Date Page

UNIT-3

MapReduce.

→ MapReduce is a programming model for data processing. The model is simple and Hadoop can run MapReduce programs written in various language like Java, Ruby, Python and C++.

⇒ A weather Dataset

→ Weather sensors collecting data every hour at many locations across the globe gather large amounts of log data, which is good candidate for analysis with MapReduce since it is a semi structured & record-oriented.

*) Data format :-

→ The data is taken from National climatic Data center (NCDC) and is stored using a line-oreinted ASCII format, in which each line is a record.

Example :- Format of NCDC record.

```
332130      # USAF weather station identifier.
19500101    # Observation date
0300        # Observation time.
+51317      # latitude. (degrees X 1000)
+028783     # longitude. (degrees X 1000)
320         # wind direction. (degrees)
-0128       # air temperature (degree celsius X 10)
10268       # atmospheric pressure (hectopascals X 10)
```

Date Page

⇒ Analysing the data with Unix Tools

→ The classical tool for processing line-oriented data is "awk".

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 awk script extracts two fields from the NCDC data: the air temperature and the quality code.

→ The script loops through compressed year files, first printing the year & then processing file using awk.

→ The air temperature value is turned into an integer by adding 0. Next, a test is applied to see if the temperature is valid (9999 = missing) and if quality code indicates that reading is not erroneous.

→ If reading is OK, the value is compared with maximum value seen so far, which is

updated if new maximum is found.
→ The 'END' block is executed after all the lines in the files have been processed, and it prints the maximum value.

Example Output/result:-

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

⇒ Analysing the Data with Hadoop.

→ 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.

Here, the input to our map phase is raw NCDC data.
→ The output from map function is processed by MapReduce framework and sent to the reduce function.
→ The processing sorts and groups the key-value pairs by key. Each year appears with a list of all its air temperature reading.
→ Reduce function iterates & picks up the maximum reading and displays.

Date Page

Java MapReduce :- It consists of three things : a
a) : map function (Mapper class)
b) : reduce function (Reducer class)
c) : run the job (Runner class)

Example :- Mapper for maximum temperature

```
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 MaxTempMapper extends Mapper < LongWritable,
  Text, Text, IntWritable >
{  private static final int MISSING = 9999;
   public void map ( LongWritable, Text, Text value,
     Context context) throws IOException
   {  String line = value.toString();
      String year = line.substring (15, 19);
      int airTemperature;
      if (line.charAt(87) == '+')
      {  airTemperature = Integer.parseInt(line.
            substring(88, 92));
      }
      else {
      {  airTemperature = Integer.parseInt(line.substring
            (87, 92));
      }
      String quality = line.substring(92, 98);
      if ( airTemperature != MISSING && quality.
           matches ("[01459]"))
```

```
    { context.write (new Text(year), new IntWritable
        (airTemperature));
    }
  }
}
```

Example: Reducer for maximum temperature.

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class MaxTempReducer extends Reducer < Text,
  IntWritable, Text, IntWritable >
{  public void reduce ( Text key, Iterable <IntWritable>
     values, Context context ) throws IOException
   {
      int maxValue = Integer.MIN_VALUE;
      for (IntWritable value : values)
      {
         maxValue = Math.max (maxValue, value.
             get());
      }
      context.write (key, new IntWritable (max
          Value));
   }
}
```

Date Page

Example : Runner to find the Maximum temperature.

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class MaxTemp
{ public static void main (String [] args) throws
    Exception
  { if (args.length!=2)
    { System.err.println ("Usage : MaxTemperature
        <inputpath> <output path >");
      System.exit(-1);
    }
    Job job = new Job();
    job.setJarByClass (MaxTemp.class);
    job.setJobName ("Max Temperature");
    FileInputFormat.AddInputPath (job, new Path(args[0]));
    FileOutputFormat.addOutputPath (job, new Path(args[1]));
    job.setMapperClass (MaxTempMapper.class);
    job.setReducerClass (MaxTempReducer.class);
    job.setOutputKeyClass (Text.class);
    job.setOutputValueClass (IntWritable.class);
    System.exit (job.waitForCompletion (true) ? 0:1);
  }
}
```

Date _____ Page _____

⇒ Scaling Out:

However, to scale out, we need to store the data in a distributed file system, typically HDFS, to allow Hadoop to move the MapReduce computation to each machine hosting a part of data.

⇒ Data flow with a single reduce task.

→ Reduce tasks don't have advantage of data locality, the input to single reduce task is normally the output from all mappers.
→ The output of the reduce is normally stored in HDFS for reliability.
→ For each HDFS block of reduce output, the first replica is stored on local node, with other replicas being stored on off-rack nodes.
→ The dotted boxes indicates nodes, the light arrow shows data transfers on a node & heavy arrow shows data transfers b/w nodes.

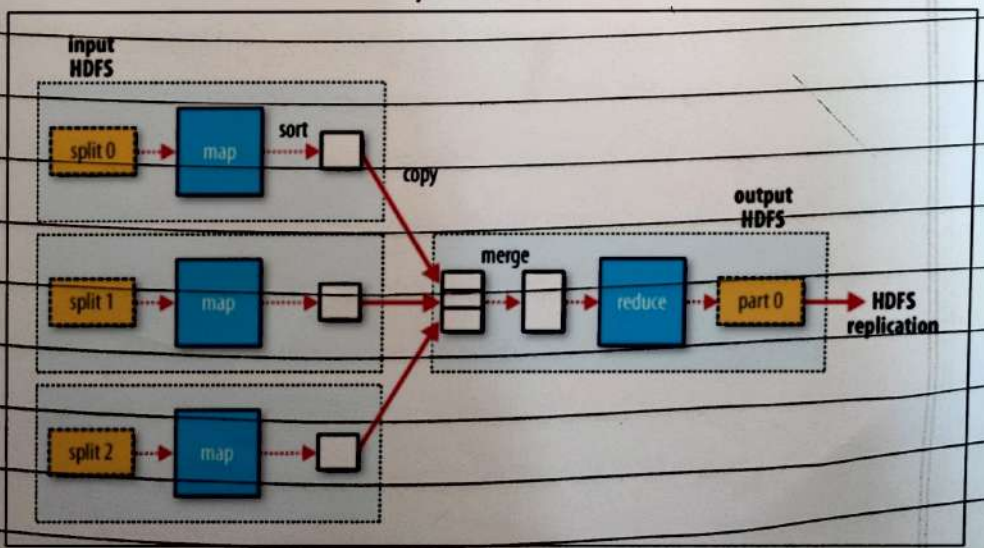


Figure 2-3. MapReduce data flow with a single reduce task

Date Page

⇒ Data flow with multiple reduce tasks.

→ When there are multiple reducers, the map tasks partition their output, each creating one partition for each reduce task.

→ There can be many keys (& their values) in each partition, but records for any given key are all in single partition.

→ The partitioning can be controlled by a user defined partitioning function.

→ The data flow for general case of multiple reduce task is given in the diagram below.

→ The data flow between map & reduce is colloquially known as "the shuffle", as each reduce task is fed by many map tasks.

→ Finally, it's also possible to have zero reduce tasks also.

→ The zero reduce tasks, can be appropriate when we don't need the shuffle since the processing can be carried out entirely in parallel.

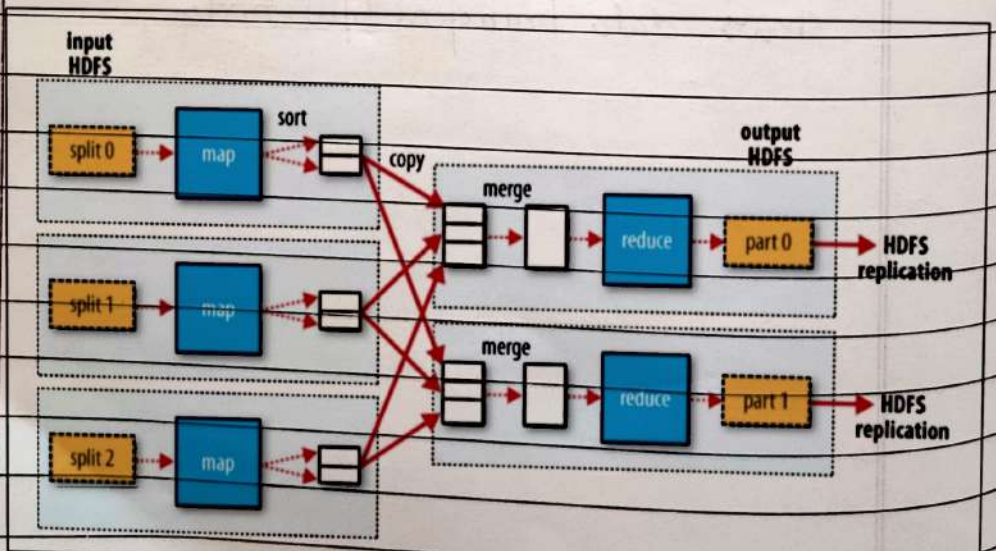


Figure 2-4. MapReduce data flow with multiple reduce tasks

Date Page

⇒ Combiner functions:-

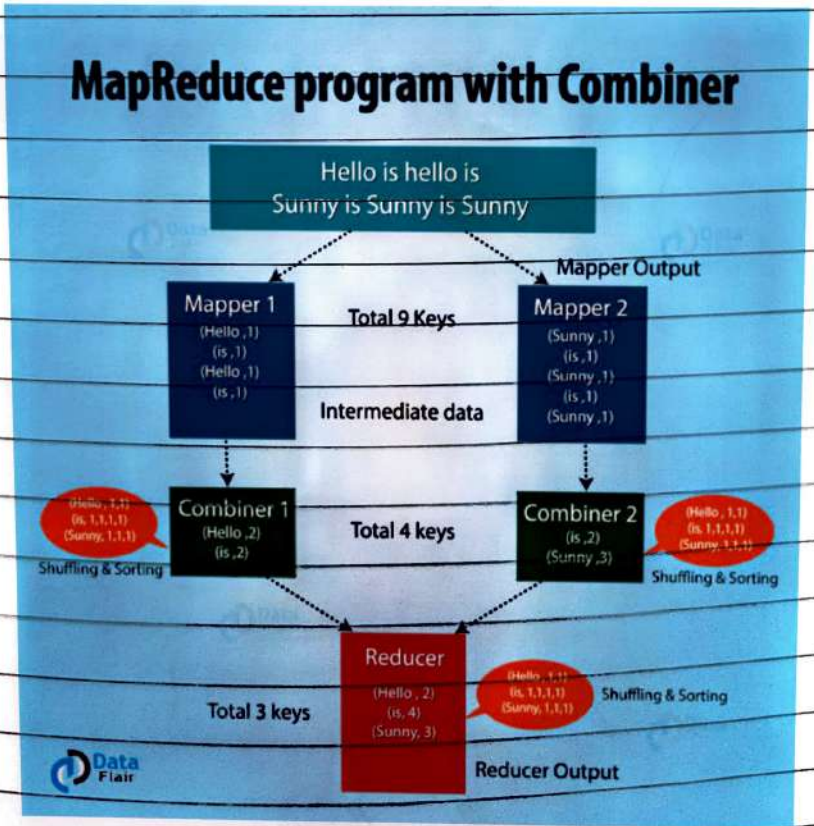
→ Many MapReduce jobs are limited by the bandwidth available on cluster, so it pays to minimize data transferred b/w map & reduce tasks.

→ The combiner function's output form the input to reduce function. Since it is an optimization function.

→ Hadoop allows the user to specify a "combiner function" to be run on the map output.

→ In other words, calling the combiner function zero, one or many times should produce same output from the reducers.

MapReduce data flow with combiner function.



How MapReduce Works

Date _____ Page _____

⇒ Anatomy of a MapReduce Job Run

1) Classic MapReduce (MapReduce 1)

⟶ A job run in classic MapReduce has four independent entities:

a) The client, which submits the MapReduce job

b) The jobtracker, which co-ordinates job run

c) The task trackers, which runs tasks that the job has been split into.

d) The distributed filesystem (HDFS), which is used for sharing job files b/w other entities

MapReduce job using the classic framework.

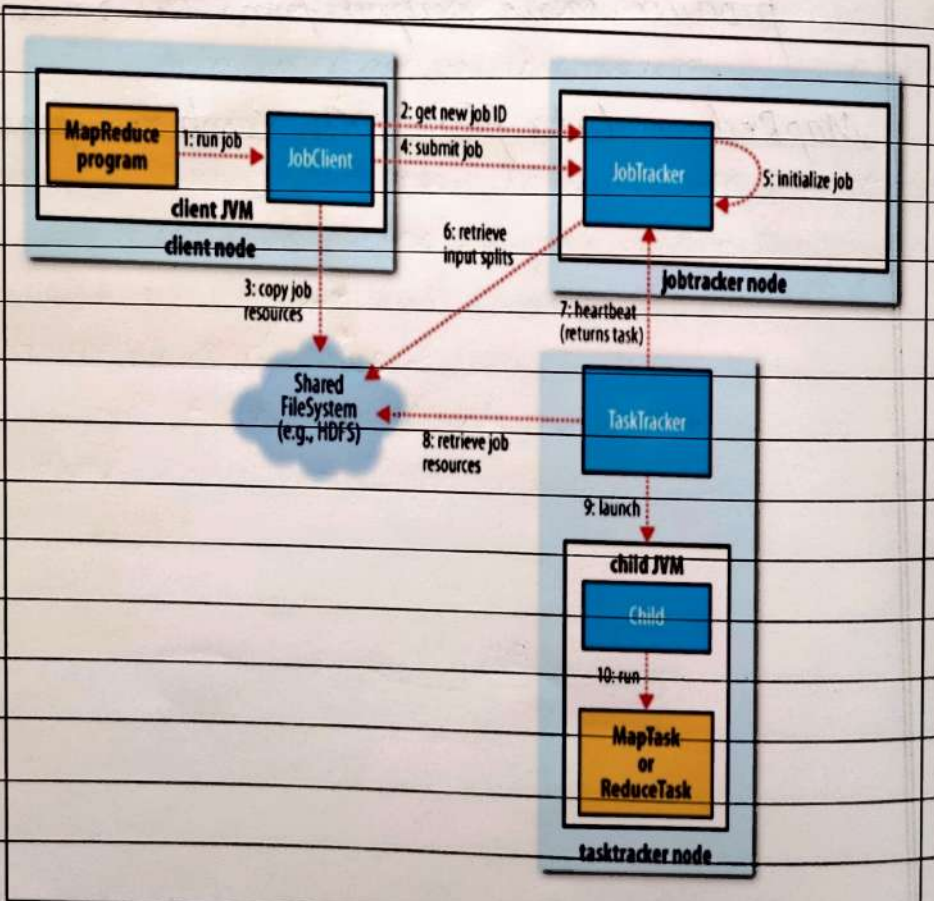


Figure 6-1. How Hadoop runs a MapReduce job using the classic framework

Date Page

- **Job submission :-**
  → The submit() method on Job creates an internal JobSummitter instance and calls submit JobInternal() on it.
  → Having submitted job, waitForCompletion() polls the job's progress and reports to console.
  → Job submission process implemented by JobSummitter includes the following:
  *) Asks jobtracker for new job ID
  *) Checks output specification of the job
  *) Computes the input splits for the job
  *) Copies the resources need to run the job.
  *) Tells job tracker that job is ready for execution.

- **Job Initialization :-**
  → When Job tracker recieves a call to its submitJob() method, it puts it into an internal queue from where job scheduler will pick it up & initialize it.
  → Initialization involves creating objects to represent the job being run, which encapsulates its tasks and keeps tracks of tasks.

- **Task Assignment :-**
  → Task trackers run a simple loop that periodically sends heartbeats method calls to job tracker.
  → If the task tracker is ready to run new task, the jobtracker will allocate a task, which it communicates using the heartbeat return value.

Date _____ Page _____

- **Task Execution :-**
  - → First, it localizes the job JAR by copying it from shared filesystem to tasktracker's filesystem.
  - → Second, it creates a local working directory for the task.
  - → Third, it creates a instance of "TaskRunner" to run the task.
  - → TaskRunner launches a new Java Virtual Machine to run each task.//

- **Progress and Status Updates :-**
  - → A job and each of its tasks have a "status", which includes such things as the state of job or task, the progress of maps & reduces, values of job counters, some message or description.
  - → When a task is running, it keeps track of it's "progress", that is, the proportion of task completed.

- **Streaming and pipes :-**
  - → Both Streaming & pipes run special map & reduce tasks for launching userdefined executables

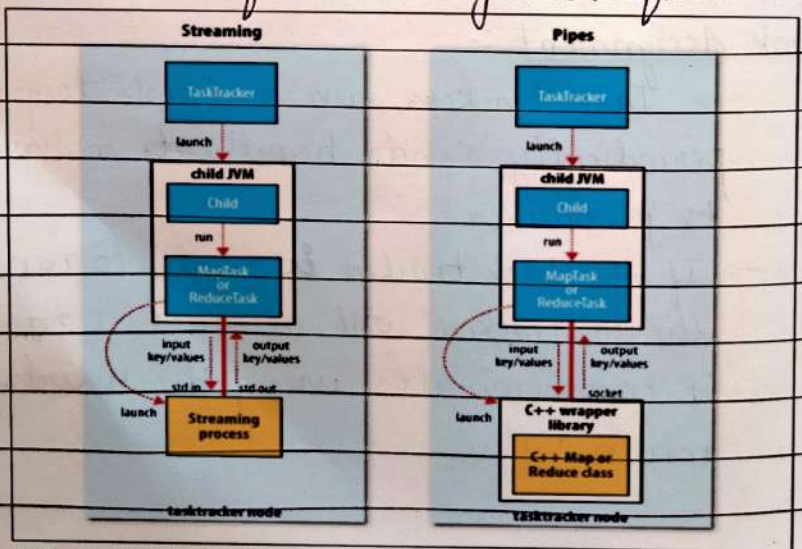


Figure 6-2. The relationship of the Streaming and Pipes executable to the tasktracker and its child

Date Page

- **Job completion :-**
  → When the job tracker recieves a notification that the last task for a job is completed, it changes status for job to "successful."
  → Then it returns from waitForCompletion() method and prints a message to user.

2) YARN (MapReduce 2) :-

→ YARN separates these two roles into two independent daemons : a resource manager and a application manager/master.
→ MapReduce on YARN involves more entities than classic MapReduce.

*) The client, submits the MapReduce job.
*) YARN resource manager, co-ordinates the allocation of compute resources on cluster
*) YARN node managers, launch and monitor the resources & containers on machines in cluster.
*) The application master, co-ordinates the tasks running the MapReduce job.
*) The distributed filesystem (HDFS), to share files b/w other entities.

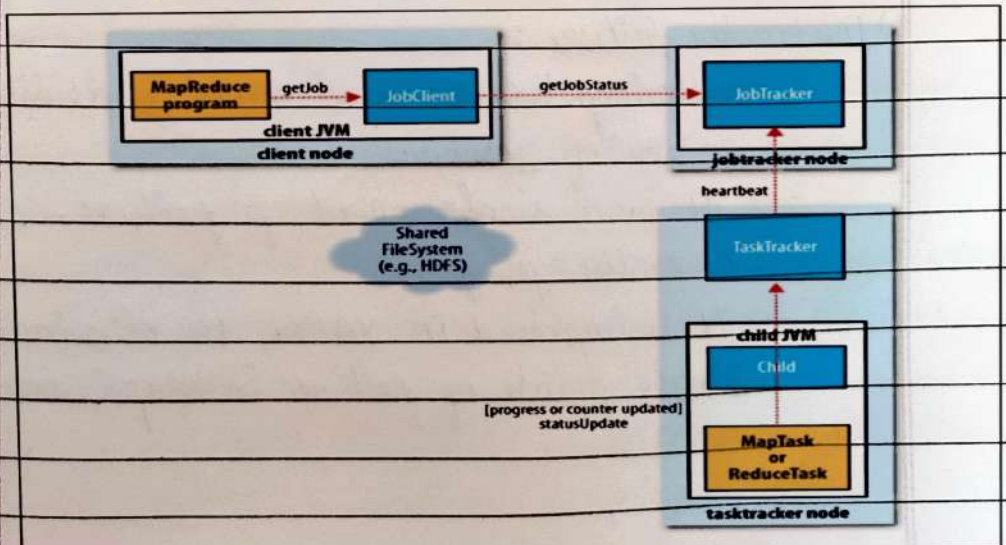


Figure 6-3. How status updates are propagated through the MapReduce 1 system

Date Page

⇒ Failures in classic MapReduce

1) Task failure :-
   → It happens when user code in map or reduce task throws runtime exception.
   → Task tracker marks as task failed.
   → Another mode of failure is sudden exit of child JVM due to some JVM bug that causes JVM to exit.
   → If streaming process exit with non zero code, it is marked as task failed.

2) Task tracker failure.
   → Tasktrack fails by crashing or running very slowly, it will stop sending heartbeats to jobtracker
   → A tasktracker can be blacklisted by jobtracker even if tasktracker has not failed.
   → Blacklisted tasktrackers are not assigned task by they communicate with jobtracker.
   → It can removed from blacklist as faults expire over time & rejoins the cluster.

3) Jobtracker failure :-
   → Hadoop has no mechanism for dealing with failure of jobtracker.
   → It is a single point of failure, so in this case the job fails.
   → It is improved in YARN, to eliminate the single points of failure in MapReduce.

Date Page

⇒ Failures in YARN :-

1) Task Failure :-

→ Failure of running task is similar to classic case.

→ Runtime exceptions & sudden JVM exit are propogated back to application master and marks task as failed.

2) Application master failure :-

→ By default, applications are marked as failed if they fail once.

→ In the event of application master failure, the resource manager will detect failure and start a new instance of master running a new one.

3) Node Manager failure :-

→ If a node manager fails, then it will stop sending heartbeats to resource manager, and node manager will be removed from resource manager's pool of available nodes.

→ Node managers are blacklisted if number of failures for application is high.

4) Resource Manager Failure :-

→ Failure of the resource manager is serious, since without it neither jobs nor tasks can be launched.

→ After a crash, a new resource manager instance is brought up and it recovers from the saved state.

→ The state consists of node managers in the system as well as running applications.

Date Page

⇒ **Job Scheduling:-**

→ MapReduce in Hadoop comes with choice of schedulers.
→ Default in MapReduce 1 (classic) is the original FIFO queue-based scheduler.
→ There are also multi-user schedulers called Fair scheduler and the Capacity Scheduler.
→ MapReduce 2 comes with Capacity Scheduler (default), and the FIFO scheduler.

1) **Fair Scheduler:-**
→ It aims to give every user a fair share of cluster capacity over time.
→ If a single job is running, it gets all of the cluster.
→ Jobs are placed in pools, & by default, each user gets their own pool.
→ Fair scheduler support preemption and is a "contrib" module.

2) **Capacity Scheduler:-**
→ A cluster is made up of number of queues which may be hierarchical and each queue has an allocated capacity.
→ It allows users to simulate a separate MapReduce cluster with FIFO scheduling for each user or organization.
→ This is like Fair Scheduler, except that within each queue, jobs are scheduled using FIFO scheduling.

Date Page

⇒ Shuffle and Sort

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

→ The process by which system performs the sort-and transfers the map outputs to the reducers as inputs – is known as shuffle.

- **The Map Side :-**
  - → When the map function starts producing output, it is not simply written to disk.
  - → Each map task has circular memory buffer that it writes the output.
  - → Before it writes to disk, the thread first divides the data into partitions corresponding to the reducers.
  - → The thread performs in-memory sort by keys and if there is combiner function, it is run on output of the sort.
  - → Running the combiner function makes for a more compact map output, so there is less data to write to local disk and to transfer to reducer.

- **The Reducer side :-**
  - → The map tasks may finish at different times, so the reduce task starts copying their outputs.
  - → This is known as copy phase of reduce task.
  - → A thread in the reducer periodically asks master for map output hosts, until it has retrieved them all.

→ When all the map outputs have been copied, the reduce task moves into the sort phase ("merge phase"; as the sorting was carried out on map side).
→ It merges the outputs, maintaining the sort ordering.
→ During reduce phase, the reduce function is invoked for each key in the sorted output.
→ The output of this phase is written directly to output of filesystem (HDFS).
→ In case of HDFS, because the node manager is also running a datanode, the first block replica will be written to the local disk.

Figure : Shuffle and sort in MapReduce.

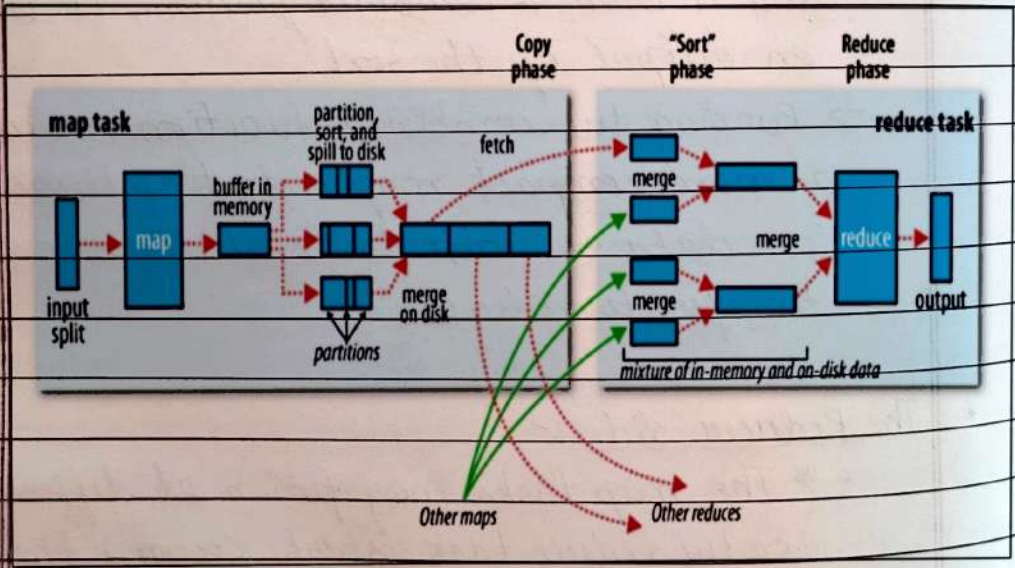


Figure 6-6. Shuffle and sort in MapReduce