An intro to

Map-Reduce

Romaric Duvignau
Associate Professor
Computer & Network Systems, CSE
Slides from Ahmed Ali-Eldin (slightly updated)

CHALMERS
UNIVERSITY OF TECHNOLOGY

Distributed Computing and Systems
Chalmers university of technology

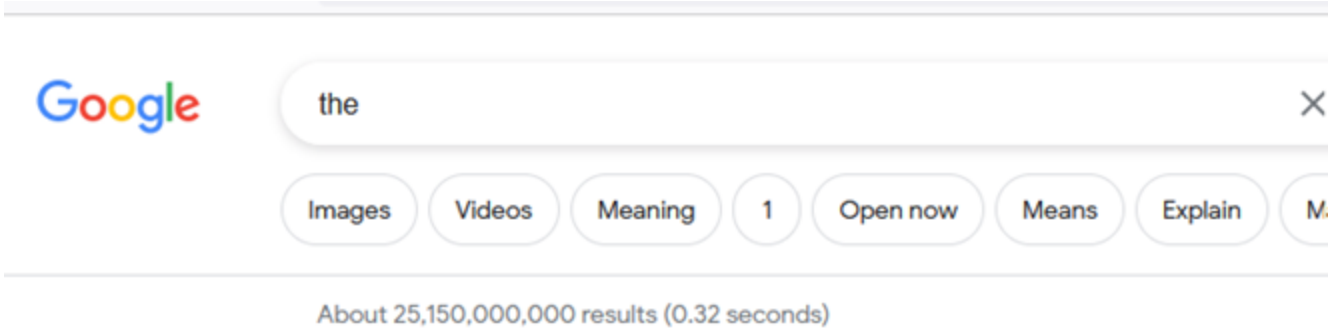
Example: Counting Words

- **Count the most popular terms on the Internet:**
- Sounds easy
- Why not this?

```
CountWords(internet):
  hash_map<string, int> word_count;
  for each page in internet:
     for each word in page:
        word_count[word] += 1
```

Motivation: Large-Scale Data Processing

- I searched for the word “the”



Motivation: Large-Scale Data Processing

- All the web = ~40 billion* web pages x 2.5MB** =
  from 100 Petabytes (1 PB = 1000 TB) up to 100 ZB (total information)
- The largest hard disk money can buy today is a 125 TB, but that is very rare, so we go for the more standard SSD 30 TB
- ~3000 hard drives just to store the small web terabytes
- One computer can read up to 1 Gbps ~ 2.63 Pb/month from disk
  - ~38 months to read the hard drives sequentially from a single machine with no failures
    - (but Google gave me an answer in 0.32s!)

*https://www.worldwidewebsize.com/

**https://www.pingdom.com/blog/webpages-are-getting-larger-every-year-and-heres-why-it-matters/

REMINDER: Don't use AI to do your 5th grade Maths!

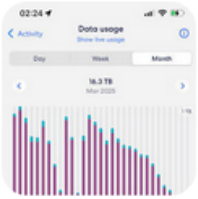
Google

1 gbps in one month

Mode IA | Tous | Images | Produits | Vidéos | Actualités | Vidéos courtes | Plus | Outils

Aperçu IA

1 Gbps is equal to approximately **2.63 terabits per month**, or about 3.24×10^{14} bytes per month. To convert this, first calculate the number of seconds in a month (around 2,629,744) and multiply that by 1 Gbps (1 billion bits per second).



Calculations

- **Seconds in a month:**
  $30.44 \text{ days} \times 24 \text{ hours/day} \times 60 \text{ minutes/hour} \times 60 \text{ seconds/minute} \approx 2,629,744 \text{ second}$
- **Gigabits per month:**
  $1 \text{ Gbps} \times 2,629,744 \text{ seconds} = 2,629,744,000,000 \text{ bits/month}$
- **Terabits per month:**
  $2,629,744,000,000 \text{ bits} \div 1000 = 2,629,744 \text{ gigabits} = 2,629.744 \text{ terabits}$
- **Bytes per month (Base 10):**
  $1 \text{ Gbps} \times 2,629,744 \text{ seconds} \div 8 \text{ bits/byte} = 328,718,000,000 \text{ bytes/month} = 3.287 \times$

Large-Scale Data Processing

1. **Problems**:
    1. Time
    2. Errors / Problems

2. **Solution**:
    1. Distribute processing on many machines

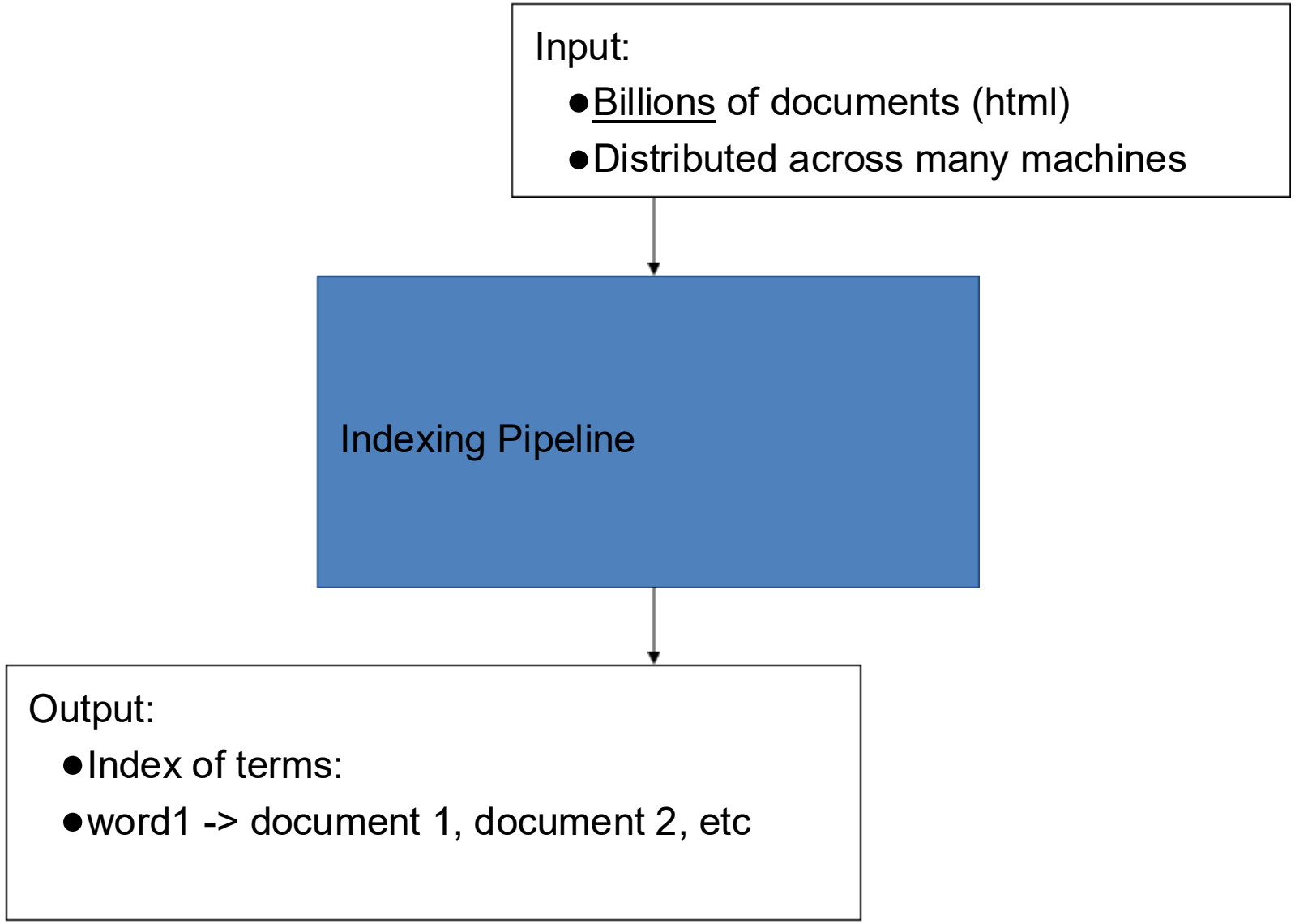
Typical things companies do

1. **Indexing algorithms**
   a) Which web pages contain a certain word

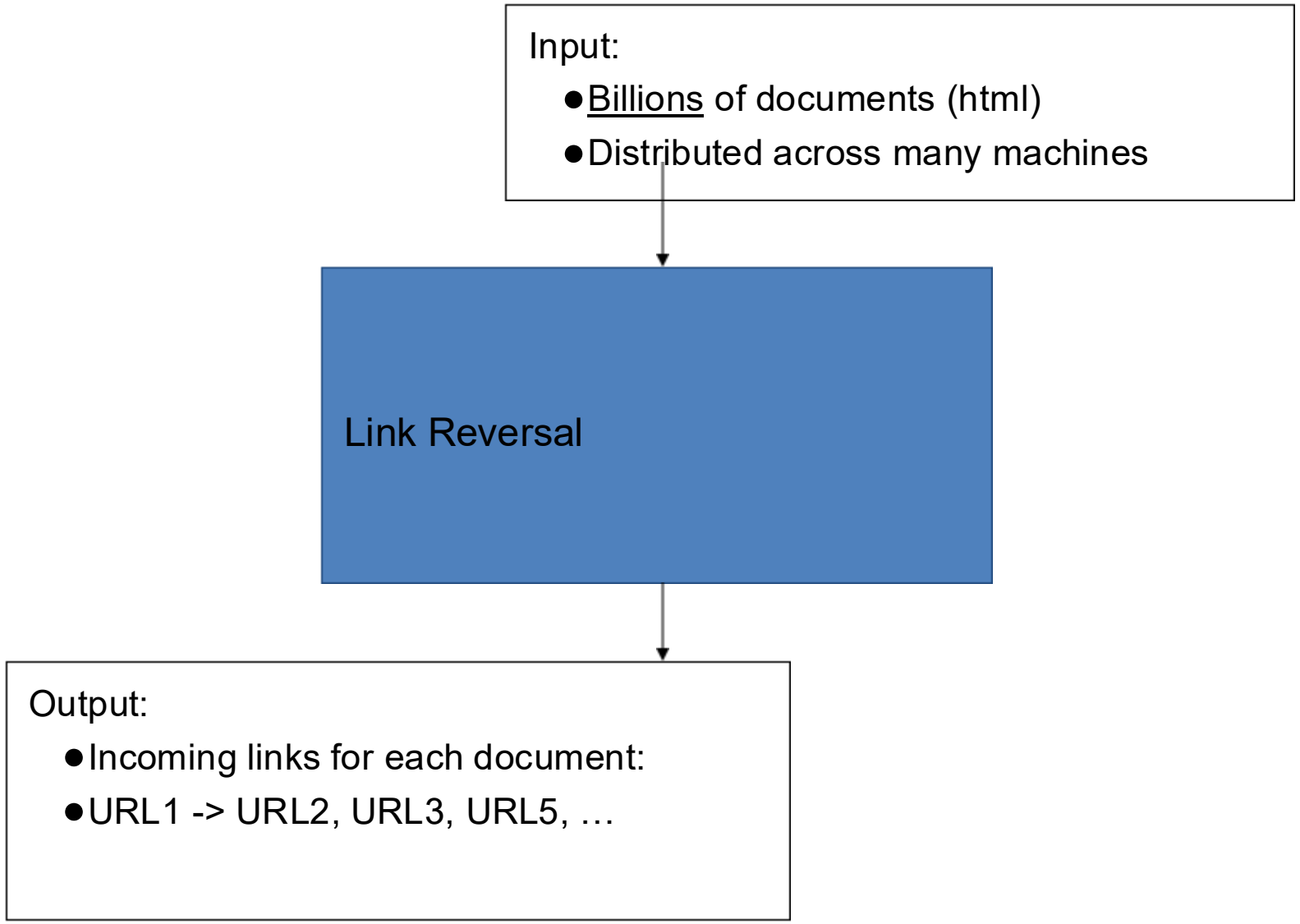
2. **Link Reversal**
   a) Which pages link to this page
      - For example
        - “https://www.cse.chalmers.se/~duvignau/” ← ”Chalmers, Google scholar, linkedin, Facebook, twitter, etc”

3. **Sorting large data alphabetically**

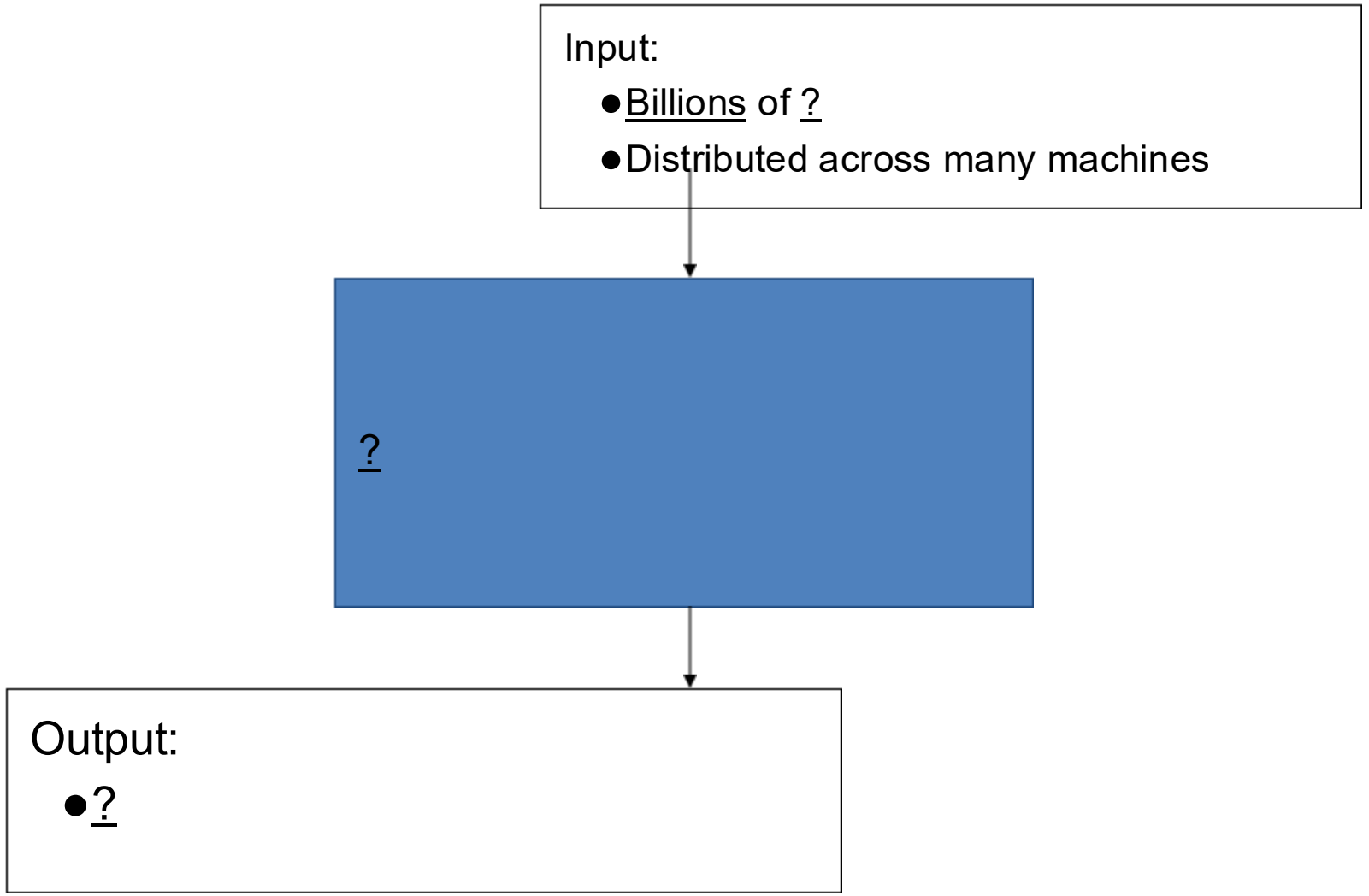
Indexing Algorithm



Link Reversal



Any Algorithm



Before MapReduce

- Each team needed to understand distributed computing
- Extra code
- Extra complexity
- ....

Algorithm

- Requirements:
- Has to be scalable (terabytes of data)
- Has to be fast
- Has to be fault-tolerant (data & machines)
- Has to be easy to implement and flexible

- The solution:
- **MapReduce: a distributed algorithm**

MapReduce: Introduction

- ***"MapReduce is a programming model and an associated implementation for processing and generating large data sets."***

- Can run on thousands of machines
- Can process terabytes of information (input / output)
- It is fault-tolerant
- It is easy to implement
- It is flexible

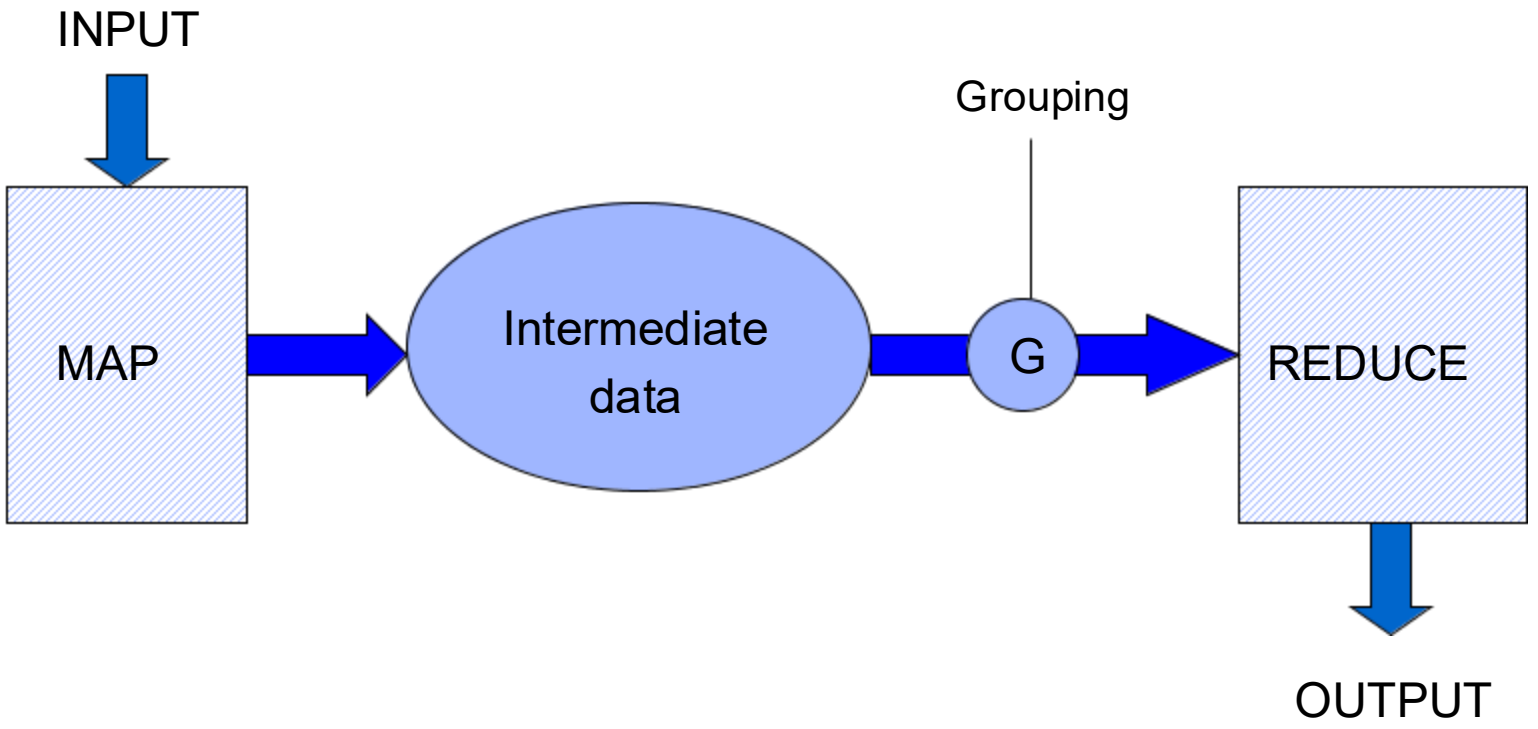
MapReduce Model

- Programming model inspired by functional language primitives, e.g., LISP

1. Read a lot of data
2. **Map**: extract something you care about from each record
3. Shuffle and Sort
4. **Reduce**: aggregate, summarize, filter, or transform
5. Write the results

- Outline stays the same, map and reduce change to fit the problem

MapReduce: Logical Diagram



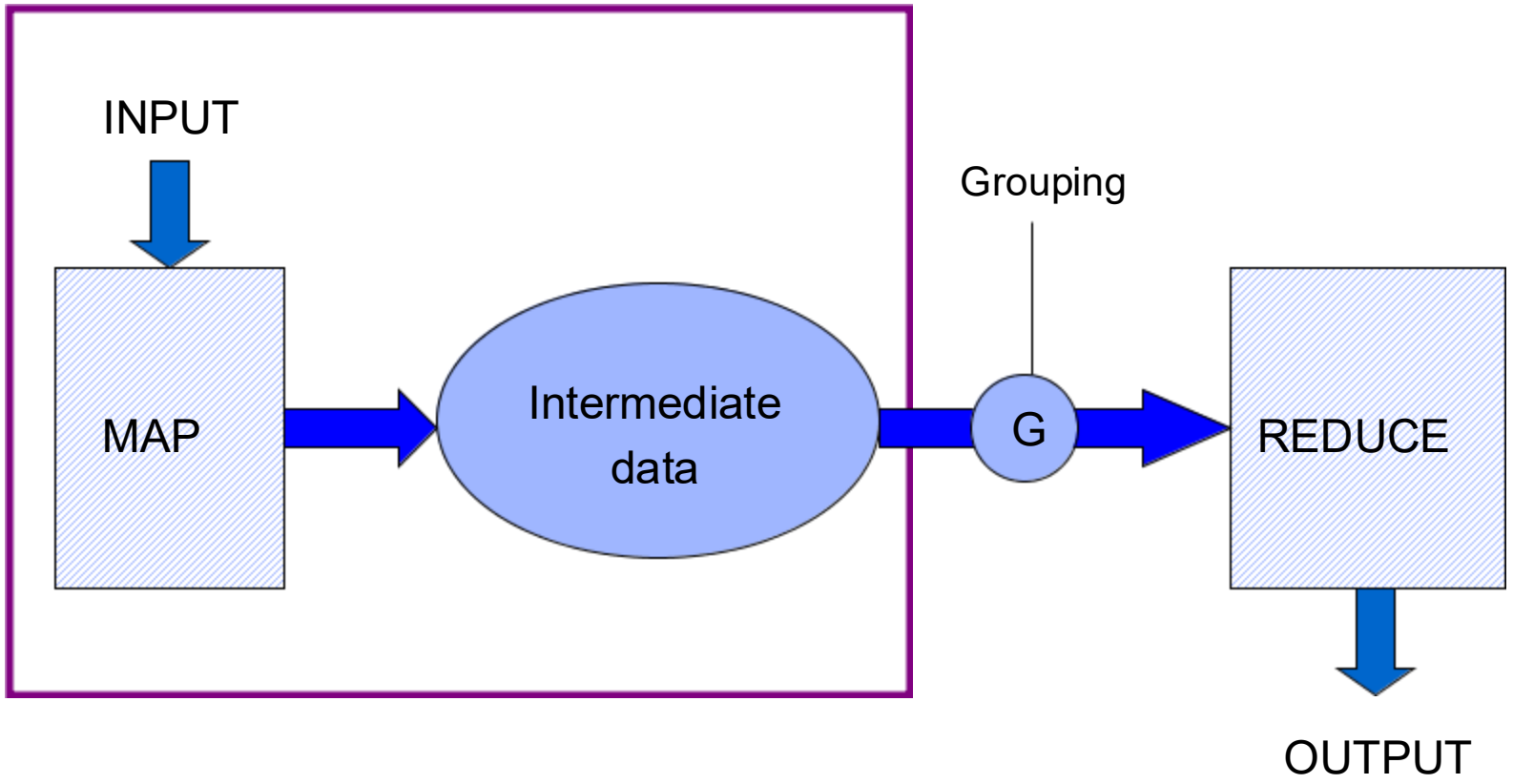
Programming Model

- 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
- GROUPING
  - Done by the system
- 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)

MapReduce: Logical Diagram



MapReduce: Map Phase

- First, the **MAP** function:
- Programmer supplies this function
  - map (in_key, in_value)

  → list(out_key, intermediate_value)
- Input = any pair (e.g., URL, document)
- Output = list of pairs (key/value)

MapReduce: Map Example

- Example: word counting
  - <**URL**, document text> → Output: list(<word, count>)

- E.g. for one document
  - <**URL1**, “Chalmers university is the best university”>
  - Emit → list(<Chalmers, 1>, <university, 1>, <is, 1>, <the, 1>, <best, 1>, <university, 1>)

- For another document
  - <**URL2**, "UNIX is simple and elegant">
  - Emit → list(<unix, 1>, <is, 1>, <simple, 1> . . .)

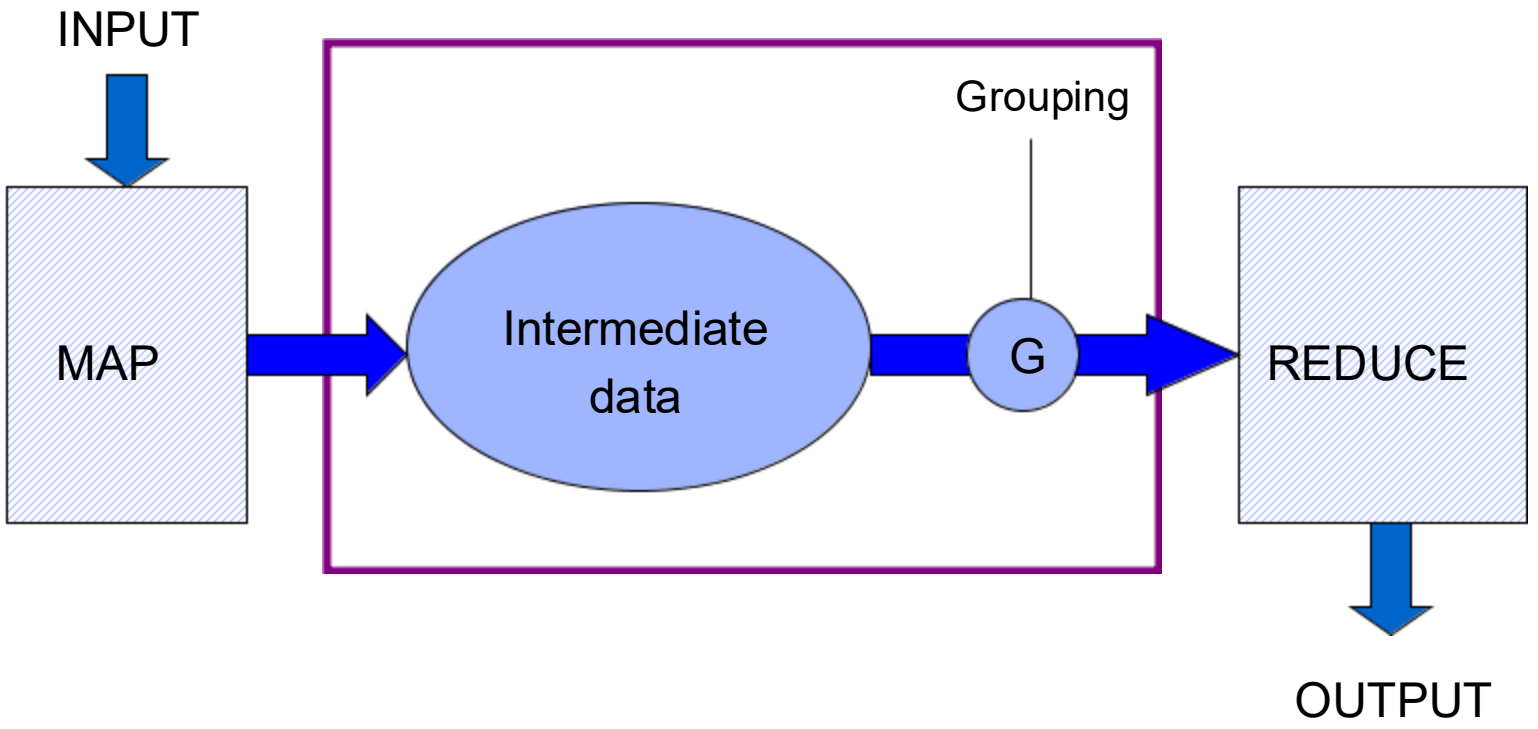
MapReduce: Map Example

- Example: word counting
  - <**URL**, document text> • Output: list(<word, count>)

- Code:

```
Map(string input_key, string input_value):
// key: document URL
// value: document text
  for each word in input_value:
    OutputIntermediate(word, “1”);
```

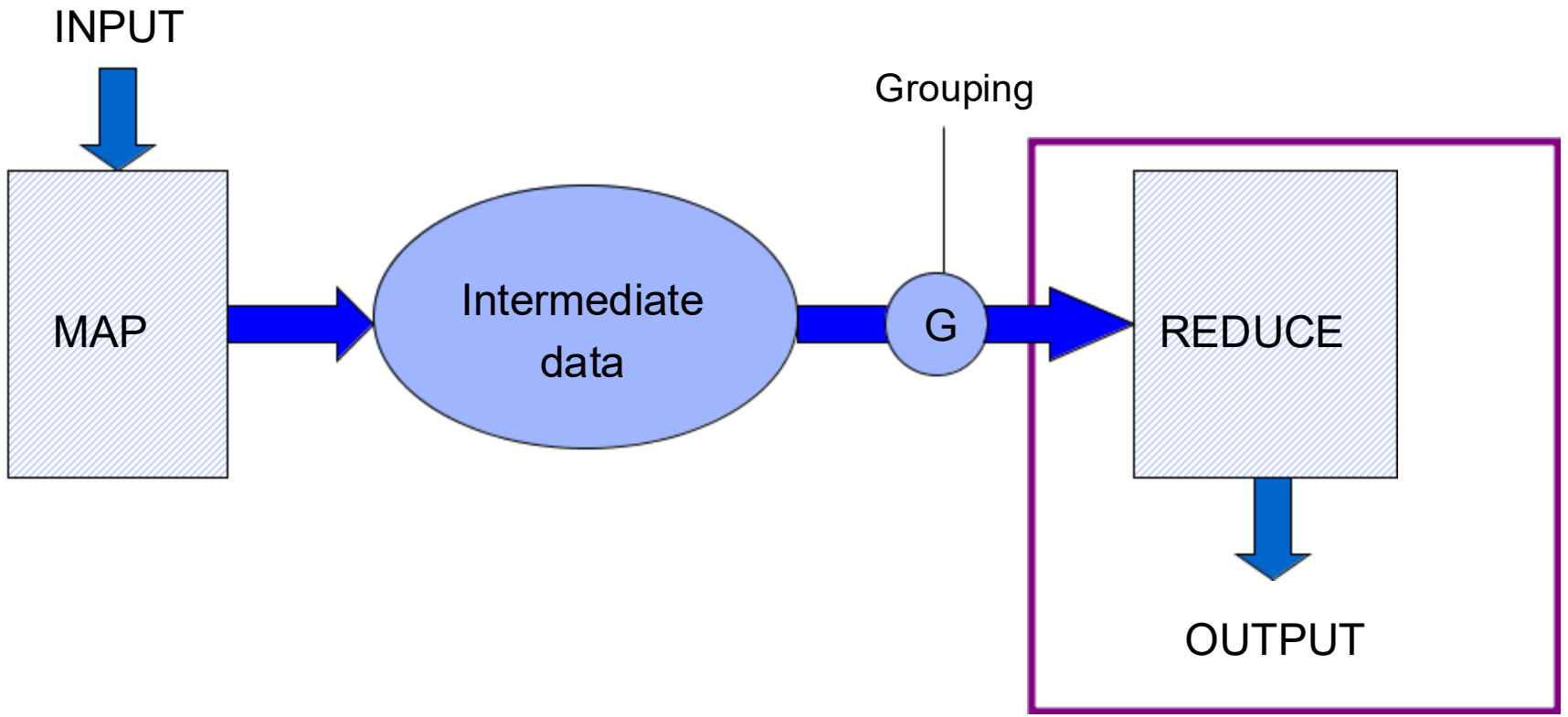
MapReduce: Logical Diagram



MapReduce: Grouping

- The **GROUPING** (or **shuffling**)
- This happens automatically (by the library)
- Put all values for same key together
- For our example:
  - <Chalmers, list(1)>
  - <university, list(1, 1)>
  - <the, list(1)>
  - <is, list(1, 1)>
  - <best, list(1)>
  - ...

MapReduce: Logical Diagram



MapReduce: Reduce

- Last phase is **REDUCE**
- Takes the output from **grouping:**
- reduce (out_key, list(intermediate_value)) → list(out_value)

MapReduce: Reduce Example

- Example (word-counting):
  - <university, list(1, 1)> • <university, 2>

```
Reduce(string key, List values):
// key: word, same for input and output
// values: list of counts
  int sum = 0;
  for each v in values:
     sum += v;
  Output(sum);
```

MapReduce: Word Count

- Example's summary: Want to count how many times each word occurs:
  - **MAP:**
    - Input: <**DocumentId**, Full Text>
    - For each time a word occurs:
      - Output: <Word, 1>
  - **REDUCE:**
    - Input: <Word, list(1, 1, …)>
    - Sum over each word:
      - Output <Word, Count>

MapReduce: Grep (search)

- Want to search terabytes of documents for where a word occurs:
  - **MAP:**
    - Input: <**DocumentId**, Full Text>
    - For each line where the word occurs:
      - Output: <**DocumentId**, LineNumber>
  - **REDUCE:**
    - Input: <**DocumentId**, list(LineNumber, ...)>
    - Do nothing (*Identity* function, i.e. output = input)

MapReduce: Link Reversal

- Want to build a reverse-link index:
  - **MAP:**
    - Input: <**source_URL**, Full Text>
    - For each outgoing link <a href='(*destination_URL*)'>:
      - Output: <destination_URL, source_URL>
  - **REDUCE:**
    - Input: <destination_URL, list(source_URL, ...)>
    - Join source_URLs
      - Output: for each URL, a list of all other URLs that link to it.

MapReduce Usage

- Index construction
- Web access log stats, e.g., search trends (Google Zeitgeist)
- Distributed grep
- Distributed sort
- Most popular terms on the Internet
- Document clustering (news, products, etc.)
- Find all the pages that link to each document (link reversal)
- Machine learning
- Statistical machine translation
- …

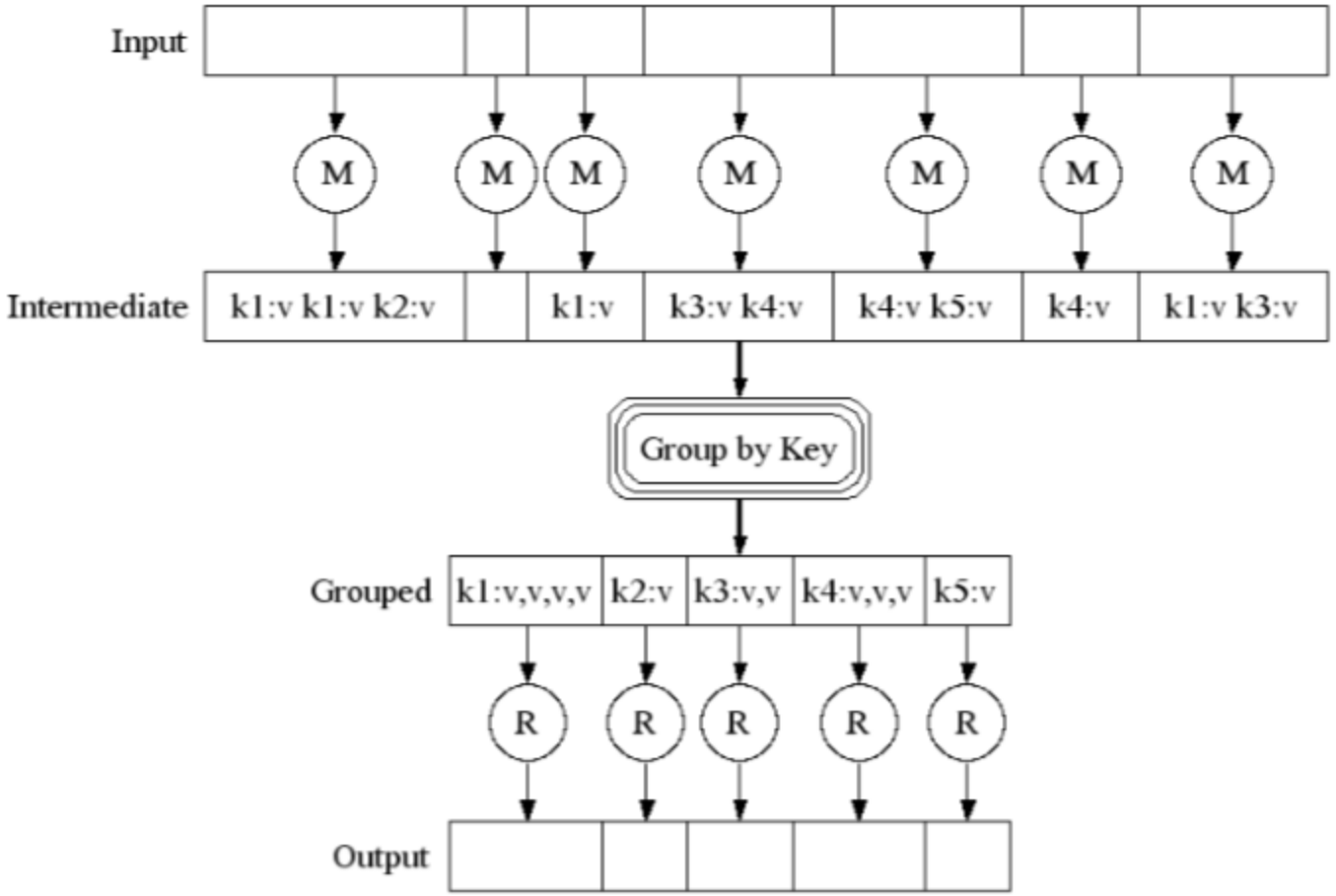
Why complicate things?

MapReduce: Parallelism

- Each Map call is independent (parallelize)
  - Run **M** different **Map tasks**
    (each takes a chunk of input; make use of locality!)

- Each Reduce call is independent (parallelize)
  - Run **R** different **Reduce tasks**
    (each produces a chunk of output; there are R output files.)

MapReduce: Parallelism



Framework

- User:
  - **Map** function (used in Map Phase)
  - **Reduce** function (used in Reduce Phase)
  - Options (Map tasks, Reduce tasks, Workers, Input, Output)
- Framework:
  - Execute map and reduce phase (distributedly)
  - Collect/move data:
    - input
    - intermediate
    - output data
  - Handle faults
  - Dynamic load-balancing and scheduling

MapReduce: Scheduling

- **One coordinator node C, many workers W_1, W_2, etc**
  - Input data split into *M* map tasks (16-64MB in size)
  - Reduce phase partitioned into *R* reduce tasks
  - Tasks are assigned to workers dynamically
  - Often: *M*=200,000; *R*=4,000; workers=2,000

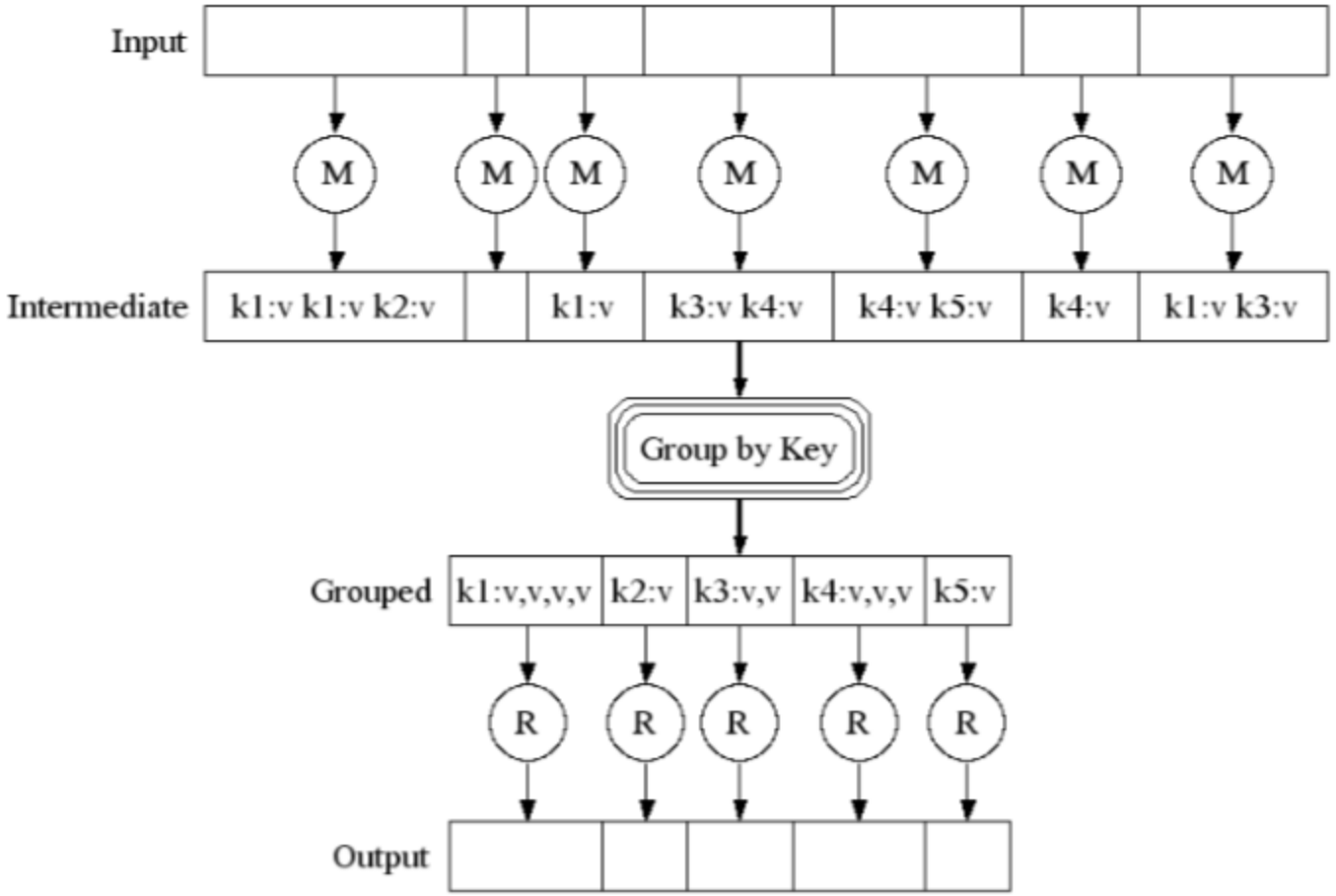
- **Coordinator assigns each map task to a free worker**
  - Considers locality of data to worker when assigning task
  - Worker reads task input (often from local disk! or *Distributed File System*!)
  - Worker produces R **local files** (splill files) containing intermediate k/v pairs

- **Coordinator assigns each reduce task to a free worker**
  - Worker reads intermediate k/v pairs from map workers
  - Worker sorts & applies user's *Reduce* operation to produce the output

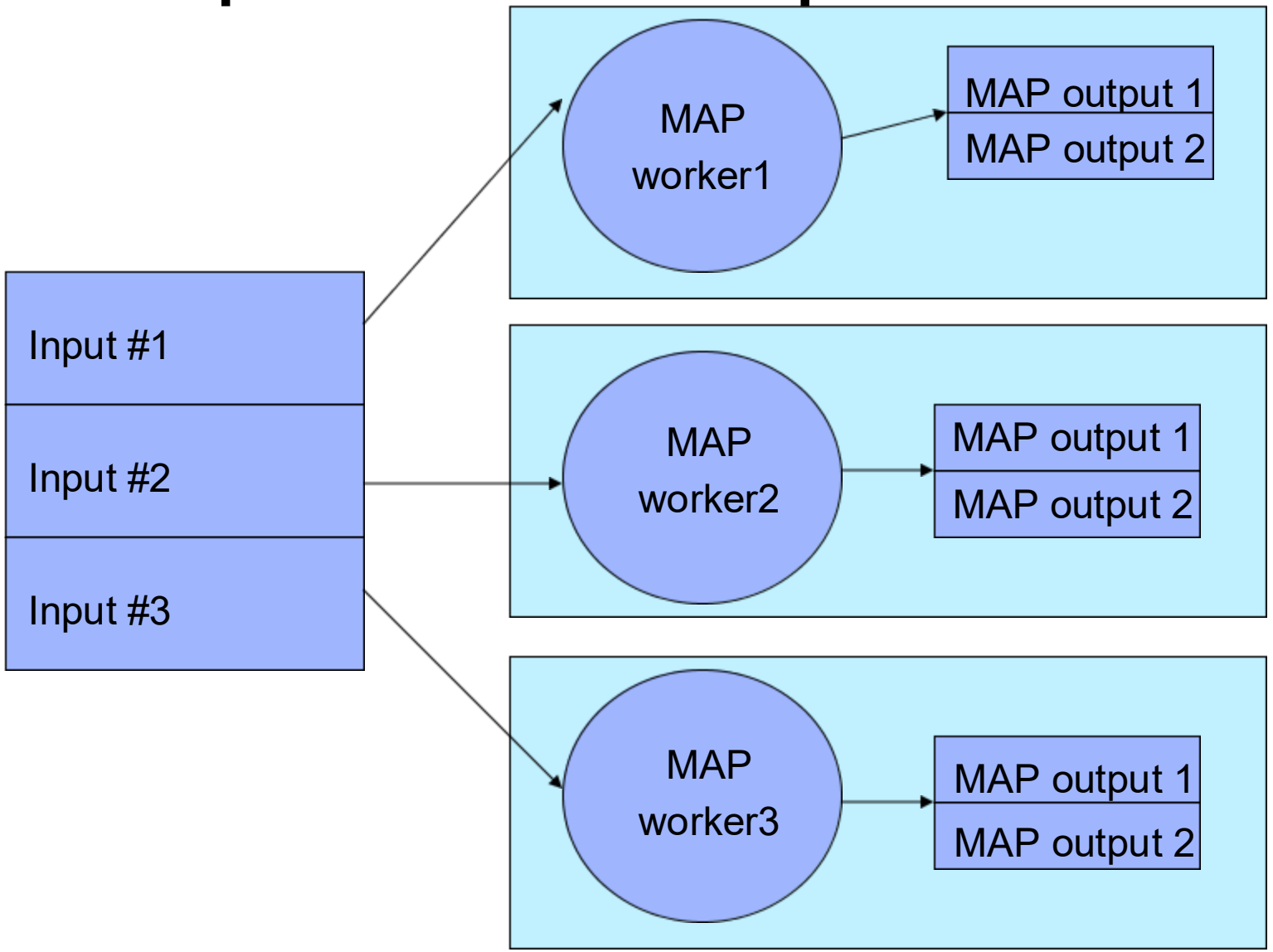
MapReduce: The “Coordinator”

- One machine is the “Coordinator” = Control Plane
- Controls other machines (workers), start or stop MAP/REDUCE
- Keeps track of progress of each worker
- Stores location of input, output of each worker
- There are MAP workers and REDUCE workers

MapReduce: Parallelism



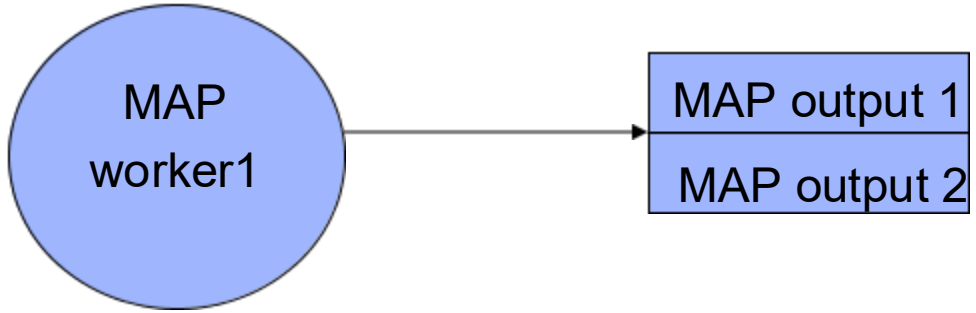
MapReduce: Map Parallelism



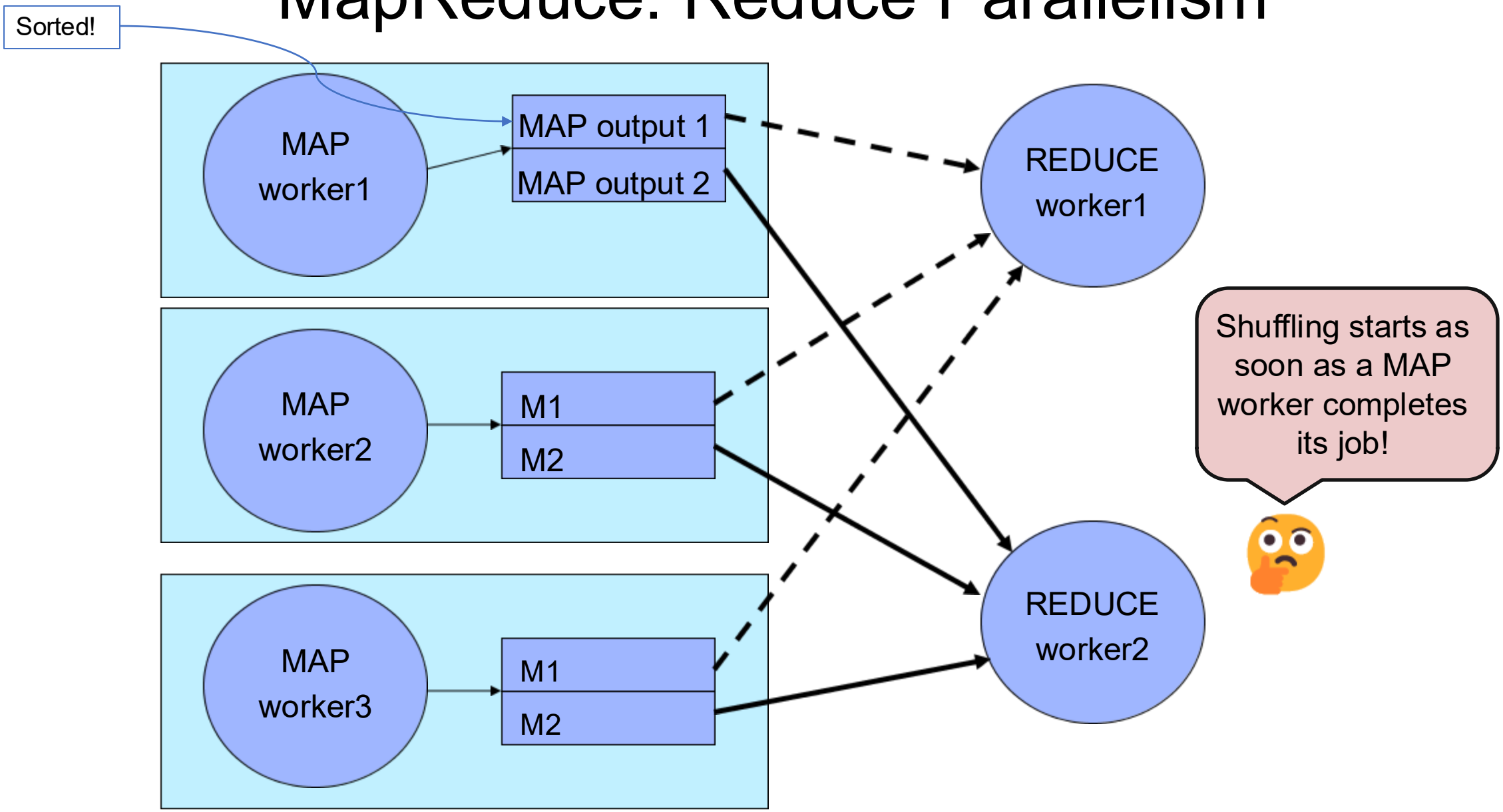
MapReduce: Map Parallelism

- MAP output is split for REDUCERs by key
- “Bins” for keys
  - E.g. keys starting (a – m) → output1 and (n – z) → output2

- (Usually use <hash(key) modulo (# of reducers)> for more random partition)

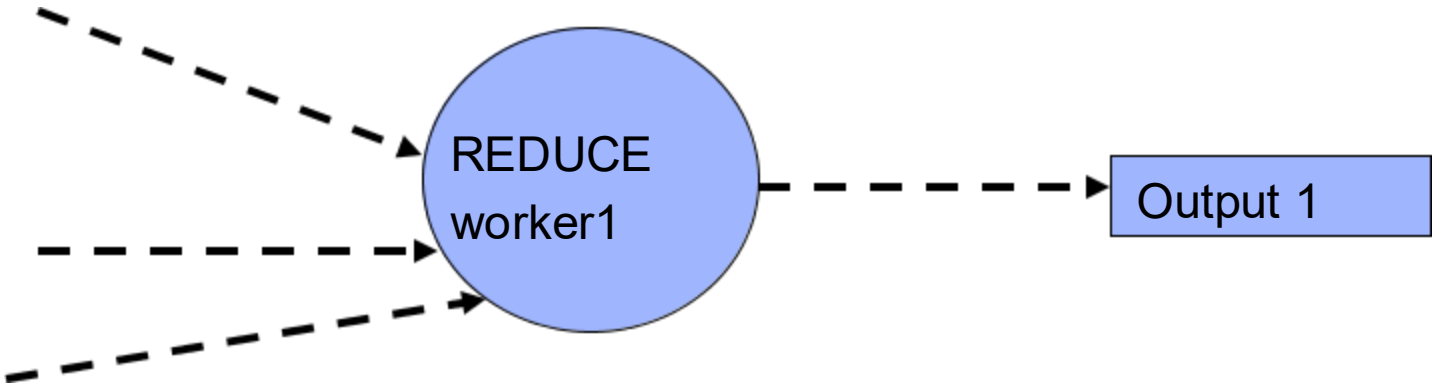


MapReduce: Reduce Parallelism



MapReduce: Reduce Parallelism

- Each REDUCE worker reads one section of input from all MAP workers
- Performs GROUPing
- Performs reducing
- Stores output



MapReduce: Fault Tolerance

- More machines = more chance of failure
- Probability of one machine failing → p (small)
- If each machine fails independently, chance of any one out of k machine failing → **$1 - (1 - p)^k$** (larger than p)
- If p = 0.01, probability of one failure:
  - 1 machine • 0.01 = 1% a machine will fail
  - 2 machines • 0.0199 = 2% a machine will fail
  - 10 machines • 0.096 = 10% a machine will fail
  - 100 machines • 0.63 = 2/3 chance a machine will fail
  - 1000 machines • **0.9999 = 99.99% chance!**

MapReduce: Fault Tolerance

"If one machine fails, the program should not fail"

"If <u>many</u> machines fail, the program should not fail"

MapReduce: Fault Tolerance

- Worker fails (happens often):
  - Coordinator checks workers
  - If worker responds as failed, or does not respond:
    - Coordinator restarts that worker on different machine

- Bad/buggy input:
  - Coordinator keeps track of bad input
  - Asks workers to skip bad input

- Coordinator fails (happens rarely):
  - Can abort operation
  - Or, can restart the coordinator on a different machine

MapReduce: Optimizations

- No REDUCE can start until all MAPs are complete (bottleneck)
  - At least the second part (after shuffling)

- Sometimes one machine is really slow (bad RAM, corrupted disk, etc)
  - Coordinator notices slow machines (called **stagglers**)
  - Executes input on another machine
  - Uses output of machine that finishes first

- Other optimizations
  - Locality optimization
  - Intermediate data compression ("combiner" pre network transfer)

Task Granularity and Pipelining

- Fine granularity tasks: many more map tasks than machines
  - e.g., 200,000 map/5000 reduce tasks w/ 2000 machines

- Minimizes time for fault recovery
- Can pipeline grouping with map execution
- Better dynamic load balancing

MapReduce: Beyond Google

- Hadoop: Open-source implementation of MapReduce
  - Hadoop v1: reimplementation of Google's MapReduce
  - Hadoop v2: separation of the Control Plane (Coordinator, resource management, etc) to YARN

- MapReduce implementations in C++, Java, Python, Perl, Ruby, Erlang ...

- Most companies that run distributed algorithms have an implementation of MapReduce (Facebook, Nokia, MySpace, ...)

Extra: Let us take a look at how to code with MR

- We will show code for Hadoop streaming running Python code for word count.
  - Hadoop was built for Java
  - Hadoop streaming is the way you write code in other languages: https://hadoop.apache.org/docs/r1.2.1/streaming.html

- Example from here:
  - https://www.geeksforgeeks.org/hadoop-streaming-using-python-word-count-problem/

mapper.py

Python3

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

# import sys because we need to read and write data to STDIN and STDOUT
import sys

# reading entire line from STDIN (standard input)
for line in sys.stdin:
    # to remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()

    # we are looping over the words array and printing the word
    # with the count of 1 to the STDOUT
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        print '%s\t%s' % (word, 1)
```

reducer.py

Python3

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

from operator import itemgetter
import sys

current_word = None
current_count = 0
word = None

# read the entire line from STDIN
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # splitting the data on the basis of tab we have provided in mapper.py
    word, count = line.split('\t', 1)
    # convert count (currently a string) to int
    try:
        count = int(count)
    except ValueError:
        # count was not a number, so silently
        # ignore/discard this line
        continue

    # this IF-switch only works because Hadoop sorts map output
    # by key (here: word) before it is passed to the reducer
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # write result to STDOUT
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word

# do not forget to output the last word if needed!
if current_word == word:
    print '%s\t%s' % (current_word, current_count)
```

Some comments

- Here is another example with optimizations using iterators and generators in Python from: https://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/

mapper.py

```
#!/usr/bin/env python
"""A more advanced Mapper, using Python iterators and generators."""

import sys

def read_input(file):
    for line in file:
        # split the line into words
        yield line.split()

def main(separator='\t'):
    # input comes from STDIN (standard input)
    data = read_input(sys.stdin)
    for words in data:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        for word in words:
            print '%s%s%d' % (word, separator, 1)

if __name__ == "__main__":
    main()
```

reducer.py

```
#!/usr/bin/env python
"""A more advanced Reducer, using Python iterators and generators."""

from itertools import groupby
from operator import itemgetter
import sys

def read_mapper_output(file, separator='\t'):
    for line in file:
        yield line.rstrip().split(separator, 1)

def main(separator='\t'):
    # input comes from STDIN (standard input)
    data = read_mapper_output(sys.stdin, separator=separator)
    # groupby groups multiple word-count pairs by word,
    # and creates an iterator that returns consecutive keys and their group:
    #   current_word - string containing a word (the key)
    #   group - iterator yielding all ["&lt;current_word&gt;", "&lt;count&gt;"] items
    for current_word, group in groupby(data, itemgetter(0)):
        try:
            total_count = sum(int(count) for current_word, count in group)
            print "%s%s%d" % (current_word, separator, total_count)
        except ValueError:
            # count was not a number, so silently discard this item
            pass

if __name__ == "__main__":
    main()
```

Java example?

WordCount.java

```
1.  package org.myorg;
2.
3.  import java.io.IOException;
4.  import java.util.*;
5.
6.  import org.apache.hadoop.fs.Path;
7.  import org.apache.hadoop.conf.*;
8.  import org.apache.hadoop.io.*;
9.  import org.apache.hadoop.mapred.*;
10. import org.apache.hadoop.util.*;
11.
12. public class WordCount {
13.
14.  public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {
15.   private final static IntWritable one = new IntWritable(1);
16.   private Text word = new Text();
17.
18.   public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
19.    String line = value.toString();
20.    StringTokenizer tokenizer = new StringTokenizer(line);
21.    while (tokenizer.hasMoreTokens()) {
22.     word.set(tokenizer.nextToken());
23.     output.collect(word, one);
24.    }
25.   }
26.  }
27.
28.  public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
29.   public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
30.    int sum = 0;
31.    while (values.hasNext()) {
32.     sum += values.next().get();
33.    }
34.    output.collect(key, new IntWritable(sum));
35.   }
36.  }
37.
38.  public static void main(String[] args) throws Exception {
39.   JobConf conf = new JobConf(WordCount.class);
40.   conf.setJobName("wordcount");
41.
42.   conf.setOutputKeyClass(Text.class);
43.   conf.setOutputValueClass(IntWritable.class);
44.
45.   conf.setMapperClass(Map.class);
46.   conf.setCombinerClass(Reduce.class);
47.   conf.setReducerClass(Reduce.class);
48.
49.   conf.setInputFormat(TextInputFormat.class);
50.   conf.setOutputFormat(TextOutputFormat.class);
51.
52.   FileInputFormat.setInputPaths(conf, new Path(args[0]));
53.   FileOutputFormat.setOutputPath(conf, new Path(args[1]));
54.
55.   JobClient.runJob(conf);
57.  }
58. }
59.
```

C++ code example

```
#include "hadoop/Pipes.hh"
#include "hadoop/TemplateFactory.hh"
#include "hadoop/StringUtils.hh"

class WordCountMap: public HadoopPipes::Mapper {
public:
  WordCountMap(HadoopPipes::TaskContext& context){}
  void map(HadoopPipes::MapContext& context) {
    std::vector<std::string> words =
      HadoopUtils::splitString(context.getInputValue(), " ");
    for(unsigned int i=0; i < words.size(); ++i) {
      context.emit(words[i], "1");
    }
  }
};

class WordCountReduce: public HadoopPipes::Reducer {
public:
  WordCountReduce(HadoopPipes::TaskContext& context){}
  void reduce(HadoopPipes::ReduceContext& context) {
    int sum = 0;
    while (context.nextValue()) {
      sum += HadoopUtils::toInt(context.getInputValue());
    }
    context.emit(context.getInputKey(), HadoopUtils::toString(sum));
  }
};

int main(int argc, char *argv[]) {
  return HadoopPipes::runTask(HadoopPipes::TemplateFactory<WordCountMap,
                              WordCountReduce>());
}
```

Can I play with it?

YES!
Please do using EMR which is available for you on the AWS lab we have. Documentations:
https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-what-is-emr.html

I like this, what should I read?

- Spark: https://www.usenix.org/legacy/event/hotcloud10/tech/full_papers/Zaharia.pdf
  - You can get an online free playground for spark here
    - Databricks: https://docs.databricks.com/en/getting-started/community-edition.html
    - EMR: https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark.html

I like this, what should I read?

- Flink (coming from KTH!): https://www.diva-portal.org/smash/get/diva2:1059537/FULLTEXT01.pdf
  - You can get an online free playground for spark here
    - Flink: https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-flink.html