

# UNIT-3



***Map Reduce***

**Objective:** To familiarize with the Map Reduce of Big data

**Syllabus:**

- Java Map Reduce
- Introduction to Weather Dataset
- Analyzing weather data with UNIX tools
- Analyzing weather data with Map and Reduce
- Word Count Program using Map Reduce
- Combiner Functions
  
- Running a Distributed Map Reduce Job
- Anatomy of a Map Reduce Job Run
- Shuffle and Sort.

## Learning Outcomes:

At the end of the unit, students will be able to:

1. Analyzing Map Reduce with Unix ,hadoop, java tools
2. Explain the data flow.
3. Develop the Map Reduce using the java.
4. Develop the Map Reduce in distributed Environment.

## Introduction

- MapReduce is a **programming model** for data processing.
- Hadoop can run MapReduce programs written in various languages like in **Java, Ruby, Python, and C++**.
- MapReduce programs are **inherently parallel**.

## A Weather Dataset

The task is to mine weather data. Weather **sensors collecting data** every hour at many locations **across the globe** gather a large volume of log data, for analysis with MapReduce, it is **semistructured** and record-oriented.

### Data Format

- The data collect 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. Take basic elements, such as **temperature**, which are always present and are of fixed width.
- 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**.

# Input data

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

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

Valid quality codes:

**0,1,4,5,9**

Missing value : **9999**

**WBAN** is a five-digit **station** identifier  
**Weather-Bureau-Army-Navy**

Hourly observations taken by U.S. Air Force personnel at bases in the United States and around the world. Foreign observations concentrated in the Middle East and Japan. Stations assigned WBAN numbers. Original forms sent from the Air Force to NCDC by agreement and stored in the NCDC archives.

- 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.

**Ex:** 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
```

- There are tens of **thousands of weather stations**, the whole dataset is made up of a **large number of relatively small files**.
- It is easier and more efficient to process a smaller number of relatively large files, so the data was **preprocessed** by each year's readings were concatenated into a single file.
- Each year of readings comes in a separate file. Here's a partial directory listing of the files:

```
1901.tar.bz2  
1902.tar.bz2  
1903.tar.bz2  
...  
2000.tar.bz2
```

## Analyzing the Data with Unix Tools

- Without using Hadoop can analyze data with Unix tools.
- The classic tool for processing line-oriented data is *awk*, a small script for finding maximum recorded temperature by year from NCDC weather records is :

```
#!/usr/bin/env bash
for year in all/*           // loops through compressed year files
do
echo -ne `basename $year .gz`"\t" // print year
gunzip -c $year | \
awk '{ temp = substr($0, 88, 5) + 0; // extract air temperature
q = substr($0, 93, 1); // extract quality code
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.

Run the program

```
% ./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 temperature of 31.7°C for 1901.
- The complete run for the century took 42 minutes in one run on a single EC2 High-CPU Extra Large Instance.
- To speed up the processing, we need to run parts of the program in parallel.
- process different years in different processes, using all the available hardware threads on a machine.

There are a few problems with this,

**First, dividing the work into equal-size pieces isn't always easy** , the file size for different years varies widely, so some processes will finish much earlier than others. The whole run is dominated by the longest file.

A better approach, that requires more work, is to split the input into fixed-size chunks and assign each chunk to a process.

**Second, combining the results from independent processes** may need further processing, the result for each year is independent of other years and may be combined by concatenating all the results, and sorting by year.

If using the fixed-size chunk approach, the combination is more delicate. For this example, data for a particular year will typically be split into several chunks, each processed independently.

Find maximum temperature for each chunk, in the final step is to look for the highest of these maximums, for each year.

**Third, you are still limited by the processing capacity of a single machine.** If the best time you can achieve is 20 minutes with the number of processors you have, then that's it. You can't make it go faster.

Some datasets grow beyond the capacity of a single machine. When we start using multiple machines, consider the factors category of coordination and reliability.

Who runs the overall job? How do we deal with failed processes?

Using a framework like Hadoop to take care of these issues is a great help.

## 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, run it on a cluster of machines.

### Map and Reduce

MapReduce works by breaking the processing into **two phases**:

- 1) the map phase and
- 2) the reduce phase.

- Each phase has **key-value pairs** as input and output,
- The input to our map phase is the raw NCDC data, 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

**Problem** : Find maximum global temperature recorded in each year

- **map function** : pull out the **year** and the **air temperature**.

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.

- **reduce function** : **finding the maximum temperature** for each year.
- To visualize the way the map works, consider the following sample lines of input data

# Input data



- 1) sample lines of input data  
fields are packed into one line with no delimiters

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

```
00670119909999991950051507004...9999999N9+00001+99999999999...
00430119909999991950051512004...9999999N9+00221+99999999999...
00430119909999991950051518004...9999999N9-00111+99999999999...
00430126509999991949032412004...0500001N9+01111+99999999999...
00430126509999991949032418004...0500001N9+00781+99999999999...
```

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

(15-19)

(87-92) (92-93)

```
(0, 00670119909999991950051507004...9999999N9+00001+99999999999...)
(106, 00430119909999991950051512004...9999999N9+00221+99999999999...)
(212, 00430119909999991950051518004...9999999N9-00111+99999999999...)
(318, 00430126509999991949032412004...0500001N9+01111+99999999999...)
(424, 00430126509999991949032418004...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.

- Input for the reducer is :

(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:

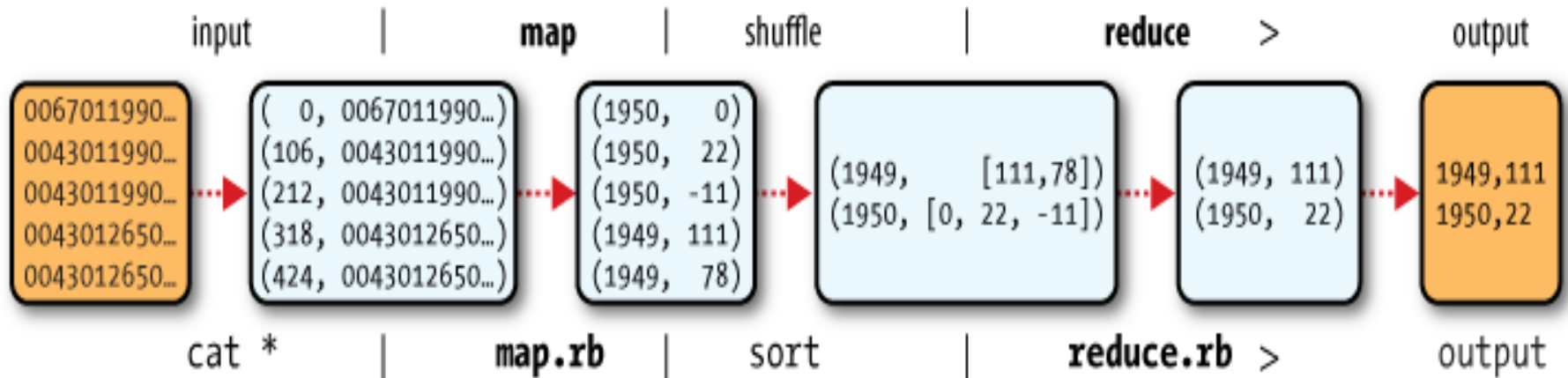
(1949, 111)

(1950, 22)

- This is the final output:** the **maximum global temperature recorded in each year.**



The whole data flow is illustrated in Figure



*MapReduce logical data flow*

## Java MapReduce

- 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

## 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 reduce function is similarly defined using a Reducer



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

The third piece of code runs the MapReduce job

### *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 test run

After writing a MapReduce job,

```
% 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 committing
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 committing
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 output.FileOutputCommitter: Saved output of task 'attempt_loca
l_0001_r_000000_0' to file:/Users/tom/workspace/hadoop-book/output
11/09/15 21:35:15 INFO mapred.LocalJobRunner: reduce > reduce
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

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
```

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



# 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**.

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.

Calling the combiner function zero, one, or many times should produce the same output from the reducer.

illustrated with maximum temperature example, readings for the year 1950 were processed by two maps (because they were in different splits).

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)**

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.

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.

For example, if we were calculating mean temperatures, then we couldn't use the mean as our combiner function,

since:  $\text{mean}(0, 20, 10, 25, 15) = 14$

but:  $\text{mean}(\text{mean}(0, 20, 10), \text{mean}(25, 15)) = \text{mean}(10, 20) = 15$

The combiner function doesn't replace the reduce function.

The reduce function is still needed to process records with the **same key from different maps**.

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

In 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

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);  
    }  
}
```



# Running a Distributed MapReduce 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.

## **practical aspects of developing a MapReduce application in Hadoop.**

Writing a program in MapReduce has a certain flow to it. You start by writing your map and reduce functions, ideally with unit tests to make sure they do what you expect.

Then you write a driver program to run a job, which can run from your IDE using a small subset of the data to check that it is working.

If it fails, then you can use your IDE's debugger to find the source of the problem.

With this information, you can expand your unit tests to cover this case and improve your mapper or reducer as appropriate to handle such input correctly

## UNIT-IV

### How MapReduce works

**Syllabus** : Classic map reduce, job submission, job initialization, task assignment, task execution, progress and status updates, job completion, shuffle and sort on map and reducer side.

#### Introduction

- Run a MapReduce job with method call: `submit()` on a Job object which will submit the job and call `waitForCompletion()`, wait for it to finish.
- Hadoop executes a MapReduce program depends on configuration settings.
- In releases of Hadoop up to 0.20 release series, `mapred.job.tracker` determines the means of execution.
- \* If the configuration property is set to local, local job runner is used. This runner runs the whole job in a single JVM. It's designed for testing and for running MapReduce programs on small datasets.

\* if `mapred.job.tracker` is set to a colon-separated host and port pair, then the property is interpreted as a jobtracker address, and the runner submits the job to the jobtracker at that address.

## 4.1 Classic MapReduce (MapReduce 1)

- A job run in classic MapReduce is illustrated in Figure .
- At the highest level, there are **four independent entities**:
  1. The **client**, which submits the MapReduce job.
  2. The **jobtracker**, which coordinates the job run. The jobtracker is a Java application whose main class is `JobTracker`.
  3. The **tasktrackers**, which run the tasks that the job has been split into. Tasktrackers are Java applications whose main class is `TaskTracker`.
  4. The distributed filesystem (normally **HDFS**), which is used for sharing job files between the other entities.

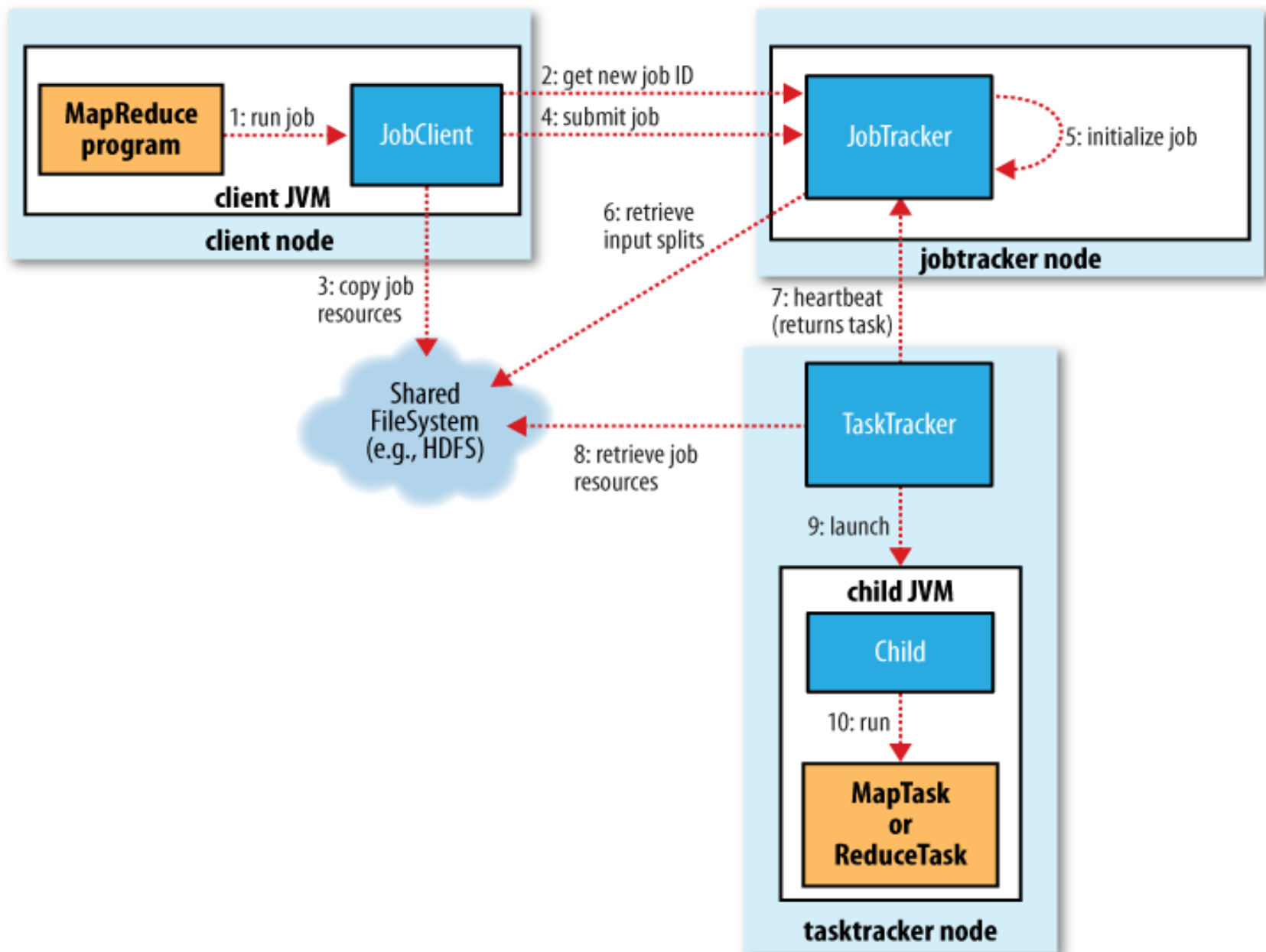


Figure 1 : How Hadoop runs a MapReduce job using the classic framework



## 4.2 Job Submission

The `submit()` method on Job creates an internal JobSubmitter instance and calls `submitJobInternal()` on it (**step 1** in Figure).

After submitted the job, `waitForCompletion()` polls the job's progress once a second and reports the progress to the console.

When the job is complete, if it was successful, the job counters are displayed. Otherwise, the error that caused the job to fail is logged to the console.

**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. Ex: 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

- the job JAR file
- the configuration file
- 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**).

## 4.3 Job Initialization

- When the JobTracker receives a call to its submitJob() method, it **puts it into** an **internal queue** from where the job scheduler will pick it up and initialize it

### Initialization involves

Bookkeeping information to keep track of the tasks' status and progress (**step 5**).  
creating an object to represent the job being run, which encapsulates its tasks

1. 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**).
2. It then **creates** one **map task** for each split.
3. The number of reduce tasks to create is determined by the **mapred.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.
4. Tasks are given IDs at this point. In addition to the map and reduce tasks, **two further tasks** are created: a job **setup** task and a job **cleanup** task.
5. These are run by tasktrackers and are used to run code to setup the job before any map tasks run, and to cleanup after all the reduce tasks are complete.

6. The **OutputCommitter** that is configured for the job determines the **code to be run**, and by default this is a FileOutputCommitter.
7. The job setup task will create the final output directory for the job and the temporary working space for the task output.
8. The job cleanup task it will delete the temporary working space for the task output.

## 4.4 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**, but they also double as a channel for messages.
- 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)**.
- Before it can choose a task for the tasktracker, the jobtracker must choose a job to select the task from. There are various scheduling algorithms but the default one simply maintains **a priority list of jobs**.
- Having chosen a job, the jobtracker now **chooses a task for the job**. Tasktrackers have a fixed number of slots for map tasks and for reduce tasks:  
    Ex: 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)

- The default scheduler fills empty map task slots before reduce task slots.
- If the tasktracker has at least one empty map task slot, the jobtracker will select a map task; otherwise, it will select a reduce task.
- To choose a **reduce task**, the jobtracker simply takes the next in its list of yet-to-be-run reduce tasks, since there are **no data locality considerations**.
- For a **map task**, however, it takes account of the tasktracker's **network location** and **picks a task** whose input split is as **close as possible to the tasktracker**.
- In the optimal case, the **task is data-local**, *that is*, running on the same node that the split resides on.
- Alternatively, the task may be **rack-local**: *on the same rack, but not the same node, as the split. Some tasks are neither data-local nor rack-local and retrieve their data from a different rack from the one they are running on.*
- You can tell the proportion of each type of task by looking at a job's counters

## 4.5 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**
- Second, 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. TaskRunner launches a new Java Virtual Machine (**step 9**) to run each task in (**step 10**) .
- The **child process communicates with its parent** through the *interface to* informs the task's progress every few seconds until the task is complete.

- Each task can perform **setup and cleanup** actions, which are run in the same JVM as the task itself, and are determined by the OutputCommitter for the job
- The **cleanup** action is used to **commit the task**, in the case of file-based jobs its output is written to the final location for that task.
- Both **Streaming and Pipes** run special map and reduce tasks for the purpose of launching the **user-supplied executable** and communicating with it (Figure 2).
- In the case of **Streaming**, the Streaming task communicates with the process (which may be written in any language) using **standard input and output streams**.
- The **Pipes** task, on the other hand, listens on a socket and passes the C++ process a **port number** in its environment, so that on startup, the C++ process can establish a persistent **socket connection** back to the parent Java Pipes task.



- In both cases, during execution of the task, the Java process passes **input key-value pairs** to the external process, which runs it through the user-defined map or reduce function and passes the **output key-value** pairs back to the Java process.
- From the tasktracker's point of view, it is as if the tasktracker child process ran the map or reduce code itself.

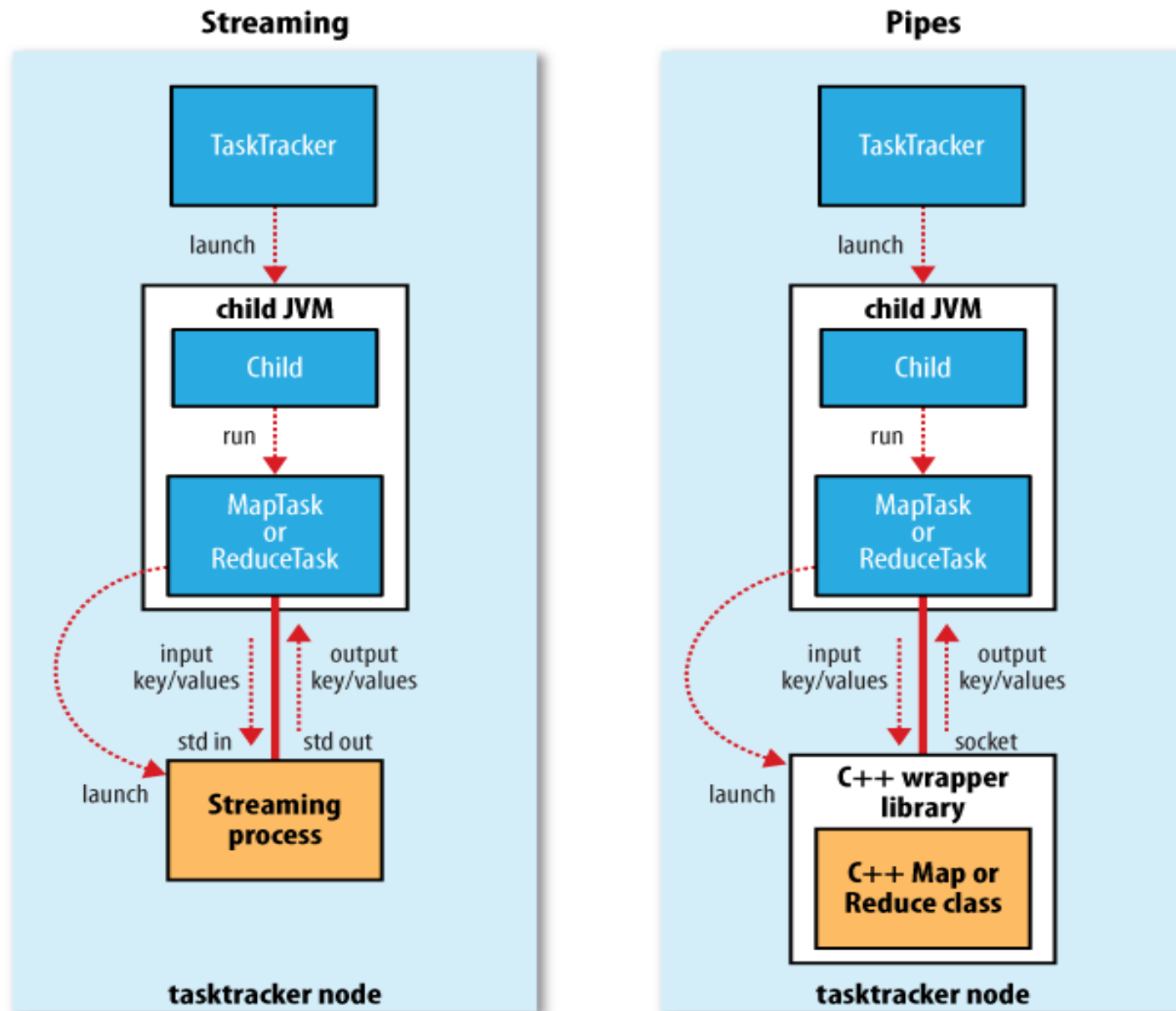


Figure 2 : Streaming and Pipes

## 4.6 Progress and Status Updates

- MapReduce jobs are long-running batch jobs, taking anything from minutes to hours to run, the user has to get feedback on how the job is progressing.
- A job and each of its tasks have a *status* includes
  1. state of the job or task (e.g., running, successfully completed, failed)
  2. the progress of maps and reduces
  3. the values of the job's counters
  4. status message or description (which may be set by user code)
- These statuses change over the course of the job
- When a task is running, it keeps track of its *progress, that is, the proportion of the task* completed.
  - For map tasks, this is the *proportion of the input* that has been *processed*.
  - For reduce tasks, system estimate the proportion of the reduce input processed by dividing the total *progress* into *three parts*, corresponding to the three phases of the shuffle.

- Tasks also have a set of **counters** that count various events as the task runs such as the number of map output records written .
- It sets a **flag** to indicate that the **status change** should be sent to the tasktracker.
- The flag is **checked** in a separate **thread** every three seconds.
- The jobtracker **combines** these updates to produce a **global view** of the status of all the jobs being run and their constituent tasks.
- Finally, the **Job** receives the latest **status** by polling the jobtracker every second.
- **Clients** can also use Job's **getStatus()** method to obtain a JobStatus instance, which contains all of the status information for the job.

## 4.7 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.”
- when the **Job polls** for status, it learns that the job has completed successfully, it prints a message to tell the user and then returns from the `waitForCompletion()` method.
- Last, the jobtracker **cleans up** its working state for the job and instructs tasktrackers to do the same (so intermediate output is deleted, for example).

## 4.8 Shuffle and Sort

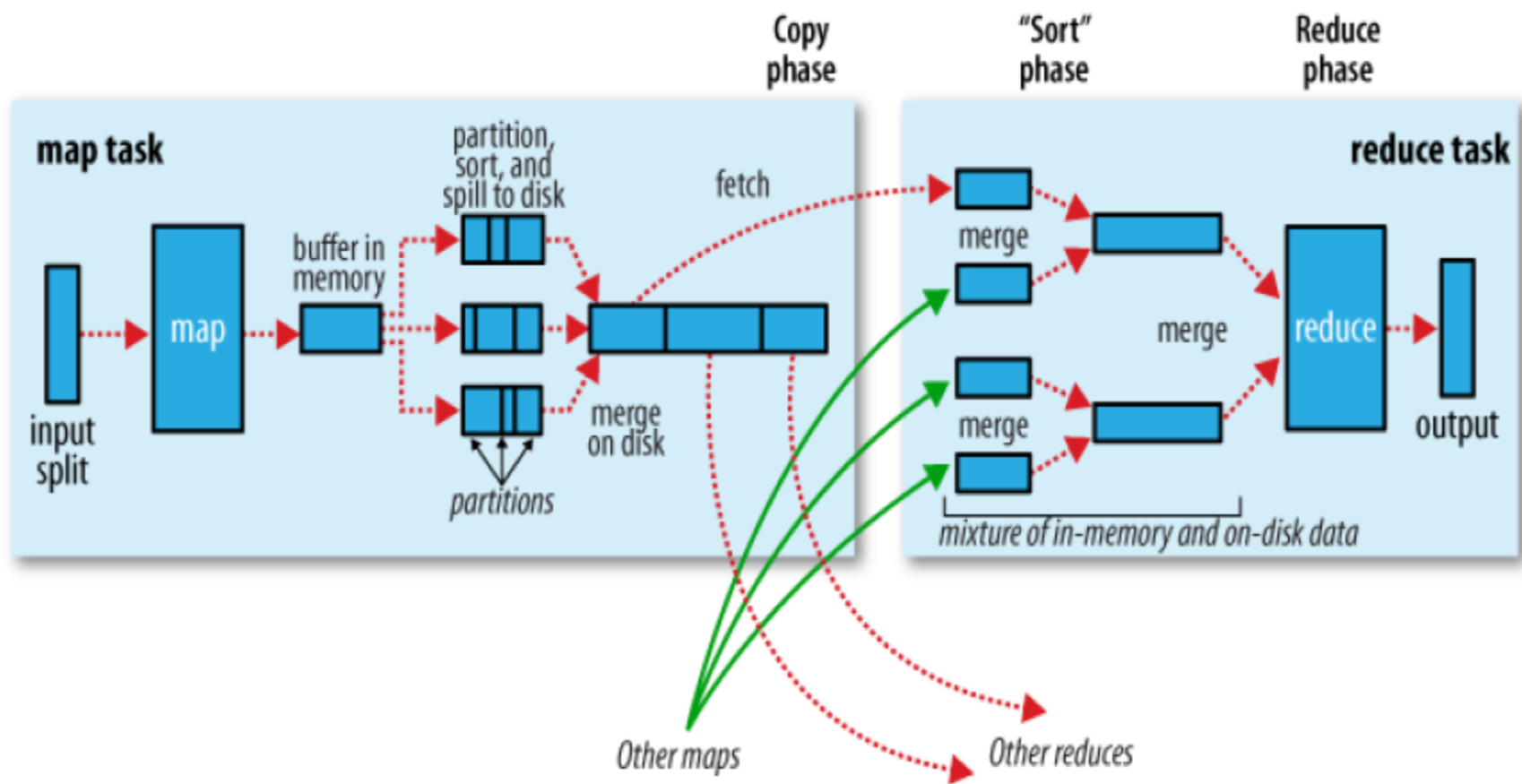
MapReduce makes the guarantee that the **input to every reducer is sorted** by key.

- The process by which the system performs the sort—and transfers the map outputs to the reducers as inputs—is known as the **shuffle**.
- The shuffle is an area of the codebase where **refinements and improvements** are continually being made.

### Shuffle sort on Map Side

When the map function starts producing **output**, it is **not** simply **written to disk**.

The process is more involved, and takes advantage of **buffering** writes in memory and doing some presorting for efficiency reasons.



Shuffle and sort in MapReduce

The buffer is **100 MB** by default, change the size by using **io.sort.mb** property.

- When the contents of the buffer reaches a certain **threshold** size a background thread will start to **spill** the contents to disk.
- Map outputs will continue to be written to the buffer while the spill takes place, but if the buffer fills up during this time, the **map will block** until the spill is complete.
- Spills are written in **round-robin fashion** to the directories specified by the **mapred.local.dir** property.
- Before it writes to disk, the thread first **divides the data** into partitions corresponding to the reducers to send.
- Within **each partition**, the background thread performs an in-memory **sort by key**, and if there is a combiner function, it is run on the output of the sort.
- Running the combiner function makes more compact map output, so less data to write to local disk and to transfer to the reducer.



- Each time the memory buffer reaches the spill threshold, a **new spill** file is created .
- Before the task is finished, the spill files are **merged** into a single partitioned and sorted output file.
- The configuration property **io.sort.factor** controls the maximum **number of streams to merge** at once; the default is 10.
- If there are at least three spill files then the combiner is run again before the output file is written.
- Compress the map output as it is written to disk, makes it faster to write to disk, saves disk space, and reduces the amount of data to transfer to the reducer.
- By default, the output is not compressed, but it is easy to enable by setting **mapred.compress.map.output** to true.
- The output file's partitions are made available to the reducers over HTTP.

- The maximum number of worker threads used to serve the file partitions is controlled by the `tasktracker.http.threads` property. The default of `40` may need increasing for large clusters running large jobs.

## **Shuffle and sort on Reduce Side**

The map output file is sitting on the local disk of the machine that ran the map task. The reduce task needs the map output for its particular partition from several map tasks across the cluster.

There are three phases for reducer 1) copy phase 2) sort phase 3) reduce phase.

### **1) Copy phase:**

- The map tasks may finish at different times, so the reduce task starts copying their outputs as soon as each completes.
- The reduce task has a small number of copier threads so that it can fetch map outputs in parallel.
- The default is five threads, but this number can be changed by setting the `mapred.reduce.parallel.copies` property.

- The map outputs are copied to reduce task JVM's memory otherwise, they are copied to disk. When the in-memory buffer reaches a threshold size or reaches a threshold number of map outputs it is merged and spilled to disk.
- Any map outputs that were compressed have to be decompressed in memory in order to perform a merge on them.
- When all the map outputs have been copied, the reduce task moves into the sort phase.

## **2) Sort phase:**

- In this phase merge the map outputs, maintaining their sort ordering.
- This is done in rounds. For example, if there were 50 map outputs, and the merge factor was 10, then there would be 5 rounds. Each round would merge 10 files into one, so at the end there would be five intermediate files.
- These five files into a single sorted file, the merge saves a trip to disk by directly feeding the reduce function. This final merge can come from a mixture of in-memory and on-disk segments.

### **3) Reduce phase:**

- During the reduce phase, the reduce function is invoked for each key in the sorted output.
- The output of this phase is written directly to the output filesystem, typically HDFS.
- In the case of HDFS, since the tasktracker node is also running a datanode, the first block replica will be written to the local disk.