# COSI 129a

Introduction to Big Data Analysis

Fall 2016

Map Reduce - Implementation

# The Google Cluster

- Single Google query reads 100s MBs, consumes $10^9$s CPU cycles
- Peak workload: 1000s queries/sec
- Need for large supercomputer installations

- Google's alternative solution: 1000s of commodity PCs + fault-tolerant software

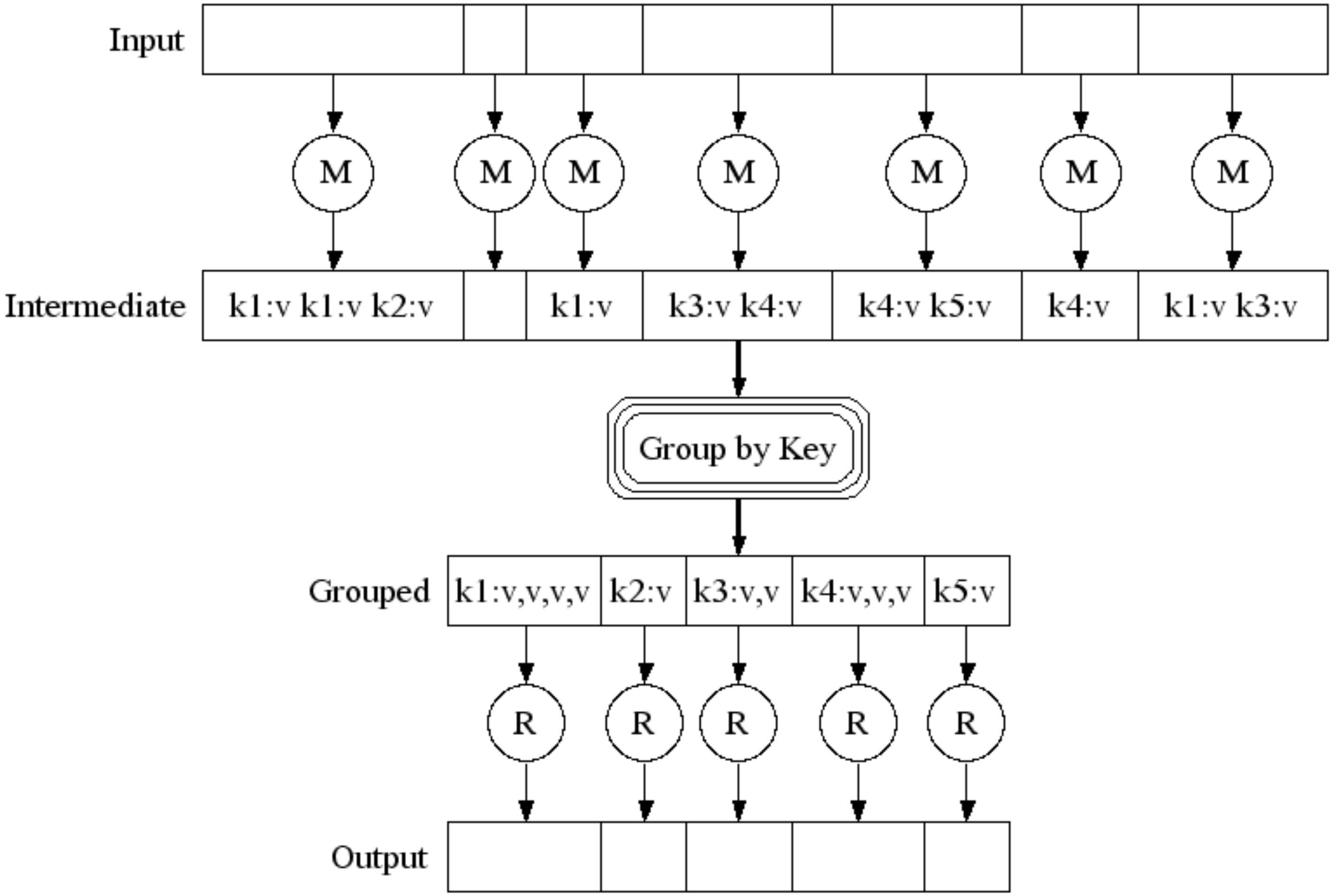
# Google Cluster Architecture Overview

- **Single-thread performance doesn't matter**
  - For large problems, **total throughput/$** is more important than peak performance per computer.
- **Stuff breaks**
  - If you have 1 server, it may stay up three years (1,000 days).
  - If you have 10,000 servers, expect to lose 10 per day.
- **"Ultra-reliable" hardware doesn't really help**
  - At large scales, the most reliable hardware still fails
    - Software still needs to be fault-tolerant
    - Commodity machines without fancy hardware give better **performance/$**

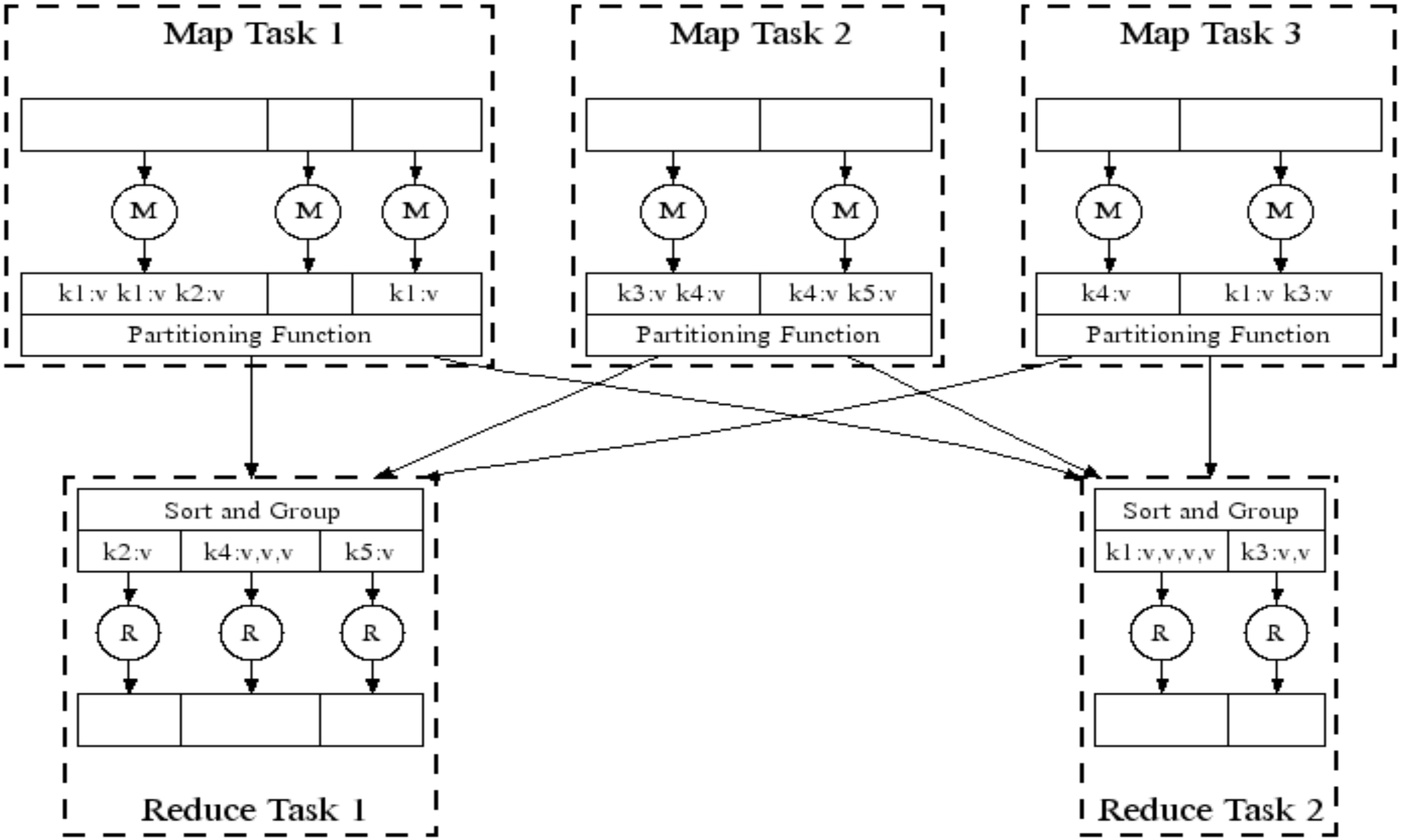
# Google Cluster Design Principles

- Have a reliable software-based computing infrastructure from clusters of unreliable commodity PCs.
- Replicate services across many machines to increase request throughput and availability.
  - Automatically detect and handle failures
- Favor price/performance over peak performance.

# Back to the MR Framework…



# Automatic Parallel Execution in MR



# Map Reduce Implementation

- Many different implementations are possible
- Today: Implementation @ Google
- Next class: Hadoop implementation

# Google Environment

- PCs with 2-core processors 2-4GB memory
- Cluster of 1000s of PCs
- Inexpensive IDE disks

- User submit jobs to a scheduler
- Job consists of set of tasks
- Tasks are assigned by the scheduler to available machines

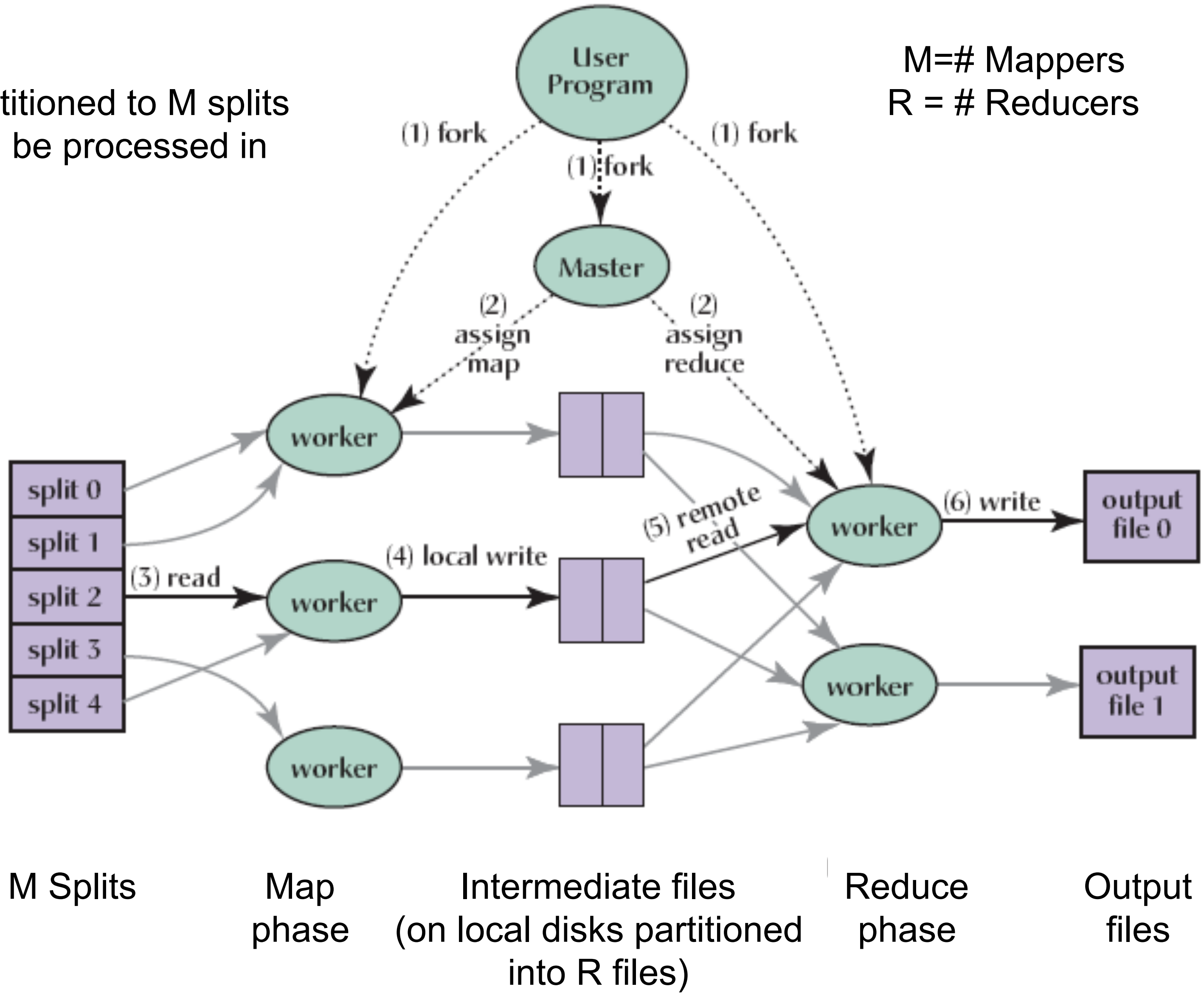
OSDI Paper on Map Reduce (2004)

# MapReduce Execution Overview

Step 0

- Input data partitioned to M splits
- Each split can be processed in parallel

M=# Mappers
R = # Reducers

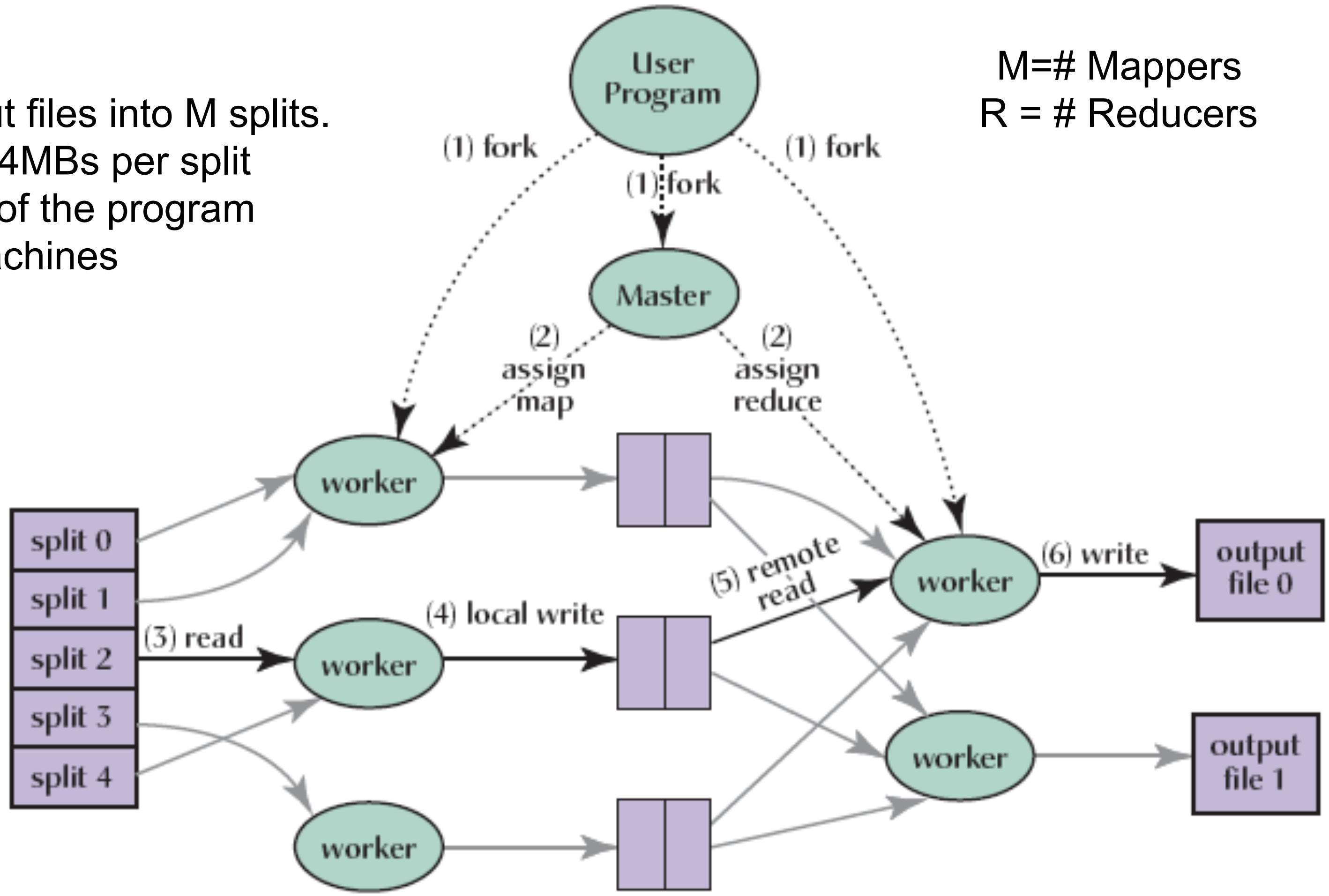


# MapReduce Execution Overview

Step 1

- MR splits input files into M splits.
- Typically 16-64MBs per split
- Starts copies of the program on cluster machines

M=# Mappers
R = # Reducers




M Splits

Map phase

Intermediate files (on local disks partitioned into R files)

Reduce phase

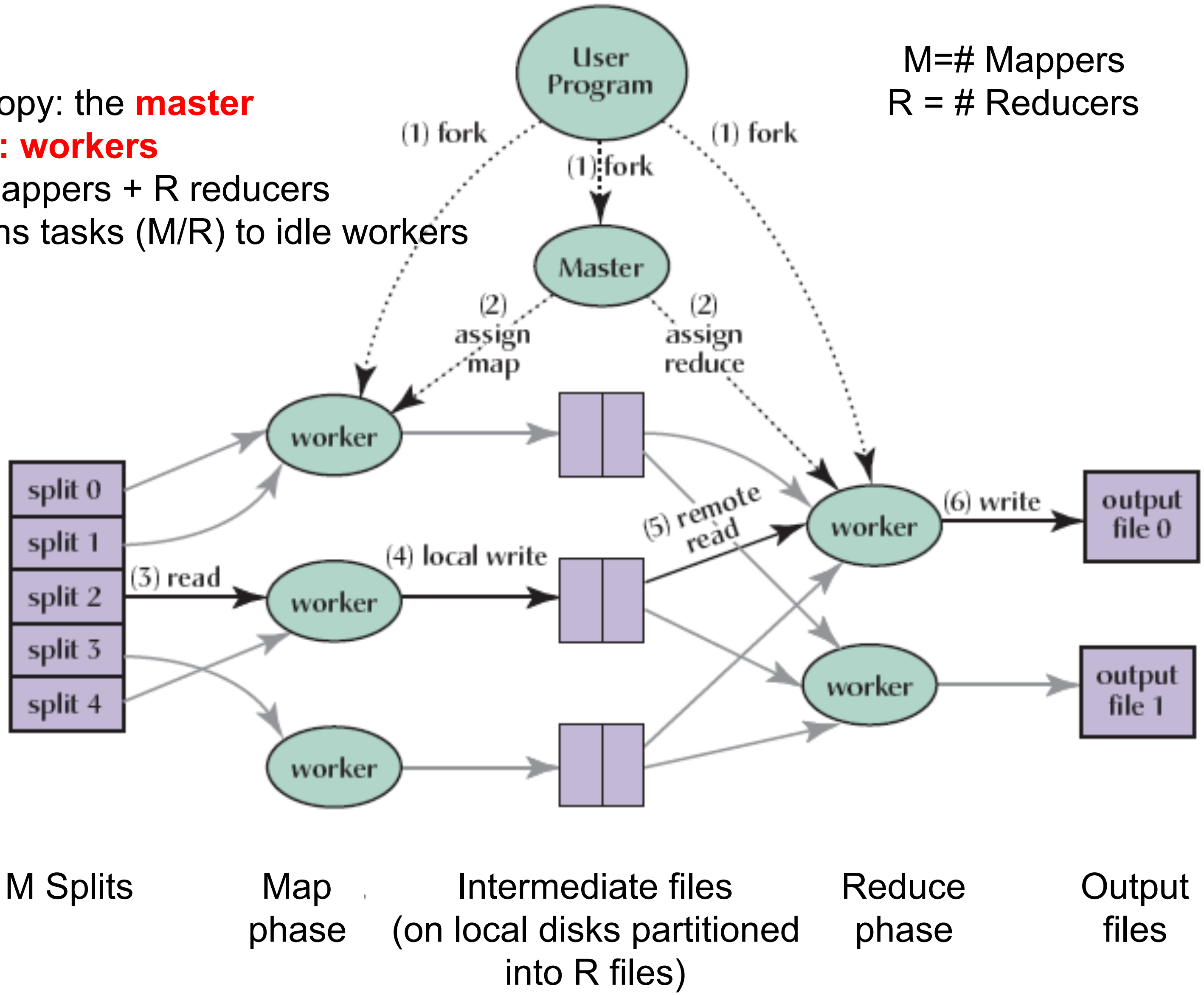
Output files

# MapReduce Execution Overview

Step 2

- One special copy: the **master**
- **Other copies: workers**
- Workers: M mappers + R reducers
- Master  assigns tasks (M/R) to idle workers

M=# Mappers
R = # Reducers

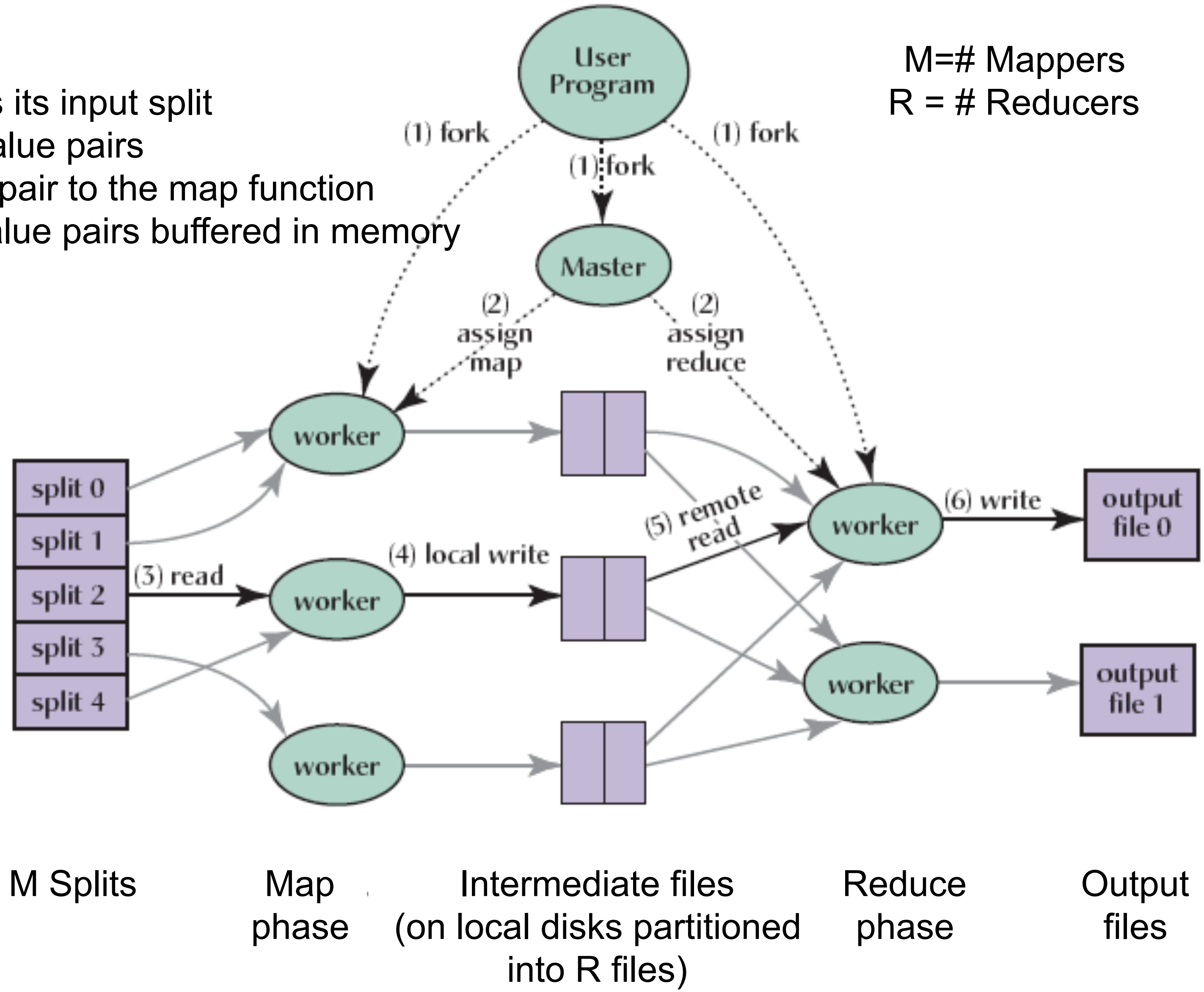


# MapReduce Execution Overview

Step 3

- Mapper reads its input split
- Parses key/value pairs
- Passes each pair to the map function
- Output key/value pairs buffered in memory

M=# Mappers
R = # Reducers

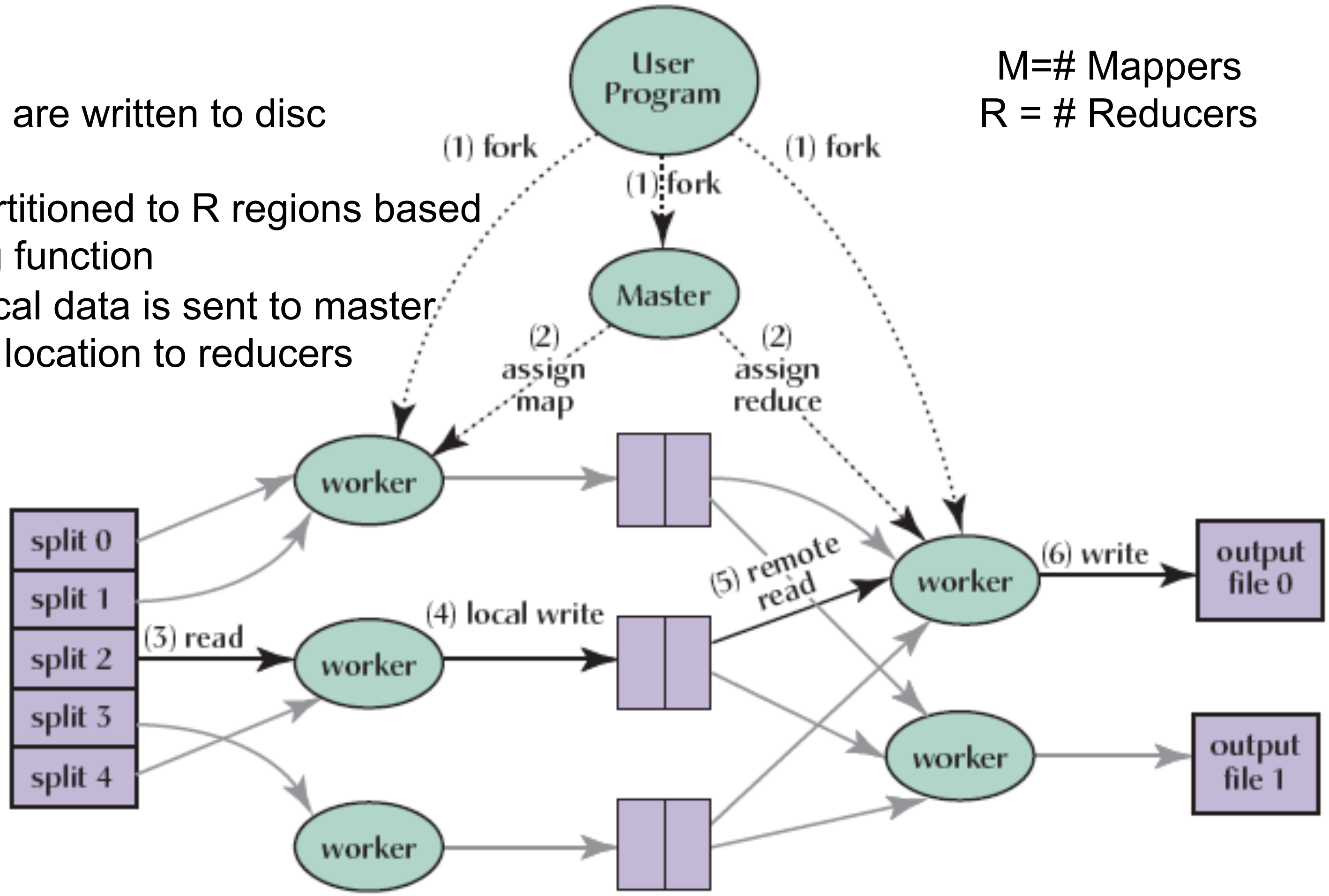


# MapReduce Execution Overview

Step 4

- Buffered pairs are written to disc (periodically)
- Local data partitioned to R regions based on partitioning function
- Location of local data is sent to master
- Master sends location to reducers

M=# Mappers
R = # Reducers




M Splits

Map phase

Intermediate files (on local disks partitioned into R files)

Reduce phase

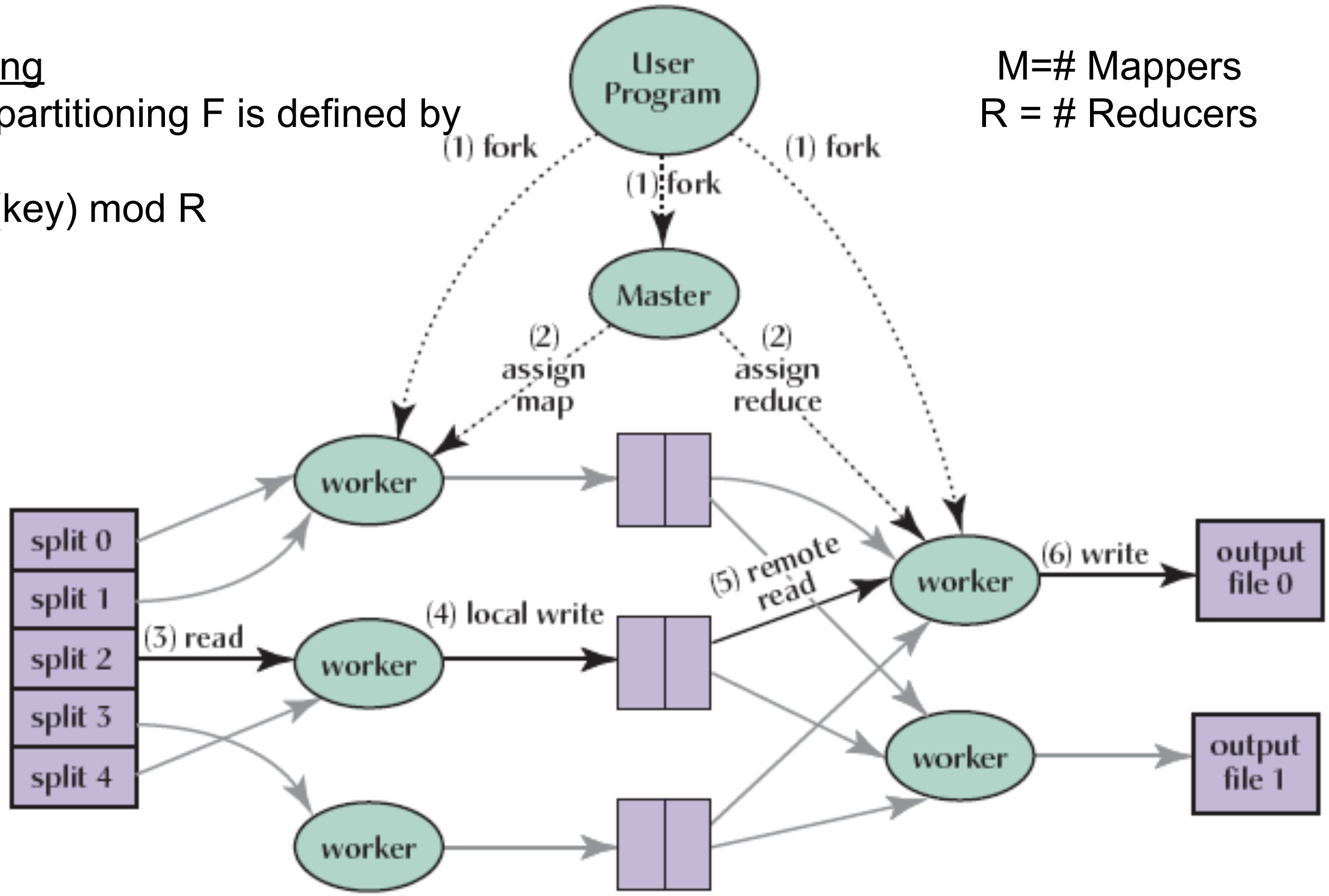
Output files

# MapReduce Execution Overview

Step 4 - partitioning

- # partitions + partitioning F is defined by user
- Default: hash(key) mod R

M=# Mappers
R = # Reducers

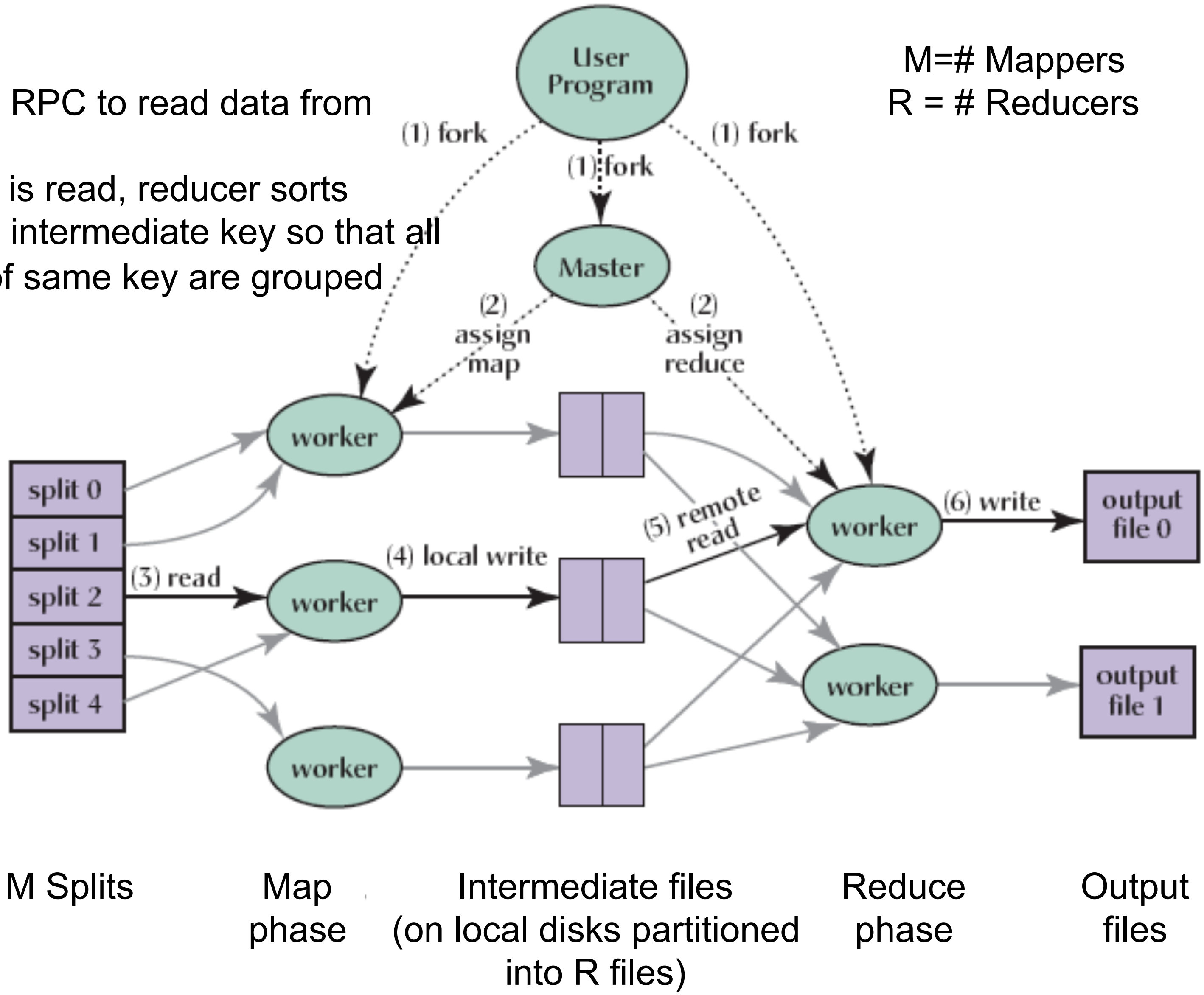



M Splits | Map phase | Intermediate files (on local disks partitioned into R files) | Reduce phase | Output files

# MapReduce Execution Overview

Step 5
- Reducers use RPC to read data from mappers
- When all data is read, reducer sorts
- Sorting by the intermediate key so that all occurrences of same key are grouped together

M=# Mappers
R = # Reducers

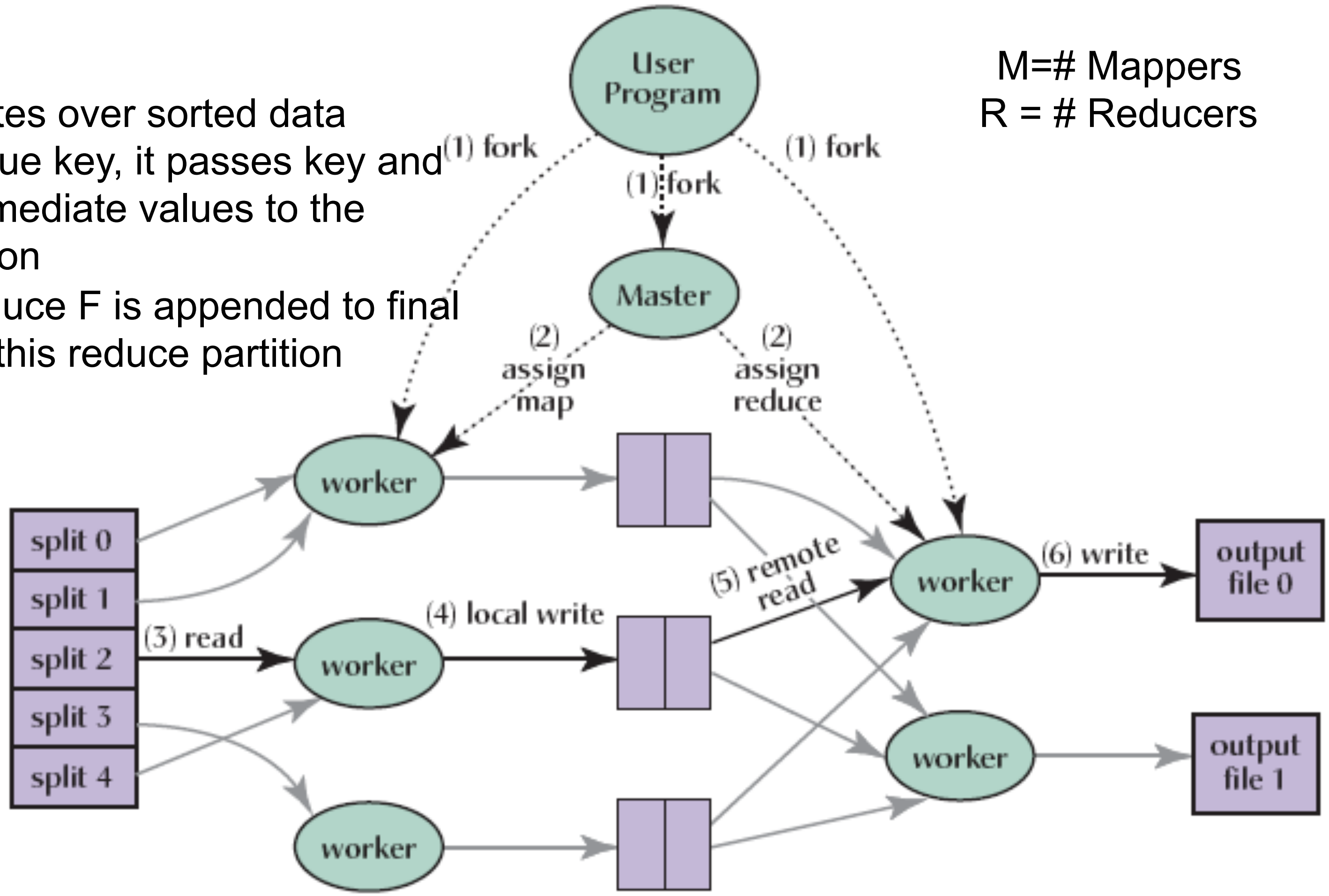


# MapReduce Execution Overview

Step 6

- Reducer iterates over sorted data
- For each unique key, it passes key and its list of intermediate values to the Reduce function
- Output of Reduce F is appended to final output file for this reduce partition

M=# Mappers
R = # Reducers




M Splits

Map phase

Intermediate files (on local disks partitioned into R files)

Reduce phase

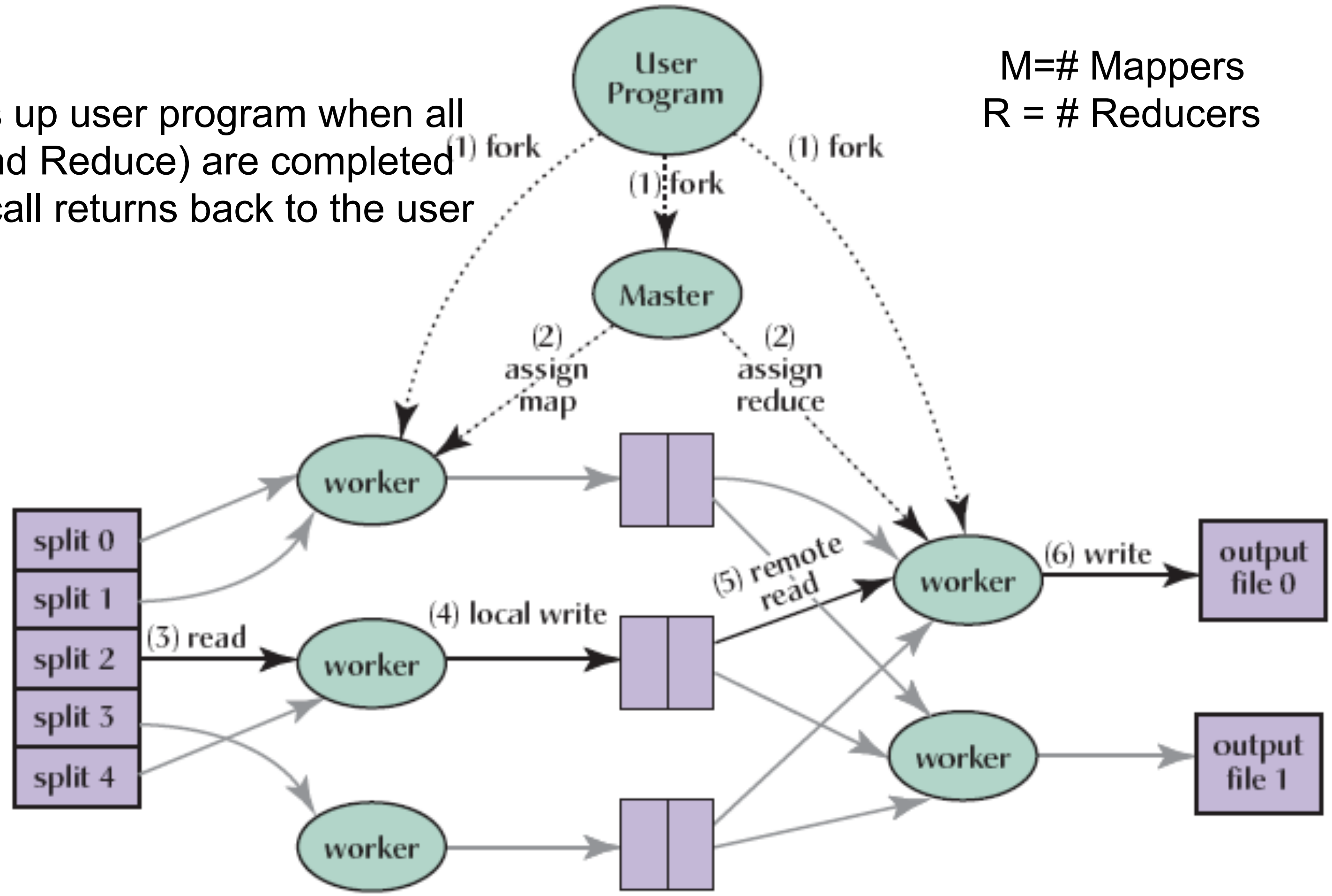
Output files

# MapReduce Execution Overview

Step 7

- Master wakes up user program when all tasks (Map and Reduce) are completed
- MapReduce call returns back to the user code

M=# Mappers
R = # Reducers




M Splits | Map phase | Intermediate files (on local disks partitioned into R files) | Reduce phase | Output files

# Output Data

- Output of the execution is available on R files
  - One files per reduce tasks
- You can combine these files into one file
- Typically users pass these files to another MR call

# MapReduce Scheduling Overview

- One master, many workers
  - Input data split into M map tasks (typically 64 MB in size)
  - Reduce phase partitioned into R reduce tasks (hash(k) mod R)
  - Tasks are assigned to workers dynamically
- Master assigns each map task to a free worker
  - Considers locality of data to worker when assigning a task
  - **Mapper reads task input (often from local disk)**
  - **Mapper produces R local files containing intermediate k/v pairs**
  - **Mapper sends the location of intermediate data to Master**
- Master assigns each reduce task to a free worker
  - **Reducer reads intermediate k/v pairs from locations of k/v pairs**
  - **Reducer sorts & applies user's reduce function to produce the output**
- **When all tasks are completed, master wakes up the user program**

# Master Data Structures

- For each map & reduce task stores its state
  - Idle, in-progress or completed
- Also stores the id of the workers that are running some tasks
- For each complete map task, stores the location and the sizes of the R intermediate files regions produced by the map task
  - Updates on location & size are sent incrementally to in-progress reducers

# MapReduce Fault Tolerance

- MR library is designed to tolerate failures gracefully
- On worker failure:
  - Master detects failure via periodic heartbeats sent to workers.
  - Mapper: Completed and in-progress map tasks on failed worker should be re-executed (→ output stored on local disk).
  - Reducer: Only in-progress reduce tasks on failed worker should be re-executed (→ output stored in global file system).
- On master failure:
  - State is check-pointed to the file system: new master recovers & continues.
- Robustness:
  - Example: Lost 1600 of 1800 machines once, but finished fine.

# MapReduce Data Locality

- Goal: To conserve network bandwidth input data are stored on the same machines the process the data (e.g., on the cluster)

- In GFS (Google File System), data files are divided into 64 MB blocks and 3 copies of each are stored on different machines.
  - HDFS (Hadoop Distributed File System) has a similar structure

- Master program schedules map() tasks based on the location of these replicas:
  - Put map() tasks physically on the same machine as one of the input replicas (or, at least on the same rack / network switch).
- This way, thousands of machines can read input at local disk speed. Otherwise, rack switches would limit read rate.

# Task Granularity

- Map phase is divided into M pieces (tasks)
- Reduce phase is divided into R pieces (tasks)
- Ideally we want M >> R
  - improves load balancing (a worker can executed many tasks)
  - speeds up recovery when a worker fails
  - Practical bounds on M,R since master keeps O(M*R) state
- R is often constrained since it produces R output files
- Practical rules
  - choose M so that each task is 64MB of input data
  - choose R to be small multiple of # machines you will use
  - Often M=200,000 and R=5,000 and # workers=2,000

# Redundant Execution

- Slow workers significantly delay completion time
  - Other jobs consuming resources on machine
  - Bad disks w/ soft errors transfer data slowly
  - Weird things: processor caches disabled (!!)
- Solution: Near end of phase, spawn backup tasks
  - Whichever one finishes first "wins"
- Dramatically shortens job completion time

# Refinement: Partitioning Function

- Partitioning function
  - Output keys of map function are partitioned to R tasks
  - Divides up key space for parallel reduce operations

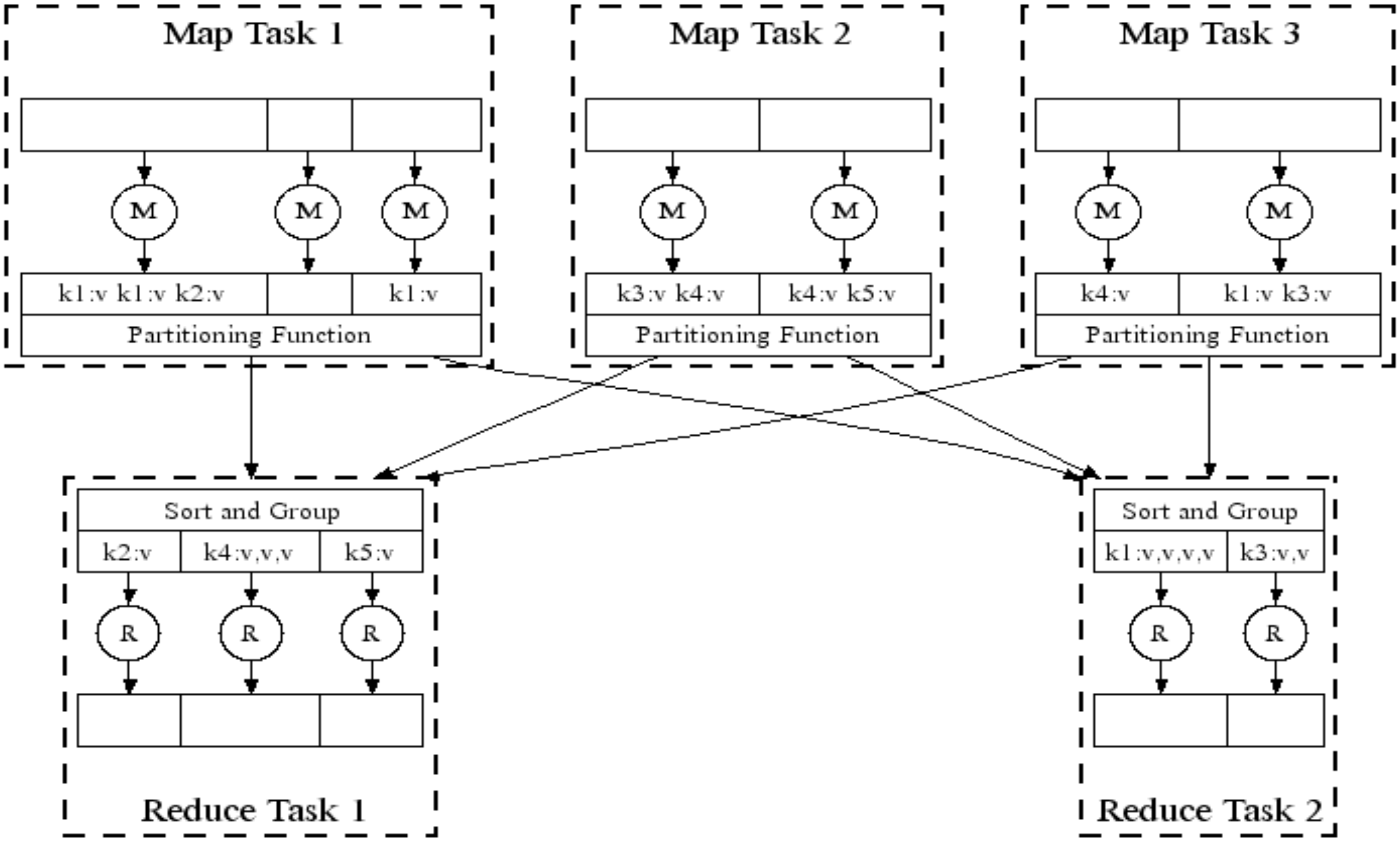
map(in_key, in_value) → list(out_key, intermediate value)

**partition (out_key, number of partitions) → partition for out_key**

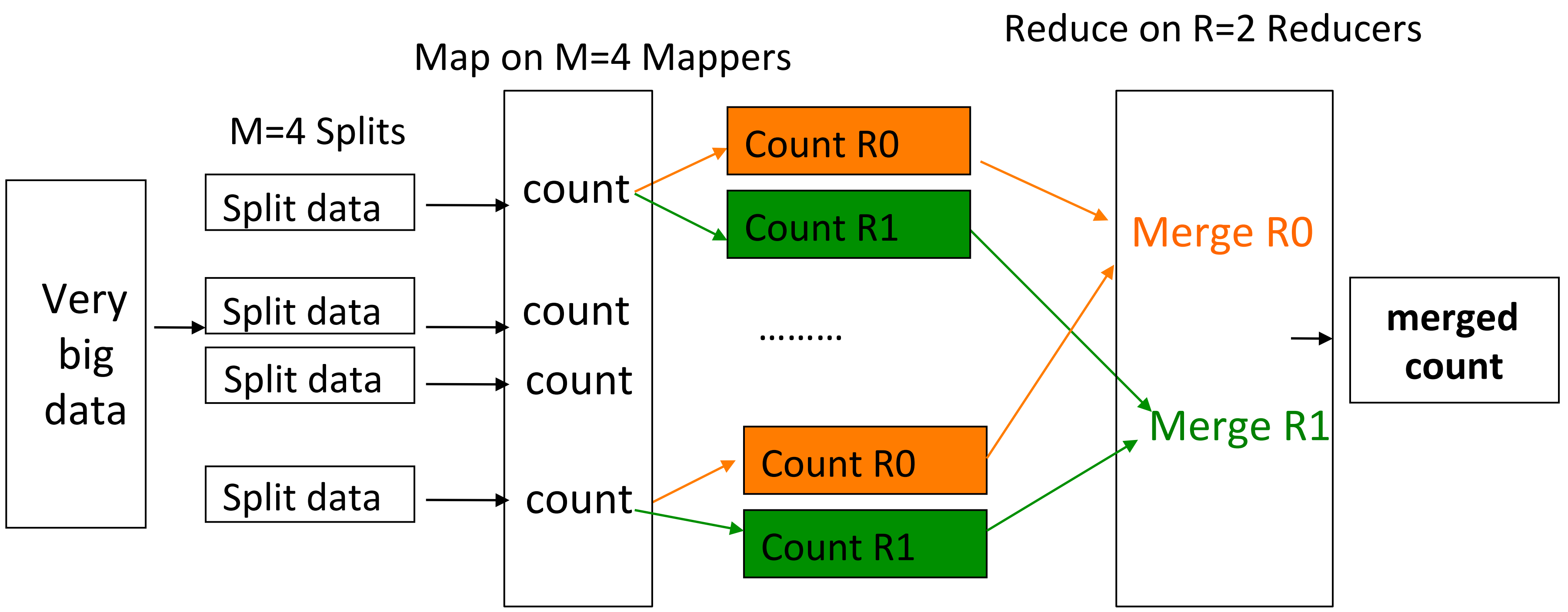
  - Often a simple hash of the key, e.g., hash(out_key) mod R

  - User can change this function:
    - hash(hostname(URL) mod R: all URLs from same host will be in the same output file

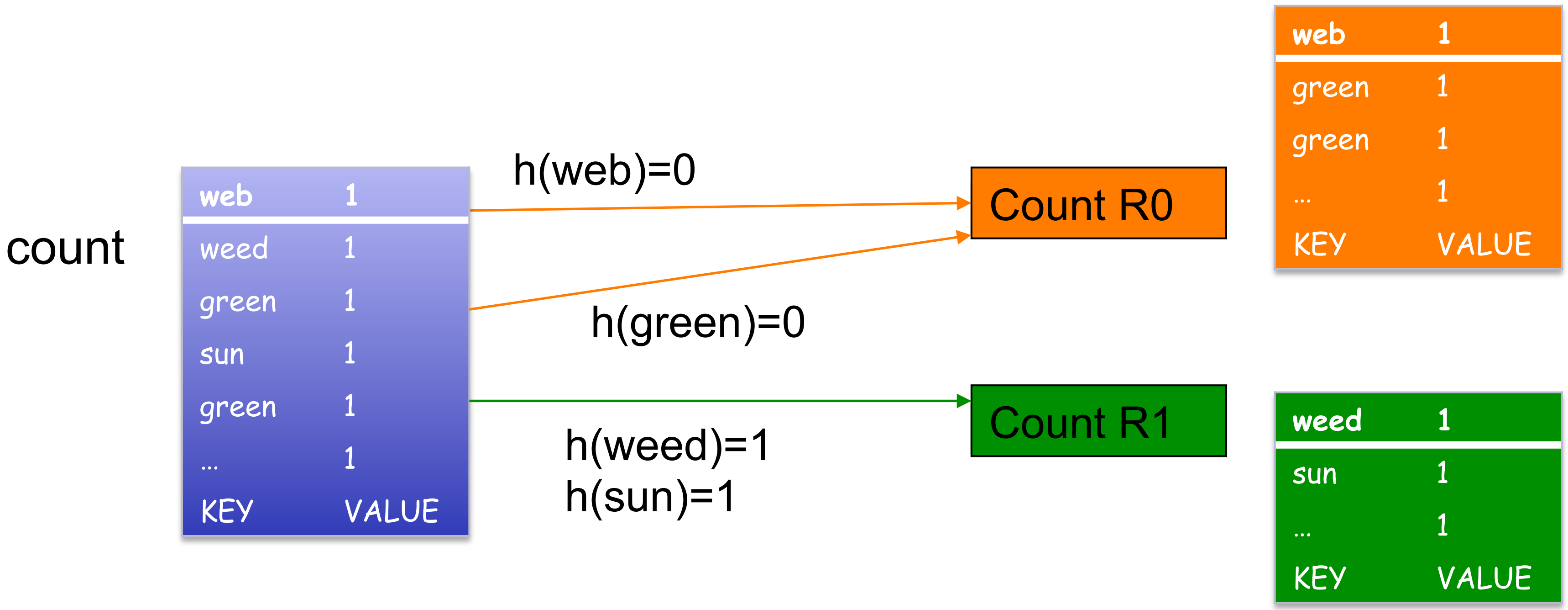
# Automatic Parallel Execution in MR



# Partitioning for Distributed Word Count



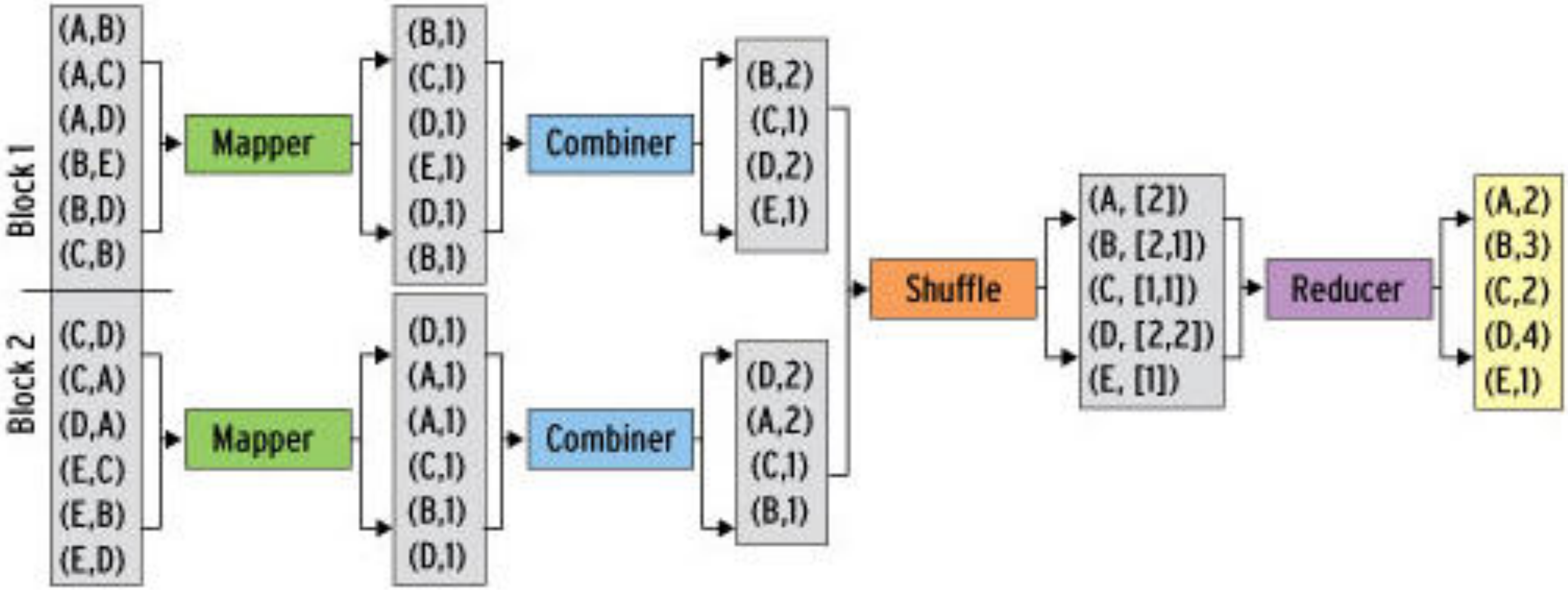
# Partitioning for Distributed Word Count



# Refinement: Combiner Function

- Mini reducers that run in memory after the map phase
  - Can be used when reduce function is commutative and associative
- Applies partial merging of map output
  - E.g. instead of sending pairs of <word, 1> sum all the “1” for the same word before you send it to the reducer
- Executes on each mapper
- Typically similar to the reduce function but outputs to an intermediate file
- It reduces network traffic and speeds up some MR operations

# MapReduce with Combiner Function



# Refinements: Input/Output Types

- MapReduce library offers support for multiple formats
  - Supports predefined and user-defined input/output formats
  - Each input type implementation knows how to split the data to separate map tasks
- E.g., “Text”input: each line is a (key,value) pair
  - Key: offset of the file
  - Value: contents of the line
- E.g., read tuples from a database
- Each input type implementation should know how to split itself into meaningful ranges to generate splits for the map tasks

# Refinement: Skipping Bad Records

- Map/Reduce functions sometimes fail for particular inputs
  - Best solution is to debug & fix
    - Not always possible ~ third-party source libraries
  - On segmentation fault:
    - Send UDP packet to master from signal handler
    - Include sequence number of record being processed
  - If master sees two failures for same record:
    - Next worker is told to skip the record

# Other Refinements

- Sorting guarantees
  - Within each reduce partition keys are sorted
- Local execution for debugging/testing
  - Sequentially executes all the work of the MR operation on the local machine
- User-defined counters in map/reduce functions
  - E.g., count # words processed, # of documents indexed,..
  - Master aggregate counter values from different mapper/ reducers

# Performance

Tests run on cluster of 1800 machines:

- 4 GB of memory
- Dual-processor 2 GHz Xeons with Hyperthreading
- Dual 160 GB IDE disks
- Gigabit Ethernet per machine
- Bisection bandwidth approximately 100 Gbps

Two benchmarks:

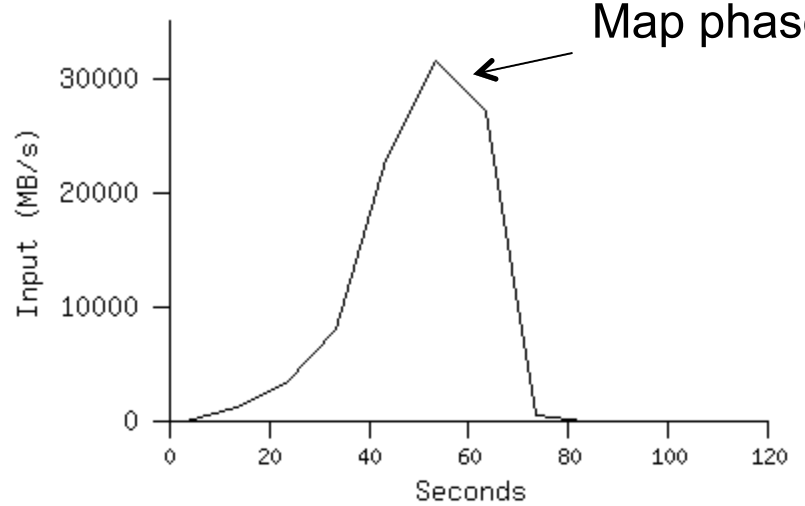
MR_GrepScan $10^{10}$ 100-byte records to extract records matching a rare pattern (92K matching records)

MR_SortSort $10^{10}$ 100-byte

# Grep

Map phase finishes

Data scanning rate: It increases as we increase # machines

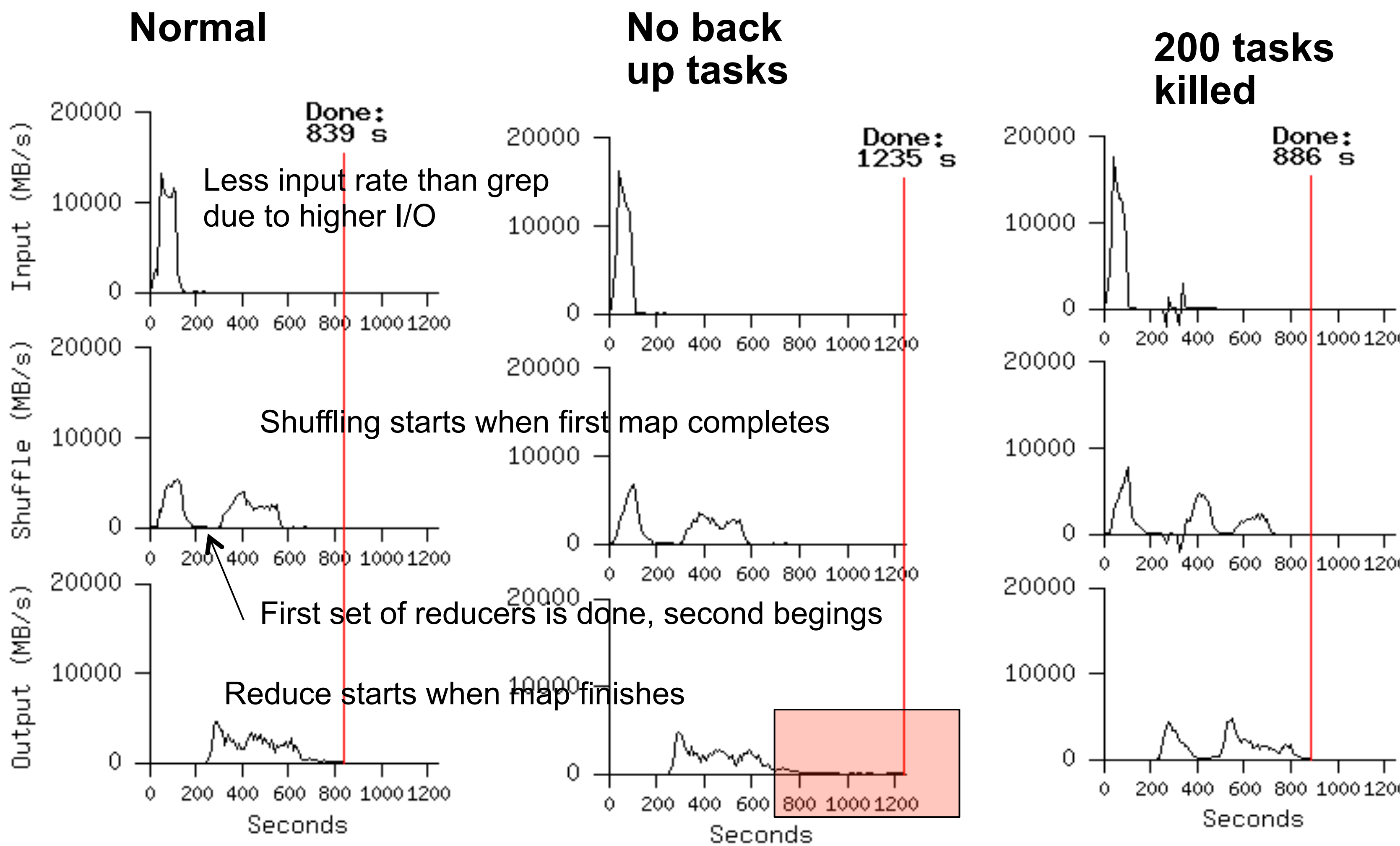



Locality optimization helps:

- 1800 machines read 1 TB at peak ~31 GB/s (@1,700 workers)
- W/out this, rack switches would limit to 10 GB/s

- Startup overhead is significant for short jobs: about a minute of startup overhead (~150secs for the whole job)
  - Overhead due to propagating programs to workers, opening 100 input files, getting locality information

# Sort




- Backup tasks reduce job completion time a lot!
- System deals well with failures

# MapReduce Summary

- MapReduce has proven to be a useful abstraction.
  - greatly simplifies large-scale computations at Google
- Functional programming paradigm can be applied to large-scale applications.
- MapReduce seems easy to use.
  - focus on the problem, let the MapReduce library deal with any messy details