Big Data Technologies

Jnaneshwar Bohara

Chapter 3: Map-Reduce Framework

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.

Example

Summing the integers 1 to 10 in Haskell:

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

The computation method is function application.

Summary: ILs vs FPLs

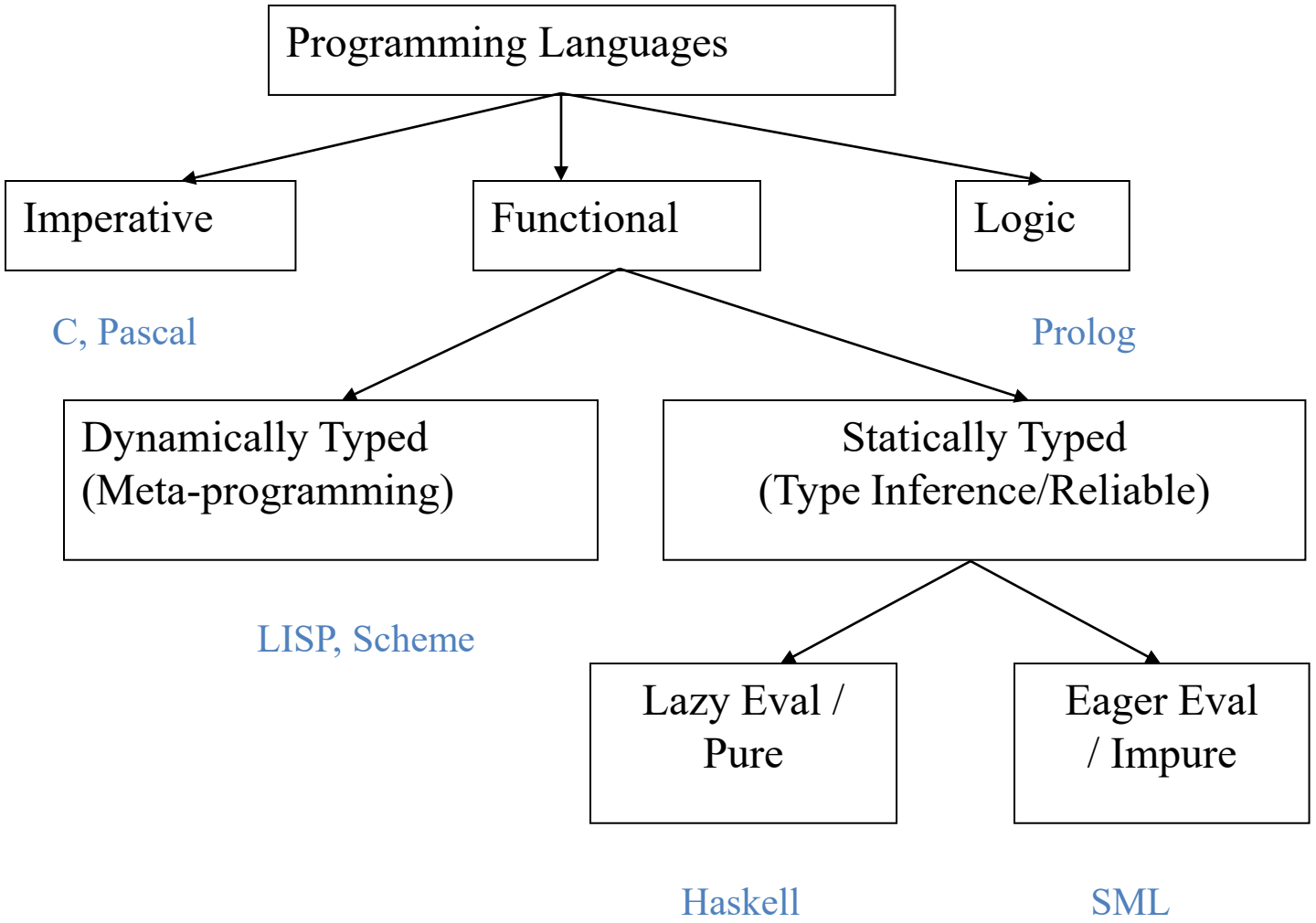
Imperative Languages:

- 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?

- 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)

Functional Abstractions Hide Parallelism

- Map and Reduce
- Functions borrowed from functional programming languages (eg. Lisp)
- Map()
  - Process a key/value pair to generate intermediate key/value pairs
- Reduce()
  - Merge all intermediate values associated with the same key

Example: Counting Words

- Map()
  - Input <filename, file text>
  - Parses file and emits <word, count> pairs
    - eg. <”hello”, 1>
- Reduce()
  - Sums values for the same key and emits <word, TotalCount>
    - eg. <”hello”, (3 5 2 7)> => <”hello”, 17>

Example Use of MapReduce

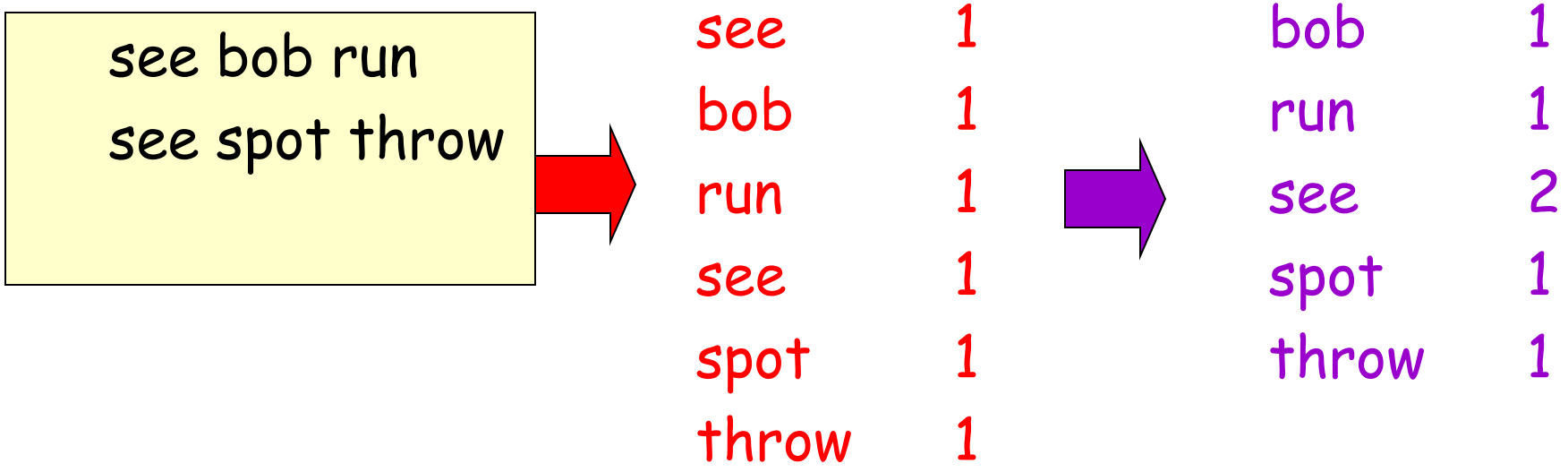
- Counting words in a large set of documents

```
map(string key, string value)
    //key: document name
    //value: document contents
    for each word w in value
        EmitIntermediate(w, “1”);

reduce(string key, iterator values)
    //key: word
    //values: list of counts
    int results = 0;
    for each v in values
        result += ParseInt(v);
    Emit(AsString(result));
```

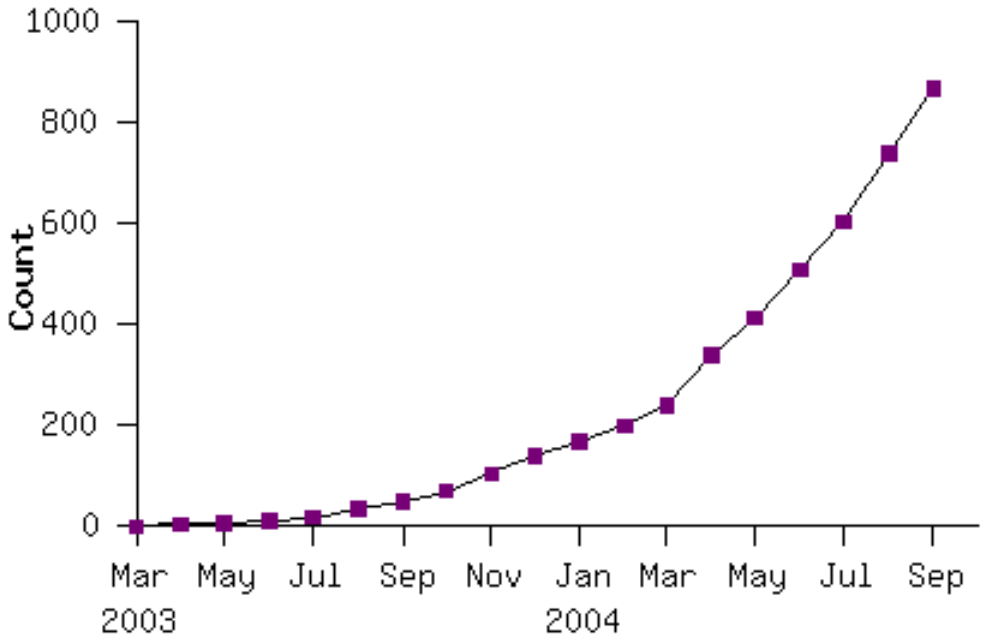
Count, Illustrated

map(key=url, val=contents):
For each word *w* in contents, emit (w, "1")
reduce(key=word, values=uniq_counts):
Sum all "1"s in values list
Emit result "(word, sum)"



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

...

...

...

How MapReduce Works

- User to do list:
  - indicate:
    - Input/output files
    - **M**: number of map tasks
    - **R**: number of reduce tasks
    - **W**: number of machines
  - Write *map* and *reduce* functions
  - Submit the job
- This requires no knowledge of parallel/distributed systems!!!
- What about everything else?

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

Assigning Tasks

- Many copies of user program are started
- Tries to utilize data localization by running *map* tasks on machines with data
- One instance becomes the Master
- Master finds idle machines and assigns them tasks

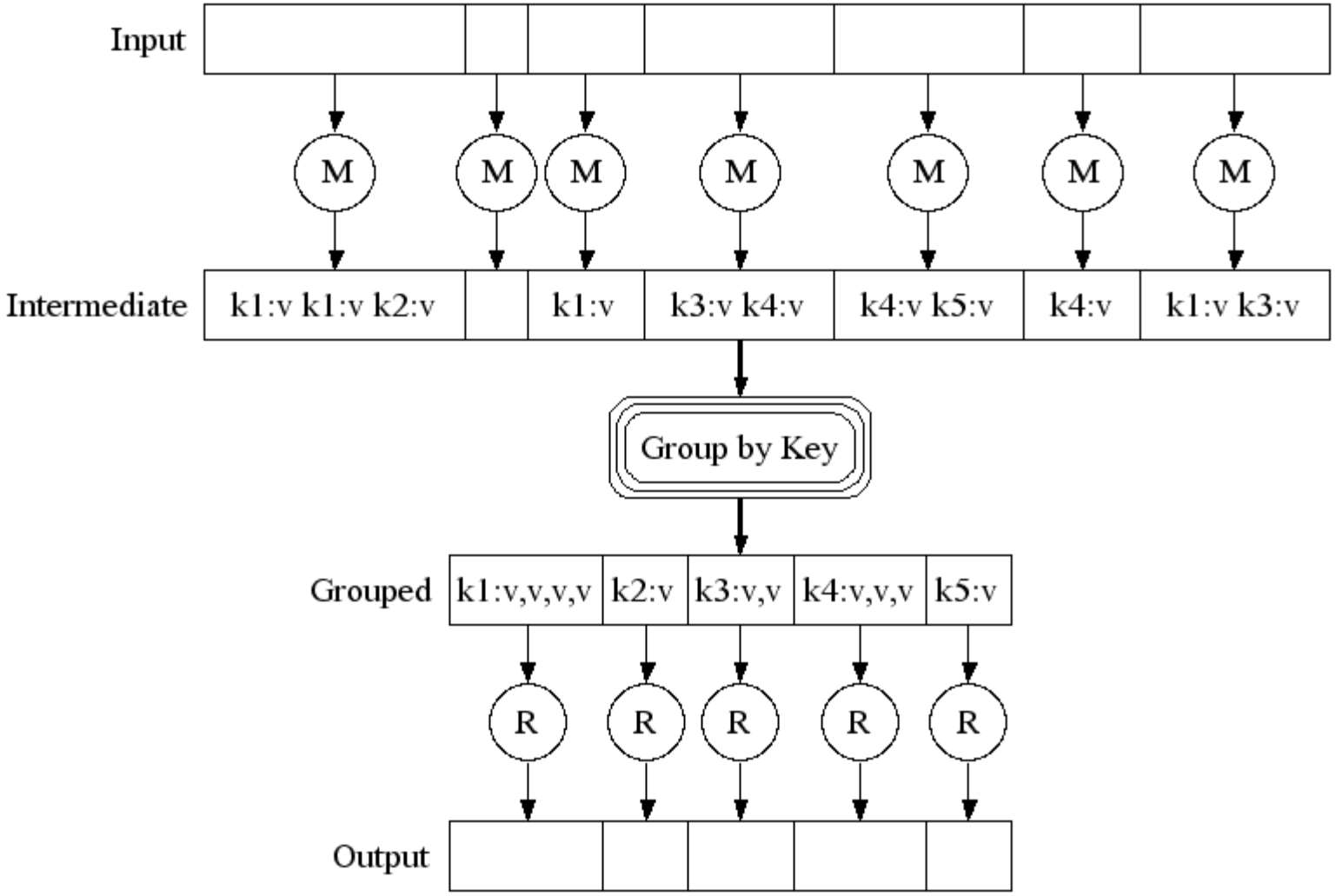
Execution (map)

- *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

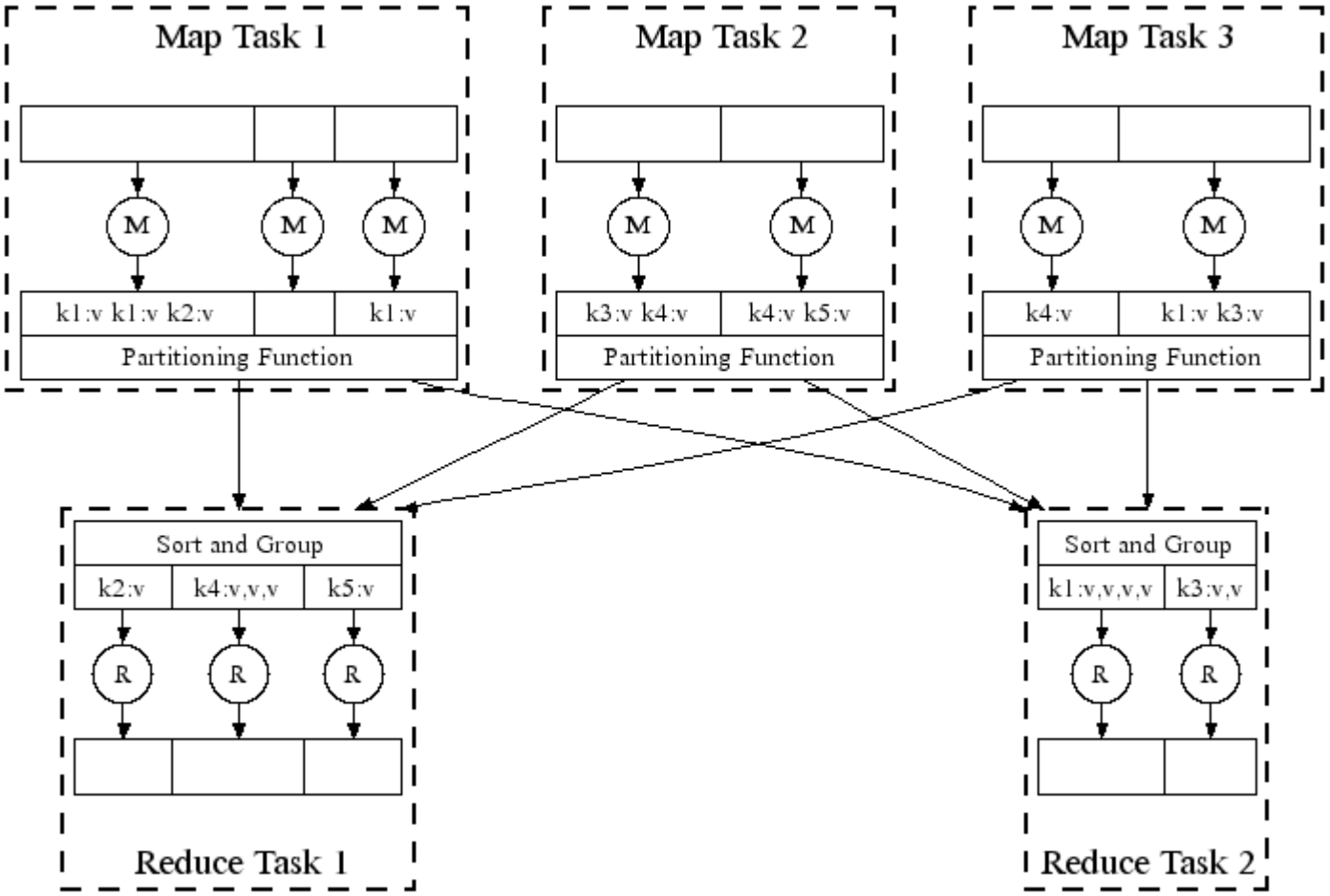
Execution (reduce)

- Reduce workers iterate over ordered intermediate data
  - Each unique key encountered – values are passed to user's reduce function
  - eg. <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

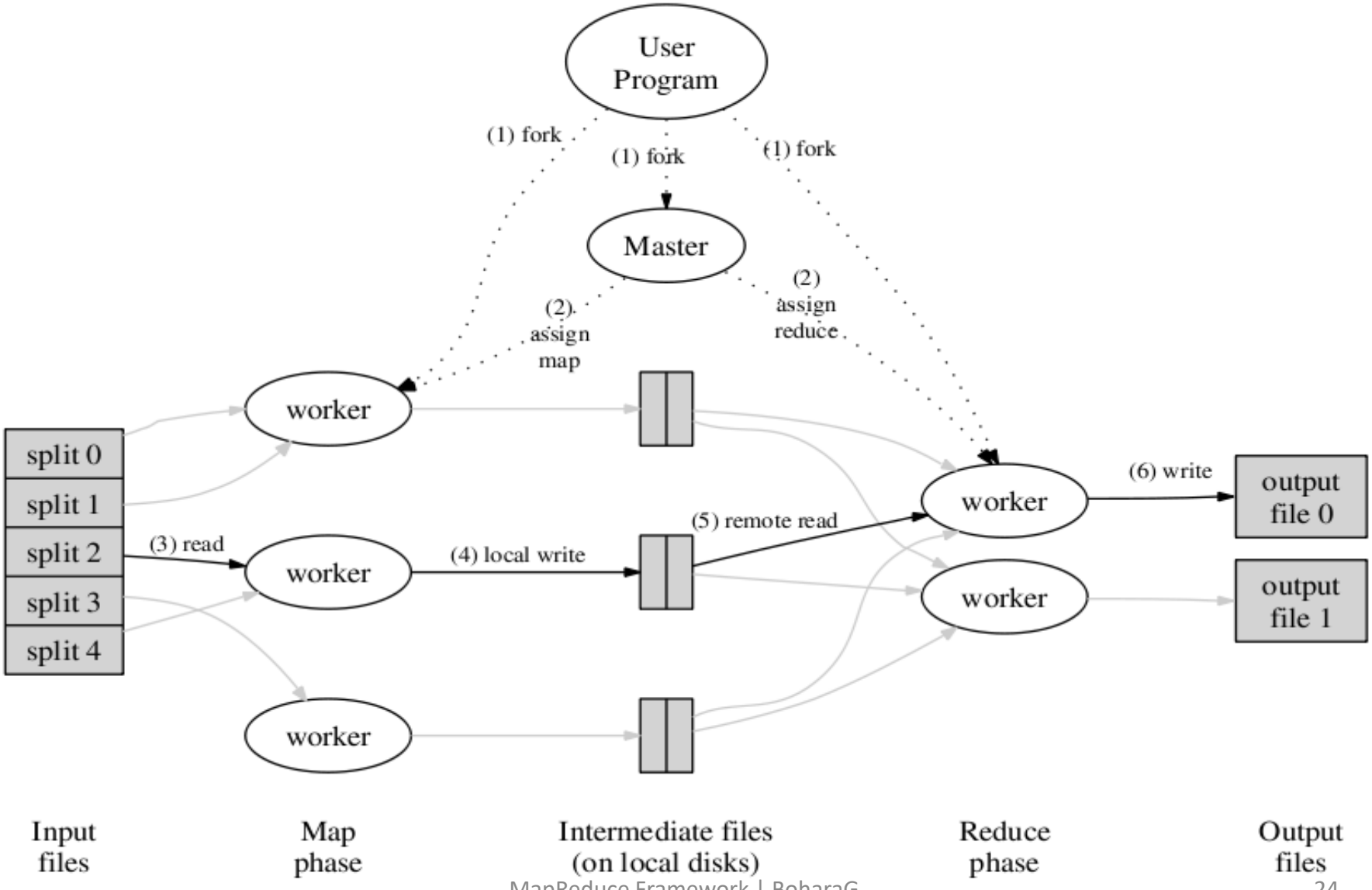
Execution


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

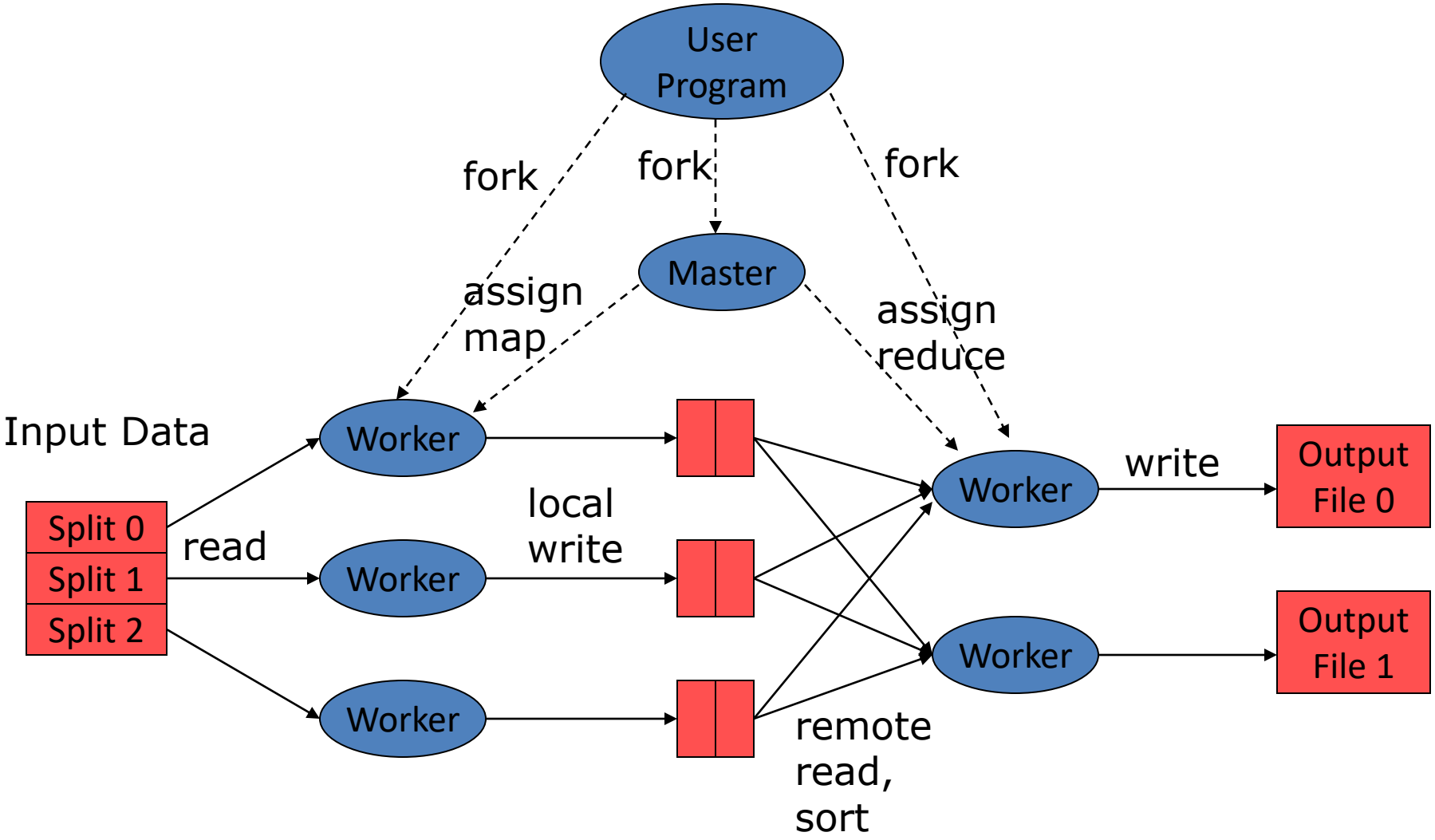
Parallel Execution



Distributed Execution Overview



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

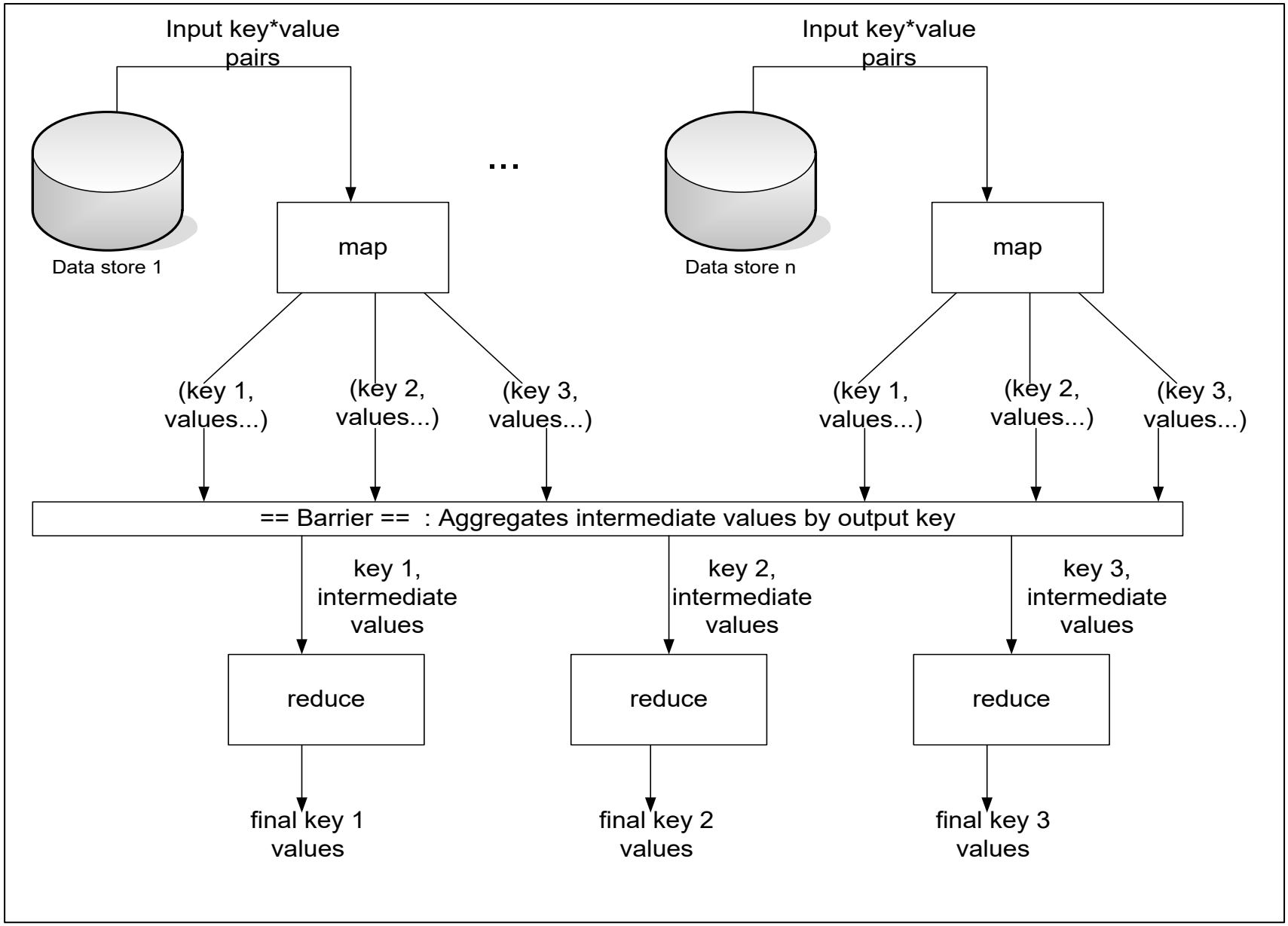
- Master data structures
  - Task status: (idle, in-progress, completed)
  - Idle tasks get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its R intermediate files, one for each reducer
  - Master pushes this info to reducers
- Master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

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

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