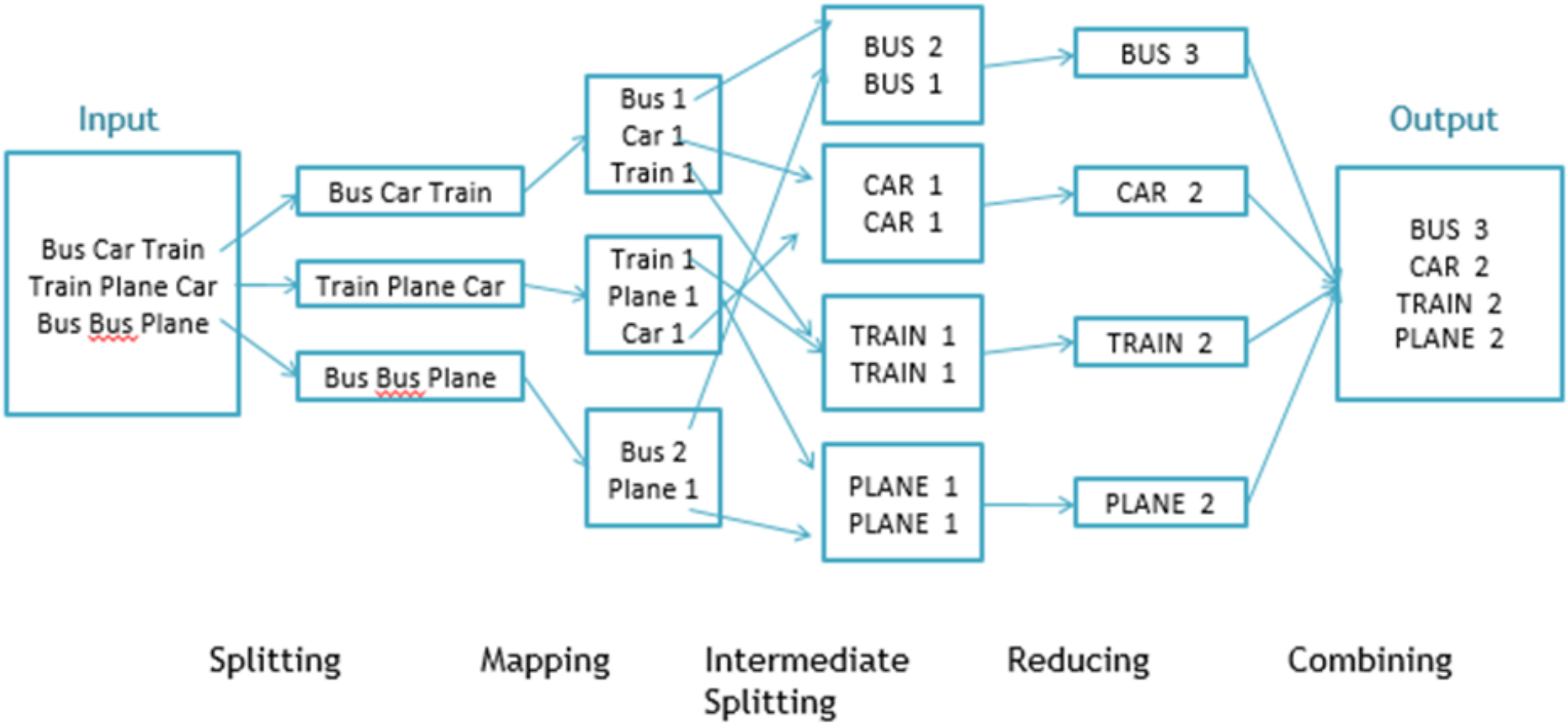
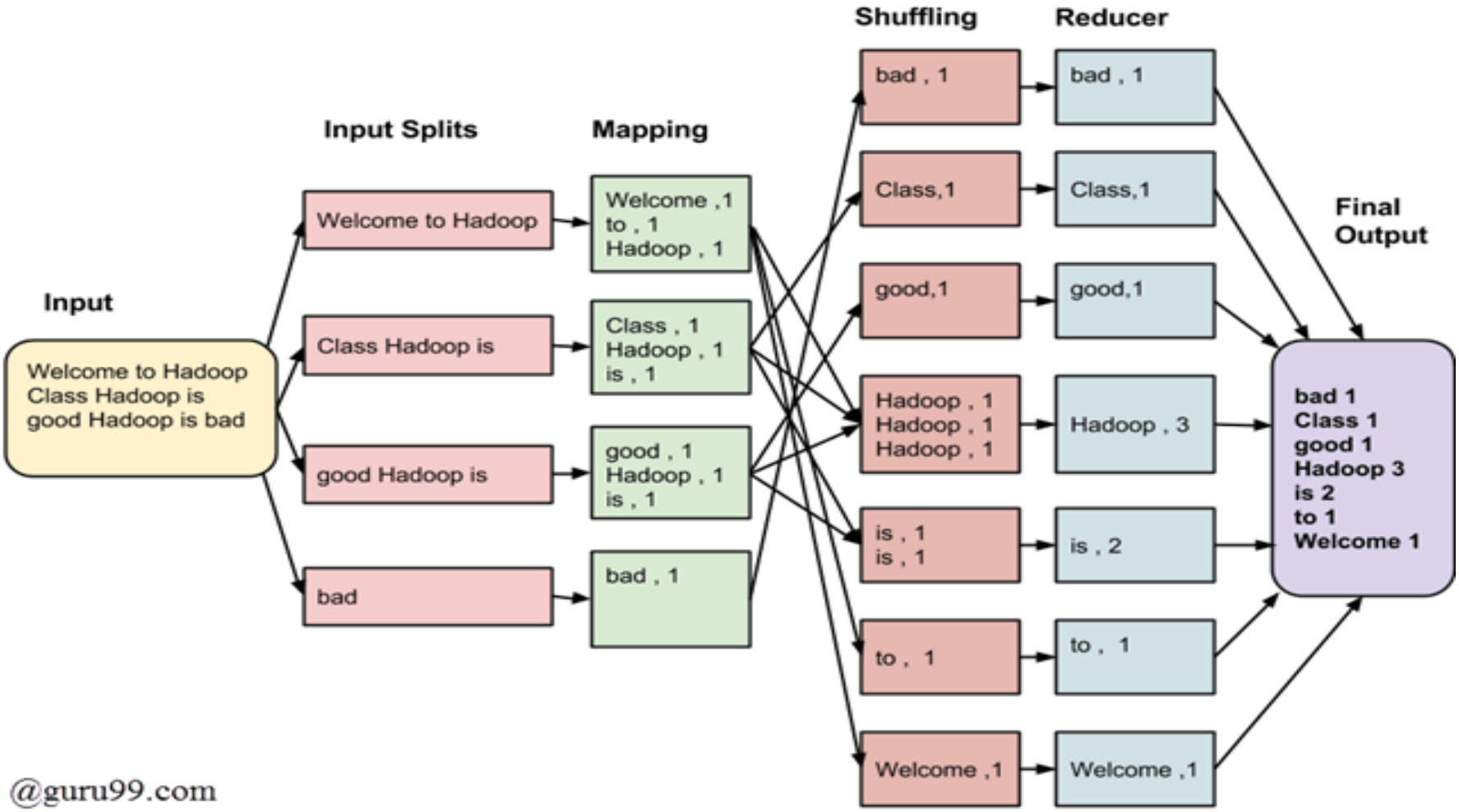
Big Data Analytics – Module 3 (MapReduce)

MapReduce Workflow



A famous example of WordCount



What is MapReduce?

- MR is a programming model for data processing
- Hadoop can run MapReduce programs written in various languages
- MapReduce programs are inherently parallel

A Weather Dataset

- Let's see a program that mines weather data
- It is semi-structured and record-oriented

A Weather Dataset

Data Format

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

Map and Reduce for analysis

- 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

Map and Reduce

- The input to our map phase is the raw NCDC data
- Our map function is simple. We pull out the year and the air temperature because these are the only fields we are interested in
- 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

Map and Reduce

- To visualize the way the map works, consider the following sample lines of input data

0067011990999991950051507004...9999999N9+00001+99999999999...
0043011990999991950051512004...9999999N9+00221+99999999999...
0043011990999991950051518004...9999999N9-00111+99999999999...
0043012650999991949032412004...0500001N9+01111+99999999999...
0043012650999991949032418004...0500001N9+00781+99999999999...

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

(0, 0067011990999991950051507004...9999999N9+00001+99999999999...)

(106, 0043011990999991950051512004...9999999N9+00221+99999999999...)

(212, 0043011990999991950051518004...9999999N9-00111+99999999999...)

(318, 0043012650999991949032412004...0500001N9+01111+99999999999...)

(424, 0043012650999991949032418004...0500001N9+00781+99999999999...)

Map and Reduce

- 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, and emits them as its output

  (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

Map and Reduce

- So, continuing the example, our reduce function sees the following input:

(1949, [111, 78])

(1950, [0, 22, −11])

- All the reduce function has to do now is iterate through the list and pick up the maximum reading:

(1949, 111)

(1950, 22)

Map and Reduce

- The whole data flow is illustrated in this Figure

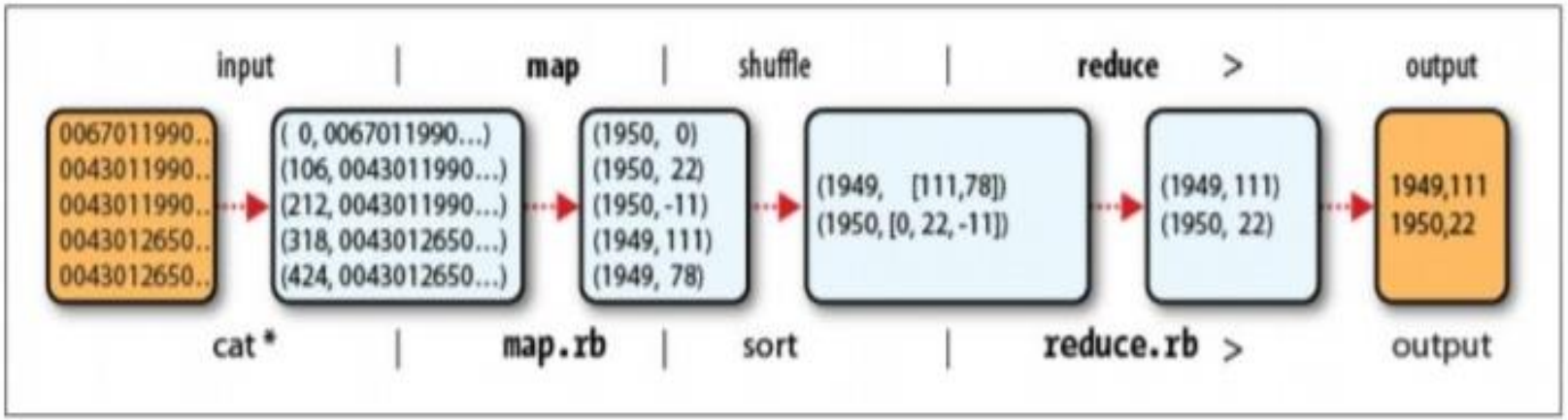


Figure 2-1. MapReduce logical data flow

Java MapReduce

- Having run through how the MapReduce program works, the next step is to express it in code.
- We need three things:
  - a map function
  - a reduce function
  - some code to run the job

(0, 006701199099999**1950**051507004...9999999N9+**0000**1+99999999999...)
(106, 004301199099999**1950**051512004...9999999N9+**0022**1+99999999999...)
(212, 004301199099999**1950**051518004...9999999N9-**0011**1+99999999999...)
(318, 004301265099999**1949**032412004...0500001N9+**0111**1+99999999999...)
(424, 004301265099999**1949**032418004...0500001N9+**0078**1+99999999999...)

Mapper class

(1950, 0)
(1950, 22)
(1950, −11)
(1949, 111)
(1949, 78)

```
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));
    }
  }
}
```

(1949, [111, 78])
(1950, [0, 22, −11])

Reducer Class

```
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));
  }
}
```

(1949, [111, 78])
(1950, [0, 22, -11])

(1949, 111)
(1950, 22)

MapReduce Job

```
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);
   }
}
```

Scaling Out

To allow Hadoop to move the MapReduce computation to each machine hosting a part of the data, the process is as follows:

- 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.

Scaling Out

- 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 user- defined 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 when 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.

Scaling Out

- On the other hand, if splits are too small, 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, 128 MB by default, although this can be changed for the cluster (for all newly created files) or specified when each file is created.

Scaling Out

- 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 because 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.

Scaling Out

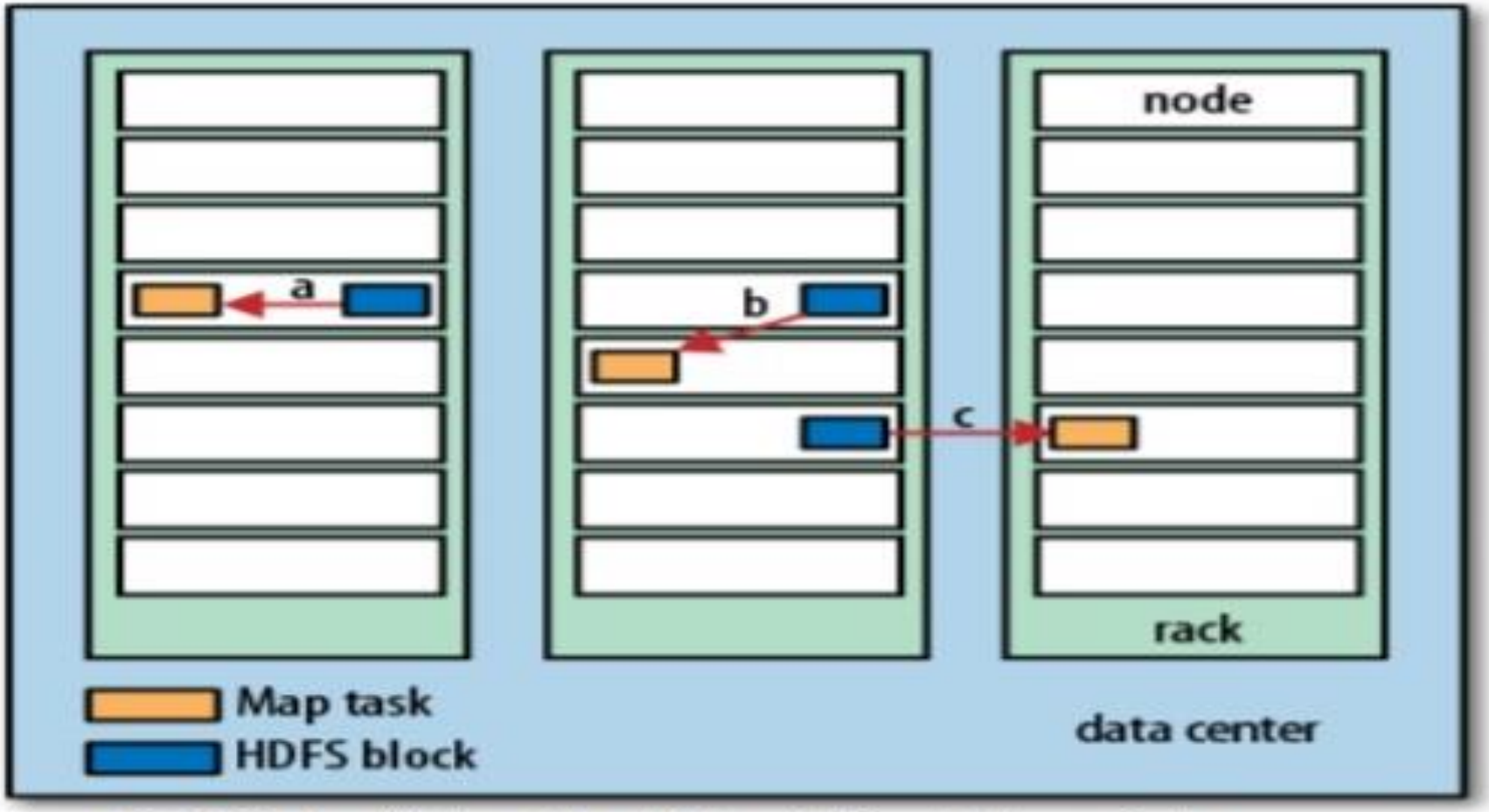


Figure 2-2. Data-local (a), rack-local (b), and off-rack (c) map tasks

Scaling Out

- 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.

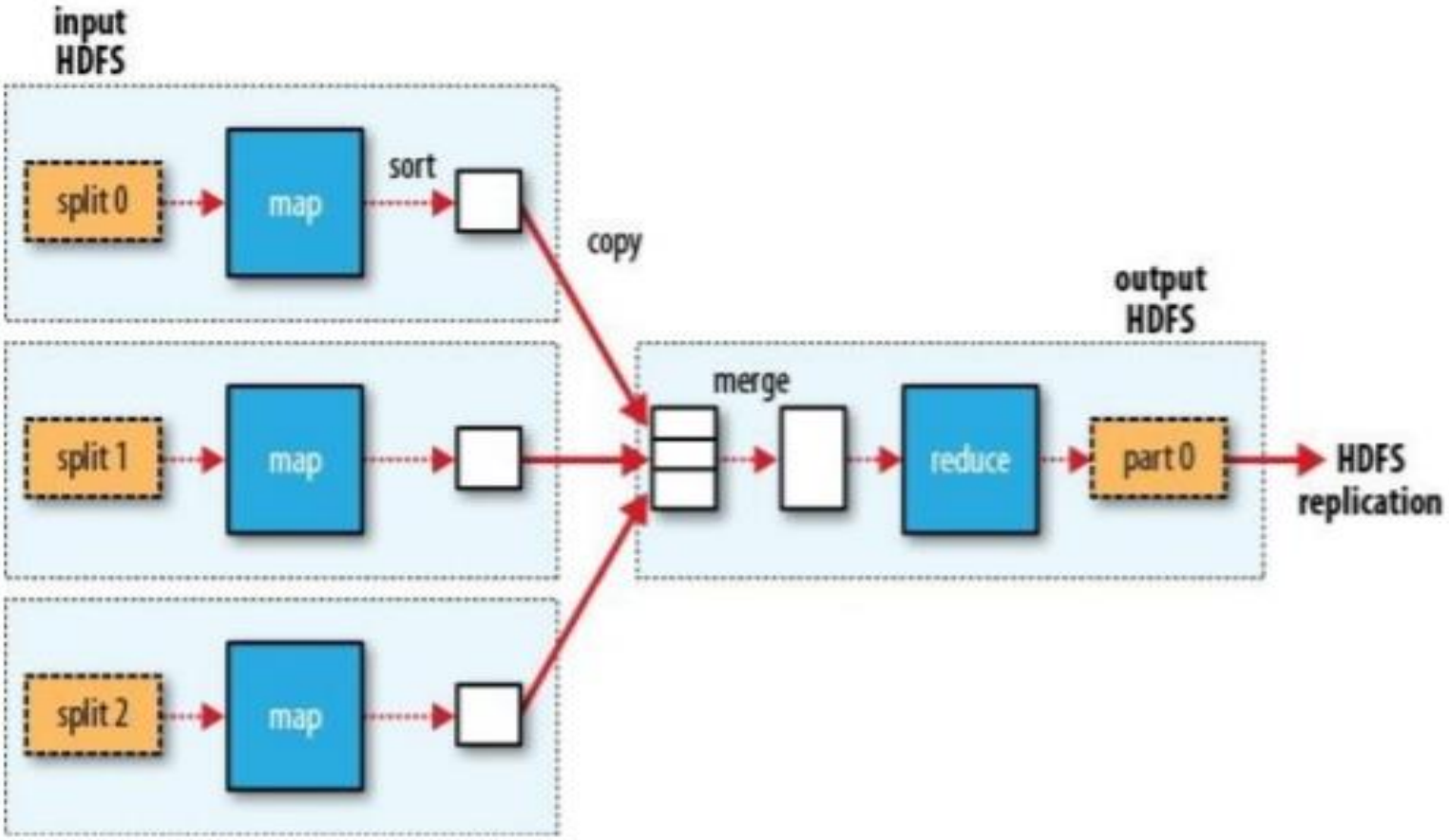
Scaling Out

- Map tasks write their output to the local disk, not to HDFS because 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.

Scaling Out

- 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.
- 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.

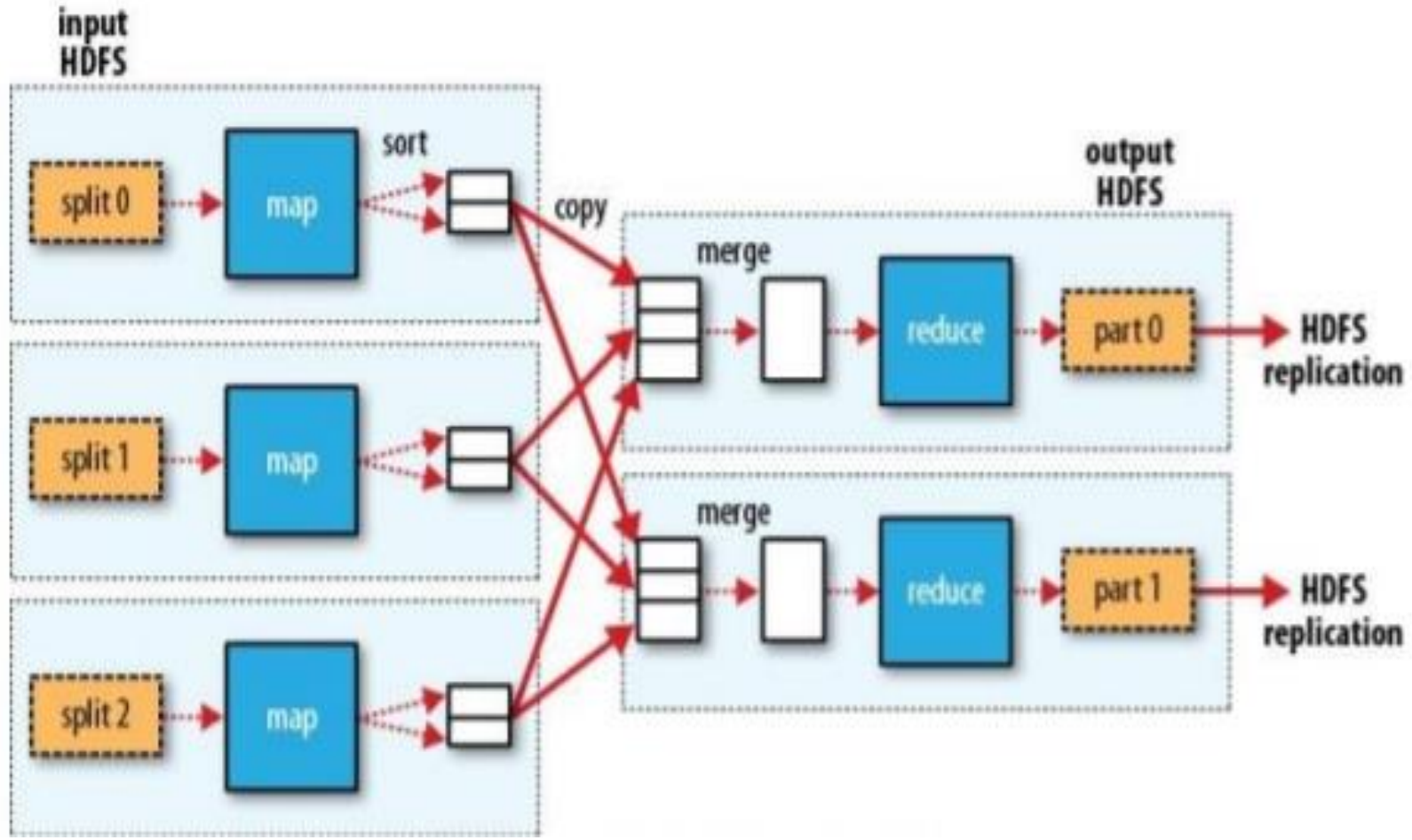
MR data flow with a single reduce task



Scaling Out

- The number of reduce tasks is not governed by the size of the input, but instead is specified independently.
- 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

Data flow for multiple reduce tasks



Scaling Out

- The 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 the diagram suggests, and tuning it can have a big impact on job execution time.
- Finally, it’s also possible to have zero reduce tasks.
- This can be appropriate when you don’t need the shuffle because 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.

MR data flow with no reduce tasks