Chapter:3 –Map reduce

Om Prakash Mahato
Assistant Director,
Nepal Telecommunications Authority

What is a Functional Language?

Opinions differ, and it is difficult to give a precise definition, but generally speaking:

- Functional programming is style of programming in which the basic method of computation is the application of functions to arguments.
- A functional language is one that supports and encourages the functional style.

Functional Programming

- The Functional Programming Paradigm is one of the major programming paradigms.
  - FP is a type of declarative programming paradigm
  - Also known as *applicative programming* and *value-oriented programming*
- Idea: everything is a function
- Based on sound theoretical frameworks (e.g., the lambda calculus)
- Examples of FP languages
  - First (and most popular) FP language: Lisp
  - Other important FPs: ML, Haskell, Miranda, Scheme, Logo

Functional Programming Languages

The design of the imperative languages is based directly on the von Neumann architecture[Same memory holds data, instructions]

- Efficiency is the primary concern, rather than the suitability of the language for software development

The design of the functional languages is based on mathematical functions

- A solid theoretical basis that is also closer to the user, but relatively unconcerned with the architecture of the machines on which programs will run

Characteristics of Pure FPLs

Pure FP languages tend to

- Have no side-effects
- Have no assignment statements
- Often have no variables!
- Be built on a small, concise framework
- Have a simple, uniform syntax
- Be implemented via interpreters rather than compilers
- Be mathematically easier to handle

Example

Summing the integers 1 to 10 in Java:

```
int total = 0;
for (int i = 1; i ≤ 10; i++)
    total = total + i;
```

The computation method is variable assignment.

6

Example

Summing the integers 1 to 10 in Haskell:

```
sum [1..10]
```

The computation method is function application.

7

Summary: ILs vs FPLs

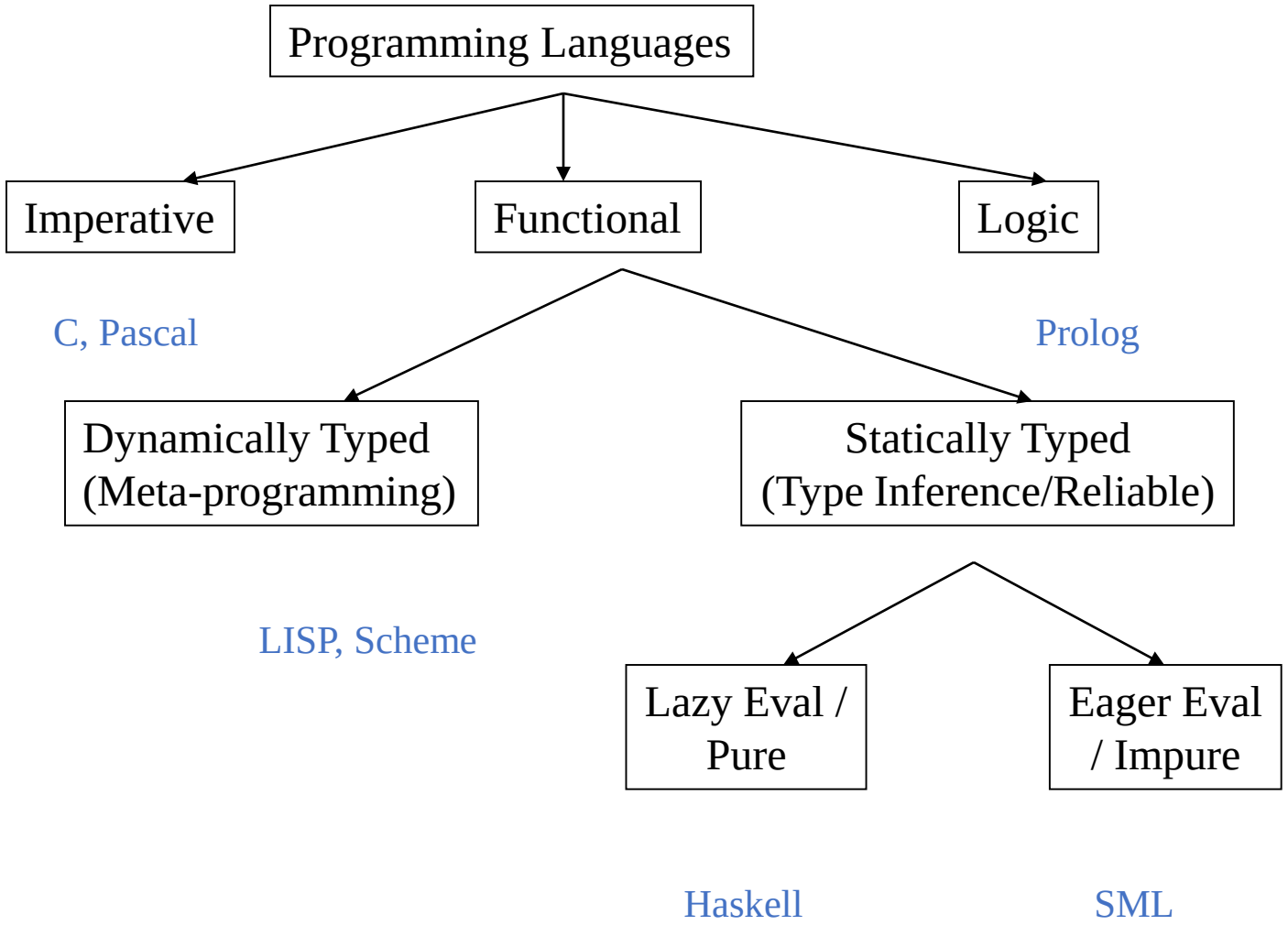
Imperative Languages:

- Efficient execution
- Complex semantics
- Complex syntax
- Concurrency is programmer designed

Functional Languages:

- Simple semantics
- Simple syntax
- Inefficient execution
- Programs can automatically be made concurrent

Summary


Programming Languages
Imperative
Functional
Logic
C, Pascal
Prolog
Dynamically Typed
(Meta-programming)
Statically Typed
(Type Inference/Reliable)
LISP, Scheme
Lazy Eval /
Pure
Eager Eval
/ Impure
Haskell
SML

What is MapReduce?

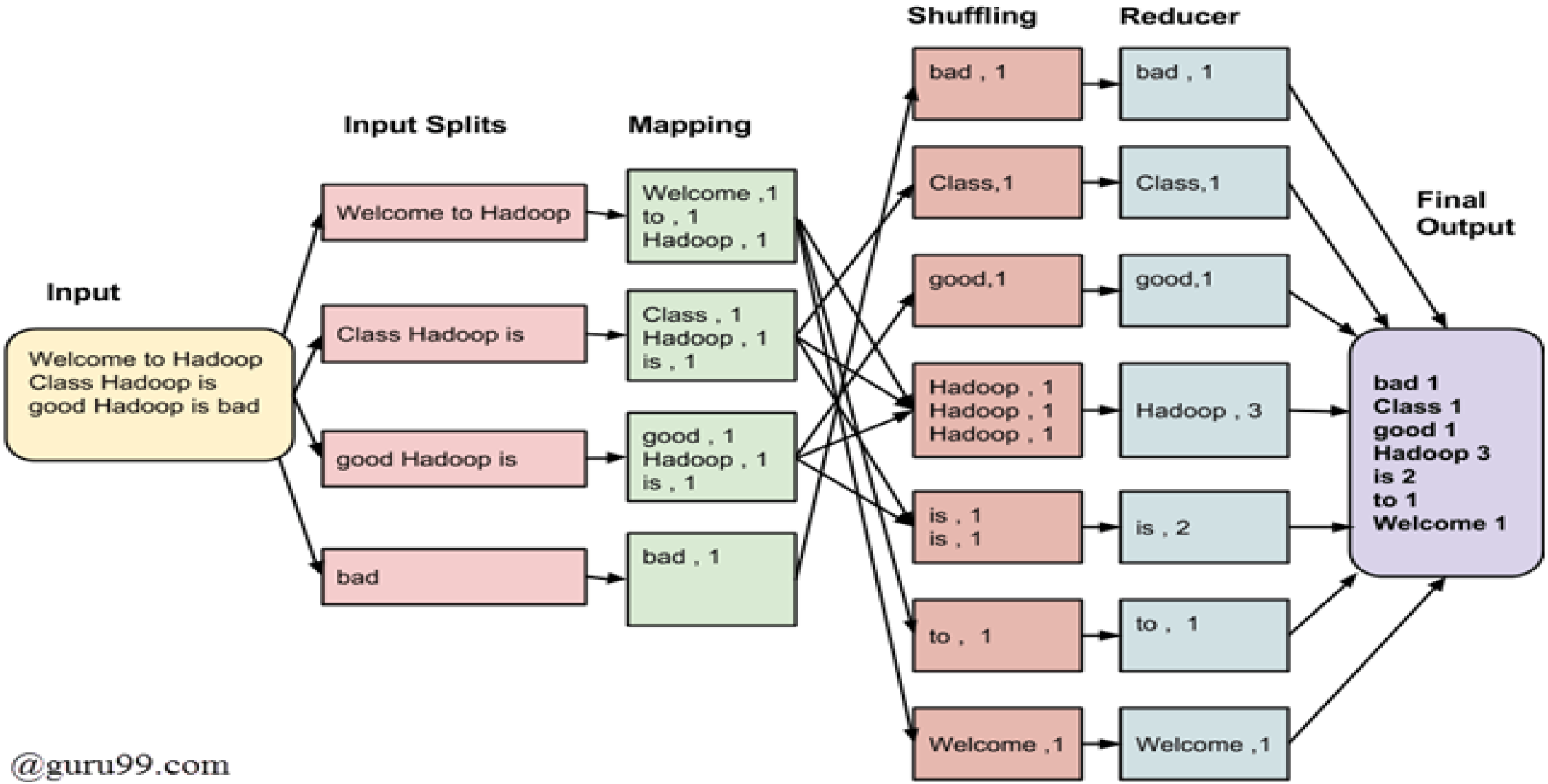
- **MAPREDUCE** is a software framework and programming model used for processing huge amounts of data. **MapReduce** program work in two phases, namely, Map and Reduce. Map tasks deal with splitting and mapping of data while Reduce tasks shuffle and reduce the data.
- Hadoop is capable of running MapReduce programs written in various languages: Java, Ruby, Python, and C++. MapReduce programs are parallel in nature, thus are very useful for performing large-scale data analysis using multiple machines in the cluster.
- The input to each phase is **key-value** pairs. In addition, every programmer needs to specify two functions: **map function** and **reduce function**.

How MapReduce Works?

- The whole process goes through four phases of execution namely, splitting, mapping, shuffling, and reducing.
- Let's understand this with an example –
- Consider you have following input data for your Map Reduce Program

Welcome to Hadoop Class
Hadoop is good
Hadoop is bad

How MapReduce Works?



How MapReduce Works? Conti..

The final output of the MapReduce task is

bad	1
Class	1
good	1
Hadoop	3
is	2
to	1
Welcome	1

How MapReduce Works? Conti..

- The data goes through the following phases
- **Input Splits:**

An input to a MapReduce job is divided into fixed-size pieces called **input splits** Input split is a chunk of the input that is consumed by a single map

- **Mapping**

This is the very first phase in the execution of map-reduce program. In this phase data in each split is passed to a mapping function to produce output values. In our example, a job of mapping phase is to count a number of occurrences of each word from input splits (more details about input-split is given below) and prepare a list in the form of <word, frequency>

- **Shuffling**

This phase consumes the output of Mapping phase. Its task is to consolidate the relevant records from Mapping phase output. In our example, the same words are clubed together along with their respective frequency.

- **Reducing**

In this phase, output values from the Shuffling phase are aggregated. This phase combines values from Shuffling phase and returns a single output value. In short, this phase summarizes the complete dataset.

In our example, this phase aggregates the values from Shuffling phase i.e., calculates total occurrences of each word.

MapReduce Architecture explained in detail

- One map task is created for each split which then executes map function for each record in the split.
- It is always beneficial to have multiple splits because the time taken to process a split is small as compared to the time taken for processing of the whole input. When the splits are smaller, the processing is better to load balanced since we are processing the splits in parallel.
- However, it is also not desirable to have splits too small in size. When splits are too small, the overload of managing the splits and map task creation begins to dominate the total job execution time.
- For most jobs, it is better to make a split size equal to the size of an HDFS block (which is 64 MB, by default).
- Execution of map tasks results into writing output to a local disk on the respective node and not to HDFS.
- Reason for choosing local disk over HDFS is, to avoid replication which takes place in case of HDFS store operation.
- Map output is intermediate output which is processed by reduce tasks to produce the final output.
- Once the job is complete, the map output can be thrown away. So, storing it in HDFS with replication becomes overkill.
- In the event of node failure, before the map output is consumed by the reduce task, Hadoop reruns the map task on another node and re-creates the map output.
- Reduce task doesn't work on the concept of data locality. An output of every map task is fed to the reduce task. Map output is transferred to the machine where reduce task is running.
- On this machine, the output is merged and then passed to the user-defined reduce function.
- Unlike the map output, reduce output is stored in HDFS (the first replica is stored on the local node and other replicas are stored on off-rack nodes). So, writing the reduce output

How MapReduce Organizes Work?

- Hadoop divides the job into tasks. There are two types of tasks:

Map tasks (Splits & Mapping)

Reduce tasks (Shuffling, Reducing)

as mentioned above.

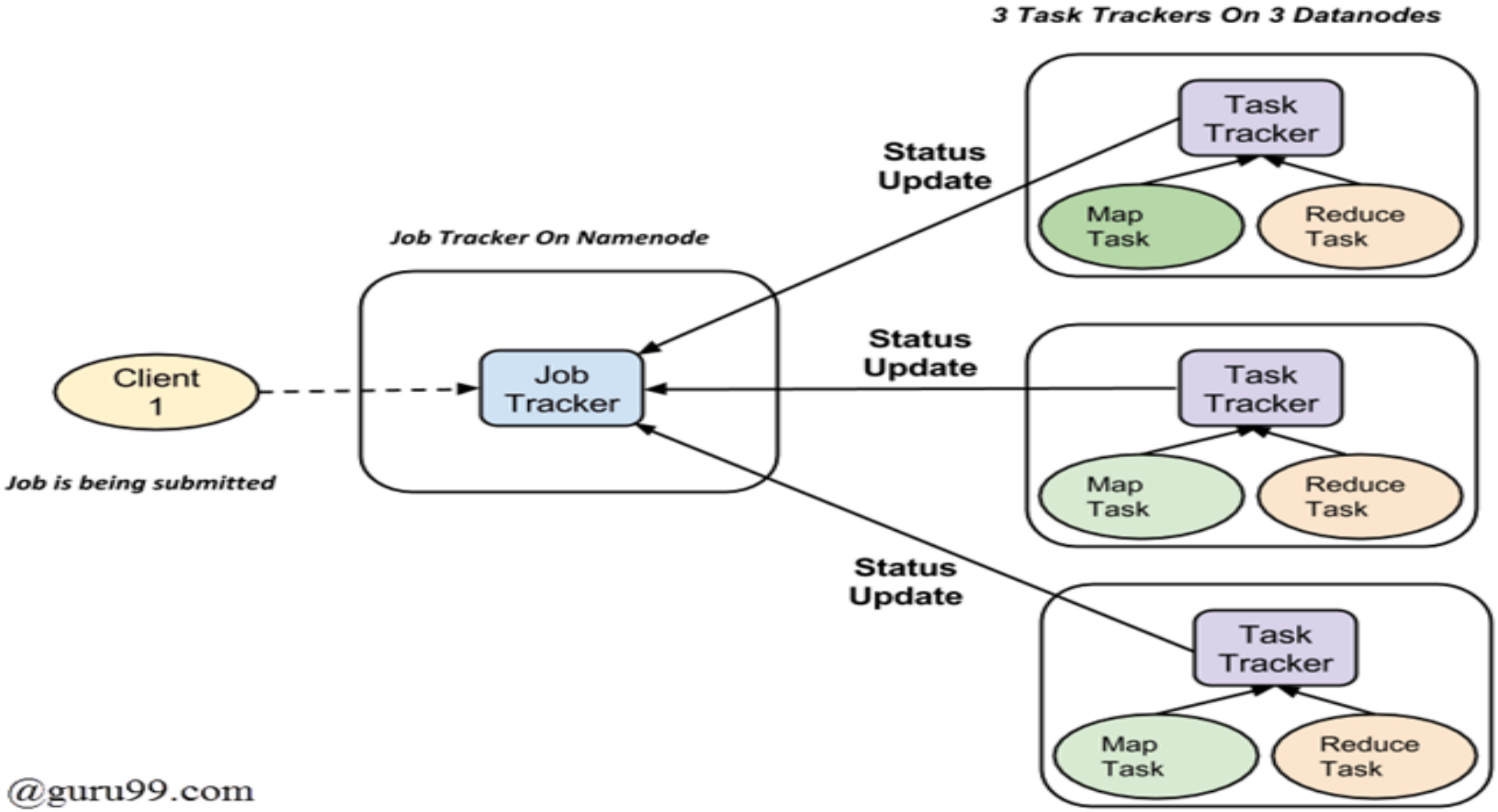
- The complete execution process (execution of Map and Reduce tasks, both) is controlled by two types of entities called a

1. Jobtracker: Acts like a **master** (responsible for complete execution of submitted job)

2. Multiple Task Trackers: Acts like **slaves,** each of them performing the job

- For every job submitted for execution in the system, there is one **Jobtracker** that resides on **Namenode** and there are **multiple tasktrackers** which reside on **Datanode**.

How MapReduce Organizes Work?



How MapReduce Organizes Work?

- A job is divided into multiple tasks which are then run onto multiple data nodes in a cluster.
- It is the responsibility of job tracker to coordinate the activity by scheduling tasks to run on different data nodes.
- Execution of individual task is then to look after by task tracker, which resides on every data node executing part of the job.
- Task tracker's responsibility is to send the progress report to the job tracker.
- In addition, task tracker periodically sends **'heartbeat'** signal to the Jobtracker so as to notify him of the current state of the system.
- Thus job tracker keeps track of the overall progress of each job. In the event of task failure, the job tracker can reschedule it on a different task tracker.

Map Reduce -Examples

Another map-reduce example

	A	B	C	D	E
1	Transaction_date	Product	Price	Payment_	Name
2	01-02-2009 06:17	Product1	1200	Mastercar	caroli
3	01-02-2009 04:53	Product1	1200	Visa	Betin
4	01-02-2009 13:08	Product1	1200	Mastercar	Feder
5	01-03-2009 14:44	Product1	1200	Visa	Gouy
6	01-04-2009 12:56	Product2	3600	Visa	Gerd
7	01-04-2009 13:19	Product1	1200	Visa	LAUR
8	01-04-2009 20:11	Product1	1200	Mastercar	Fleur
9	01-02-2009 20:09	Product1	1200	Mastercar	adam
10	01-04-2009 13:17	Product1	1200	Mastercar	Rene
11	01-04-2009 14:11	Product1	1200	Visa	Aidar
12	01-05-2009 02:42	Product1	1200	Diners	Stacy
13	01-05-2009 05:39	Product1	1200	Amex	Heidi
14	01-02-2009 09:16	Product1	1200	Mastercar	Sean
15	01-05-2009 10:08	Product1	1200	Visa	Georg
16	01-02-2009 14:18	Product1	1200	Visa	Richa
17	01-04-2009 01:05	Product1	1200	Diners	Leanr

SalesMapper.java

```
package SalesCountry;

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.mapred.*;

public class SalesMapper extends MapReduceBase implements Mapper <LongWrit
able, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);

        public void map(LongWritable key, Text value, OutputCollector <Tex
t, IntWritable> output, Reporter reporter) throws IOException {

                String valueString = value.toString();
                String[] SingleCountryData = valueString.split(",");
                output.collect(new Text(SingleCountryData[7]), one);
        }
}
```

Example Conti..

Reducer

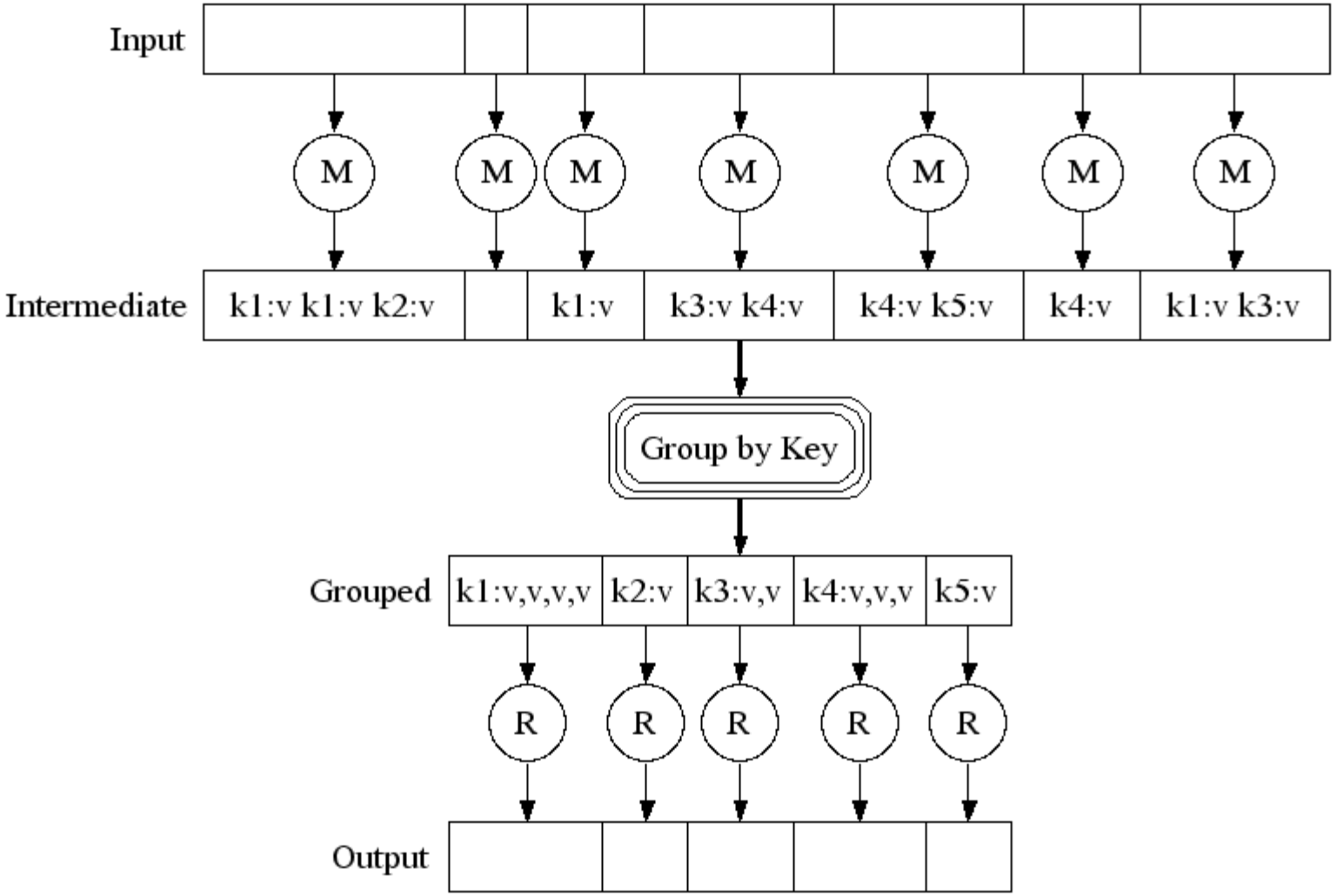
```
package SalesCountry;

import java.io.IOException;
import java.util.*;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.*;

public class SalesCountryReducer extends MapReduceBase implements Reducer<
Text, IntWritable, Text, IntWritable> {

        public void reduce(Text t_key, Iterator<IntWritable> values, Outpu
tCollector<Text,IntWritable> output, Reporter reporter) throws IOException
{
                Text key = t_key;
                int frequencyForCountry = 0;
                while (values.hasNext()) {
                        // replace type of value with the actual type of o
ur value
                        IntWritable value = (IntWritable) values.next();
                        frequencyForCountry += value.get();

                }
                output.collect(key, new IntWritable(frequencyForCountry));
        }
}
```

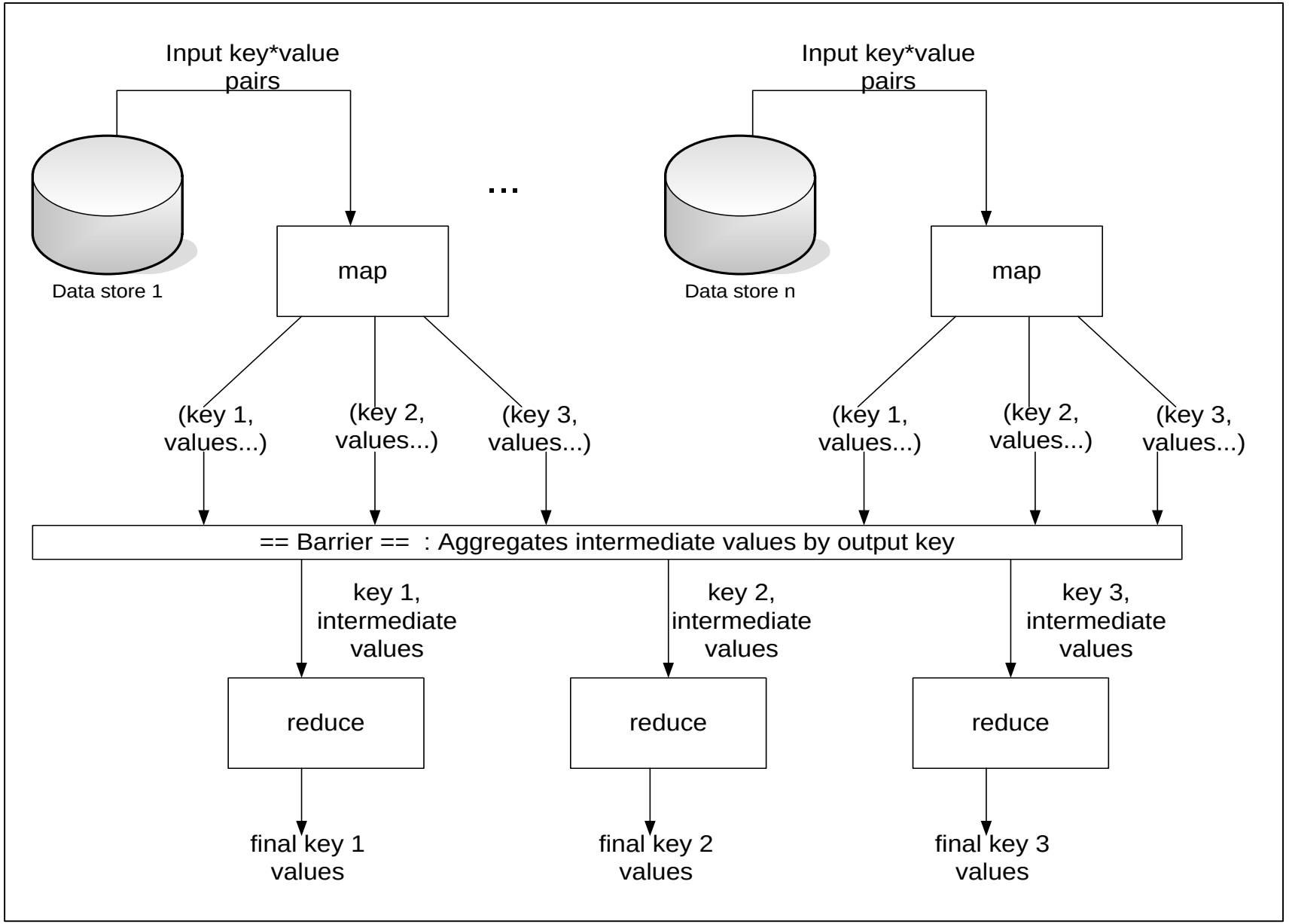
SalesCountryDriver.java

```
package SalesCountry;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

public class SalesCountryDriver {
    public static void main(String[] args) {
        JobClient my_client = new JobClient();
        // Create a configuration object for the job
        JobConf job_conf = new JobConf(SalesCountryDriver.class);

        // Set a name of the Job
        job_conf.setJobName("SalePerCountry");

        // Specify data type of output key and value
        job_conf.setOutputKeyClass(Text.class);
        job_conf.setOutputValueClass(IntWritable.class);

        // Specify names of Mapper and Reducer Class
        job_conf.setMapperClass(SalesCountry.SalesMapper.class);
        job_conf.setReducerClass(SalesCountry.SalesCountryReducer.class);

        // Specify formats of the data type of Input and output
        job_conf.setInputFormat(TextInputFormat.class);
        job_conf.setOutputFormat(TextOutputFormat.class);

        // Set input and output directories using command line arguments,
        //arg[0] = name of input directory on HDFS, and arg[1] =  name of output directory to be created to store the
output file.

        FileInputFormat.setInputPaths(job_conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(job_conf, new Path(args[1]));

        my_client.setConf(job_conf);
        try {
            // Run the job
            JobClient.runJob(job_conf);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

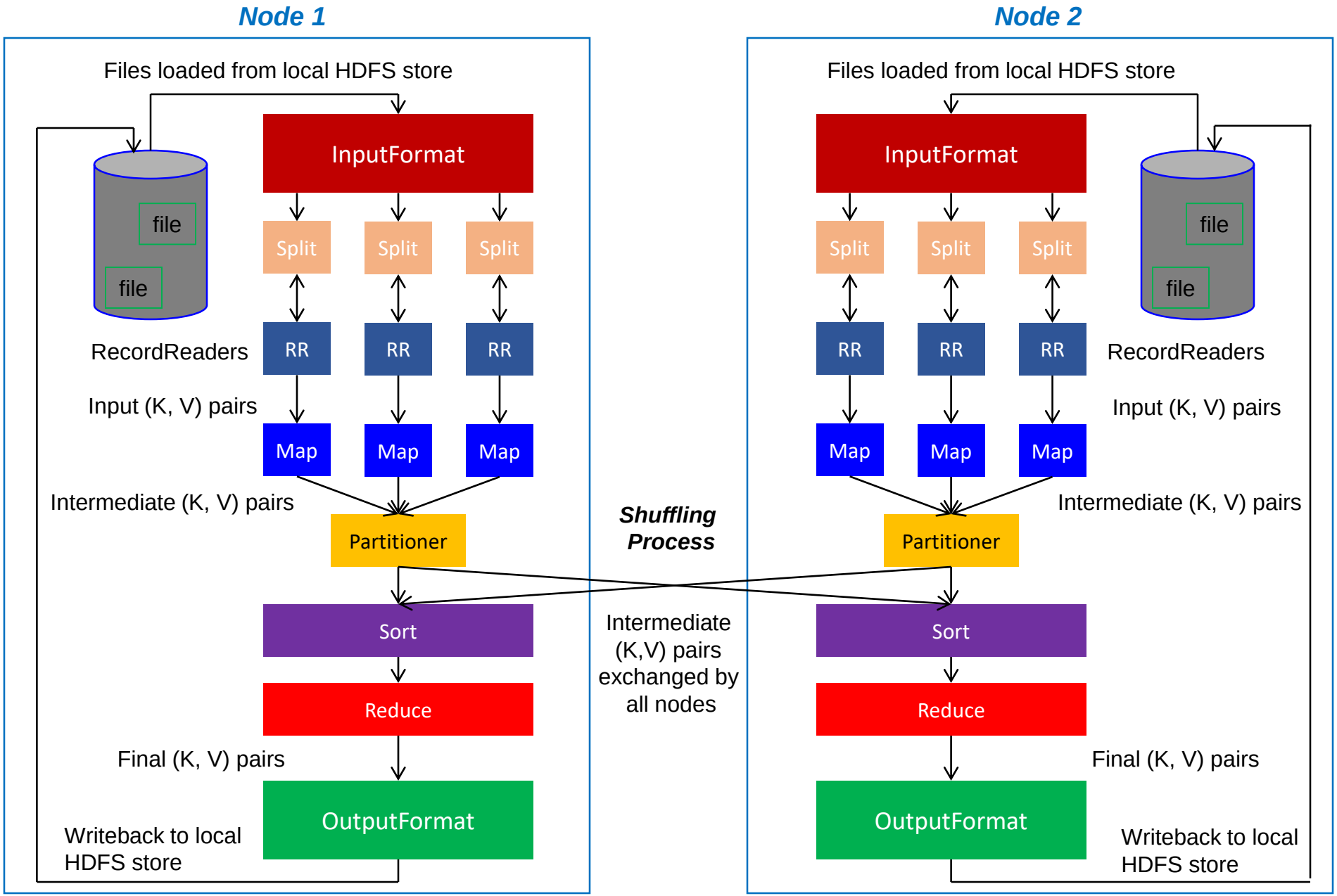
Main program

Execution Summary

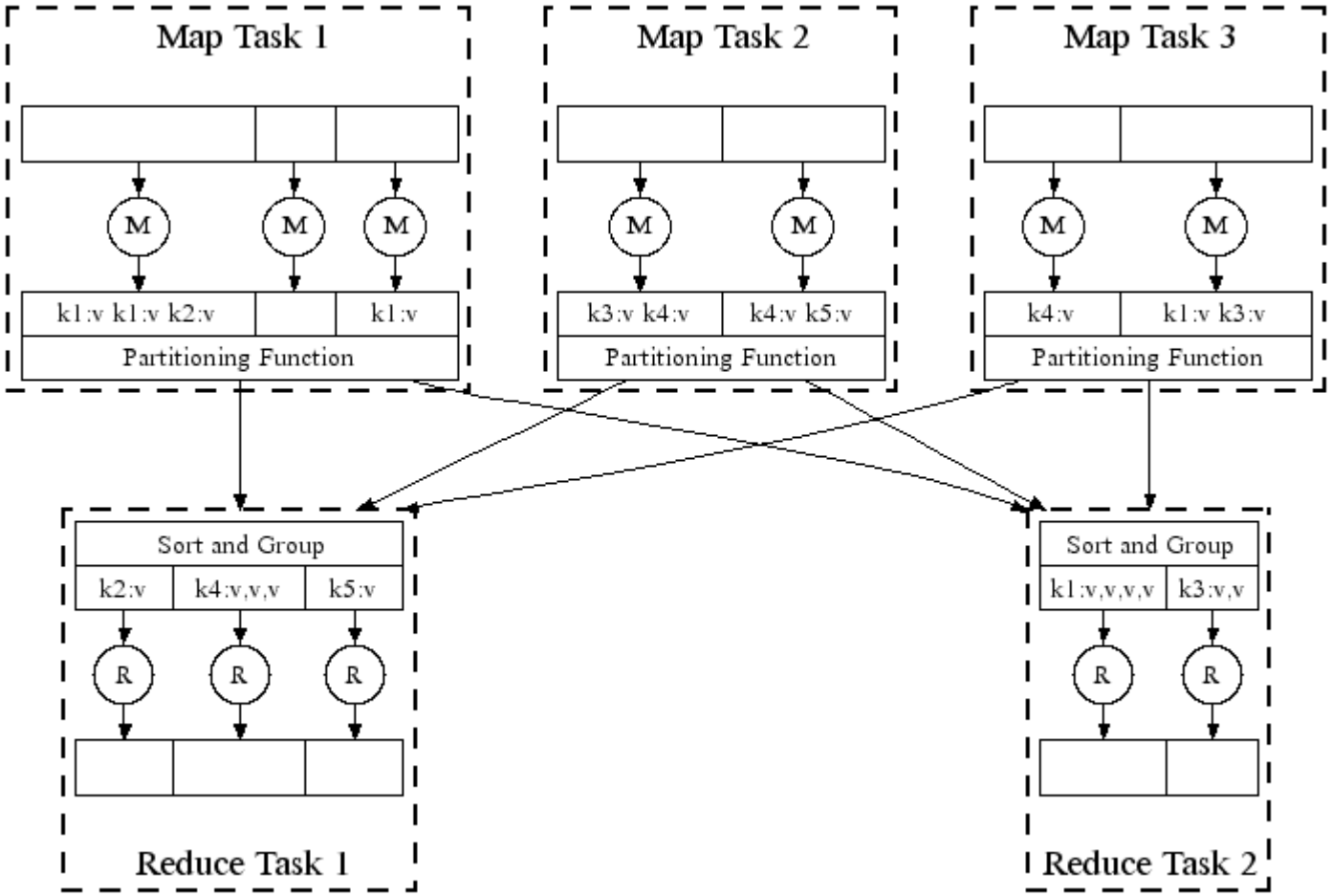



Input key*value pairs
Input key*value pairs
...
Data store 1
Data store n
map
map
(key 1, values...)
(key 2, values...)
(key 3, values...)
(key 1, values...)
(key 2, values...)
(key 3, values...)
== Barrier == : Aggregates intermediate values by output key
key 1, intermediate values
key 2, intermediate values
key 3, intermediate values
reduce
reduce
reduce
final key 1 values
final key 2 values
final key 3 values

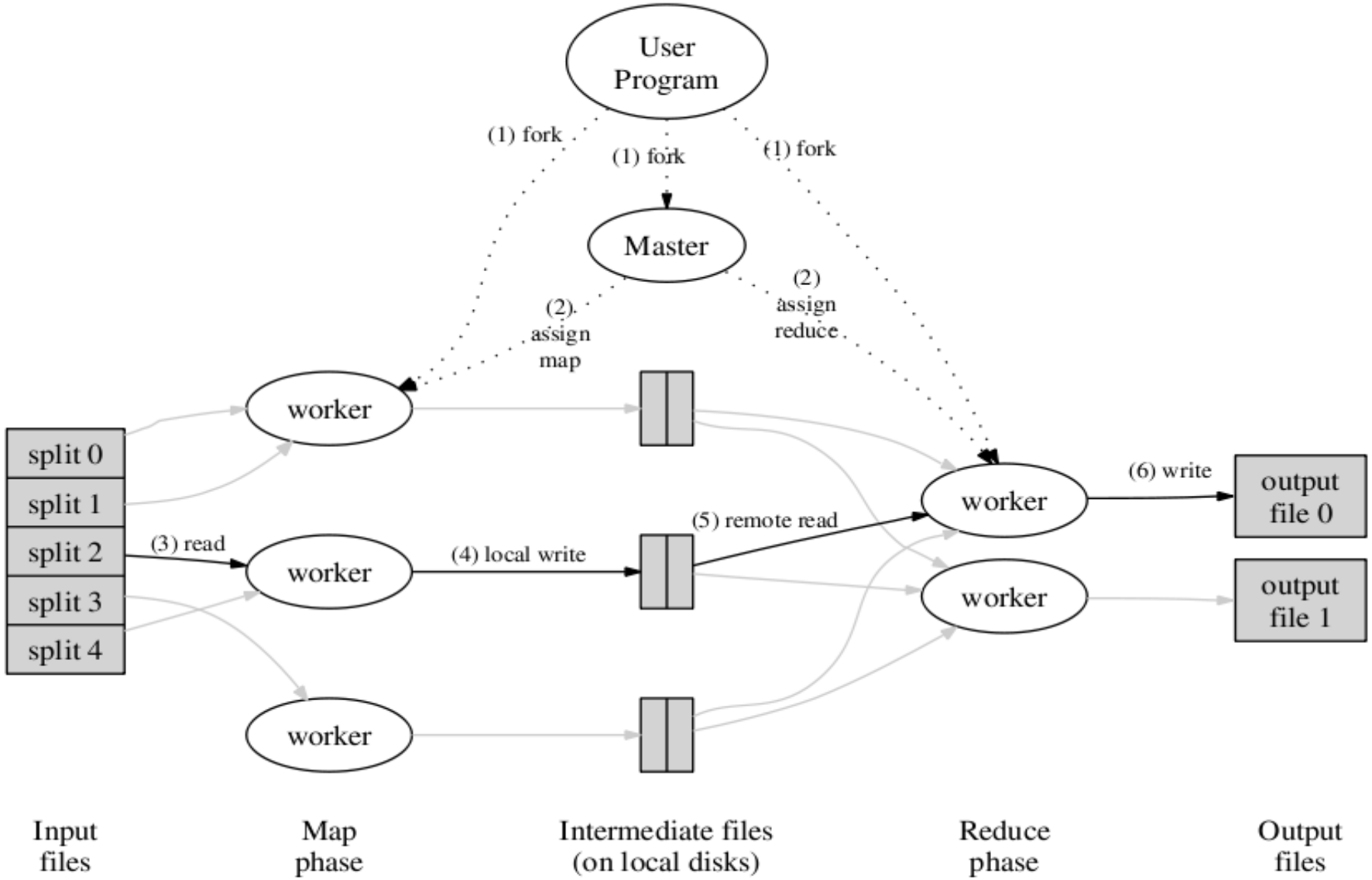
Hadoop MapReduce: A Closer Look



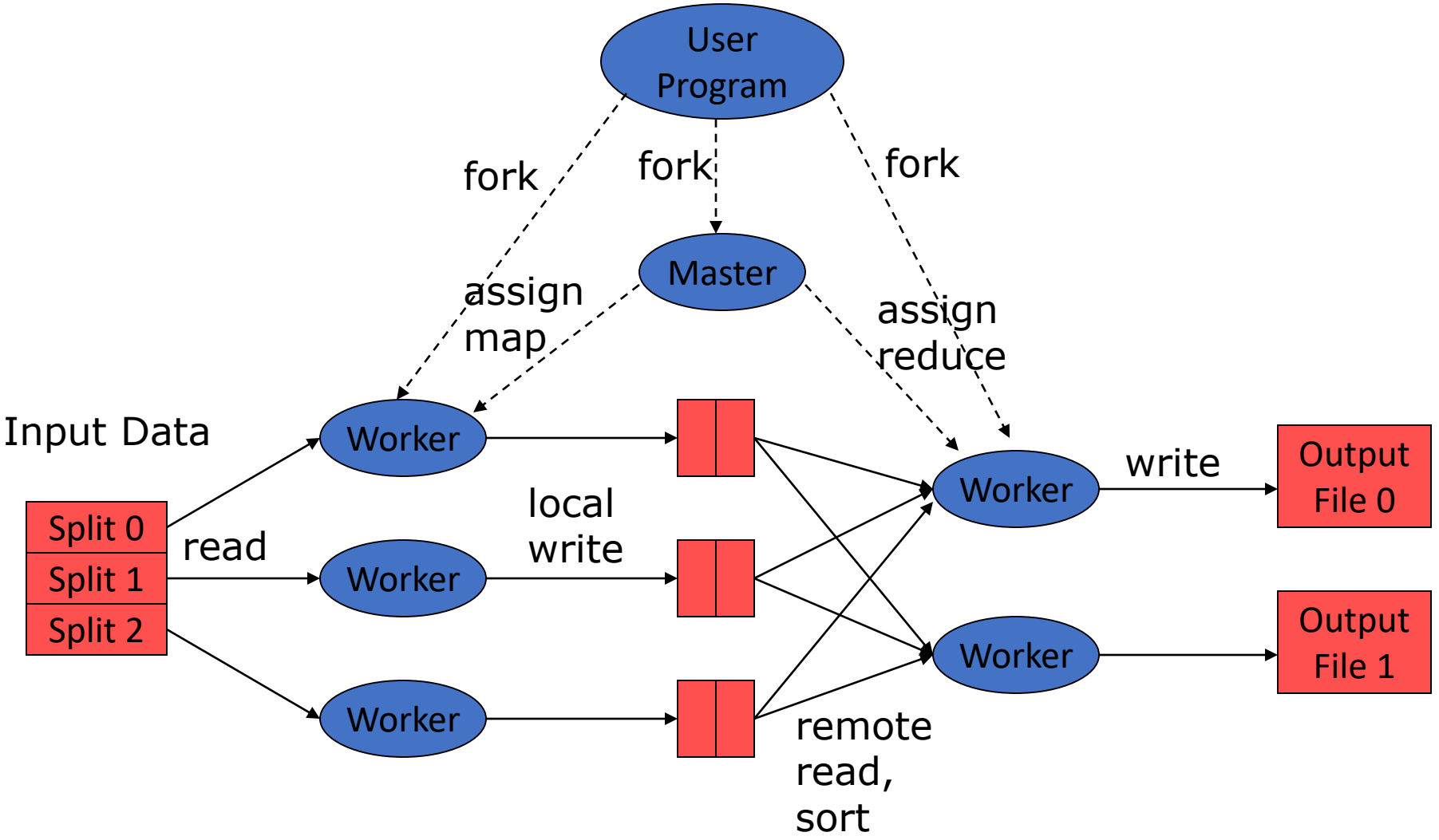
Parallel Execution



Distributed Execution Overview



Distributed Execution Overview



Distributed Execution Overview

1. The MapReduce library in the user program first splits the input files into **M** pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.
2. One of the copies of the program is special - the master. The rest are workers that are assigned work by the master. There are **M** map tasks and **R** reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.

Distributed Execution Overview

3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.
4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

Distributed Execution Overview

5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.
6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.
7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.

Data flow

- Input, final output are stored on a distributed file system
  - Scheduler tries to schedule map tasks “close” to physical storage location of input data
- Intermediate results are stored on local FS of map and reduce workers
- Output is often input to another map reduce task

Coordination

- Master data structures
  - Task status: (idle, in-progress, completed)
  - Idle tasks get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its R intermediate files, one for each reducer
  - Master pushes this info to reducers
- Master pings workers periodically to detect failures

Failures

- Map worker failure
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- Reduce worker failure
  - Only in-progress tasks are reset to idle
- Master failure
  - MapReduce task is aborted and client is notified

Robust: [Google's experience] lost 1600 of 1800 machines, but finished fine

Observations

- No *reduce* can begin until *map* is complete
- Tasks scheduled based on location of data
- If *map* worker fails any time before *reduce* finishes, task must be completely rerun
- Master must communicate locations of intermediate files
- MapReduce library does most of the hard work for us!

Fault Tolerance

- Workers are periodically pinged by master
  - No response = failed worker
- Master writes periodic checkpoints
- On errors, workers send “last gasp” UDP packet to master
  - Detect records that cause deterministic crashes and skips them

- Worker failure – handled via re-execution
  - Identified by no response to heartbeat messages
  - *In-progress* and *Completed* map tasks are re-scheduled
  - Workers executing reduce tasks are notified of re-scheduling
  - *Completed* reduce tasks are not re-scheduled
- Master failure
  - Rare
  - Can be recovered from checkpoints
  - All tasks abort

MapReduce Conclusions

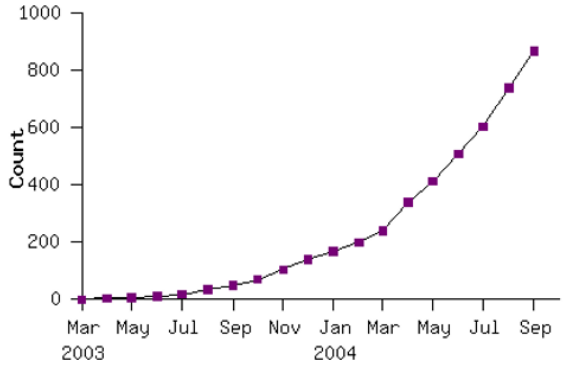
- Simplifies large-scale computations that fit this model
- Allows user to focus on the problem without worrying about details
- Computer architecture not very important
  - Portable model

Implementations

- Google
  - Not available outside Google
- Hadoop
  - An open-source implementation in Java
  - Uses HDFS for stable storage
  - Download: http://lucene.apache.org/hadoop/
- Aster Data
  - Cluster-optimized SQL Database that also implements MapReduce
- Aster Data and Hadoop can both be run on EC2
- And several others, such as Cassandra at Facebook, etc.

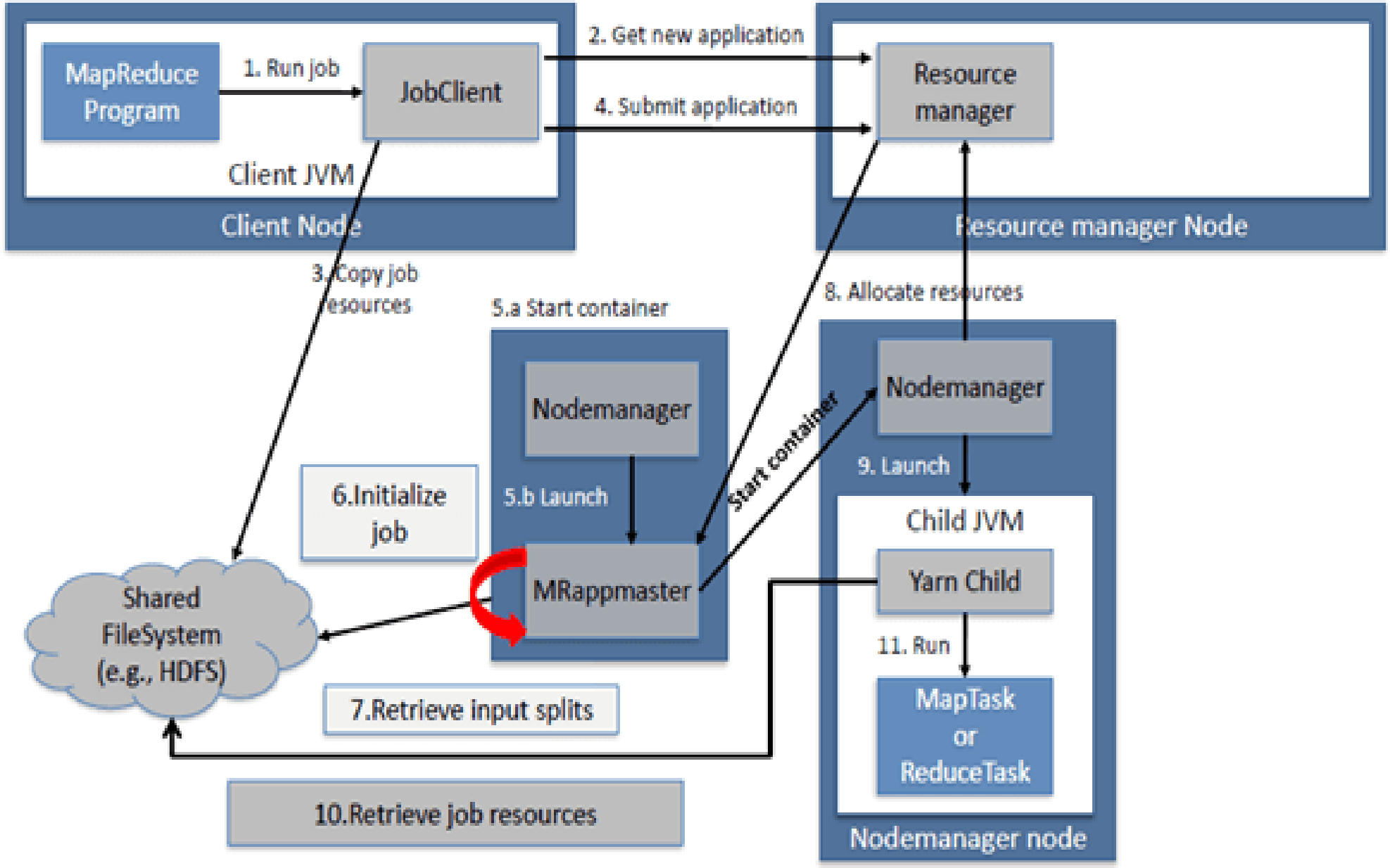
Model is Widely Applicable

MapReduce Programs In Google Source Tree

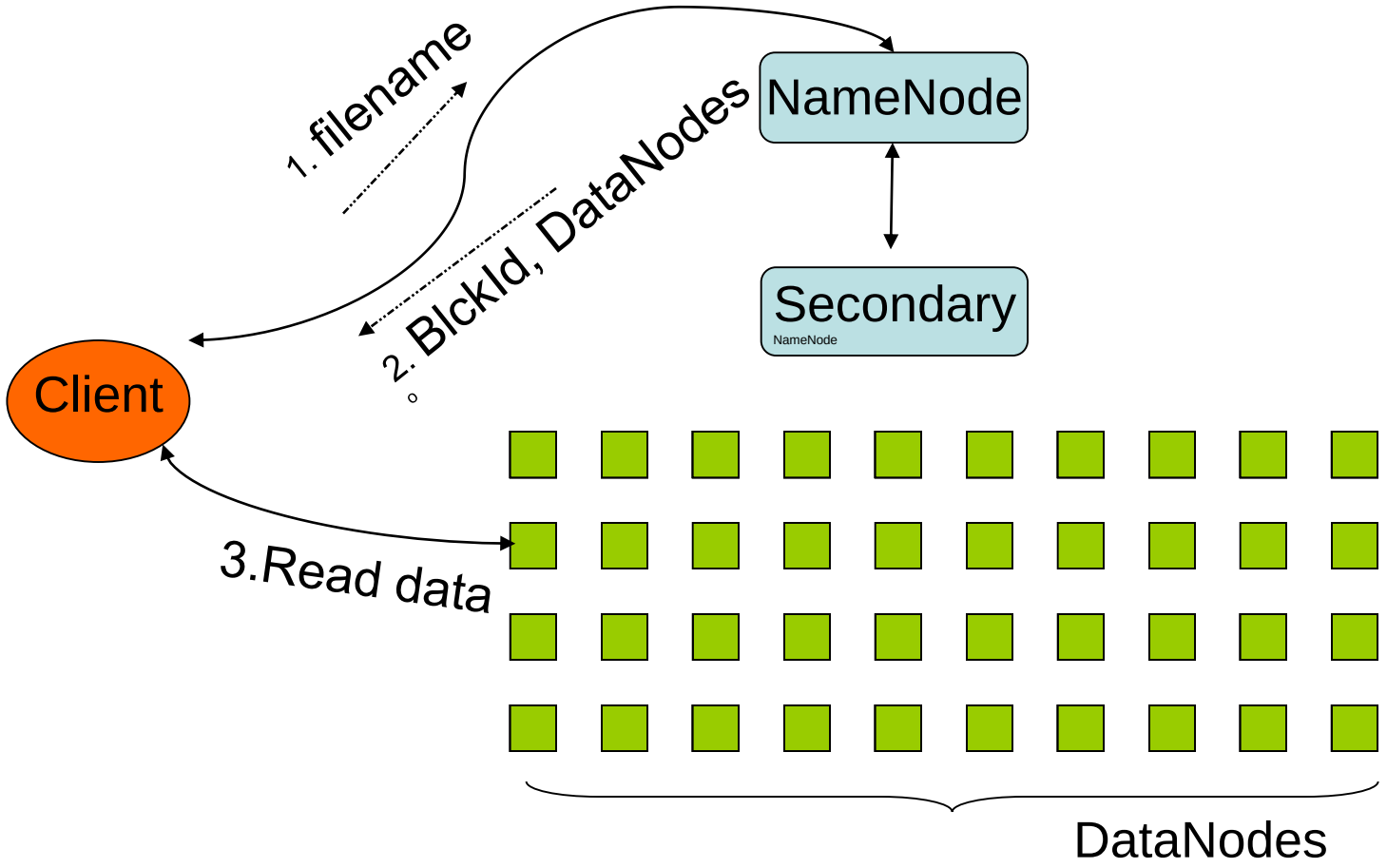


Example uses:

distributed grep
distributed sort
web link-graph reversal
term-vector / host
web access log stats
inverted index construction
document clustering
machine learning
statistical machine translation


MapReduce Program
1. Run job
JobClient
Client JVM
Client Node
2. Get new application
4. Submit application
Resource manager
Resource manager Node
3. Copy job resources
5.a Start container
Nodemanager
6.Initialize job
5.b Launch
MRappmaster
Start container
8. Allocate resources
Nodemanager
9. Launch
Child JVM
Yarn Child
11. Run
MapTask or ReduceTask
Nodemanager node
Shared FileSystem (e.g., HDFS)
7.Retrieve input splits
10.Retrieve job resources

HDFS Architecture



NameNode : Maps a file to a file-id and list of MapNodes
DataNode : Maps a block-id to a physical location on disk

HDFS Architecture