Chapter 3
Map-Reduce Framework

Bal Krishna Nyaupane
Assistant Professor
Department of Electronics and Computer Engineering
Institute of Engineering, Tribhuvan University
bkn@wrc.edu.np

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.***
  - The Functional Programming Paradigm is one of the ***major programming paradigms.***
  - ***Idea: everything is a function.***

What is a Functional Language?

- FP is a type of ***declarative programming paradigm***. Its main ***focus is on "what to solve"*** in ***contrast to an imperative style*** where the main focus is ***"how to solve***".
- It uses ***expressions instead of statements***. An expression is ***evaluated to produce a value*** whereas a statement is executed to assign variables.
- Also known as applicative programming and value-oriented programming.
- 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

What is a Functional Language?

- 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*

Functional Programming Language

Summing the integers 1 to 10 in Java:

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

Summing the integers 1 to 10 in Haskell:

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

The computation method is function application.

Functional Programming Language

Traditional Imperative Loop:

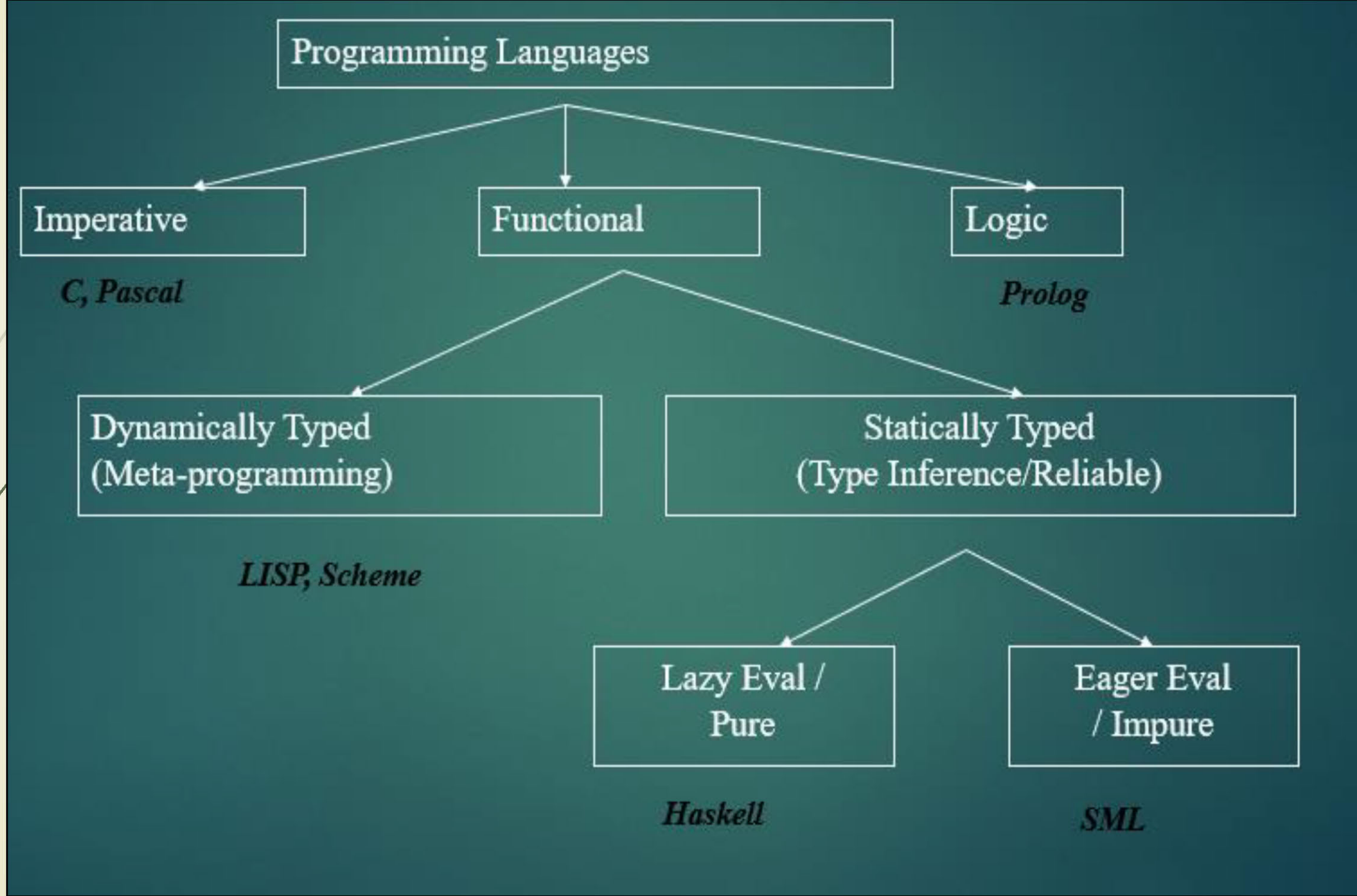
```
const numList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
let result = 0;
for (let i = 0; i < numList.length; i++) {
  if (numList[i] % 2 === 0) {
    result += numList[i] * 10;
  }
}
```

Functional Programming with higher-order functions:

```
const result = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
               .filter(n => n % 2 === 0)
               .map(a => a * 10)
               .reduce((a, b) => a + b);
```

Imperative Languages vs. Functional Languages

- ***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


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?

- It was developed ***at Google for indexing Web pages*** and replaced their original indexing algorithms and ***heuristics in 2004.***
- MapReduce is the ***heart of Hadoop.***
- Apache has produced an ***open source MapReduce platform*** called Hadoop.
- MapReduce is a ***software framework*** that allows developers to write programs that process ***massive amounts of unstructured*** data in parallel, ***on large clusters of commodity hardware*** in a reliable manner.
- MapReduce ***allows data to be distributed across a large cluster***, and can ***distribute out tasks across*** the data set to work on pieces of it ***independently, and in parallel.***
- MapReduce is a processing ***technique and a program model*** for ***distributed computing*** based on java.

What is MapReduce?

- ***MapReduce is*** a software framework for easily writing applications which process big amounts of data in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.
- The term MapReduce actually refers to the ***following two different tasks*** that Hadoop programs perform:
  - ***The Map Task:*** This is the first task, which takes ***input data*** and ***converts it into a set of data***, where individual elements ***are broken down into tuples (key/value pairs).***
  - ***The Reduce Task:*** This task takes the output from a ***map task as input*** and ***combines those data tuples into a smaller set of tuples***. The reduce task is always performed after the map task.
- The ***major advantage*** of MapReduce is that ***it is easy to scale data processing*** over multiple computing nodes.

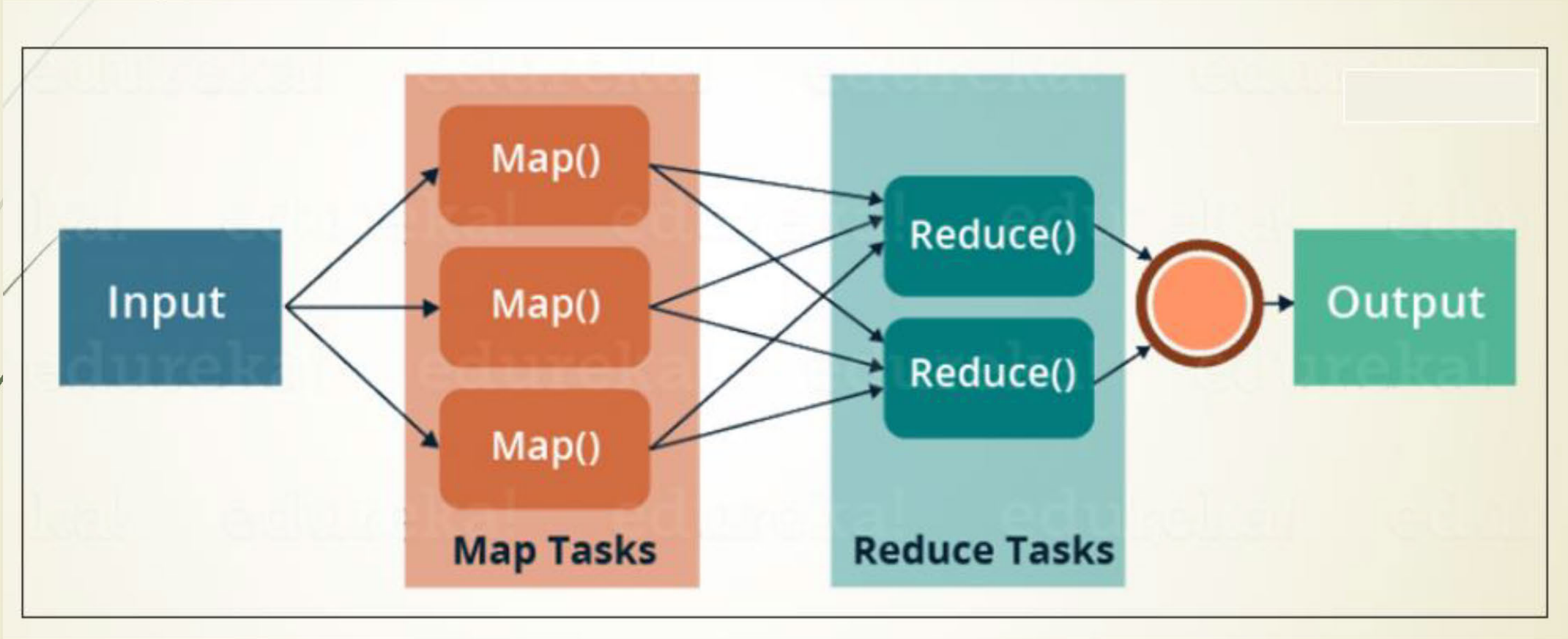
What is MapReduce?

- ***Parallel programming*** model meant ***for large clusters***
  - User implements ***Map() and Reduce()***
- ***Parallel computing framework***
  - Libraries take ***care of EVERYTHING*** else
    - ***Parallelization***
    - ***Fault Tolerance***
    - ***Data Distribution***
    - ***Load Balancing***
- ***Useful model for many practical tasks (large data)***

The Algorithm

- MapReduce program executes in three stages, namely map stage, shuffle stage, and reduce stage.
- **Map stage**: The map or mapper's job is to process the input data. Generally the input data is in the form of file or directory and is stored in the Hadoop file system (HDFS). The input file is passed to the mapper function line by line. The mapper processes the data and creates several small chunks of data.
- **Reduce stage**: This stage is the combination of the Shuffle stage and the Reduce stage. The Reducer's job is to process the data that comes from the mapper. After processing, it produces a new set of output, which will be stored in the HDFS.

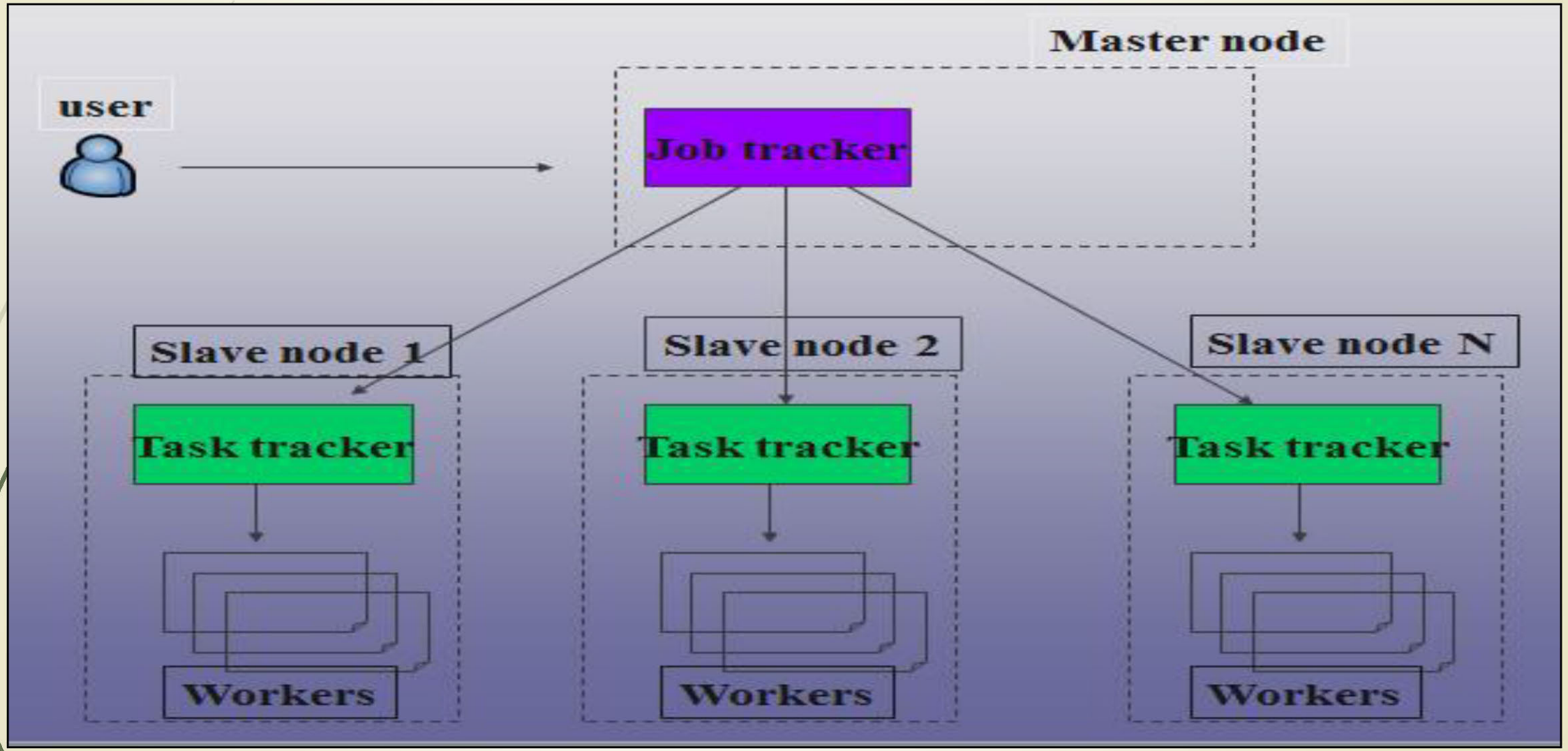
The Algorithm



Inserting Data into HDFS

- The MapReduce framework ***operates on <key, value> pairs***, that is, the framework views the input to the ***job as a set of <key, value> pairs*** and ***produces a set of <key, value> pairs*** as the output of the job, conceivably of different types.
- The key and the ***value classes should be in serialized manner*** by the framework and hence, need to implement the Writable interface. Additionally, the key classes have to implement the Writable-Comparable interface to facilitate sorting by the framework.
- ***Input and Output types of a MapReduce job:*** (Input)<k1,v1> -> map -> <k2, v2>-> reduce -> <k3, v3> (Output).

MapReduce Architecture



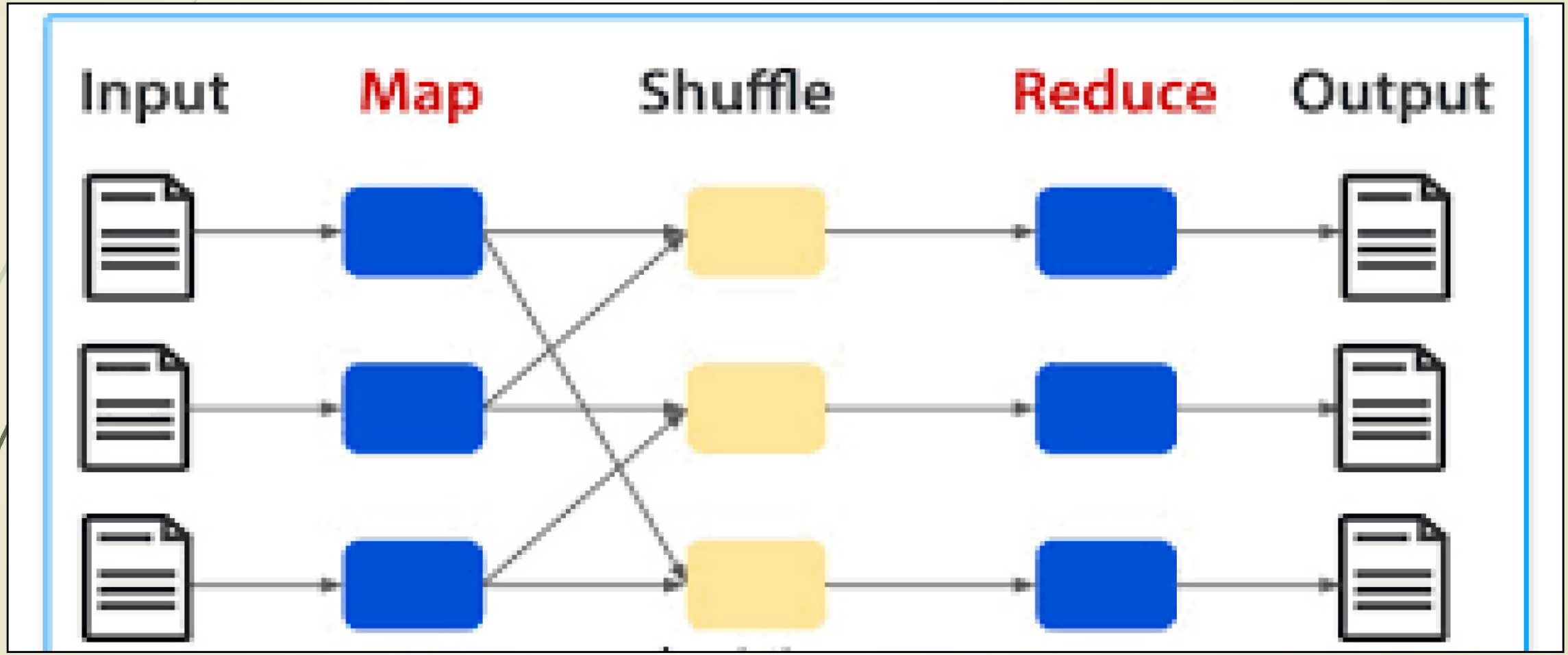
MapReduce Architecture

- The ***MapReduce framework*** consists of ***a single master JobTracker*** and ***one slave TaskTracker*** per cluster-node.
- The ***master is responsible for***
  - ***resource management,***
  - ***tracking resource consumption/availability*** and
  - ***scheduling the jobs component tasks*** on the slaves, monitoring them and re-executing the failed tasks.
- The ***slaves TaskTracker***
  - execute the ***tasks as directed by the master*** and
  - ***provide task-status information to the master*** periodically.

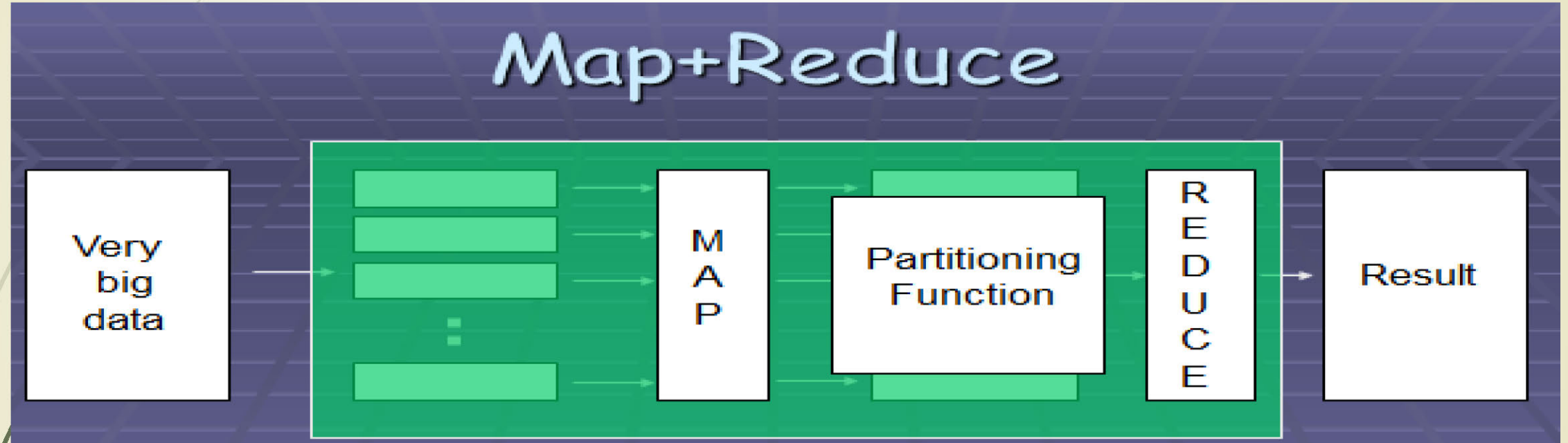
MapReduce Architecture

- The ***JobTracker is a single point of failure*** for the Hadoop MapReduce service which means ***if JobTracker goes down***, all running ***jobs are halted.***
- The ***MapReduce framework is fault-tolerant*** because ***each node in the cluster is expected to report back periodically*** with completed work and status updates.
  - If a node ***remains silent for longer than the expected interval***, a master node ***makes note and re-assigns*** the work to other nodes.

MapReduce Architecture

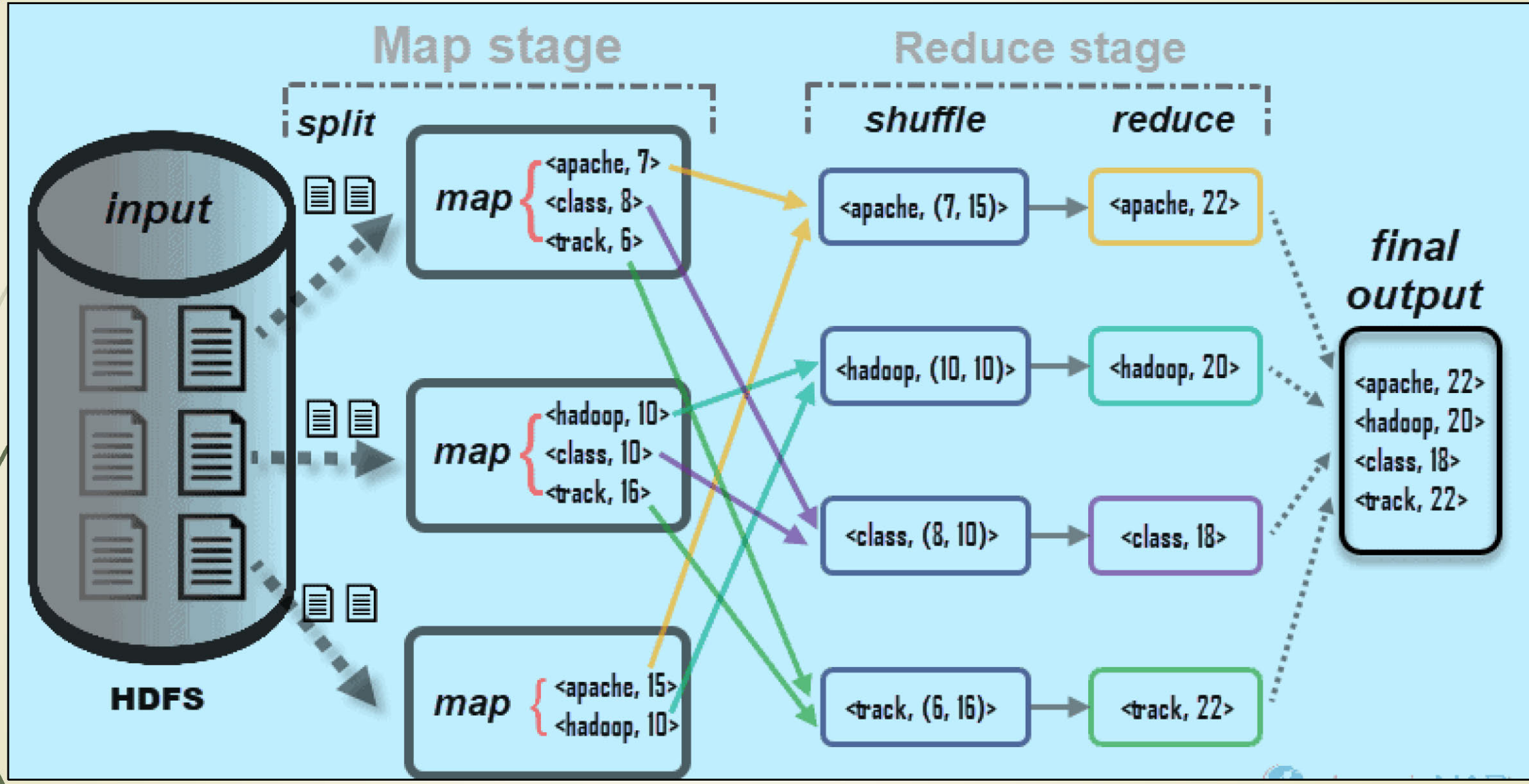


MapReduce Architecture

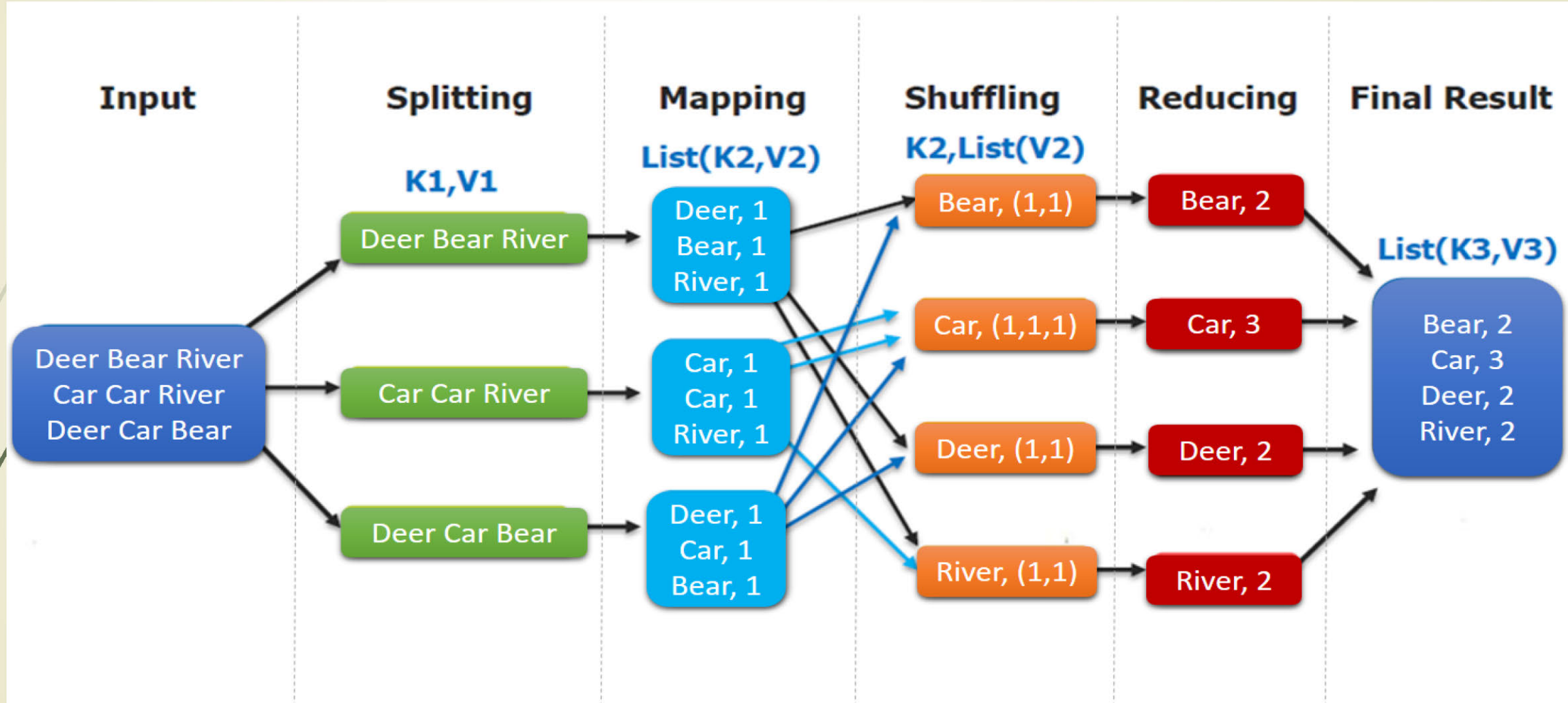


- Map:
  - Accepts *input* key/value pair
  - Emits *intermediate* key/value pair
- Reduce :
  - Accepts *intermediate* key/value* pair
  - Emits *output* key/value pair

MapReduce Wordcount Example



MapReduce Wordcount Example



MapReduce Word Count Process

MapReduce Programming Model

- Inspired from map and reduce operations commonly used in functional programming languages like Lisp.
- ***Map Abstraction***
  - Inputs a key/value pair
    - –Key is a reference to the input value
    - –Value is the data set on which to operate
  - Evaluation
    - –Function defined by user
    - –Applies to every value in value input
      - Might need to parse input
  - Produces a new list of key/value pairs
    - –Can be different type from input pair

- Input: a set of key/value pairs
- User supplies two functions:
  - map(k, v) → list(k1,v1)
  - reduce(k1, list(v1)) → v2
- (k1,v1) is an intermediate key/value pair
- Output is the set of (k1,v2) pairs

MapReduce Programming Model

- ***Reduce Abstraction***
  - ***Starts with intermediate Key / Value pairs***
  - ***Ends with finalized Key / Value pairs***
  - *Starting pairs are sorted by key*
  - *Iterator supplies the values for a given key to the Reduce function.*

```
def map(key, value):
    list = []
    for x in value:
        if test:
            list.append( (key, x) )
    return list
```

```
def reduce(key, listOfValues):
    result = 0
    for x in listOfValues:
        result += x
    return (key, result)
```

Data Distribution

Input files are split into **M** pieces on distributed file system

Typically ~ 64 MB blocks

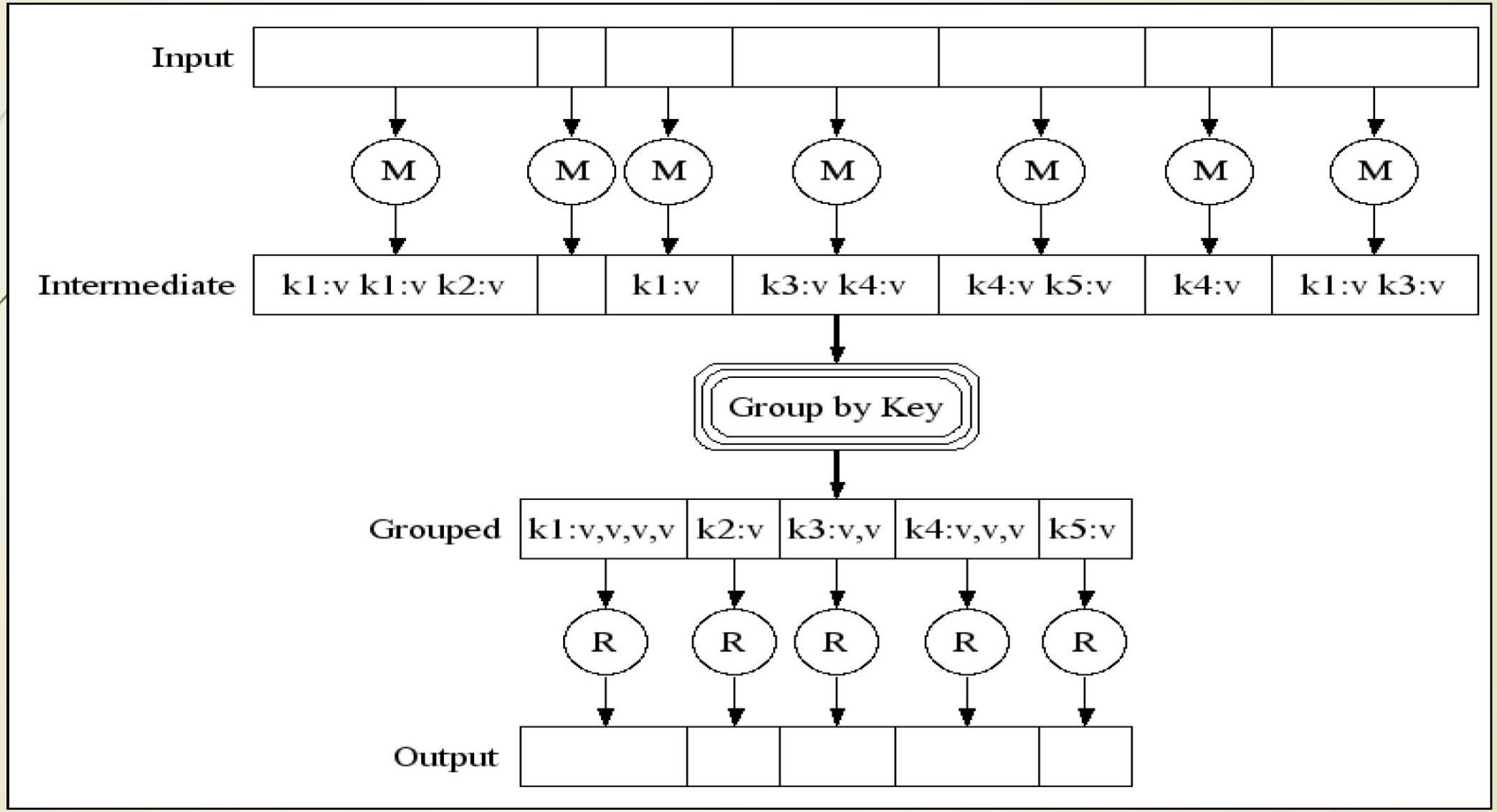
Intermediate files created from *map* tasks are written to local disk

Output files are written to distributed file system

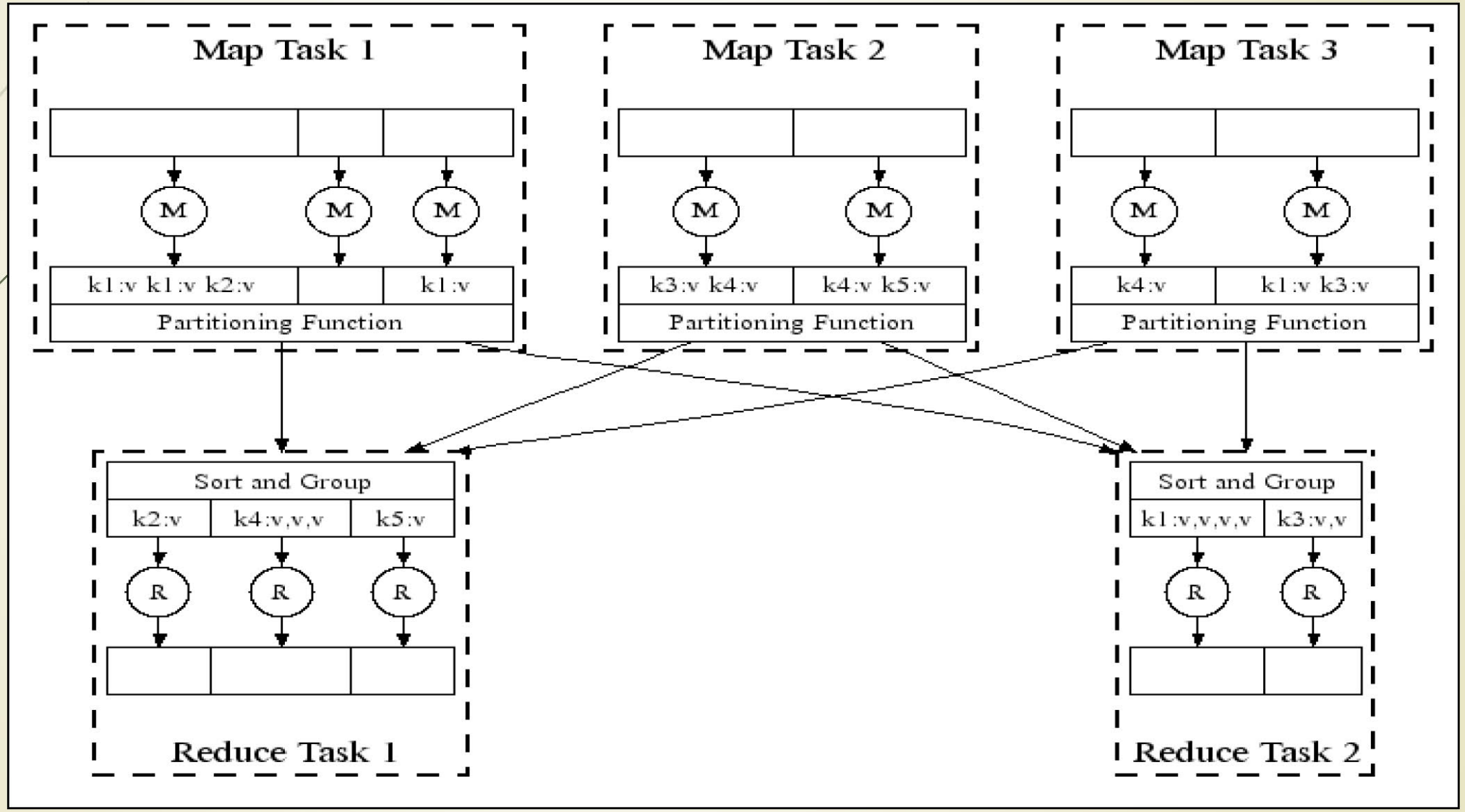
Execution

- ***Execution (map)***
  - Map workers ***read in contents of corresponding input partition***
  - ***Perform user-defined map computation*** to create intermediate <key,value> pairs
  - ***Periodically buffered output*** pairs written to local disk
    - Partitioned into **R** regions by a partitioning function
- ***Execution (reduce)***
  - Reduce workers ***iterate over ordered intermediate data***
  - Each unique key encountered – values are passed to user's reduce function e.g. <key, [value1, value2,..., valueN]>
  - Output of user's reduce function is written ***to output file on global file system***
  - When all tasks have completed, master wakes up user program

Execution


Input
M
M
M
M
M
M
M
Intermediate
k1:v k1:v k2:v
k1:v
k3:v k4:v
k4:v k5:v
k4:v
k1:v k3:v
Group by Key
Grouped
k1:v,v,v,v
k2:v
k3:v,v
k4:v,v,v
k5:v
R
R
R
R
R
Output

Parallel Execution



Distributed Execution Overview

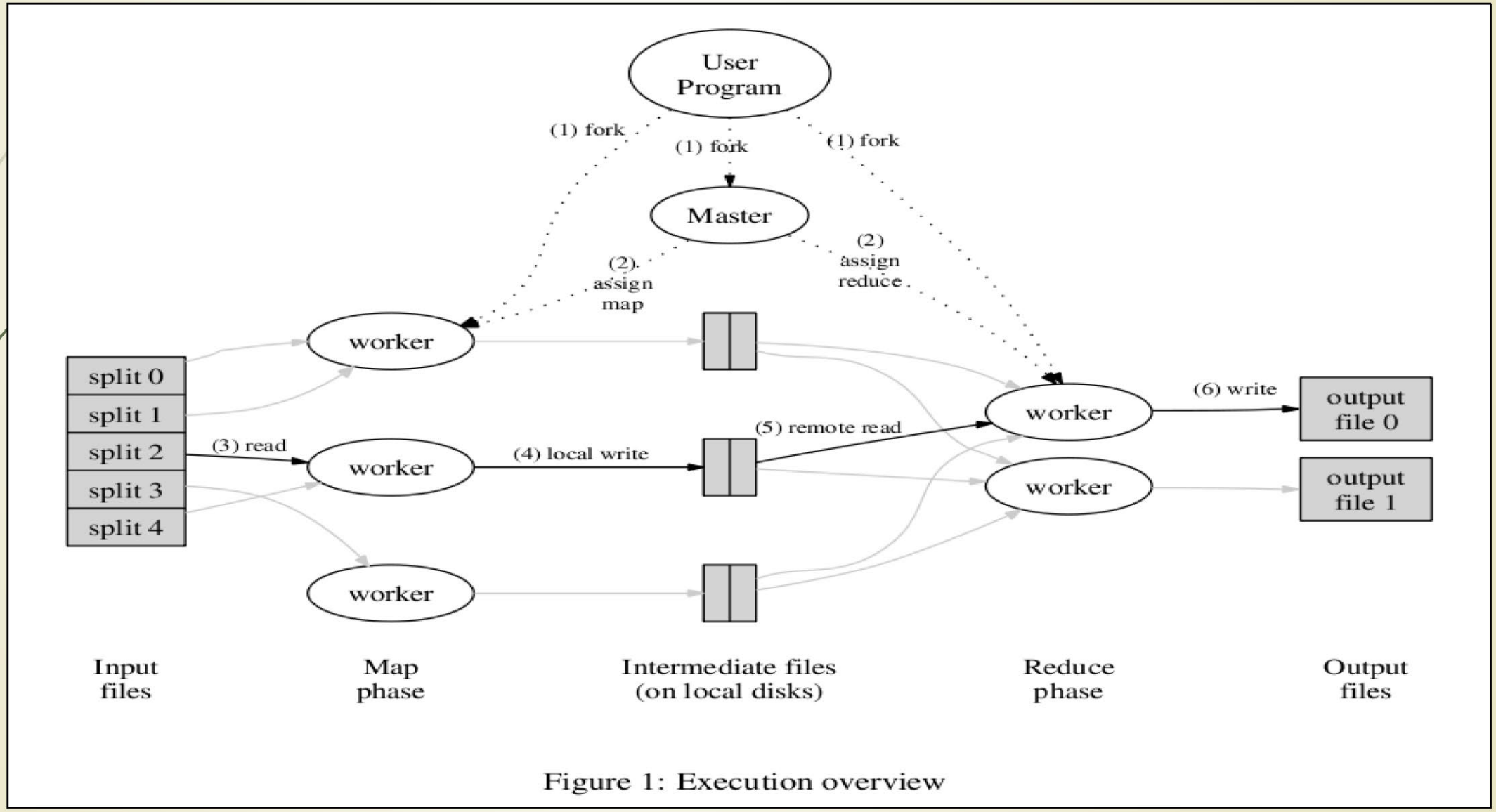


Figure 1: Execution overview

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.***
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.

Distributed Execution Overview

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.***

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.

Distributed Execution Overview

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

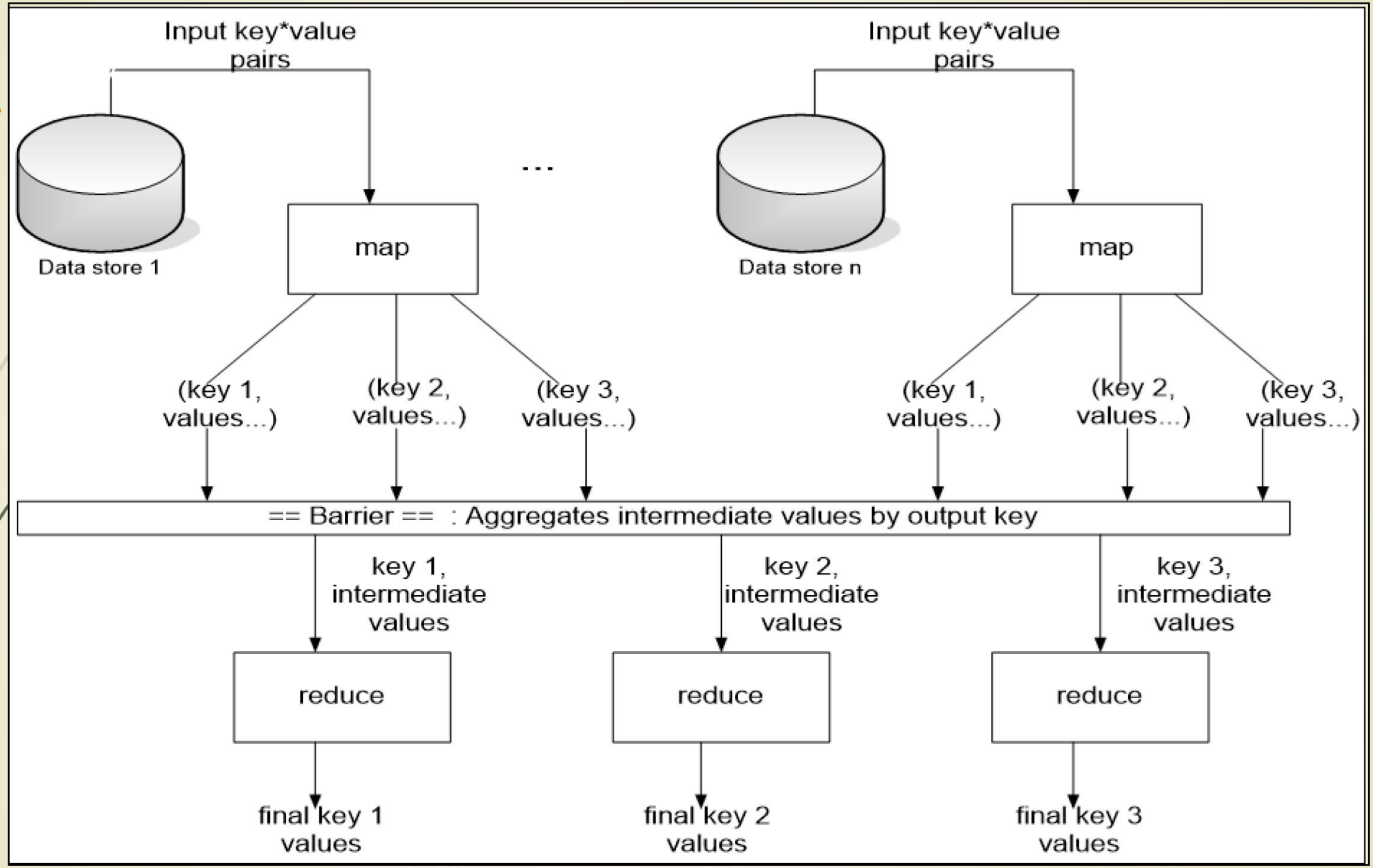
- ***Master data structures***
  - ***Task status:*** (idle, in-progress, completed)
  - Idle tasks get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its R intermediate files, one for each reducer
  - Master pushes this info to reducers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

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!


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

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

```
public class WordCount {
 public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
 }}}
```

Mapper

```
public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
}}
```

Reducer

```
 public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
}}
```

Run this program as a MapReduce job

Cloud Computing with Map-Reduce and Hadoop

Author: Dwight D. Anderson Mentors: Wenjun Zeng, Qia Wang

SUSLA

Objectives

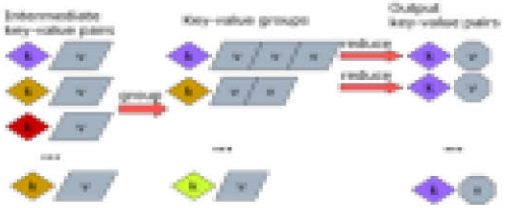
- Cloud Computing
- Parallel processing
- Map-Reduce algorithm
- Hadoop environment

Map-Reduce

- Simple data-parallel programming model designed for scalability and fault-tolerance
- Pioneered by Google
- Popularized by Hadoop project

The Map and Reduce Step

map(k,v) → list(k1,v1)
reduce(k1, list(v1)) → v2

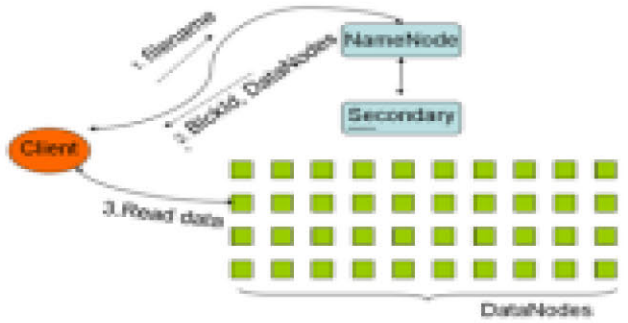


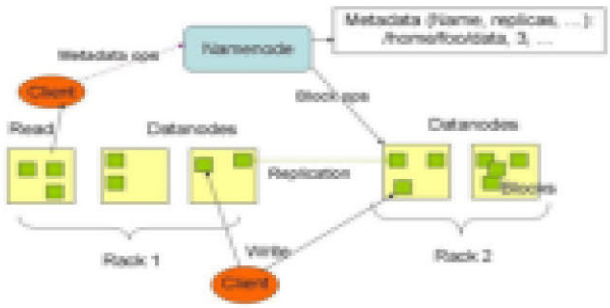
Distributed Execution Overview

The Hadoop Environment

- Distributed, highly fault-tolerant file system
- Handling large data sets
- Communication protocols are build on the TCP/IP model.
- HDFS: Hadoop Distributed File System

HDFS Architecture

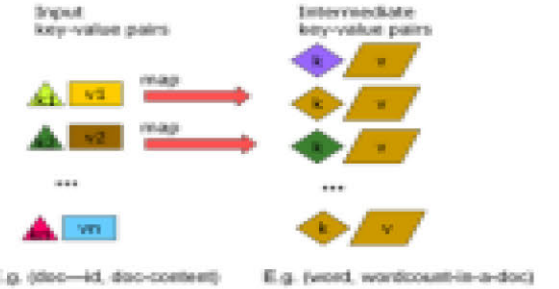




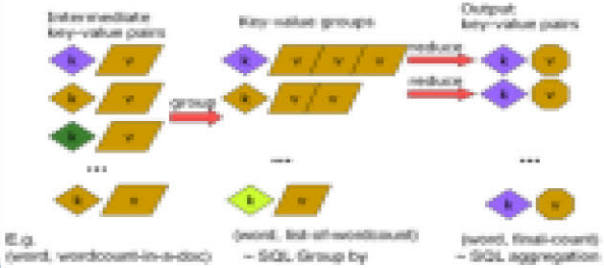
NameNode : Maps a file to a file-id and list of MapNodes
DataNode : Maps a block-id to a physical location on disk

Implementation: Word Count

The Map Step



The Reduce Step



Verification

- All the tasks are simulated within a single thread
- Results are verified correctly

Future Work

- Run the task with multi-threads or multi-servers to compare performance
- Apply Map-Reduce to more complicated data-mining algorithms

Thank You
???