term t, count H{t})
```

Are combiners still needed?

Word Count: Version 2

```
1: class Mapper
2:     method Initialize
3:         H ← new AssociativeArray
4:     method Map(docid a, doc d)
5:         for all term t ∈ doc d do
6:             H{t} ← H{t} + 1                ▷ Tally counts across documents
7:     method Close
8:         for all term t ∈ H do
9:             Emit(term t, count H{t})
```

Key: preserve state across input key-value pairs!

Design Pattern for Local Aggregation

"In-mapper combining"

- Fold the functionality of the combiner into the mapper by preserving state across multiple map calls

Advantages

- Speed
- Why is this faster than actual combiners?

Disadvantages

- Explicit memory management required
- Potential for order-dependent bugs

Combiner Design

Both input and output data types must be consistent with the output of mapper (or input of reducer)

Combiners and reducers share same method signature

- Sometimes, reducers can serve as combiners
- Often, not…

Hadoop do not guarantee how many times it will call combiner function for a particular map output record

- It is just optimization
- The number of calling (even zero) does not affect the output of Reducers

```
max(0, 20, 10, 25, 15) = max(max(0, 20, 10), max(25, 15)) = max(20, 25) = 25
```

Applicable on problems that are commutative and associative

- Commutative: max(a, b) = max(b, a)
- Associative: max (max(a, b), c) = max(a, max(b, c))

Computing the Mean: Version 1

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, integer r)

1: class Reducer
2:     method Reduce(string t, integers [r_1, r_2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r_1, r_2, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:         r_avg ← sum/cnt
9:         Emit(string t, integer r_avg)
```

Why can't we use reducer as combiner?

Mean(1, 2, 3, 4, 5) != Mean(Mean(1, 2), Mean(3, 4, 5))

Computing the Mean: Version 2

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, integer r)

1: class Combiner
2:     method Combine(string t, integers [r_1, r_2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r_1, r_2, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:         Emit(string t, pair (sum, cnt))          ▷ Separate sum and count

1: class Reducer
2:     method Reduce(string t, pairs [(s_1, c_1), (s_2, c_2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         r_avg ← sum/cnt
9:         Emit(string t, integer r_avg)
```

Why doesn't this work?

Combiners must have the same input and output type, consistent with the input of reducers (output of mappers)

Computing the Mean: Version 3

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, pair (r, 1))

1: class Combiner
2:     method Combine(string t, pairs [(s_1, c_1), (s_2, c_2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         Emit(string t, pair (sum, cnt))

1: class Reducer
2:     method Reduce(string t, pairs [(s_1, c_1), (s_2, c_2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         r_avg ← sum/cnt
9:         Emit(string t, pair (r_avg, cnt))
```

Fixed?

Check the correctness by removing the combiner

Computing the Mean: Version 4

```
1: class Mapper
2:     method Initialize
3:         S ← new AssociativeArray
4:         C ← new AssociativeArray
5:     method Map(string t, integer r)
6:         S{t} ← S{t} + r
7:         C{t} ← C{t} + 1
8:     method Close
9:         for all term t ∈ S do
10:            Emit(term t, pair (S{t}, C{t}))
```

How to Implement In-mapper Combiner in MapReduce?

Lifecycle of Mapper/Reducer

Lifecycle: setup -> map -> cleanup

- setup(): called once at the beginning of the task
- map(): do the map
- cleanup(): called once at the end of the task.
- We do not invoke these functions

In-mapper Combining:

- Use setup() to initialize the state preserving data structure
- Use clearnup() to emit the final key-value pairs

Word Count: Version 2

setup()

```
1: class Mapper
2:     method Initialize
3:         H ← new AssociativeArray
4:     method Map(docid a, doc d)
5:         for all term t ∈ doc d do
6:             H{t} ← H{t} + 1                ▷ Tally counts across documents
7:     method Close
8:         for all term t ∈ H do
9:             Emit(term t, count H{t})
```

cleanup()

References

Chapter 2, Hadoop The Definitive Guide

Chapters 3.3. Data-Intensive Text Processing with MapReduce. Jimmy Lin and Chris Dyer. University of Maryland, College Park.

End of Chapter2