UNIVERSITY OF WATERLOO

Data-Intensive Distributed Computing

431/451/631/651 (Fall 2021)

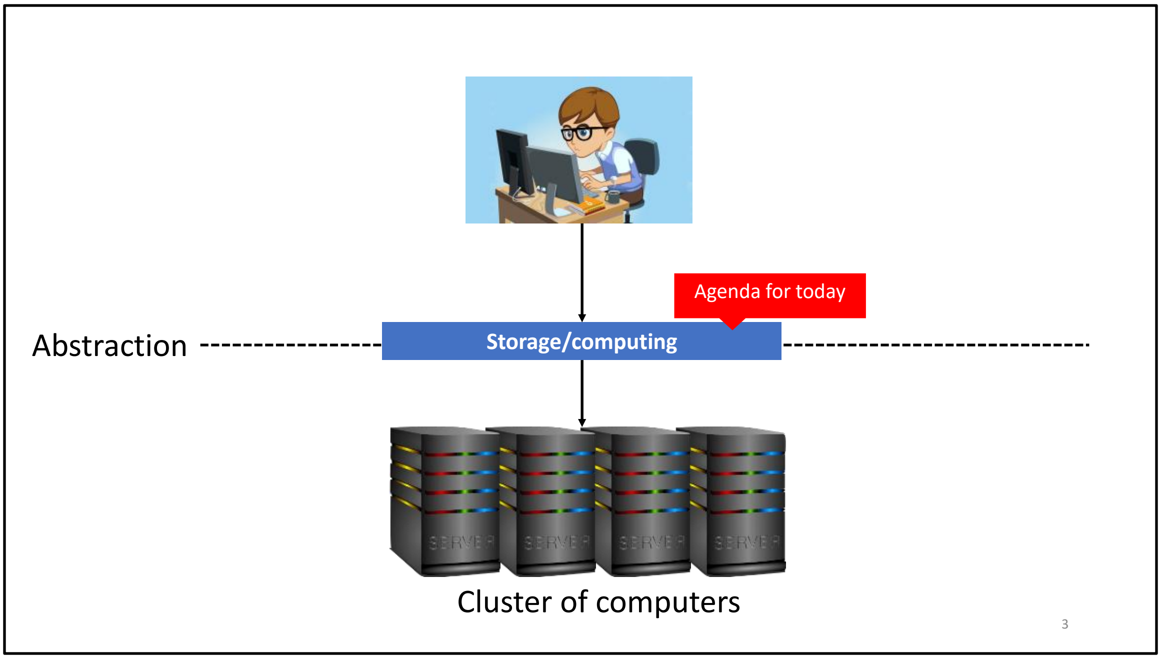
Part 1: MapReduce Algorithm Design (1/3)

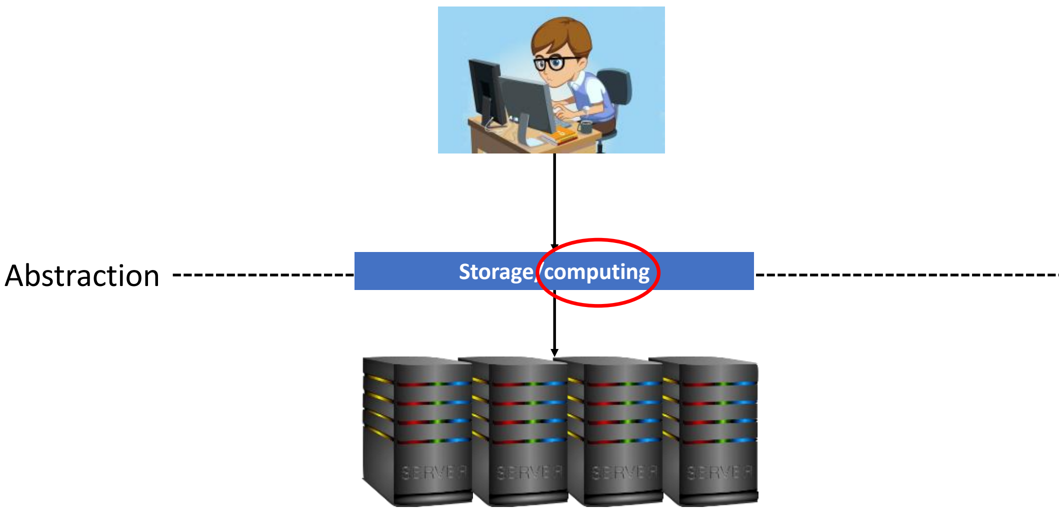
Ali Abedi

These slides are available at https://www.student.cs.uwaterloo.ca/~cs451/


BIG DATA IS GOING TO IMPACT US SOON!
CALL BRUCE WILLIS!!
DATA
www.nuxeo.com
Soulcié

2


Agenda for today
Abstraction
Storage/computing
Cluster of computers
3



Cluster of computers

Data-intensive distributed computing

How can we process a large file on a distributed system?

MapReduce

File.txt

10 TB

How many times do we see "Waterloo" in this file?

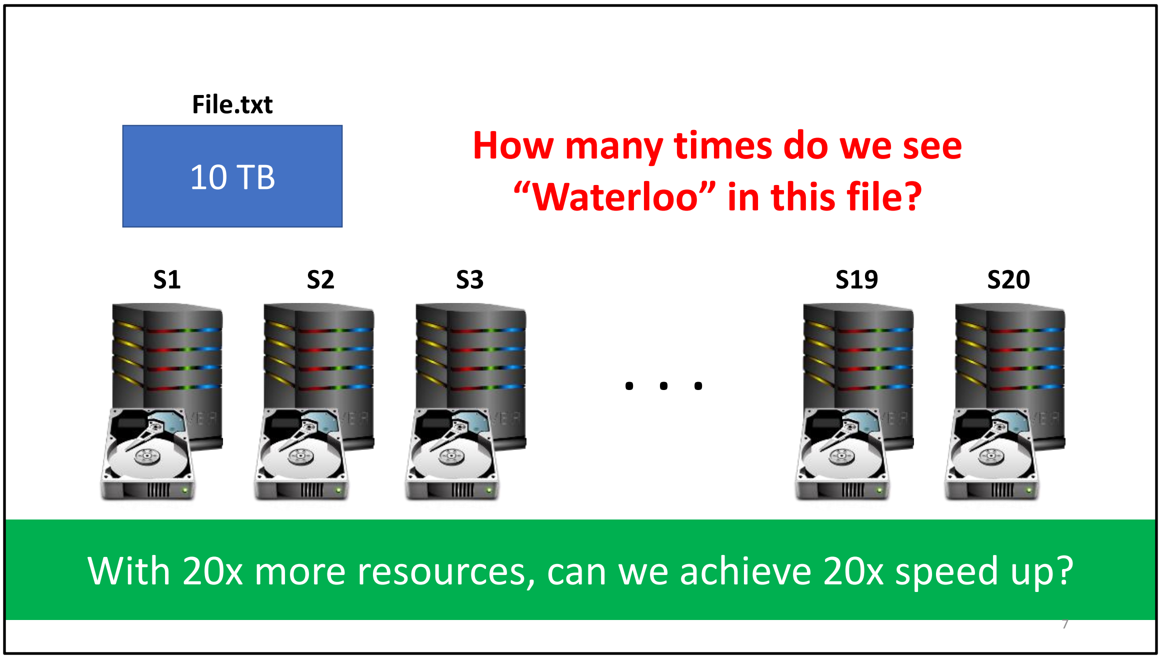
Sequential read: 100 MB/s

$$\frac{10\,TB}{100\,MB/s} = 28\,hours$$

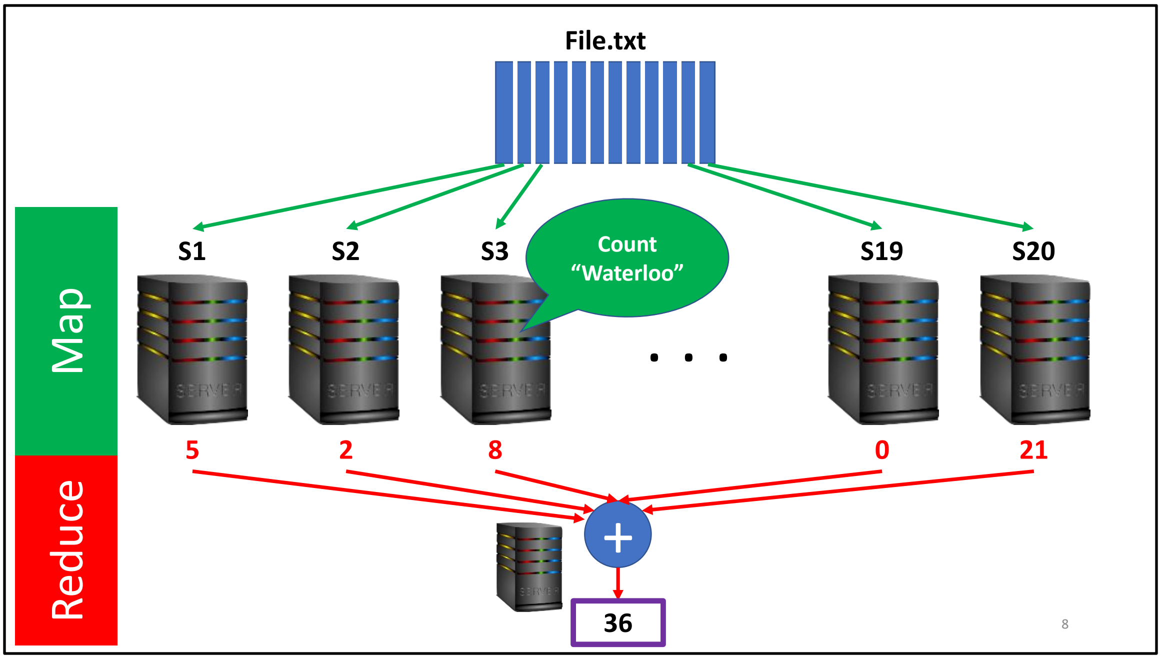
It takes 28 hours just to read the file (ignoring computation)

6

Just a single machine



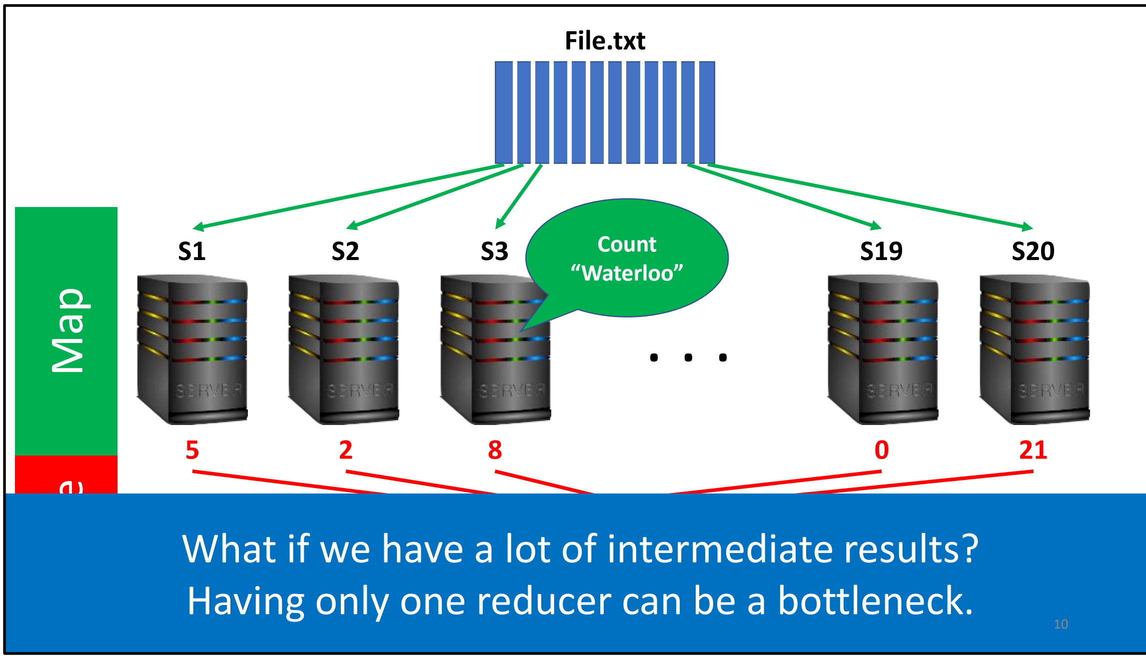
Can we speed up this process by using more resources?
How can we solve this problem using 20 servers instead?
For simplicity assume that all 20 servers have a copy of the 10 TB file.



This is the logical view of how MapReduce works in our simple count Waterloo example.
Each of the 20 servers are responsible for a chunk of the 10TB file. Each server counts the number of times Waterloo appears in the text assigned to it.
Then, all servers send these partial results to another server (can be one of the 20 servers). This server adds up all of the partial results to find the total number of times Waterloo appears in the 10TB file.
Physical view details such as how each server gets the chunk it should process, and how intermediate results are moved to the reducer should be ignored for now.

MapReduce is essentially distributed divide and conquer …

9



In our simple example, one reducer was enough because it only had to add up some (i.e., number of mappers) numbers.
But in general we might have a ton of partial results from the map phase. Let’s see another example.

File.txt

10 TB

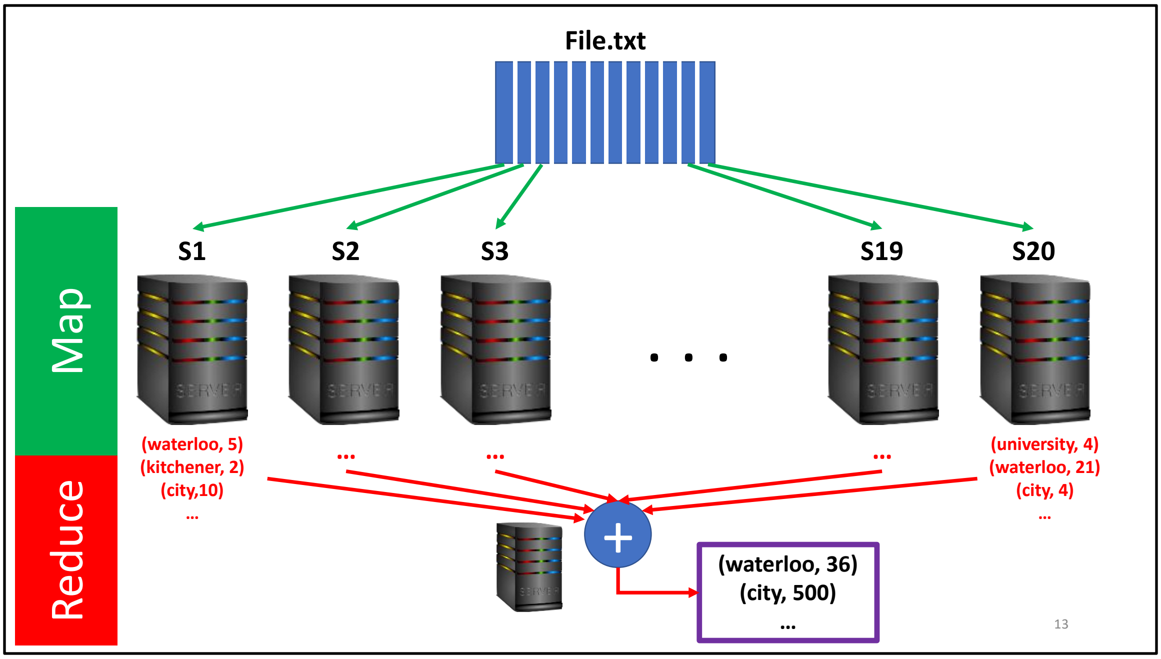
How many times do we see each word in this file?

Word count is the "hello world" of MapReduce

The expected output is …

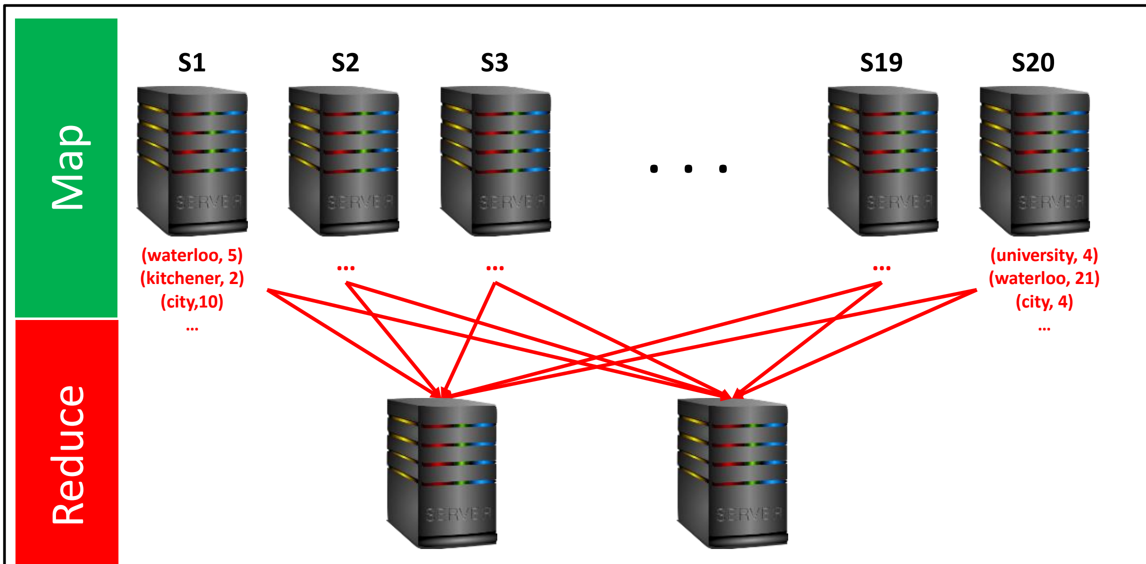
For each word in the input file, count how many times it appears in the file.

Word	Count
Waterloo	36
Kitchener	27
City	512
Is	12450
The	16700
University	123
…	



All mappers send list of (key, value) pairs to the reducer, where the key is word and value is its count.
The reducer adds up all intermediate results. But it can now be a bottleneck.

Can we have multiple reducers like mappers?



What intermediate result should be moved to which reducer?

Sending partial results to the right reducer

- Each word should be processed by one reducer, otherwise we will have partial results again!
  - E.g., all (Waterloo, *) should be processed by the same reducer

- So we partition intermediate results by key

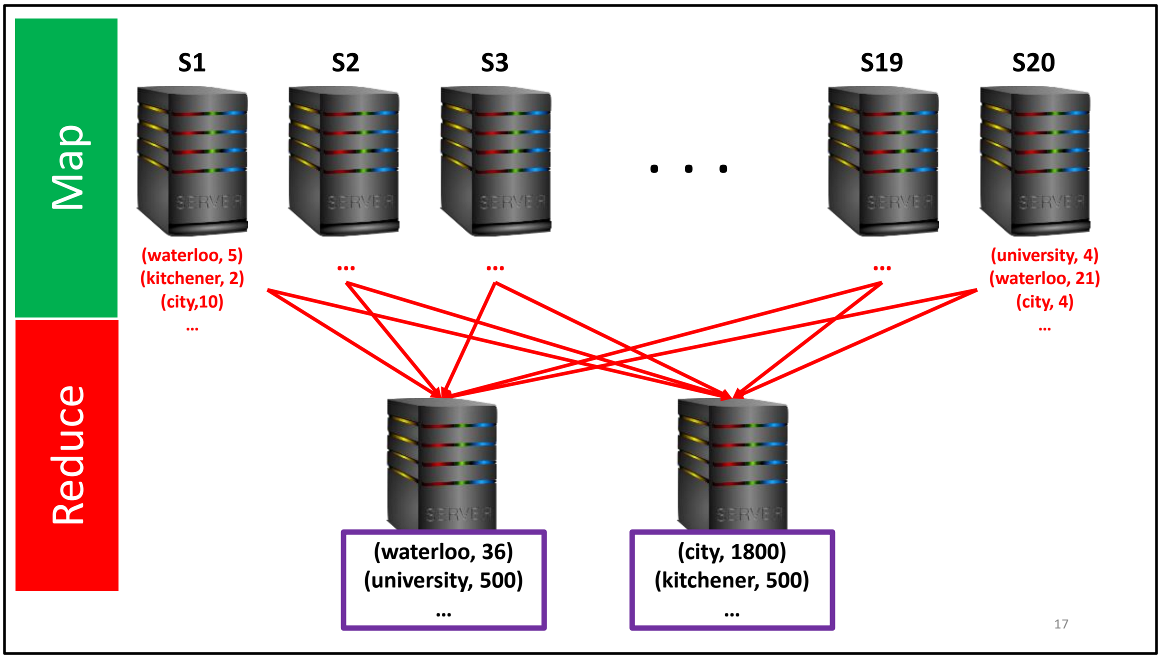
How can mapper x know which reducer mapper y will sent key k?

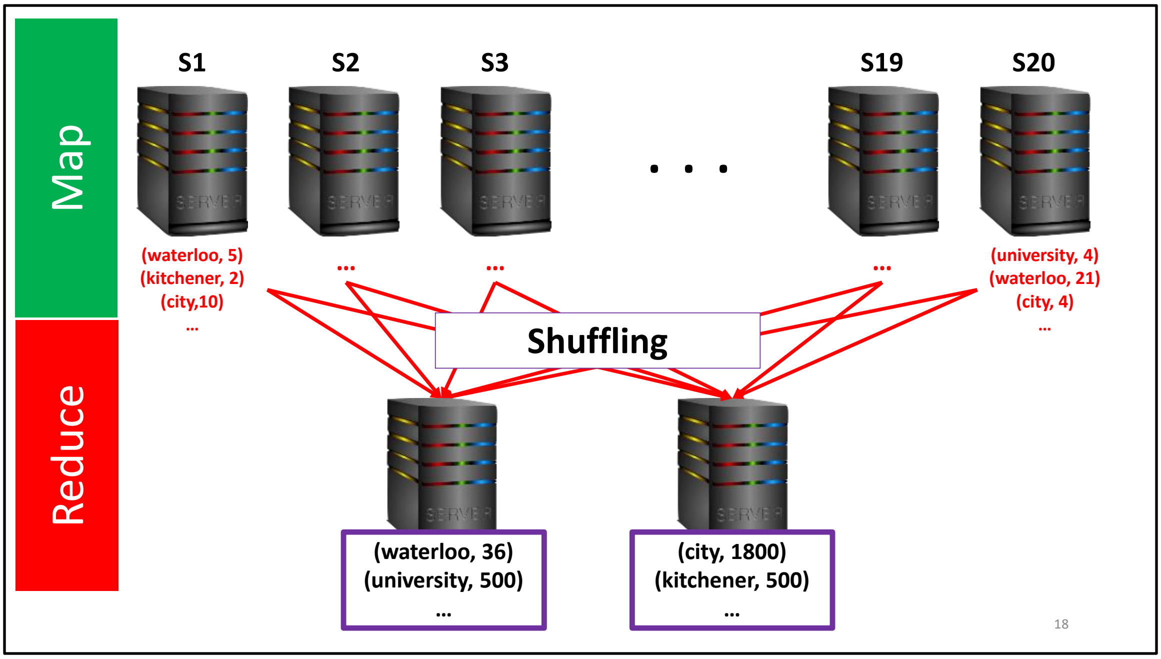
Hash functions to rescue …

- Mapper x and y can send key k to the same reducer by hashing k
- Mapper x: Hash(k) = i → I will send k to reducer i
- Mapper y: Hash(k) = i → I will send k to reducer i
- E.g., Hash("waterloo") = 2

16

Each mapper can independently hash any key like k to find out which reducer it should go to.


Map
Reduce
S1
S2
S3
S19
S20
(waterloo, 5)
(kitchener, 2)
(city,10)
…
…
…
…
(university, 4)
(waterloo, 21)
(city, 4)
…
(waterloo, 36)
(university, 500)
…
(city, 1800)
(kitchener, 500)
…
17



The process of moving intermediate results from mappers to reducers called shuffling

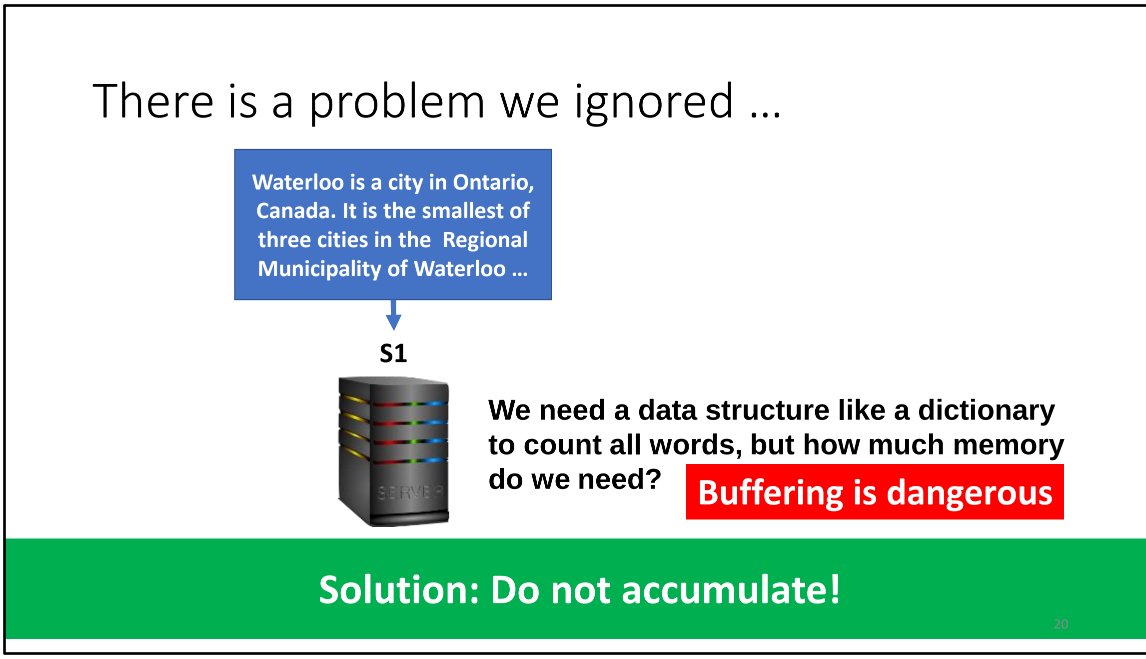
There is a problem we ignored …

S1

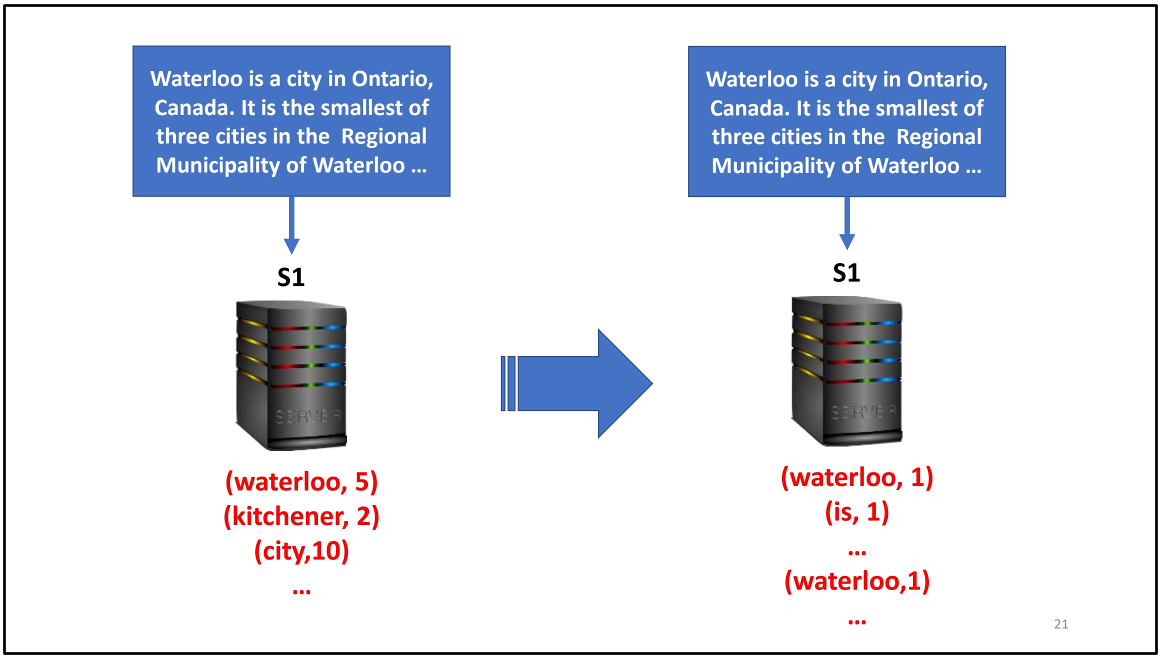
(waterloo, 5)
(kitchener, 2)
(city,10)
…

What if this list is too long?

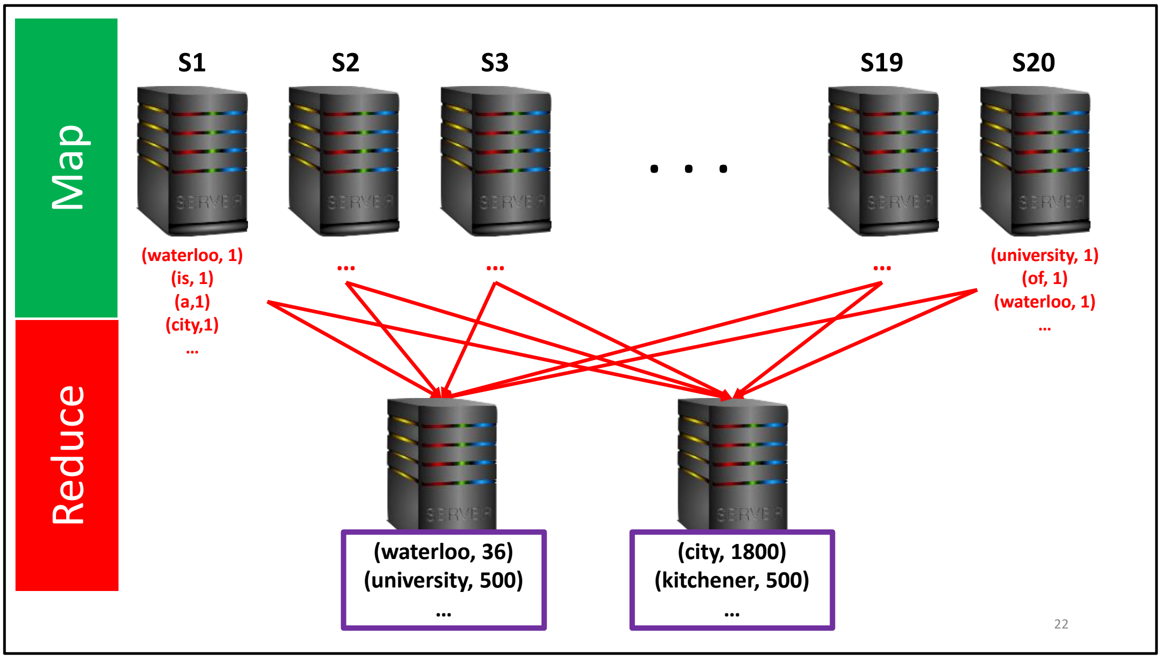
We might have memory overflow on mappers!



Unfortunately if we want to accumulate all stats in a dictionary, it may need too much memory. Although in the case of English Text the size of the dictionary is limited to the number of English words, no assumption can be made for an arbitrary input.



For every word we read emit (word, 1) to the reducer! This way the memory we need is almost 0.



We need no change in the reduce phase. Reducers should still add all numbers for each key.

MapReduce “word count” pseudo-code

```
def map(key: Long, value: String) = {
  for (word <- tokenize(value)) {
    emit(word, 1)
  }
}


def reduce(key: String, values: Iterable[Int]) = {
  for (value <- values) {
    sum += value
  }
  emit(key, sum)
}
```

Mapper: simply process line by line. For every line emit (word, 1).
Reducer: for every word, count all of the 1s.

Apache Hadoop is the most famous open-source implementation of MapReduce

MapReduce Implementations

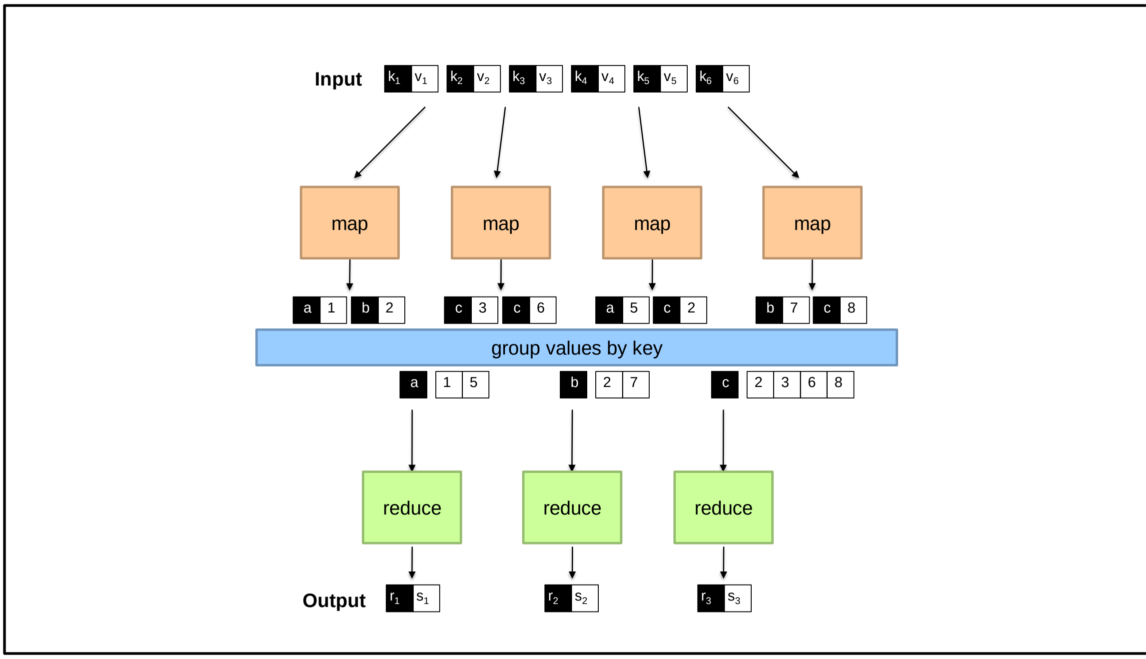
Google has a proprietary implementation in C++

Bindings in Java, Python

Hadoop provides an open-source implementation in Java

Development begun by Yahoo, later an Apache project
Used in production at Facebook, Twitter, LinkedIn, Netflix, …
Large and expanding software ecosystem
Potential point of confusion: Hadoop is more than MapReduce today

Lots of custom research implementations


Input
k1 v1 k2 v2 k3 v3 k4 v4 k5 v5 k6 v6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
group values by key
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
Output
r1 s1
r2 s2
r3 s3

MapReduce

Programmer specifies two functions:

map $(k_1, v_1) \rightarrow List[(k_2, v_2)]$

reduce $(k_2, List[v_2]) \rightarrow List[(k_3, v_3)]$

All values with the same key are sent to the same reducer

The execution framework handles everything else…

What's "everything else"?

MapReduce “Runtime”

Handles scheduling

Assigns workers to map and reduce tasks

Handles “data distribution”

Moves processes to data

Handles synchronization

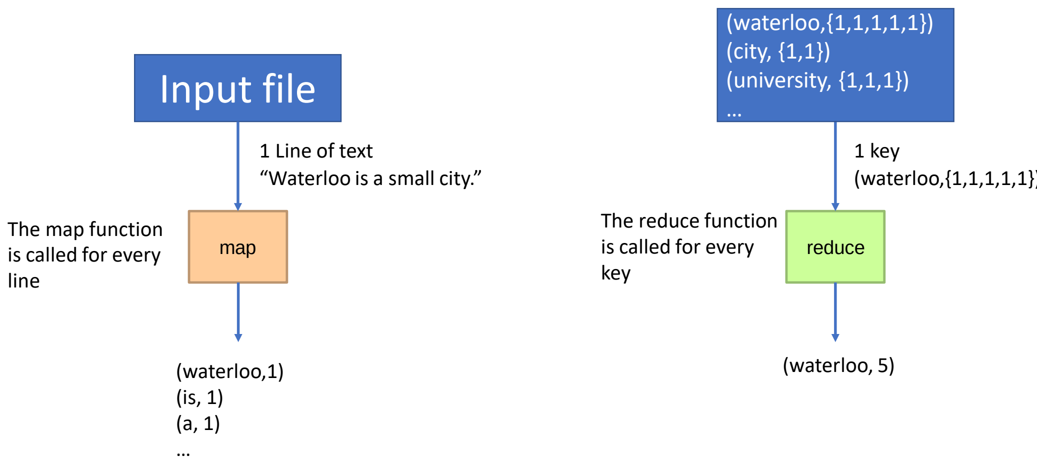
Groups intermediate data

Handles errors and faults

Detects worker failures and restarts

Everything happens on top of a distributed FS

The word count example …



MapReduce

Programmer specifies two functions:

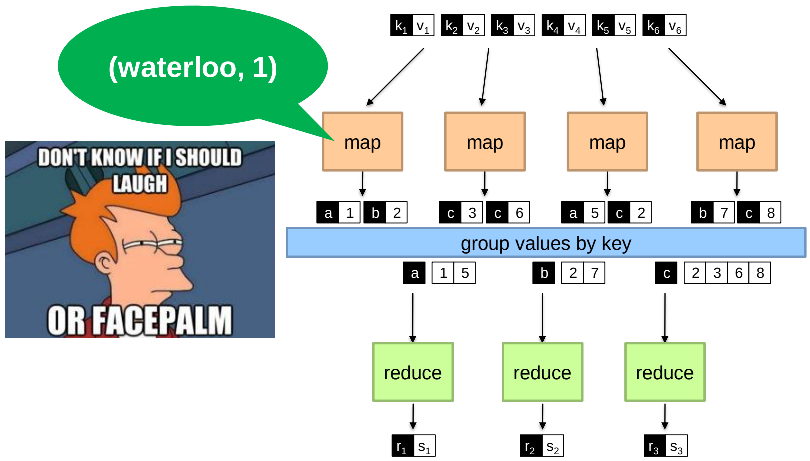
map $(k_1, v_1) \rightarrow List[(k_2, v_2)]$

reduce $(k_2, List[v_2]) \rightarrow List[(k_3, v_3)]$

All values with the same key are sent to the same reducer

The execution framework handles everything else…

Not quite…



What’s the most complex and slowest operation here?

The slowest operation is shuffling intermediate results from mappers to reducers

MapReduce

Programmer specifies ~~two~~ four functions:

map $(k_1, v_1) \rightarrow$ List[(k_2, v_2)]

reduce $(k_2,$ List[v_2]$) \rightarrow$ List[(k_3, v_3)]

All values with the same key are sent to the same reducer

partition $(k', p) \rightarrow 0 \ldots p-1$

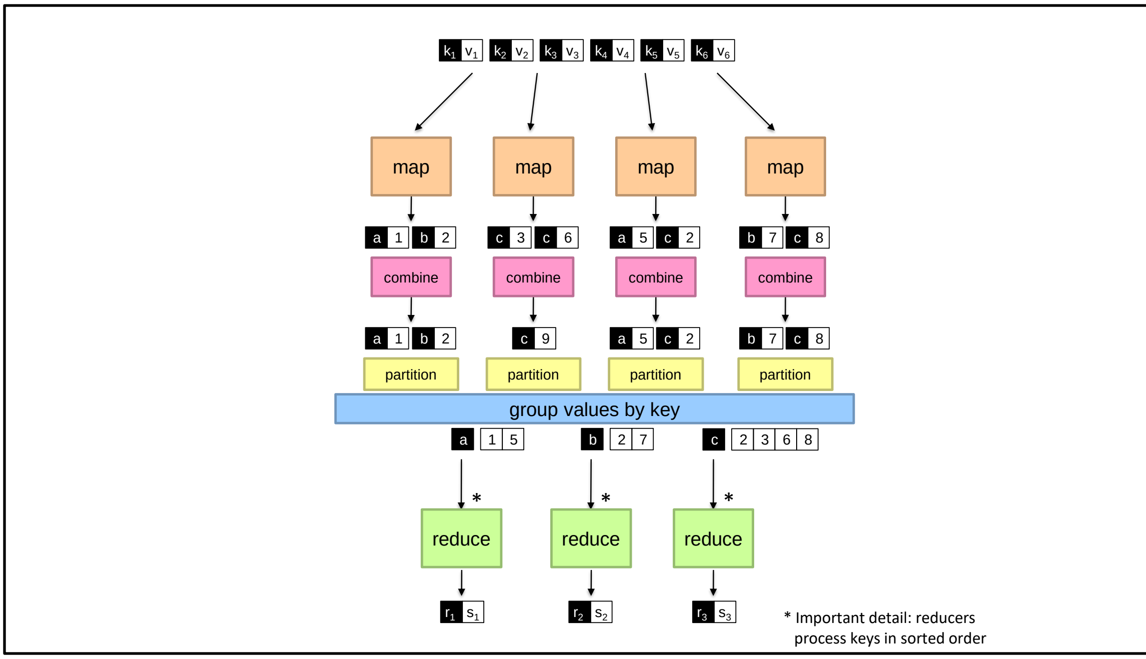
Often a simple hash of the key, e.g., hash(k') mod n

Divides up key space for parallel reduce operations

combine $(k_2,$ List[v_2]$) \rightarrow$ List[(k_2, v_2)]

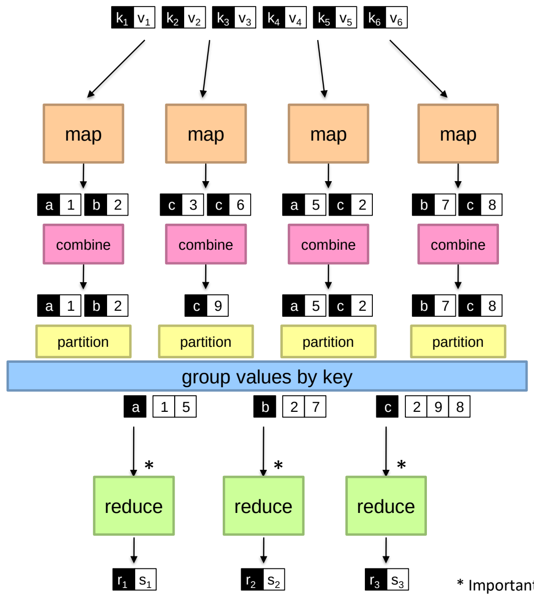
Mini-reducers that run in memory after the map phase

Used as an optimization to reduce network traffic



Partition is not a component that the data goes through, but rather a policy that determines to which reducer the output of mappers should go.

Logical View

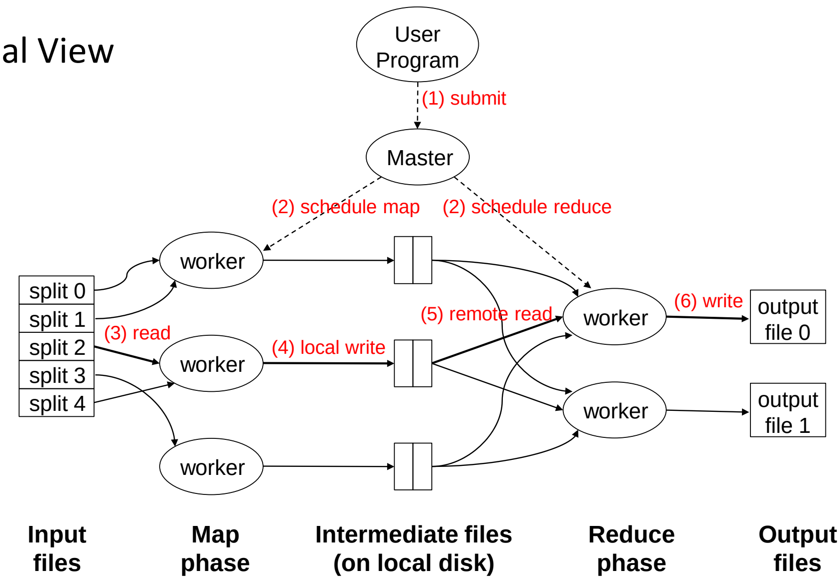


* Important detail: reducers process keys in sorted order

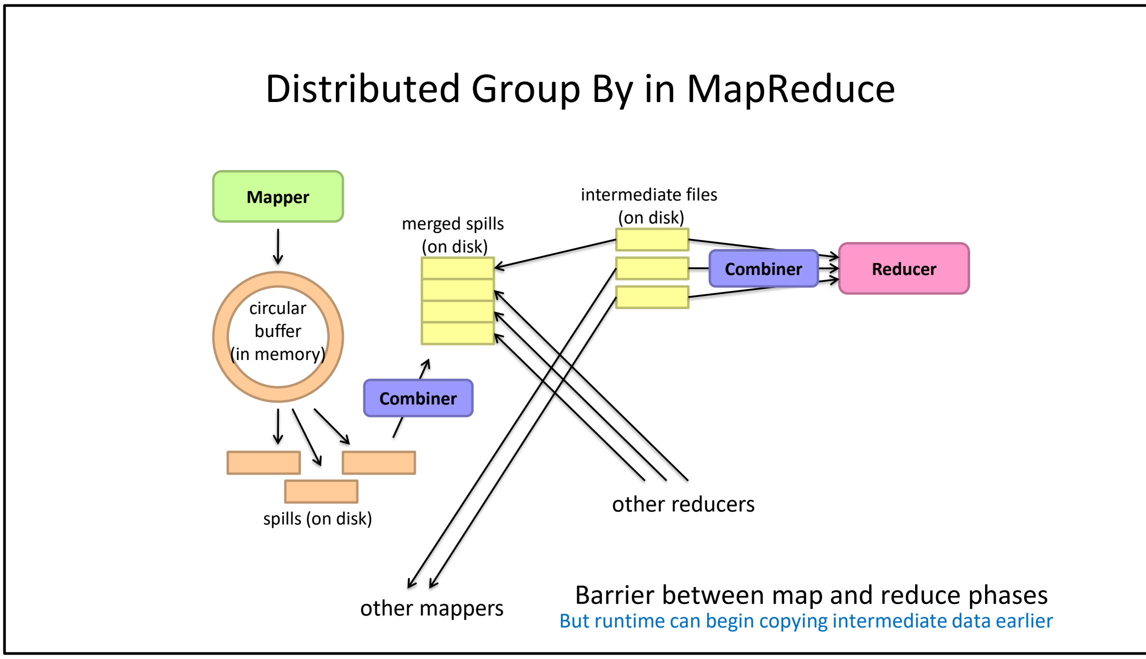
Physical view

What happens behind the scenes

Physical View

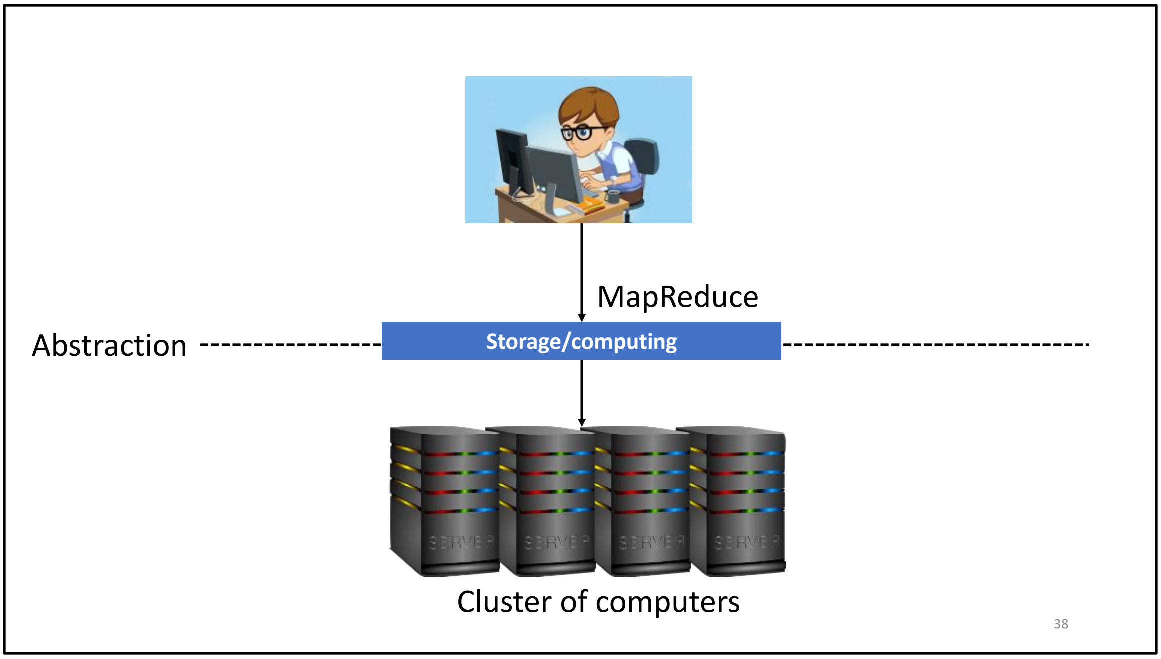


Adapted from (Dean and Ghemawat, OSDI 2004)



Map side:
Map outputs are buffered in memory in a circular buffer
When buffer reaches threshold, contents are "spilled" to disk
Spills are merged into a single, partitioned file (sorted within each partition)
Combiner runs during the merges

First, map outputs are copied over to reducer machine
"Sort" is a multi-pass merge of map outputs (happens in memory and on disk)
Combiner runs during the merges
Final merge pass goes directly into reducer



MapReduce hides the complexities of the physical view so that the programmer can focus on “what” rather than “how” it’s done

With this approach, the datacenter with all of its complexities is like a computer.