2 WRITING MAPREDUCE PROGRAMS

IMPORTANT QUESTIONS

PART-A

Q.1 ***What do you mean by map task?***

Ans. Map Task: This is the first task, which takes input data and converts it into a set of data, where individual elements are broken down into tuples (key/value pairs).

Q.2 ***What is reduce task? Explain.***

Ans. Reduce Task: This task takes the output from a map task as input and combines those data tuples into a smaller set of tuples. The reduce task is always performed after the map task.

Q.3 ***What main configuration parameters are specified in MapReduce?***

Ans. The MapReduce programmers need to specify following configuration parameters to perform the map and reduce jobs:

(i) The input location of the job in HDFS.

(ii) The output location of the job in HDFS.

(iii) The input's and output's format.

(iv) The classes containing map and reduce functions, respectively.

(v) The .jar file for mapper, reducer and driver classes.

Q.4 ***Explain JobConf in MapReduce.***

Ans. It is a primary interface to define a map-reduce job in the Hadoop for job execution. JobConf specifies mapper, combiner, partitioner, Reducer,InputFormat, OutputFormat implementations and other advanced job facts like comparators.

These are described in Mapreduces online reference guide and on Mapreduce community.

Q.5 ***What is a MapReduce combiner?***

Ans. MapReduce is also known as semi-reducer, combiner is an optional class to combine the map out records using the same key. The main function of a combiner is to accept inputs from Map Class and pass those key-value pairs to Reducer class.

Q.6 ***What is RecordReader in a MapReduce?***

Ans. RecordReader is used to read key/value pairs form the InputSplit by converting the byte-oriented view and presenting record-oriented view to Mapper.

Q.7 ***Define Writable data types in MapReduce.***

Ans. Hadoop reads and writes data in a serialized form in writable interface. The Writable interface has several classes like Text (storing String data), IntWritable, LongWritable, FloatWritable, BooleanWritable users are free to define their personal Writable classes as well.

Q.8 ***How Hadoop MapReduce works?***

Ans. In MapReduce, during the map phase, it counts the words in each document, while in the reduce phase it

aggregates the data as per the document spanning the entire collection. During the map phase, the input data is divided into splits for analysis by map tasks running in parallel across Hadoop framework.

Q.9 ***Explain what is shuffling in MapReduce?***

Ans. The process by which the system performs the sort and transfers the map outputs to the reducer as inputs is known as the shuffle.

Q.10 ***Explain what is distributed cache in MapReduce framework?***

Ans. Distributed cache is an important feature provided by the MapReduce framework. When you want to share some files across all nodes in Hadoop Cluster, distributed cache is used. The files could be an executable jar files or simple properties file.

Q.11 ***How is identity mapper different from chain mapper?***

Ans.

Identity Mapper	Chain Mapper
This is the default mapper that is chosen when no mapper is specified in the MapReduce driver class.	This class is used to run multiple mappers in a single map task.
It implements its identity function, which directly writes all its key-value pairs into output.	The output of the first mapper becomes the input to the second mapper, second to third and so on.
It is defined in old MapReduce API (MR1) in: org.apache.Hadoop.mapred.lib.package	It is defined in : org.apache.Hadoop.mapreduce.lib.chain.ChainMapperpackage

Q.12 ***Explain the core methods of a Reducer.***

Ans. There are three core methods of a reducer. They are :

setup( ) : This is used to configure different parameters like heap size, distributed cache and input data.

reduce( ) : A parameter that is called once per key with the concerned reduce task.

cleanup( ) : Clears all temporary files and called only at the end of a reducer task.

Q.13 ***What role do RecordReader, Combiner and Partitioner play in a MapReduce operation?***

Ans. RecordReader : This communicates with the InputSplit and converts the data into key-value pairs suitable for the mapper to read.

Combiner : This is an optional phase; it is like a mini reducer. The combiner receives data from the map tasks, works on it, and then passes its output to the reducer phase.

Partitioner : The partitioner decides how many reduced tasks would be used to summarize the data. It also confirms how outputs from combiners are sent to the reducer, and controls the partitioning of keys of the intermediate map outputs.

PART-B

Q.14 ***Explain MapReduce program with neat figure.***

Ans. Hadoop MapReduce is a software framework for easily writing applications which process big amounts of data in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.

The term MapReduce actually refers to the following two different tasks that Hadoop programs perform:

- **The Map Task :** *Refer to Q.1.*
- **The Reduce Task :** *Refer to Q.2.*

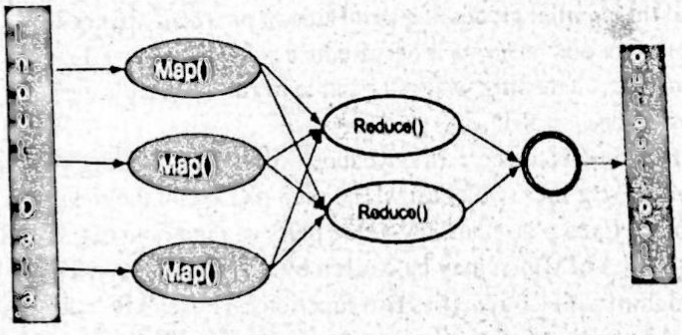

Fig.

Typically both the input and the output are stored in a file-system. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks. The MapReduce framework consists of a single master JobTracker and one slave TaskTracker per cluster-node. The master is responsible for resource management, tracking

resource consumption/availability and scheduling the jobs component tasks on the slaves, monitoring them and re-executing the failed tasks. The slaves TaskTracker execute the tasks as directed by the master and provide task-status information to the master periodically. The JobTracker is a single point of failure for the Hadoop MapReduce service which means if JobTracker goes down, all running jobs are halted.

The major advantage of MapReduce is that it is easy to scale data processing over multiple computing nodes. Under the MapReduce model, the data processing primitives are called mappers and reducers. Decomposing a data processing application into mappers and reducers is sometimes nontrivial. But, once we write an application in the MapReduce form, scaling the application to run over hundreds, thousands, or even tens of thousands of machines in a cluster is merely a configuration change. This simple scalability is what has attracted many programmers to use the MapReduce model.

During a MapReduce job, Hadoop sends the Map and Reduce tasks to the appropriate servers in the cluster. The framework manages all the details of data-passing such as issuing tasks, verifying task completion, and copying data around the cluster between the nodes.

Most of the computing takes place on nodes with data on local disks that reduces the network traffic. After completion of the given tasks, the cluster collects and reduces the data to form an appropriate result, and sends it back to the Hadoop server.

Q.15 *Explain the process of analyzing the data with Hadoop.*

Ans. 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. After some local, small-scale testing, we will be able to run it on a cluster of machines.

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, but as we have no need for this, we ignore it.

Our map function is simple. We pull out the year and the air temperature, since these are the only fields we are interested in. 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, continuing the 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 the below figure. At the bottom of the diagram is a Unix pipeline, which mimics the whole MapReduce flow.

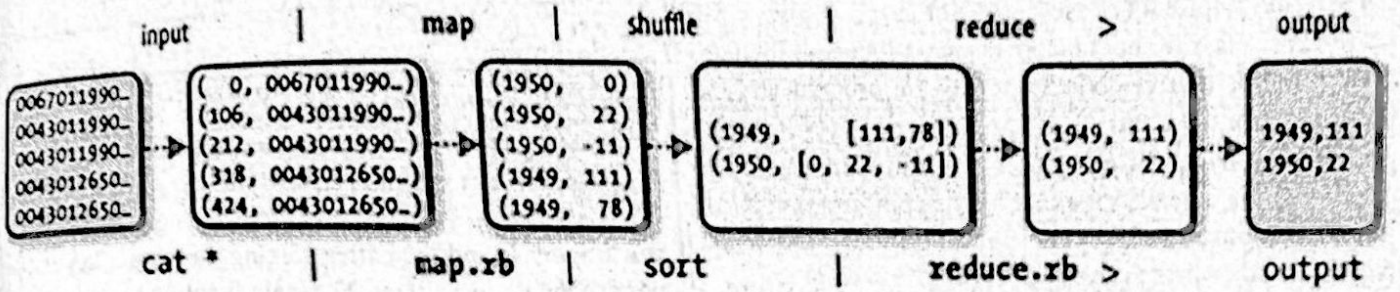

Fig.

Q.16 ***Explain the various difference between old and new java MapReduce API.***

Ans.

Difference	New API	Old API
Mapper and Reducer	New API using Mapper and Reducer as Class. So can add a method (with a default implementation) to an abstract class without breaking old implementations of the class	In Old API used Mapper and Reducer as Interface (still exist in New API as well)
Package	New API is in the org.apache.hadoop.mapreduce package	Old API can still be found in org.apache.hadoop.mapred.
User Code to communicate with MapReduce System	Use "context" object to communicate with MapReduce system	JobConf, the OutputCollector, and the Reporter object use for communicate with MapReduce System
Control Mapper and Reducer execution	New API allows both mappers and reducers to control the execution flow by overriding the run()method.	Controlling mappers by writing a MapRunnable, but no equivalent exists for reducers.
Job Control	Job control is done through the Job class in New API	Job control was done through JobClient (not exists in the new API)
Job Configuration	Job configuration done through configuration class via some of the helper methods on Job.	jobconf object was use for Job configuration which is extension of configuration class. java.lang.Object extended by org.apache.hadoop.conf. Configuration extended by org.apache.hadoop.mapred.JobConf
OutPut File Name	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).	In the old API both map and reduce outputs are named part-nnnnn
reduce ( ) method passes values	In the new API, the reduce() method passes values as a java.lang.Iterable	In the old API, the reduce() method passes values as a java.lang.Iterator

Q.17 *Explain mapper.How it works?*

Ans. Mapper : To serve as the mapper, a class implements from the Mapper interface and inherits the MapReduceBase class. The MapReduceBase class, not surprisingly, serves as the base class for both mappers and reducers. It includes two methods that effectively act as the constructor and destructor for the class:

void configure(JobConf job) —In this function you can extract the parameters set either by the configuration XML files or in the main class of your application. Call this function before any data processing begins.

void close ()—As the last action before the map task terminates, this function should wrap up any loose ends–database connections, open files, and so on.

The Mapper interface is responsible for the data processing step. It utilizes Java generics of the form Mapper<K1,V1,K2,V2> where the key classes and value classes implement the WritableComparable and Writable interfaces, respectively. Its single method is to process an individual (key/value) pair:

```
void map{K1 key,
    VI value,
    OutputCollector<K2,V2> output,
    Reporter reporter
    }throws IOException
```

The function generates a (possibly empty) list of (K2, V2) pairs for a given (K1, V1) input pair. The OutputCollector receives the output of the mapping process, and the Reporter provides the option to record extra information about the mapper as the task progresses. Hadoop provides a few useful mapper implementations. You can see some of them in the table.

Class	Description
IdentityMapper<K,V>	Implements Mapper<K,V,K,V> and maps inputs directly to outputs
InverseMapper<K, V>	Implements Mapper<K,V,V,K> and reverses the key/value pair
RegexMapper<K>	Implements Mapper<K, Text, Text, Long Writable> and generates a (match, 1) pair for every regular expression match
TokenCountMapper<K>	Implements Mapper<K, Text, Text, Long Writable> and generates a (token, 1) pair when the input value is tokenized

Q.18 *Define combiner.*

Ans. Combiner : A combiner, also known as a semi-reducer, is an optional class that operates by accepting the inputs from the Map class and thereafter passing the output key-value pairs to the Reducer class. The main function of a combiner is to summarize the map output records with the same key. The output (key-value collection) of the combiner will be sent over the network to the actual Reducer task as input.

The combiner class is used in between the Map class and the Reduce class to reduce the volume of data transfer between Map and Reduce. Usually, the output of the map task is large and the data transferred to the reduce task is high.

The following MapReduce task diagram shows the combiner phase.

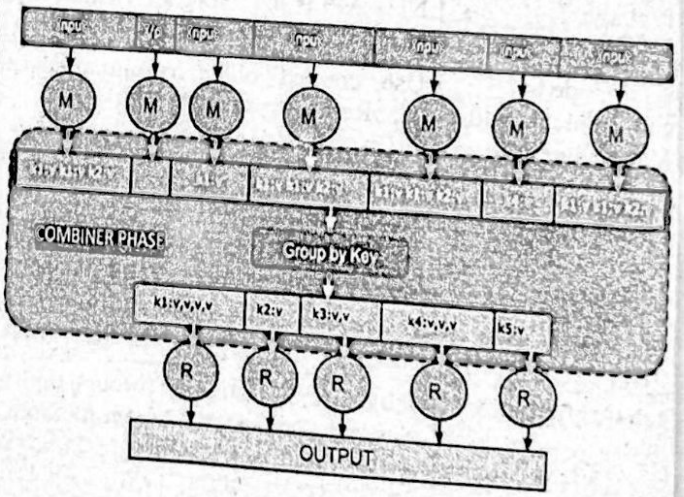

Fig.

Here is a brief summary on how MapReduce combiner works :

(i) A combiner does not have a predefined interface and it must implement the Reducer interface's reduce() method.

(ii) A combiner operates on each map output key. It must have the same output key-value types as the Reducer class.

(iii) A combiner can produce summary information from a large dataset because it replaces the original Map output.

Although, combiner is optional yet it helps segregating data into multiple groups for Reduce phase, which makes it easier to process.

Q.19 ***Explain how MapReduce work diagrametically with example.***

Ans. In MapReduce, during the map phase, it counts the words in each document, while in the reduce phase it aggregates the data as per the document spanning the entire collection. During the map phase, the input data is divided into splits for analysis by map tasks running in parallel across Hadoop framework.

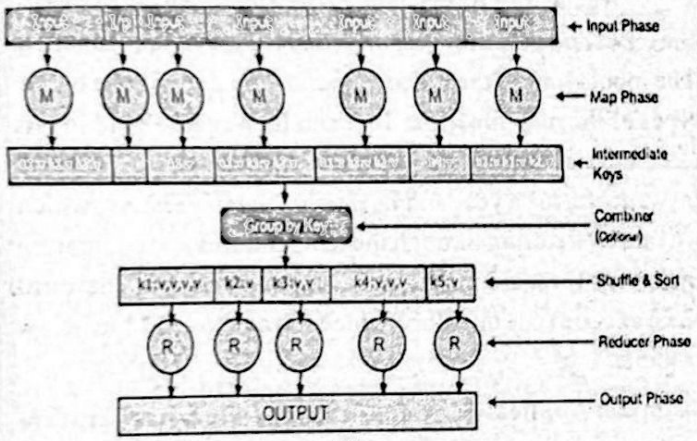

Fig.

Example

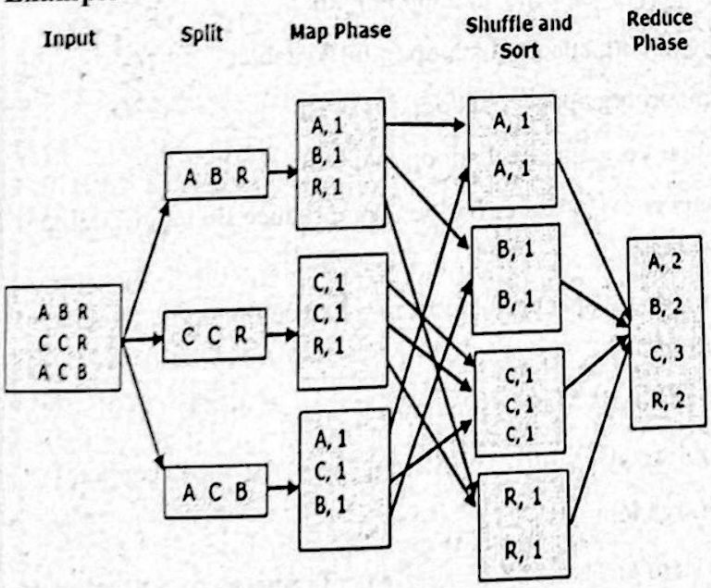

Fig.

PART-C

Q.20 ***What is Java Mapreduce? Explain in detail.***

Ans. Java MapReduce : Having run through how the MapReduce program works, the next step is to express it in code. We need three things: A map function, a reduce function, and some code to run the job. The map function is represented by the Mapper class, which declares an abstract map() method. The below example shows the implementation of our map method.

Example:Mapper for 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.char At(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 (air
Temperature));
}
}
}
```

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). Rather than use built-in Java types, Hadoop provides its own set of basic types that are optimized 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 the below example.

Example: Reducer for 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 max Value = Integer. MIN_VALUE;
for (IntWritable value : values) {
max Value = Math.max(max Value, value. get());
}
context.write(key, new IntWritable(max Value));
}
}
```

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. The third piece of code runs the MapReduce job as shown in the below example.

Example: 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.FileInput
Format;
import org.apache. hadoop. mapreduce.lib.output.FileOutput
Format;
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. 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 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.

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. As the name suggests, addInputPath() can be called more than once to use input from multiple paths. 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, as 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, as they are in our case. If they are different, then the map output types can be set using the methods setMapOutputKeyClass() and setMapOutput ValueClass().

The input types are controlled via the input format, which we have not explicitly set since we are using the default TextInputFormat.

After setting the classes that define the map and reduce functions, we are ready to run the job. 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.

Q.21 ***Explain a weather dataset in detail.***

Ans. MapReduce is a programming model for data processing. For our example, we will write a program that mines weather data. 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 MapReduce, since it is semistructured and record-oriented.

Data Format : The data we will use is from the National Climatic Data Center (NCDC, http://www.ncdc.noaa.gov/). The data is stored using a line-oriented ASCII format, in which each line is a record. The format supports a rich set of meteorological elements, many of which are optional or with variable data lengths. For simplicity, we shall focus on the basic elements, such as temperature, which are always present and are of fixed width.

The below example 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.

BDA.34

Example : 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 × 1000)
+028783  # longitude (degrees × 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 × 10)
1        # quality code
–0139    # dew point temperature (degrees Celsius × 10)
1        # quality code
10268    # atmospheric pressure (hectopascals × 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
```

Since there are tens of thousands of weather stations, the whole dataset is made up of a large number of relatively small files. It' s generally easier and more efficient to process a smaller number of relatively large files, so the data was preprocessed so that each year's readings were concatenated into a single file.

Example : 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 if the temperature is valid (the value 9999 signifies a missing value in the NCDC dataset) and if 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 (there were very few readings at the beginning of the century, so this is plausible). The complete run for the century took 42 minutes in one run on a single EC2 High-CPU Extra Large Instance.

Q.22 ***Write notes on followings :***

(a) Reducer

(b) Partitioner

Ans.(a) Reducer : As with any mapper implementation, a reducer must first extend the MapReduce base class to allow for configuration and cleanup. In addition, it must also implement the Reducer interface which has the following single method:

```
void reduce{K2 key,
Iterator<V2> values,
OutputCollector<K3,V3> output,
Reporter reporter
} throws IOException
```

When the reducer task receives the output from the various mappers, it sorts the incoming data on the key of the (key/value) pair and groups together all values of the same key. The reduce() function is then called, and it generates a (possibly empty) list of (K3, V3) pairs by iterating over the values associated with a given key. The OutputCollector receives the output of the reduce process and writes it to an output file. The Reporter provides the option to record extra information about the reducer as the task progresses.

The below table lists a couple of basic reducer implementations provided by Hadoop.

Class	Description
IdentityReducer <K, V>	Implements Reducer <K, V, K, V> and maps inputs directly to outputs
LongSumReducer<K>	Implements Reducer <K, LongWritable, K, LongWritable> and determines the sum of all values corresponding to the given key

Although we have referred to Hadoop programs as MapReduce applications, there is a vital step between the two stages: Directing the result of the mappers to the different reducers. This is the responsibility of the partitioner.

Ans.(b) Partitioner : A common misconception for first-time MapReduce programmers is to use only a single reducer. After all, a single reducer sorts all of your data before processing—and who doesn't like sorted data? Our discussions regarding MapReduce expose the folly of such thinking. We would have ignored the benefits of parallel computation. With one reducer, our compute cloud has been demoted to a compute raindrop.

With multiple reducers, we need some way to determine the appropriate one to send a (key/value) pair outputted by a mapper. The default behavior is to hash the key to determine

the reducer. Hadoop enforces this strategy by use of the HashPartitioner class. Sometimes the HashPartitioner will steer you awry.

Suppose you used the Edge class to analyze flight information data to determine the number of passengers departing from each airport. Such data may be:

(San Francisco, Los Angeles) Chuck Lam

(San Francisco, Dallas) James Warren

.....

If you used HashPartitioner, the two rows could be sent to different reducers. The number of departures would be processed twice and both times erroneously.

How do we customize the partitioner for your applications? In this situation, we want all edges with a common departure point to be sent to the same reducer. This is done easily enough by hashing the departureNode member of the Edge :

```
public class EdgePartitioner implements Partitioner
<Edge, Writable>
    {
@Override
public int getPartition(Edge key, Writable value, int
numPartitions)
    {
return key.getDepartureNode().hashCode() %
numPartitions;
    }
@Override
public void configure(JobConf conf) { }
    }
```

A custom partitioner only needs to implement two functions: Configure() and getPartition(). The former uses the Hadoop job configuration to configure the partitioner, and the latter returns an integer between 0 and the number of reduce tasks indexing to which reducer the (key/value) pair will be sent.

The exact mechanics of the partitioner may be difficu to follow. Between the map and reduce stages, a MapRedu application must take the output from the mapper tasks an distribute the results among the reducer tasks. This proce is typically called shuffling, because the output of a mappe on a single node may be sent to reducers across multipl nodes in the cluster.

Q.23 *Explain the component Driver.*

Ans. Driver : There is one important component of a Hadoop MapReduce program, called the Driver. The driver initializes the job and instructs the Hadoop platform to execute your code on a set of input files, and controls where the output files are placed. A cleaned-up version of the driver from the example Java implementation that comes with Hadoop is presented below:

```
public void run(String inputPath, String outputPath)
throws Exception
    {
      JobConf conf = new JobConf(WordCount.class);
      conf.setJobName("wordcount");
      // the keys are words (strings)
      conf.setOutputKeyClass(Text.class);
      // the values are counts (ints)
      conf.setOutputValueClass(IntWritable.class);
      conf.setMapperClass(MapClass.class);
      conf.setReducerClass(Reduce.class);
      FileInputFormat.addInputPath(conf, new Path
(inputPath));
      FileOutputFormat.setOutputPath(conf, new
Path(outputPath));
      JobClient.runJob(conf);
    }
```

The output from the reducers are written into files in the directory identified by outputPath. The configuration information to run the job is captured in the JobConf object.

The mapping and reducing functions are identified by the setMapperClass() and setReducerClass() methods. The data types emitted by the reducer are identified by setOutputKeyClass() and setOutputValueClass(). By default, it is assumed that these are the output types of the mapper as well. If this is not the case, the methods setMapOutputKey Class() and setMapOutputValueClass() methods of the JobConf class will override these. The input types fed to the mapper are controlled by the InputFormat used.

The default input format, "TextInputFormat," will load data in as (LongWritable, Text) pairs. The long value is the byte offset of the line in the file. The text object holds the string contents of the line of the file.

The call to JobClient.runJob(conf) will submit the job to MapReduce. This call will block until the job completes. If the job fails, it will throw an IOException. JobClient also provides a non- blocking version called submitJob().

Q.24 ***Explain RecordReader in detail.***

Ans. RecordReader : The RecordReader class actually loads the data from its source and converts it into (key, value) pairs suitable for reading by the Mapper. The RecordReader instance is defined by the InputFormat. The default InputFormat, TextInputFormat, provides a LineRecordReader, which treats each line of the input file as a new value. The key associated with each line is its byte offset in the file. The RecordReader is invoke repeatedly on the input until the entire InputSplit has been consumed. Each invocation of the RecordReader leads to another call to the map() method of the Mapper.

In the context of file-based input, the "start" is the byte position in the file where the RecordReader should start generating key/value pairs. The "end" is where it should stop reading records. These are not hard boundaries as far as the API is concerned—there is nothing stopping a developer from reading the entire file for each map task. While reading the entire file is not advised, reading outside of the boundaries it often necessary to ensure that a complete record is generated.

Example (LineRecordReader) : Let's get an example first that will hopefully make the code slightly more readable. Suppose my data set is composed on a single 300Mb file, spanned over 3 different blocks (blocks of 128Mb), and suppose that I have been able to get 1 InputSplit for each block. Let's imagine now 3 different scenarios.

File is composed on 6 lines of 50 Mb each as shown below :

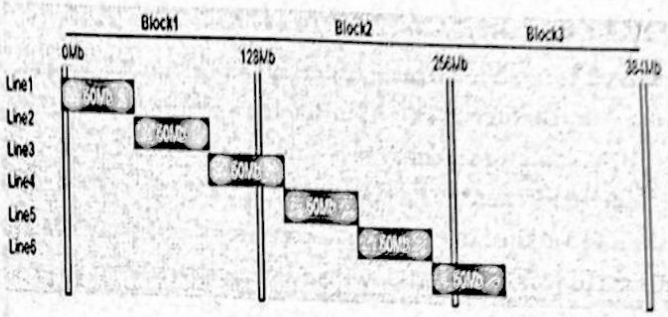

Fig.

The first Reader will start reading bytes from Block B1, position 0. The first two EOL will be met at respectively 50Mb and 100Mb. 2 lines (L1 & L2) will be read and sent as key / value pairs to Mapper 1 instance. Then, starting from byte 100Mb, we will reach end of our Split (128Mb) before having found the third EOL. This incomplete line will be completed by reading, the bytes in Block B2 until position 150Mb. First part of Line L3 will be read locally from Block B1, second part will be read remotely from Block B2 (by the mean of FSDataInputStream), and a complete record will be finally sent as key / value to Mapper 1.

The second Reader starts on Block B2, at position 128Mb. Because 128Mb is not the start of a file, there are strong chance our pointer is located somewhere in an existing record that has been already processed by previous Reader. We need to skip this record by jumping out to the next available EOL, found at position 150Mb. Actual start of RecordReader 2 will be at 150Mb instead of 128Mb.

We can wonder what happens in case a block starts exactly on a EOL. By jumping out until the next available record (through readLine method), we might miss 1 record. Before jumping to next EOL, we actually need to decrement initial "start" value to "start – 1". Being located at least 1 offset before EOL; we ensure no record is skipped!

Remaining process is following same logic, and everything is summarized in below table.

	Start	Actual Start	End	Line (s)
Mapper1	B1:0	B1:0	B2:150	L1,L2,L3
Mapper2	B2:128	B2:150	B3:300	L4,L5,L6
Mapper3	B3:256	B3:300	B3:300	N/A

Q.25 ***Explain MapReduce in detail.***

Ans. MapReduce : The MapReduce paradigm provides the means to break a large task into smaller tasks, run the tasks in parallel, and consolidate the outputs of the individual tasks into the final output. As its name implies, MapReduce consists of two basic parts—a map step and a reduce step—detailed as follows:

Map:

- Applies an operation to a piece of data
- Provides some intermediate output

Reduce:

- Consolidates the intermediate outputs from the map steps
- Provides the final output

Each step uses key/value pairs, denoted as <key, value>, as input and output. It is useful to think of the key/value pairs as a simple ordered pair. However, the pairs can take fairly complex forms. For example, the key could be a filename, and the value could be the entire contents of the file.

The simplest illustration of MapReduce is a word count example in which the task is to simply count the number of times each word appears in a collection of documents. In practice, the objective of such an exercise is to establish a list of words and their frequency for purposes of search or establishing the relative importance of certain words. Figure illustrates the MapReduce processing for a single input—in this case, a line of text.

<1234, "For each word in each string">

↓ Map

<For, 1> <each, 1> <word, 1> <in, 1> <each, 1> <string, 1>

↓ Reduce

<For, 1>

<each, 2>

<word, 1>

<in, 1>

<string, 1>

Fig.

In this example, the map step parses the provided text string into individual words and emits a set of key/value pairs of the form <word, 1>. For each unique key—in this example, word—the reduce step sums the 1 values and outputs the <word, count > key/value pairs. Because the word each appeared twice in the given line of text, the reduce step provides a corresponding key/value pair of <each, 2>.

It should be noted that, in this example, the original key, 1234, is ignored in the processing. In a typical word count application, the map step may be applied to millions of lines of text, and the reduce step will summarize the key/value pairs generated by all the map steps.

Expanding on the word count example, the final output of a MapReduce process applied to a set of documents might have the key as an ordered pair and the value as an ordered tuple of length 2n. A possible representation of such a key/value pair follows:

<(filename, datetime), (word1, 5, word2, 7,..., wordn, 6)>

In this construction, the key is the ordered pair filename and datetime. The value consists of the n pairs of the words and their individual counts in the corresponding file.

Of course, a word count problem could be addressed in many ways other than MapReduce. However, MapReduce has the advantage of being able to distribute the workload over a cluster of computers and run the tasks in parallel. In a word count, the documents, or even pieces of the documents, could be processed simultaneously during the map step. A key characteristic of MapReduce is that the processing of one portion of the input can be carried out independently of

the processing of the other inputs. Thus, the workload can be easily distributed over a cluster of machines.

U.S. Navy rear admiral Grace Hopper (1906-1992), who was a pioneer in the field of computers, provided one of the best explanations of the need for using a group of computers. She commented that during pre-industrial times, oxen were used for heavy pulling, but when one ox couldn't budge a log, people didn't try to raise a larger ox; they added more oxen. Her point was that as computational problems grow, instead of building a bigger, more powerful, and more expensive computer, a better alternative is to build a system of computers to share the workload. Thus, in the MapReduce context, a large processing task would be distributed across many computers.

Although the concept of MapReduce has existed for decades, Google led the resurgence in its interest and adoption starting in 2004 with the published work by Dean and Ghemawat. MapReduce has been used in functional programming languages such as Lisp, which obtained its name from being readily able to process lists (List processing).

In 2007, a well-publicized MapReduce use case was the conversion of 11 million New York Times news-paper articles from 1851 to 1980 into PDF files. The intent was to make the PDF files openly available to users on the Internet. After some development and testing of the MapReduce code on a local machine, the 11 million PDF files were generated on a 100-node cluster in about 24 hours.

What allowed the development of the MapReduce code and its execution to proceed easily was that the MapReduce paradigm had already been implemented in Apache Hadoop.

❑❑❑