Unit 3

Writing MapReduce Programs: A Weather Dataset, Understanding Hadoop API for MapReduce Framework (Old and New), Basic programs of Hadoop MapReduce: Driver code, Mapper code, Reducer code, RecordReader, Combiner, Partitioner

Introduction:

- MapReduce is a programming model for data processing.
- Hadoop can run MapReduce programs written in various languages such as Java, Ruby, Python, and C++.
- MapReduce programs are inherently parallel, this means we can perform very large-scale data analysis on commodity hardware.
- MapReduce comes into its own for large datasets, one of such dataset is a weather data set, we will write a program that mines weather data

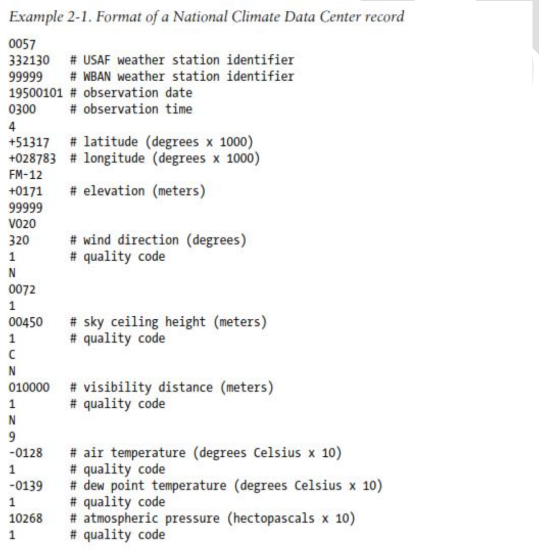
A Weather Dataset:

- Weather sensors collect data every hour at many locations across the globe and gather a large volume of log data.
- This data is very much suitable for analysis with MapReduce because it is **semi- structured** and **record-oriented**.

<u>Data Format</u>

- The data we will use is from the National Climatic Data Centre (NCDC).
- The data is stored using a line-oriented ASCII format, in which each line is a record.

Example 2-1 shows a sample line with some of the salient fields highlighted. The line has been split into multiple lines to show each field; in the real file, fields are packed into one line with no delimiters.



Example 2-1. Format of a National Climate Data Center record

```
0057
332130   # USAF weather station identifier
99999    # WBAN weather station identifier
19500101 # observation date
0300     # observation time
4
+51317   # latitude (degrees x 1000)
+028783  # longitude (degrees x 1000)
FM-12
+0171    # elevation (meters)
99999
V020
320      # wind direction (degrees)
1        # quality code
N
0072
1
00450    # sky ceiling height (meters)
1        # quality code
C
N
010000   # visibility distance (meters)
1        # quality code
N
9
-0128    # air temperature (degrees Celsius x 10)
1        # quality code
-0139    # dew point temperature (degrees Celsius x 10)
1        # quality code
10268    # atmospheric pressure (hectopascals x 10)
1        # quality code
```

- 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.

- For example, here are the first entries for 1990:

```
% 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
```

- What's the highest recorded global temperature for each year in the dataset? We will try to answer this question by using UNIX file processing tools and then using MapReduce to understand the performance baseline.

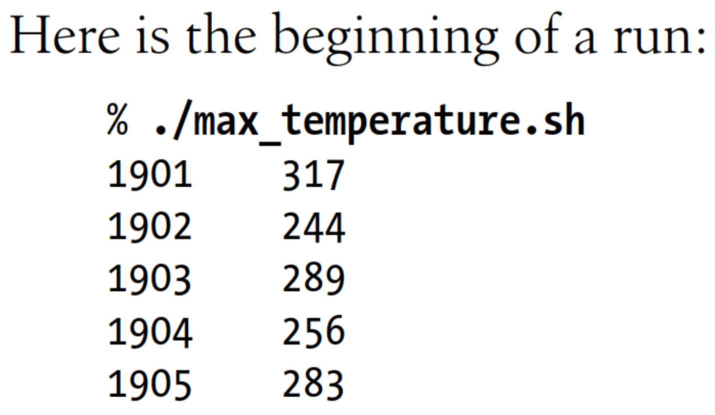
Analyzing the Data with Unix Tools

- The classic tool for processing line-oriented data is *awk*.
- Example 2-2 is a small script to calculate the maximum temperature for each year.

Example 2-2. A program for finding the maximum recorded temperature by year from NCDC weather records

```
#!/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
```

- The script loops through the compressed year files, first printing the year, and then processing each file using *awk*.
- The *awk* script extracts two fields from the data: the air temperature and the quality code.
- The air temperature value is turned into an integer by adding 0. Next, a test is applied to see whether the temperature is valid (the value 9999 signifies a missing value in the NCDC dataset) and whether the quality code indicates that the reading is not suspect or erroneous.
- If the reading is OK, the value is compared with the maximum value seen so far, which is updated if a new maximum is found.
- The END block is executed after all the lines in the file have been processed, and it prints the maximum value.


Here is the beginning of a run:

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

- The temperature values in the source file are scaled by a factor of 10, so this works out as a maximum temperature of 31.7°C for 1901.
- The complete run for the century took 42 minutes in one run on a single EC2 High-CPU Extra Large Instance
- To speed up the processing, we need to run parts of the program in parallel but there are few **problems**.
    1. Dividing the work into equal-size pieces isn't always easy, In this case, the file size for different years varies widely, so some processes will finish much earlier than others. Even if they pick up further work, the whole run is dominated by the longest file. A better approach is to split the file into equal size blocks.
    2. Second, combining the results from independent processes may need further processing. In this case, the result for each year is independent of other years. If using the fixed-size chunk approach, the combination is more delicate. For this example, data for a particular year will typically be split into several chunks, each processed independently. We'll end up with the maximum temperature for each chunk, so the final step is to look for the highest of these maximums for each year.
    3. Third, you are still limited by the processing capacity of a single machine

Analyzing the Data with Hadoop

- To take advantage of the parallel processing that Hadoop provides, we need to express our query as a MapReduce job.

Map and Reduce

- MapReduce works by breaking the processing into two phases: the map phase and the reduce phase.
- Each phase has **key-value** pairs as **input** and **output**, the types of which may be chosen by the programmer.
- The programmer also specifies two functions: the map function and the reduce function.
- The input to our map phase is the raw NCDC data.
- We choose a text input format that gives us each line in the dataset as a text **value**.
- The **key** is the offset of the beginning of the line from the beginning of the file.
- In map function, We pull out the year and the air temperature
- In this case, the map function is just a data preparation phase, setting up the data in such a way that the reducer function can do its work on it: finding the maximum temperature for each year.
- The map function is also a good place to drop bad records: here we filter out temperatures that are missing, suspect, or erroneous.
- To visualize the way the map works, consider the following sample lines of input data

(Some unused columns have been dropped to fit the page, indicated by ellipses):

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

These lines are presented to the map function as the key-value pairs:

```
(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...)
```

The **keys** are the **line offsets** within the file, which we ignore in our map function. The map function merely extracts the year and the air temperature (indicated in bold text), and emits them as its output (the temperature values have been interpreted as integers):

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

- The output from the map function is processed by the MapReduce framework before being sent to the reduce function. This processing sorts and groups the key-value pairs by key.
- So, in our example, our reduce function sees the following input:

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

- Each year appears with a list of all its air temperature readings. All the reduce function has to do now is iterate through the list and pick up the maximum reading:

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

- This is the final output: the maximum global temperature recorded in each year.
- The whole data flow is illustrated in Figure 2-1. At the bottom of the diagram is a Unix pipeline, which mimics the whole MapReduce flow .

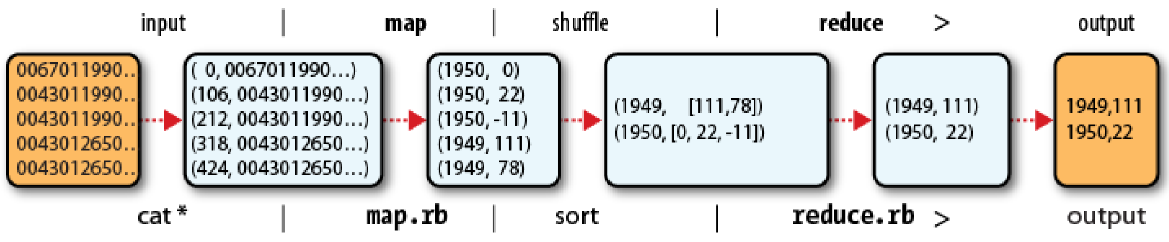


Figure 2-1. MapReduce logical data flow

Java MapReduce

- In order to implement MapReduce in Java we need three things: **a map function, a reduce function**, and some code to run the job **(Driver)**
- The map function is represented by the `Mapper` class, which declares an abstract `map ()` method.
- Example 2-3 shows the implementation of our map method.

Example 2-3. Mapper for the maximum temperature example

```
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));
}
}
}
```

- The `Mapper` class is a generic type, with four formal type parameters that specify the input key, input value, output key, and output value types of the map function.
- For the present example, the input key is a long integer offset, the input value is a line of text, the output key is a year, and the output value is an air temperature (an integer).
- Hadoop provides its own set of basic types that are opti- mized for network serialization. These are found in the `org.apache.hadoop.io` package.
- Here we use `LongWritable`, which corresponds to a Java `Long`, `Text` (like Java `String`), and `IntWritable` (like Java `Integer`).
- The `map()` method is passed a key and a value. We convert the `Text` value containing the line of **input** into a Java `String`, then use its `substring ()` method to extract the columns we are interested in.
- The `map()` method also provides an instance of `Context` to write the output to. In this case, we write the year as a

Text object (since we are just using it as a key), and the temperature is wrapped in an IntWritable.

- We write an output record only if the temperature is present and the quality code indicates the temperature reading is OK.

The reduce function is similarly defined using a Reducer, as illustrated in Example 2-4.

Example 2-4. Reducer for the maximum temperature example

```
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));
         }
}
```

- For the reduce function, four formal type parameters are used to specify the input and output types.
- The input types of the reduce function must match the output types of the map function: Text and IntWritable
- The output types of the reduce function are Text and IntWritable, for a year and its maximum temperature, which we find by iterating through the temperatures and comparing each with a record of the highest found so far.

The third piece of code runs the MapReduce job (see Example 2-5).

Example 2-5. Application to find the maximum temperature in the weather dataset

```
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.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;


public class MaxTemperature {


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

```
            }

    Job job = new Job();
    job.setJarByClass(MaxTemperature.class);
    job.setJobName("Max temperature");

    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);
  }
}
```

- A **Job** object forms the specification of the job and gives you control over how the job is run.
- When we run this job on a Hadoop cluster, we will package the code into a JAR file (which Hadoop will distribute around the cluster). Rather than explicitly specify the name of the JAR file, we can pass a class in the Job's `setJarByClass()` method, which Hadoop will use to locate the relevant JAR file by looking for the JAR file containing this class.
- Next, we specify the input and output paths. An input path is specified by calling the static `addInputPath()` method on `FileInputFormat`, and it can be a single file, a directory (in which case the input forms all the files in that directory)
- The output path (of which there is only one) is specified by the static `setOutput Path()` method on `FileOutputFormat`. It specifies a directory where the output files from the reducer functions are written.
- The directory shouldn't exist before running the job because Hadoop will complain and not run the job. This precaution is to prevent data loss.
- Next, we specify the map and reduce types to use via the `setMapperClass()` and `setReducerClass()` methods.
- The `setOutputKeyClass()` and `setOutputValueClass()` methods control the output types for the map and the reduce functions which are often the same.
- If they are different, the map output types can be set using the methods `setMapOutputKeyClass()` and `setMapOutputValueClass()`.
- After setting the classes that define the map and reduce functions, The `waitForCompletion()` method on `Job` submits the job and waits for it to finish
- The method's Boolean argument is a verbose flag, so in this case the job writes information about its progress to the console.
- The return value of the `waitForCompletion()` method is a Boolean indicating success (`true`) or failure (`false`), which we translate into the program's exit code of `0` or `1`.

A test run

Let's test it on the weather data

```
% export HADOOP_CLASSPATH=hadoop-examples.jar
% hadoop MaxTemperature input/ncdc/sample.txt output
```

- The output was written to the *output* directory, which contains one output file per reducer. The job had a single reducer, so we find a single file, named *part-r-00000*:

```
% cat output/part-r-00000
1949    111
1950    22
```

- We interpret this as saying that the maximum temperature recorded in 1949 was 11.1°C, and in 1950 it was 2.2°C

The old and the new Java MapReduce APIs

- The Java MapReduce API used in the previous example was first released in Hadoop 0.20.0. This new API, sometimes referred to as **"Context Objects**," was designed to make the API easier to evolve in the future.
- The new API is largely complete in the latest 1.x release series.

There are several notable differences between the two APIs:

- The new API favours abstract classes over interfaces, since these are easier to evolve. This means that we can add a method (with a default implementation) to an abstract class without breaking old implementations of the class. For example, the `Mapper` and `Reducer` interfaces in the old API are abstract classes in the new API.
- The new API is in the `org.apache.hadoop.mapreduce` package (and subpackages).The old API can still be found in `org.apache.hadoop.mapred`.
- The new API makes extensive use of context objects that allow the user code to communicate with the MapReduce system. The new `Context`, for example, essentially unifies the role of the `JobConf`, the `OutputCollector`, and the `Reporter` from the old API.
- Job control is performed through the `Job` class in the new API, rather than the old `JobClient`, which no longer exists in the new API.
- Configuration has been unified. The old API has a special `JobConf` object for job configuration, which is an extension of Hadoop's vanilla `Configuration` object. In the new API, job configuration is done through a `Configuration`, possibly via some of the helper methods on `Job`.
- Output files are named slightly differently: in the old API both map and reduce outputs are named part-nnnnn, whereas in the new API map outputs are named part-m-nnnnn, and reduce outputs are named part-r-nnnnn (where nnnnn is an

integer designating the part number, starting from zero).

- User-overridable methods in the new API are declared to throw `java.lang.Inter ruptedException`. This means that you can write your code to be responsive to interrupts so that the framework can gracefully cancel long-running operations if it needs to.
- In the new API, the reduce () method passes values as a java.lang.Iterable, rather than a java.lang.Iterator (as the old API does). This change makes it easier to iterate over the values using Java's for-each loop construct:

  for (VALUEIN value : values) { ... }

Example 2-6 shows the `MaxTemperature` application rewritten to use the old API. The differences are highlighted in bold

Example 2-6. Application to find the maximum temperature, using the old MapReduce API

```
public class OldMaxTemperature {

  static class OldMaxTemperatureMapper extends MapReduceBase implements Mapper<LongWritable, Text,
    Text, IntWritable> {

    private static final int MISSING = 9999;

    @Override
    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter
        reporter) throws IOException {

      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]")) {
        output.collect(new Text(year), new IntWritable(airTemperature));
      }
    }
  }
  static class OldMaxTemperatureReducer extends MapReduceBase implements Reducer<Text,
    IntWritable, Text, IntWritable> {

    @Override
    public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output,
        Reporter reporter) throws IOException {

      int maxValue = Integer.MIN_VALUE;
      while (values.hasNext()) {
        maxValue = Math.max(maxValue, values.next().get());
      }
     output.collect(key, new IntWritable(maxValue));


  }
  }

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

    JobConf conf = new JobConf(OldMaxTemperature.class);
    conf.setJobName("Max temperature");

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

    conf.setMapperClass(OldMaxTemperatureMapper.class);
    conf.setReducerClass(OldMaxTemperatureReducer.class);

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

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

Combiner Functions

- MapReduce jobs are limited by the bandwidth available on the cluster, so to minimize the data transferred between map and reduce tasks, Hadoop allows the user to specify a *combiner function* to be run on the map output.
- The combiner function's output forms the input to the reduce function.
- The combiner function is an optimization calling the combiner function zero, one, or many times should produce the same output from the reducer.
- Suppose that for the maximum temperature example, readings for the year 1950 were processed by two maps (because they were in different splits). Imagine the first map produced the output:

```
(1950, 0)
(1950, 20)
 (1950, 10)
```

  And the second produced:

```
(1950, 25)
 (1950, 15)
```

- The reduce function would be called with a list of all the values:

```
(1950, [0, 20, 10, 25, 15])
```

  With output:

```
(1950, 25)
```

- Since 25 is the maximum value in the list. We could use a combiner function that, just like the reduce function, finds the maximum temperature for each map output.
- The reduce would then be called with:

```
(1950, [20, 25])
```

- we may express the function calls on the temperature values in this case as follows:

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

Specifying a combiner function

- In Java MapReduce program, the combiner function is defined using the `Reducer` class, and for this application, it is the same implementation as the reducer function in `MaxTemperatureReducer`.
- The only change we need to make is to set the combiner class on the `Job` (see Example 2-7).

Example 2-7. Application to find the maximum temperature, using a combiner function for efficiency

```
public class MaxTemperatureWithCombiner {

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

    Job job = new Job();
    job.setJarByClass(MaxTemperatureWithCombiner.class);
    job.setJobName("Max temperature");

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

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setCombinerClass(MaxTemperatureReducer.class);
    job.setReducerClass(MaxTemperatureReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

RecordReader

- RecordReader is an interface declared in org.apache.hadoop.mapred.
- RecordReader reads <key, value> pairs from an InputSplit.
- RecordReader, typically, converts the byte-oriented view of the input, provided by the InputSplit, and presents a record-oriented view for the Mapper & Reducer tasks for processing. It thus assumes the responsibility of processing record boundaries and presenting the tasks with keys and values

Method Summary

Methods

Modifier and Type	**Method and Description**
void	**close**() Close this **InputSplit** to future operations.
K	**createKey**() Create an object of the appropriate type to be used as a key.
V	**createValue**() Create an object of the appropriate type to be used as a value.
long	**getPos**() Returns the current position in the input.
float	**getProgress**() How much of the input has the **RecordReader** consumed i.e.
boolean	**next**(**K** key, **V** value) Reads the next key/value pair from the input for processing.

Partitioner

- Class Partitioner<KEY,VALUE> is defined in org.apache.hadoop.mapreduce

public abstract class Partitioner<KEY,VALUE>extends Object

- Partitions the key space.
- 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.
- Hence this controls which of the m reduce tasks the intermediate key (and hence the record) is sent for reduction.

Note: If you require your Partitioner class to obtain the Job's configuration object, implement the Configurable interface.

Method Summary

Methods

Modifier and Type	**Method and Description**
abstract int	**getPartition(KEY** key, **VALUE** value, int numPartitions) Get the partition number for a given key (hence record) given the total number of partitions i.e.

- **Methods inherited from class java.lang.Object**

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait,wait, wait