15-388/688 - Practical Data Science: Big data and MapReduce

J. Zico Kolter
Carnegie Mellon University
Spring 2018

Outline

Big data

Some context in distributed computing

map + reduce

MapReduce

MapReduce in Python (very briefly)

Outline

Big data

Some context in distributed computing

map + reduce

MapReduce

MapReduce in Python (very briefly)

“Big data”

My laptop
8GB RAM
500GB Disk

Big data?
No

?

Google Data Center
??? RAM/Disk
(>> PBs)

Big data?
Yes

Some notable inflection points

1. Your data fits in RAM on a single machine

2. Your data fits on disk on a single machine

3. Your data fits in RAM/disk on a “small” cluster of machines (you don’t need to worry about machines dying)

4. Your data fits in RAM/disk on a “large” cluster of machine (you need to worry about machines dying)

It’s probably reasonable to refer to 3+ as “big data”, but many would only consider 4

Do you have big data?

If your data fits on a single machine (even on disk), then it's almost always better to think about how you can design an efficient single-machine solution, unless you have extremely good reasons for doing otherwise

scalable system	cores	twitter	uk-2007-05
GraphChi [10]	2	3160s	6972s
Stratosphere [6]	16	2250s	-
X-Stream [17]	16	1488s	-
Spark [8]	128	857s	1759s
Giraph [8]	128	596s	1235s
GraphLab [8]	128	249s	833s
GraphX [8]	128	419s	462s
Single thread (SSD)	1	300s	651s
Single thread (RAM)	1	275s	-

Table 2: Reported elapsed times for 20 PageRank iterations, compared with measured times for single-threaded implementations from SSD and from RAM. GraphChi and X-Stream report times for 5 PageRank iterations, which we multiplied by four.

scalable system	cores	twitter	uk-2007-05
Stratosphere [6]	16	950s	-
X-Stream [17]	16	1159s	-
Spark [8]	128	1784s	$\geq$ 8000s
Giraph [8]	128	200s	$\geq$ 8000s
GraphLab [8]	128	242s	714s
GraphX [8]	128	251s	800s
Single thread (SSD)	1	153s	417s

Table 3: Reported elapsed times for label propagation, compared with measured times for single-threaded label propagation from SSD.

Tables from [McSherry et al., 2015 "Scalability! But at what COST"]

Outline

Big data

Some context in distributed computing

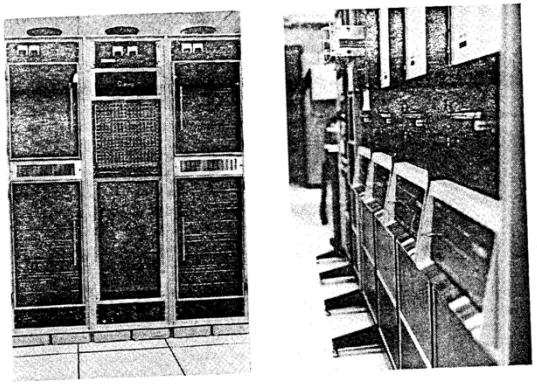
map + reduce

MapReduce

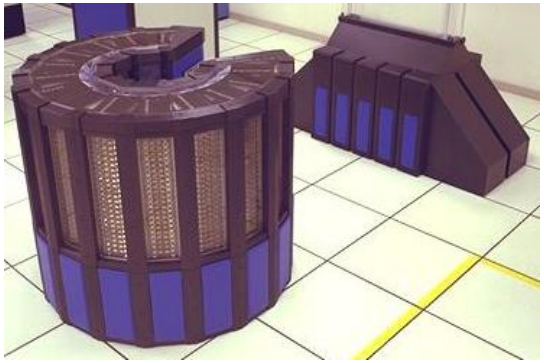
MapReduce in Python (very briefly)

Distributed computing

Distributed computing rose to prominence in the 70s/80s, often built around “supercomputing,” for scientific computing applications



1971 – CMU C.mmp
(16 PDP-11 processors)



1984 – Cray-2
(4 vector processors)

Message passing interface

In mid-90s, researchers built a common interface for distributed computing called the message passing interface (MPI)

MPI provided a set of tools to run multiple processes (on a single machine or across many machines), that could communicate, send data between each other (all of “scattering”, “gathering”, “broadcasting”), and synchronize execution

Still common in scientific computing applications and HPC (high performance computing)

Downsides to MPI

MPI is extremely powerful but has some notable limitations

1. MPI is complicated: programs need to explicitly manage data, synchronize threads, etc

2. MPI is brittle: if machines die suddenly, can be difficult to recover (unless explicitly handled by the program, making them more complicated)

A new paradigm for data processing

When Google was building their first data centers, they used clusters of off-the-shelf commodity hardware; machines had different speeds and failures were common given cluster sizes

Data itself was distributed (redundantly) over many machines, as much as possible wanted to do the computation on the machine where the data is stored

Led to the development of the MapReduce framework at Google [Dean and Ghemawat, 2004], later made extremely popular through the Apache Hadoop open source implementation

MapReduce

A simple paradigm for distributed computation where users write just two functions: a *mapper* and a *reducer*

Work can be automatically farmed out to a large collection of machines

As much as possible, computation is done on the machine where the data lives

Node failures or “stragglers” (nodes that are slow for some reason) are automatically handled

Big data since MapReduce

MapReduce is a wonderful, but many disadvantages (discussed shortly)

Since ~2010s, big data community has been “slowly” trying to re-integrate some of the ideas from the HPC community

Aside: GPUs are really the natural descendants of the HPC line of work, which are doing pretty well in data science these days…

Remember:

$$\text{Speed(network)} < \text{Speed(disk)} < \text{Speed(RAM)} < \text{Speed(Cache)}$$

(use the fastest data storage mechanism possible)

Outline

Big data

Some context in distributed computing

map + reduce

MapReduce

MapReduce in Python (very briefly)

Primer: map and reduce functions

We can get some intuition on MapReduce by inspiration from the map and reduce functions in Python (but MapReduce ≠ map + reduce)

The map call takes a function and a list (iterable) and generates a new list of the function applied to each element:

```
map(f, [a, b, c, ...]) -> [f(a), f(b), f(c), ...]
```

The reduce call takes a function and a list (iterable) and iteratively applies the function to two elements (next item in the list and result of previous function)

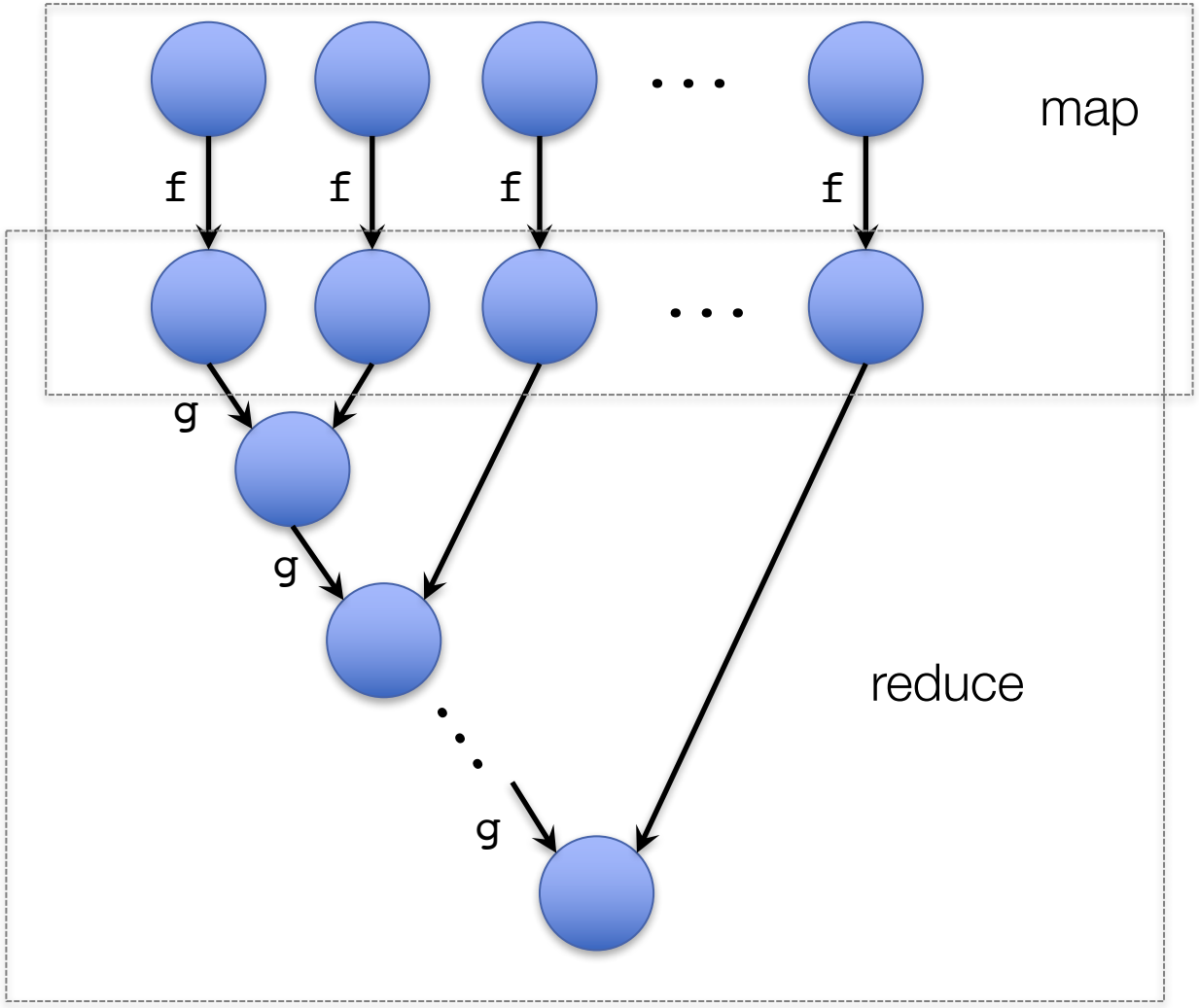
```
reduce(g, [a, b, c, ...]) -> g(g(g(a,b),c), ...)
```

Example: Sum of squared elements

We could take a list, square each element, and add these squared terms together using the following code

```
data = [1,2,3,4]
values = map(lambda x : x*x, data)
# values = [1, 4, 9, 16]
output = reduce(lambda x,y: x+y, values)
# output = 30
```

Map and reduce graphically



Mappers, reducers, and execution engines

We'll specifically refer to the ***mapper*** function, the ***reducer*** function and the ***execution engine*** (the supporting code that actually calls the map and reduce functions)

```
def map_reduce_execute(data, mapper, reducer):
    values = map(mapper, data)
    output = reduce(reducer, values)
    return output

def mapper_square(x):
    return x**2

def reducer_sum(x,y):
    return x+y

map_reduce_execute([1,2,3,4], mapper_square, reducer_sum)
```

Abstracting map + reduce

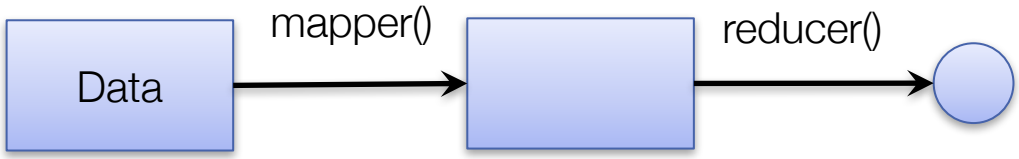
Key point: to use this framework, the programmer only needs to implement the mapper and reducer function, and the execution engine can use whatever method it wants to actually compute the result

For instance, the application of the mapper functions is inherently parallel, can be carried out in separate threads/machines

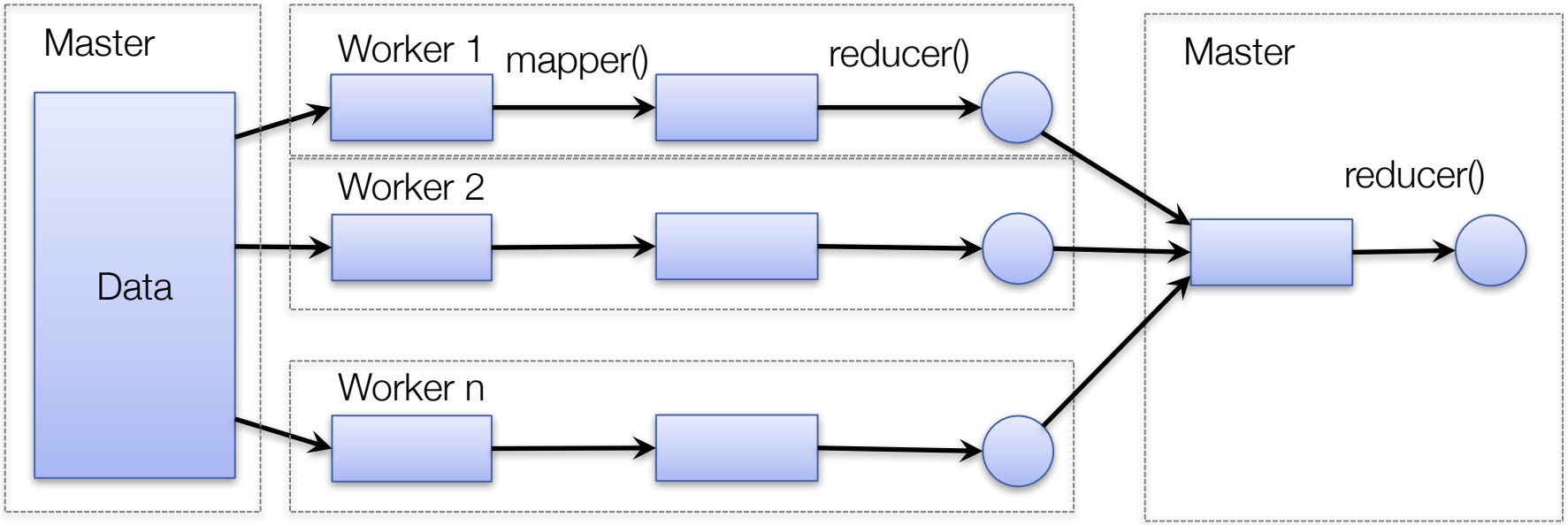
In many cases, the reduce step can also be carried out incrementally

Distributed map + reduce

Single machine execution engine:



Distributed execution engine:



Poll: properties for distributed map + reduce

Suppose we run a distributed map + reduce execution engine as in the previous slide, which distributes data randomly to each worker before applying the mapper, reducer, collecting the data, and applying the reducer again.

Which properties of the mapper f and reducer g are required to ensure the results are deterministic?

1. g is *commutative* $g(a, b) = g(b, a)$

2. g is associative $g(g(a, b), c) = g\big(a, g(b, c)\big)$

3. f and g are distributive $f\big(g(a, b)\big) = g(f(a), f(b))$

Outline

Big data

Some context in distributed computing

map + reduce

MapReduce

MapReduce in Python (very briefly)

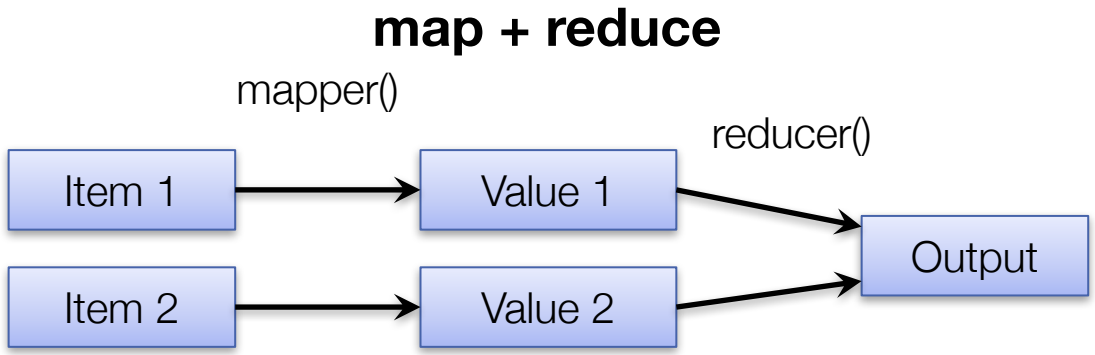
MapReduce (≠ map + reduce)

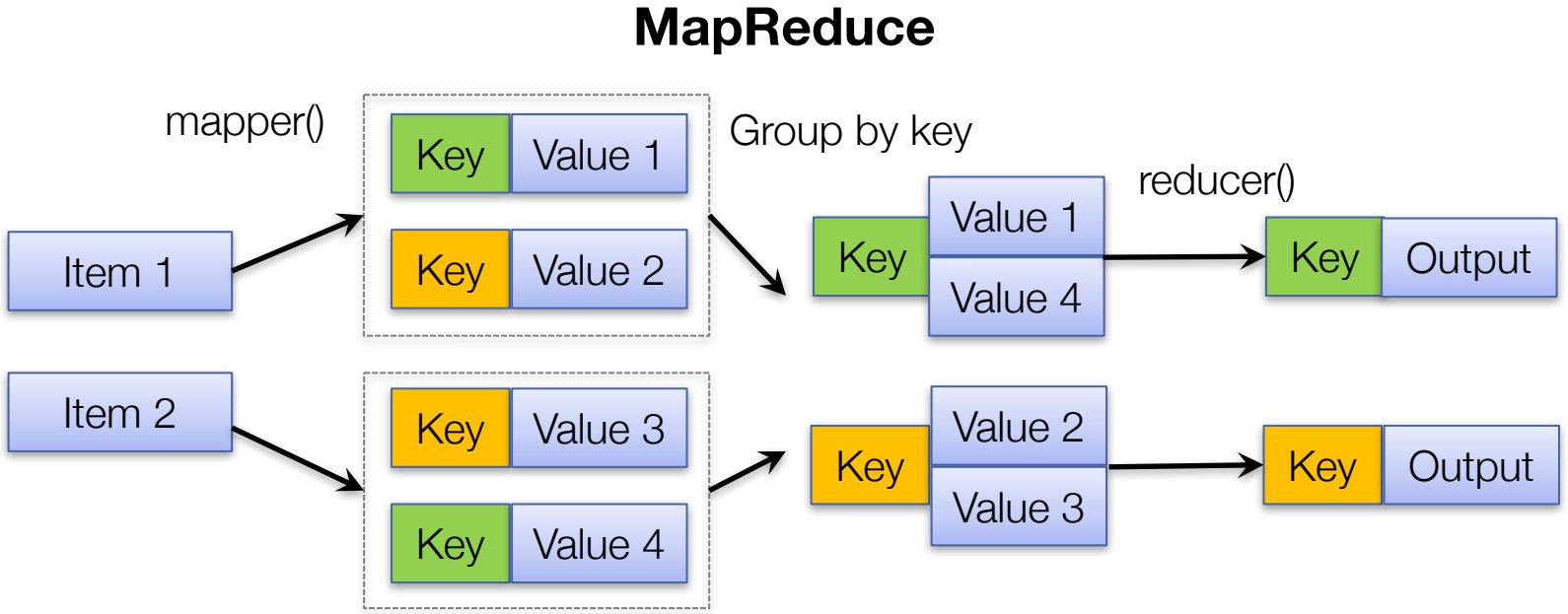
You can think of MapReduce as map + reduce **“by key”**

Mapper function doesn’t just return a single value, but a *list* of key-value pairs (with potentially multiple instances of the same key)

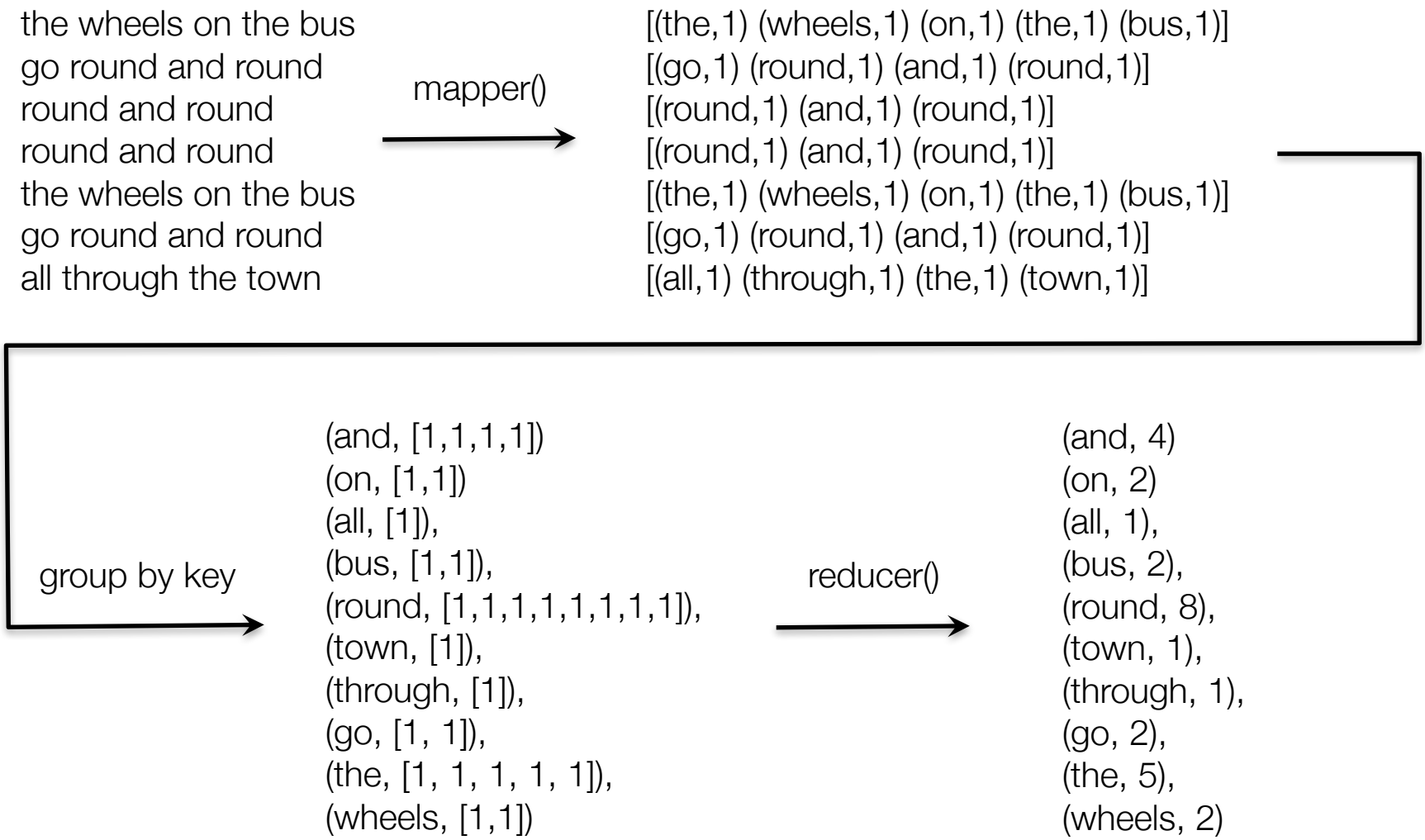
Before calling the reducer, the execution engine groups all results by key

map + reduce vs. MapReduce





Example: word count



MapReduce execution engine

A simple MapReduce execution engine (no parallelism, so not particularly useful), can be written as follows

```
def mapreduce_execute(data, mapper, reducer):
    values = map(mapper, data)

    groups = {}
    for items in values:
        for k,v in items:
            if k not in groups:
                groups[k] = [v]
            else:
                groups[k].append(v)
    output = [reducer(k,v) for k,v in groups.items()]
    return output
```

MapReduce word occurrence count example

In this engine, we can run our word occurrence counter by specifying the following mapper and reducer

```
def mapper_word_occurrence(line):
    return [(word, 1) for word in line.split(" ")]

def reducer_sum(key, val):
    return (key, sum(val))

lines = ["the wheels on the bus",
         "go round and round",
         "round and round",
         "round and round",
         "the wheels on the bus",
         "go round and round",
         "all through the town"]

mapreduce_execute(lines, mapper_word_occurrence, reducer_sum)
```

More advanced usage

In original paper, and most implementations, input data is *also* in key/value form, so the mapper also is provided with a key value pair

Many real applications require chaining together multiple map/reduce steps

“Combiners” are local reducers that run after each map to potentially reduce network overhead

Optional ability for functions to all share some additional context (i.e., shared read-only memory between multiple mappers / reducers)

Advantages of MapReduce

MapReduce isn't popular because of what it can do, it's popular because of what it can't do (i.e., what you don't need to do)

End user just needs to implement two functions: mapper and reducer

No exposure of interprocess communication, data splitting, data locality, redundancy mechanisms (can all be handled by underlying system)

Disadvantages of MapReduce

Can be extremely slow: in traditional MapReduce, resilience is attained by reading/writing data from/to disk between each stage of processing

Sometimes you really do want communication between processes

Distributed data systems beyond MapReduce: Spark, GraphLab, parameter servers, many others

All of them will frequently be slower than a single machine, if your data fits on the disk of a single machine

Outline

Big data

Some context in distributed computing

map + reduce

MapReduce

MapReduce in Python (very briefly)

Practical MapReduce

(Obviously) you don't want to write you own MapReduce execution engine, use one of the many engines available

Python **`mrjob`** library: write simple mappers/reducers in Python, and execute on Hadoop systems, Amazon Elastic MapReduce, Google Cloud

Word occurrence count example:

```
from mrjob.job import MRJob

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

    def reducer(self, key, values):
        yield key, sum(values)
```