UNIT 3

# HADOOP

## 3.1 HADOOP

Hadoop is an open source framework from Apache and is used to store process and analyze data which are very huge in volume. Hadoop is written in Java and is not OLAP (online analytical processing). It is used for batch/offline processing.It is being used by Facebook, Yahoo, Google, Twitter, LinkedIn and many more. Moreover it can be scaled up just by adding nodes in the cluster.

### Modules of Hadoop

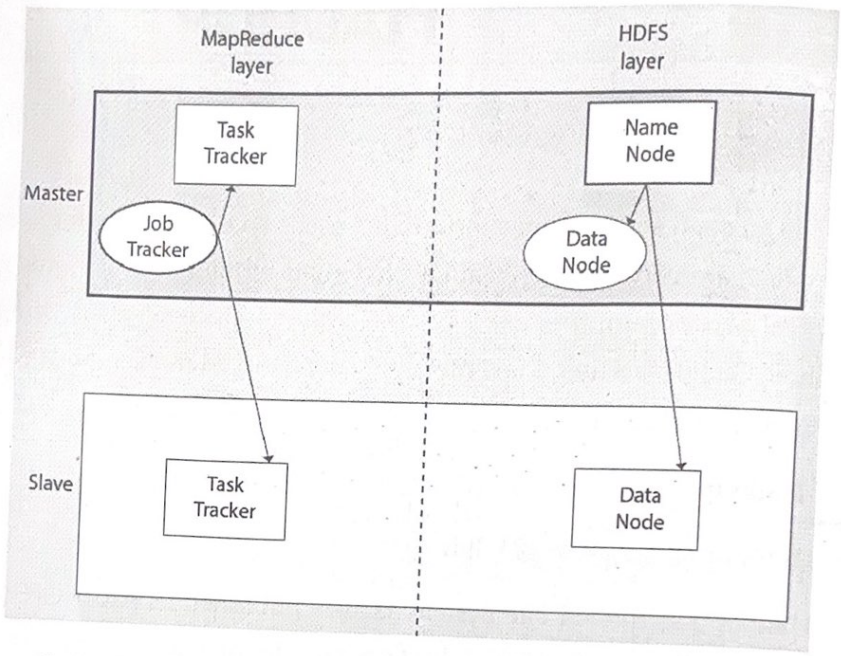
The Modules of Hadoop are as follows :

1. **HDFS:** Hadoop Distributed File System. Google published its paper GFS and on the basis of that HDFS was developed. It states that the files will be broken into blocks and stored in nodes over the distributed architecture.
2. **Yarn:** Yet another Resource Negotiator is used for job scheduling and manage the cluster.
3. **Map Reduce:** This is a framework which helps Java programs to do the parallel computation on data using key value pair. The Map task takes input data and converts it into a data set which can be computed in Key value pair. The output of Map task is consumed by reduce task and then the out of reducer gives the desired result.
4. **Hadoop Common:** These Java libraries are used to start Hadoop and are used by other Hadoop modules.

### Hadoop Architecture

The Hadoop architecture is a package of the file system, MapReduce engine and the HDFS (Hadoop Distributed File System). The MapReduce engine can be MapReduce/MR1 or YARN/MR2.

A Hadoop cluster consists of a single master and multiple slave nodes. The mas node includes Job Tracker, Task Tracker, NameNode, and DataNode whereas the sla node includes DataNode and TaskTracker.




**Advantages of Hadoop**

The Advantages of Hadoop are as follows :

1 **Fast :** In HDFS the data distributed over the cluster and are mapped which helps in faster retrieval. Even the tools to process the data are often on the same servers, thus reducing the processing time. It is able to process terabytes of data in minutes and Peta bytes in hours.

2 **Scalable:** Hadoop cluster can be extended by just adding nodes in the cluster.

3 **Cost Effective:** Hadoop is open source and uses commodity hardware to store data so it really cost effective as compared to traditional relational database management system.

4 **Resilient to failure:** HDFS has the property with which it can replicate data over the network, so if one node is down or some other network failure happens, then Hadoop takes the other copy of data and use it. Normally, data are replicated thrice but the replication factor is configurable.

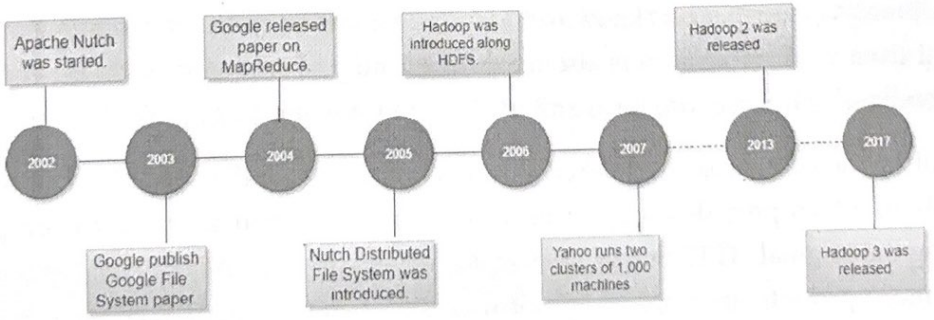
## 3.2 A BRIEF HISTORY OF HADOOP

It is an open-source software framework for distributed storage and distributed processing of very large data sets on computer clusters built from commodity hardware. It was created by Doug Cutting in 2005 who was working at Yahoo, named it after his son's toy elephant. It was originally developed to support distribution for the Nutch search engine project. Nutch was started in 2002, and a working crawler and search system quickly emerged. However, they realized that their architecture wouldn't scale to the billions of pages on the Web. NDFS and the MapReduce implementation in Nutch were applicable beyond the realm of search, and in February 2006 they moved out of Nutch to form an independent subproject of Lucene called Hadoop.

In January 2008, Hadoop was made its own top-level project at Apache, confirming its success and its diverse, active community. In April 2008, Hadoop broke a world record to become the fastest system to sort a terabyte of data. Running on a 910-node cluster, Hadoop sorted one terabyte in 209 seconds (just under 3½ minutes), beating the previous year's winner of 297 seconds (described in detail in "TeraByte Sort on Apache Hadoop on page 553). In November of the same year, Google reported that its MapReduce implementation sorted one terabyte in 68 seconds.

### Steps in History of Hadoop

The Hadoop was started by Doug Cutting and Mike Cafarella in 2002. Its origin was the Google File System paper, published by Google.




Let's focus on the history of Hadoop in the following steps:

✓ In 2002, Doug Cutting and Mike Cafarella started to work on a project, **Apache Nutch.** It is an open source web crawler software project.

- ✓ While working on Apache Nutch, they were dealing with big data. To store that data they have to spend a lot of costs which becomes the consequence of that project. This problem becomes one of the important reason for the emergence of Hadoop.
- ✓ In 2003, Google introduced a file system known as GFS (Google file system). It is a proprietary distributed file system developed to provide efficient access to data.
- ✓ In 2004, Google released a white paper on Map Reduce. This technique simplifies the data processing on large clusters.
- ✓ In 2005, Doug Cutting and Mike Cafarella introduced a new file system known as NDFS (Nutch Distributed File System). This file system also includes Map reduce.
- ✓ In 2006, Doug Cutting quit Google and joined Yahoo. On the basis of the Nutch project, Dough Cutting introduces a new project Hadoop with a file system known as HDFS (Hadoop Distributed File System). Hadoop first version 0.1.0 released in this year.
- ✓ Doug Cutting gave named his project Hadoop after his son's toy elephant.
- ✓ In 2007, Yahoo runs two clusters of 1000 machines.
- ✓ In 2008, Hadoop became the fastest system to sort 1 terabyte of data on a 900 node cluster within 209 seconds.
- ✓ In 2013, Hadoop 2.2 was released.
- ✓ In 2017, Hadoop 3.0 was released.

### 3.2.1 Apache Hadoop and the Hadoop Ecosystem

Although Hadoop is best known for MapReduce and its distributed filesystem (HDFS, renamed from NDFS), the term is also used for a family of related projects that fall under the umbrella of infrastructure for distributed computing and large-scale data processing.

All of the core projects covered in this book are hosted by the Apache Software Foundation, which provides support for a community of open source software projects, including the original HTTP Server from which it gets its name. As the Hadoop ecosystem grows, more projects are appearing, not necessarily hosted at Apache, which provide complementary services to Hadoop, or build on the core to add higher-level abstractions.

**The Hadoop projects that are covered in this book are described briefly here:**

1. **Common** : A set of components and interfaces for distributed filesystems and general I/O (serialization, Java RPC, persistent data structures).

2. **Avro :** A serialization system for efficient, cross-language RPC, and persistent data storage.
3. **MapReduce** : A distributed data processing model and execution environment that runs on large clusters of commodity machines.
4. **HDFS :** A distributed filesystem that runs on large clusters of commodity machines.
5. **Pig** : A data flow language and execution environment for exploring very large datasets. Pig runs on HDFS and MapReduce clusters.
6. **Hive** : A distributed data warehouse. Hive manages data stored in HDFS and provides a query language based on SQL (and which is translated by the runtime engine to MapReduce jobs) for querying the data.
7. **HBase :** A distributed, column-oriented database. HBase uses HDFS for its underlying storage, and supports both batch-style computations using MapReduce and point queries (random reads).
8. **ZooKeeper :** A distributed, highly available coordination service. ZooKeeper provides primitives such as distributed locks that can be used for building distributed applications.
9. **Sqoop** : A tool for efficiently moving data between relational databases and HDFS.

## 3.3 MAPREDUCE

MapReduce is a programming model for data processing. The model is simple, yet not too simple to express useful programs in. Hadoop can run MapReduce programs written in various languages; in this chapter, we shall look at the same program expressed in Java, Ruby, Python, and C++. Most important, MapReduce programs are inherently parallel, thus putting very large-scale data analysis into the hands of anyone with enough machines at their disposal. MapReduce comes into its own for large datasets, so let's start by looking at one.

### 3.3.1 Analyzing the Data with Hadoop

Hadoop takes advantage of the parallel processing for analyzing large volume of data in a cluster of computers. MapReduce is a distributed data processing model and execution environment that runs on large clusters of commodity machines, Hadoop can run MapReduce programs written in various languages.

### 3.3.2 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. 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. 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):

```
(0, 006701199099999**1950**051507004...9999999N9+**00001**+99999999999...)
(106, 004301199099999**1950**051512004...9999999N9+**00221**+99999999999...)
(212, 004301199099999**1950**051518004...9999999N9-**00111**+99999999999...)
(318, 004301265099999**1949**032412004...0500001N9+**01111**+99999999999...)
(424, 004301265099999**1949**032418004...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 Figure. At the bottom of the diagram is a Unix pipeline, which mimics the whole MapReduce flow, and which we will see again later in the chapter when we look at Hadoop Streaming.

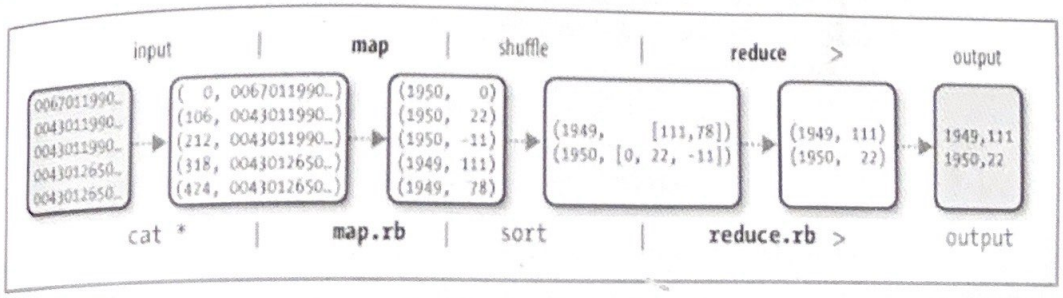



Figure: MapReduce logical data flow

### 3.3.3 Java Map Reduce

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. 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.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 th input key, input value, output key, and output value types of the map function. For th present example, the input key is a long integer offset, the input value is a line of text, th

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 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;
```

```
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));
  }
}
```

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 .

**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.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. 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 (it can be very annoying to accidentally overwrite the output of a long job with another).

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 setMapOutputValueClass().

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.

A test run After writing a MapReduce job, it's normal to try it out on a small dataset to flush out any immediate problems with the code. First install Hadoop in standalone mode– there are instructions for how to do this in Appendix A. This is the mode in which Hadoop runs using the local filesystem with a local job runner. Then install and compile the examples using the instructions on the book's website. Let's test it on the five-line sample discussed earlier (the output has been slightly reformatted to fit the page):

```
% export HADOOP_CLASSPATH=hadoop-examples.jar
% hadoop MaxTemperature input/ncdc/sample.txt output
11/09/15 21:35:14 INFO jvm.JvmMetrics: Initializing JVM Metrics with processName=JobT
racker, sessionId=
11/09/15 21:35:14 WARN util.NativeCodeLoader: Unable to load native-hadoop library fo
r your platform... using builtin-java classes where applicable
11/09/15 21:35:14 WARN mapreduce.JobSubmitter: Use GenericOptionsParser for parsing t
he arguments. Applications should implement Tool for the same.
11/09/15 21:35:14 INFO input.FileInputFormat: Total input paths to process : 1
11/09/15 21:35:14 WARN snappy.LoadSnappy: Snappy native library not loaded
11/09/15 21:35:14 INFO mapreduce.JobSubmitter: number of splits:1
11/09/15 21:35:15 INFO mapreduce.Job: Running job: job_local_0001
11/09/15 21:35:15 INFO mapred.LocalJobRunner: Waiting for map tasks
11/09/15 21:35:15 INFO mapred.LocalJobRunner: Starting task: attempt_local_0001_m_000
000_0
11/09/15 21:35:15 INFO mapred.Task:  Using ResourceCalculatorPlugin : null
11/09/15 21:35:15 INFO mapred.MapTask: (EQUATOR) 0 kvi 26214396(104857584)
11/09/15 21:35:15 INFO mapred.MapTask: mapreduce.task.io.sort.mb: 100
11/09/15 21:35:15 INFO mapred.MapTask: soft limit at 83886080


11/09/15 21:35:15 INFO mapred.MapTask: bufstart = 0; bufvoid = 104857600
11/09/15 21:35:15 INFO mapred.MapTask: kvstart = 26214396; length = 6553600
11/09/15 21:35:15 INFO mapred.LocalJobRunner:
11/09/15 21:35:15 INFO mapred.MapTask: Starting flush of map output
11/09/15 21:35:15 INFO mapred.MapTask: Spilling map output
11/09/15 21:35:15 INFO mapred.MapTask: bufstart = 0; bufend = 45; bufvoid = 104857600
11/09/15 21:35:15 INFO mapred.MapTask: kvstart = 26214396(104857584); kvend = 2621438
0(104857520); length = 17/6553600
11/09/15 21:35:15 INFO mapred.MapTask: Finished spill 0
11/09/15 21:35:15 INFO mapred.Task: Task:attempt_local_0001_m_000000_0 is done. And i
s in the process of commiting
11/09/15 21:35:15 INFO mapred.LocalJobRunner: map
11/09/15 21:35:15 INFO mapred.Task: Task 'attempt_local_0001_m_000000_0' done.
11/09/15 21:35:15 INFO mapred.LocalJobRunner: Finishing task: attempt_local_0001_m_00
0000_0
11/09/15 21:35:15 INFO mapred.LocalJobRunner: Map task executor complete.
11/09/15 21:35:15 INFO mapred.Task:  Using ResourceCalculatorPlugin : null
11/09/15 21:35:15 INFO mapred.Merger: Merging 1 sorted segments
11/09/15 21:35:15 INFO mapred.Merger: Down to the last merge-pass, with 1 segments le
ft of total size: 50 bytes
11/09/15 21:35:15 INFO mapred.LocalJobRunner:
11/09/15 21:35:15 WARN conf.Configuration: mapred.skip.on is deprecated. Instead, use
 mapreduce.job.skiprecords
11/09/15 21:35:15 INFO mapred.Task: Task:attempt_local_0001_r_000000_0 is done. And i
s in the process of commiting
11/09/15 21:35:15 INFO mapred.LocalJobRunner:
11/09/15 21:35:15 INFO mapred.Task: Task attempt_local_0001_r_000000_0 is allowed to
commit now
```

```
11/09/15 21:35:15 INFO mapred.Task: Task 'attempt_local_0001_r_000000_0' done.
11/09/15 21:35:16 INFO mapreduce.Job:  map 100% reduce 100%
11/09/15 21:35:16 INFO mapreduce.Job: Job job_local_0001 completed successfully
11/09/15 21:35:16 INFO mapreduce.Job: Counters: 24
    File System Counters
        FILE: Number of bytes read=255967
        FILE: Number of bytes written=397273
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
    Map-Reduce Framework
        Map input records=5
        Map output records=5
        Map output bytes=45
        Map output materialized bytes=61
        Input split bytes=124
        Combine input records=0
        Combine output records=0
        Reduce input groups=2
        Reduce shuffle bytes=0
        Reduce input records=5
        Reduce output records=2
        Spilled Records=10
        Shuffled Maps =0

        Failed Shuffles=0
        Merged Map outputs=0
        GC time elapsed (ms)=10
        Total committed heap usage (bytes)=379723776
    File Input Format Counters
        Bytes Read=529
    File Output Format Counters
        Bytes Written=29
```

When the hadoop command is invoked with a classname as the first argument, it launches a JVM to run the class. It is more convenient to use hadoop than straight java since the former adds the Hadoop libraries (and their dependencies) to the classpath and picks up the Hadoop configuration, too. To add the application classes to the classpath, we've defined an environment variable called HADOOP_CLASSPATH, which the hadoop script picks up.

When running in local (standalone) mode, the programs in this book all assume that you have set the HADOOP_CLASSPATH in this way. The commands should be run from the directory that the example code is installed in the output from running the job provides

some useful information. For example, we can see that the job was given an ID of job_local_0001, and it ran one map task and one reduce task (with the IDs attempt _ local_0001_m_000000_0 and attempt _ local_0001_r_000000_0). Knowing the job and task IDs can be very useful when debugging MapReduce jobs.

The last section of the output, titled "Counters," shows the statistics that Hadoop generates for each job it runs. These are very useful for checking whether the amount of data processed is what you expected. For example, we can follow the number of records that went through the system: five map inputs produced five map outputs, then five reduce inputs in two groups produced two reduce outputs.

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
```

This result is the same as when we went through it by hand earlier. We interpre as saying that the maximum temperature recorded in 1949 was 11.1°C, and in 1950 it w 2.2°C.

The old and the new Java MapReduce APIs The Java MapReduce API used in the previous section 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. It is type-incompatible with the old, however, so applications need to be rewritten to take advantage of it.

The new API is not complete in the 1.x (formerly 0.20) release series, so the old API is recommended for these releases, despite having been marked as deprecated in the early 0.20 releases. (Understandably, this recommendation caused a lot of confusion so the deprecation warning was removed from later releases in that series.)

**There are several notable 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 class2. For example, the Mapper and Reducer interfaces 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 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.

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. For example, records can be processed in batches, or the execution can be terminated before all the records have been processed. In the old API this is possible for mappers by writing a Map Runnable, but no equivalent exists for reducers.

5. 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 (used for configuring daemons; see "The Configuration API" on page 146). In the new API, this distinction is dropped, so job configuration is done through a Configuration.

6. Job control is performed through the Job class in the new API, rather than the old Job Client, which no longer exists in the new API.

7. Output files are named slightly differently: in the old API both map and reduce outputs are named part-nnnnn, while in the new API map outputs are named partm-nnnnn, and reduce outputs are named part-r-nnnnn (where nnnnn is an integer designating the part number, starting from zero).

8. User-overridable methods in the new API are declared to throw java.lang.Inter ruptedException. What this means is that you can write your code to be reponsive to interupts so that the framework can gracefully cancel long-running operations if it needs to3.

9. 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 :Shows the Max Temperature application rewritten to use the old API. The differences are highlighted in bold.

   When converting your Mapper and Reducer classes to the new API, don't forget to change the signature of the map() and reduce() methods to the new form. Just changing

your class to extend the new Mapper or Reducer classes will not produce a compilation error or warning, since these classes provide an identity form of the map() or reduce() method (respectively). Your mapper or reducer code, however, will not be invoked, which can lead to some hard-to-diagnose errors.

Example :. 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;

    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> {

    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(MaxTemperatureWithCombiner.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);
 }
}
```

## 3.4 SCALING OUT

Scaling out is the way of adding m ore data into system for processing. To scale out, we need to store the data in a distributed filesystem, typically HDFS to allow Hadoop to move the MapReduce computation to each machine hosting a part of the data.

### 3.4.1 Data Flow

#### Terminologies

A MapReduce job is a unit of work that the client wants to be performed: it consists of the input data, the MapReduce program, and configuration information. Hadoop runs the job by dividing it into tasks, of which there are two types: map tasks and reduce tasks. There are two types of nodes that control the job execution process: a jobtracker and a number of tasktrackers.

The jobtracker coordinates all the jobs run on the system by scheduling tasks to run on tasktrackers. Tasktrackers run tasks and send progress reports to the jobtracker, which keeps a record of the overall progress of each job. If a task fails, the jobtracker can reschedule it on a different tasktracker.

## Procedure

Hadoop divides the input to a MapReduce job into fixed-size pieces called input splits, or just splits. Hadoop creates one map task for each split, which runs the userdefined map function for each record in the split.

Having many splits means the time taken to process each split is small compared to the time to process the whole input. So if we are processing the splits in parallel, the processing is better load-balanced if the splits are small, since a faster machine will be able to process proportionally more splits over the course of the job than a slower machine. Even if the machines are identical, failed processes or other jobs running concurrently make load balancing desirable, and the quality of the load balancing increases as the splits become more fine-grained.

On the other hand, if splits are too small, then the overhead of managing the splits and of map task creation begins to dominate the total job execution time. For most jobs, a good split size tends to be the size of an HDFS block, 64 MB by default, although this can be changed for the cluster (for all newly created files), or specified when each file is created.

Hadoop does its best to run the map task on a node where the input data resides in HDFS. This is called the data locality optimization since it doesn't use valuable cluster bandwidth. Sometimes, however, all three nodes hosting the HDFS block replicas for a map task's input split are running other map tasks so the job scheduler will look for a free map slot on a node in the same rack as one of the blocks. Very occasionally even this is not possible, so an off-rack node is used, which results in an inter-rack network transfer. The three possibilities are illustrated in Figure.

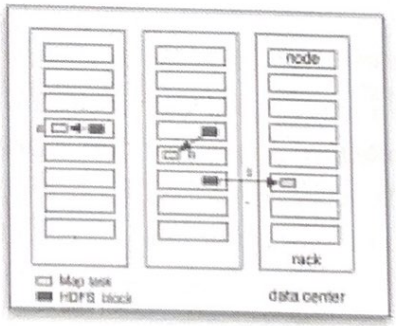



**Figure : Data-local (a), rack-local (b), and off-rack (c) map tasks.**

It should now be clear why the optimal split size is the same as the block size: it is the largest size of input that can be guaranteed to be stored on a single node. If the split spanned two blocks, it would be unlikely that any HDFS node stored both blocks, so some

of the split would have to be transferred across the network to the node running the map task, which is clearly less efficient than running the whole map task using local data.

Map tasks write their output to the local disk, not to HDFS. Why is this? Map output is intermediate output: it's processed by reduce tasks to produce the final output, and once the job is complete the map output can be thrown away. So storing it in HDFS, with replication, would be overkill. If the node running the map task fails before the map output has been consumed by the reduce task, then Hadoop will automatically rerun the map task on another node to re-create the map output.

Reduce tasks don't have the advantage of data locality—the input to a single reduce task is normally the output from all mappers. In the present example, we have a single reduce task that is fed by all of the map tasks. Therefore, the sorted map outputs have to be transferred across the network to the node where the reduce task is running, where they are merged and then passed to the user-defined reduce function.

The output of the reduce is normally stored in HDFS for reliability. As explanation for each HDFS block of the reduce output, the first replica is stored on the local node, with other replicas being stored on off-rack nodes. Thus, writing the reduce output does consume network bandwidth, but only as much as a normal HDFS write pipeline consumes. The whole data flow with a single reduce task is illustrated in Figure. The dotted boxes indicate nodes, the light arrows show data transfers on a node, and the heavy arrows show data transfers between nodes.

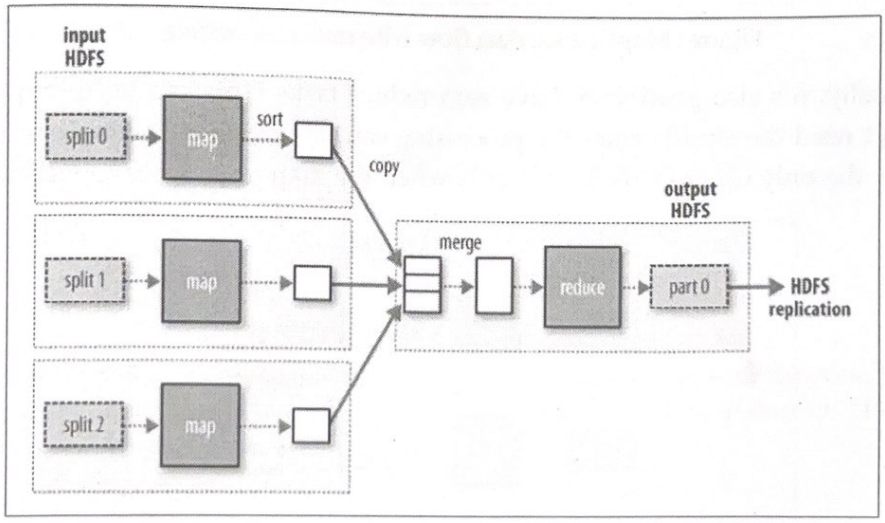



**Figure : MapReduce data flow with a single reduce task**

The number of reduce tasks is not governed by the size of the input, but is specified independently. In "The Default MapReduce Job". You will see how to choose the number

of reduce tasks for a given job. When there are multiple reducers, the map tasks partition their output, each creating one partition for each reduce task. There can be many keys (and their associated values) in each partition, but the records for any given key are all in a single partition. The partitioning can be controlled by a user-defined partitioning function, but normally the default partitioner—which buckets keys using a hash function—works very well.

The data flow for the general case of multiple reduce tasks is illustrated in figure. This diagram makes it clear why the data flow between map and reduce tasks is colloquially known as "the shuffle," as each reduce task is fed by many map tasks. The shuffle is more complicated than this diagram suggests, and tuning it can have a big impact on job execution time.

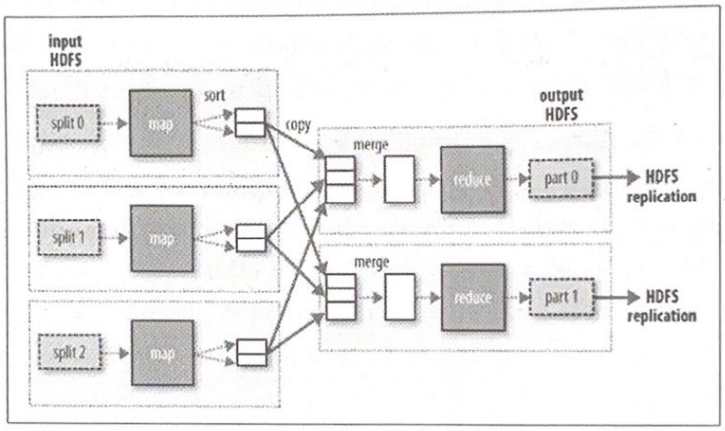



**Figure : MapReduce data flow with multiple reduce tasks**

Finally, it's also possible to have zero reduce tasks. This can be appropriate when you don't need the shuffle since the processing can be carried out entirely in parallel. In this case, the only off-node data transfer is when the map tasks write to HDFS.

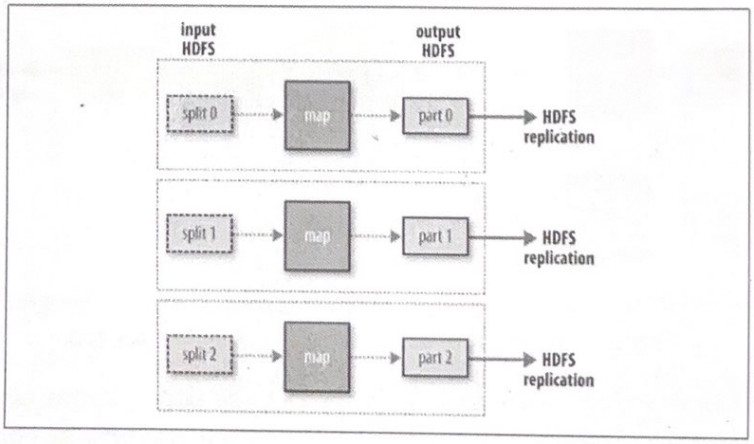



**Figure : MapReduce data flow with no reduce tasks**

### 3.4.2 Combiner Functions

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 contract for the combiner function constrains the type of function that may be used. This is best illustrated with an example. 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:

The contract for the combiner function constrains the type of function that may be used. This is best illustrated with an example. 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])

and the reduce would produce the same output as before. More succinctly, 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

Not all functions possess this property.4 For example, if we were calculating mean temperatures, then we couldn't use the mean as our combiner function, since:

```
mean(0, 20, 10, 25, 15) = 14
```

but:

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

The combiner function doesn't replace the reduce function. (How could it? The reduce function is still needed to process records with the same key from different maps.) But it can help cut down the amount of data shuffled between the maps and the reduces, and for this reason alone it is always worth considering whether you can use a combiner function in your MapReduce job.

Specifying a combiner function Going back to the 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).

Example :. 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);
  }
}
```

### 3.4.3 Running a Distributed Map Reduce Job

The same program will run, without alteration, on a full dataset. This is the point of MapReduce: it scales to the size of your data and the size of your hardware. Here's one data point: on a 10-node EC2 cluster running High-CPU Extra Large Instances, the program took six minutes to run.

### 3.4.4 Hadoop Streaming

It is provides as API to MapReduce that allows you to write your map and reduce functions in languages other than Java. It uses UNIX standard streams as the interface between Hadoop and user program, so we can use any language that can read standard input and write to standard output to write your MapReduce program. Map input data is passed over standard input to your map function, which processes it line by line and writes lines to standard output.

A map output key-value pair is written as a single tabdelimited line. Input to the reduce function is in the same format—a tab-separated key-value pair— passed over standard input. The reduce function reads lines from standard input, which the framework guarantees are sorted by key, and writes its results to standard output.

The map function can be expressed in Ruby.

**Example: Map function for maximum temperature in Ruby**

```
#!/usr/bin/env ruby

STDIN.each_line do |line|
  val = line
  year, temp, q = val[15,4], val[87,5], val[92,1]

  puts "#{year}\t#{temp}" if (temp != "+9999" && q =~ /[01459]/)
end
```

The program iterates over lines from standard input by executing a block for each line from STDIN (a global constant of type IO). The block pulls out the relevant fields from each input line, and, if the temperature is valid, writes the year and the temperature separated by a tab character \t to standard output (using puts).

Since the script just operates on standard input and output, it's trivial to test the script without using Hadoop, simply using Unix pipes:

```
% cat input/ncdc/sample.txt | ch02/src/main/ruby/max_temperature_map.rb
1950    +0000
1950    +0022
1950    -0011
1949    +0111
1949    +0078
```

The reduce function shown in Example is a little more complex.

**Example: Reduce function for maximum temperature in Ruby**

```
#!/usr/bin/env ruby

last_key, max_val = nil, 0
STDIN.each_line do |line|
  key, val = line.split("\t")
  if last_key && last_key != key
    puts "#{last_key}\t#{max_val}"
    last_key, max_val = key, val.to_i
  else
    last_key, max_val = key, [max_val, val.to_i].max
  end
end
puts "#{last_key}\t#{max_val}" if last_key
```

Again, the program iterates over lines from standard input, but this time we have to store some state as we process each key group. In this case, the keys are weather station identifiers, and we store the last key seen and the maximum temperature seen so far for that key. The MapReduce framework ensures that the keys are ordered, so we know that if a key is different from the previous one, we have moved into a new key group. In contrast to the Java API, where you are provided an iterator over each key group, in streaming you have to find key group boundaries in your program

For each line, we pull out the key and value, then if we've just finished a group (last_key && last_key != key), we write the key and the maximum temperature for that group, separated by a tab character, before resetting the maximum temperature for the new key. If we haven't just finished a group, we just update the maximum temperature for the current key.