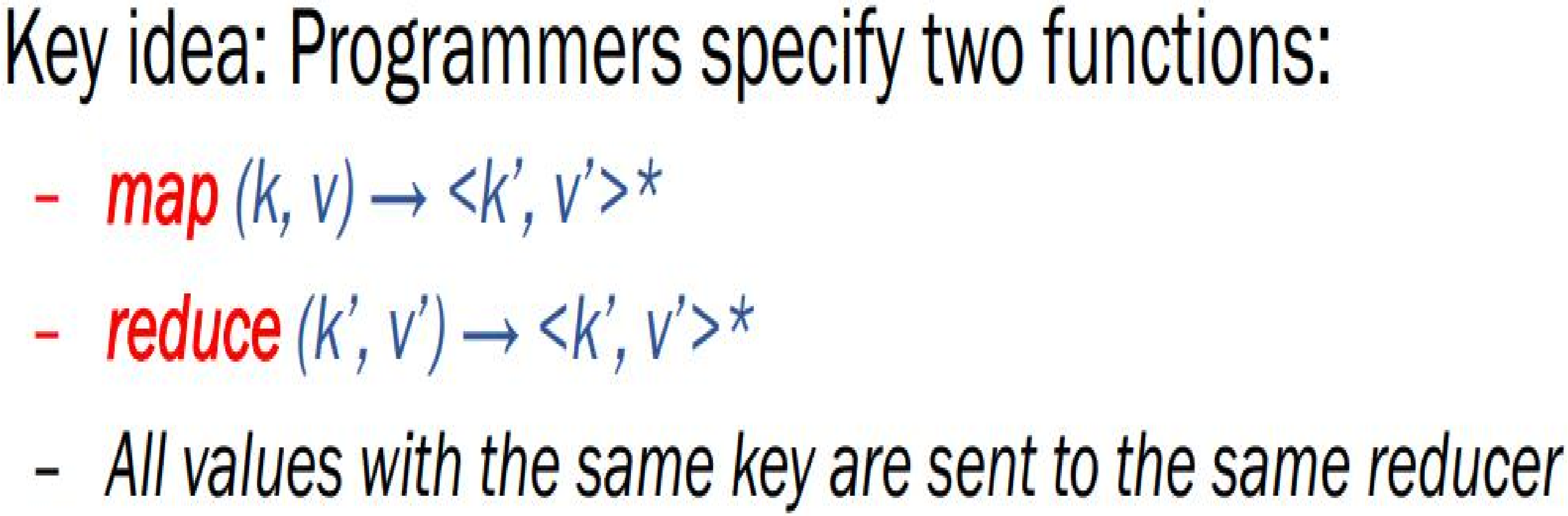
MapReduce

Lec -03

MapReduce

- MapReduce is a software framework and programming model used for processing huge amounts of data parallel by dividing the problem into some smaller and independent tasks.
- The **MapReduce** program works in two phases, namely
  - Map
  - Reduce.
- **Map tasks** deal with the splitting and mapping of data
- **Reduce tasks** shuffle and reduce the data. (Aggregate, summarize, filter or transform)

MapReduce- Key Idea

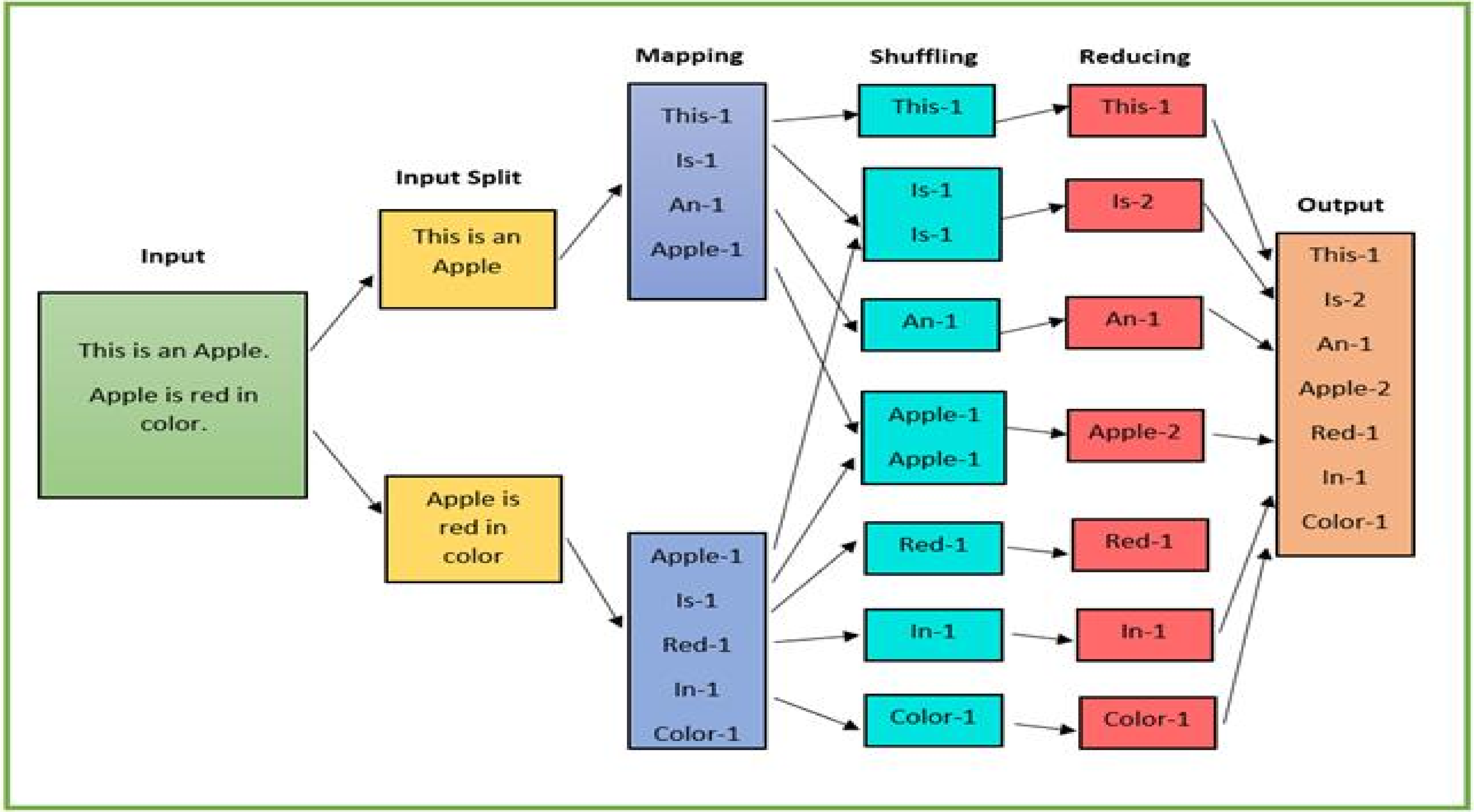


Key idea: Programmers specify two functions:

- *map* (k, v) → <k', v'>*
- *reduce* (k', v') → <k', v'>*
- *All values with the same key are sent to the same reducer*

MapReduce – Word Count

- We have a huge text document
- Count the number of times each distinct word appears in the file
- Sample application:
  - Analyze web server logs to find popular URLs



Input

This is an Apple.

Apple is red in color.

Input Split

This is an Apple

Apple is red in color

Mapping

This-1
Is-1
An-1
Apple-1

Apple-1
Is-1
Red-1
In-1
Color-1

Shuffling

This-1

Is-1
Is-1

An-1

Apple-1
Apple-1

Red-1

In-1

Color-1

Reducing

This-1

Is-2

An-1

Apple-2

Red-1

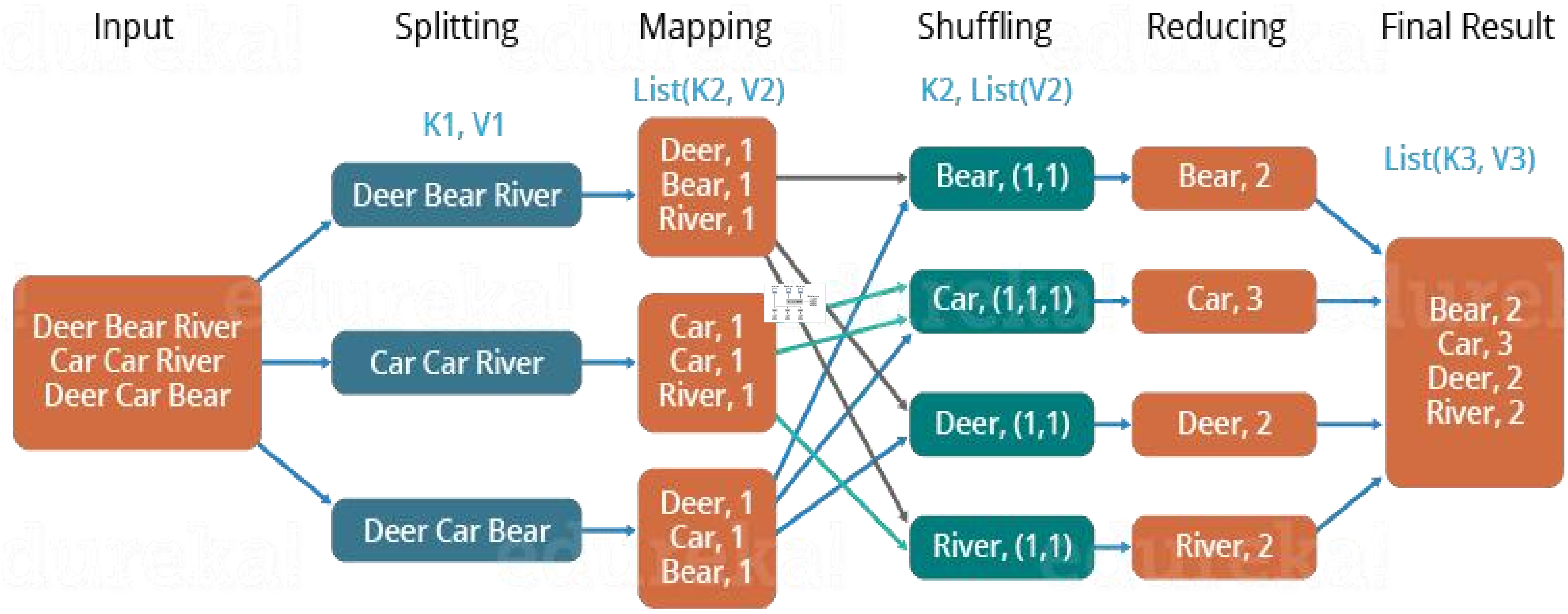
In-1

Color-1

Output

This-1
Is-2
An-1
Apple-2
Red-1
In-1
Color-1

The Overall MapReduce Word Count Process



Input

Deer Bear River
Car Car River
Deer Car Bear

Splitting

K1, V1

Deer Bear River

Car Car River

Deer Car Bear

Mapping

List(K2, V2)

Deer, 1
Bear, 1
River, 1

Car, 1
Car, 1
River, 1

Deer, 1
Car, 1
Bear, 1

Shuffling

K2, List(V2)

Bear, (1,1)

Car, (1,1,1)

Deer, (1,1)

River, (1,1)

Reducing

Bear, 2

Car, 3

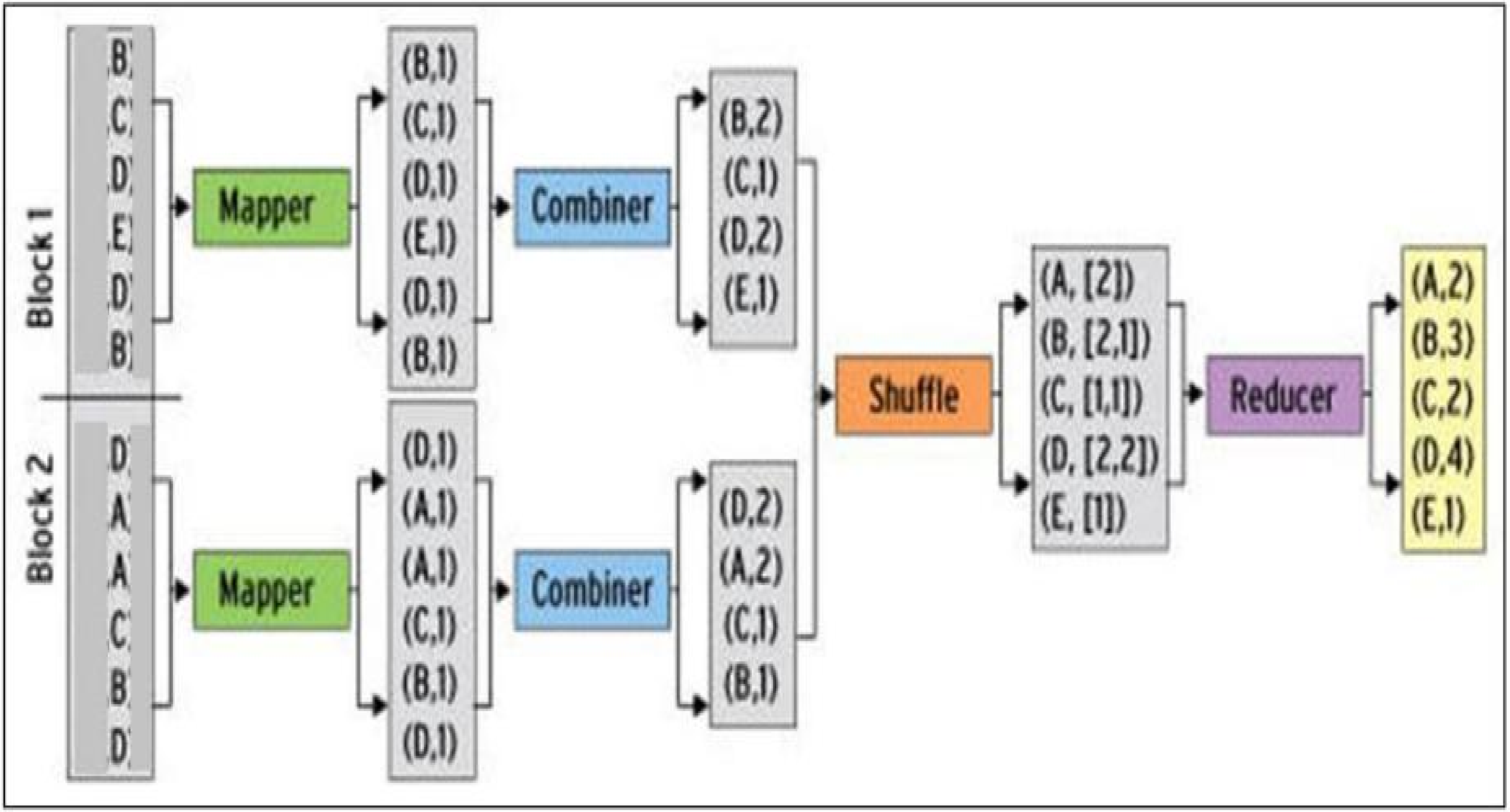
Deer, 2

River, 2

Final Result

List(K3, V3)

Bear, 2
Car, 3
Deer, 2
River, 2


Block 1
Block 2
B
C
D
E
D
B
D
A
A
C
B
D
Mapper
Mapper
(B,1)
(C,1)
(D,1)
(E,1)
(D,1)
(B,1)
(D,1)
(A,1)
(A,1)
(C,1)
(B,1)
(D,1)
Combiner
Combiner
(B,2)
(C,1)
(D,2)
(E,1)
(D,2)
(A,2)
(C,1)
(B,1)
Shuffle
(A, [2])
(B, [2,1])
(C, [1,1])
(D, [2,2])
(E, [1])
Reducer
(A,2)
(B,3)
(C,2)
(D,4)
(E,1)

Map-Reduce Environment is responsible for

Partitioning the input data

Scheduling the program’s execution across a set of machines

- *Assigns workers to map and reduce tasks*

Handles synchronization

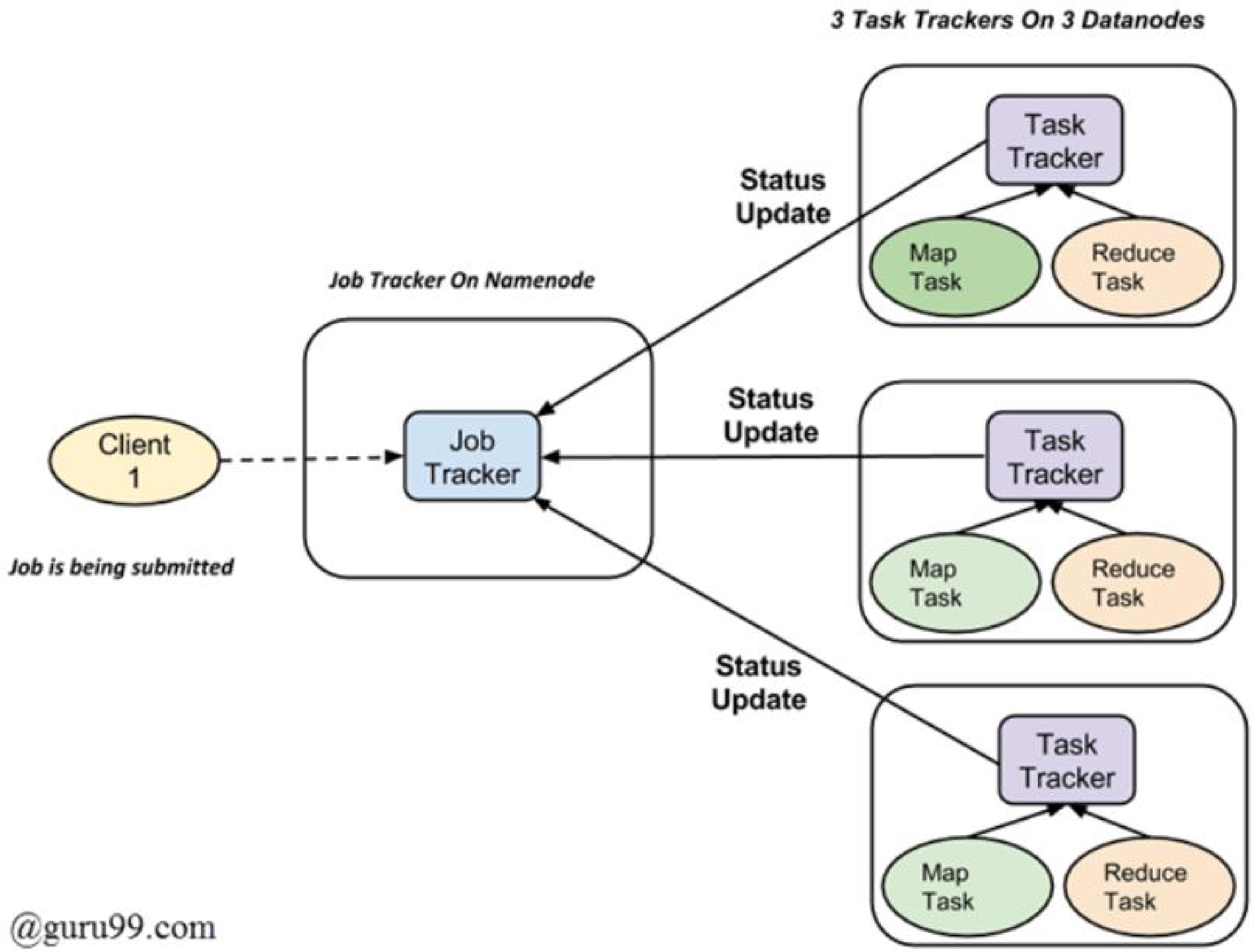
- *Gathers, sorts, and shuffles intermediate data*

Handling machine failures

- *Detects worker failures and restarts*

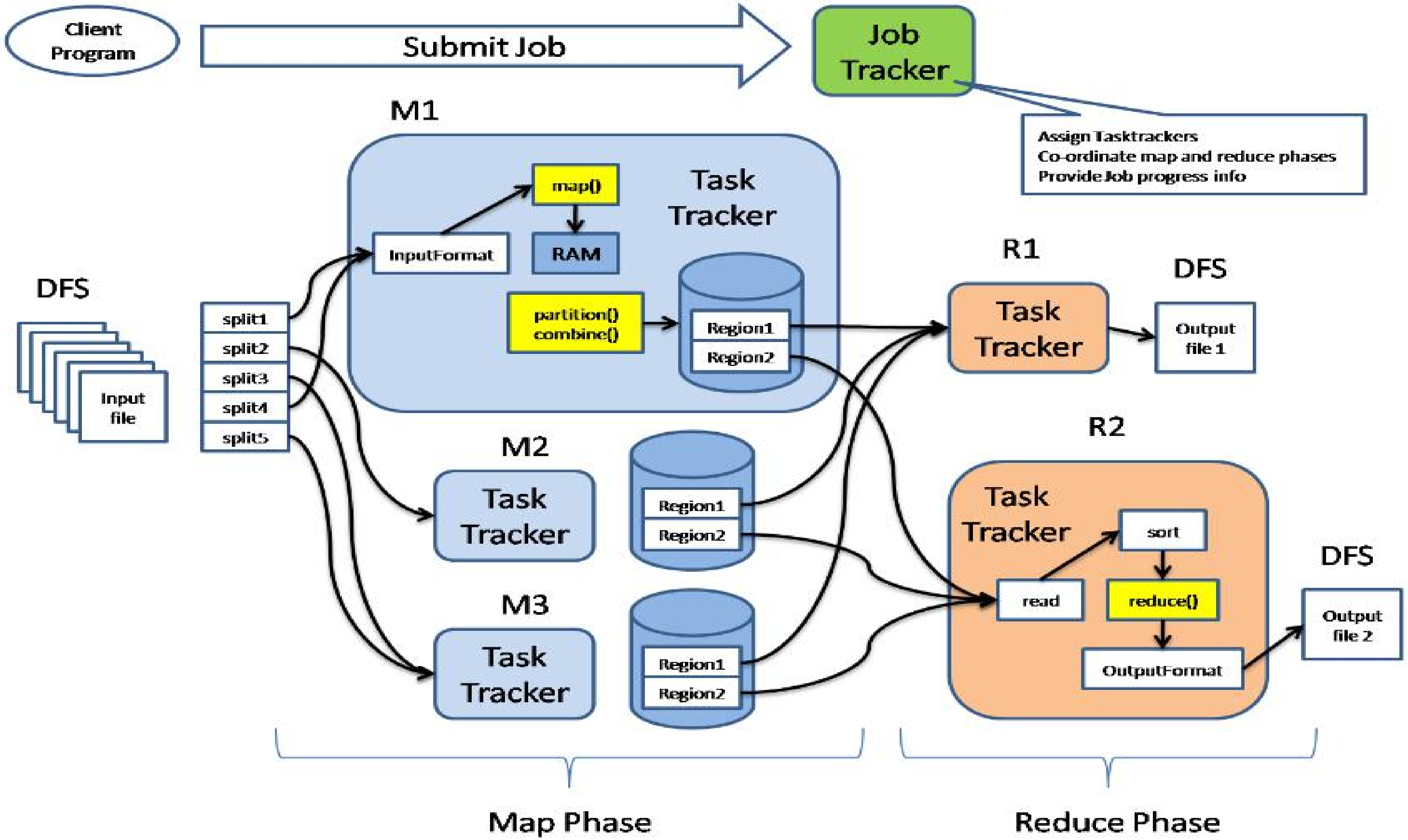
Managing required inter-machine communication

- The complete execution process (Map & Reduce task)is controlled by
    - Jobtracker
    - Task Trackers
- **Job Tracker:** This tracker plays the role of scheduling jobs and tracking all jobs assigned to the task tracker. Act like a master.
- **Task Tracker:** This tracker plays the role of tracking tasks and reporting the status of tasks to the job tracker. Act like a slave.
- Task tracker periodically sends **‘heartbeat’** signal to the Jobtracker so as to notify him of the current state of the system.
- In the event of task failure, the job tracker can reschedule it on a different task tracker.



@guru99.com

How Hadoop MapReduce Works



Client Program

Submit Job

Job Tracker

Assign Tasktrackers
Co-ordinate map and reduce phases
Provide Job progress info

DFS

Input file

split1
split2
split3
split4
split5

M1

Task Tracker

InputFormat

map()

RAM

partition()
combine()

Region1
Region2

M2

Task Tracker

Region1
Region2

M3

Task Tracker

Region1
Region2

R1

Task Tracker

DFS

Output file 1

R2

Task Tracker

read

sort

reduce()

OutputFormat

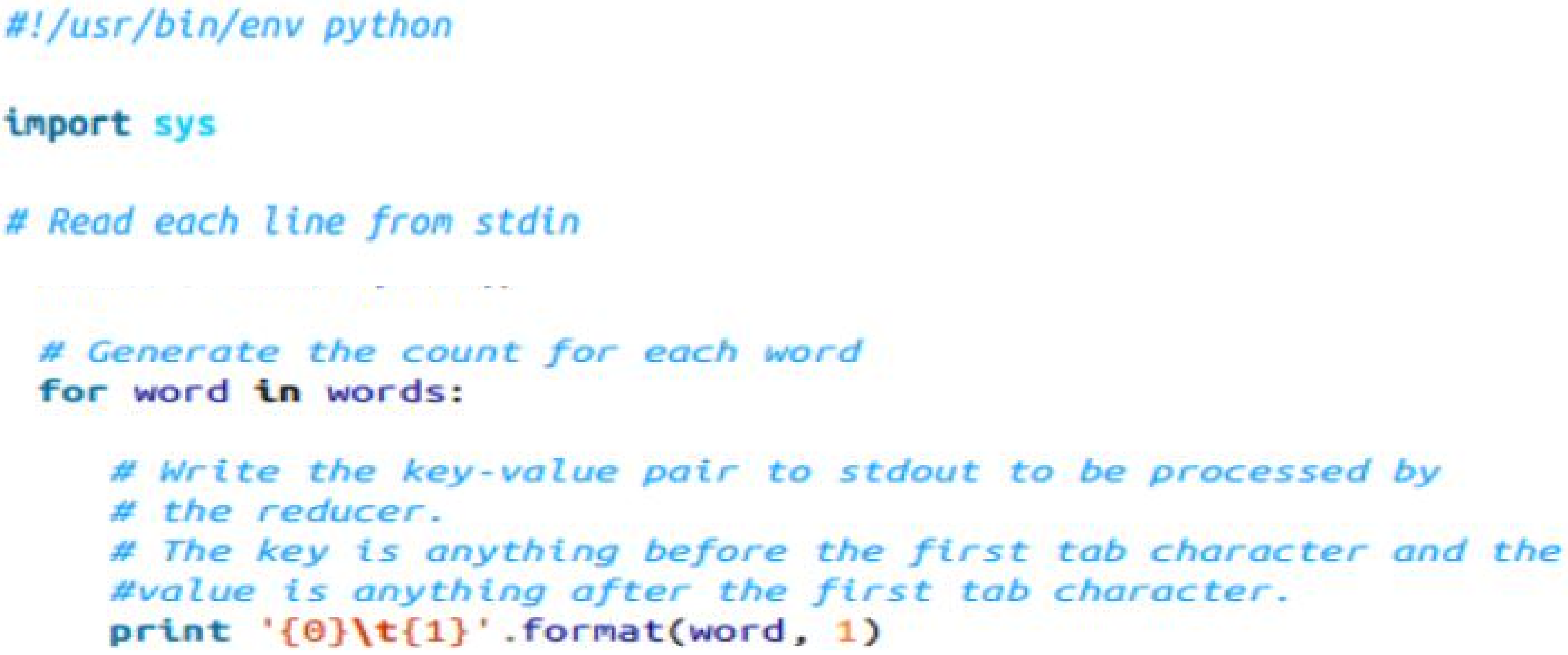
DFS

Output file 2

Map Phase

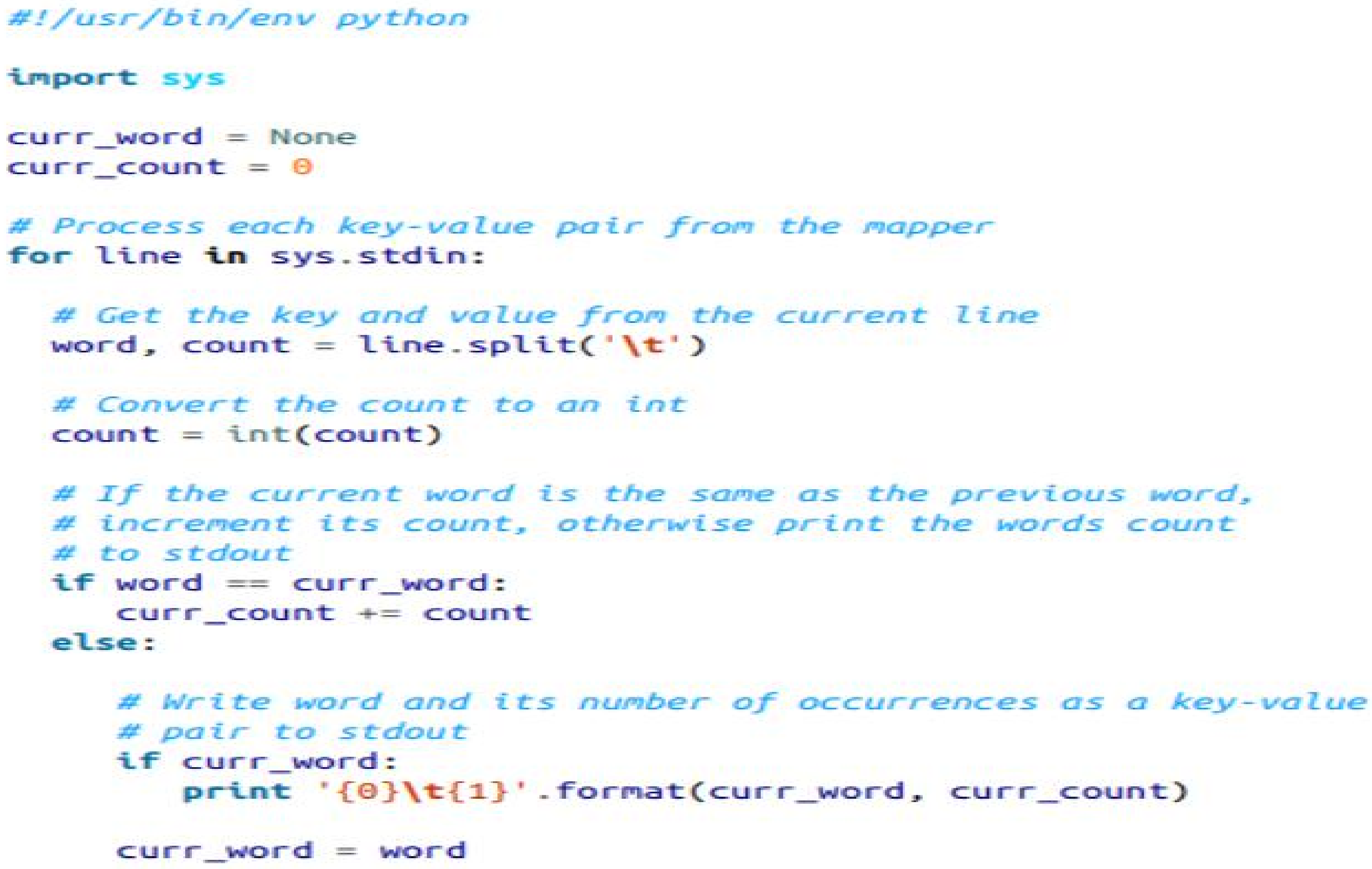
Reduce Phase

Mapper.py

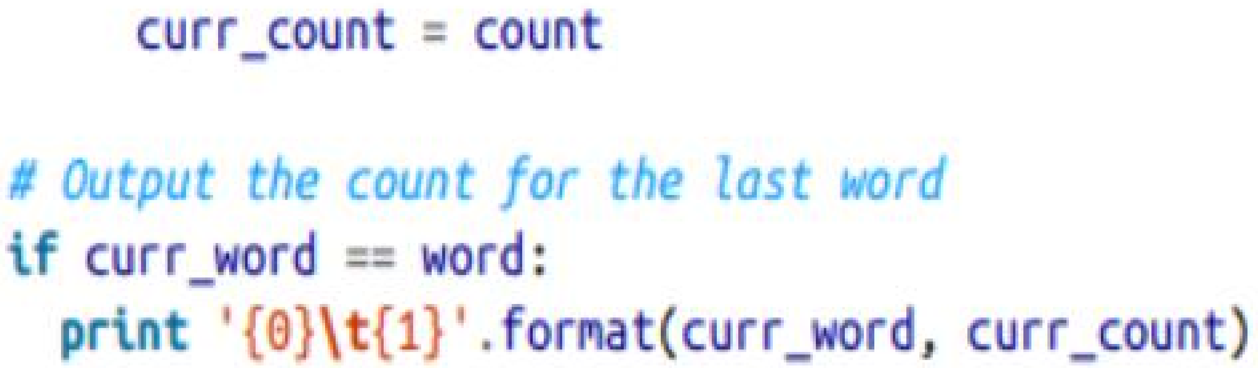


```
#!/usr/bin/env python

import sys

# Read each line from stdin

# Generate the count for each word
for word in words:

    # Write the key-value pair to stdout to be processed by
    # the reducer.
    # The key is anything before the first tab character and the
    #value is anything after the first tab character.
    print '{0}\t{1}'.format(word, 1)
```

Reducer.py

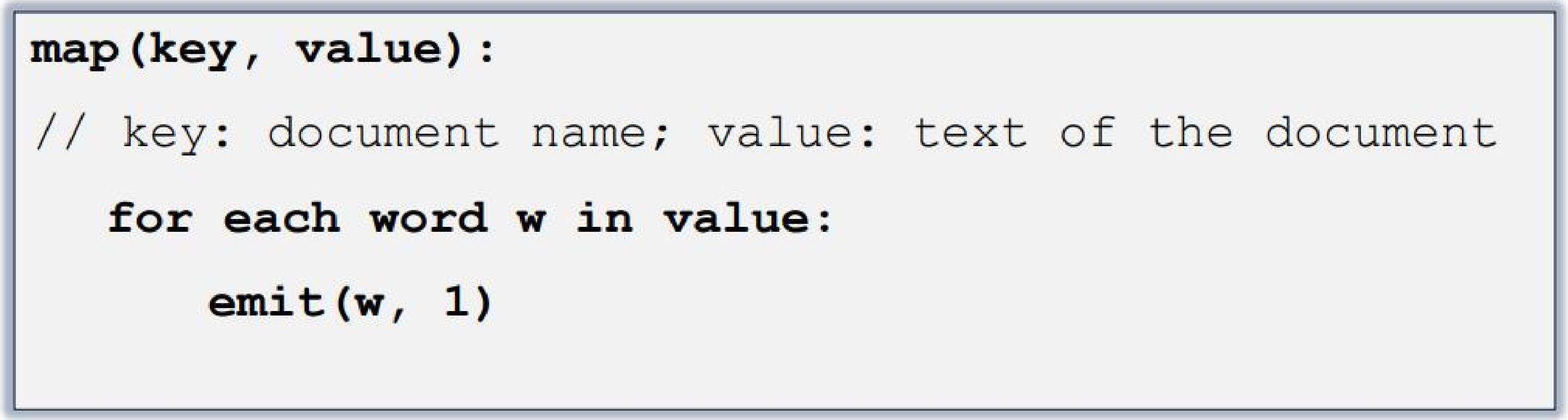


```
#!/usr/bin/env python

import sys

curr_word = None
curr_count = 0

# Process each key-value pair from the mapper
for line in sys.stdin:

  # Get the key and value from the current line
  word, count = line.split('\t')

  # Convert the count to an int
  count = int(count)

  # If the current word is the same as the previous word,
  # increment its count, otherwise print the words count
  # to stdout
  if word == curr_word:
     curr_count += count
  else:

     # Write word and its number of occurrences as a key-value
     # pair to stdout
     if curr_word:
        print '{0}\t{1}'.format(curr_word, curr_count)

     curr_word = word
```

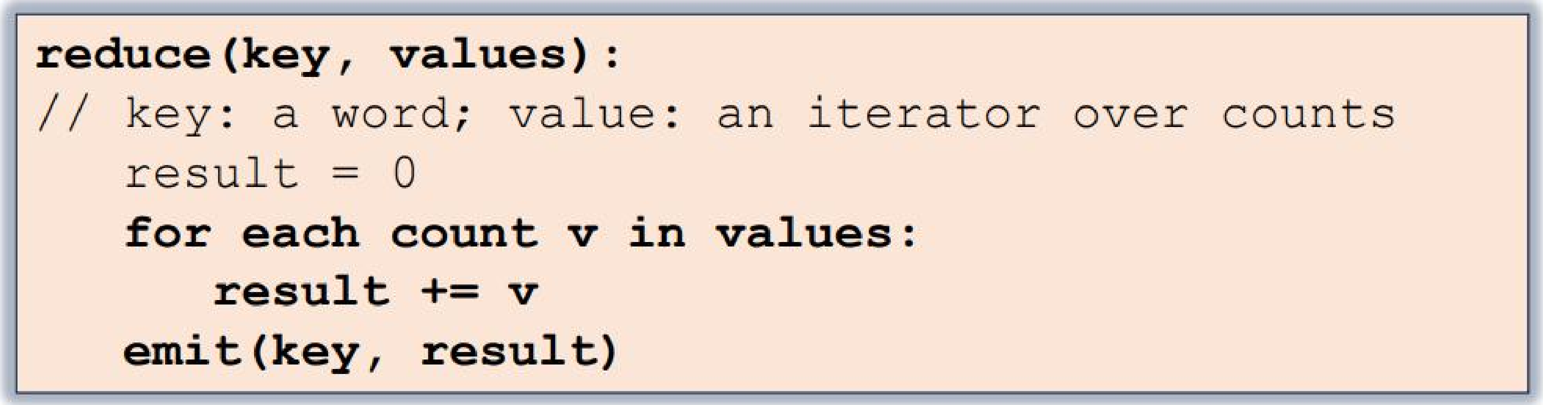


```
    curr_count = count

# Output the count for the last word
if curr_word == word:
  print '{0}\t{1}'.format(curr_word, curr_count)
```

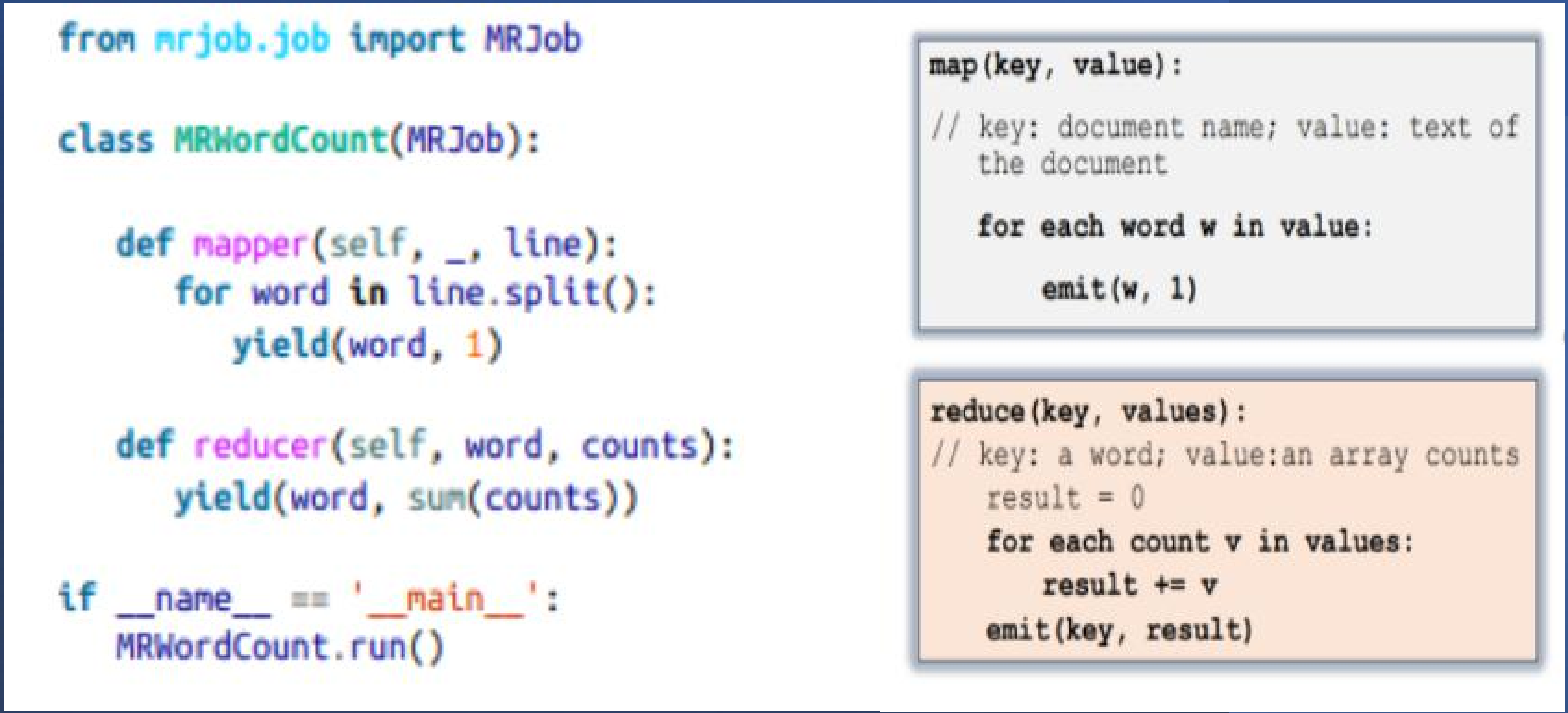
Word Count using MapReduce



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



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



```
from mrjob.job import MRJob

class MRWordCount(MRJob):

    def mapper(self, _, line):
        for word in line.split():
            yield(word, 1)

    def reducer(self, word, counts):
        yield(word, sum(counts))

if __name__ == '__main__':
    MRWordCount.run()
```

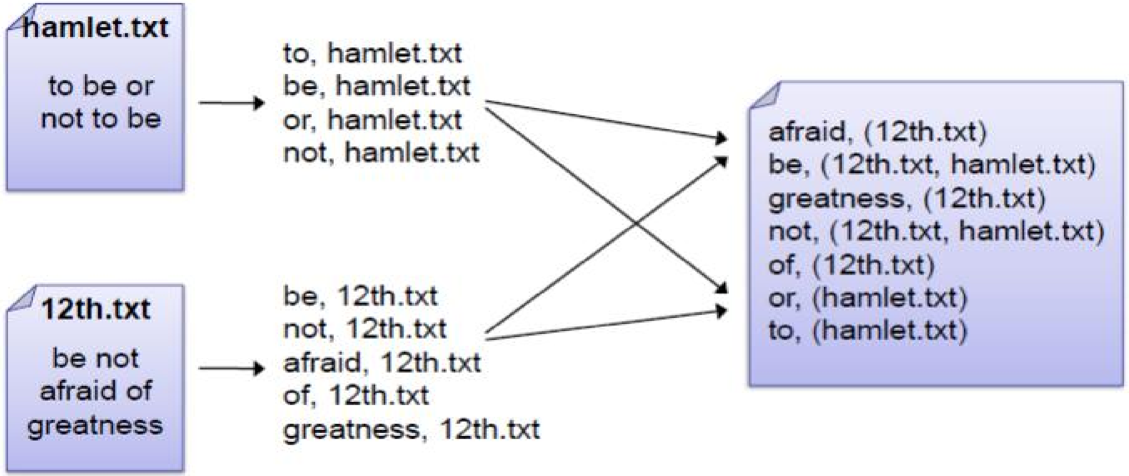
```
map(key, value):
// key: document name; value: text of
   the document
   for each word w in value:
       emit(w, 1)
```

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

Example: Inverted Index

- This was the original Google's usecase
- Generate an inverted index of words from a given set of files

- **Map:**
  - parses a document and emits <word, docId> pairs
- **Reduce:**
  - takes all pairs for a given word, sorts the docId values, and emits a <word,list(docId)> pair



hamlet.txt

to be or
not to be

to, hamlet.txt
be, hamlet.txt
or, hamlet.txt
not, hamlet.txt

12th.txt

be not
afraid of
greatness

be, 12th.txt
not, 12th.txt
afraid, 12th.txt
of, 12th.txt
greatness, 12th.txt

afraid, (12th.txt)
be, (12th.txt, hamlet.txt)
greatness, (12th.txt)
not, (12th.txt, hamlet.txt)
of, (12th.txt)
or, (hamlet.txt)
to, (hamlet.txt)

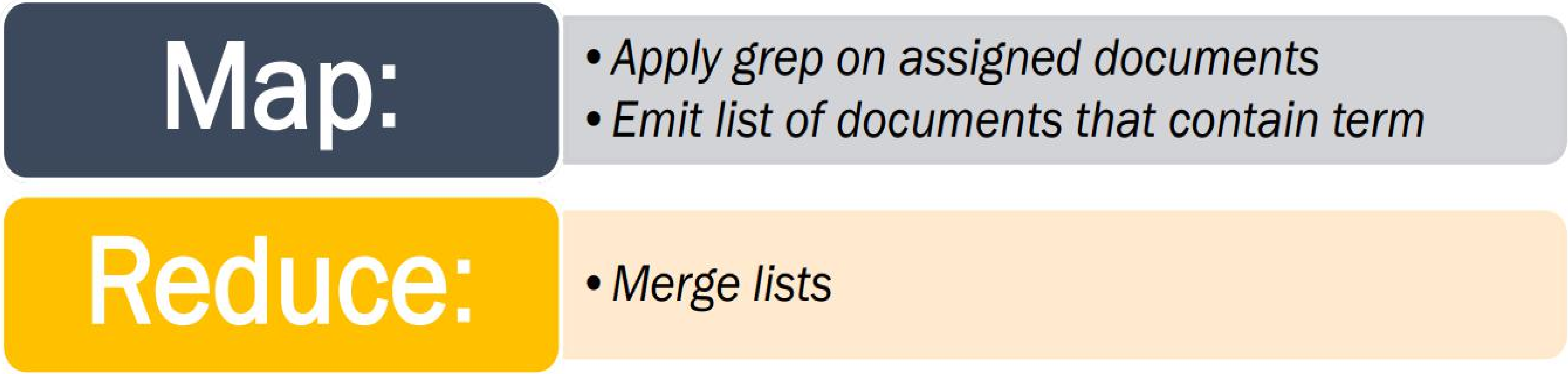
Example: Language modeling

- Statistical machine translation:
  - *Need to count number of times every 5-word sequence occurs in a large corpus of documents*

Map	• *extract (5-word sequence, count) from document*
Reduce	• *combine counts*

Example: Distributed Grep

- Find all occurrences of the **given pattern** in a very large set of files.



Map:

- *Apply grep on assigned documents*
- *Emit list of documents that contain term*

Reduce:

- *Merge lists*

Dealing with Failures

Map worker failure

- *Map tasks completed or in-progress at worker are restarted*

Reduce worker failure

- *Only in-progress tasks are restarted*

Master failure

- *MapReduce task is aborted and client is notified*

References

- https://www.guru99.com/introduction-to-mapreduce.html
- https://www.section.io/engineering-education/understanding-map-reduce-in-hadoop/
- https://www.geeksforgeeks.org/introduction-to-hadoop-distributed-file-systemhdfs/
- https://ia600201.us.archive.org/7/items/HadoopInPractice/Hadoop%20in%20Practice.pdf
- https://pepa.holla.cz/wp-content/uploads/2016/10/hadoop-with-python.pdf