CE 528 Cloud Computing

Lecture 5: MapReduce
Spring 2026

Prof. Yigong Hu

BOSTON UNIVERSITY

Slides courtesy of Chang Lou and Alan Liu

Administrivia

Please fill out the survey to submit your AWS account ID

- 12-digit number
- Link is in Piazza

Please use the link on Piazza to obtain your Google Cloud credits.

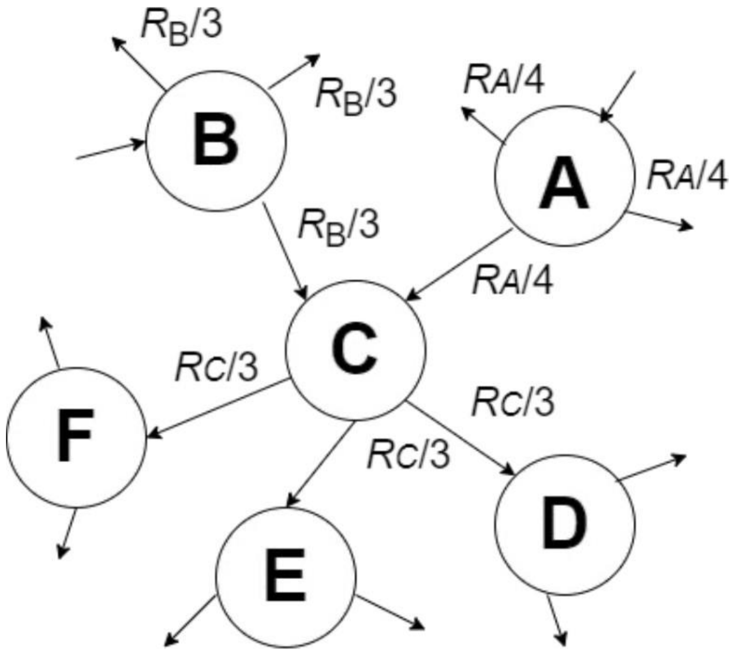
Please check and accept the invitation to the GitHub organization.

Large-Scale Analytics

- WordCount: Compute the frequency of words in a corpus of documents.

- Count how many times users have clicked on each of a (large) set of ads.

- PageRank: Compute the "importance" of a web page based on the "importances" of the pages that link to it



Option 1: SQL

Before MapReduce, analytics mostly done in SQL, or manually.

Example: Count word appearances in a corpus of documents.

With SQL, the rough query might be:

```
SELECT COUNT(*) FROM (
     SELECT UNNEST(string_to_array(doc_content, ‘ ’)) as
     word FROM Corpus)
 GROUP BY word
```

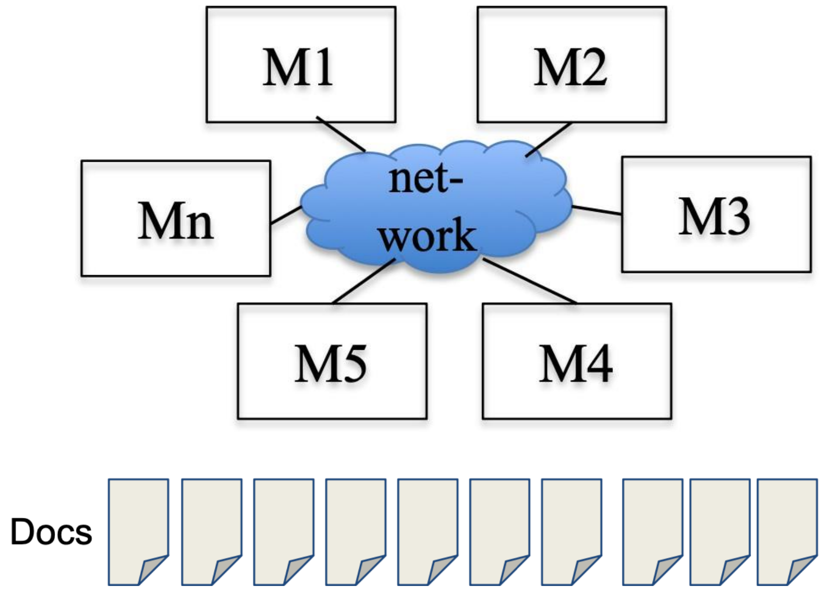
Very expressive, convenient to program

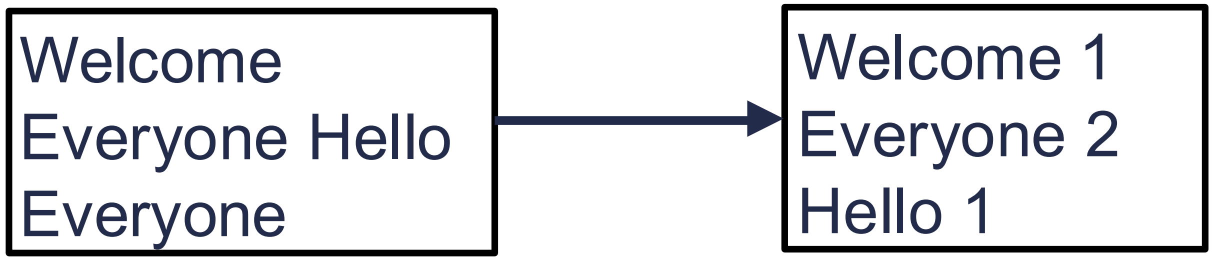
But no one knew how to scale SQL execution!

Option 2: Manual

Example: Count word appearances in a corpus of documents.

In real worlds, you are given a huge dataset (e.g., Wikipedia dump or all of Shakespeare's works)





Welcome
Everyone Hello
Everyone

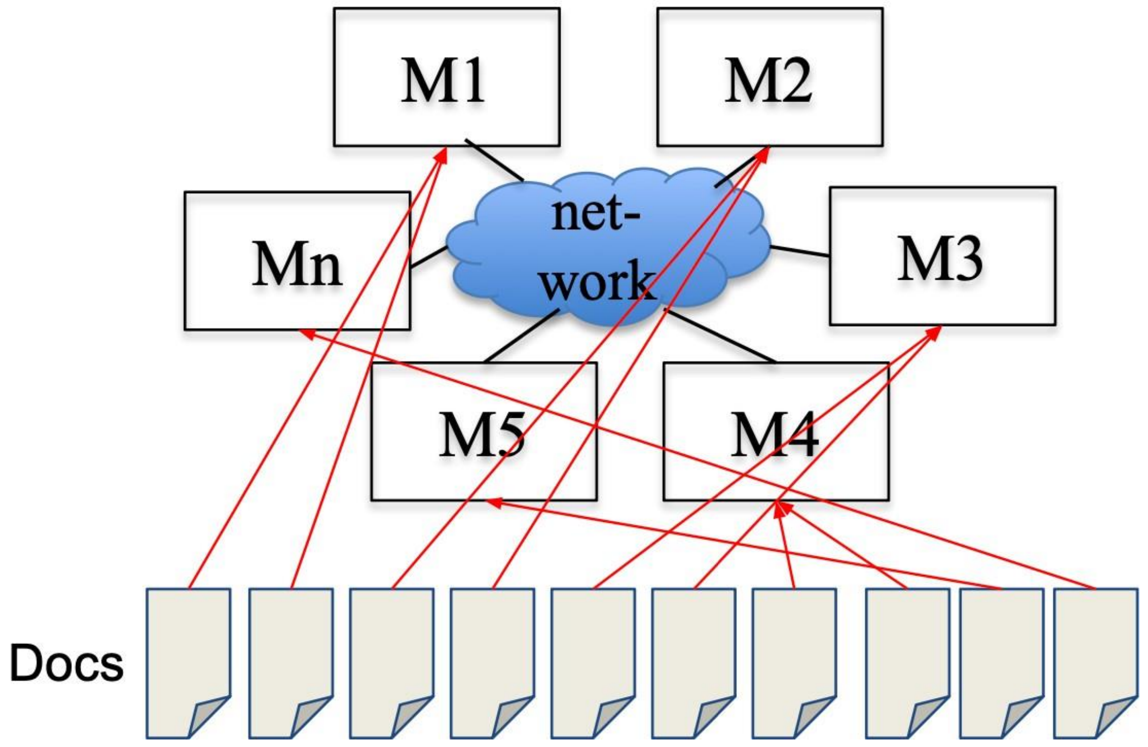
Welcome 1
Everyone 2
Hello 1

Option 2: Manual

Example: Count word appearances in a corpus of documents.

Phase 1: Assign documents to different machines/nodes.

- Each computes a dictionary: {word: local_freq}.



Option 2: Manual

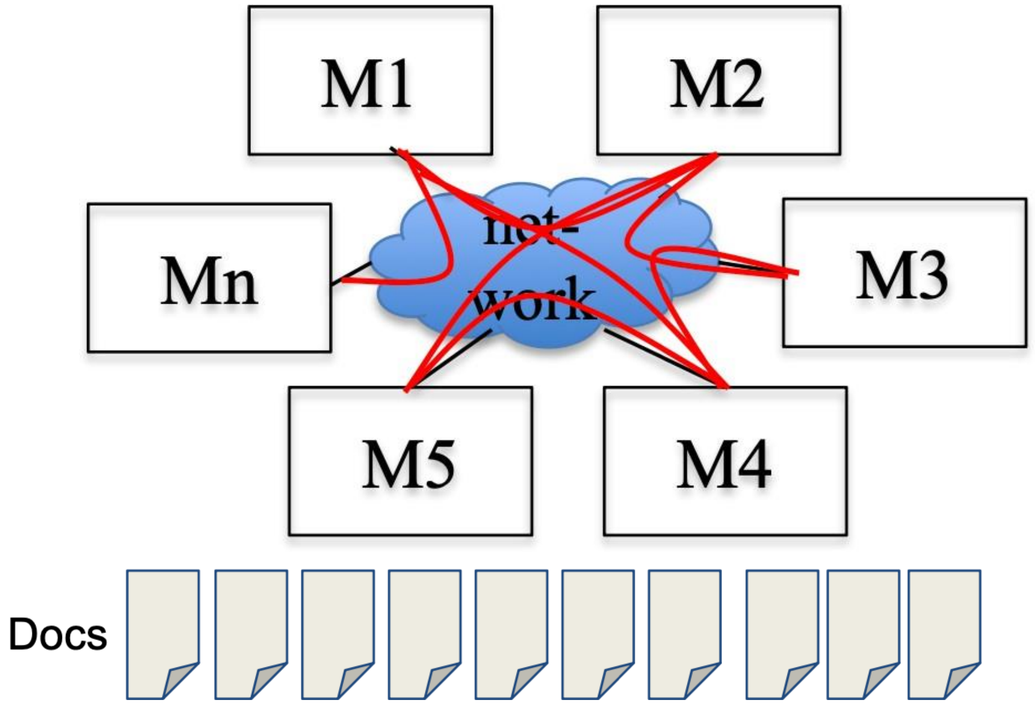
Example: Count word appearances in a corpus of documents.

Phase 1: Assign documents to different machines/nodes.

- Each computes a dictionary: {word: local_freq}.

Phase 2: Nodes exchange dictionaries (how?) to aggregate local_freq's.

- But how to make this scale??

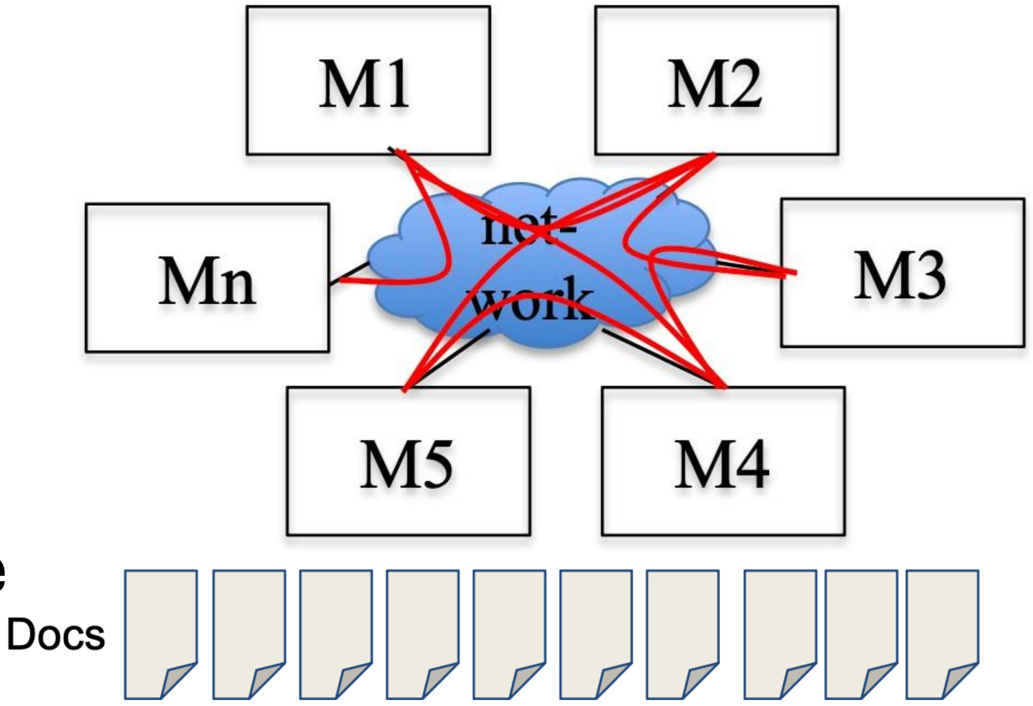


Option 2: Manual

Phase 2, Option a: Send all {word: local_freq} dictionaries to one node, who aggregates.

- But what if it's too much data for one node?

Phase 2, Option b: Each node sends (word, local_freq) to a designated node, e.g., node with ID hash(word) % N



Limitation of Option 2

How to generalize to other applications?

How to deal with failures?

How to deal with slow nodes?

How to deal with load balancing (some docs are very large, others small)?

How to deal with skew (some words are very frequent, so nodes designated to aggregate them will be pounded)?

They are common challenges for large-scale analytics!

Recap: Higher-Level Computation Frameworks

Give programmer a high-level abstraction for computation

Map computation automatically
onto a large cluster of machines

MapReduce

Parallelizable programming model:

- Consists of three phases, each intrinsically parallelizable:
    - Map: processes input elements independently to emit relevant (key, value) pairs from each.
    - Transparently, the runtime system groups all the values for each key together: (key, [list of values]), called "shuffle".
    - Reduce: aggregates all the values for each key to emit a global value for each key.
- Applies to a broad class of analytics applications.
- Isn’t as expressive as SQL but it is easier to scale.

Scalable, efficient, fault tolerant runtime system (discuss in later slides).

How MapReduce Works

Input: a collection of elements of (key, value) pair type.

Programmer defines two functions:

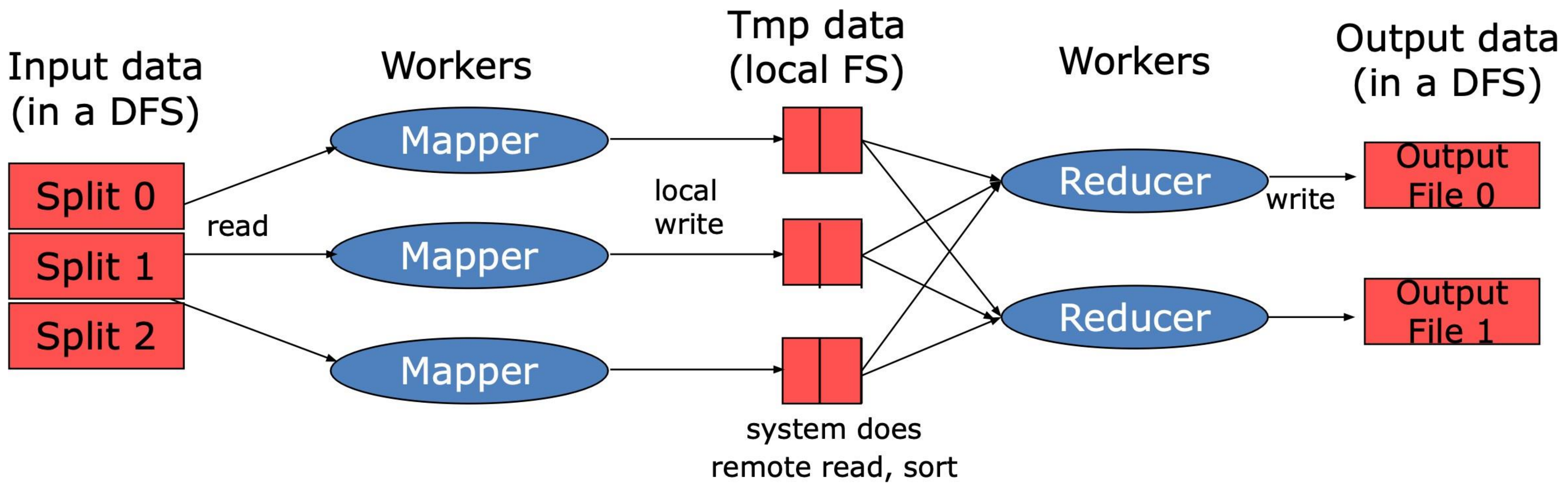
- Map(key, value) → a list of (key’, value’) pairs
- Reduce(key, value-list) → output

Execution

- Apply Map to each input key-value pair, in parallel for different keys.
- Sort emitted (key’, value’) pairs to produce (key’ value’-list) pairs.
- Apply Reduce to each (key’, value’-list) pair, in parallel for different keys.

Output is the union of all Reduce invocations’ outputs.

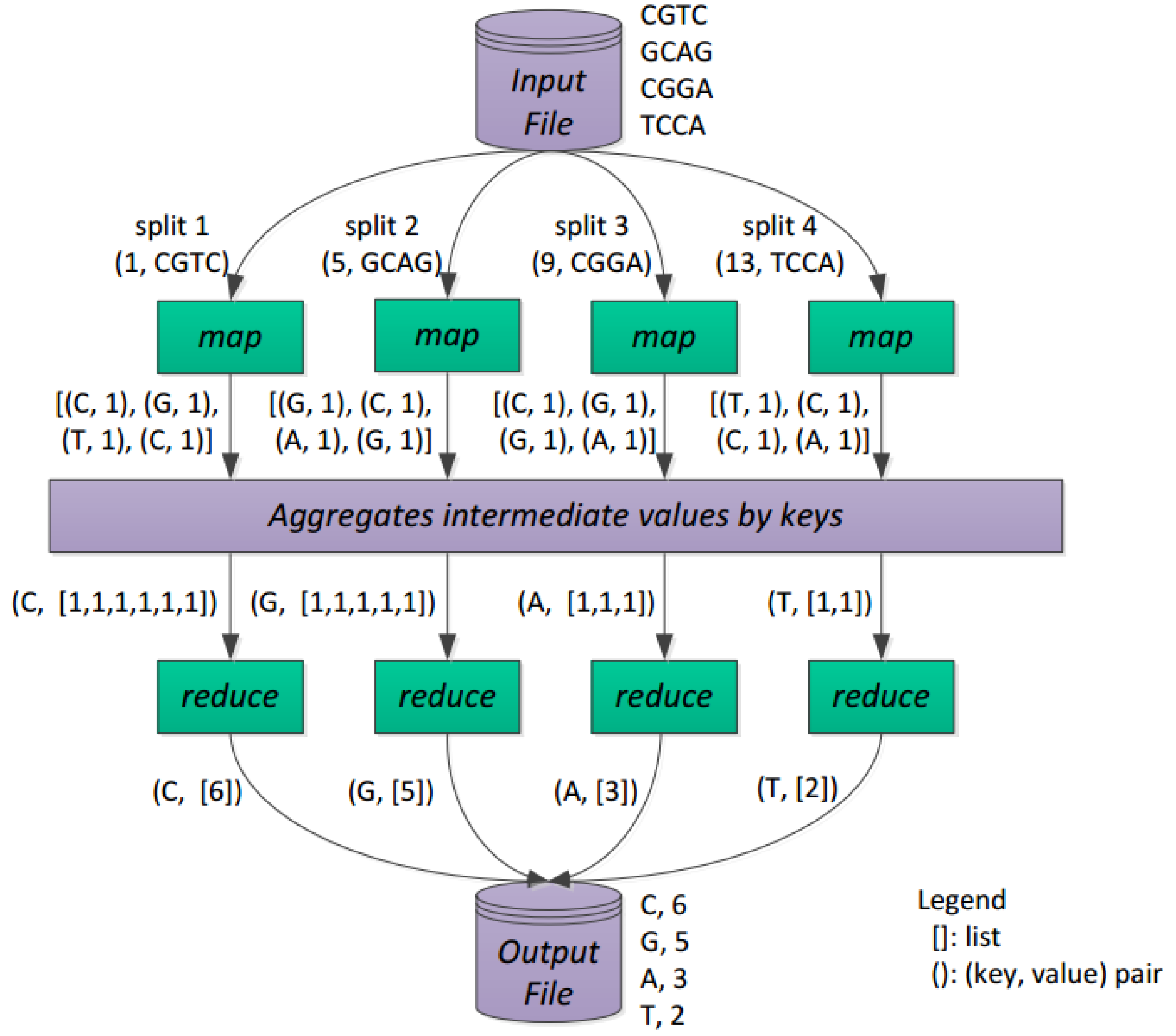
MapReduce: Workflow



Map phase:
extract something you care
about from each record

Reduce phase:
aggregate

One Example



CGTC
GCAG
CGGA
TCCA
Input File
split 1
(1, CGTC)
split 2
(5, GCAG)
split 3
(9, CGGA)
split 4
(13, TCCA)
map
map
map
map
[(C, 1), (G, 1), (T, 1), (C, 1)]
[(G, 1), (C, 1), (A, 1), (G, 1)]
[(C, 1), (G, 1), (G, 1), (A, 1)]
[(T, 1), (C, 1), (C, 1), (A, 1)]
Aggregates intermediate values by keys
(C, [1,1,1,1,1,1])
(G, [1,1,1,1,1])
(A, [1,1,1])
(T, [1,1])
reduce
reduce
reduce
reduce
(C, [6])
(G, [5])
(A, [3])
(T, [2])
Output File
C, 6
G, 5
A, 3
T, 2
Legend
[]: list
(): (key, value) pair

Recap: Word Count Example

We have a directory, which contains many documents.

The documents contain words separated by whitespace and punctuation.

Goal: Count the number of times each distinct word appears across the files in the directory.

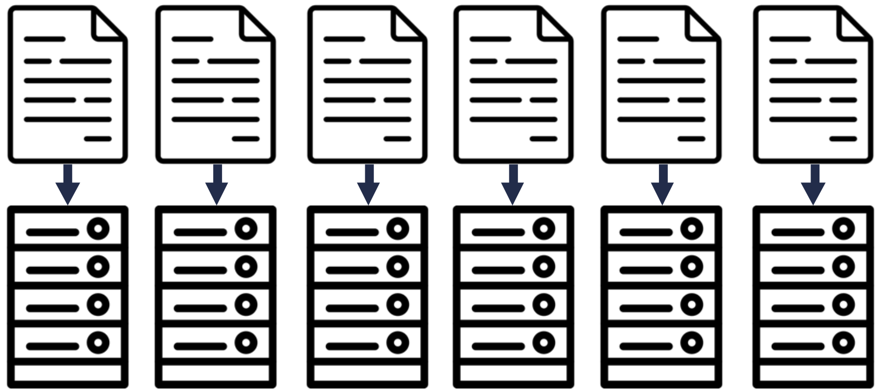
Map Phase

Mapper is given key: document ID; value: document content, say:

D1,“The teacher went to the store. The store was closed; the store opens in the morning. The store opens at 9am.”)

It will emit the following pairs:

<The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <store,1> <the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> <store,1> <opens, 1> <in, 1> <the, 1> <morning, 1> <the 1> <store, 1> <opens, 1> <at, 1> <9am, 1>



Intermediary Phase (Shuffle)

Transparently, the runtime sorts emitted (key, value) pairs by key:

```
<9am, 1>
<at, 1>
<closed, 1>
<in, 1>
<morning, 1>
<opens, 1>
<opens, 1>
<store, 1>
<store,1>
<store, 1>
<store,1>
```

Reducer 1

```
<teacher, 1>
<the, 1>
<the, 1>
<the, 1>
<the, 1>
<the 1>
<to, 1>
<went, 1>
<was, 1>
<The, 1>
```

Reducer 2

Reduce Phase

For each unique key emitted from the Map Phase, function Reduce(key, value-list) is invoked on Reducer 1 or Reducer 2.

Across their invocations, these Reducers will emit:

```
<9am, 1>
<at, 1>
<closed, 1>
<in, 1>
<morning, 1>
<opens, 2>
<store, 4>
```

Reducer 1

```
<teacher, 1>
<the, 5>
<to, 1>
<went, 1>
<was, 1>
<The, 1>
```

Reducer 2

Word Count With MapReduce

```
Map(key, value): // key: document ID; value: document content
  FOR (each word w IN value)
    emit(w, 1);
```

```
Reduce(key, value-list): // key: a word; value-list: a list of integers
  result = 0;
  FOR (each integer v on value-list)
    result += v;
  emit(key, result);
```

Exercise

(MapReduce) You are given a symmetric social network (like Facebook) where a is a friend of b implies that b is also a friend of a. The input is a dataset D (sharded) containing such pairs (a, b) – note that either a or b may be a lexicographically lower name. Pairs appear exactly once and are not repeated. Find the last names of those users whose first name is “Kanye” and who have at least 300 friends.

- Write pseudocode code for Map() and Reduce(). Your pseudocode may assume the presence of appropriate primitives (e.g., “firstname(user_id)”, etc.). The Map function takes as input a tuple (key=a,value=b)

Example: Word Frequency

Suppose instead of word count, we wanted to compute word frequency: the probability that a word would appear in a document.

This means computing the fraction of times a word appears, out of the total number of words in the corpus.

Solution: Chain two mapreduce’s

First Map/Reduce: Word Count (like before)

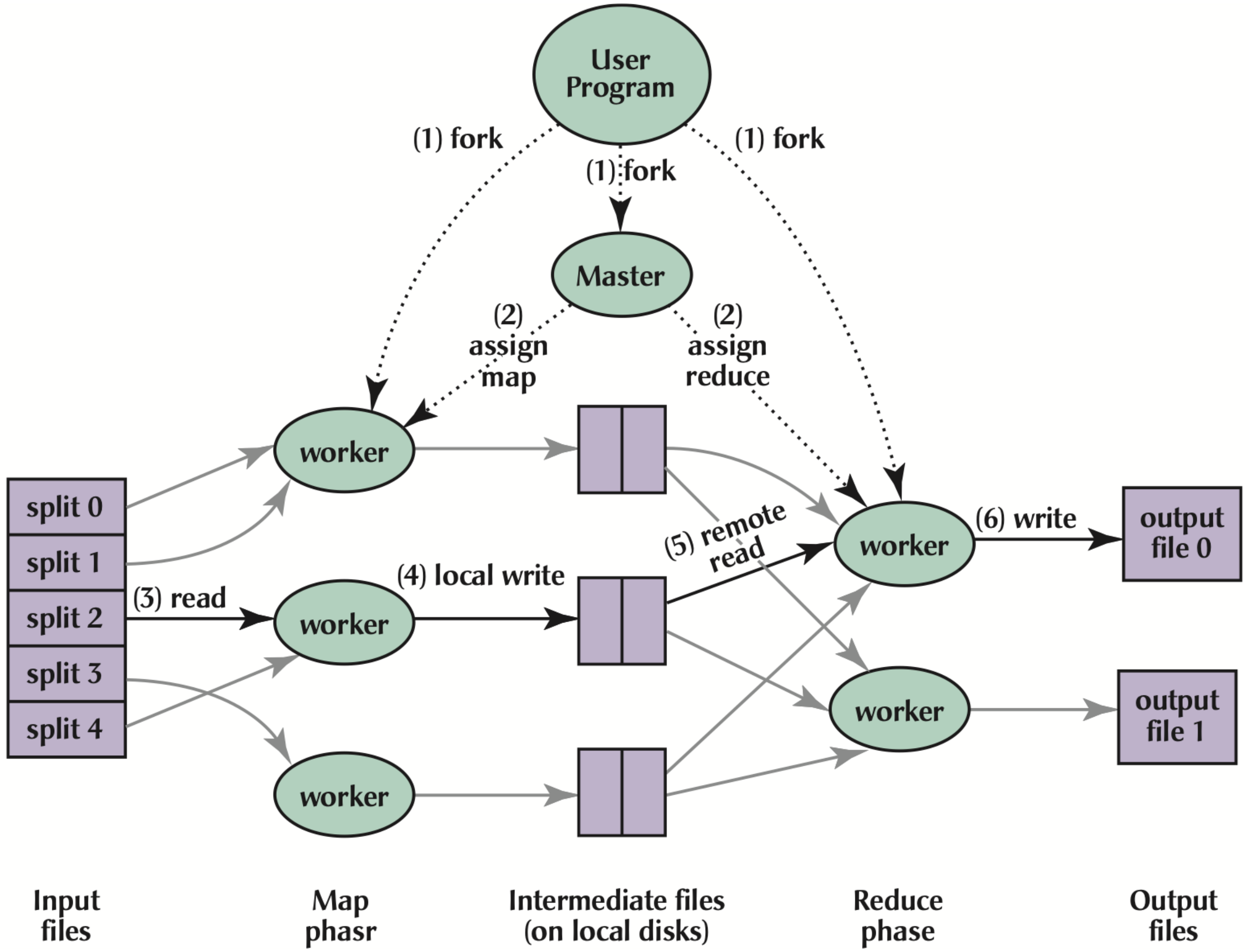
- Map: process documents and output pairs.
- Multiple Reducers: emit for each word.

Second MapReduce:

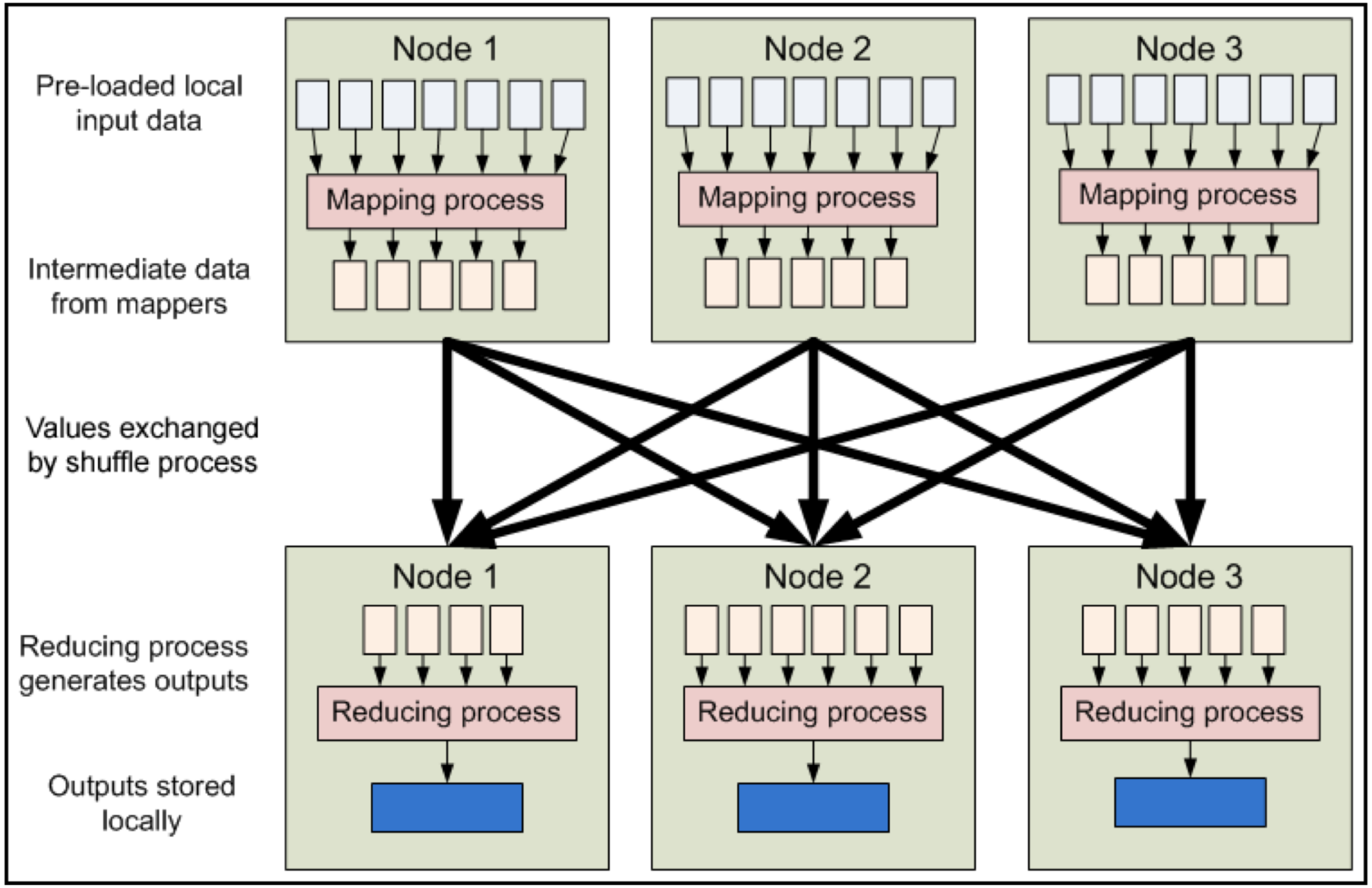
- Map: process<word, word_count> and output (1, (word, word_count)).
- 1 Reducer: perform two passes:
  - In first pass, sum up all word_count’s to calculate overall_count.
  - In second pass calculate fractions and emit multiple .

Scalability is not too bad, as first stage’s output is a rather small dictionary (maximum # of English words with an integer for each).

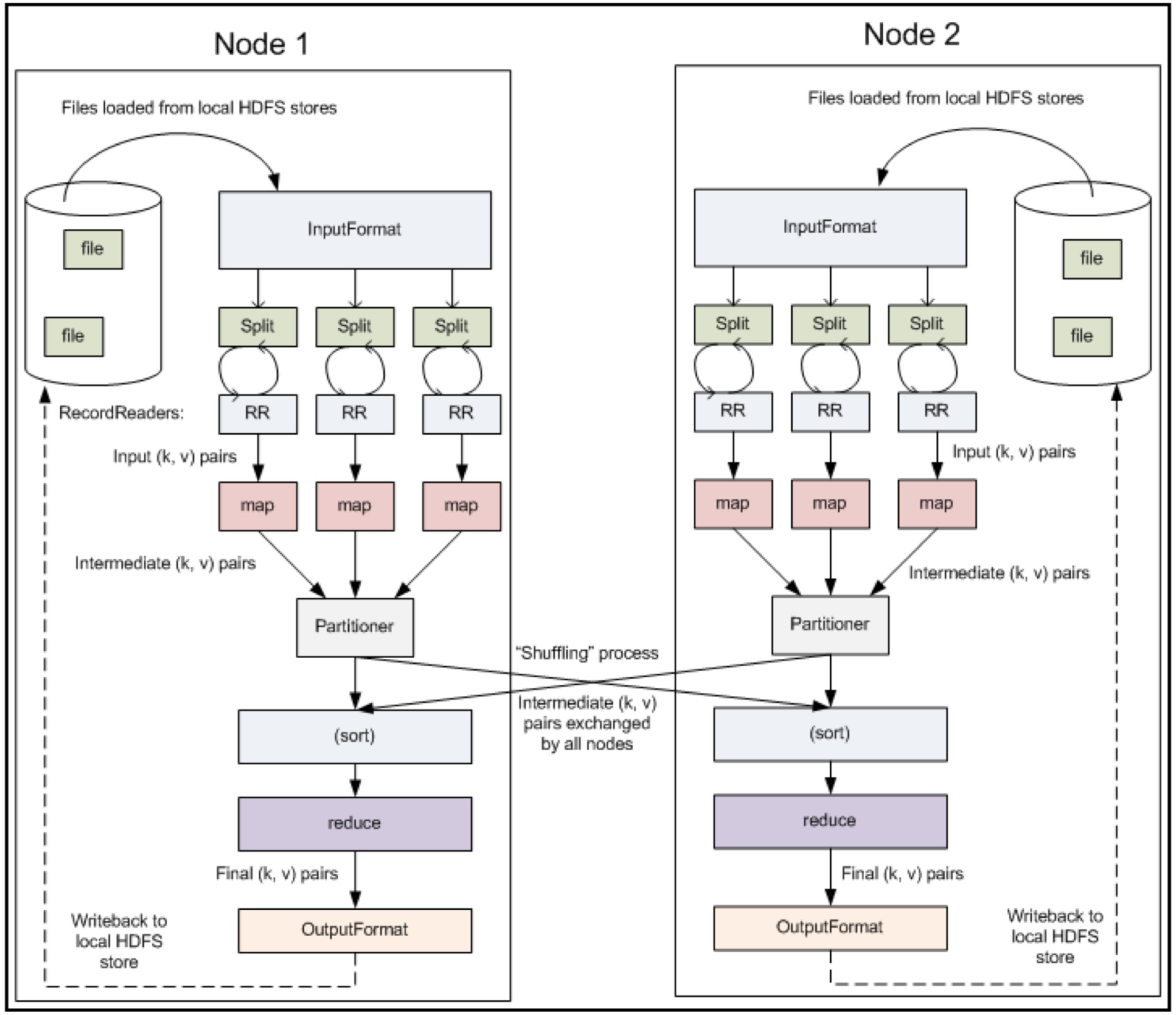
MapReduce Architecture



Detailed Architecture



GSF in MapReduce


Node 1
Node 2
Files loaded from local HDFS stores
Files loaded from local HDFS stores
file
file
InputFormat
Split
RecordReaders:
RR
Input (k, v) pairs
map
Intermediate (k, v) pairs
Partitioner
"Shuffling" process
Intermediate (k, v) pairs exchanged by all nodes
(sort)
reduce
Final (k, v) pairs
OutputFormat
Writeback to local HDFS store

Performance

A common bottleneck: network

- input data transferred all over the cluster, how to solve?

Solution: scheduling policy for data locality

- Asks GFS for locations of replicas of input file blocks.
- Map tasks are scheduled so GFS input block replica are on the same machine or on the same rack.

Effect: Thousands of machines read input at local speed.

- eliminate network bottleneck!
- But shuffle is still heavy

Fault Tolerance

Failures are the norm in data centers.

- Worker failure
- Master failure

Worker failure solution: Heartbeats

- Master detects if workers failed by periodically pinging them.
- Re-execute in-progress map/reduce tasks.

Master failure solution: Replication

- Initially, was single point of failure; Resume from Execution Log. Subsequent versions used replication and consensus.

Effect: From Google's paper: once, a Map/Reduce job lost 1600 of 1800 machines, but it still finished fine.

Stragglers

Slow workers or stragglers significantly lengthen completion time.

Slowest worker can determine the total latency!

- Other jobs consuming resources on machine.
- Bad disks with errors transfer data very slowly.
- This is why many systems measure 99th percentile latency.

Solution: spawn backup copies of tasks (redundant execution).

- Whichever one finishes first "wins.”
- I.e., treat slow executions as failed executions!