CS 789 ADVANCED BIG DATA ANALYTICS

BIG DATA AND MAP REDUCE

* The contents are adapted from Dr. Chengkai Li at UT Arlington, Jeff Dean, Sanjay Ghemawat, Anand Rajaraman

Mingon Kang, Ph.D.

Department of Computer Science, University of Nevada, Las Vegas

Map? Reduce?

Higher-order function in functional programming languages.

Example: Scheme (variant of LISP)

- (map square ‘(1 2 3))
  - (1 4 9)
- (reduce + (map square ‘(1 2 3)))
  - 14

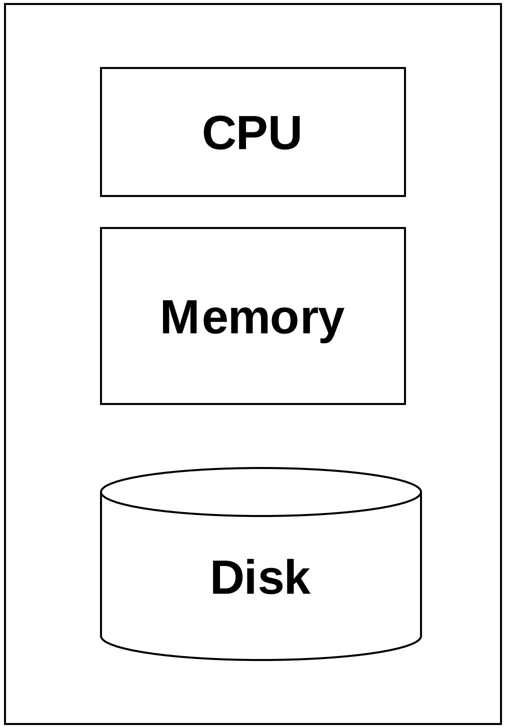
Motivation: Large Scale Data Processing

- Many tasks:

  Process lots of data to produce other data
- Want to use hundreds or thousands of CPUs

  ... but this needs to be easy
- MapReduce provides:
  - Automatic parallelization and distribution
  - Fault-tolerance
  - I/O scheduling
  - Status and monitoring

Single-node architecture



Machine Learning, Statistics

"Classical" Data Mining

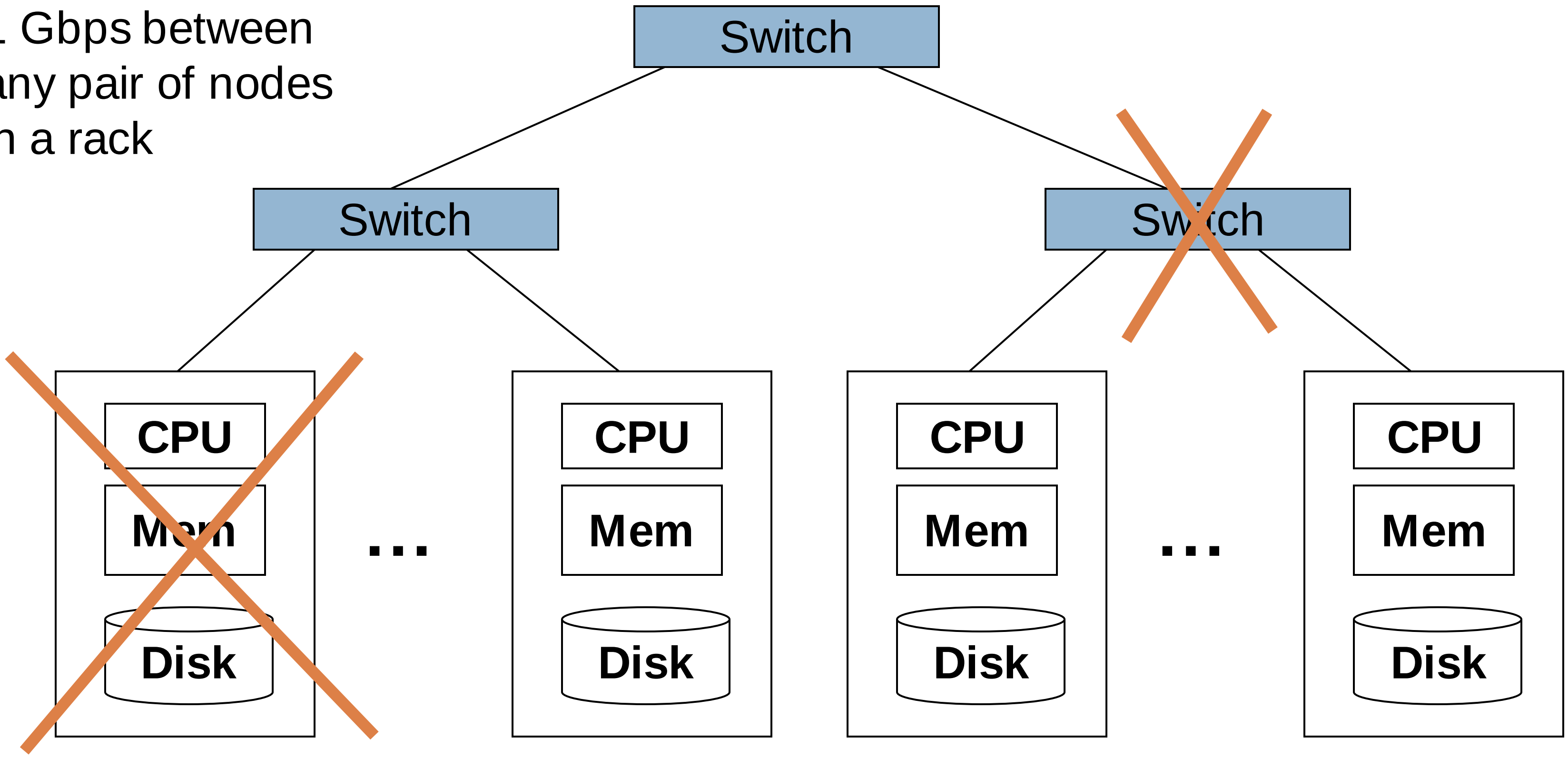
Commodity Clusters

- Web data sets can be very large
  - Tens to hundreds of terabytes
- Cannot mine on a single server (why?)
- Standard architecture emerging:
  - Cluster of commodity Linux nodes
  - Gigabit ethernet interconnect
- How to organize computations on this architecture?
  - Mask issues such as hardware failure

Cluster Architecture

2-10 Gbps backbone between racks

1 Gbps between
any pair of nodes
in a rack



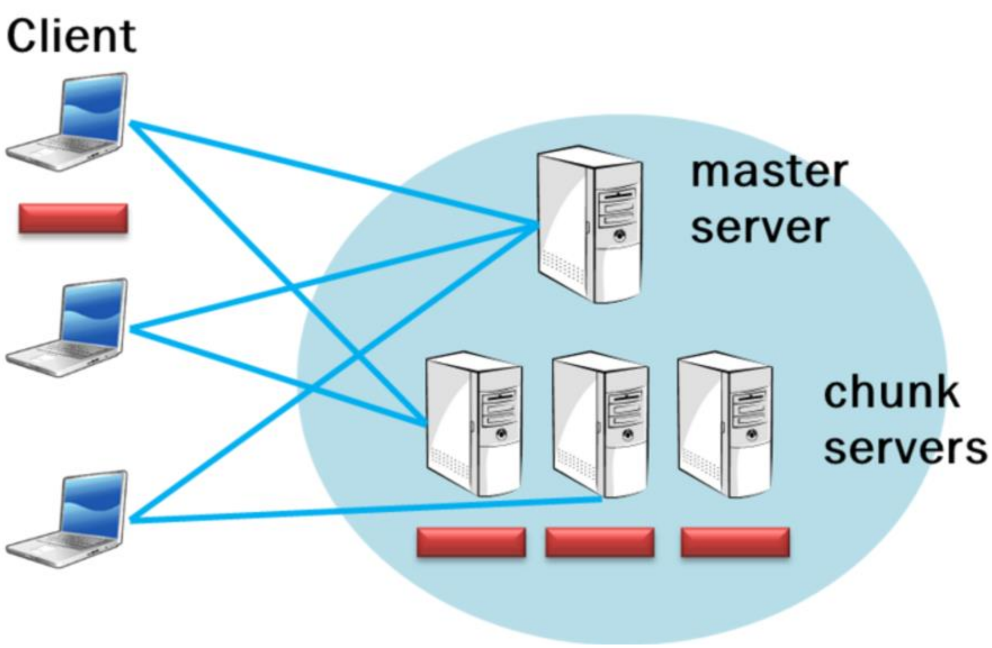
Each rack contains 16-64 nodes

Stable storage

- First order problem: if nodes can fail, how can we store data persistently?
- Answer: Distributed File System
  - Provides global file namespace
  - Google GFS; Hadoop HDFS; Kosmix KFS
- Typical usage pattern
  - Huge files (100s of GB to TB)
  - Data is rarely updated in place
  - Reads and appends are common

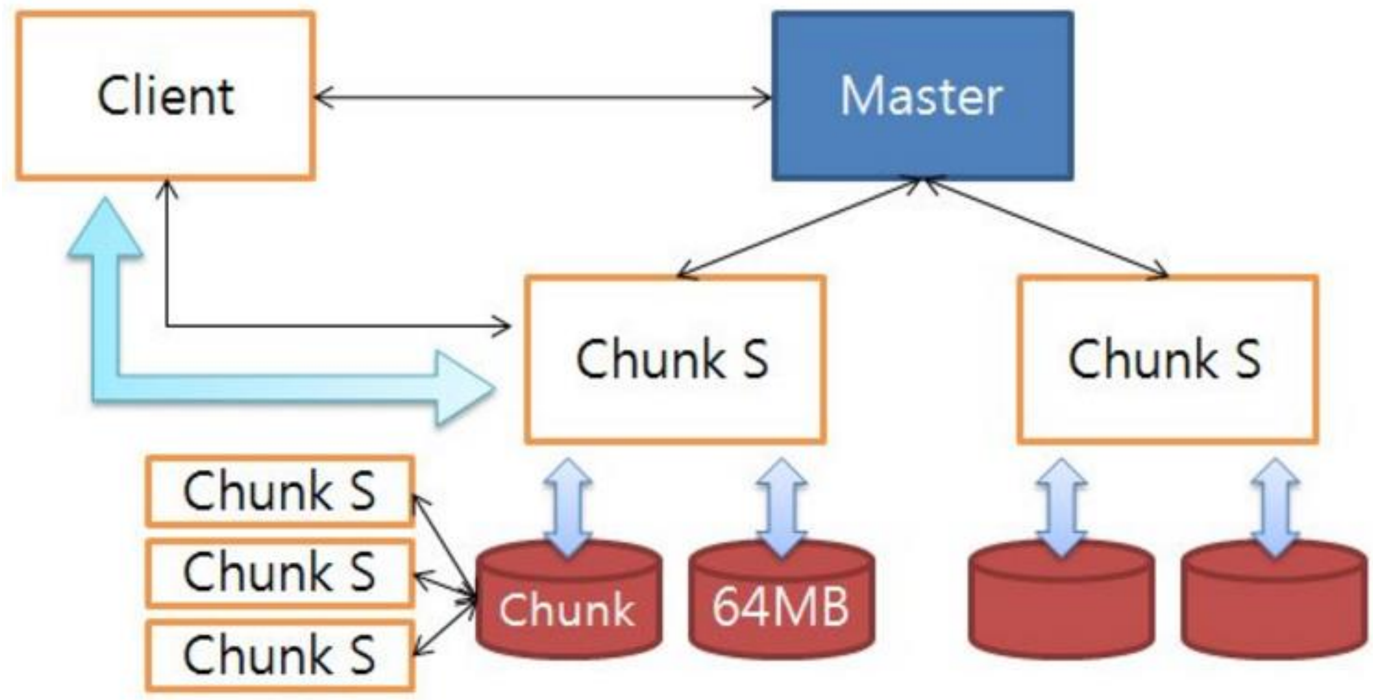
Google File System

- Distribute File System
  - Master: control tower that monitors GFS's status and manages
  - Chunk Server: physical I/O operations
  - Client: request I/O operations



Google File System

- A client requests I/O operations
- Master replies the information of the chunk server which is nearest to the client
- Client communicates with the chunk server directly for I/O operations



Google File System

- Fault-tolerance
  - If a chunk server fails
    - Master uses other available chunk server
  - If master server fails
    - There is another device that monitors master server
    - Master will be replaced with others

Warm up: Word Count

- We have a large file of words, one word to a line
- Count the number of times each distinct word appears in the file
  - `sort datafile | uniq -c`

- Sample application: analyze web server logs to find popular URLs

Word Count (2)

- Case 1: Entire file fits in memory
- Case 2: File too large for mem, but all <word, count> pairs fit in mem
- Case 3: File on disk, too many distinct words to fit in memory

Word Count (3)

- To make it slightly harder, suppose we have a large corpus of documents
- Count the number of times each distinct word occurs in the corpus
  - ```
    cat datafile | sed -r 's/[[:space:]]+/\n/g' | sed '/^$/d' | sort | uniq -c
    ```
- The above captures the essence of MapReduce
  - Great thing is it is naturally parallelizable
    ```

Example: Spam Collection

- Count words in spam
  - https://www.kaggle.com/uciml/sms-spam-collection-dataset#spam.csv

Programming model

- Input & Output: each a set of key/value pairs
- Programmer specifies two functions:

```
map (in_key, in_value) -> list(out_key, intermediate_value)
```

  - Processes input key/value pair
  - Produces set of intermediate pairs

```
reduce (out_key, list(intermediate_value)) -> list(out_value)
```

  - Combines all intermediate values for a particular key
  - Produces a set of merged output values (usually just one)
- Inspired by similar primitives in LISP and other languages

MapReduce

- Input: a set of key/value pairs
- User supplies two functions:
  - map(k1,v1) → list(k2,v2)
  - reduce(k2, list(v2)) → list(v3)
- (k2,v2) is an intermediate key/value pair
- Output: for each k2, the output is a list of (k2, v3) pairs.
  - usually just one value or empty.
  - k2 is omitted since it is pre-determined based on the input

Word Count using MapReduce

```
map(key, value):
// key: document name; value: text of document
    for each word w in value:
        emit(w, 1)


reduce(key, values):
// key: a word; values: an iterator over counts
        result = 0
        for each count v in values:
                result += v
        emit(result)
```

Two blocks of the input file

#iblock 1
1 Algorithm design with MapReduce
2 MapReduce Algorithm

Computing node 1: Invoke map function on each key value pair

#iblock 2
1 MapReduce Algorithm implementattion
2 Hadoop implmentation of MapReduce

Computing node 2: Invoke map function on each key value pair

(algorithm, 1), (design, 1), (with, 1), (MapReduce, 1) (MapReduce, 1), (algorithm, 1), (implementation, 1)
(MapReduce, 1), (algorithm, 1) (Hadoop, 1), (implementation, 1), (of, 1), (MapReduce, 1)

Shuffle and Sort

(algorithm, [1, 1, 1]), (desgin, [1]), (with, [1]), (MapReduce, [1, 1, 1, 1]), (implementation, [1, 1]), (Hadoop, [1], (of, [1])

(algorithm, [1, 1, 1]), (desgin, [1]), (Hadoop, [1])

Computing node 3 – Reducer 1: Invoke reduce function on each pair

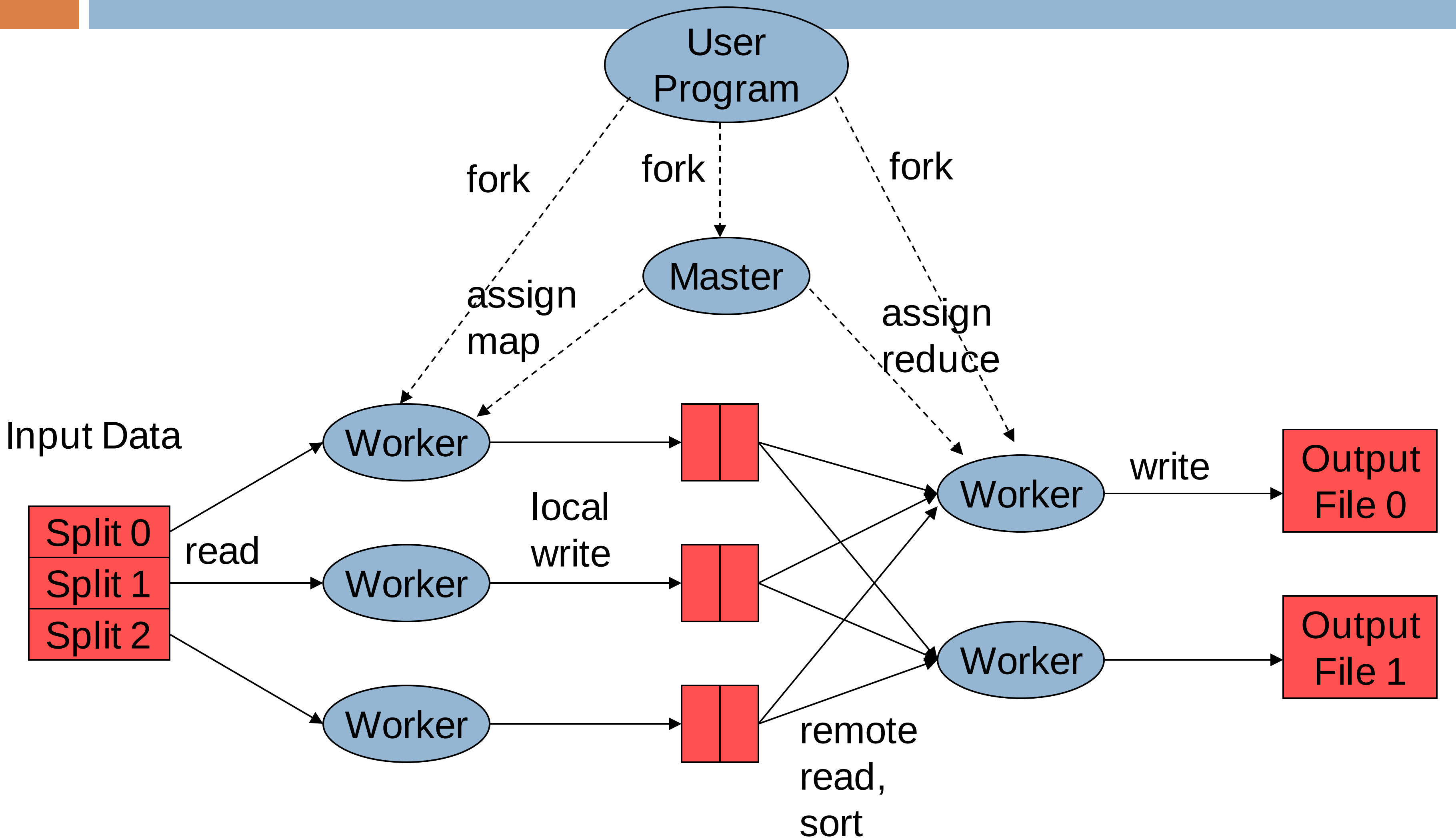
(algorithm, 3), (design, 1), (Hadoop, 1)

(implementation, [1, 1]), (MapReduce, [1, 1, 1, 1]), (of, [1]), (with, [1])

Computing node 4 – Reducer 2: : Invoke reduce function on each pair

(implementation, 2), (MapReduce, 4), (of, 1), (with, 1)

Distributed Execution Overview



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
    - **R: the number of reducers.**
  - 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

Combiners

- Often a map task will produce many pairs of the form (k,v1), (k,v2), … for the same key k
  - E.g., popular words in Word Count

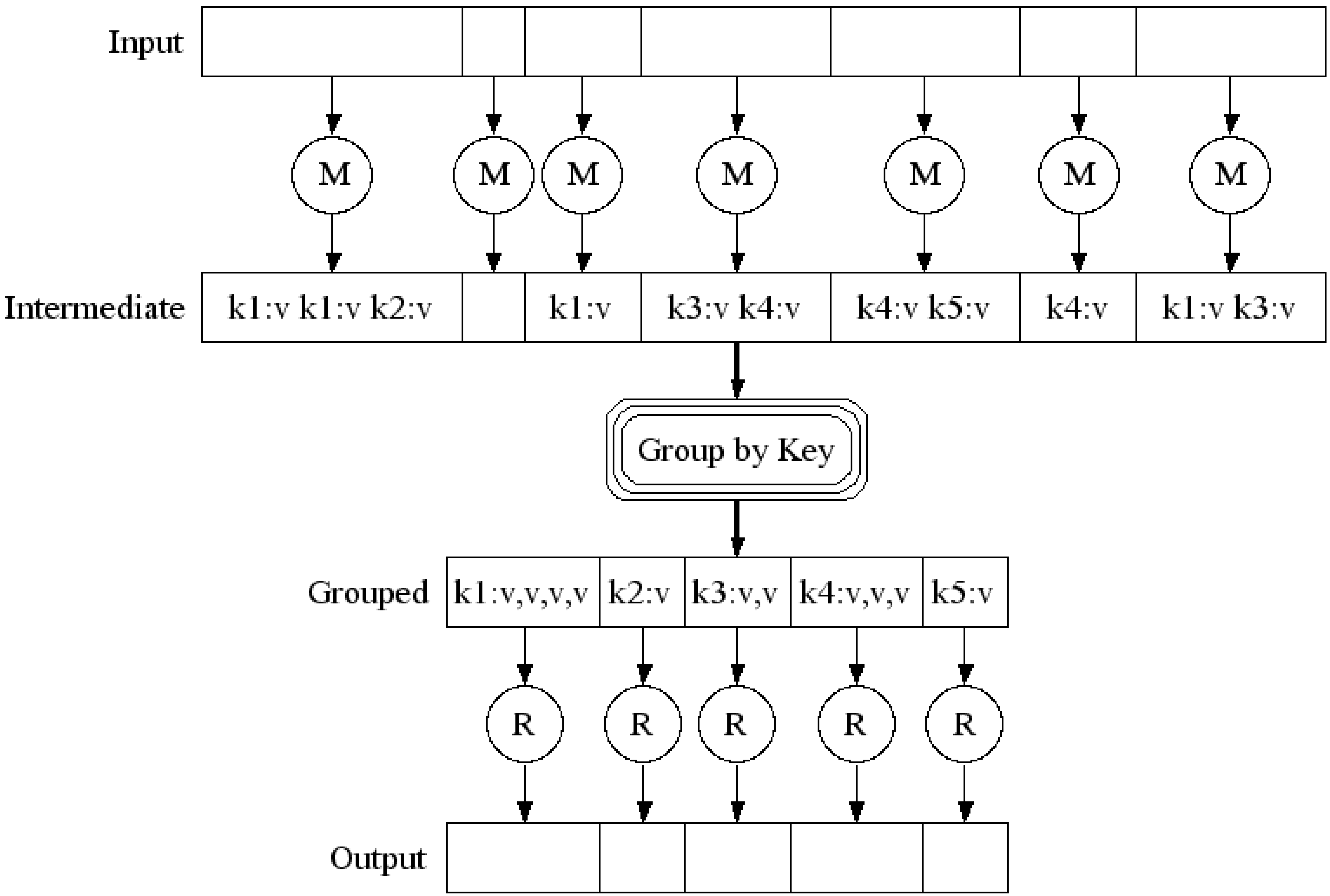
- Can save network time by pre-aggregating at mapper
  - combine(k1, list(v1)) → v2
  - Usually same as reduce function

- Works only if reduce function is commutative and associative

Partition Function

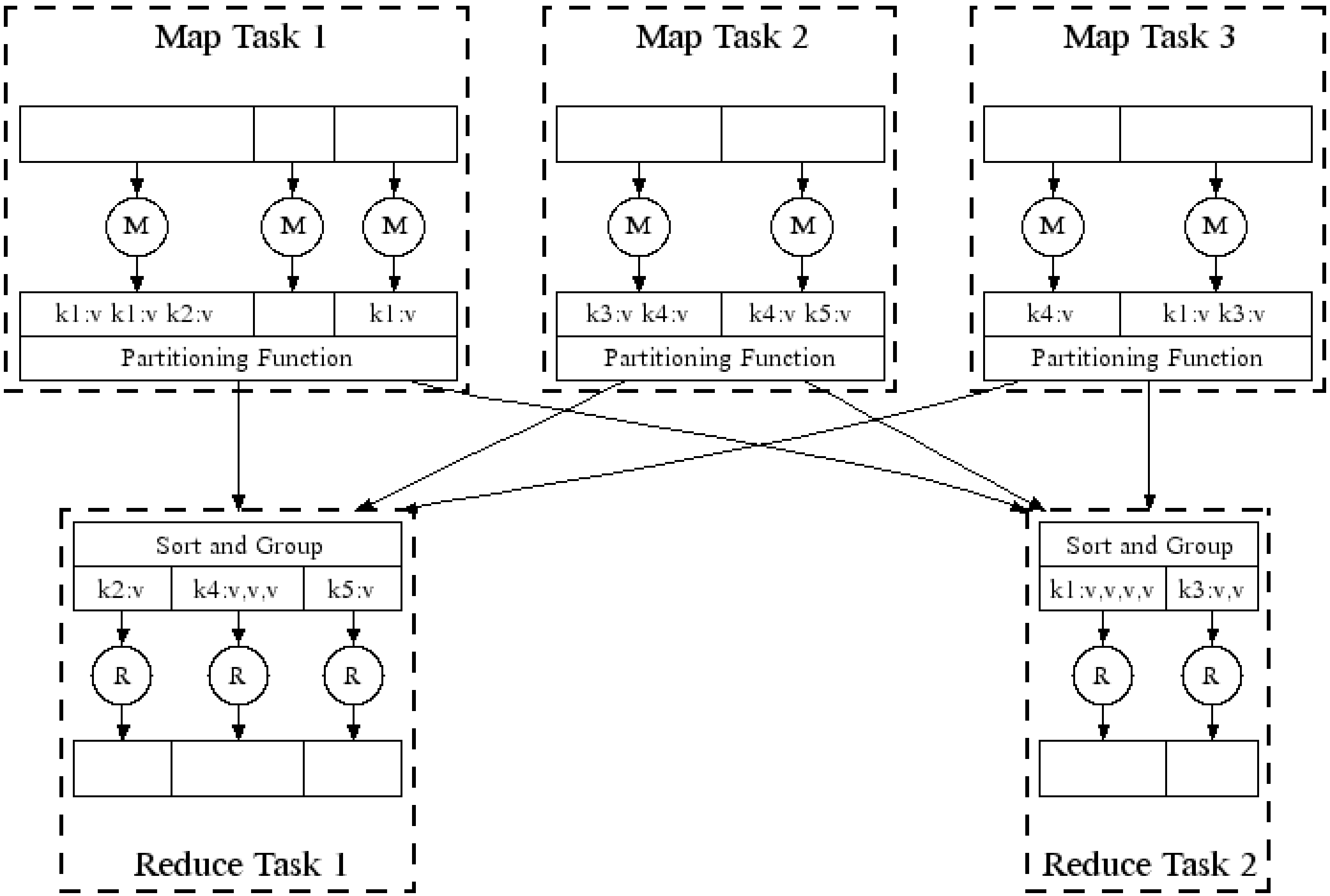
- Inputs to map tasks are created by contiguous splits of input file
- For reduce, we need to ensure that records with the same intermediate key end up at the same worker
- System uses a default partition function e.g., hash(key) mod R
- Sometimes useful to override
  - E.g., hash(hostname(URL)) mod R ensures URLs from a host end up in the same output file

Execution



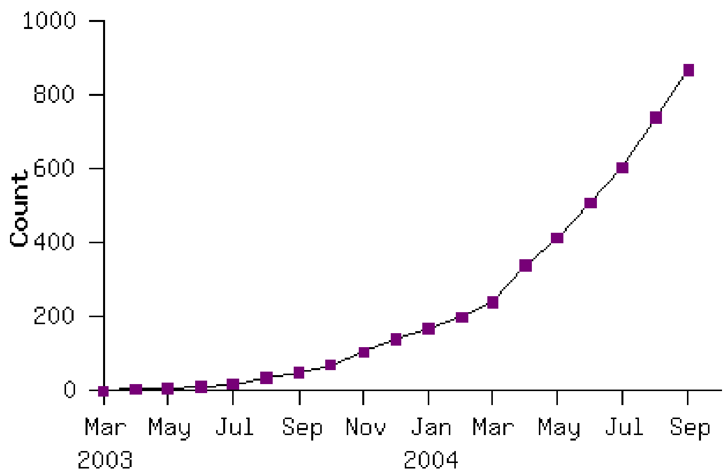
Input
M
M
M
M
M
M
M
Intermediate
k1:v k1:v k2:v
k1:v
k3:v k4:v
k4:v k5:v
k4:v
k1:v k3:v
Group by Key
Grouped
k1:v,v,v,v
k2:v
k3:v,v
k4:v,v,v
k5:v
R
R
R
R
R
Output

Parallel Execution



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
...	...	...

Exercise 1: Host size

- Suppose we have a large web corpus
- Let's look at the metadata file
  - Lines of the form (URL, size, date, …)
- For each host, find the total number of bytes
  - i.e., the sum of the page sizes for all URLs from that host

- Map (key= position, value = “URL, size, data, …”)

  foreach hostname URL

  emit(*hostname, size*)

- Reduce(key = *hostname*, value = *size*)

```
totalsize = 0
for each size v in sizes:
    totalsize += v
emit(hostname, totalsize)
```

Exercise 2: Graph reversal

- Given a directed graph as an adjacency list:

src1: dest11, dest12, …

src2: dest21, dest22, …

- Construct the graph in which all the links are reversed

- Map (key= filename, value = file content)

```
foreach line <src : destination list>
    foreach dest in destination list
        emit(dest, src)
```

- Reduce(key = *node*, value = *rev_src*)

```
    String concat = node + “ : ”
    foreach n in rev_src
        concat += n + “ ”
    emit (concat)
```

Exercise 4: Frequent Pairs

- Given a large set of market baskets, find all frequent pairs
  - Data: Basket1, Item11, Item12, …
- A lot of transaction files
- Each line of a transaction file is a list of items
- Threshold = t

- Map(key= marketbasket file, value=content)

```
foreach line=item_1, …., item_n in content
    for i=1; i<n; i++
        for j=i+1; j<=n; j++
            emit(<item_i, item_j>, 1)
```

- Reduce(key= <item_i, item_j>, value = counts)

```
total = 0
foreach count in counts
        total += count
if (total >= t)  emit(total)
```

Exercise 5: Incoming Links

Given a set of HTML pages, compute the number of incoming hyperlinks for each URL. For example, suppose the a HTML file appears in 3 pages: 3 times in page A, 3 times in page B, and 4 times in page C. Then its number of incoming hyper-links is 10.

Hadoop

- An open-source implementation of Map Reduce in Java
  - Uses HDFS for stable storage
- Download from:

http://lucene.apache.org/hadoop/