

## \* Weather Dataset :→

Weather Sensors collecting data every hour at many locations across the globe gather a large volume of log data, which is a good candidate for analysis with Map-Reduce, since it is semi-structured and record-oriented.

## Data format :→

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

## MapReduce :→

MapReduce is a Software Framework and programming model used for processing huge amounts of data. MapReduce program work in two phases, namely, Map and Reduce.

Map tasks deal with splitting and mapping of data while Reduce tasks shuffle and reduce the data.

Hadoop is capable of running MapReduce program written in various languages: Java, Ruby, Python and C++. The programs of MapReduce in cloud computing are parallel in nature, this are very useful for performing large-scale data analysis using multiple machines in the cluster.

The input to each phase is key-value pairs. In addition,

Every programmer needs to specify two functions: Map function and Reduce function.



## ⇒ Classic MapReduce (MapReduce 1) →

A job run in classic MapReduce is illustrated in figure-1. At the highest level, there are four independent entities:

- The client, which submits the MapReduce job.
- The Jobtracker, which coordinates the job run. The Jobtracker is a Java application whose main class is JobTracker.
- The tasktrackers, which run the tasks that the job has been split into. Tasktracker are Java application whose main class is TaskTracker.
- The distributed filesystem (Normally HDFS) ~~covered~~, which is used for sharing job files between the other entities.

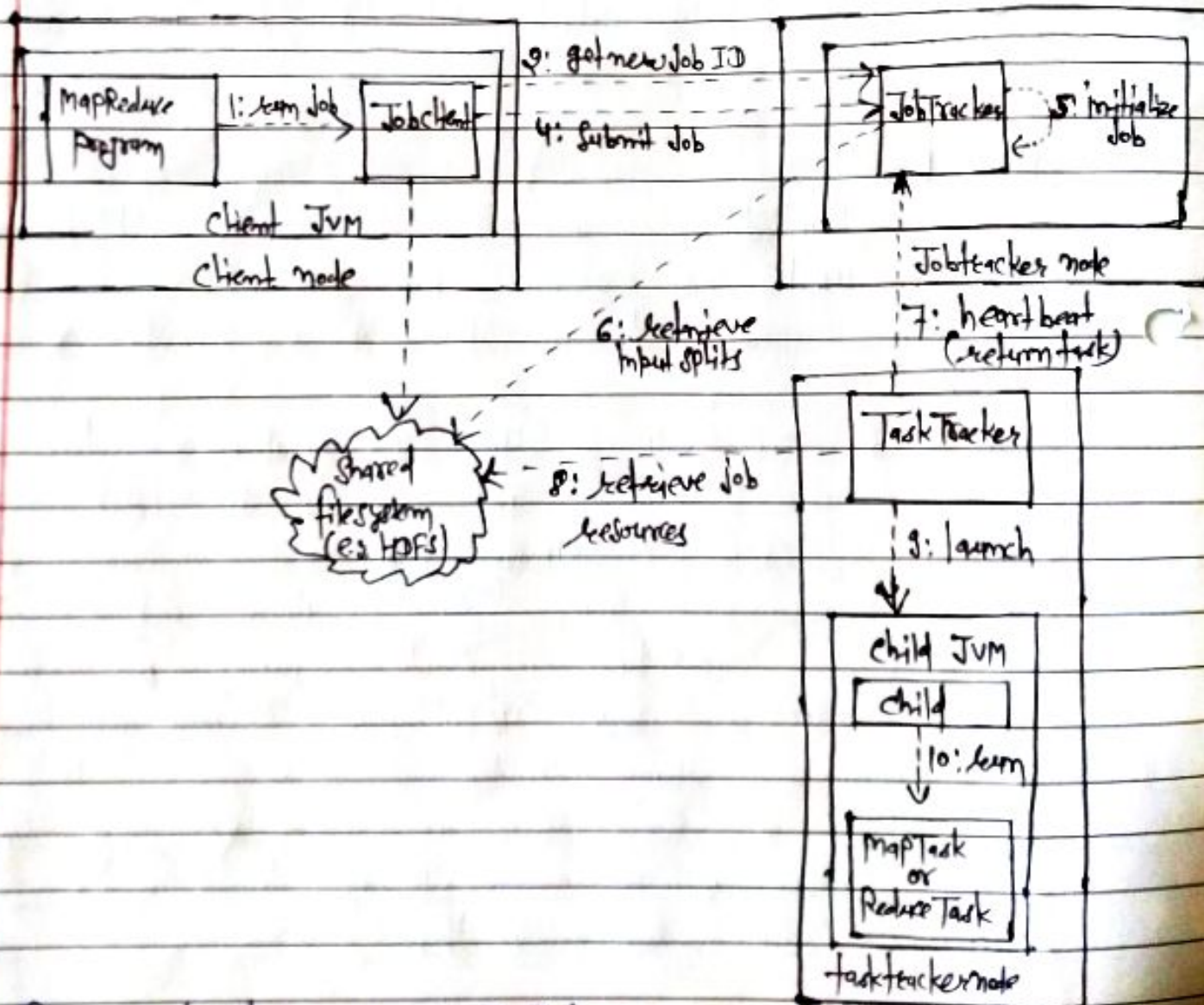


Figure: 1 → How Hadoop runs a MapReduce job using the classic framework



## ⇒ Job Submission :-

The `submit()` method on `Job` creates an internal `JobSubmitter` instance and calls `submitJobInternal()` on it (Step 1). Having submitted the job, `waitForCompletion()` polls the job's progress once a second and reports the progress to the console if it has changed since the last report.

The job submission process implemented by `JobSubmitter` does the following:

- Asks the jobtracker for a new job ID (by calling `getNewJobId` on `JobTracker`) (step 2)
- Checks the output specification of the job.  
for example, if the output directory has not been specified or it already exists, the job is not submitted and an error is thrown to the MapReduce program.
- Computes the input splits for the job. If the splits cannot be computed, because the input paths don't exist, for example, then the job is not submitted and an error is thrown to the MapReduce program.
- Copies the resources needed to run the job, including the job JAR file, the configuration file, and the computed input splits, to the jobtracker's filesystem in a directory named after the job ID. The job JAR is copied with a high replication factor (controlled by the `mapred.submit.replication` property, which defaults to 10) so that there are lots of copies across the cluster for the tasktrackers to access when they run tasks for the job. (step 3)
- Tells the jobtracker that the job is ready for execution (by calling `submitJob()` on `JobTracker`) (step 4)



### ⇒ Job Initialization :-

When the JobTracker receives a call to its `submit` method, it puts it into an internal queue from where the Job Schedulers will pick it up and initialize it. (Step 5)

To create the list of tasks to run, the Job Scheduler first retrieves the input Splits computed by the client from the shared filesystem (Step-6). It then creates one map task for each split. The number of reduce tasks to create is determined by the mapped reduce tasks property in the job, which is set by the `setNumReduceTasks()` method, and the scheduler simply creates this number of reduce tasks to be run.

### ⇒ Task Assignment :-

Tasktrackers run a simple loop that periodically sends heartbeat method calls to the JobTracker. Heartbeats tell the JobTracker that a tasktracker is alive. As a part of the heartbeat, a tasktracker will indicate whether it is ready to run a new task, and if it is, the JobTracker will allocate it a task, which it communicates to the tasktracker using the heartbeat return value (Step 7).

Tasktrackers have a fixed number of slots for map tasks and for reduce tasks. For example, a tasktracker may be able to run two map tasks and two reduce tasks simultaneously. (The precise number depends on the number of cores and the amount of memory on the tasktracker.)

Task Execution now that the tasktracker has been assigned a task, the next step is for it to run the task. First, it localizes the Job JAR by copying it from the shared filesystem to the tasktracker's filesystem.



It also copies any files needed from the distributed cache by the application to the local disk. Then, it creates a local working directory for the task, and un-jars the contents of the JAR into this directory. Third it creates an instance of TaskRunner to run the task. Task Runner launches a new Java Virtual Machine (step 9) to run each task in (step 10), so that any bugs in the user-defined map and reduce function don't affect the tasktracker.

### ⇒ Progress and Status Updates :->

MapReduce jobs are long-running batch jobs, taking anything from minutes to hours to run. Because this is a significant length of time, it's important for the user to get feedback on how the job is progressing. A job and each of its tasks have a status, which includes such things as the state of the job or task (e.g. running, successfully completed, failed).

### ⇒ Job Completion :->

When the jobtracker receives a notification that the last task for a job is complete (this will be the special job cleanup task), it changes the status for the job to "successful". Then, when the job polls for status, it learns that the job has completed successfully, so it prints a message to tell the user and then returns from the waitForCompletion() method.



## \* Understanding Hadoop API for MapReduce Framework (old and new) →

Hadoop provides two Java MapReduce APIs named as old and new respectively.

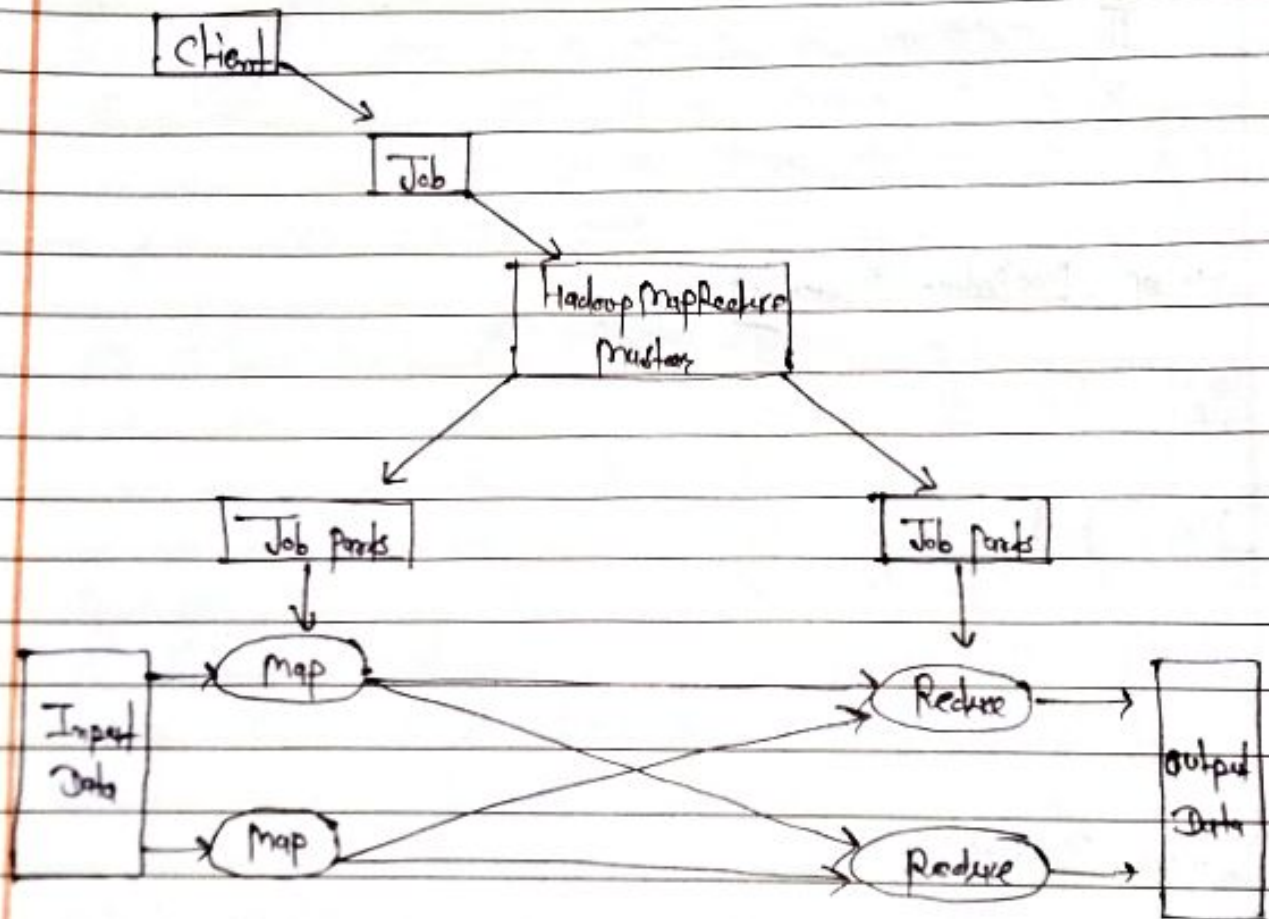
There are several notable ~~different~~ differences between the two APIs.

1. The new API favors abstract classes over interfaces, since these are easier to evolve. For example, you 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` interface in the old API are abstract classes in the new API.
2. 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`.
3. The new API makes extensive use of `Context`, for example, essentially unifies the role of the `JobConf`, the `OutputCollector` and the `Reporter` from the old API.
4. In both APIs, key-value record pairs are pushed to the `mapper` and `reducer`, but in addition, the new API allows both `mappers` and `reducers` to control the execution flow by overriding the `run()` method. In the old API this is possible for `mappers` by writing a `mapRunnable`, but no equivalent exists for `reducers`.
5. Configuration has been unified. The old API has a special `JobConf` object for job configuration. In the new API, this distinction is dropped, so job configuration is done



## \* MapReduce Architecture :-

MapReduce is a programming model used for efficient processing in parallel over large data-sets in a distributed manner. The data is first split and then combined to produce the final result. The libraries for MapReduce is written in so many programming language with various different - different optimizations.



## \* Components of MapReduce Architecture :->

### 1. Client :->

The mapReduce client is the one who brings the job to the mapReduce for processing. There can be multiple clients available that continuously send jobs for processing to the Hadoop MapReduce Manager.

### 2. Job :->

The mapReduce job is the actual work that the client wanted to do which is comprised of so many smaller tasks that the client wants to process or execute.

### 3. Hadoop MapReduce Master :->

It divides the particular job into subsequent Job - parts.

### 4. Job - parts :->

The task or sub-jobs that are obtained after dividing the main job. The result of all the Job - parts combined to produce the final output.

### 5. Input Data :->

The data set that is fed to the MapReduce for processing.

### 6. Output Data :->

The final result is obtained after the processing.



through Configuration.

6. 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.

## \* Basic Programmes of Hadoop MapReduce :->

### -> Driver Code :->

A Job object forms the specification of the job. It 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 specifying 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.

Having constructed a job object, 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), or a file pattern.

The output path (of which there is only one) is specified by the static `setOutputPath()` method on `FileOutputFormat`. It specifies a directory where the output files from the reducer functions are written.



The driver code for weather program is specified below:

```
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 ExpException
    {
        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.out.println(job.waitForCompletion(true) ? 0 : 1);
    }
}
```



## \* Mapper Code →

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). The following example shows the implementation of our map method.

```
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 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(28, 32));
        }
    }
}
```



~~Else~~

Else

{

airTemperature = Integer.parseInt(line.substring(87, 92));

}

String quality = line.substring(92, 93);

if (airTemperature != MISSING && quality.matches("[0|450]"))

{

Context.write(new Text(year), new IntWritable(airTemperature));

}

}

}

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.



\* Reducer Code →

Again, four formal type parameters are used to specify the input and output types, this time for the reduce function. The input types of the reduce function must match the output types of the map function: Text and IntWritable. And in this case, 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.

```

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));
    }
}

```



## \* RecordReader :->

RecordReader is responsible for creating key/value pairs which has been fed to map task to process. Each InputFormat has to provide its own RecordReader implementation to generate key/value pairs. For example, the default TextInputFormat provides LineRecordReader which generates byte offset of the file as key and  $n$  separated line in the input file as value.

## \* Combiner Code :->

Many MapReduce jobs are limited by the bandwidth available on the cluster, so it pays 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. Since the Combiner function is an optimization Hadoop does not provide a guarantee of how many times it will call it for a particular map output record, if at all. In other words, calling the Combiner function zero, one, or many times should produce the same output from the reducer. The Combiner function doesn't replace the reduce function. But it can help cut down the amount of data shuffled between the maps and reducers.

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

```

### \* Partitioner Code :->

The partitioning phase takes place after the map phase and before the reduce phase. The number of partitions is equal to the number of reducers. The data gets partitioned across the reducers according to the partitioning function. The difference between a partitioner and a combiner is that the partitioner divides the data according to the number of reducers so that all the data in a single partition gets ~~executed~~ executed by a single reducer. However, the combiner functions similar to the reducer and processes



the data in each partition. The combiner is an optimization to the reducer. The default partitioning function is the hash partitioning function where the hashing is done on the key. However it might be useful to partition the data according to some other function of the key or the value.