MapReduce

Hadoop: The Definitive Guide
4th edition
Tom White
O'Reilly

Chapter 3

Overview

- Mappers
- Sort and Shuffle
  - Data sorted and partitioned
  - Partitioned by keys
- Reducers

- Each reducer

  has a single output stream (usually)

  default – set of files *part-r-00000, part-r-00001 …*
- Output of mapper and reducer written to disk
- MapReduce (MR) writes to disk frequently – fault tolerance

  slow

  spark – in-memory – no disk I/O

Map Phase

- MR jobs access data through InputFormat class
- 2 methods in this class
  - getSplits()

    implements logic of how input will be distributed between map processes

    commonly used input format – TextInputFormat class
    - generates an input split per block
    - Gives the location of the block to the map task
    - 1 mapper for each of the splits

- getReader()

  provides a reader to the map task that allows it to access the data it will process

- RecordReader

  reads data blocks and returns key-value records to map task

  many file formats

  - Text delimited
  - SequenceFile
  - Avro
  - Parquet
  - …

- Mapper.set()

  before map method of map task gets called, mapper's setup() method is called once

  every component in Hadoop is configured – Configuration object

  Configuration object
  - See job.xml
  - Information regarding the cluster that the JVM needs
    - URI of NameNode
    - Process coordinating the job – Application Manager

- Mapper.map

  can use defaults for other methods, but map() method must be implemented

  3 inputs

  1. Key
  2. Value
  3. Context

  key and value provided by RecordReader

  - Contains data that map() method should possess

  context - an object that provides common actions for a mapper

  - Sending output to reducer
  - Reading values from Configuration object
  - Incrementing counters to report on progress of map task

when map task writes to reducer,

- the data it is writing is buffered and sorted.
  - sorted in memory, with the available space defined by the i.o. sort.mb configuration parameter.
  - If memory buffer is too small - data is spilled to the local disk of the node where the map task is running and sorted on disk.

- Partitioner

  implements logic of how data is partitioned between reducers.
  - default hash the key
  - divide by number of reducers.
  - equal distribution of data between the reducers – performance
  - If you want to keep certain values together for processing by reducers - override default and implement a custom partitioner.

Example - secondary sort

time series - stock market pricing information

each reducer will scan all the trades of a given stock ticker ordered by the time of the trade in order to look for correlations in pricing over time.

key - ticker-time

default partitioner - sends records belonging to the same stock ticker to different reducers

- implement your own partitioner to make sure the ticker symbol is used for partitioning the records to reducers, but the time stamp is not used.

implementing this type of partitioner

```
public static class CustomPartitioner extends Partitioner<Text, Text> {
   @Override
   public int getPartition(Text key, Text value, int numPartitions){
      String ticker = key.toString().substring(S);
      return ticker.hashCode() % numPartitions;
   }
}
```

extracts the ticker symbol out of the key

use the hash of this part for partitioning instead of entire key

- Mapper.cleanup()

  called after the map() method has executed for all records

- Combiner.

  to reduce amount of network traffic between mappers and reducers.

  word count example.

  - mapper takes each input line, splits it into individual words and writes out each word with "1" after it, to indicate current count:

    the=> 1

    cat=> 1

    and=> 1

    the=> 1

    hat=> 1

- combine() method – to aggregate values produced by mapper

  executes locally on the same node where mapper executes

  reduces output that is later sent through the network to reducer

  reducer will therefore aggregate results from different mappers, but over significantly smaller data sets

  output of combiner has to be identical in format to output of mapper

  combiner executes after output of mapper is already sorted
  - therefore input of combiner is sorted

Our example, output of combiner:

- and=> 1
- cat=> 1
- hat > 1
- the=> 2

Reducer

- Shuffle

  reduce tasks copy output of mappers from map nodes to reduce nodes.
- Reducer.setup()

  method executes before reducer starts processing individual records

  typically used to initialize variables and file handles

- Reducer.reduce()

  reduce() method is where the reducer does most of the data processing.

  differences in inputs:
  - keys are sorted.
- value parameter changed to values
  - for one key the input will be all the values for that key
  - a key and all its values will never be split across more than one reducer
- map() method calling context.write(K, V) stores output in a buffer that is later sorted and read by reducer.
- reduce() method calling context.write(KP1, V) sends output to the outputFileFormat

- Reducer.cleanup()

  called after all records are processed.
  - can close files and log overall status ·
- Outputformat

  responsible for formatting and writing output data, typically to HDFS

  large file - InputFormat splits input to multiple map tasks
  - each handling a small subset of a single input file.

  OutputFormat class - a single reducer writes a single file
  - therefore on HDFS - one file per reducer.
  - files will be named something like part-r-00000 with the numbers ascending as the task numbers of reducers ascend.

Dataset

- National Climatic Data Center, or NCDC dataset
- Line-oriented ASCII format - each line is a record.
- Temperature - fixed width.

- Sample line
  - Line has been split into multiple lines to show each field

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

- Datafiles - organized by date and weather station.
- Directory for each year from 1901 to 2001
  - each containing a file for each weather station with its readings for that year.
- Tens of thousands of weather stations
  - Whole dataset is made up of a large number of relatively small files.
  - Generally easier and more efficient to process a smaller number of relatively large files

- First entries for 1990:

% **ls raw/1990 | head**

010010-99999-1990.gz
010014-99999-1990.gz
010015-99999-1990.gz
010016-99999-1990.gz
010017-99999-1990.gz
010030-99999-1990.gz
010040-99999-1990.gz
010080-99999-1990.gz
010100-99999-1990.gz
010150-99999-1990.gz

Map and Reduce

- MapReduce breaks the processing into two phases:
  - Map phase
  - Reduce phase.
- Each phase has key-value pairs as input and output
  - Types may be chosen by the programmer.
- Programmer specifies two functions:
  - Map function
  - Reduce function.

- Input to map phase - raw NCDC data.
- Text input format - gives each line in the dataset as a text value.
- Key - offset of the beginning of the line from the beginning of the file
  - No need for this - so we ignore it.
- Map function
  - pull out year and air temperature (fields we are interested in)
  - Map function – data preparation phase
    - setting up the data for the reduce function
    - filter out temperatures that are missing, suspect, or erroneous.
- Reduce function
    - finding the maximum temperature for each year.

- Consider 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+**0000**1+99999999999...)

(106, 0043011990999991950051512004...9999999N9+**0022**1+99999999999...)

(212, 0043011990999991950051518004...9999999N9-**0011**1+99999999999...)

(318, 0043012650999991949032412004...0500001N9+**0111**1+99999999999...)

(424, 0043012650999991949032418004...0500001N9+**0078**1+99999999999...)

- keys - line offsets within the file (we ignore in our map function)
- Map function
  - extracts year and air temperature (bold text),
  - Outputs them (temperature values interpreted as integers)

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

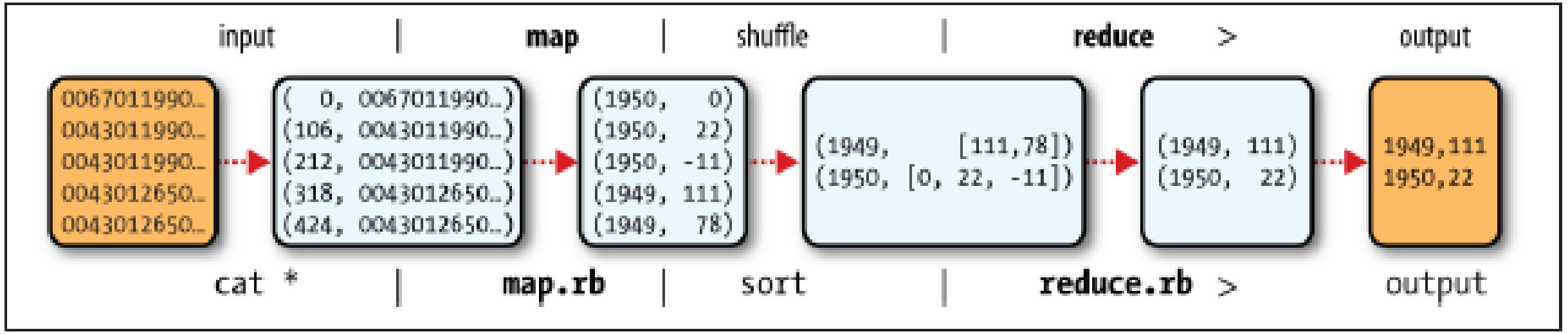
- Output from map function is processed before being sent to reduce function.
  - Sorts and groups the key-value pairs by key.
- Reduce function sees the following input:

  (1949, [111, 78])

  (1950, [0, 22, −11])
  - Each year appears with a list of all its air temperature readings.
- Reduce function iterates through the list and picks the maximum:

  (1949, 111)

  (1950, 22)
- Final output: maximum global temperature recorded each year



MapReduce Dataflow

Code

- Map function
- Reduce function
- Code to run the job.

- Map function is represented by the Mapper class, which declares an abstract `map()` method.

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

- Mapper class - generic type
- 4 formal type parameters
  - input key - long integer offset
  - input value - line of text
  - output key - year,
  - output value types - air temperature (integer)
- Hadoop provides its own set of basic types that are optimized for network serialization
  - Found in the org.apache.hadoop.io package.
  - Example – LongWritable (corresponds to a Java Long), Text (like Java String), and IntWritable (like Java Integer).

- `map()` method passed a key and a value.
- Convert the Text value containing the line of input into a Java String
- Use its substring() method to extract the relevant columns.
- map() method also provides an instance of Context to write the output to.
- Write year as a Text object (using it as a key)
- temperature is wrapped in an IntWritable.
  - Write an output record only if the temperature is present and the quality code indicates the temperature reading is OK.

```
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));
   }
}
```

- 4 formal type parameters specify input and output types,
- Input types of reduce function must match output types of the map function: Text and IntWritable.
- In this case, output types of reduce function are Text and IntWritable,
  - for a year and its maximum temperature

```
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);
  }
}
```

- `Job` object forms the specification of the job and gives control over how the job is run.
- When running this job on a Hadoop cluster, package the code into a JAR file (which Hadoop will distribute around the cluster).
- Rather than explicitly specifying the name of the JAR file, pass a class in the Job's `setJarByClass()` method
- Hadoop will use this to locate the relevant JAR file by looking for the JAR file containing this class.

- Having constructed a Job object, specify the input and output paths.
- Input path is specified by calling the static `addInputPath()` method on `FileInputFormat`. It can be a
  - single file,
  - directory (in which case, the input forms all the files in that directory),
  - a file pattern.
- `addInputPath()` can be called more than once to use input from multiple paths.

- Output path (only one) specified by the static `setOutput Path()` method on `FileOutputFormat.`
- This specifies a directory where the output files from the reduce function are written.
- Directory shouldn’t exist before running the job
- Next, specify map and reduce types to use via the `setMapperClass()` and `setReducerClass()` methods.
- setOutputKeyClass() and setOutputValueClass() methods control output types for reduce function,
  - Must match what the Reduce class produces.

- Map output types default to the same types
  - they do not need to be set if the mapper produces the same types as the reducer
  - If they are different, the map output types must be set using the `SetMapOutputKeyClass()` and `setMapOutputValueClass()` methods.
- Input types controlled via input format (not explicitly set in our case because we are using the default `TextInputFormat)`.
- `waitForCompletion()` method on `Job` submits the job and waits for it to finish.
- Single argument to the method is a flag indicating whether verbose output is generated.

- Return value of `waitForCompletion()` method is a Boolean indicating success (true) or failure (false),
  - translated into the program's exit code of 0 or 1.

Test Run

```
% export HADOOP_CLASSPATH=hadoop-examples.jar
% hadoop MaxTemperature input/ncdc/sample.txt output
```

- When **`hadoop`** command is invoked with a classname as the first argument, it launches a Java virtual machine (JVM) to run the class.
- The hadoop command 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, an environment variable called **`HADOOP_CLASSPATH`** is defined, which the hadoop script picks up.

```
14/09/16 09:48:40 INFO input.FileInputFormat: Total
input paths to process : 1
14/09/16 09:48:40 INFO mapreduce.JobSubmitter: number of
splits:1
14/09/16 09:48:40 INFO mapreduce.JobSubmitter:
Submitting tokens for job:
job_local26392882_0001
14/09/16 09:48:40 INFO mapreduce.Job: The url to track
the job:
http://localhost:8080/
14/09/16 09:48:40 INFO mapreduce.Job: Running job:
job_local26392882_0001
```

…

14/09/16 09:48:40 INFO mapred.LocalJobRunner: **map**

14/09/16 09:48:40 INFO mapred.Task: Task **'attempt_local26392882_0001_m_000000_0'**

done.

…

14/09/16 09:48:40 INFO mapred.LocalJobRunner: Waiting for **reduce** tasks

14/09/16 09:48:40 INFO mapred.LocalJobRunner: Starting task:

attempt_local26392882_0001_r_000000_0

…

```
Map input records=5
Map output records=5
Map output bytes=45
Map output materialized bytes=61
Input split bytes=129
Combine input records=0
Combine output records=0
Reduce input groups=2
Reduce shuffle bytes=61
Reduce input records=5
Reduce output records=2
Spilled Records=10
Shuffled Maps =1
Failed Shuffles=0
Merged Map outputs=1
```

- Output from running the job provides some useful information.
- Example,
  - Job given ID: job_local26392882_0001
  - Job ran one map task (ID: attempt_local26392882_0001_m_000000_0)
  - Job ran one reduce task (ID: attempt_local26392882_0001_r_000000_0).
- Useful when debugging MapReduce jobs.
- Last section of output, - Counters
  - Statistics that Hadoop generates for each job it runs.
  - Can follow the number of records that went through the system: five map input records produced five map output records
    - Mapper emitted one output record for each valid input record)
    - Five reduce input records in two groups (one for each unique key) produced two reduce output records.

- Output written to the *output* directory - contains one output file per reducer.
- Job had a single reducer, therefore single file, *part-r-00000*:

```
% cat output/part-r-00000
1949 111
1950 22
```

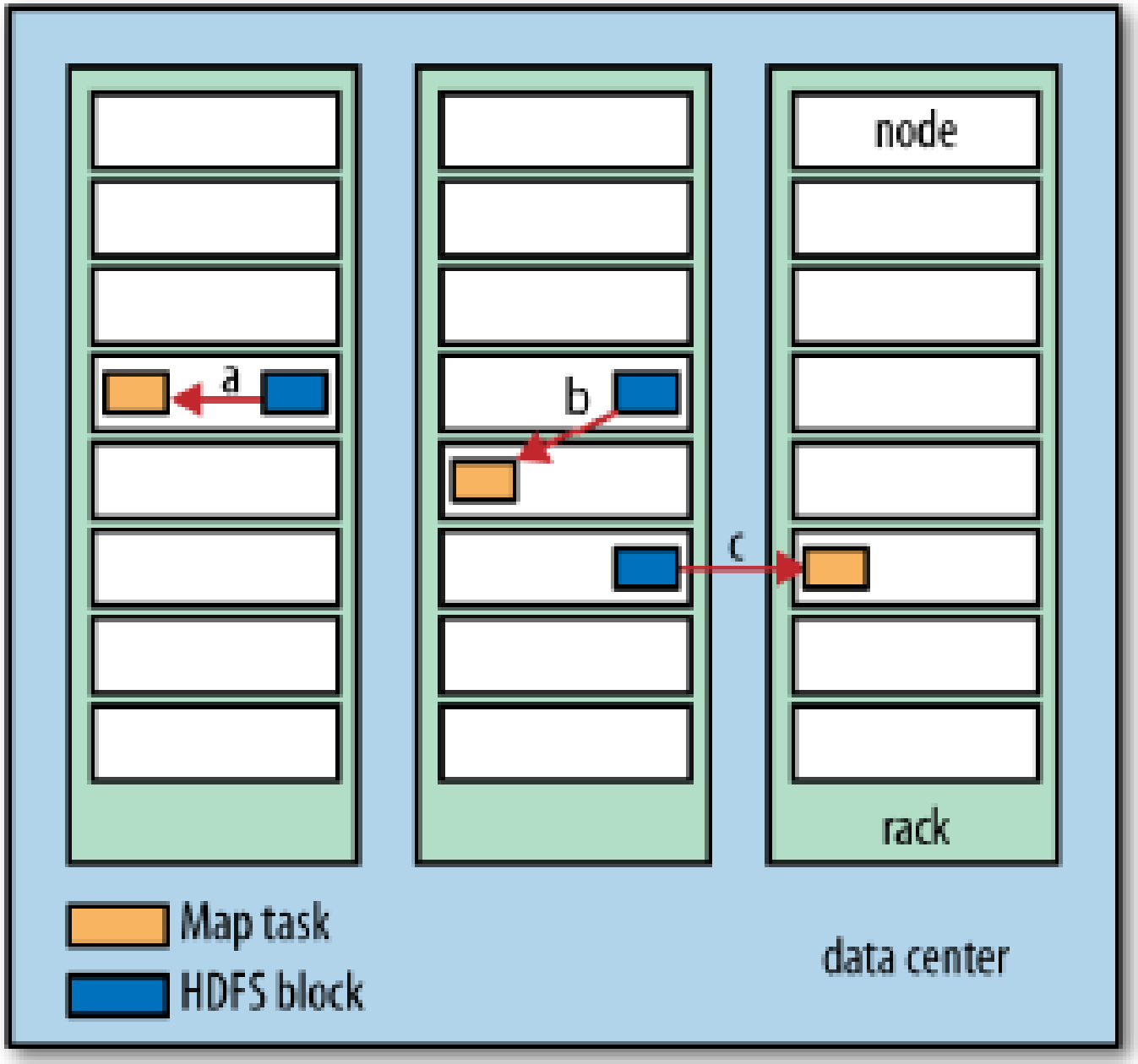
- Maximum temperature recorded in 1949 was 11.1°C, and in 1950 it was 2.2°C.

Data Flow

- A MapReduce *job* is a unit of work that the client wants to be performed
  - input data
  - MapReduce program
  - Configuration
- Hadoop runs the job by dividing it into *tasks*,
- Two types of tasks:
  - *map tasks*
  - *reduce tasks*.
- Tasks are scheduled using YARN and run on nodes in the cluster.
  - If a task fails, it will be automatically rescheduled to run on a different node

- Hadoop divides the input to a MapReduce job into fixed-size pieces called *input splits* (or *splits*)
- Hadoop creates one map task for each split
- User-defined map function run for each *record* in the split.
- For most jobs, a good split size tends to be the size of an HDFS block, which is 128 MB by default
  - Can be changed for the cluster (for all newly created files) or specified when each file is created.

- Hadoop tries to run the map task on a node where the input data resides in HDFS (*data locality optimization)*
  - doesn’t use valuable cluster bandwidth.
- If all nodes hosting HDFS block replicas for a map task’s input split are running other map tasks, job scheduler will look for a free map slot on a node in the same rack as one of the blocks.
- If this is not possible, an off-rack node is used – results in inter-rack network transfer.

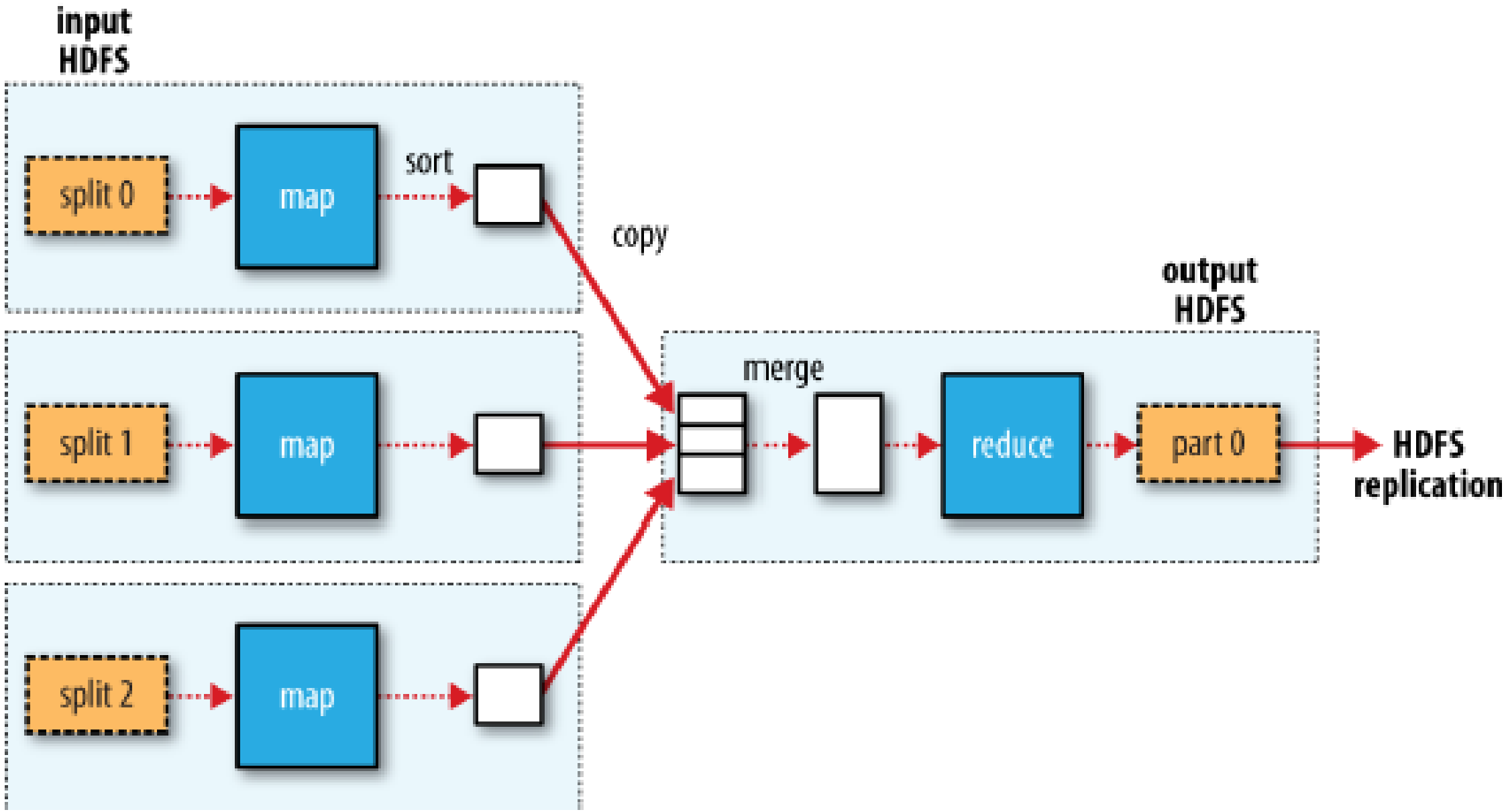


Data-local (a), rack-local (b), and off-rack (c) map tasks

- Map tasks write their output to local disk, not to HDFS.
- If node running map task fails before the map output has been consumed by the reduce task, then Hadoop will automatically rerun map task on another node to re-create the map output

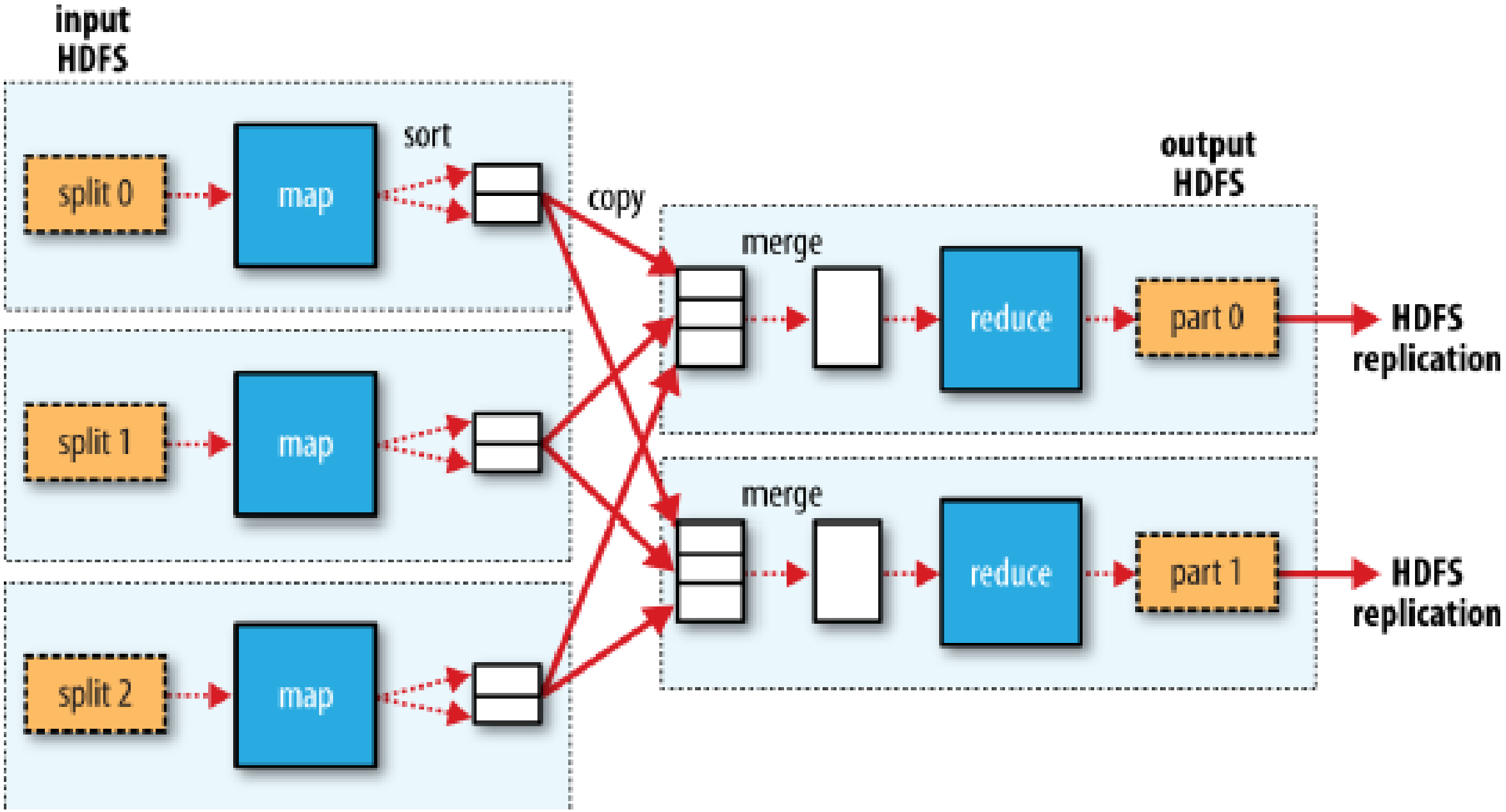
- Input to a single reduce task is the output from all mappers.
- Our example - a single reduce task that is fed by all of the map tasks.
- Sorted map outputs transferred across the network to the node where the reduce task is running
- Outputs merged and passed to the user-defined reduce function.
- 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
- Other replicas stored on off-rack nodes for reliability.

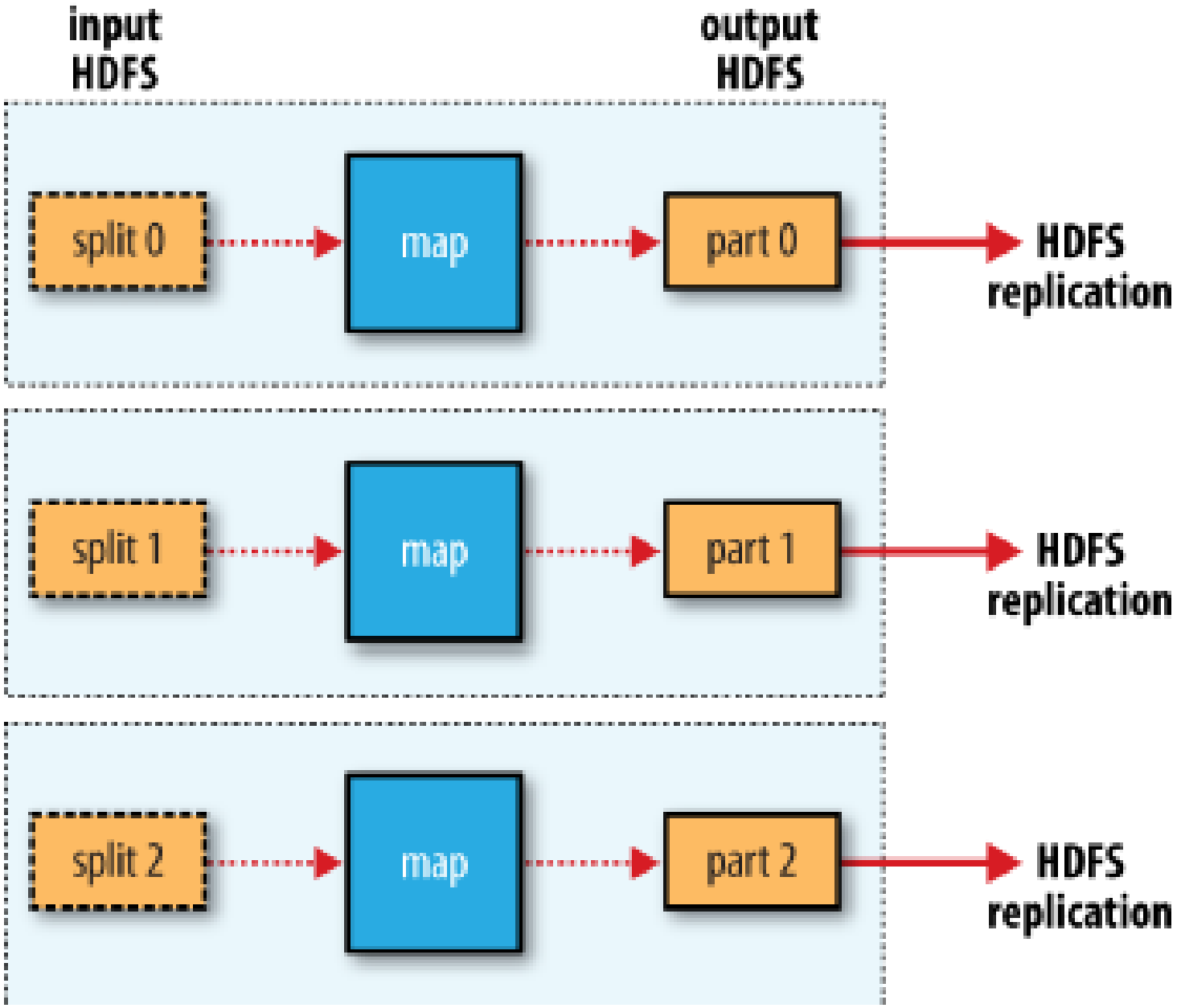
- Dotted boxes - nodes
- Dotted arrows - data transfers on a node
- Solid arrows - data transfers between nodes.


input
HDFS
split 0
map
sort
split 1
map
split 2
map
copy
output
HDFS
merge
reduce
part 0
HDFS
replication

- Multiple reducers - 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
  - records for any given key are all in a single partition.
- Partitioning can be controlled by a user-defined partitioning function
  - Or default partitioner—which buckets keys using a hash function

MapReduce data flow with multiple reduce tasks





MapReduce data flow with no reduce tasks

Combiners

- Bandwidth limitations on cluster
- Minimize the data transferred between map and reduce tasks.
- User may specify a *combiner function* to be run on the map output
- Combiner function's output forms the input to the reduce function.

- Suppose that for the maximum temperature example, readings for the year 1950 were processed by two maps (because they were in different splits).
- First map produced the output:
  (1950, 0)
  (1950, 20)
  (1950, 10)
- 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)

- Combiner function - finds the maximum temperature for each map output.
- Reduce function would then be called with:

  (1950, [20, 25])
- would produce the same output as before.
- Function calls on the temperature values :

  $max(0, 20, 10, 25, 15) = max(max(0, 20, 10), max(25, 15)) = max(20, 25) = 25$

- Calculating mean temperatures,

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

- but:

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

- The combiner function doesn’t replace the reduce function.
- Combiner function can help cut down the amount of data shuffled between the mappers and the reducers

Hadoop Streaming

- Hadoop provides an API to MapReduce that allows you to write your map and reduce functions in languages other than Java.
- *Hadoop Streaming* uses Unix standard streams as the interface between Hadoop and your program, so you can use any language that can read standard input and write to standard output to write your MapReduce program.

Python

Map function for maximum temperature in Python

```
#!/usr/bin/env python
import re
import sys
for line in sys.stdin:
  val = line.strip()
  (year, temp, q) = (val[15:19], val[87:92],
  val[92:93])
  if (temp != "+9999" and re.match("[01459]", q)):
    print "%s\t%s" % (year, temp)
```

Reduce function for maximum temperature in Python

```
#!/usr/bin/env python
import sys
(last_key, max_val) = (None, -sys.maxint)
for line in sys.stdin:
   (key, val) = line.strip().split("\t")
    if last_key and last_key != key:
      print "%s\t%s" % (last_key, max_val)
      (last_key, max_val) = (key, int(val))
   else:
      (last_key, max_val) = (key, max(max_val, int(val)))
if last_key:
   print "%s\t%s" % (last_key, max_val)
```

- Test the programs and run the job
- To run a test:

```
% cat input/ncdc/sample.txt | \
ch02-mr-intro/src/main/python/max_temperature_map.py | \
sort | ch02-mr-
intro/src/main/python/max_temperature_reduce.py
```

1949 111

1950 22