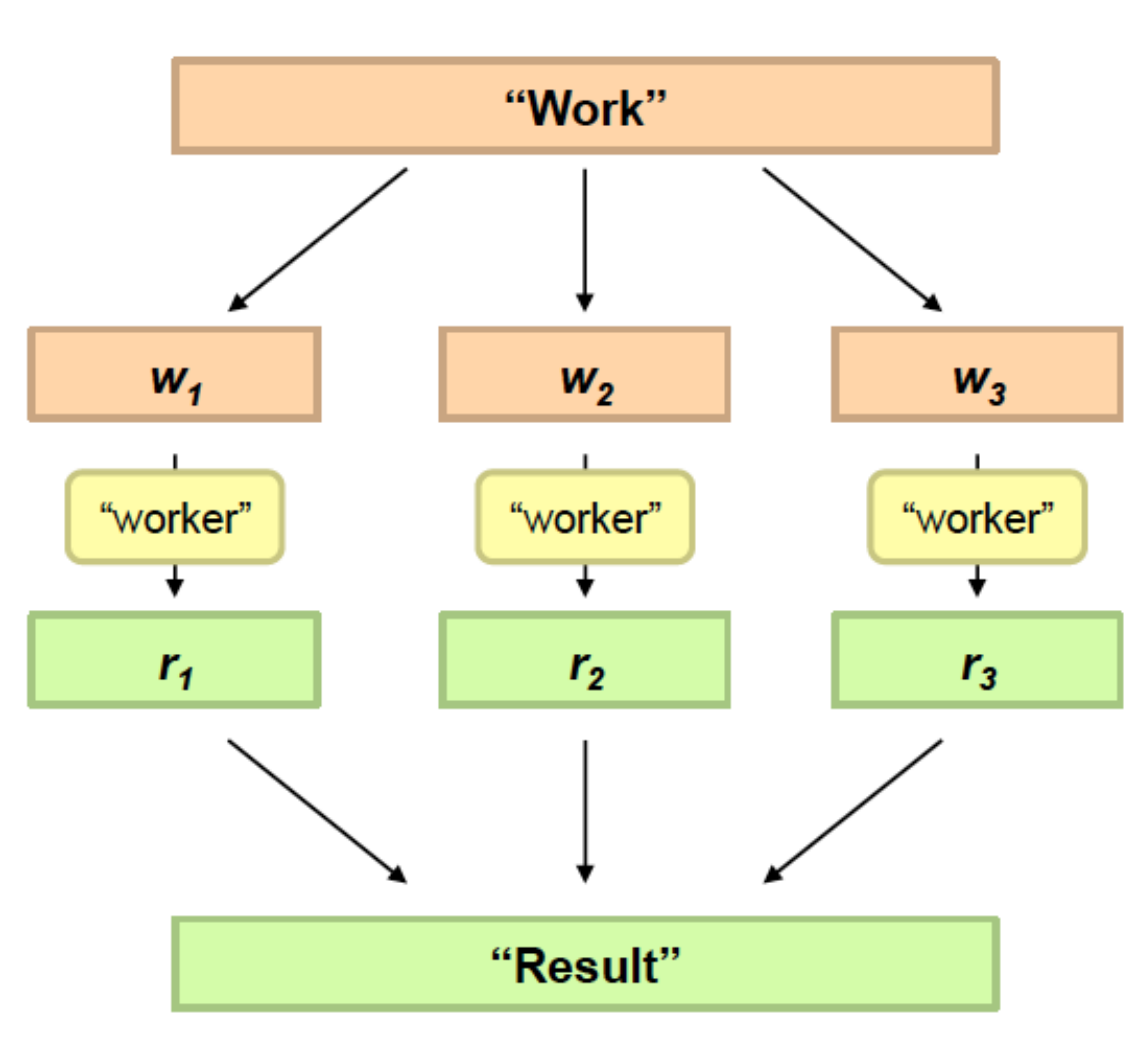
COMP9313: Big Data Management

Lecturer: Xubo Wang

Course web site: http://www.cse.unsw.edu.au/~cs9313/

Chapter 2.1: MapReduce

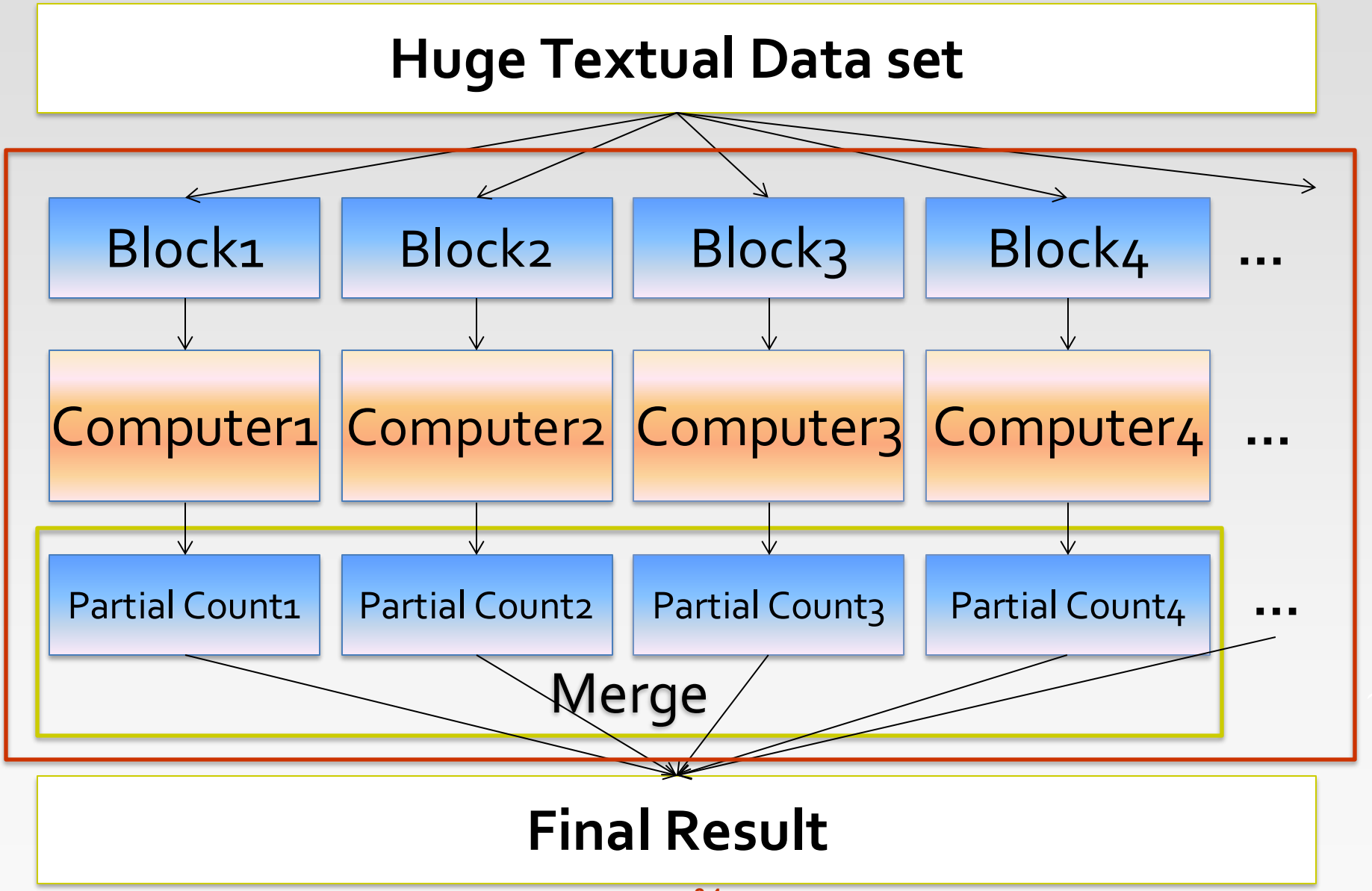
Philosophy to Scale for Big Data Processing



Divide Work

Combine Results

Distributed Word Count



What is MapReduce

- Origin from Google, [OSDI'04]
  - MapReduce: Simplified Data Processing on Large Clusters
  - Jeffrey Dean and Sanjay Ghemawat
- Programming model for parallel data processing
- Hadoop can run MapReduce programs written in various languages: e.g. Java, Ruby, Python, C++
- For large-scale data processing
  - Exploits large set of commodity computers
  - Executes process in a distributed manner
  - Offers high availability

Motivation for MapReduce

- Typical big data problem challenges:
  - How do we break up a large problem into smaller tasks that can be executed in parallel?
  - How do we assign tasks to workers distributed across a potentially large number of machines?
  - How do we ensure that the workers get the data they need?
  - How do we coordinate synchronization among the different workers?
  - How do we share partial results from one worker that is needed by another?
  - How do we accomplish all of the above in the face of software errors and hardware faults?

Motivation for MapReduce

- There was need for an abstraction that hides many system-level details from the programmer.

- MapReduce addresses this challenge by providing a simple abstraction for the developer, transparently handling most of the details behind the scenes in a ***scalable***, ***robust***, and ***efficient*** manner.

- MapReduce separates the ***what*** from the ***how***

The Idea of MapReduce

- Iterate over a large number of records
- Extract something of interest from each
- *Shuffle and sort intermediate results*
- Aggregate intermediate results
- Generate final output

Divid(Map)

Conquer(Reduce)

Key idea: provide a functional abstraction for these two operations

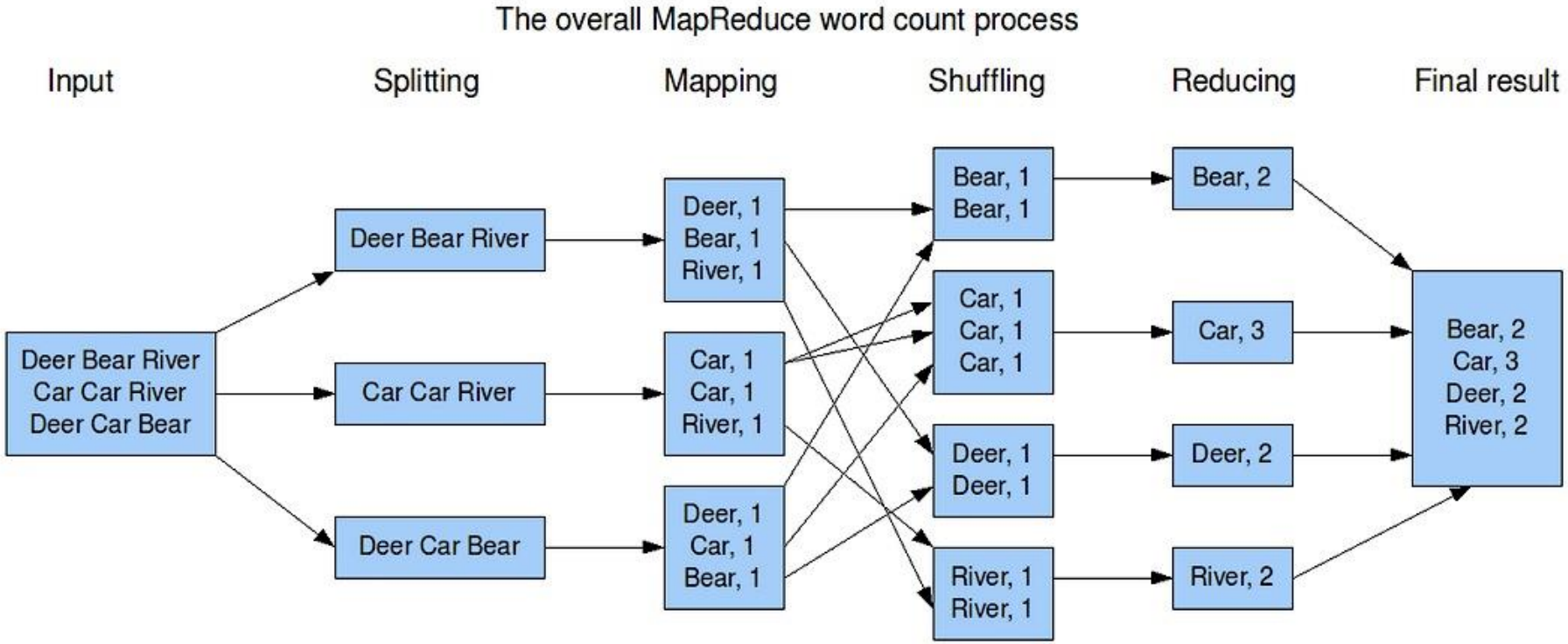
The Idea of MapReduce

- Inspired by the map and reduce functions in functional programming
- We can view map as a transformation over a dataset
  - This transformation is specified by the function *f*
  - Each functional application happens in isolation
  - The application of *f* to each element of a dataset can be parallelized in a straightforward manner
- We can view reduce as an aggregation operation
  - The aggregation is defined by the function *g*
  - Data locality: elements in the list must be "brought together"
  - If we can group elements of the list, also the reduce phase can proceed in parallel
- The framework coordinates the map and reduce phases:
  - Grouping intermediate results happens in parallel

Everything Else?

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles “data distribution”
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed file system (HDFS)
- You don’t know:
  - Where mappers and reducers run
  - When a mapper or reducer begins or finishes
  - Which input a particular mapper is processing
  - Which intermediate key a particular reducer is processing

MapReduce Example - WordCount



- Hadoop MapReduce is an implementation of MapReduce
  - MapReduce is a computing paradigm (Google)
  - Hadoop MapReduce is an open-source software

Data Structures in MapReduce

- Key-value pairs are the basic data structure in MapReduce
  - Keys and values can be: integers, float, strings, raw bytes
  - They can also be arbitrary data structures, but must be comparable (for sorting)

- The design of MapReduce algorithms involves:
  - Imposing the key-value structure on arbitrary datasets
    - E.g.: for a collection of Web pages, input keys may be URLs and values may be the HTML content
  - In some algorithms, input keys are not used (e.g., wordcount), in others they uniquely identify a record
  - Keys can be combined in complex ways to design various algorithms

Map and Reduce Functions

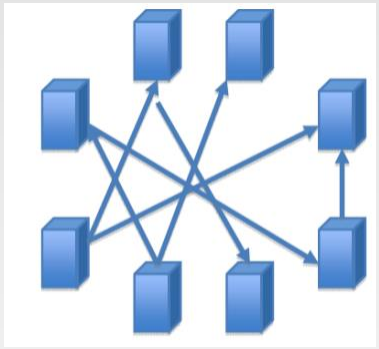
- Programmers specify two functions:
  - **map** $(k_1, v_1) \rightarrow$ list $[<k_2, v_2>]$
    - Map transforms the input into key-value pairs to process
  - **reduce** $(k_2,$ list $[v_2]) \rightarrow [<k_3, v_3>]$
    - Reduce aggregates the list of values for each key
    - All values with the same key are sent to the same reducer
  - list $[<k_2, v_2>]$ will be grouped according to key k_2 as $(k_2,$ list $[v_2])$

- The MapReduce environment takes in charge of everything else…

- A complex program can be decomposed as a succession of Map and Reduce tasks

Understanding MapReduce

- Map>>
  - (k1, v1) →
    - Info in
    - Input Split
  - list (k2, v2)
    - Key / Value out (intermediate values)
    - One list per local node
    - Can implement local Reducer (or Combiner)
- Shuffle/Sort>>



- Reduce
  - (k2, list(v2)) →
    - Shuffle / Sort phase precedes Reduce phase
    - Combines Map output into a list
  - list (k3, v3)
    - Usually aggregates intermediate values

(input) <k1, v1> → **map** → <k2, v2> → **combine** → <k2, list(V2)> → **reduce** → <k3, v3> *(output)*

WordCount

- Let's count number of each word in documents (e.g., Tweets/Blogs)
  - Reads input pair <k1,v1>
    - The input to the mapper is in format of <docID, docText>:

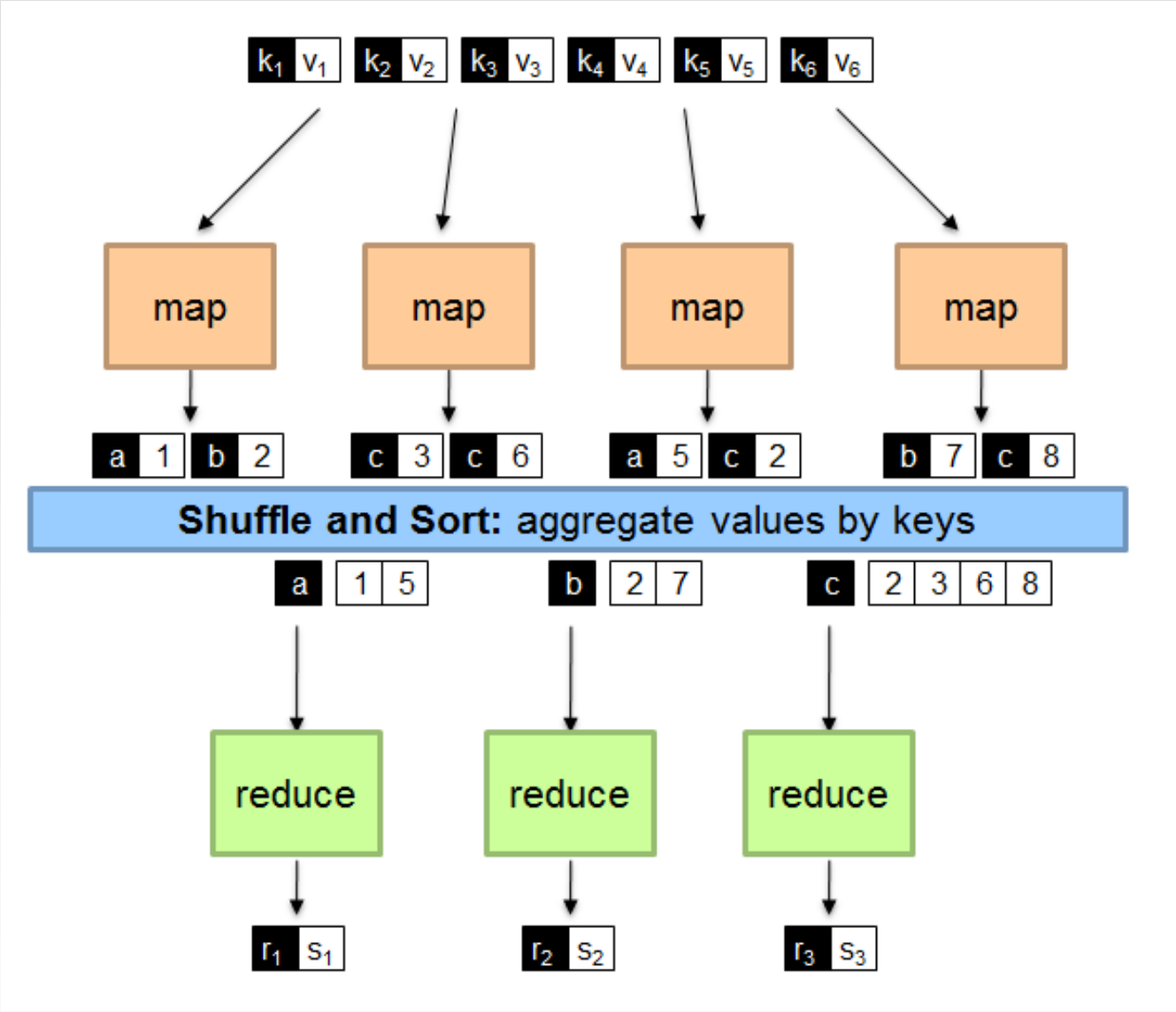
    <D1,"Hello World" >,<D2,"Hello Hadoop Bye Hadoop" >
  - Outputs pairs <k2, v2>
    - The output of the mapper is in format of <term, 1>:

  <Hello, 1><World, 1><Hello, 1><Hadoop, 1><Bye, 1><Hadoop, 1>
  - After shuffling and sort, reducer receives <k2, list(v2)>

  <Hello, {1, 1}><World, {1}><Hadoop, {1, 1}><Bye, {1}>
  - The output is in format of <k3, v3>:

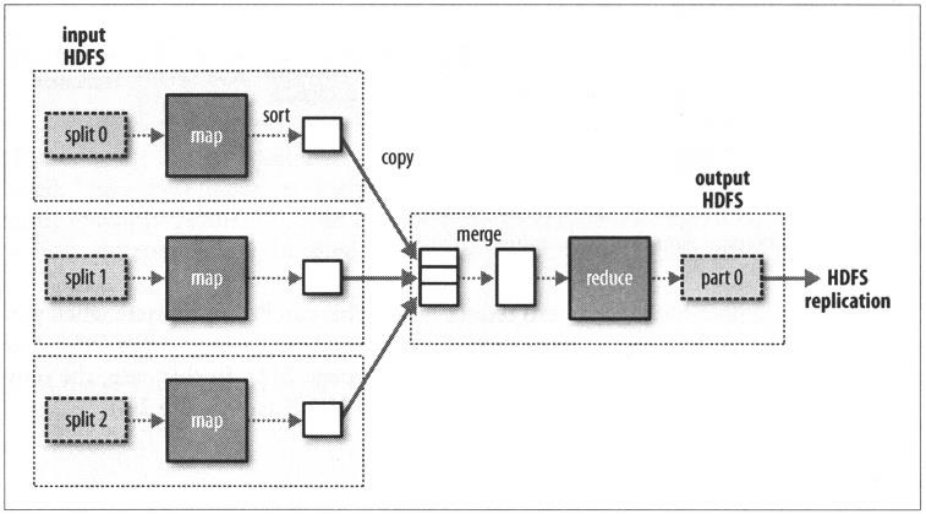
  <Hello, 2><World, 1><Hadoop, 2><Bye, 1>

A Brief View of MapReduce



Shuffle and Sort

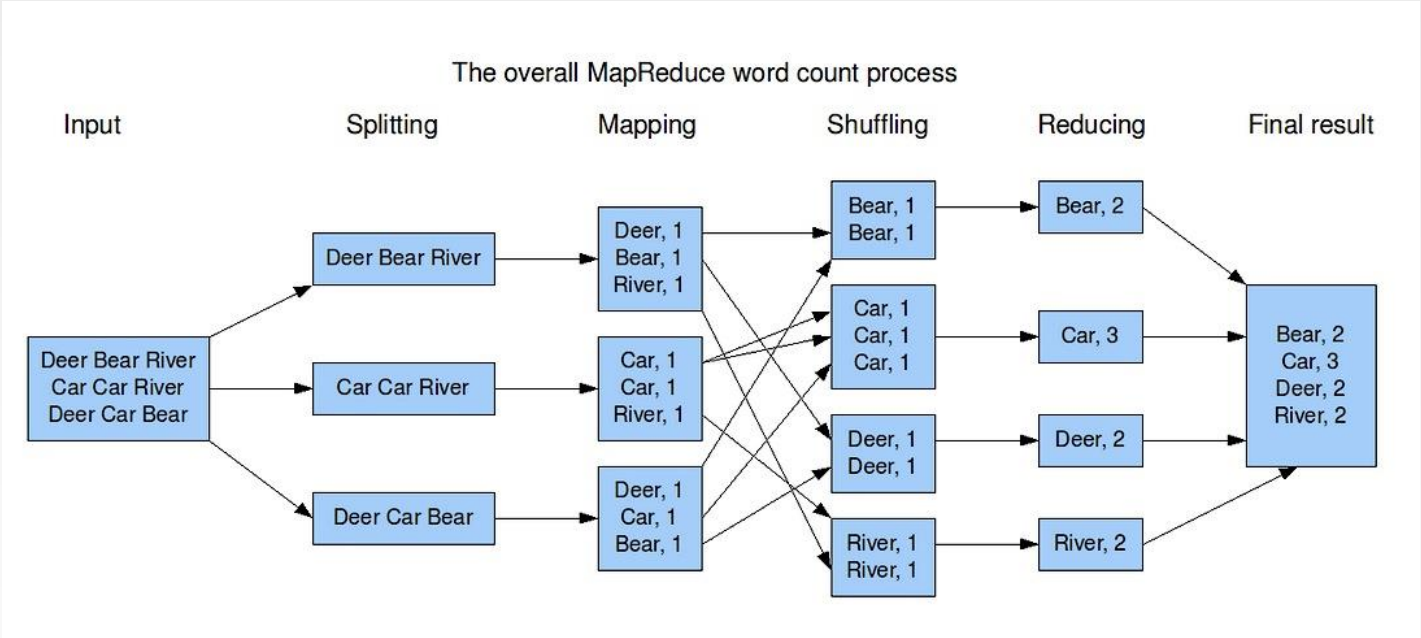
- Shuffle
  - Input to the Reducer is the sorted output of the mappers. In this phase the framework fetches the relevant partition of the output of all the mappers, via HTTP.
- Sort
  - Mapper sorts the intermediate results locally
  - The framework groups Reducer inputs by keys (since different Mappers may have output the same key) in this stage.
- Hadoop framework handles the Shuffle and Sort step .



“Hello World” in MapReduce

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ doc d do
4:             Emit(term t, count 1)

1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count s)
```



“Hello World” in MapReduce

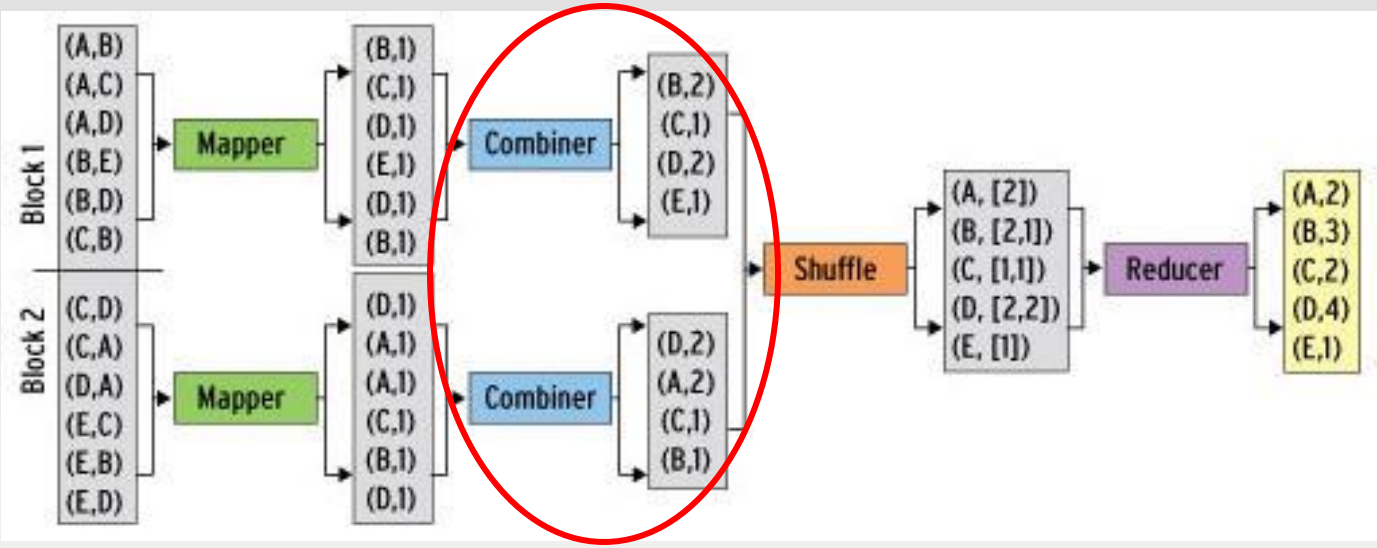
- Input:
  - Key-value pairs: (docid, doc) of a file stored on the distributed filesystem
  - docid : unique identifier of a document
  - doc: is the text of the document itself
- Mapper:
  - Takes an input key-value pair, tokenize the line
  - Emits intermediate key-value pairs: the word is the key, and the integer is the value
- The framework:
  - Guarantees all values associated with the same key (the word) are brought to the same reducer
- Reducer:
  - Receives all values associated to some keys
  - Sums the values and writes output key-value pairs: the key is the word, and the value is the number of occurrences

Combiners

- Often a Map task will produce many pairs of the form $(k,v_1), (k,v_2), \ldots$ for the same key k
  - E.g., popular words in the word count example
- Combiners are a general mechanism to reduce the amount of intermediate data, thus saving network time
  - They could be thought of as “mini-reducers”
- Warning!
  - The use of combiners must be thought carefully
    - Optional in Hadoop: the correctness of the algorithm cannot depend on computation (or even execution) of the combiners
    - A combiner operates on each map output key. It must have the same output key-value types as the Mapper class.
    - A combiner can produce summary information from a large dataset because it replaces the original Map output
  - Works only if reduce function is commutative and associative (explained later)
    - In general, reducer and combiner are not interchangeable

Combiners in WordCount

- Combiner combines the values of all keys of a single mapper node (single machine):



- Much less data needs to be copied and shuffled!
- If combiners take advantage of all opportunities for local aggregation, we have at most m × V intermediate key-value pairs
  - m: number of mappers
  - V: number of unique terms in the collection
- Note: not all mappers will see all terms

Combiners in WordCount

- You can use the reducer as the combiner in WordCount
  - This is because in this example, Reducer and Combiner do the same thing
  - Note: Most cases this is not true!
  - You need to write an extra combiner
- Given two files:
  - file1: Hello World Bye World
  - file2: Hello Hadoop Bye Hadoop
- The first map emits:
  - < Hello, 1> < World, 2> < Bye, 1>
- The second map emits:
  - < Hello, 1> < Hadoop, 2> < Bye, 1>

Partitioner

- Partitioner controls the partitioning of the keys of the intermediate map-outputs.
  - The key (or a subset of the key) is used to derive the partition, typically by a *hash function*.
  - The total number of partitions is the same as the number of reduce tasks for the job.
    - This controls which of the m reduce tasks the intermediate key (and hence the record) is sent to for reduction.
- System uses HashPartitioner by default:
  - hash(key) mod R
- Sometimes useful to override the hash function:
  - E.g., ***hash(hostname(URL)) mod R*** ensures URLs from a host end up in the same output file
    - https://www.unsw.edu.au/faculties and https://www.unsw.edu.au/about-us will be stored in one file

Write Your Own WordCount in Python?

Hadoop Streaming

- Hadoop streaming allows us to create and run Map/Reduce jobs with any executable or script as the mapper and/or the reducer. For example:

```
mapred streaming \
  -input myInputDirs \
  -output myOutputDir \
  -mapper /bin/cat \
  -reducer /usr/bin/wc
```

  - -input: specify the input folder
  - -output: specify the output folder
  - -mapper: specify the mapper script/executable
  - -reducer: specify the reducer script/executable
  - The mapper and reducer read the input from stdin (line by line) and emit the output to stdout
- Thus, you can use other languages such as C++ or Python to write MapReduce programs

Hadoop Streaming

- When an executable is specified for mappers, each mapper task will launch the executable as a separate process when the mapper is initialized.
- As the mapper task runs, it converts its inputs into lines and feed the lines to the stdin of the process.
- In the meantime, the mapper collects the line oriented outputs from the stdout of the process and converts each line into a key/value pair, which is collected as the output of the mapper.
- By default, the prefix of a line up to the first tab character is the key and the rest of the line (excluding the tab character) will be the value.
- If there is no tab character in the line, then entire line is considered as key and the value is null. However, this can be customized by setting -inputformat command option.

Hadoop Streaming

- When an executable is specified for reducers, each reducer task will launch the executable as a *separate process* when the reducer is initialized.
- As the reducer task runs, it converts its input key/values pairs into lines and feeds the lines to the stdin of the process.
- In the meantime, the reducer collects the line oriented outputs from the stdout of the process, converts each line into a key/value pair, which is collected as the output of the reducer.
- By default, the prefix of a line up to the first tab character is the key and the rest of the line (excluding the tab character) is the value. However, this can be customized by setting -outputformat command option.

Mapper

```
#!/usr/bin/env python
import sys
#--- get all lines from stdin ---
for line in sys.stdin:
        #--- remove leading and trailing whitespace---
        line = line.strip()

        #--- split the line into words ---
        words = line.split()

        #--- output tuples [word, 1] in tab-delimited format---
        for word in words:
                print ('%s\t%s' % (word, "1"))
```

- First line cannot be deleted, telling OS how to run the script

Mapper

- Let's test our mapper.py *locally* that it is working fine or not.
  - Make it executable by "chmod +x mapper.py"
  - cat inputText | python mapper.py

```
comp9313@comp9313-VirtualBox:~$ cat inputText
Hello World
Hello Hadoop Bye Hadoop
```

  - The output of the mapper is shown below

```
comp9313@comp9313-VirtualBox:~$ cat inputText | python mapper.py
Hello   1
World   1
Hello   1
Hadoop  1
Bye     1
Hadoop  1
```

  - Let's store it in a temporal file "intermediateResult": cat inputText | python mapper.py > intermediateResult

Reducer

```
#!/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()
        # slpiting 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))
```

Reducer

- Let's test our reducer.py **locally** that it is working fine or not.
  - Make it executable by "chmod +x reducer.py"
  - Run "cat intermediateResult | sort -k1,1 | python reducer.py"

```
comp9313@comp9313-VirtualBox:~$ cat intermediateResult | sort -k1,1 | python reducer.py
Bye     1
Hadoop  2
Hello   2
World   1
```

  - sort is a Linux command, used to sort a file, arranging the records in a particular order
    - **-k[n,m] Option**: sorting the records on the basis of columns n to m. Here, "sort -k1,1" means sorting the key-value pairs based on the keys (the first column)

Run On Hadoop

- Start HDFS and YARN
- Store your input files into a folder in HDFS
- Utilize the hadoop-streaming jar file to run MapReduce streaming jobs:

```
hadoop jar hadoop/share/hadoop/tools/lib/hadoop-streaming-3.3.2.jar \
        -input input \
        -output output \
        -mapper /home/comp9313/mapper.py \
        -reducer /home/comp9313/reducer.py
```

  - -input: The input folder in HDFS
  - -output: The output folder in HDFS storing the results
  - -mapper: the mapper class
  - -reducer: the reducer class
- Check your result on HDFS: hdfs dfs –cat output/part*

Run On Hadoop

- The python file do not need to pre-exist on the machines in the cluster; however, if they don't, you will need to use "-file" option to tell the framework to pack them as a part of job submission. For example:

```
hadoop jar hadoop/share/hadoop/tools/lib/hadoop-streaming-3.3.2.jar \
        -input input \
        -output output \
        -mapper /home/comp9313/mapper.py \
        -reducer /home/comp9313/reducer.py \
        -file /home/comp9313/mapper.py \
        -file /home/comp9313/reducer.py
```

  - The option "-file /home/comp9313/mapper.py" causes the python executable shipped to the cluster machines as a part of job submission.
- Using a combiner: add the "-combiner" option
  - -combiner /home/comp9313/combiner.py

Reducer (Another version)

```
#!/usr/bin/python3
import sys

results = {}
for line in sys.stdin:
    word, frequency = line.strip().split('\t', 1)
    results[word] = results.get(word, 0) + int(frequency)
words = list(results.keys())

for word in words:
    print(word, results[word])
```

- Buffer the input from stdin in memory for aggregation
- No order issue
- Memory bottleneck

MRJob

- MRJob is the easiest route to writing Python programs that run on Hadoop. If you just need to run local MapReduce jobs, you even do not need to install Hadoop.
  - You can test your code locally without installing Hadoop
  - You can run it on a cluster of your choice.
  - MRJob has extensive integration with AWS EMR and Google Dataproc. Once you're set up, it's as easy to run your job in the cloud as it is to run it on your laptop.
- MRJob has a number of features that make writing MapReduce jobs easier. In MRJob, you can:
  - Keep all MapReduce code for one job in a single class.
  - Easily upload and install code and data dependencies at runtime.
  - Switch input and output formats with a single line of code.
  - Automatically download and parse error logs for Python tracebacks.
  - Put command line filters before or after your Python code.

MRJob WordCount

- Open a file called mr_word_count.py and type this into it:

```
from mrjob.job import MRJob
import re

WORD_RE = re.compile(r"[\w']+")


class MRWordFreqCount(MRJob):

    def mapper(self, _, line):
        for word in WORD_RE.findall(line):
            yield (word.lower(), 1)

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

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


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

- Run the code locally: python mr_word_count.py inputText

How MRJob Works

- A job is defined by a class that inherits from MRJob. This class contains methods that define the steps of your job.
- A step consists of a mapper, a combiner and a reducer. All of these are optional, though you must have at least one. So you could have a step that's just a mapper, or just a combiner and a reducer.
- When you only have one step, all you have to do is write methods called mapper(), combiner() and reducer().
- The mapper() method takes a key and a value as args and yields as many key-value pairs as it likes.
- The reducer() method takes a key and an iterator of values, and also yields as many key-value pairs as it likes.
- The final required component of a job file is to include the following two lines at the end of the file, every time:

```
if __name__ == '__main__':
        MRWordCounter.run() # where MRWordCounter is your job class
```

  - These lines pass control over the command line arguments and execution to mrjob. Without them, your job will not work.

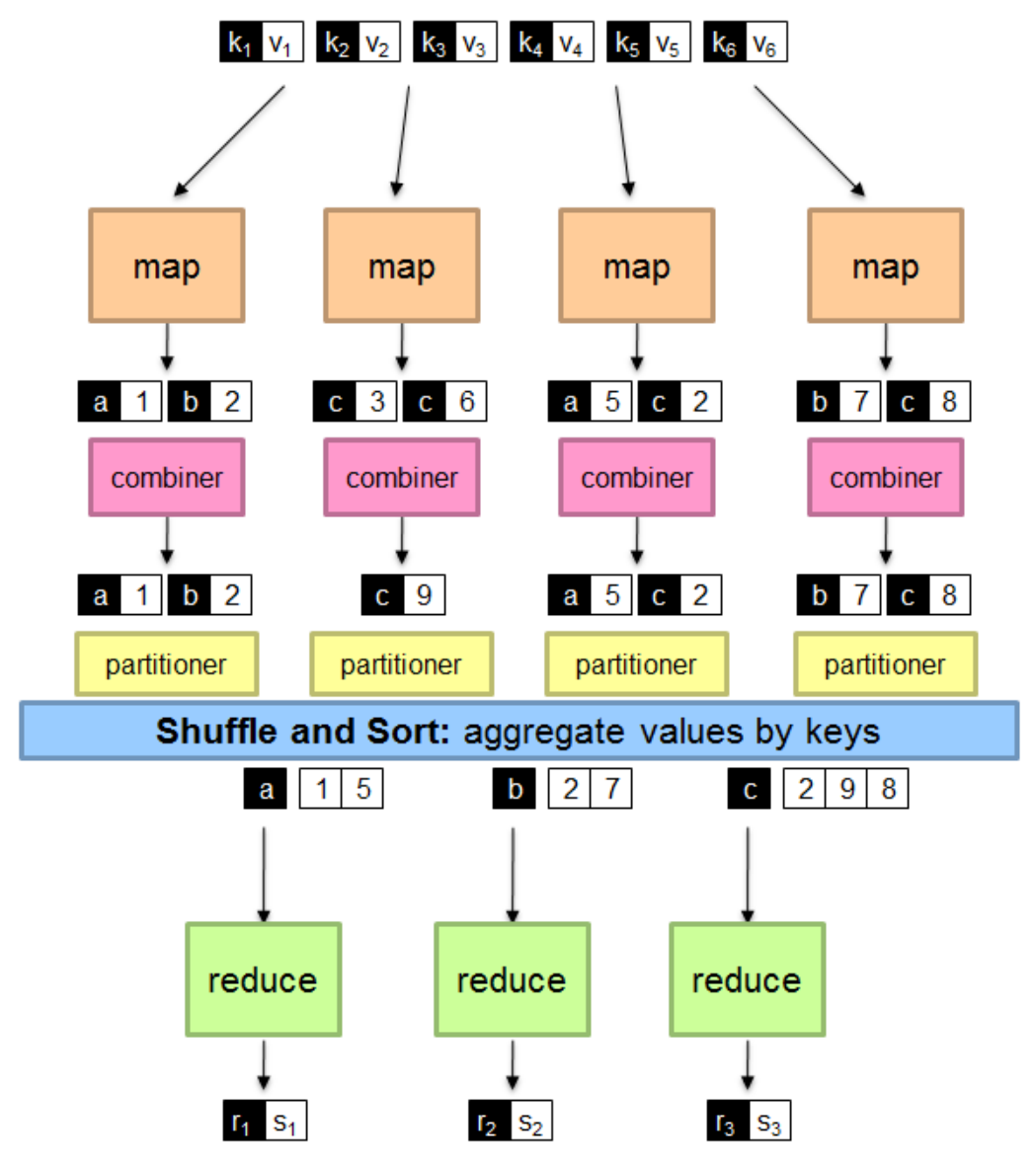
Run in Different Ways

- The most basic way to run your job is on the command line, using:
  - python my_job.py input.txt
  - By default, the output will be written to stdout.
- You can pass input via stdin, but be aware that MRJob will just dump it to a file first:
  - python my_job.py < input.txt
- By default, MRJob will run your job in a single Python process. This provides the friendliest debugging experience, but it's not exactly distributed computing!
- You change the way the job is run with the -r/--runner option. You can use -r inline (the default), -r local, -r hadoop or -r emr.
  - To run your job in multiple subprocesses with a few Hadoop features simulated, use -r local
  - To run it on your Hadoop cluster, use -r hadoop
    - python my_job.py –r hadoop hdfs:///my_home/my_file
  - If you have EMR/Dataproc configured, you can run it there with -r emr/dataproc.

MapReduce: Recap

- Programmers must specify:
  - map $(k_1, v_1) \rightarrow [(k_2, v_2)]$
  - reduce $(k_2, [v_2]) \rightarrow [<k_3, v_3>]$
  - All values with the same key are reduced together
- Optionally, also:
  - combine $(k_2, [v_2]) \rightarrow [<k_3, v_3>]$
    - Mini-reducers that run in memory after the map phase
    - Used as an optimization to reduce network traffic
  - partition $(k_2$, number of partitions$) \rightarrow$ partition for k_2
    - Often a simple hash of the key, e.g., hash(k_2) mod n
    - Divides up key space for parallel reduce operations
- The execution framework handles everything else…

MapReduce Data Flow: Recap



References

- MapReduce Chapter of <<Hadoop The Definitive Guide>>
- Hadoop Streaming. https://hadoop.apache.org/docs/current/hadoop-streaming/HadoopStreaming.html
- MRJob. https://mrjob.readthedocs.io/en/latest/index.html

End of Chapter 2.1