MapReduce: The programming model, motivation and practice

Hao Tu

(Codes and Figures are from Tom White's book and some slides are from Pietro Michiardi's Tutorial)

University of Delaware

http://www.udel.edu

Computer Architecture and Parallel Systems Laboratory
http://www.capsl.udel.edu

This slides

- Will talk about
  - A very brief introduction to MapReduce
  - Motivation: Different solutions form serial to parallel to easier parallel, with a max temperature computation example
  - Running the example code
  - Setting up a development environment for Hadoop under the Eclipse IDE
- Will not talk about
  - Detail introduction to MapReduce/Hadoop, you need to figure out the details by yourself and answer the question sets

What is MapReduce

- A programming model
  - Simple yet not too simple to express useful programs
  - Restricted, yet powerful programming construct
  - Inspired by functional programming
  - Allows expressing distributed computations on massive amounts of data
- An runtime implementation
  - Automatic parallelization
  - Across large-scale clusters of machines
  - Tolerate failures
  - Hide messy internals from users

Brief introduction

- Key-value pairs are the basic data structure in MapReduce, the programmer need:
  - Imposing the key-value structure on arbitrary datasets
  - defines a mapper and a reducer as follows
    - map: (k1, v1) → [(k2, v2)]
    - reduce: (k2, [v2]) → [(k3, v3)]
- A MapReduce job consists in:
  - A dataset stored on the underlying distributed filesystem, which is split in a number of files across machines
  - The mapper is applied to every input key-value pair to generate intermediate key-value pairs
  - The reducer is applied to all values associated with the same intermediate key to generate output key-value pairs

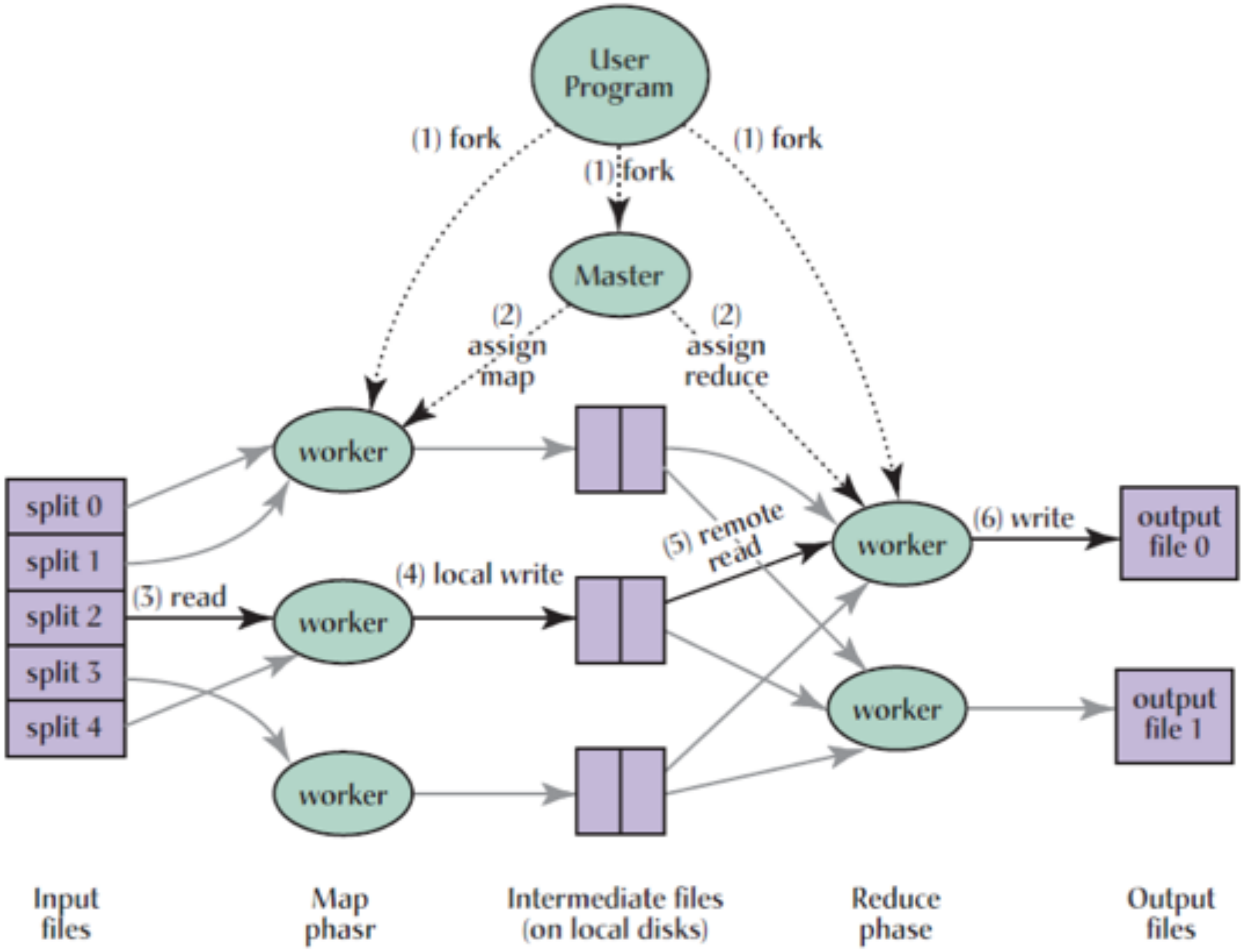


Fig. 1. Execution overview.

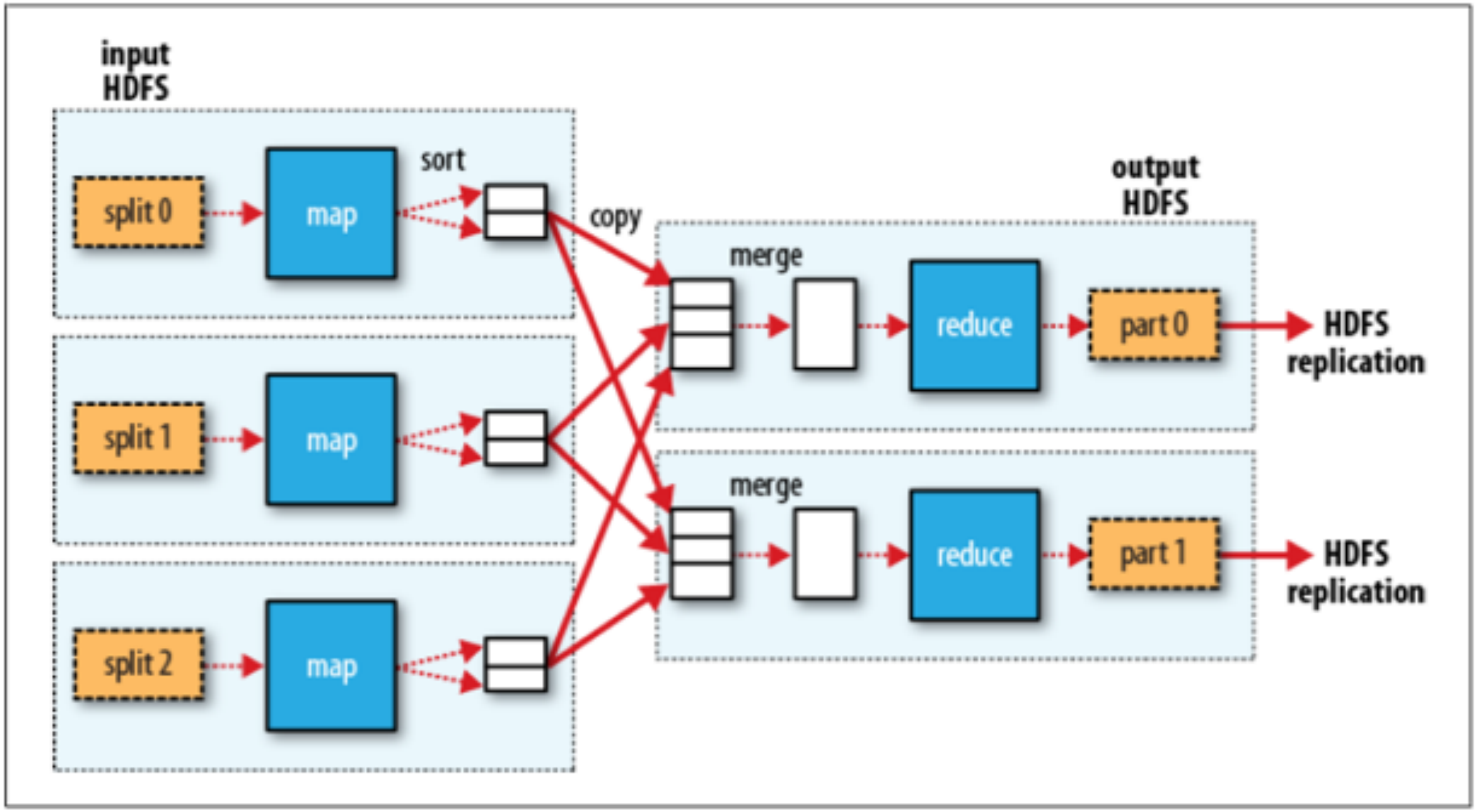


Figure 2-4. MapReduce data flow with multiple reduce tasks

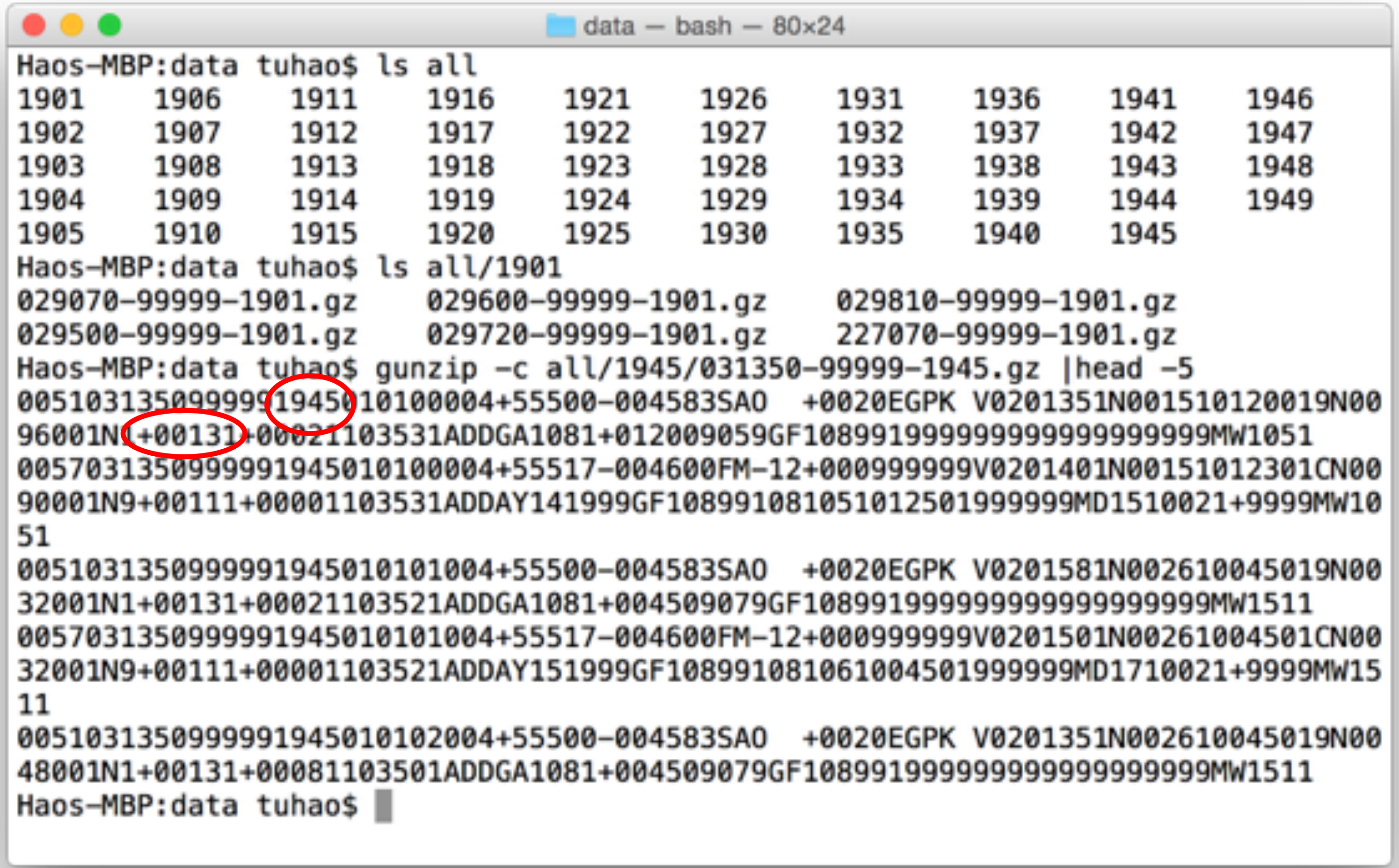
Motivation

- Most computations are conceptually straightforward.
- While parallelization conspire to obscure the original simple computation with large amounts of complex code.
- Can we design an abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault tolerance, data distribution and load balancing in a library?

A Straightforward Example

- Considering we will write a program to mine weather data.
- Objective
  - Find the max temperature for every year
- Dataset
  - Datafiles from National Climatic Data Center are organized by year and weather station
  - The data is stored using a line-oriented format, in which each line is a record.
  - The whole dataset is made up of a large number of relatively small files.

Data example



```
Haos-MBP:data tuhao$ ls all
1901    1906    1911    1916    1921    1926    1931    1936    1941    1946
1902    1907    1912    1917    1922    1927    1932    1937    1942    1947
1903    1908    1913    1918    1923    1928    1933    1938    1943    1948
1904    1909    1914    1919    1924    1929    1934    1939    1944    1949
1905    1910    1915    1920    1925    1930    1935    1940    1945
Haos-MBP:data tuhao$ ls all/1901
029070-99999-1901.gz    029600-99999-1901.gz    029810-99999-1901.gz
029500-99999-1901.gz    029720-99999-1901.gz    227070-99999-1901.gz
Haos-MBP:data tuhao$ gunzip -c all/1945/031350-99999-1945.gz |head -5
0051031350999991945010100004+55500-004583SAO  +0020EGPK V0201351N00151012001 9N00
96001N1+00131+00021103531ADDGA1081+012009059GF108991999999999999999999MW1051
0057031350999991945010100004+55517-004600FM-12+000999999V0201401N00151012301CN00
90001N9+00111+00001103531ADDAY141999GF108991081051012501999999MD1510021+9999MW10
51
0051031350999991945010101004+55500-004583SAO  +0020EGPK V0201581N00261004501 9N00
32001N1+00131+00021103521ADDGA1081+004509079GF108991999999999999999999MW1511
0057031350999991945010101004+55517-004600FM-12+000999999V0201501N00261004501CN00
32001N9+00111+00001103521ADDAY151999GF108991081061004501999999MD1710021+9999MW15
11
0051031350999991945010102004+55500-004583SAO  +0020EGPK V0201351N00261004501 9N00
48001N1+00131+00081103501ADDGA1081+004509079GF108991999999999999999999MW1511
Haos-MBP:data tuhao$ 
```

Solution 1: straightforward

- Pseudo code

```
for each year in DataSet:
    for each line in year:
            temp = fetch_temperature_from_line()
            if temp > max:
                    max = temp
    print year, max
```

Can it be parallelized?

Idea

- Divide and Conquer
  - A feasible approach to tackling big data problems
  - Partition a large problem into smaller sub-problems
  - Independent sub-problems executed in parallel
  - Combine intermediate results from each individual worker
- The workers can be:
  - Threads in a processor core
  - Multiple processors in a machine
  - Many machines in a cluster

Solution 2: parallel without MR

- In theory, it looks straightforward too
  - Process different years in different worker(threads/ processes/machines using OpenMP/pthread/MPI)
- However
  - Dividing the work into equal-size pieces isn't easy or obvious
  - Distributing the work to and combining the results from independent worker need further processing
  - What if a worker fail to finish its job
  - …

Implementation details

- Developer needs to take care of (almost) everything, including Synchronization, Concurrency, Resource allocation …
  - Decompose the original problem in smaller, parallel tasks
  - Schedule tasks on workers distributed in a cluster
    - Data locality
    - Resource availability
  - Ensure workers get the data they need
  - Coordinate synchronization among workers
  - Share partial results
  - Handle failures

Solution 3: parallel with MR

- All we need to do is
  - Code of mappers and reducers
  - Code for combiners and partitioners (optional)
  - Configuration parameters
  - All packaged together
- A MapReduce job is submitted to the cluster, the framework takes care of everything else

MR logical data flow

- Map
  - <line_offsets, line> -> [<year, temperature>]
- Reduce
  - <year, [temperature]> -> [<year, max_temperature>]

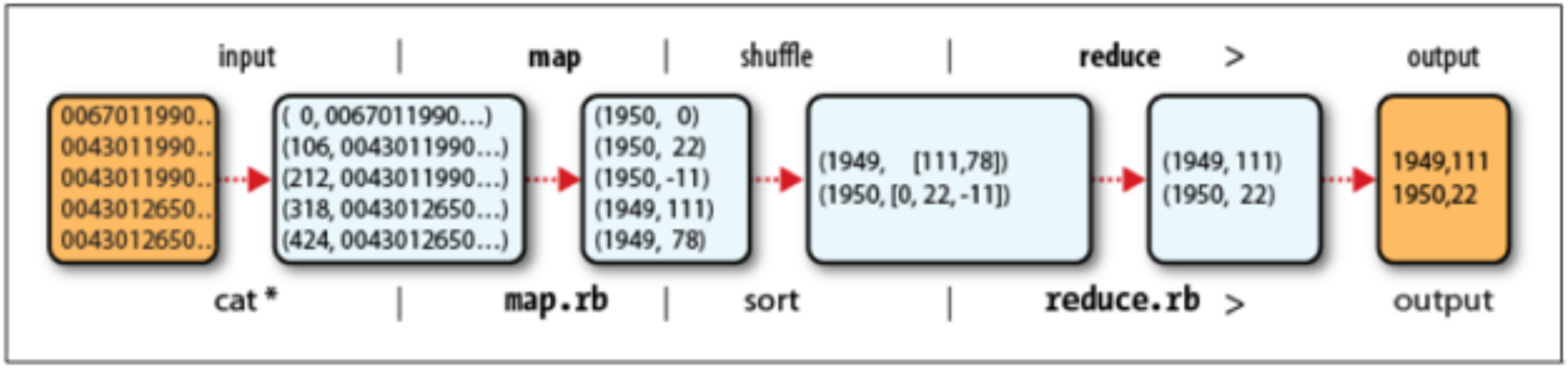


Figure 2-1. MapReduce logical data flow

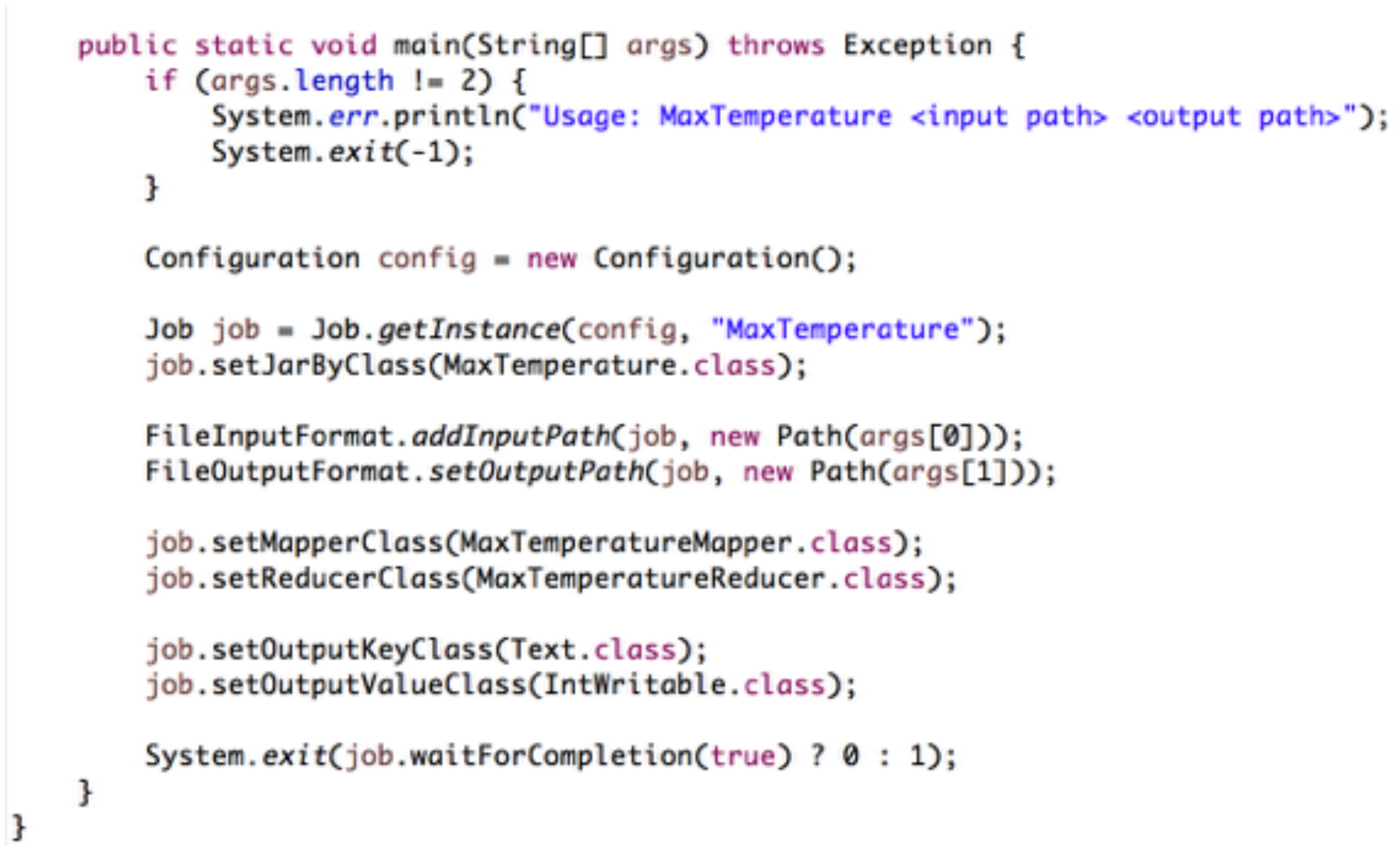
Example - Mapper

```java
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

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) == '+') { // parseInt doesn't like leading plus signs
      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));
    }
  }
}
```

Example - Reducer

```java
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

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));
  }
}
```

Example - Job



```
    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.println("Usage: MaxTemperature <input path> <output path>");
            System.exit(-1);
        }

        Configuration config = new Configuration();

        Job job = Job.getInstance(config, "MaxTemperature");
        job.setJarByClass(MaxTemperature.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(MaxTemperatureMapper.class);
        job.setReducerClass(MaxTemperatureReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Run example code

A Simplified view of MapReduce

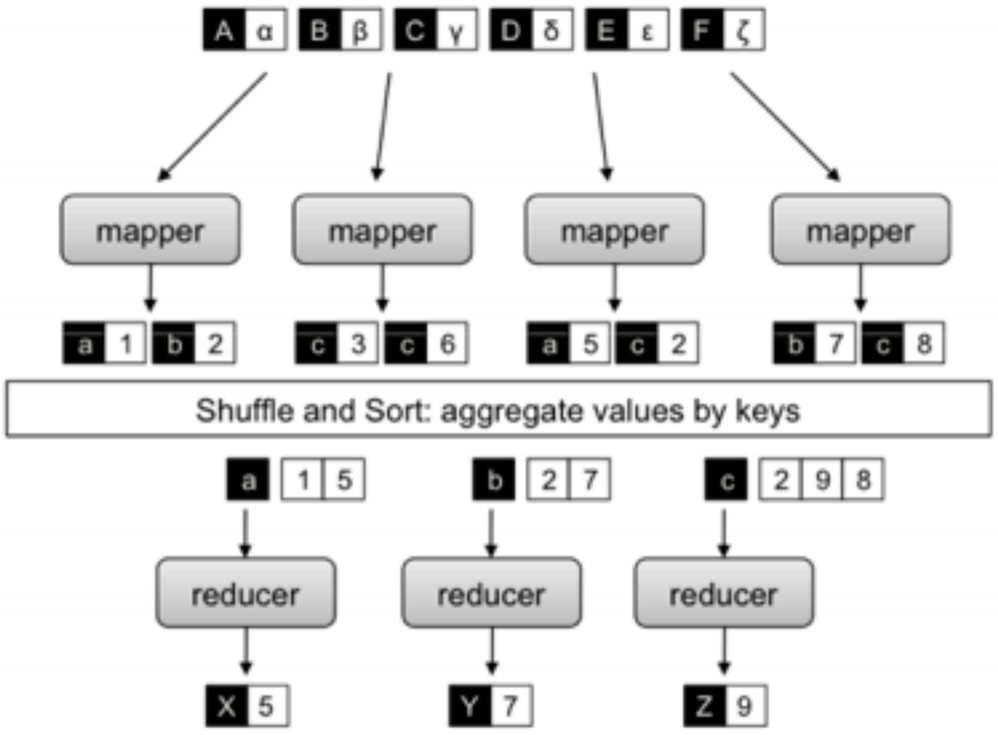


Figure: Mappers are applied to all input key-value pairs, to generate an arbitrary number of intermediate pairs. Reducers are applied to all intermediate values associated with the same intermediate key. Between the map and reduce phase lies a barrier that involves a large distributed sort and group by.

When answer project question 1

- Take into account:
  - Basic concept: such as Key/value, Map, Reduce, Partition, Sort, Shuffle
  - Describe the MapReduce data flow using WordCount as a example
  - Explain How the problems such as Synchronization, data locality and failure are solved in MapReduce/Hadoop

Set up a development environment

- Step 1: Download and install Hadoop and Eclipse

  A. One way is to install Hadoop and Eclipse on your machine.

  - Download Hadoop 2.5.1 http://hadoop.apache.org/releases.html#Download
  - Follow the tutorial[1] to install and test Hadoop
  - Download and Eclipse(Juno) Eclipse IDE for Java Developers on your machine https://www.eclipse.org/downloads/

Set up a development environment

- Step 1: Download and install Hadoop and Eclipse

B. Or alternatively, we can set up a virtual machine with already installed Hadoop and Eclipse.

  - Download and install VirtualBox on your machine: http://virtualbox.org/wiki/ Downloads
  - Download the Cloudera Quickstart VM for virtualbox at http://www.cloudera.com/content/cloudera/en/downloads.html
  - Uncompress the VM archive. It is compressed with 7-Zip. If needed, you can download a tool to uncompress the archive at http://www.7-zip.org/.
  - Start VirtualBox(sudo virtualbox) and click Import Appliance. Click the folder icon beside the location field. Browse to the uncompressed archive folder, select the .ovf file, and click the Open button. Click the Continue button. Click the Import button.

Set up a development environment

- Step 2: Configure Eclipse
  - Create a JAVA project
  - Add External JARS:
    - Install: $HADOOP/share/
      - common/
      - Common/lib
      - Hdfs
      - Mapreduce
      - yarn
    - VM: /usr/lib/Hadoop/client/

Set up a development environment

- Step 3: Run
  - A. Run in Eclipse
    - Run -> Run Configurations
  - B. Export a .jar and run with Hadoop command
    - File -> Export ->Java ->JAR file
    - Bash>hadoop jar wc.jar WordCount /in/data /out/ result

Reference

- [1] Tom White, Hadoop: The Definitive Guide, 3rd Edition. Publisher: O'Reilly Media / Yahoo Press, 2012
- [2] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.
- [3]Pietro Michiardi, Tutorial: MapReduce - Theory and Practice of Data-intensive Applications, http://www.eurecom.fr/~michiard/teaching/slides/clouds/tutorial-mapreduce.pdf
- [4] Jerry Zhao, Jelena Pjesivac-Grbovic, MapReduce - The Programming Model and Practice, http://static.googleusercontent.com/media/research.google.com/en/us/archive/papers/mapreduce-sigmetrics09-tutorial.pdf

Question?