

Advanced Cloud Computing

MapReduce

Wei Wang
CSE@HKUST
Spring 2025



THE DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
計算機科學及工程學系

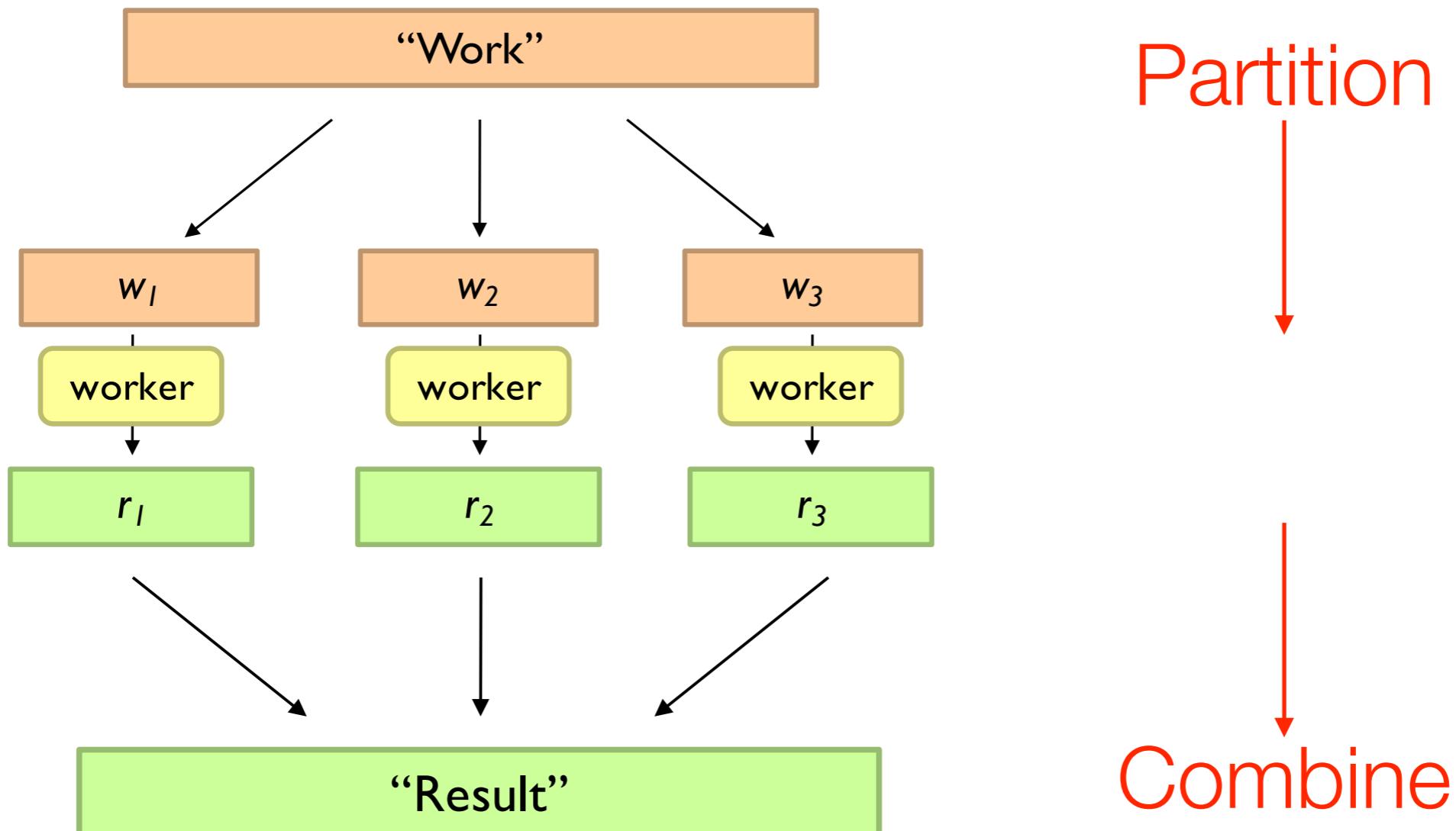
In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers.

— Grace Hopper

A photograph of a woman with long hair, wearing a dark blue Google polo shirt, standing in a dimly lit server room. She is leaning over a mobile cart, looking at a laptop screen. On the cart next to her is a white styrofoam container. The server racks on either side are filled with glowing blue and yellow lights from the internal hardware. The floor has some equipment labels like "G tweden LE 049".

How do we program this thing?

Divide and conquer



Parallelization challenges

How do we assign work units to workers?

What if we have more work units than workers?

What if workers need to share partial results?

How do we aggregate partial results?

How do we know all the workers have finished?

What if workers die?

What's the common theme of all of these problems?

Common theme?

Parallelization problem arise from

- ▶ communication between workers (e.g., state exchange)
- ▶ access to shared resources (e.g., data)

Thus, we need a **synchronization mechanism**



Managing multiple workers

Damn hard!

- ▶ don't know the order in which workers run
- ▶ don't know when workers interrupt each other
- ▶ don't know when workers need to communicate partial results
- ▶ don't know the order in which workers access shared data

Managing multiple workers

Thus, we need

- ▶ semaphores (lock, unlock)
- ▶ conditional variables (wait, notify, broadcast)
- ▶ barriers (a job cannot start until its prerequisites have completed)

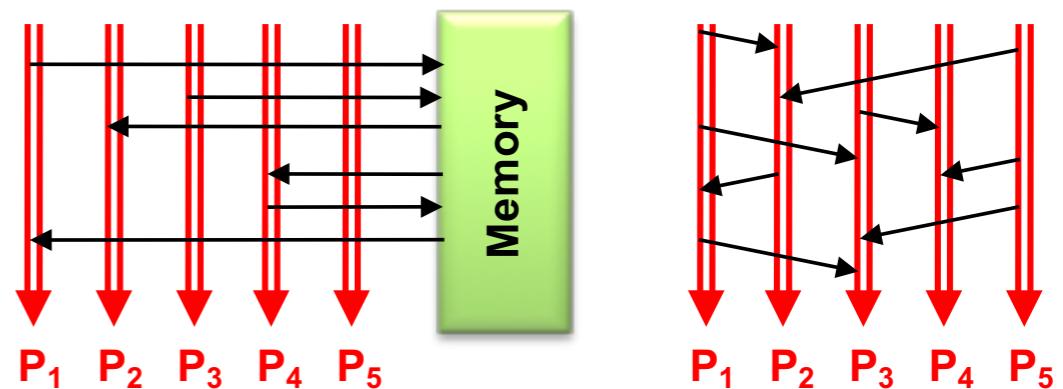
But still...

- ▶ deadlock, race conditions...
- ▶ dinning philosophers, sleeping barbers...

Current tools

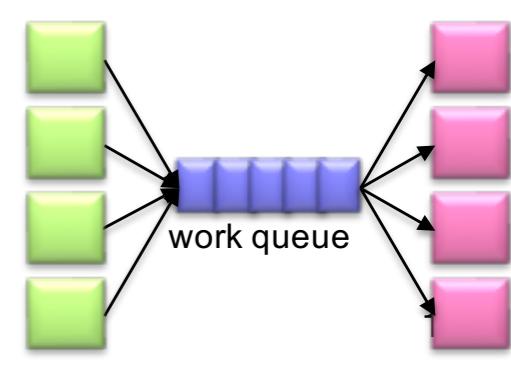
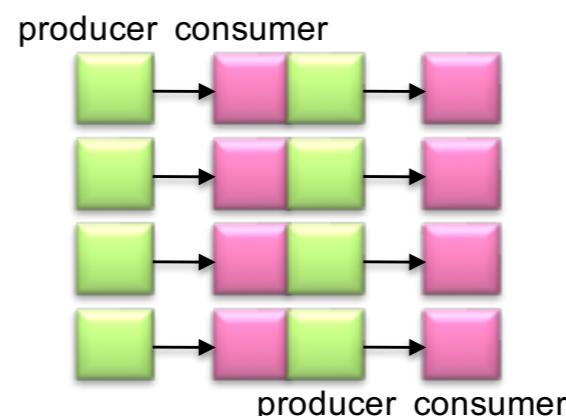
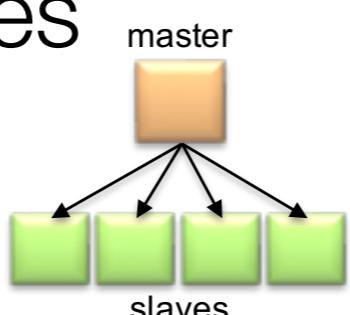
Programming models

- ▶ shared memory (pthreads)
- ▶ message passing (MPI)



Design Patterns

- ▶ master-slaves
- ▶ producer-consumer flows
- ▶ shared work queues



Where the rubber meets the road

Concurrency is difficult to reason about

And it is even more so

- ▶ at the scale of datacenters
- ▶ in the presence of failures
- ▶ in terms of multiple interacting services

Not to mention debugging...

Where the rubber meets the road

The reality:

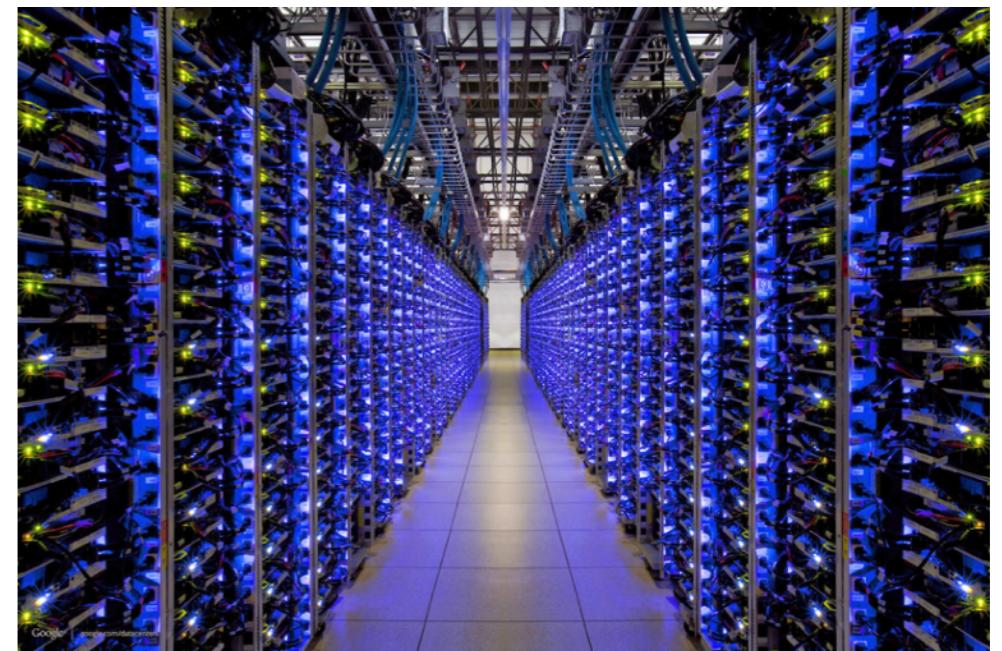
- ▶ lots of one-off solutions, workaround, custom code
- ▶ write your own dedicated library, then code with it
- ▶ burden on the programmer to explicitly manage everything
- ▶ ...



The datacenter *is* a computer



OS X Yosemite



MapReduce

A wide-angle photograph of a massive server room. The floor is a polished concrete grid. Numerous server racks are arranged in long rows, their front panels glowing with blue and white lights. Above the racks, a complex steel truss ceiling is illuminated by numerous rectangular fluorescent lights. The perspective is from a low angle, looking up at the ceiling, emphasizing the scale of the facility.

Typical big data problems

Log analysis:

- ▶ how many warning messages were received last week?
 - ▶ e.g., [Warning] 21:07/01/04/2017 Low memory!

Web mining:

- ▶ which wiki pages about Donald Trump have been viewed the most times during the 2016 US Election?
- ▶ How many tweets mentioned the word “terrorism” yesterday?

Typical big data problems

Iterate over a large number of records

Extract something of interest from each

Shuffle and sort intermediate results

Aggregate intermediate results

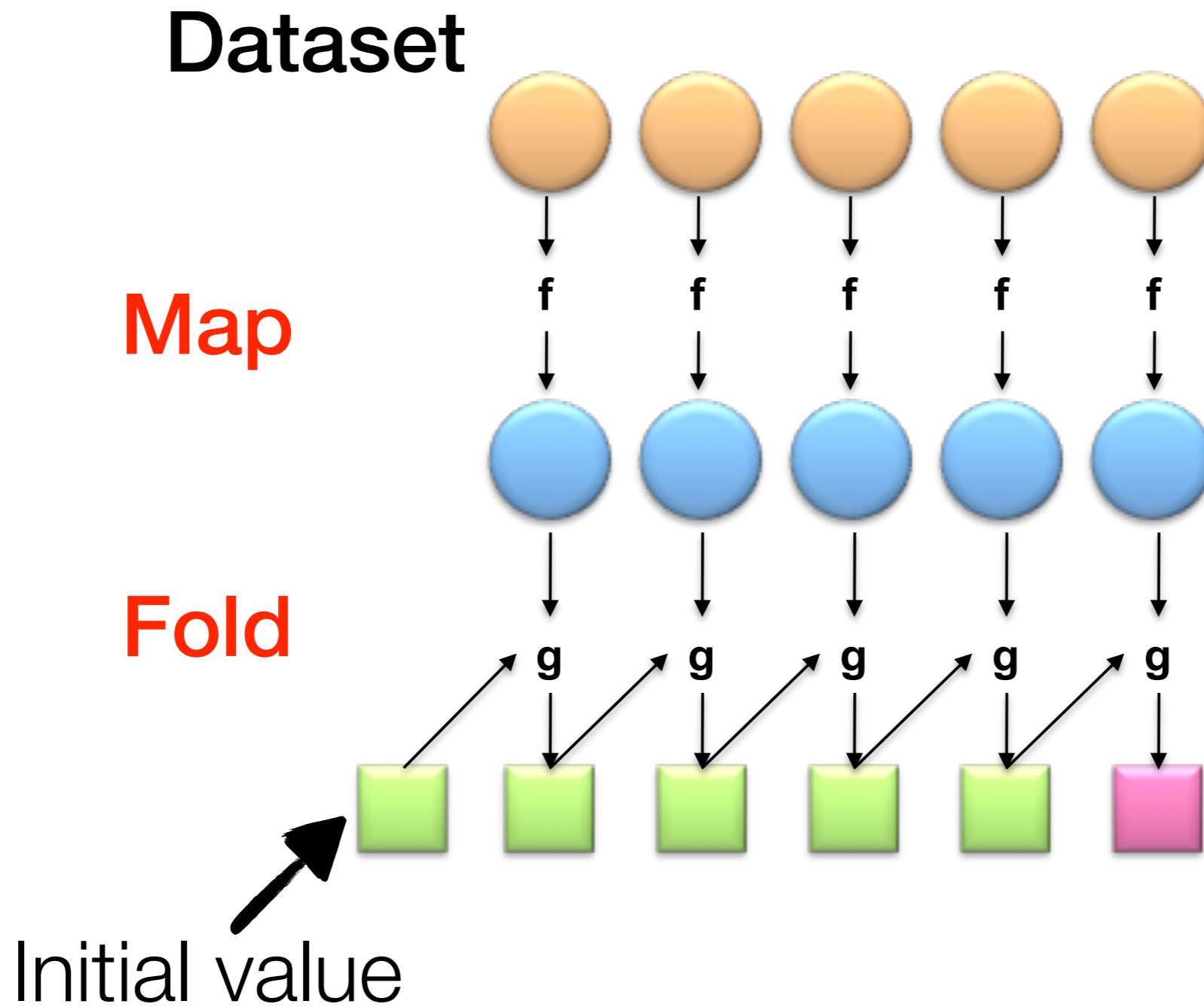
Generate final output

Map

Reduce

Key idea: provide a *functional* abstraction for these two operations

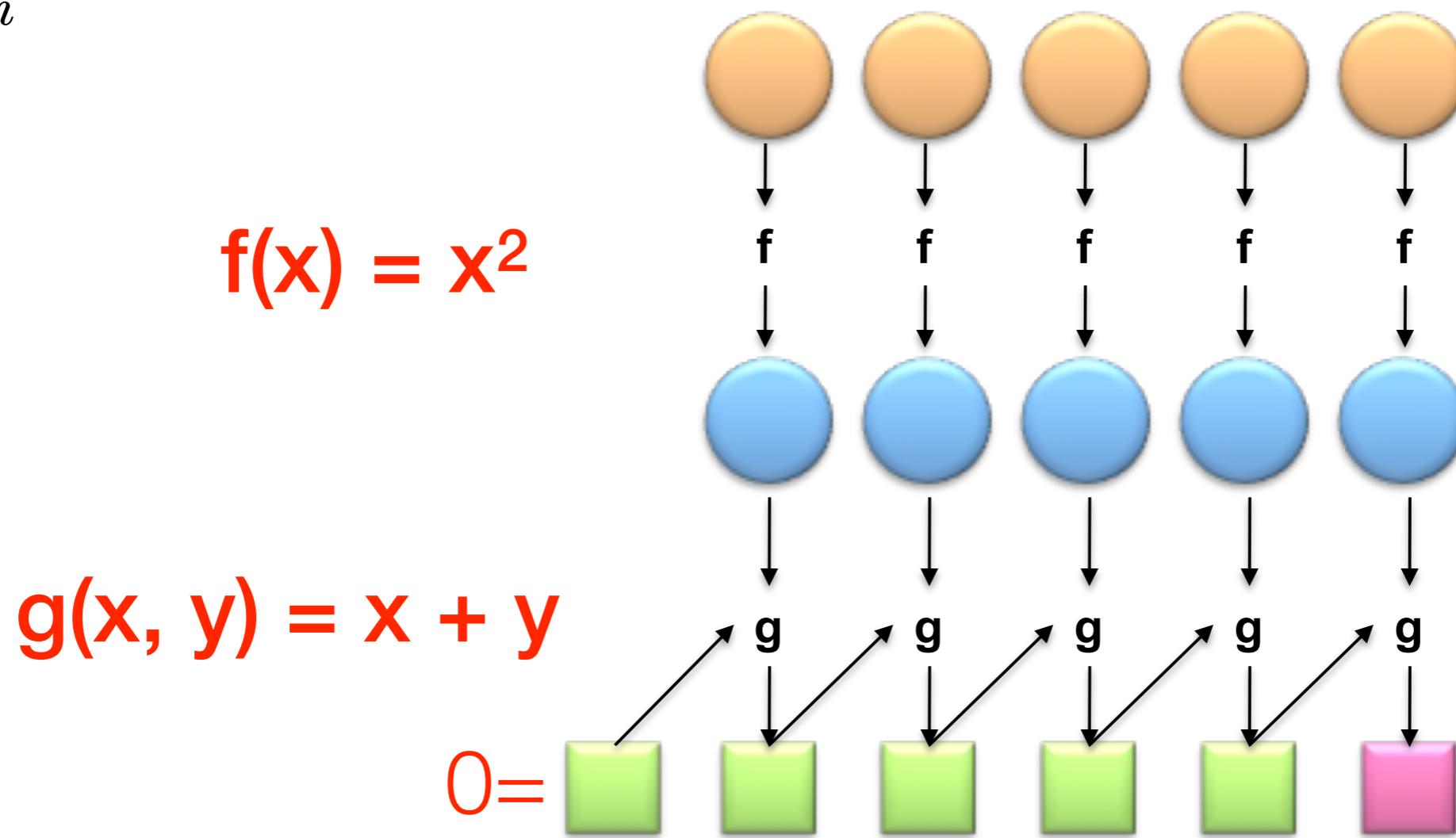
Roots in functional programming



Functional programming

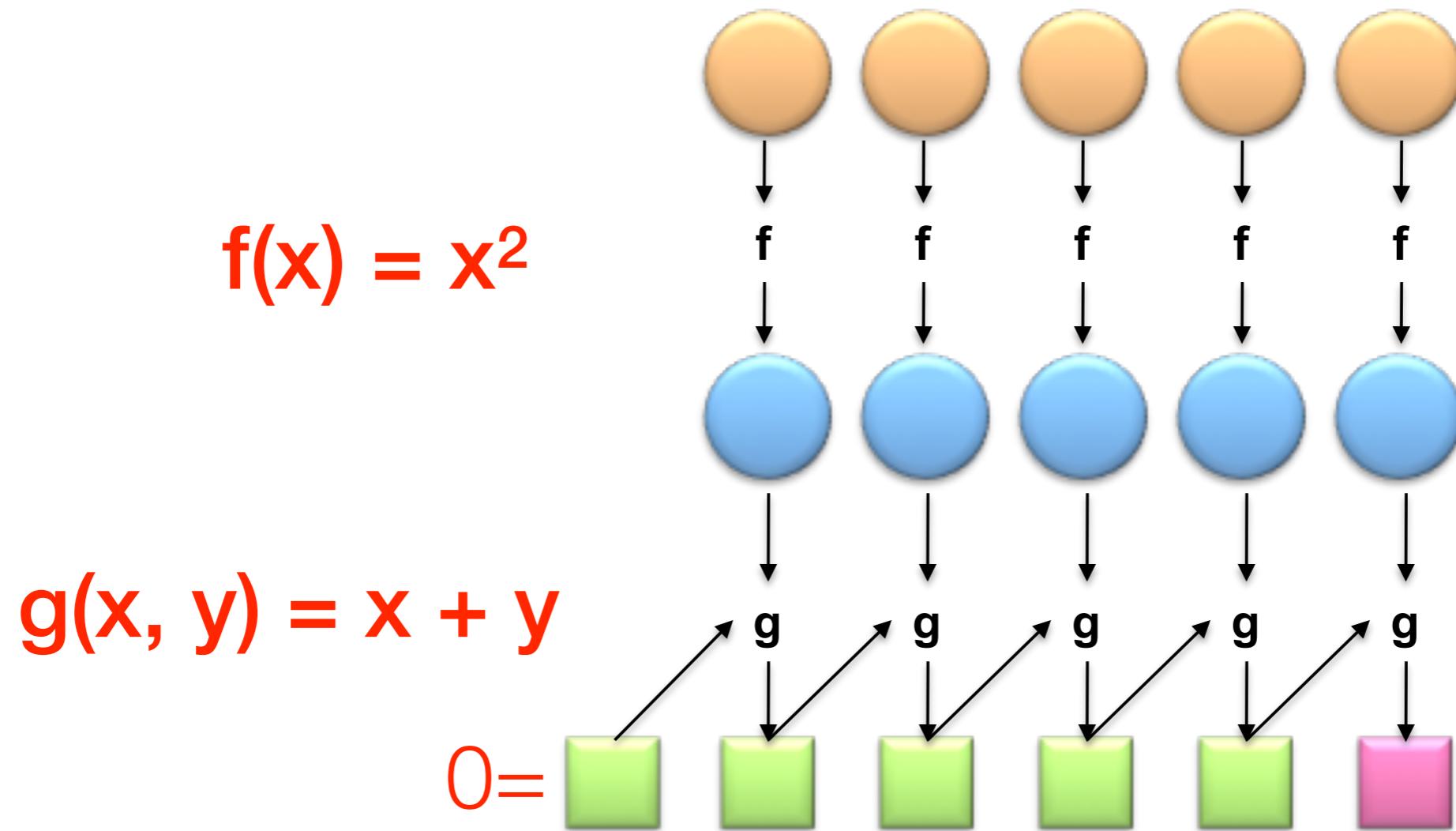
Given a dataset $X = [x_1, \dots, x_n]$, compute the square sum

$$\sum_i x_i^2$$



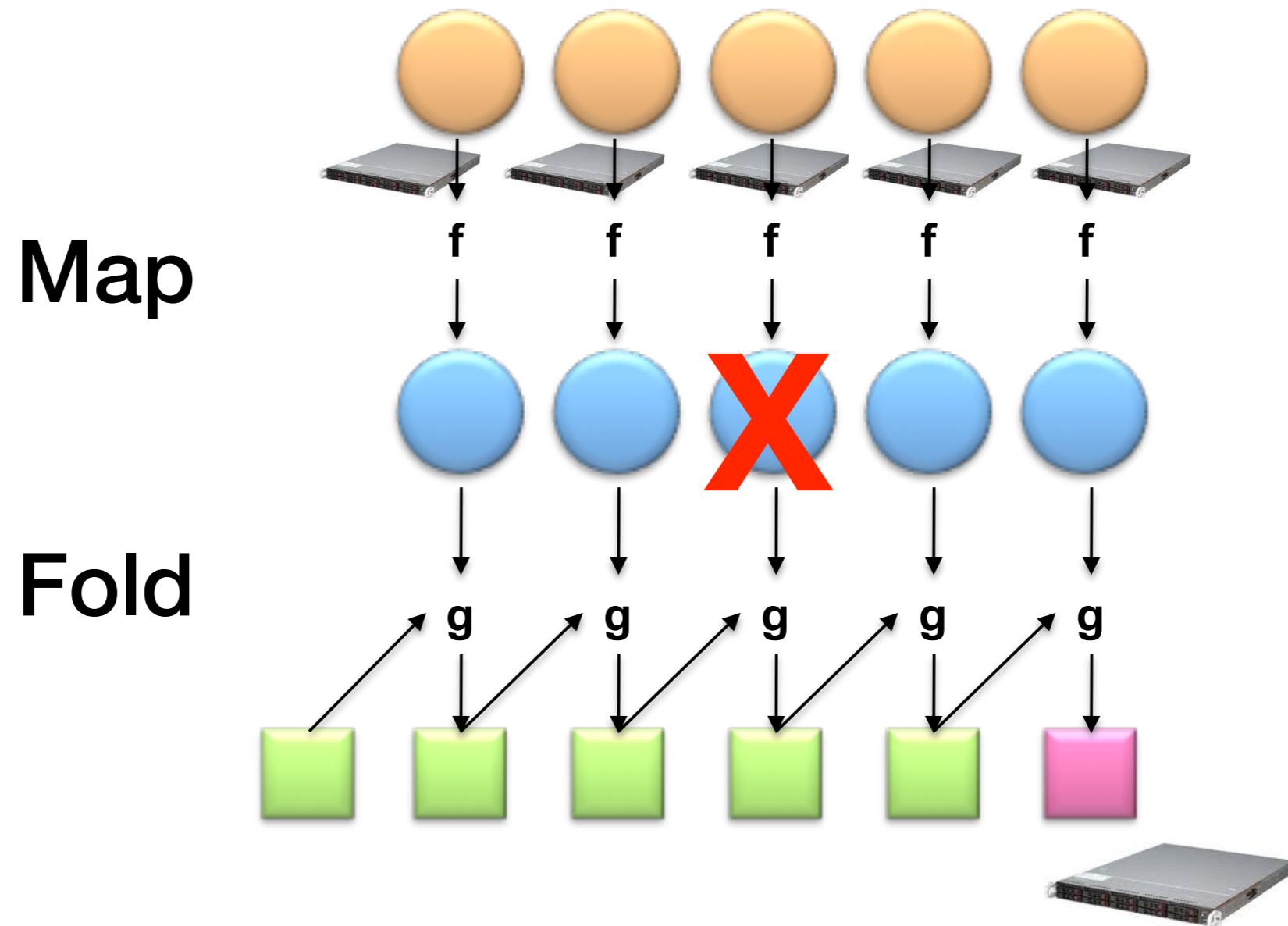
Functional programming

Functional operations never modify existing datasets, but they **create new ones**



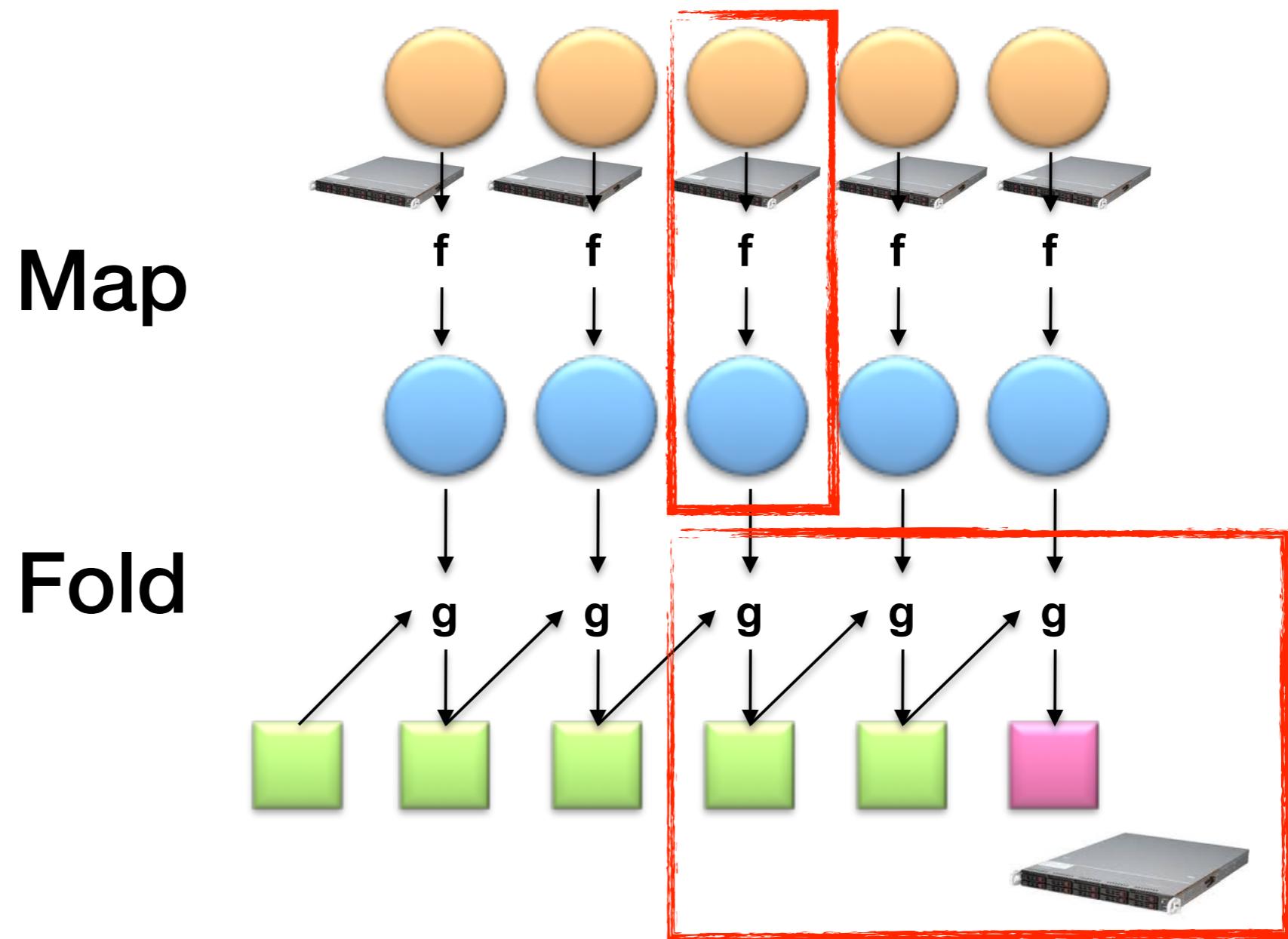
Ideal for parallelization

What if a worker fails?

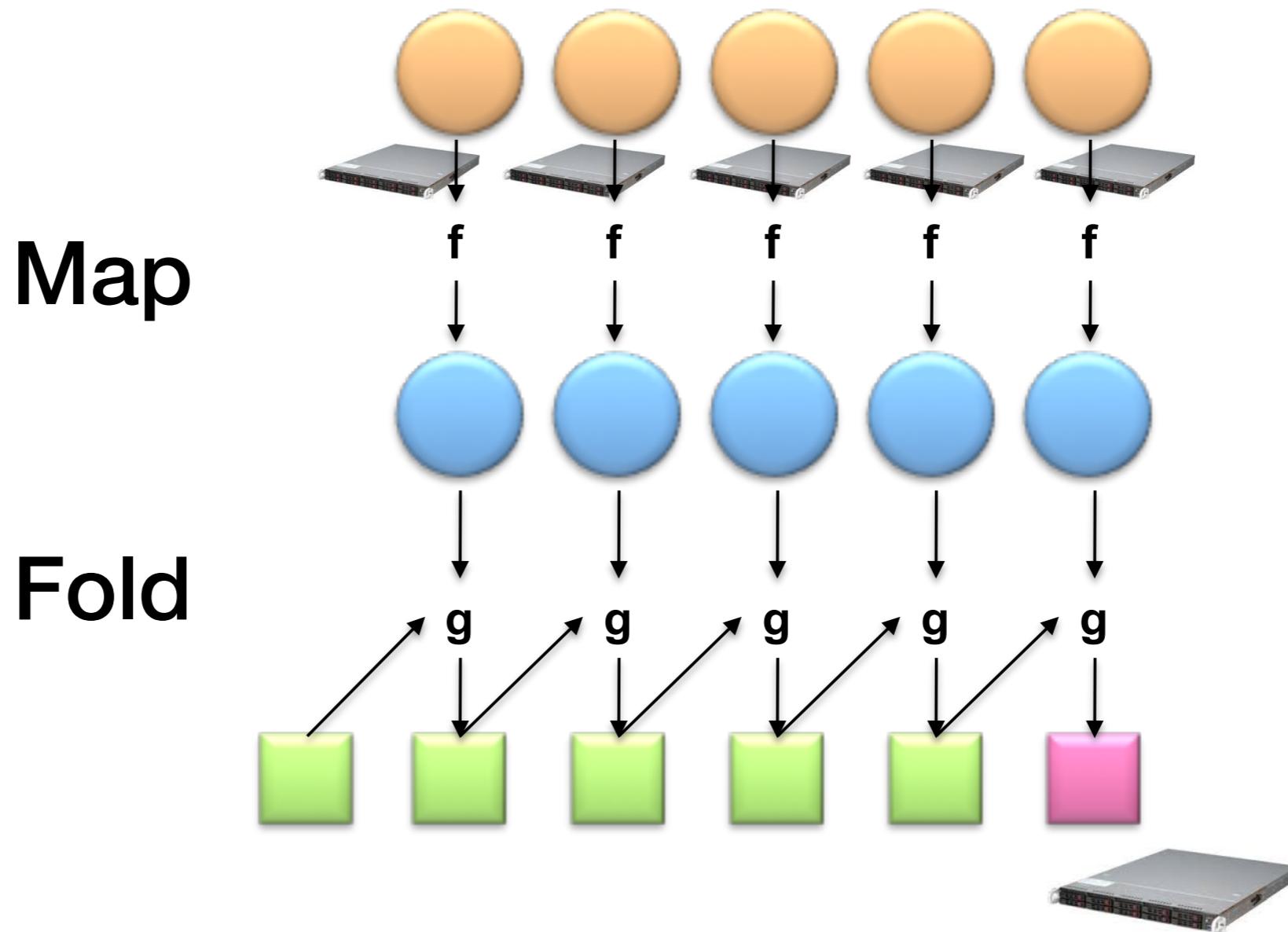


Ideal for parallelization

Do the work again, on some other machines

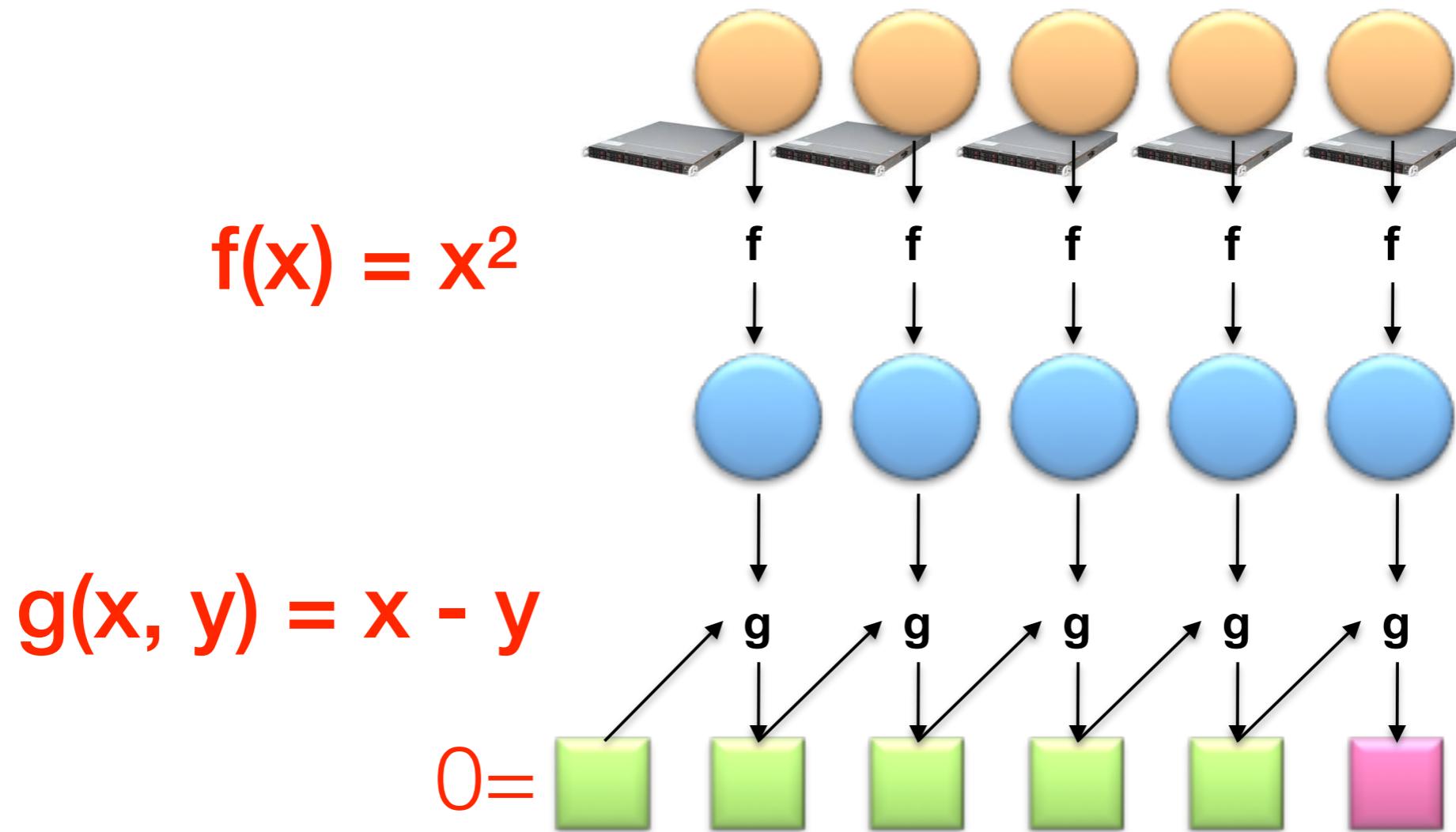


Can we apply any function?



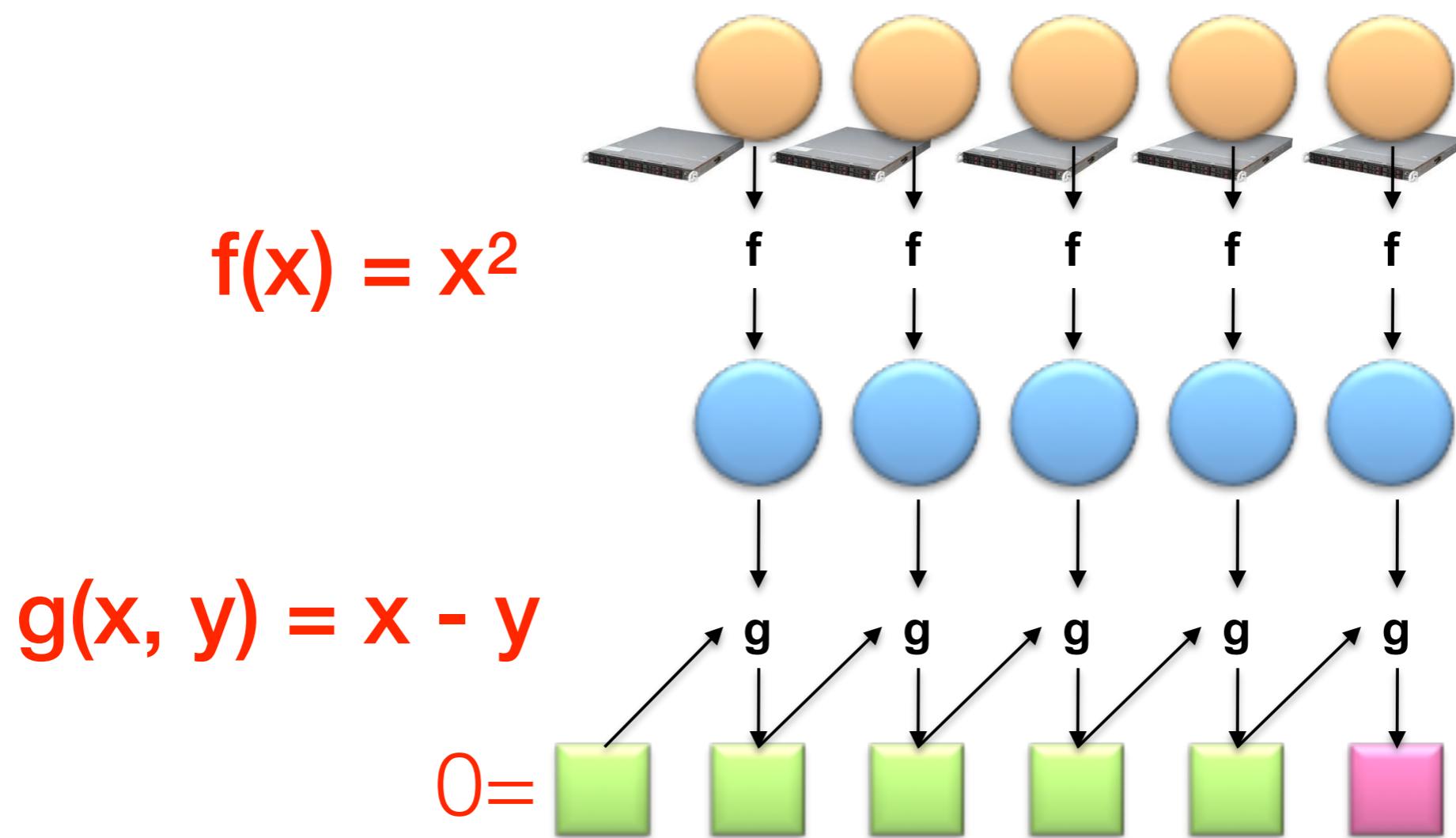
Can we apply any function?

What if $g(x, y) = x - y$?



Nope...

The order matters, making the results indefinite and hard to reason about!



Thus, we require...

Commutativity

- ▶ $g(x, y) = g(y, x)$
- ▶ e.g., $x + y = y + x$

Associativity

- ▶ $g(g(x, y), z) = g(x, g(y, z))$
- ▶ e.g., $(x + y) + z = x + (y + z)$

The programming model of MapReduce
borrows from functional programming

Records from the data source are fed as
key-value pairs, e.g., [**<filename, line>**]

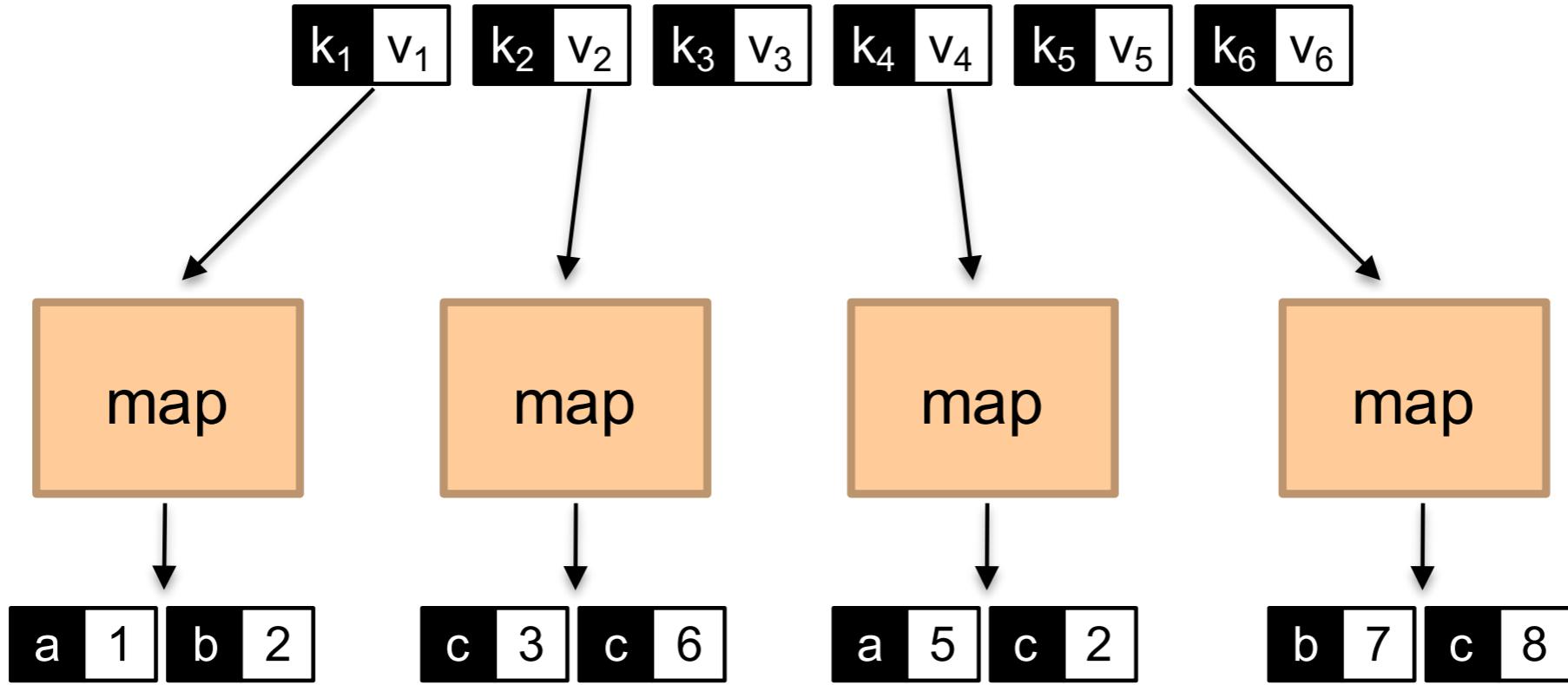
MapReduce

Programmers specify two functions

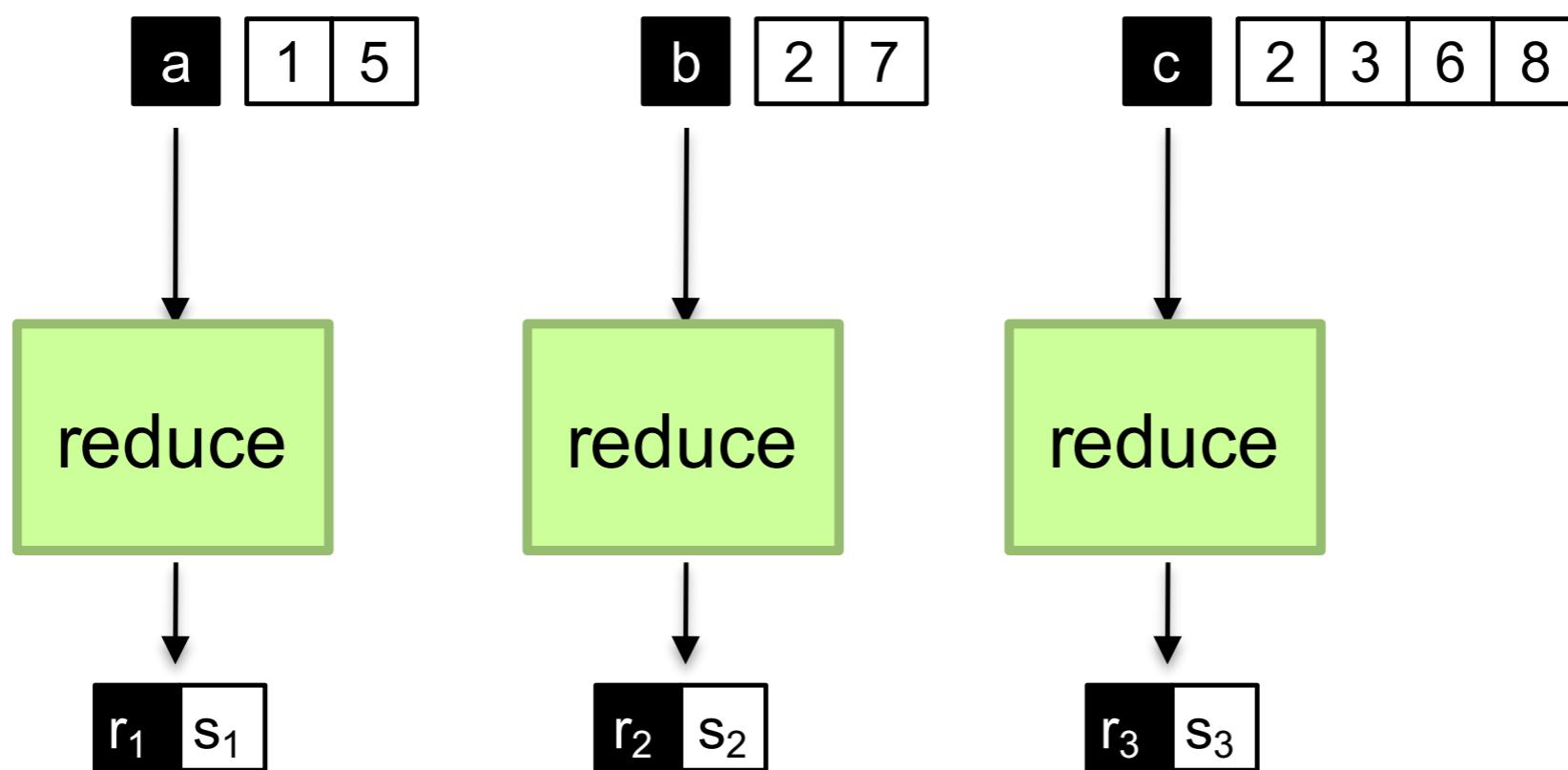
- ▶ **map** $(k, v) \rightarrow [k_2, v_2]$
- ▶ **reduce** $(k_2, [v_2]) \rightarrow [k_3, v_3]$

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

The execution framework handles ***everything else***...



Shuffle and Sort: aggregate values by keys

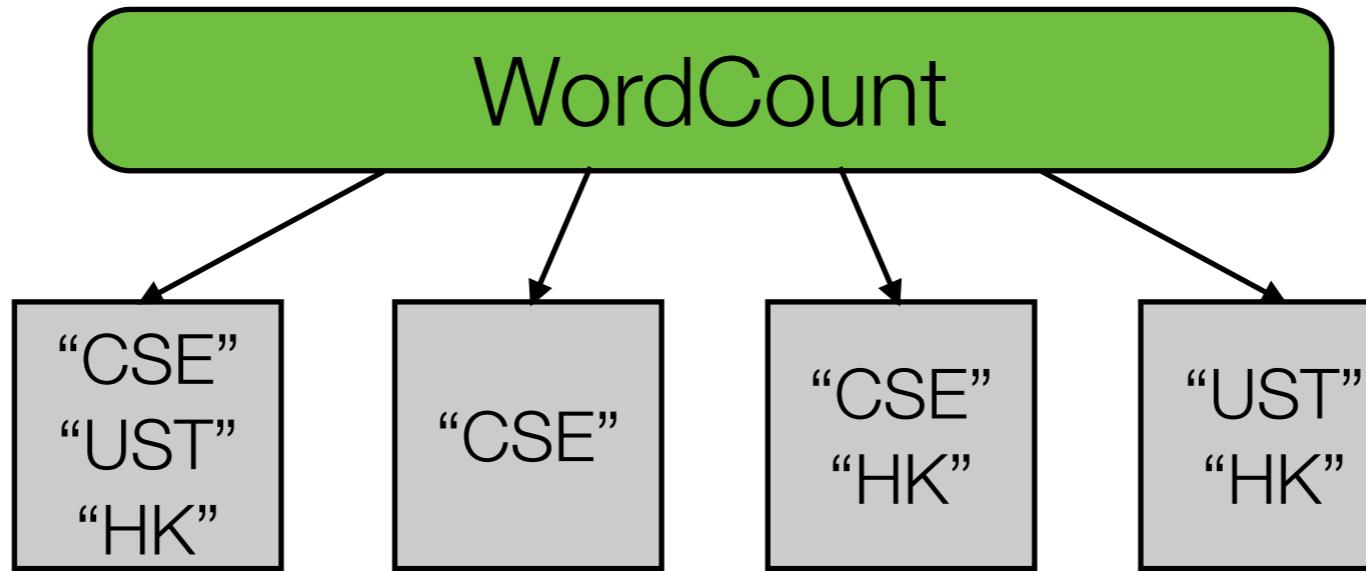


WordCount: a “Hello World” from MapReduce

WordCount

Count the occurrence of each word in a large document

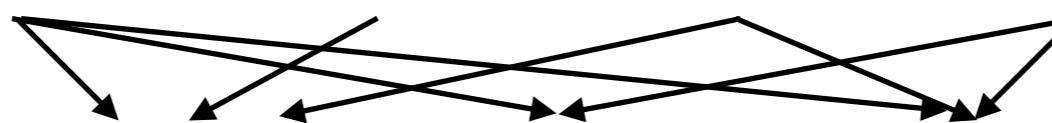
HDFS



Map

(“CSE”, 1)
("UST", 1) (“CSE”, 1)
("HK", 1) (“CSE”, 1)
 ("UST", 1)
 ("HK", 1)

Sort & Shuffle



(“CSE”, [1, 1, 1]) (“UST”, [1, 1]) (“HK”, [1, 1, 1])

Reduce

↓
("CSE", 3) ("UST", 2) ("HK", 3)

Collect

A yellow rounded rectangle at the bottom containing the final output: ("CSE", 3), ("UST", 2), ("HK", 3). Three arrows point up from the boxes in the Sort & Shuffle stage to this output box.

(“CSE”, 3), (“UST”, 2), (“HK”, 3)

A tale of two functions...

What to emit?

Map(String docid, String text):

for each word w in text:

 Emit(w, 1);

Reduce(String term, Iterator<Int> values):

int sum = 0;

for each v in values:

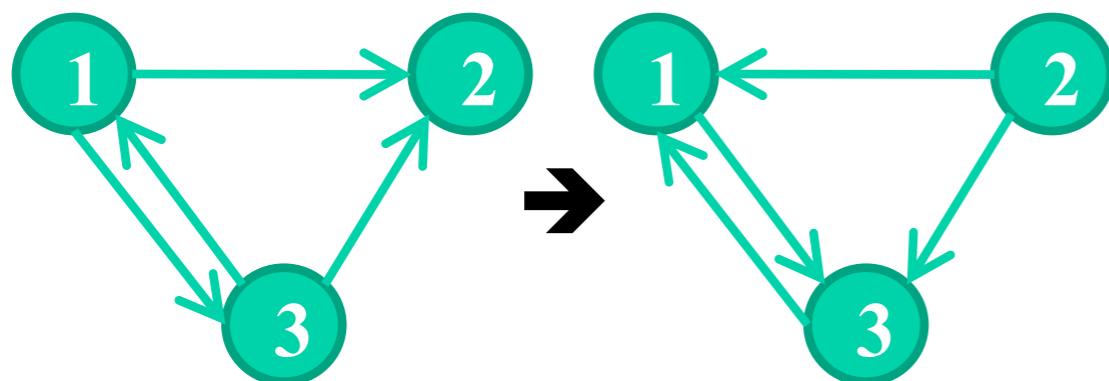
 sum += v;

 Emit(term, sum)

How to reduce?

Reverse graph edge directions

Adjacency list of graph (3 nodes and 4 edges)



Input

$(3, [1, 2])$
 $(1, [2, 3])$

Map

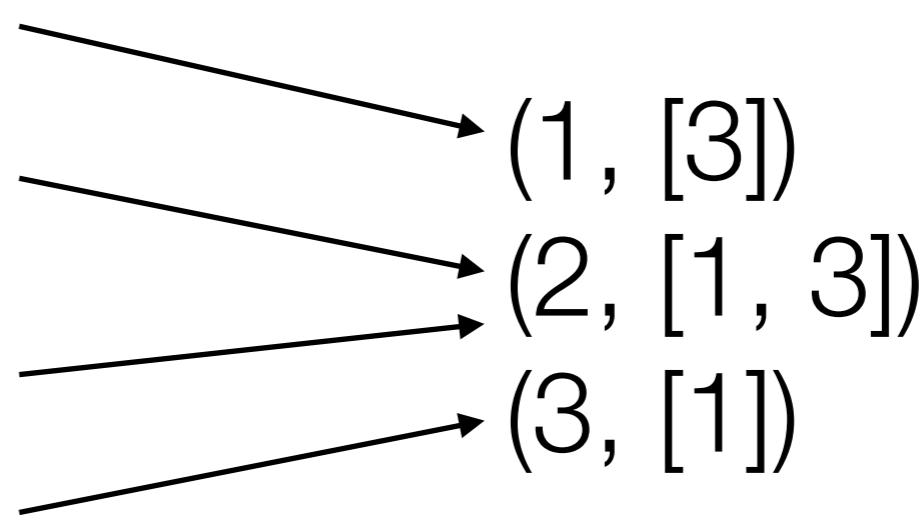
$(1, [3])$
 $(2, [3])$

\Rightarrow

$(2, [1])$
 $(3, [1])$

Shuffle

Output



Problems solved by MapReduce

Read a lot of data

Map: $(k_1, v_1) \rightarrow [<k_2, v_2>]$

- ▶ extract something you care about from each record

Shuffle and sort

Reduce: $(k_2, [v_2]) \rightarrow [<k_3, v_3>]$

- ▶ aggregate, summarize, filter, or transform

Write the results

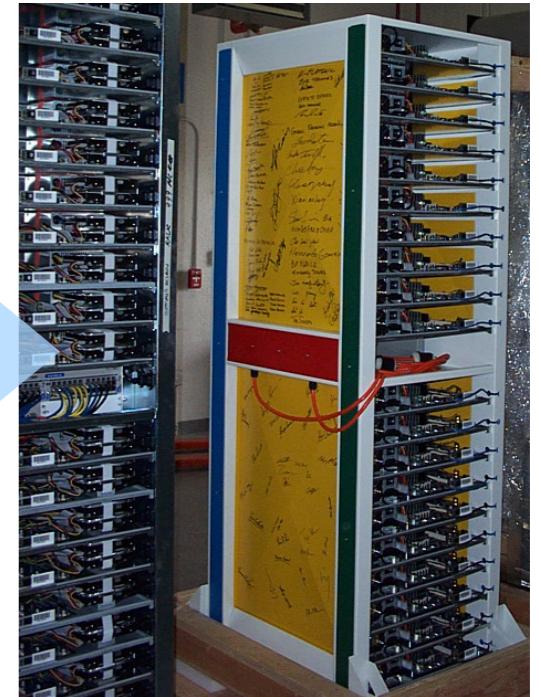
Google Map



Geographic Data



Index Files

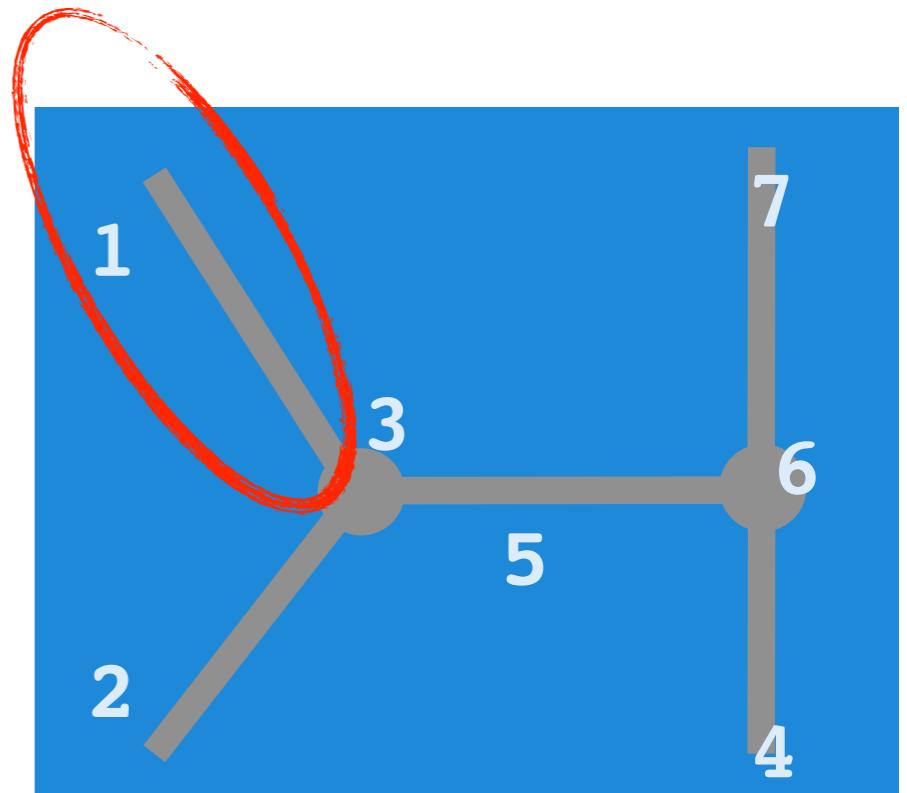


Datacenter

Input

Feature List

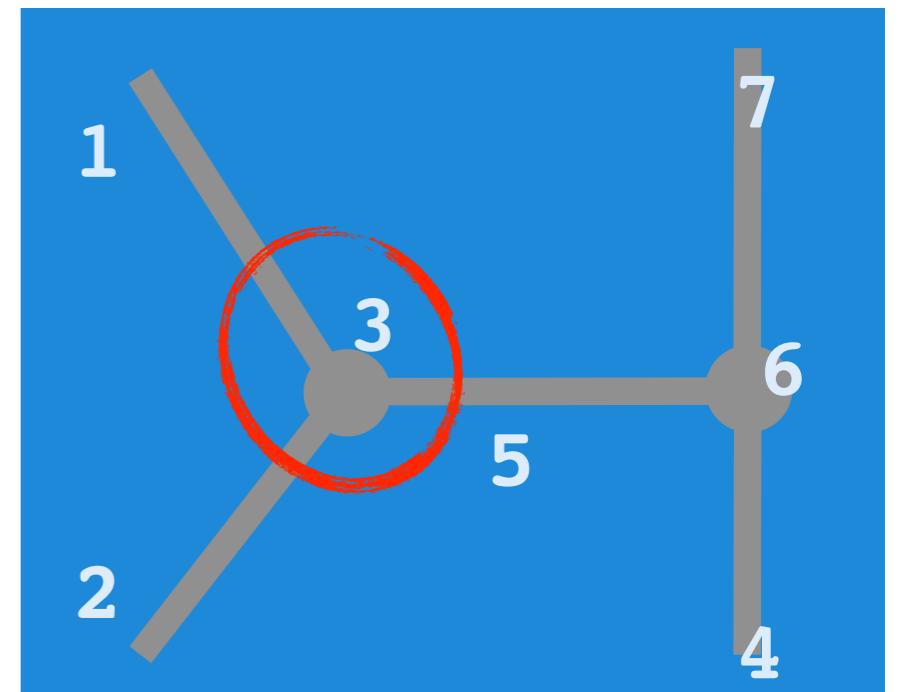
- 1: <type=Road>, <intersections=(3)>, <geom>, ...
- 2: <type=Road>, <intersections=(3)>, <geom>, ...
- 3: <type=Intersection>, <roads=(1,2,5)>, ...
- 4: <type=Road>, <intersections=(6)>, <geom>, ...
- 5: <type=Road>, <intersections=(3, 6)>, <geom>, ...
- 6: <type=Intersection>, <roads=(5,6,7)>, ...
- 7: <type=Road>, <intersections=(6)>, <geom>, ...



Input

Feature List

- 1: <type=Road>, <intersections=(3)>, <geom>, ...
- 2: <type=Road>, <intersections=(3)>, <geom>, ...
- 3: <type=Intersection>, <roads=(1,2,5)>, ...
- 4: <type=Road>, <intersections=(6)>, <geom>, ...
- 5: <type=Road>, <intersections=(3, 6)>, <geom>, ...
- 6: <type=Intersection>, <roads=(5,6,7)>, ...
- 7: <type=Road>, <intersections=(6)>, <geom>, ...



Output

Intersection List

3: <type=Intersection>, <roads=(

1: <type=Road>, <geom>, <name>, ...

2: <type=Road>, <geom>, <name>, ...

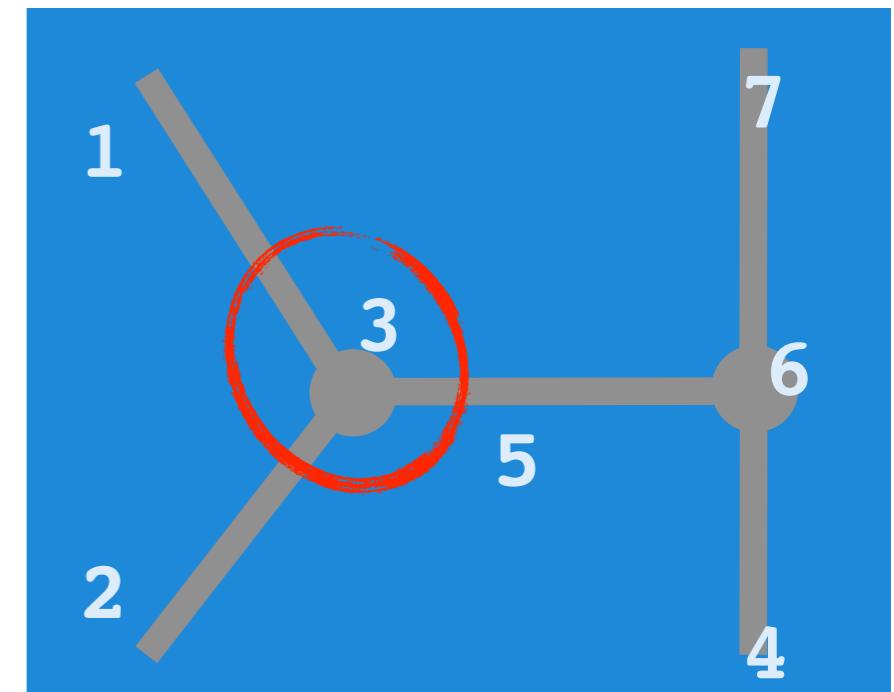
5: <type=Road>, <geom>, <name>, ...)>, ...

6: <type=Intersection>, <roads=(

4: <type=Road>, <geom>, <name>, ...

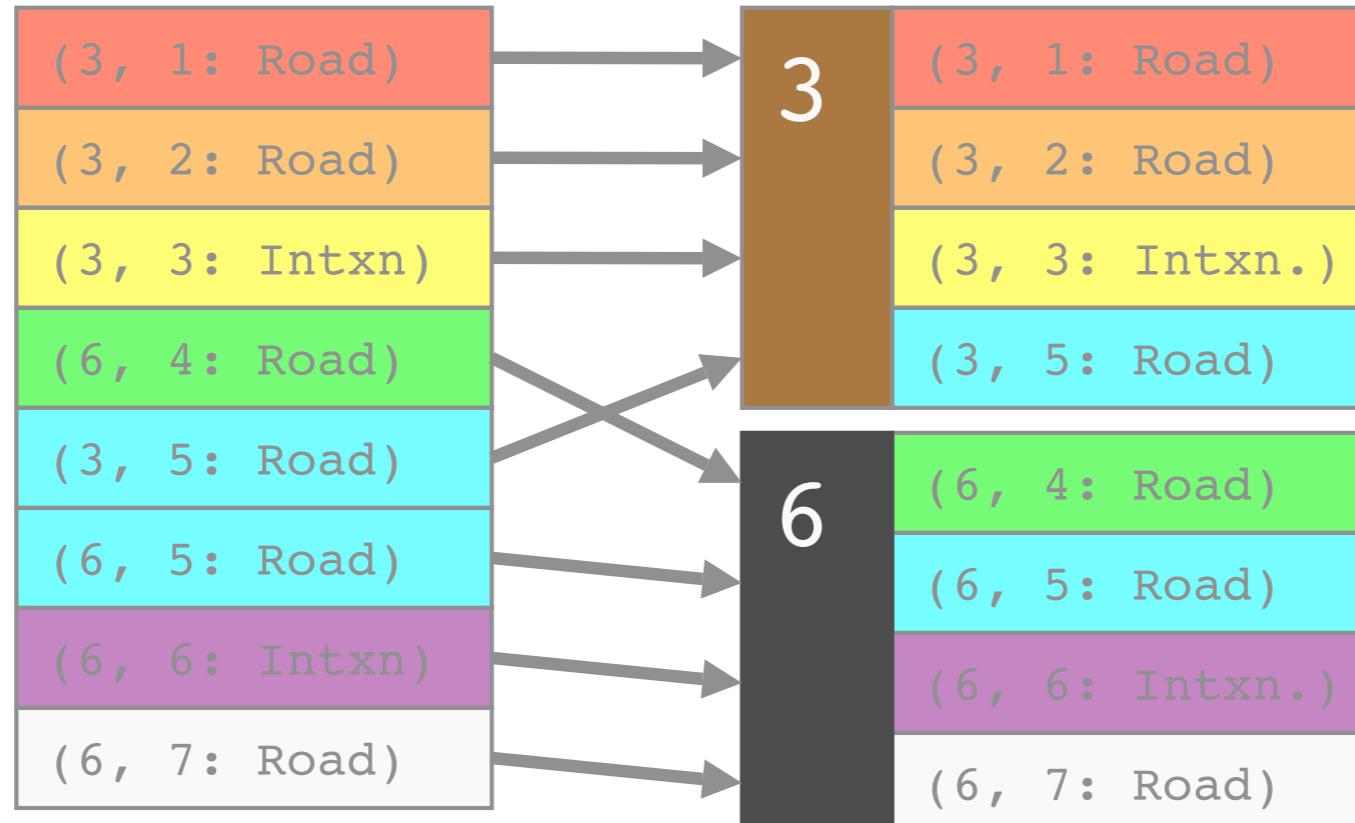
5: <type=Road>, <geom>, <name>, ...

7: <type=Road>, <geom>, <name>, ...)>, ...



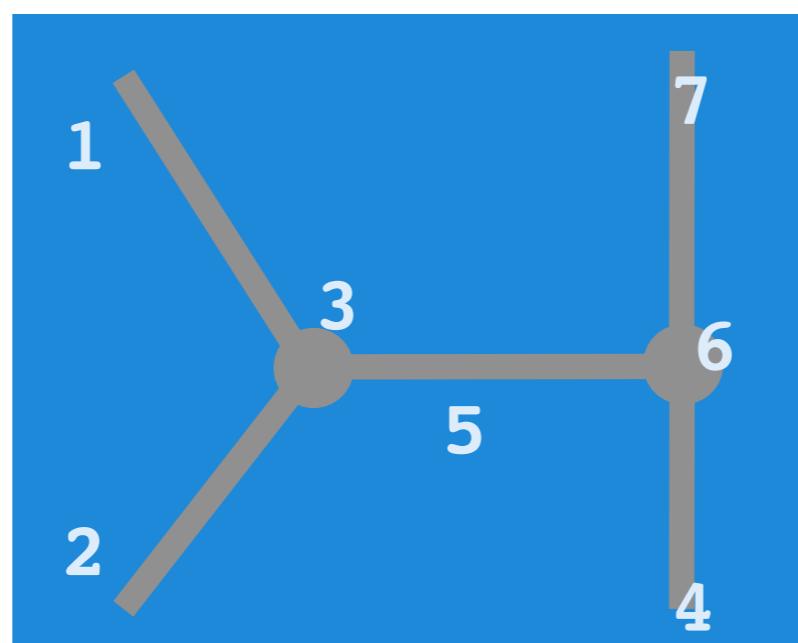
Input
list of (k, v) Map Shuffle Reduce Output
emit (k', v') sort by key (k', [v']) new list of items

1: Road
2: Road
3: Intersection
4: Road
5: Road
6: Intersection
7: Road



3: Intersection
1: Road,
2: Road,
5: Road

6: Intersection
4: Road,
5: Road,
7: Road



MapReduce

Programmers specify two functions

- ▶ **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 sent to the **same reducer**

The execution framework handles ***everything else...***

What's “*everything else*”?

Everything else...

Handles scheduling

- ▶ assigns workers to map and reduce tasks
- ▶ load balancing

Handles “data distribution”

- ▶ move processes to data
- ▶ automatic parallelization

Everything else...

Handles synchronization

- ▶ gathers, sorts, and shuffles intermediate data
- ▶ network and disk transfer optimization

Handles errors and faults

- ▶ detects worker failures and restarts

Everything happens on top of a distributed filesystem

MapReduce refinement

Programmers specify two functions

- ▶ **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

Not quite... usually, programmers also specify **combiner** and **partitioner**

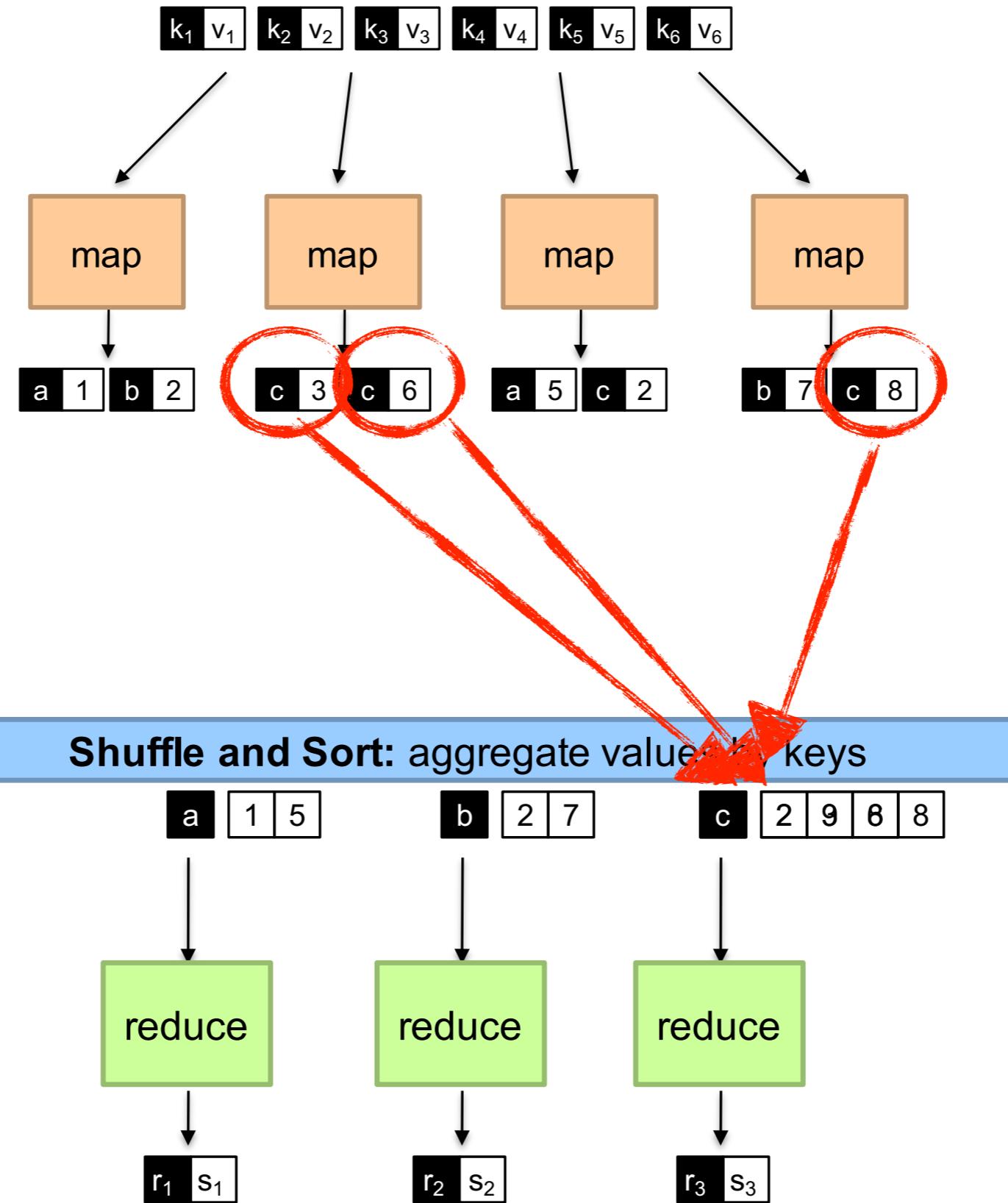
Combiner and partitioner

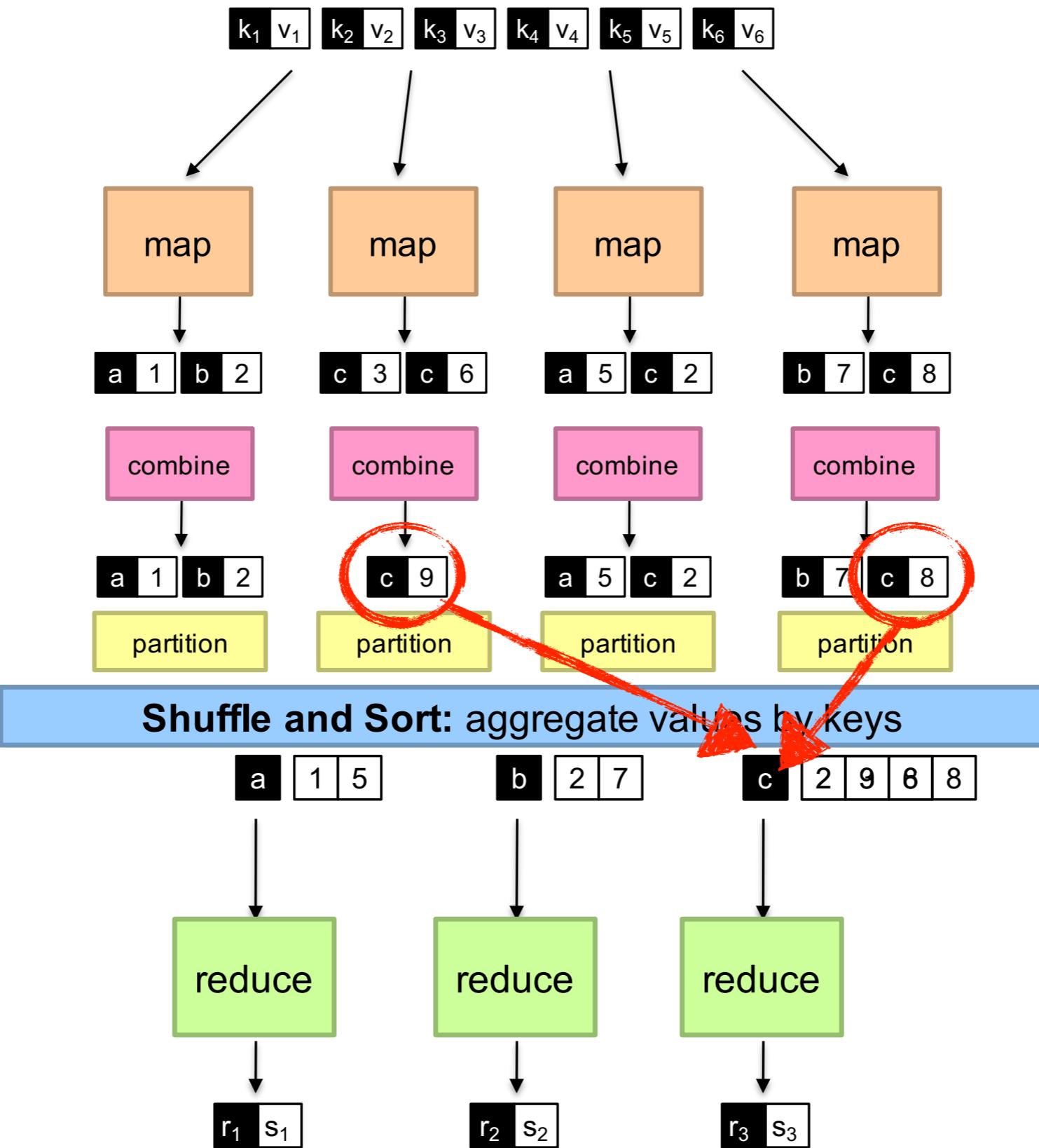
combine ($k, [v]$) $\rightarrow \langle k, v \rangle$

- ▶ mini-reducers that run in memory after the map phase
- ▶ used as an optimization to reduce network traffic

partition ($k, \# \text{ of partitions}$) \rightarrow partition for k

- ▶ divides up key spaces for parallel reduce operations
- ▶ often a simple hash of the key, e.g., $\text{hash}(k) \bmod n$





MapReduce

Programmers specify:

- ▶ **map** (k_1, v_1) \rightarrow [$<k_2, v_2>$]
- ▶ **combine** ($k_2, [v_2]$) \rightarrow $<k_2, v_2>$
- ▶ **partition** ($k_2, \# \text{ of partitions}$) \rightarrow partition for k_2
- ▶ **reduce** ($k_2, [v_2]$) \rightarrow [$<k_3, v_3>$]

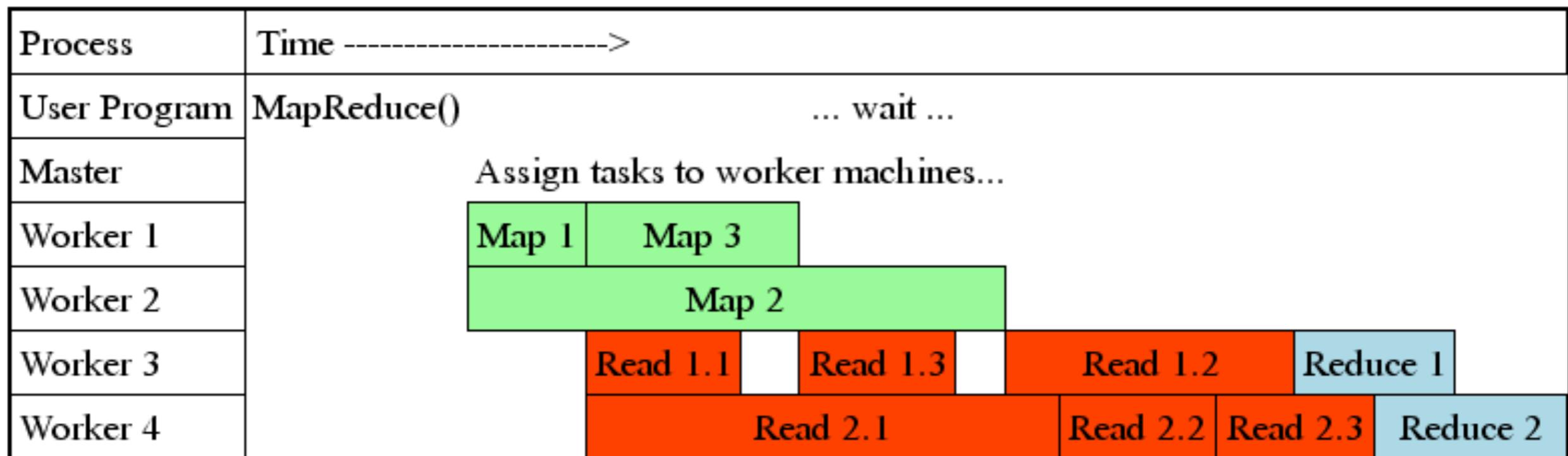
All values with the **same key** are reduced together

The execution framework handles **everything else...**

Two more details...

Barrier between map and reduce phases

- ▶ no reduce can start until map is complete
- ▶ but we can begin transferring intermediate data earlier to **pipeline** shuffling with map execution



Two more details...

Barrier between map and reduce phases

- ▶ no reduce can start until map is complete
- ▶ but we can begin transferring intermediate data earlier to **pipeline** shuffling with map execution

Keys arrive at each reducer in **sorted order**

- ▶ no enforced ordering across reducers

MapReduce can refer to...

The programming model

The execution framework (aka “runtime”)

The specific implementation

Usage is usually clear from context!

MapReduce Implementations

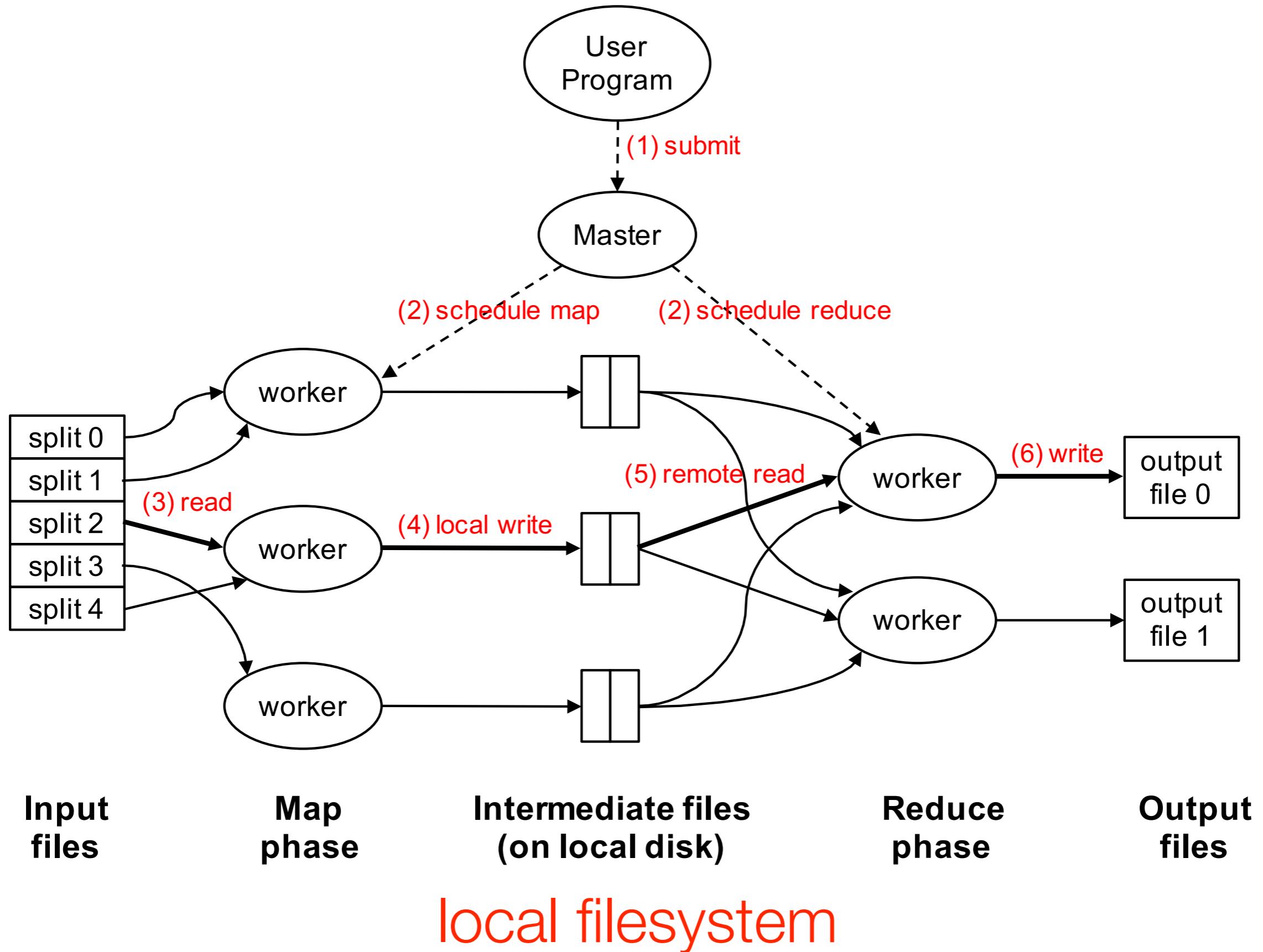
MapReduce implementations

Google has a proprietary implementation in C++

- ▶ bindings in Java, Python
- ▶ master-slave architecture

One of the most influential work in computer systems:

- ▶ J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In USENIX OSDI, 2004.



Credits

Some slides are adapted from Prof. Jimmy Lin's slides at the University of Waterloo