Introduction
to
MapReduce

MapReduce?

- A programming model or abstraction.
- It is not a programming language
- A novel way of thinking about designing a solution to certain big data problems…
- It enables us to

  1. Partition data into small chunks (called partitions)
  2. Execute tasks in parallel by:
     - Mappers
     - Reducers

MapReduce as a Model/Paradigm/Architecture

Implementations of MapReduce:

- Google App Engine
- Apache Hadoop
- Apache Tez
- Apache Spark (implements superset of MapReduce)
- Snowflake
- Amazon Athena

Motivation of MapReduce

- Process lots of data in parallel
  - Google processed about 24 petabytes of data per day in 2009.
  - Facebook processed 60 petabytes of data per day in 2020
- **A single machine** cannot serve all the data
  - You need a distributed system (called cluster computing) to store and process **in parallel**
- Parallel programming

Motivation

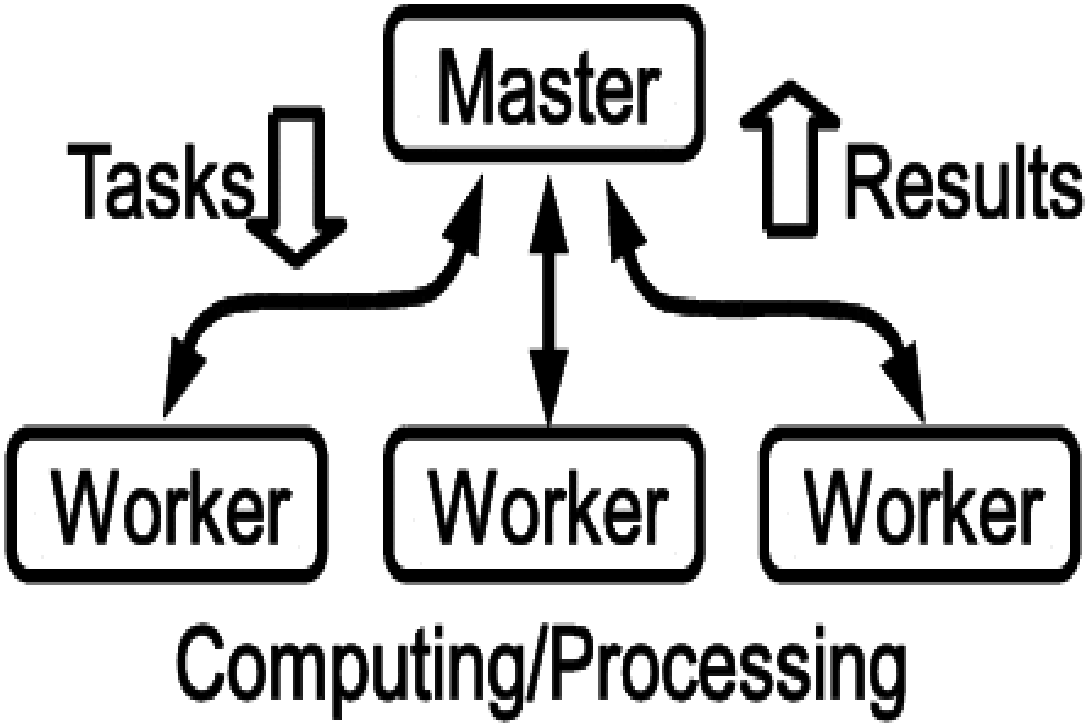
- Parallel programming?
  - Concurrency/Threading/Parallelism is hard!
  - How do you facilitate communication between servers/nodes?
  - How do you scale to more machines?
  - How do you handle machine failures?
  - How do we handle disk failures?

- MapReduce uses cluster computing and enables parallelism

What is a Cluster Computing?

- MapReduce uses cluster computing and enables parallelism
- A computer cluster is a set of computers (nodes or servers) that work together so that they can be viewed as a single system.
- Example: A cluster of 101 nodes:
  - One Master (as a cluster manager)
  - 100 worker nodes

What is a Cluster Computing?
Perform tasks in parallel using a set of computers



MapReduce reintroduced…

- Google created the awareness by publishing a paper
- *Apache Hadoop* made it into a sensation
- *Apache Hadoop* is an open-source MapReduce implementation based on Google's paper.
- *Spark implements superset of MapReduce*

MapReduce: Simplified Data Processing on Large Clusters

By Jeffrey Dean and Sanjay Ghemawat, Google

OSDI'04: Sixth Symposium on Operating System Design and Implementation. December, 2004.

MapReduce reintroduced…

Google's big problem:

- Index billions of web documents *everyday!*
- *Takes too much time and effort!*

Solution:

Use MapReduce to utilize 100's or 1000's of servers (cluster computing) to index billions of documents

(key, value) in MapReduce

For MapReduce, everything (input and output) is expressed as a tuple of 2 values: `(key, value)`

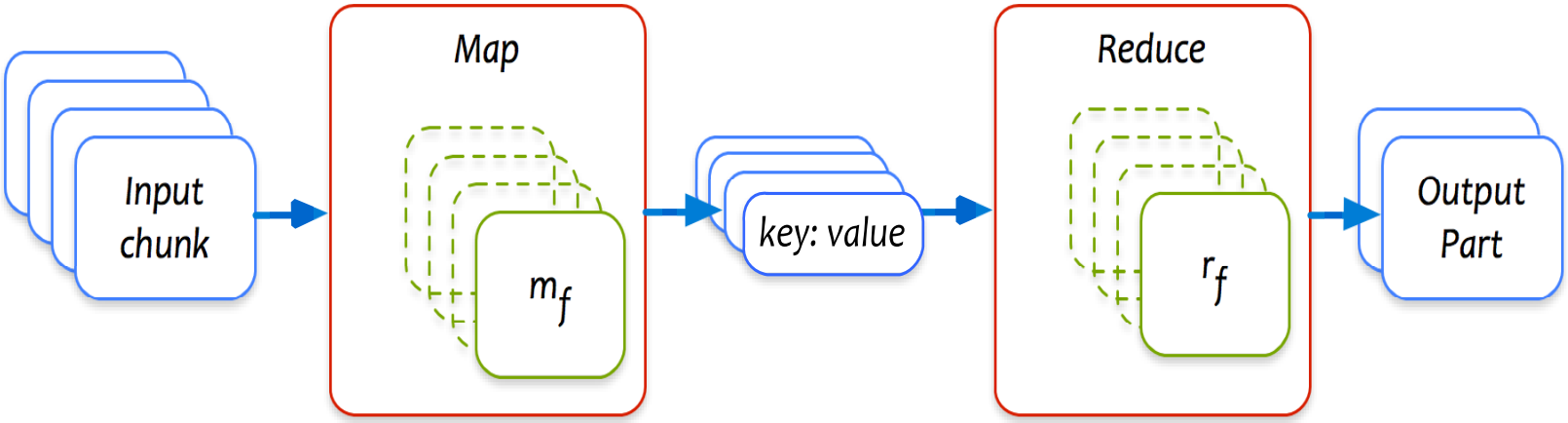
NOTE: key and value can be any data types

- **Mapper:**
  - `map(key, value)`
    - `key: as a partition number, or a record number`
    - `value: an actual input record`
  - `Outputs { {k2, v2), … }`

- **Reducer:**
  - `reduce(key2, value2)`
    - `key2 : a k2 in mappers`
    - `value2 : Iterable<objects> OR list of objects`
  - `Outputs: { (k3, v3), …}`

Hadoop implements MapReduce

1. Partition input into small chunks
2. Parallelize chunks in many servers
3. Apply `map()` and `reduce()`



Word Count: Notation

() denotes a Tuple

- Example-1: (1, 2, 4) denotes tuple of 3 integers
- Example-2: ("fox", 3) denotes a (**key**, **value**) pair
  Where **key** is "fox" and **value** is 3
- Example-3: () denotes an empty tuple

[] denotes a List of 0, 1, 2, … items/objects

- Example-1: [1, 2, 6] is a list of 3 integers
- Example-2: ["to", "be"] is a list of 2 strings
- Example-3: [] denotes an empty list

{} denotes a Set of 0, 1, 2, … or more objects

- Example-1: { ("a", 2), ("b", 3), ("z", 9) }
  denotes a set of 3 pair objects
- Example-2: {} denotes an empty set

Word Count: Notation

Iterable<object> denotes a list of objects

- Example-1: `Iterable<Integer>`
  - `[1, 2, 2, 3, 4]`
  - `[1, 1, 1, 1, 1, 1, 1]`
- Example-2: `Iterable<String>`
  - `["a", "fox", "jumped"]`
  - `["a", "a", "b", "b", "b"]`
- Example-3: `Iterable<(String, Integer)>`
  - `[("a", 10), ("fox", 7), ("jumped", 8)]`
  - `[("key1", 10), ("key200", 700)]`

MapReduce flow... 1

1. Input is partitioned and passed to mappers
2. Mappers get input as (**key, value**) pairs
   - **key** might be a partition number or record number and might be ignored (if not needed)
   - **value:** as an input record (as a String)
   - map(key, value): emits a set of {(key2, value2)} pairs

3. Output of mappers is passed into **Sort & Shuffle** system (provided by MapReduce implementation)

MapReduce flow... 2

4. Assume that output from all mappers are:

(there are N unique keys: {Key_1, Key_2, …, Key_N})

(Key_1, v_11), (Key_1, v_12), …

(Key_2, v_21), (Key_2, v_22), …

…

(Key_N, v_N1), (Key_N, v_N2), …

Then Sort & Shuffle groups values of mappers by their associated keys.
Sort & Shuffle outputs (key, value) pairs as:

(Key_1, [v_11, v_12, …])

(Key_2, [v_21, v_22, …])

…

(Key_N, [v_N1, v_N2, …])

Sort & Shuffle ←→ SQL's GROUP BY

MapReduce flow... 3

5. Output of Sort & Shuffle is passed to reducers:

```
(Key_1, [v_11, v_12, …])
(Key_2, [v_21, v_22, …])
…
```

6. A reducer will operate/execute on (key, value) produced by Sort & Shuffle

```
# key : one of Key_1 or Key_2, …
# values: associated values for a given key Key_1 or Key_2, …
reduce(key, values){
   <reducer's logic, which may emit any number
    of (K3, V3) pairs>
}
```

MapReduce Example: word count

MapReduce works with **(key, value)** pairs

- Mappers Input: as (key, value) pair
- map(123, "fox jumped and fox jumped and jumped and jumped")
  - key: 123 as a key is a record number (ignored here)
  - value: "fox jumped and fox jumped and jumped and jumped"

Mappers output as (K, V) pairs: K is a word, V is a frequency

```
(fox, 1), (jumped, 1), (and, 1),
(fox, 1), (jumped, 1),  (and, 1),
(jumped, 1), (and, 1), (jumped, 1)
```

MapReduce Example continued...

Mappers output:

```
(fox, 1), (jumped, 1), (and, 1),
(fox, 1), (jumped, 1),  (and, 1),
(jumped, 1), (and, 1), (jumped, 1)
```

Sort & Shuffle output:

GROUP BY mapper's output KEY

```
(fox, [1, 1])
(jumped, [1, 1, 1, 1])
(and, [1, 1, 1])
```

MapReduce Example continued...

Sort&Shuffle output: (used as input to reducers)

(fox, [1, 1])

(jumped, [1, 1, 1, 1])

(and, [1, 1, 1])

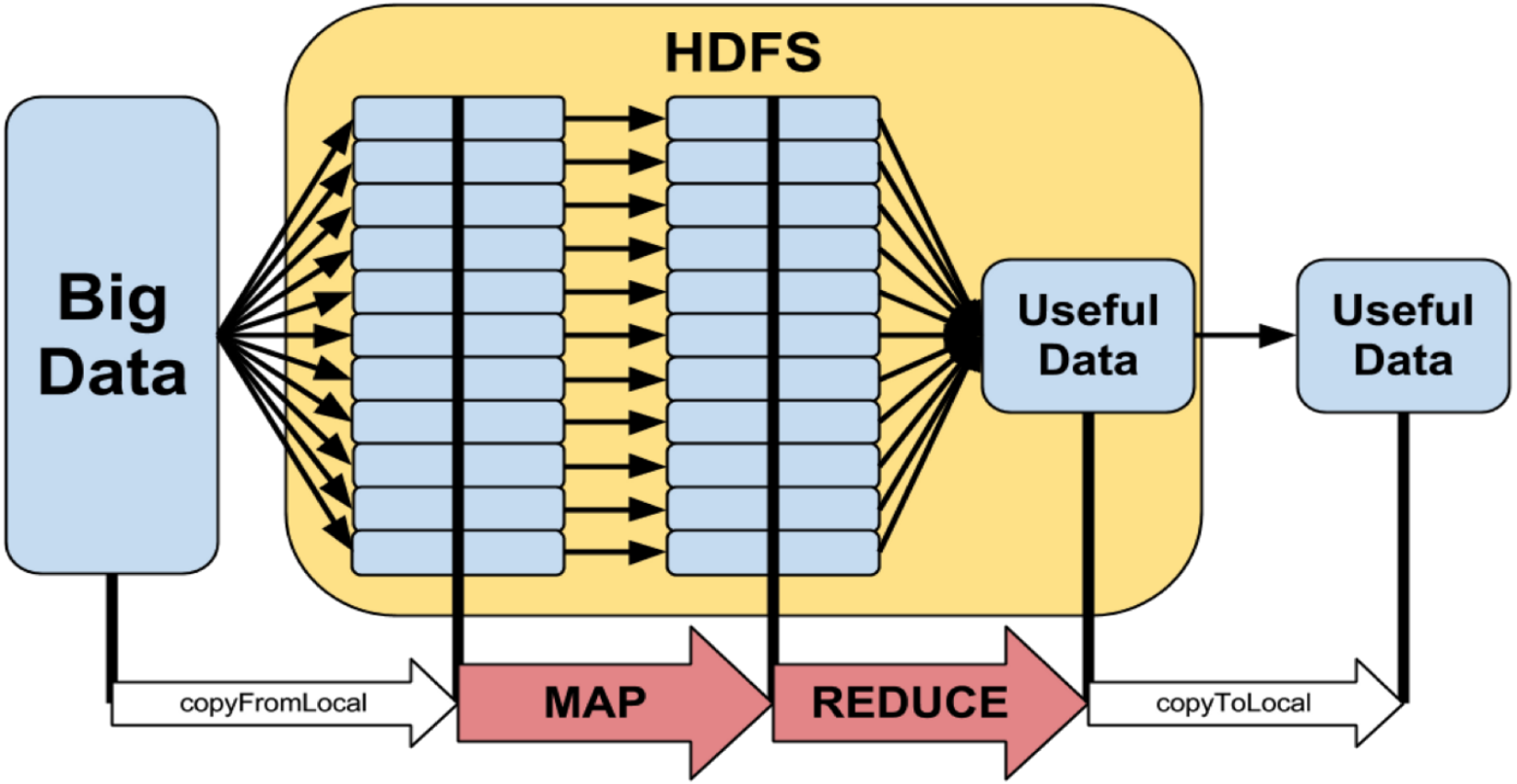
Reducers output:

(fox, 2)

(jumped, 4)

(and, 3)

MapReduce Model: Hadoop Implementation



MapReduce provides:

- Partition data into small chunks
- Automatic parallelization, distribution
- I/O scheduling
- Load balancing
- Network and data transfer optimization
- Fault tolerance
  - Handling of machine failures

MapReduce provides:

- Fault tolerance
  - Handling of machine failures
  - Fault tolerance refers to the ability of a system (computer, network, cloud cluster, etc.) to continue operating without interruption when one or more of its components (disk, computer, …) fail.

MapReduce: Scale-Out, but do NOT Scale-Up

- Need more power: Scale-Out
  - Large number of commodity servers
  - Not expensive to add or replace servers
  - Add more servers any time
- Do NOT Scale-Up
  - **Do NOT use** high end specialized servers
  - Very Expensive
  - Replacement very costly

MapReduce: Scale-Out, but do NOT Scale-Up

- Need more power: Scale-Out
  - Large number of commodity servers
  - Not expensive to add or replace servers
  - Add more servers any time
- A commodity server is **a commodity computer that is dedicated to running server programs and carrying out associated tasks.** In many environments, multiple low-end servers share the workload. Commodity servers are often considered disposable and, as such, are replaced rather than repaired.

MapReduce Implementations

Google App Engine
(proprietary, not open-source)

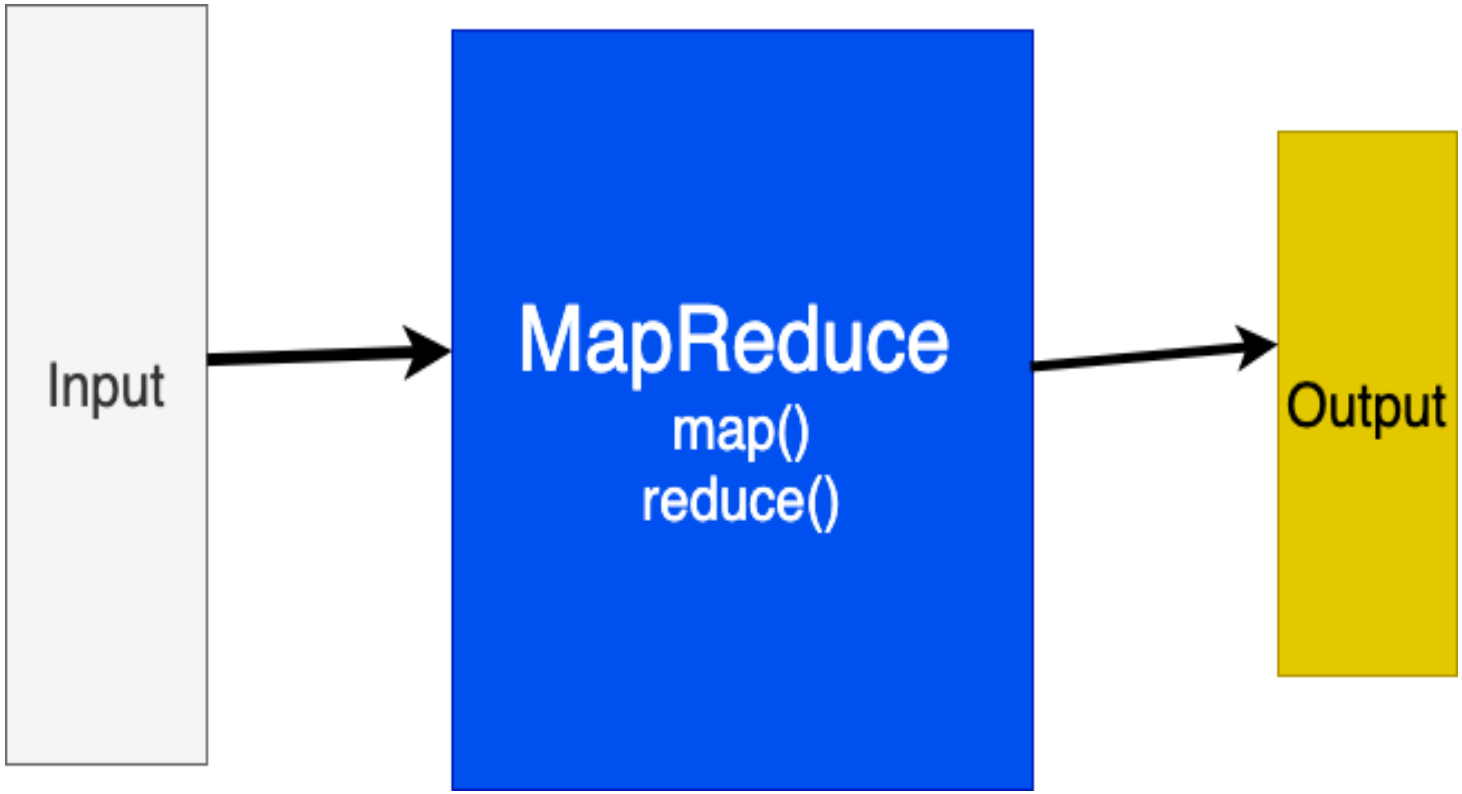
Apache Hadoop:
implementation of MapReduce
(open-source, based on Google's paper)

Apache Spark:
open-source, superset of MapReduce
In-memory computing, very fast

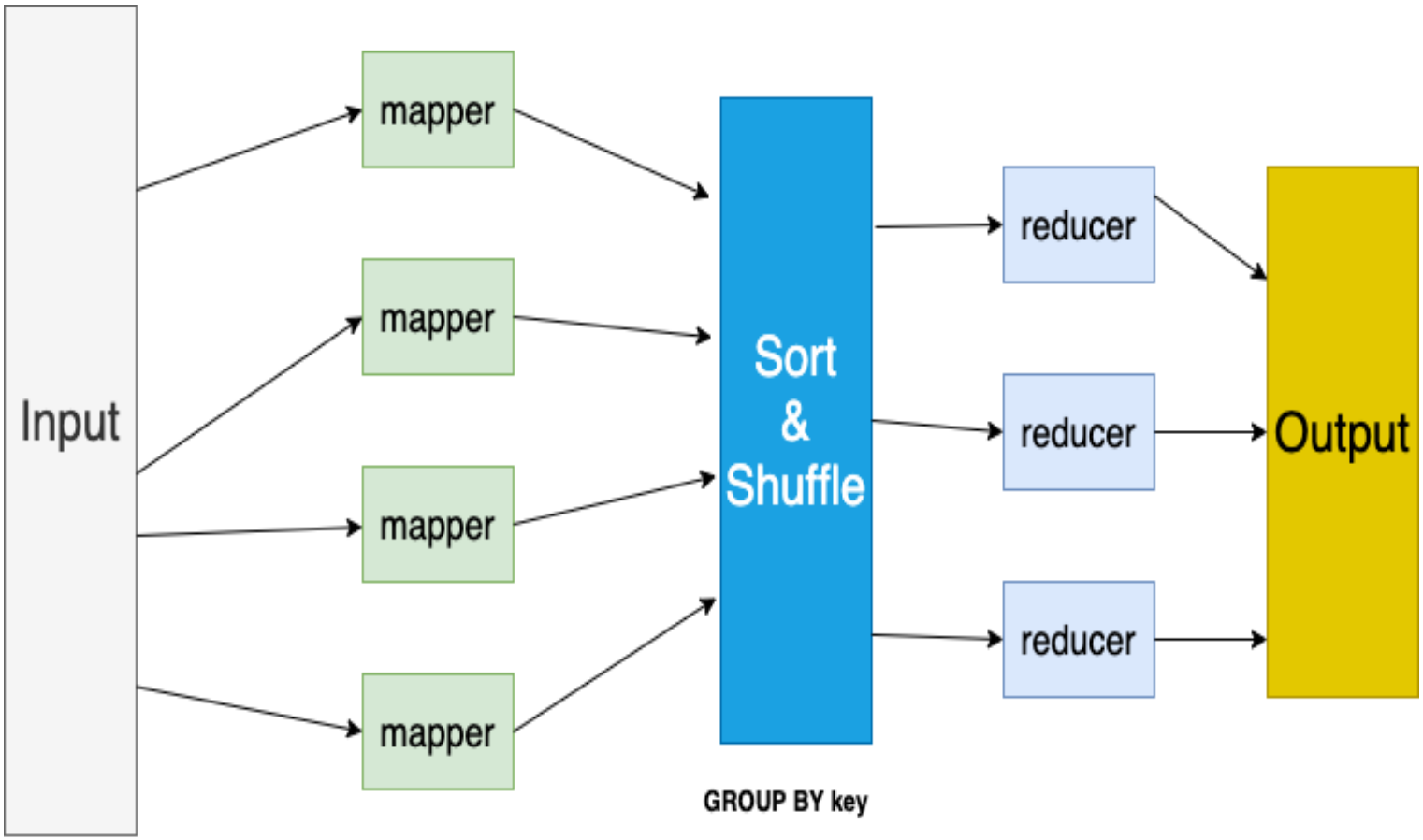
Typical problem solved by MapReduce

- Read a lot of data
- Partition data into small chunks
- map(key, value): extract something you care about from each record
  - output: set of (key2, value2) pairs
- Shuffle and Sort [done by MapReduce Implementation]
  - Output as (key2, [value_21, value_22, value_23, …])
- reduce(key2, values): aggregate, summarize, filter, or transform
  - Output: set of (key3, value3) pairs
- Write the results

MapReduce model



MapReduce model

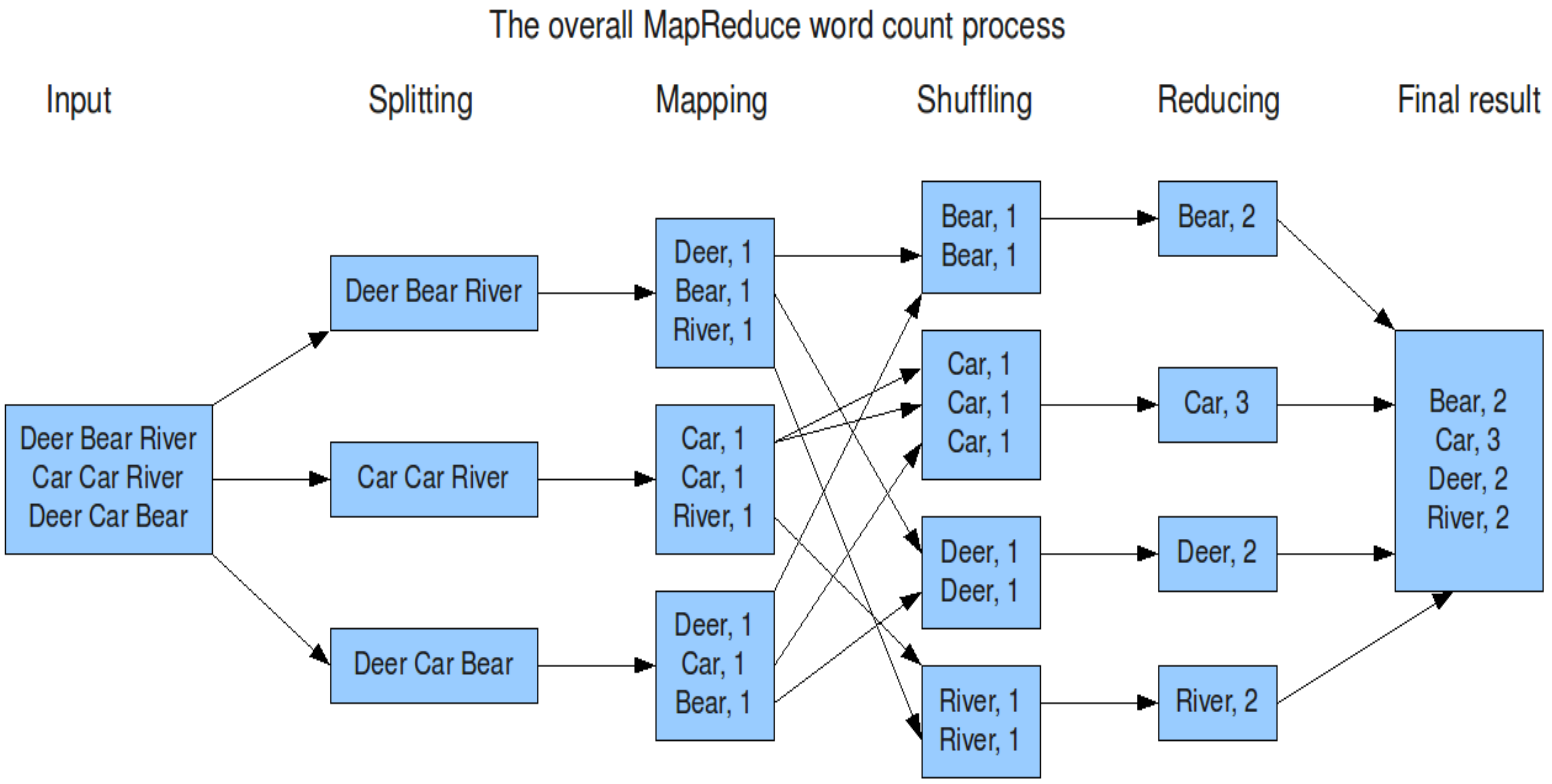


Mappers in parallel

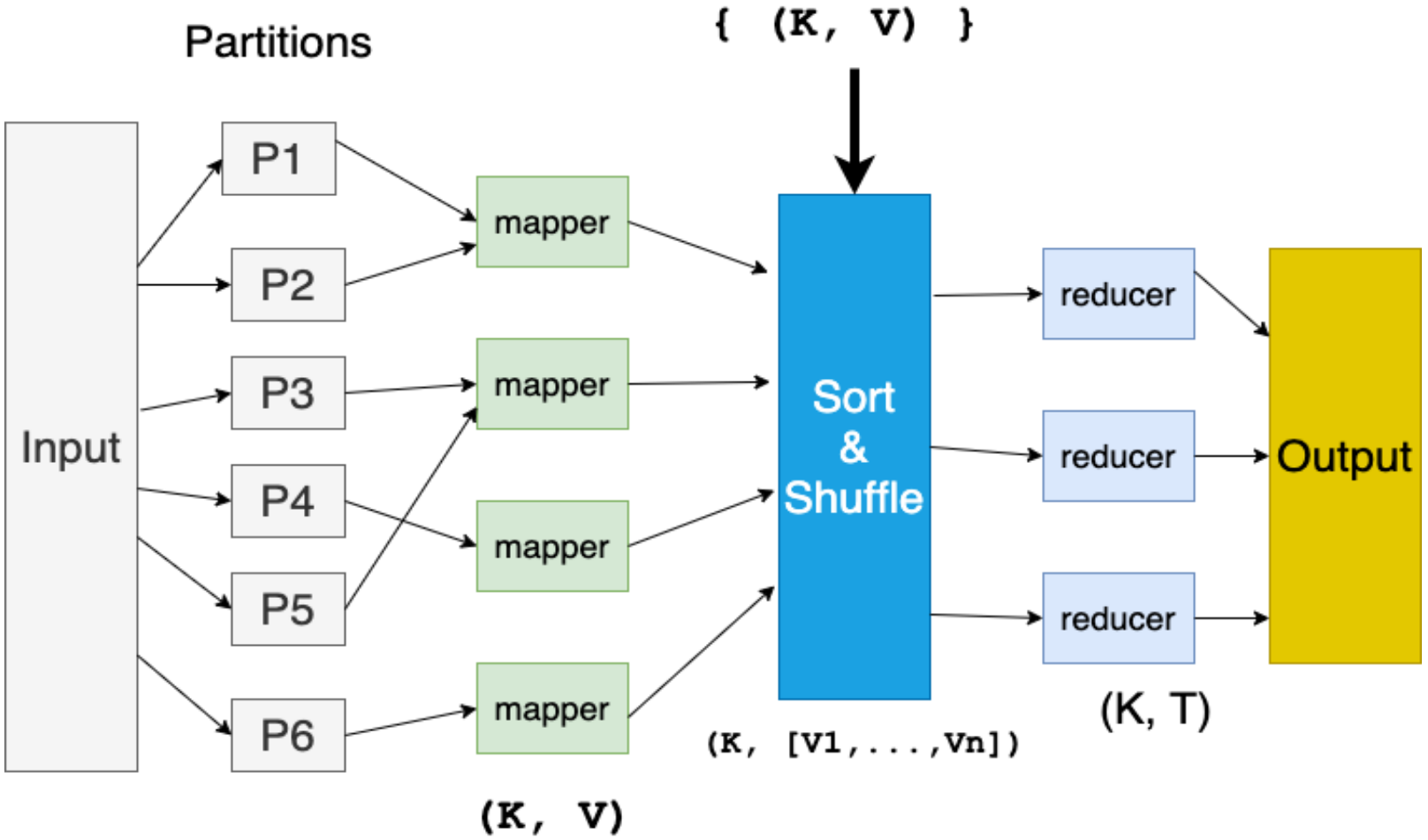
- map() function run in parallel.
- Each mapper operates on a set of chunks assigned to it by the job tracker.
- Mappers write to local disk.

```
# key: a record number or hash of a record
# value: a single record (actual data)
map(key, value):
can emit any number  of (K, V) pairs:
   (K_1, V_1), (K_2, V_2), … (K_n, V_n)
```

MapReduce model



MapReduce model



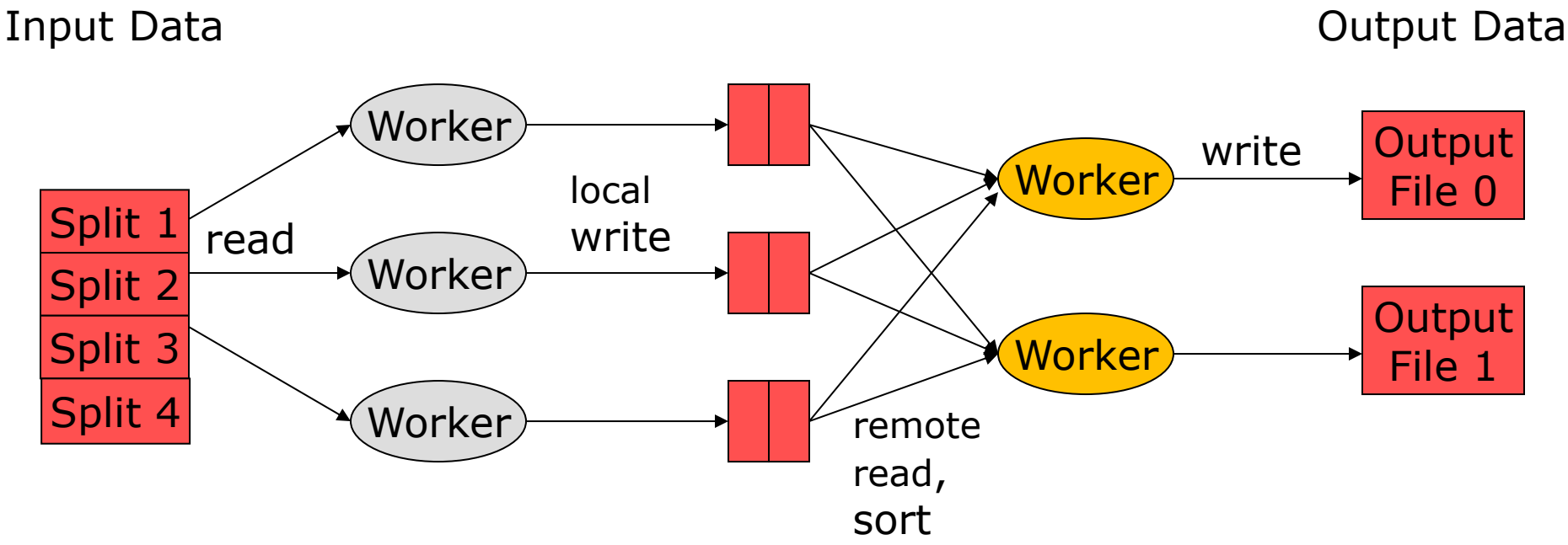
Reducers in parallel

- reduce() function run in parallel.
- Each reducer() operates on
  (key, [V_1, V_2, …, V_n])

- reduce((key, [V_1, V_2, …, V_n]):

  Reducers create final outputs as:
  - (K1, T1)
  - (K2, T2)
  - …

MapReduce workflow



Map

extract something you care about from each record

Reduce

aggregate, summarize, filter, or transform

Mappers and Reducers

- Need to handle more data?
  - Just add more Mappers/Reducers!
- No need to handle multithreaded code ☺
  - Mappers and Reducers are typically single threaded and deterministic
    - Determinism allows for restarting of failed jobs
  - Mappers/Reducers run entirely independent of each other
  - Mappers/Reducers run in separate JVM processes

Word Count Problem: solve it by MapReduce

- INPUT: Given a set of text documents/files
- OUTPUT: Find frequencies of each unique word

- Input: "gray fox red fox jumped over red gray fox"
- Output: dictionary[(word, frequency)]

```
(gray, 2)
(red, 2)
(fox, 3)
(jumped, 1)
(over, 1)
```

Mapper: `map(key, value)`

- Reads in input pair as (Key, Value)
- Outputs a set of pairs (K', V')
  - Let's count number of each word in user queries (or Tweets/Blogs)
  - The input to the mapper will be (Query_ID, Query_Text):

```
<Q1,"the teacher went to the store. the store was
closed; the store opens in the morning. the store
opens at 9am." >
```

  - The mappers output would be:

```
(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)
```

Sort & Shuffle: SQL's GROUP BY

- Accepts the Mapper output, and aggregates values on the key
  - For our example, the mappers output would be:

    (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)

  - The output of Sort & Shuffle would be:

    (the, [1, 1, 1, 1, 1, 1])

    (store, [1, 1, 1])

    …

Reducer: `reduce(key, values)`

- Accepts the Sort & Shuffle output,
- and aggregates values on the key
  - Input to reducer: (store, [1, 1, 1])
    - Output: (store, 3)
  - Input to reducer : (the, [1, 1, 1, 1, 1, 1])
    - Output: (the, 6)
  - …

MapReduce Job Components

1. **Input path** (identify your input files)
2. **Output path** (where to write output)
3. **map()** function

```
map(key, value)
  emits {(K2, V2), …}
```

- 3.5 [Sort & Shuffle is done by MapReduce]

4. **reduce()** function

```
reduce(K2, [value_1, value_2, …])
  emits {(K3, V3), …}
```

5. OPTIONAL **combine()** function

MapReduce Job: Input Path

- Input path: 3 files will be read
- Example: `s3://my_bucket/project7/`

```
s3://my_bucket/project7/file1.txt
s3://my_bucket/project7/file2.txt
s3://my_bucket/project7/file3.txt
```

MapReduce Job: Output Path
Reducers output will be written to output path

- Output path:
- Example: `s3://my_bucket/output7/`

```
s3://my_bucket/output7/_SUCCESS
s3://my_bucket/output7/part1
s3://my_bucket/output7/part2
s3://my_bucket/output7/part3
s3://my_bucket/output7/part4
```

MapReduce Job: map() function for Word Count

```
# pseudo-code
# key: may be a record number and ignored here
# value: a single record of your input data
# "fox jumped and jumped"
map(key, value) {
  # tokenize record
  words = value.split(" ")
  #  array index: 0    1        2           3
  # words = ["fox", "jumped", "and", "jumped"]
  for word in words {
    emit (word, 1)
  }
}
```

MapReduce Job: map() function

```
# pseudo-code
# key: may be a record number and ignored here
# value: a single record of your input data
# "fox jumped and jumped"
map(key, value) {
  # tokenize record
  words = value.split(" ")
 # array index: 0    1        2           3
 # words = ["fox", "jumped", "and", "jumped"]
  for word in words {
    emit (word, 1)
  }
}
```

Output of a mapper:

```
(fox, 1)
(jumped, 1)
(and, 1)
(jumped, 1)
```

MapReduce Job: map() function

```
# 103 is a record number
INPUT: (103, “a fox of jumped over red fox and jumped”)
```

Mappers output:

```
(a, 1)
(fox, 1)
(of, 1)
(jumped, 1)
(over, 1)
(red, 1)
(fox, 1)
(and, 1)
(jumped, 1)
```

MapReduce Job: map() function
FILTER: Ignore words with length of less than 3 Chars.

```
# pseudo-code
# key: may be a record number
# value: the entire record such
# as "fox jumped and jumped"
map(key, value) {
   words = value.split(" ")
   for word in words {
      # filter non-desired words
      if (len(word) > 2) {
         emit( word, 1)
      }
   }
}
```

MapReduce Job: map() function
Ignore words with length of less than 3 Chars.

INPUT: (103, “a fox of jumped over red fox and jumped”)

Mappers output:

(fox, 1)

(jumped, 1)

(over, 1)

(red, 1)

(fox, 1)

(and, 1)

(jumped, 1)

NOTE: FILTERING: “a” and “of” were dropped
since their length is less than 3

MapReduce Job: map() function
FILTER-1: Ignore records with length of less than 80 chars
FILTER-2: ignore words less than 3 characters

```
# pseudo-code
# key: may be a record number
# value: the entire record such as "fox jumped and jumped"
map(key, value) {
  # filter-1
  if (len(value) < 80) {
    # no (K, V) is emitted at all
    return
  }
  words = value.split(" ")
  for word in words {
    # filter-2
    if (len(word) > 2) {
        emit( word, 1) # filter non-desired words
    }
  }
}
```

MapReduce Job: map() function

```
# key: may be a record number such as 100 (is ignored)
# value: the entire record such as
# (103, "fox jumped and jumped")
map(key, value) output:
    (fox, 1)
    (jumped, 1)
    (and, 1)
    (jumped, 1)
```

MapReduce Job: map() function

```
# key: may be a record number such as 1234 (is ignored)
# value: the entire record such as
# (1234, "fox jumped over fox")
map(key, value) output:
    (fox, 1)
    (jumped, 1)
    (over, 1)
    (fox, 1)
```

Sort & Shuffle: Receives Mappers Output

- **Sort & Shuffle is the genie of MapReduce**
- **Similar to SQL's "GROUP BY"**
- **Output of Sort & Shuffle:**

```
(key_1, [V_11, V_12, …])
(key_2, [V_21, V_22, …])
…
(key_N, [V_N1, V_N2, …])
```

- Therefore, N reducers are required.

Input to Reducers

```
(key_1, [V_11, V_12, …])
(key_2, [V_21, V_22, …])
…
(key_N, [V_N1, V_N2, …])
```

NOTE:

- All keys {key_1, key_2, …, key_N} are unique
- A key may have any number of values
- Values are not sorted (they can be in any order)

MapReduce Job Components: reduce()

- Reducers receive output of Sort & Shuffle phase.
- reduce(key, values) accepts a
  - key: a single unique key
  - values: [V_1, V_2, …, V_n]
- Creates any number of new (K′, V′) pairs

MapReduce Job: reduce() function for Word Count

```
# pseudo-code
# handle (key, [v1, v2, …])
# key: is a unique word
# values: Iterable<Integer>
#     such as [1, 1, …, 1]
reduce(key, values) {
  count = 0
  # iterate values
  for v in values {
    count += v
  }
  emit(key, count)
}
```

MapReduce Job: reduce() function + FILTER: ignore words with frequencies of less than 5

```
# pseudo-code
# key: is a unique word
# values: [1, 1, …, 1]
reduce(key, values) {
   count = 0
   for v in values {
      count += v
   }
   # filter
   if (count >= 5) {
      emit(key, count)
   }
}
```

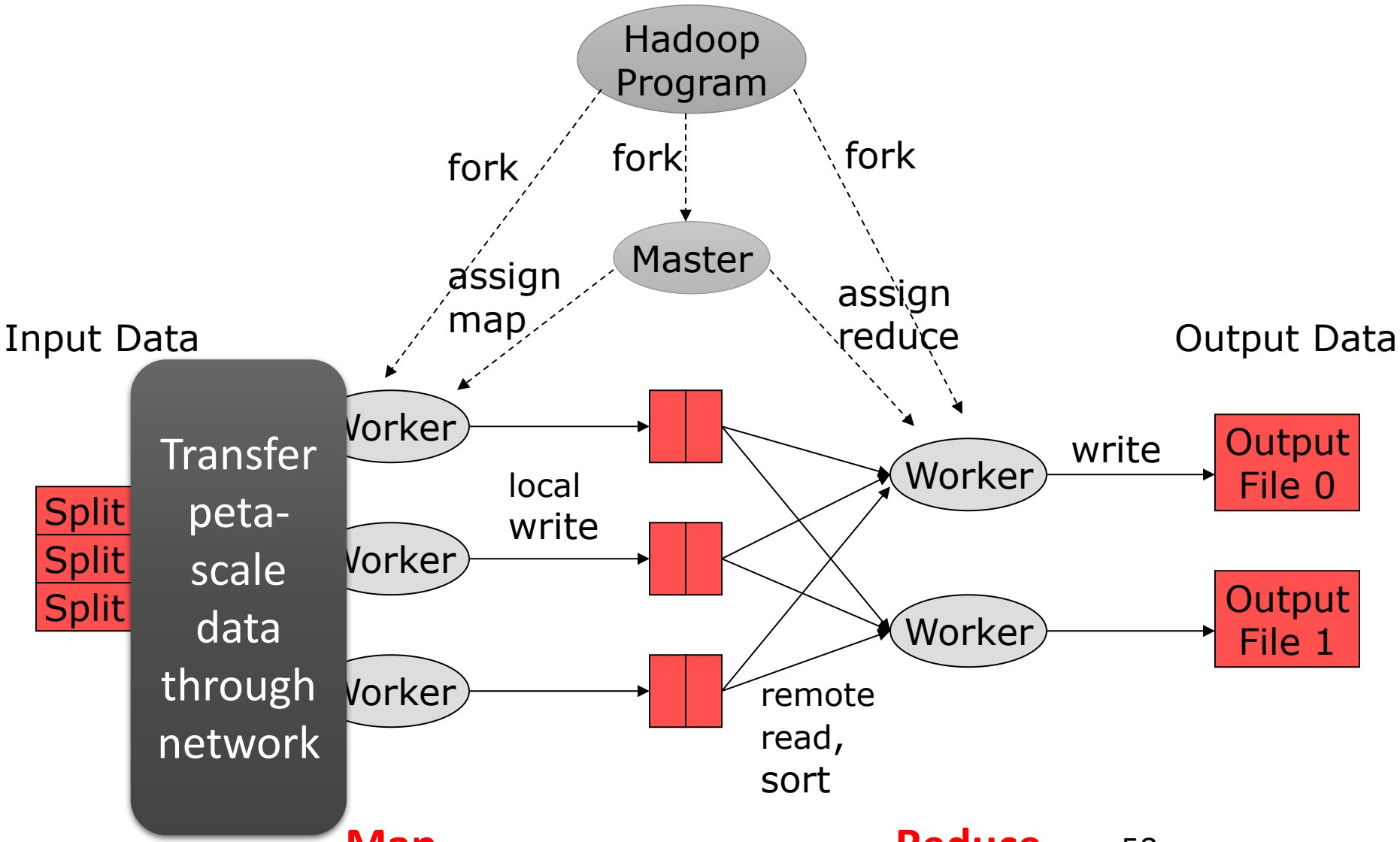
MapReduce Job: reduce() function + what if we want to ignore words with frequencies of less than 5
Ignore words with length of less than 3 Chars.

```
# pseudo-code
# key: is a unique word
# values: [1, 1, …, 1]
reduce(key, values) {
  # not a proper filter for reducer (should be done in mapper)
  if (len(key) <= 2) {
     return
  }
  count = 0
  for v in values {
    count += v
  }
  # proper filter
  if (count < 5) {
    return
  }
  else {
    emit(key, count)
  }
}
```

MapReduce Job: reduce() function and output length of key

```
# pseudo-code
# key: is a unique word
# values: [1, 1, …, 1]
reduce(key, values) {
  count = 0
  for v in values {
    count += v
  }
  # create a new composite value
  new_value = (len(key), count)
  emit(key, new_value)
}
```

MapReduce


Hadoop Program
fork
fork
fork
Master
assign map
assign reduce
Input Data
Output Data
Split
Split
Split
Transfer peta-scale data through network
Worker
Worker
Worker
local write
Worker
Worker
write
Output File 0
Output File 1
remote read, sort
Map
Reduce

Failure in MapReduce

- Failures are norm in commodity hardware
- **Worker** failure
  - Detect failure via periodic heartbeats
  - Re-execute in-progress map/reduce tasks
- **Master** failure
  - Single point of failure; Resume from Execution Log
- **Robust**
  - Google's experience: lost 1600 of 1800 machines once!, but finished fine

```
public class WordCount {

  public static class Map extends MapReduceBase implements
              Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable>
                    output, Reporter reporter) throws IOException {
      String line = value.toString();
      StringTokenizer tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken());
        output.collect(word, one);
  }}}

  public static class Reduce extends MapReduceBase implements
              Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text,
                       IntWritable> output, Reporter reporter) throws IOException {
      int sum = 0;
      while (values.hasNext()) { sum += values.next().get(); }
      output.collect(key, new IntWritable(sum));
  }}

  public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);
    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    JobClient.runJob(conf);
  }}
```

Mapper

Reducer

Run this program as a MapReduce job

Complete Word Count Solution in MapReduce/Hadoop

Word Count Solution in MapReduce/Hadoop

- Driver Program
- Mapper Program
- Reducer Program

Summary: MapReduce

- Programming paradigm for data-intensive computing
- Distributed & parallel execution model
- Simple to program
- The MapReduce framework automates many tedious tasks:
  - Data partitioning
  - Machine selection,
  - Failure handling
  - Sort & Shuffle

Word Count in Hadoop/MapReduce

- We will NOT study Hadoop
- For reference, I am including a pointer to MapReduce Tutorial in Hadoop, which can help you to understand an implementation of MapReduce.