Introduction to MapReduce

Adapted from Jimmy Lin (U. Maryland, USA)

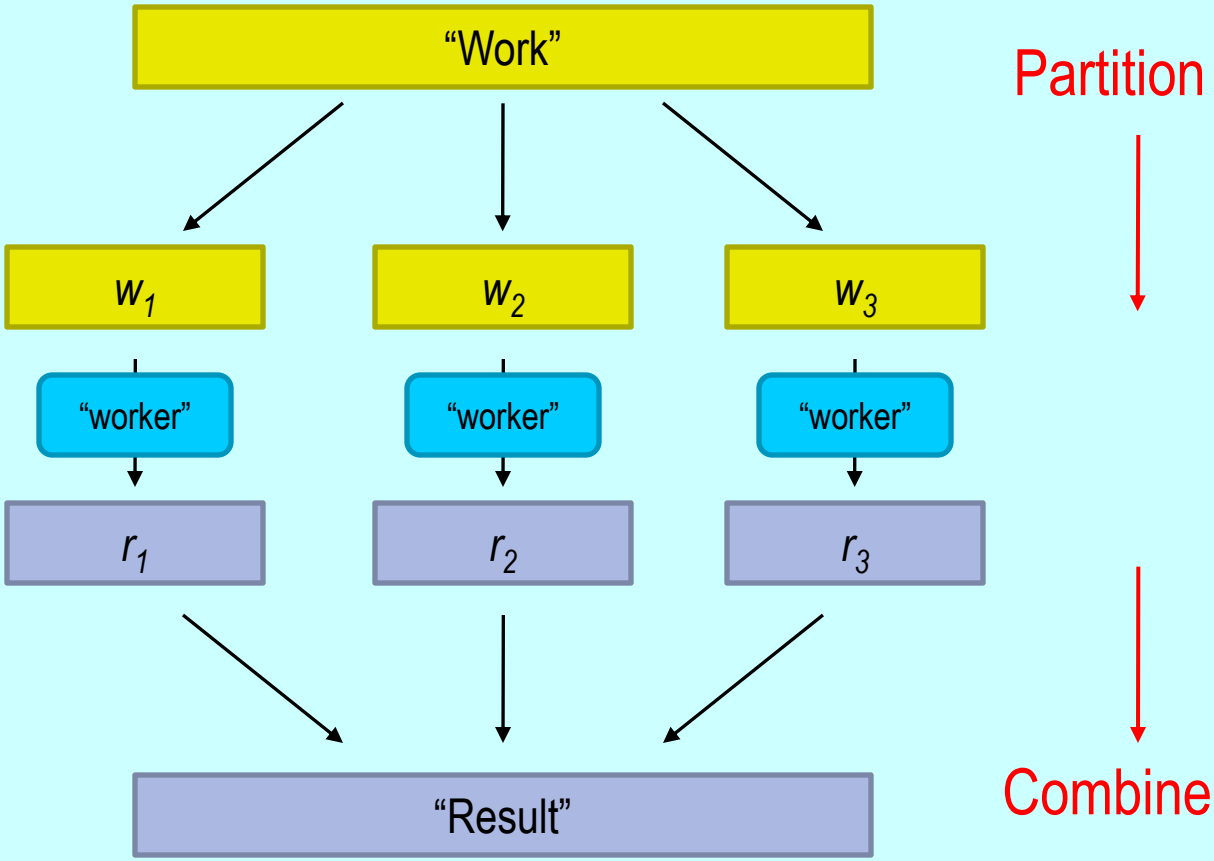
Overview

- Motivation
  - Need for handling “big data”
  - New programming paradigm
- Review of functional programming
  - mapReduce uses this abstraction
- mapReduce details

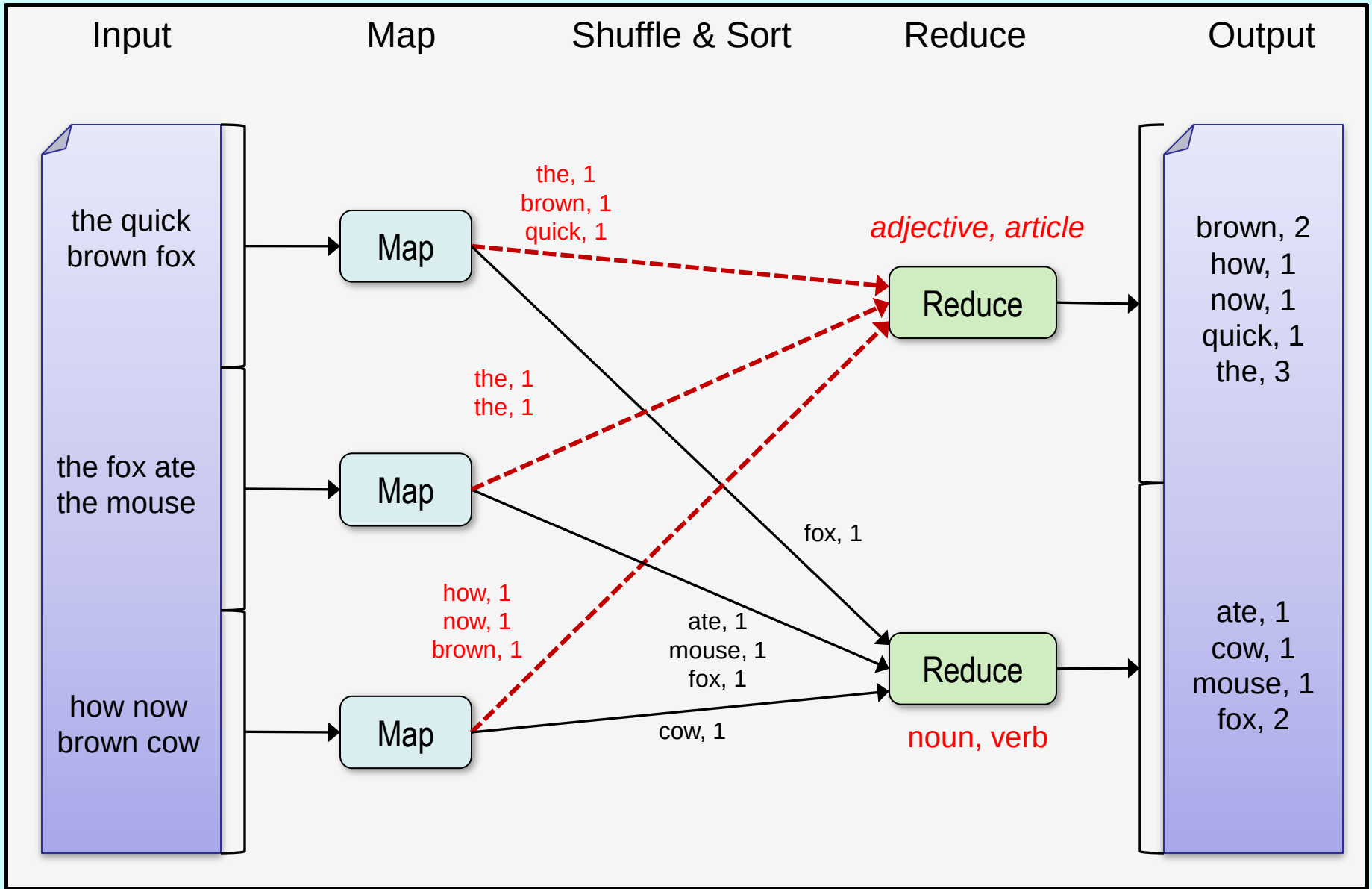
Motivation: Large Scale Data Processing

- Want to process lots of data (> 1 TB)
- Want to parallelize across hundreds/thousands of CPUs
- … Want to make this easy

Divide and Conquer



Word Count Example



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 is the common theme of all of these problems?

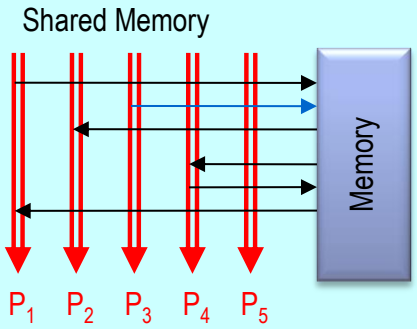
Source: Ricardo Guimarães Herrmann

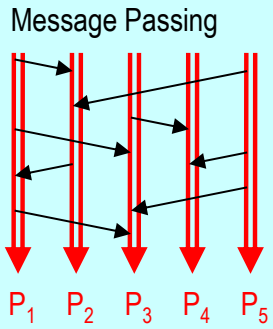
Managing Multiple Workers

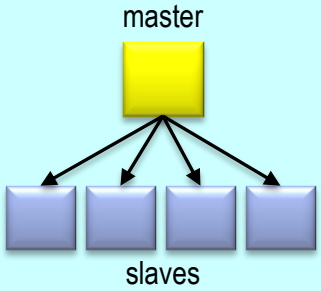
- Difficult because
  - We don't know the order in which workers run
  - We don't know when workers interrupt each other
  - We don't know the order in which workers access shared data
- Thus, we need:
  - Semaphores (lock, unlock)
  - Conditional variables (wait, notify, broadcast)
  - Barriers
- Still, lots of problems:
  - Deadlock, livelock, race conditions...
  - Dining philosophers, sleeping barbers, cigarette smokers...
- Moral of the story: be careful!

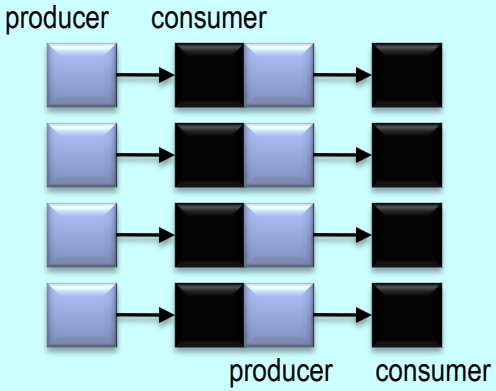
Current Tools

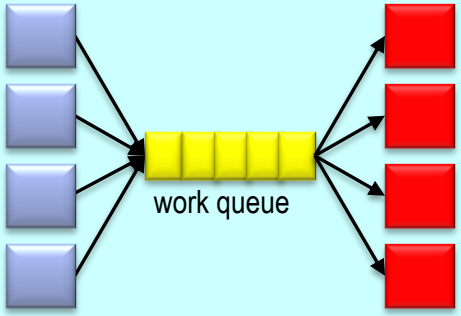
- Programming models
  - Shared memory (pthreads)
  - Message passing (MPI)
- Design Patterns
  - Master-slaves
  - Producer-consumer flows
  - Shared work queues











Concurrency Challenge!

- Concurrency is difficult to reason about
- Concurrency is even more difficult to reason about
  - At the scale of datacenters (even across datacenters)
  - In the presence of failures
  - In terms of multiple interacting services
- Not to mention debugging…
- The reality:
  - Lots of one-off solutions, custom code
  - Write you own dedicated library, then program with it
  - Burden on the programmer to explicitly manage everything

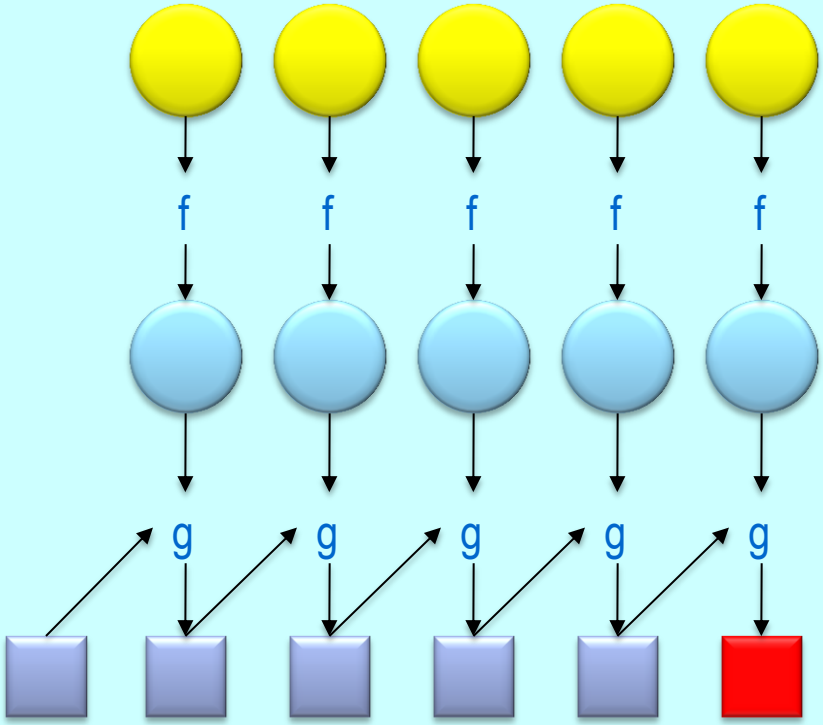
Overview

- Motivation
  - Need for handling “big data”
  - New programming paradigm
- Review of functional programming
  - mapReduce uses this abstraction
- mapReduce details

MapReduce:
Roots in Functional Programming

Map

Reduce



Functional Programming Review

- Functional operations do not modify data structures
  - They always create new ones
- Original data still exists in unmodified form
- Data flows are implicit in program design
- Order of operations does not matter

Functional Programming Review

```
fun foo(l: int list) =
  sum(l) + mul(l) + length(l)
```

Order of sum() and mul(), etc does not matter – they do not modify *l*

“Updates” Don’t Modify Structures

```
fun append(x, lst) =
  let lst' = reverse lst in
    reverse ( x :: lst' )
```

The append() function above reverses a list, adds a new element to the front, and returns all of that, reversed, which appends an item.

But it *never modifies lst*!

Functions Can Be Used As Arguments

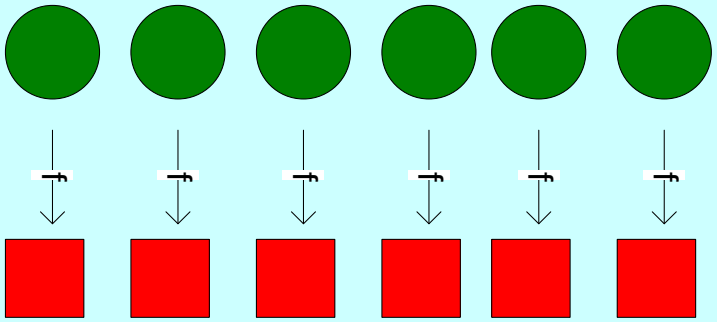
```
fun DoDouble(f, x) = f (f x)
```

It does not matter what f does to its argument; DoDouble() will do it twice.

Map

map f lst: ('a->'b) -> ('a list) -> ('b list)

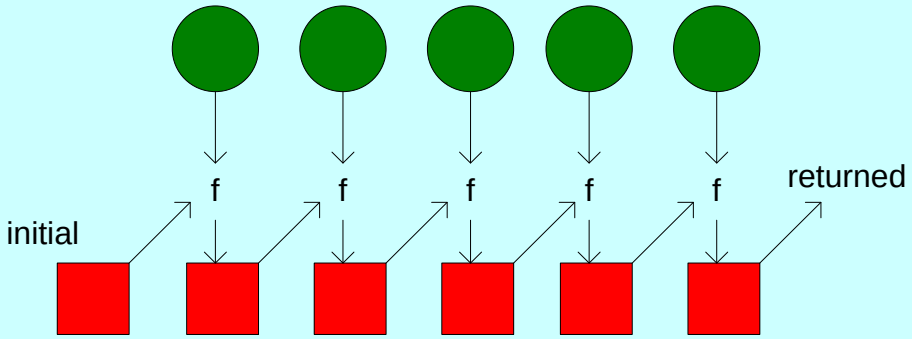
Creates a new list by applying f to each element of the input list; returns output in order.



Reduce (Fold)

fold f x_0 lst: ('a*'b->'b)->'b->('a list)->'b

Moves across a list, applying *f* to each element plus an *accumulator*. f returns the next accumulator value, which is combined with the next element of the list



MapReduce Details

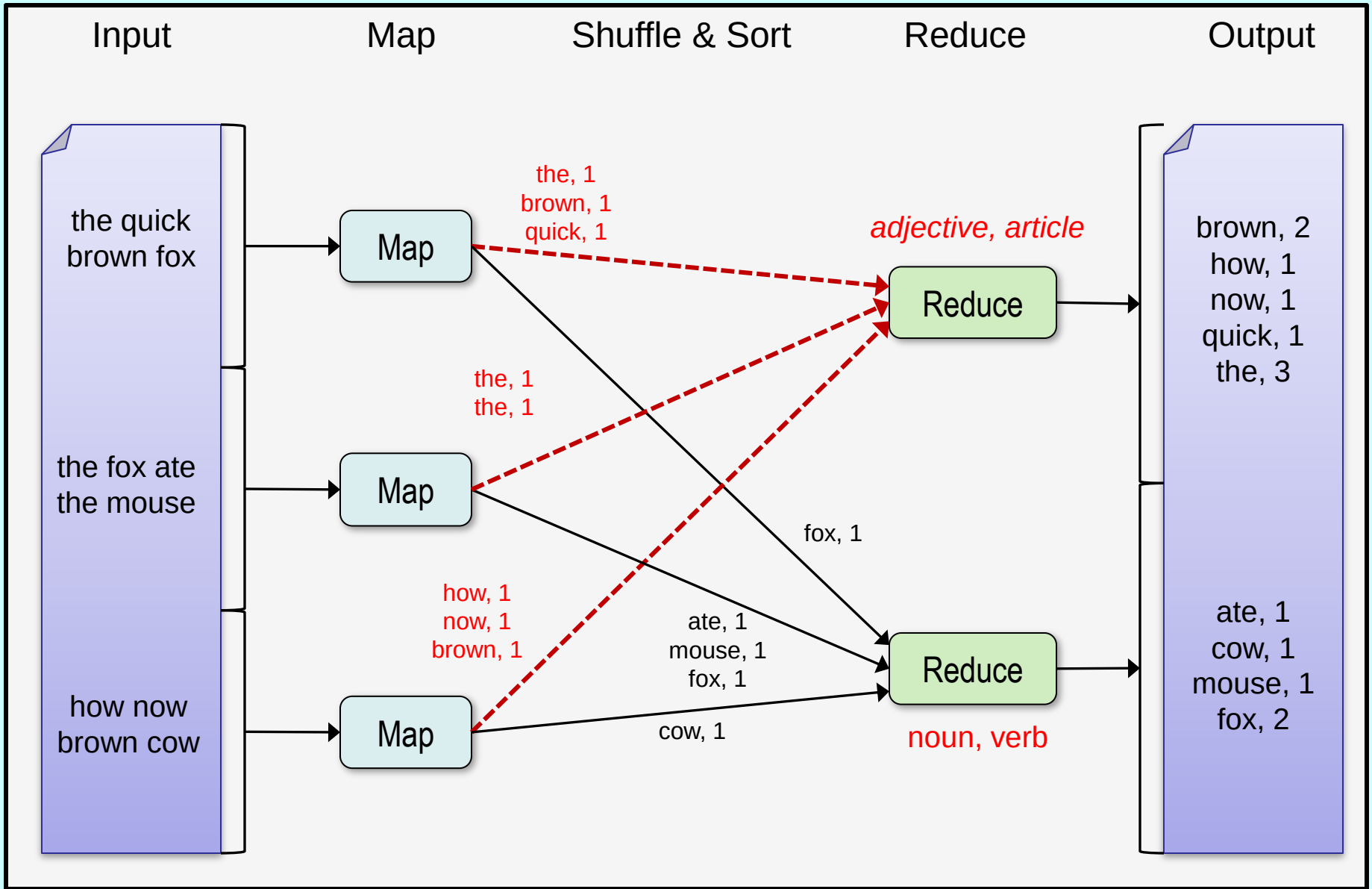
Typical Large-Data Problem

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

Key idea: provide a functional abstraction for these two operations

(Dean and Ghemawat, OSDI 2004)

Word Count Example



Programming Model

- Borrows from functional programming
- Users implement interface of two functions:

  - `map (in_key, in_value) ->`
    `(out_key, intermediate_value) list`

  - `reduce (out_key, intermediate_value list) ->`
    `out_value list`

MapReduce

- Programmers specify two functions:

  map (k, v) → [(k', v')]

  reduce (k', [v']) → [(k', v')]

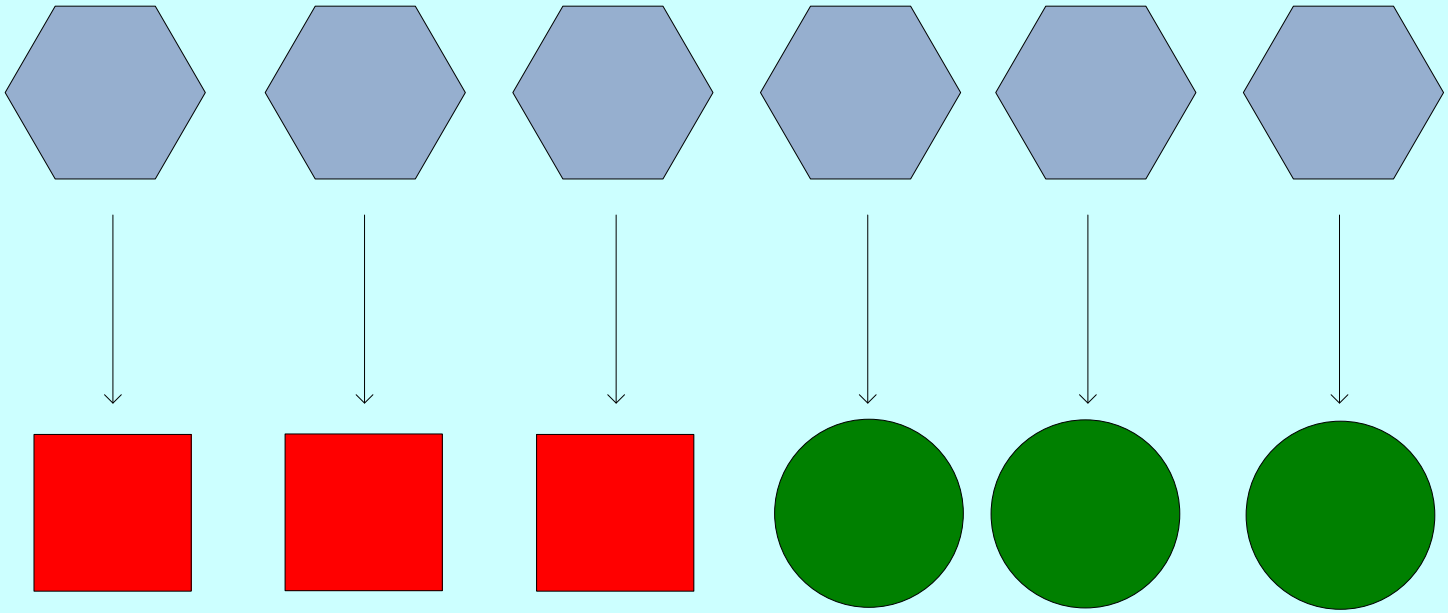
  - All values with the same key are sent to the same reducer
- The execution framework handles everything else…

Map

- Records from the data source (lines out of files, rows of a database, etc) are fed into the map function as key*value pairs: e.g., (filename, line).
- map() produces one or more *intermediate* values along with an output key from the input.

Map

```
map   (in_key, in_value) ->
    (out_key, intermediate_value) list
```

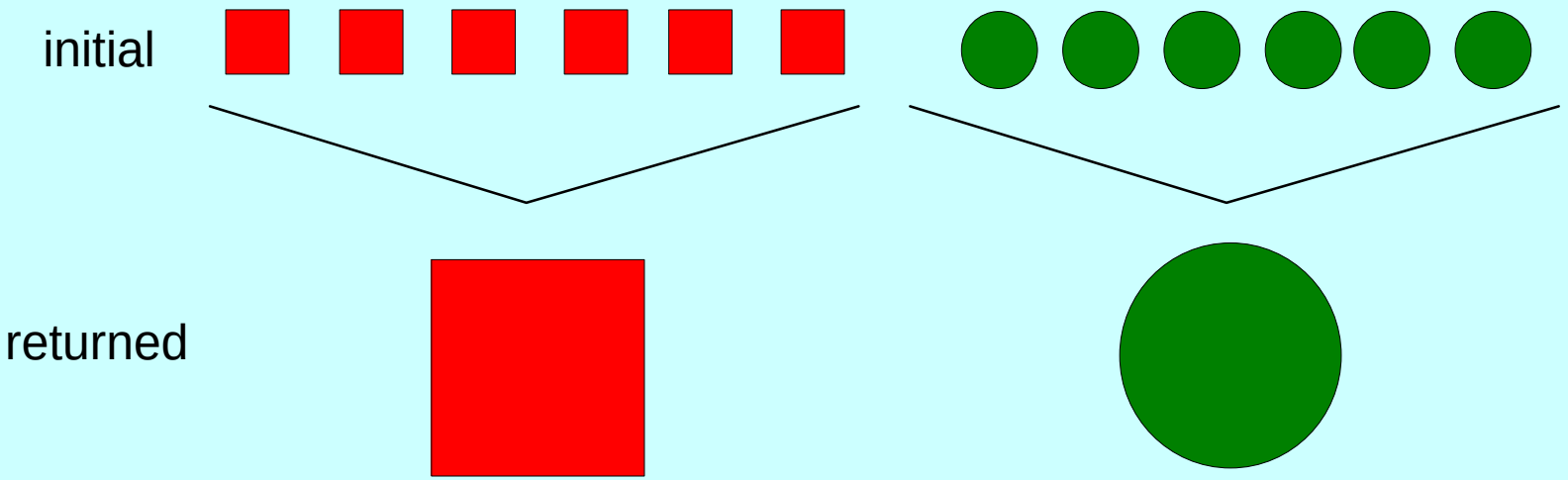


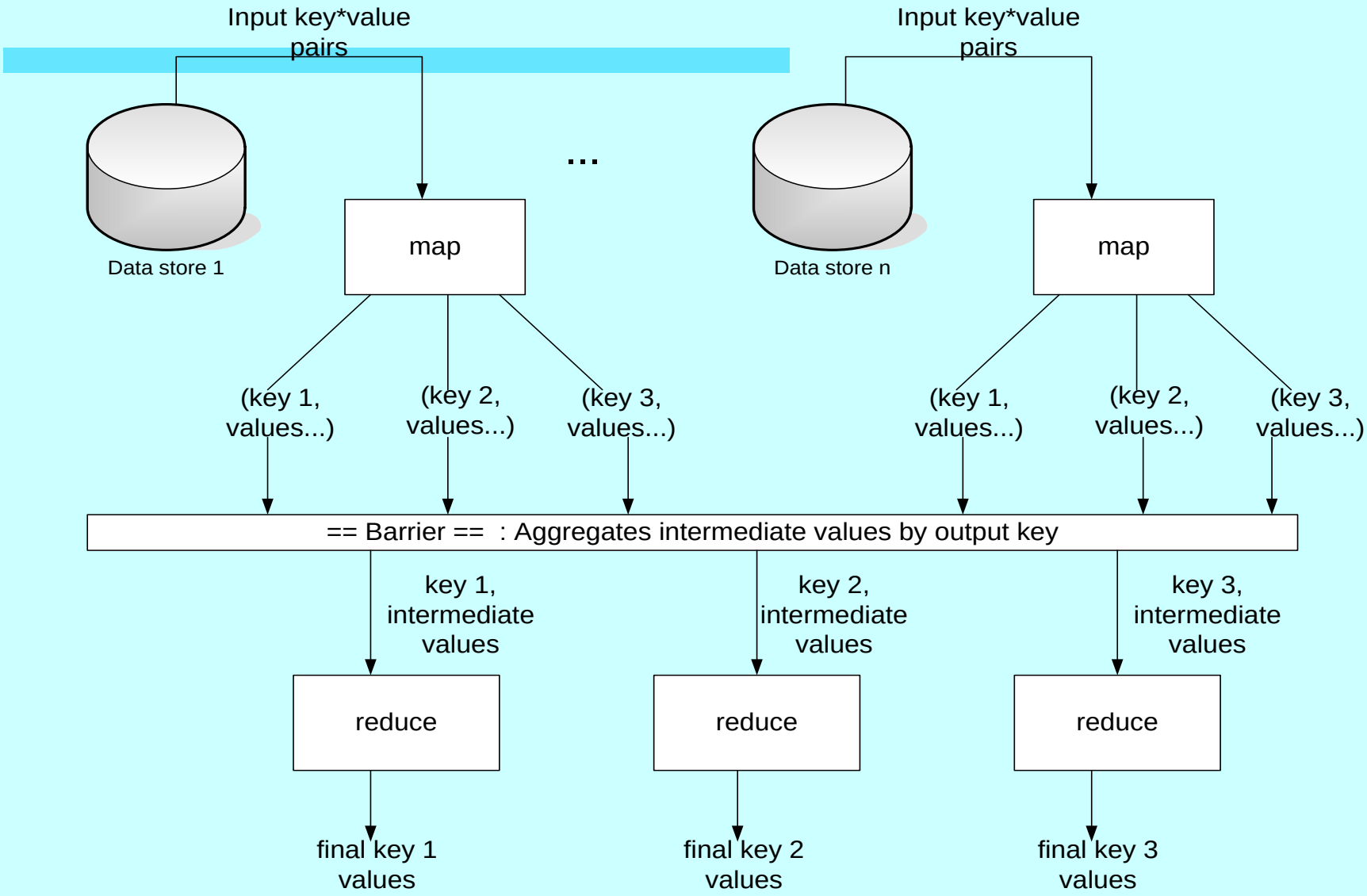
Reduce

- After the map phase is over, all the intermediate values for a given output key are combined together into a list
- reduce() combines those intermediate values into one or more *final values* for that same output key
- (in practice, usually only one final value per key)

Reduce

```
reduce (out_key, intermediate_value list) ->
          out_value list
```




Input key*value pairs
Input key*value pairs
...
Data store 1
Data store n
map
map
(key 1, values...)
(key 2, values...)
(key 3, values...)
(key 1, values...)
(key 2, values...)
(key 3, values...)
== Barrier == : Aggregates intermediate values by output key
key 1, intermediate values
key 2, intermediate values
key 3, intermediate values
reduce
reduce
reduce
final key 1 values
final key 2 values
final key 3 values

Parallelism

- map() functions run in parallel, creating different intermediate values from different input data sets
- reduce() functions also run in parallel, each working on a different output key
- All values are processed *independently*
- Bottleneck: reduce phase can't start until map phase is completely finished.

Example: Count word occurrences

```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
    EmitIntermediate(w, 1);

reduce(String output_key, Iterator<int>
  intermediate_values):
  // output_key: a word
  // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += v;
  Emit(result);
```

Optimizations

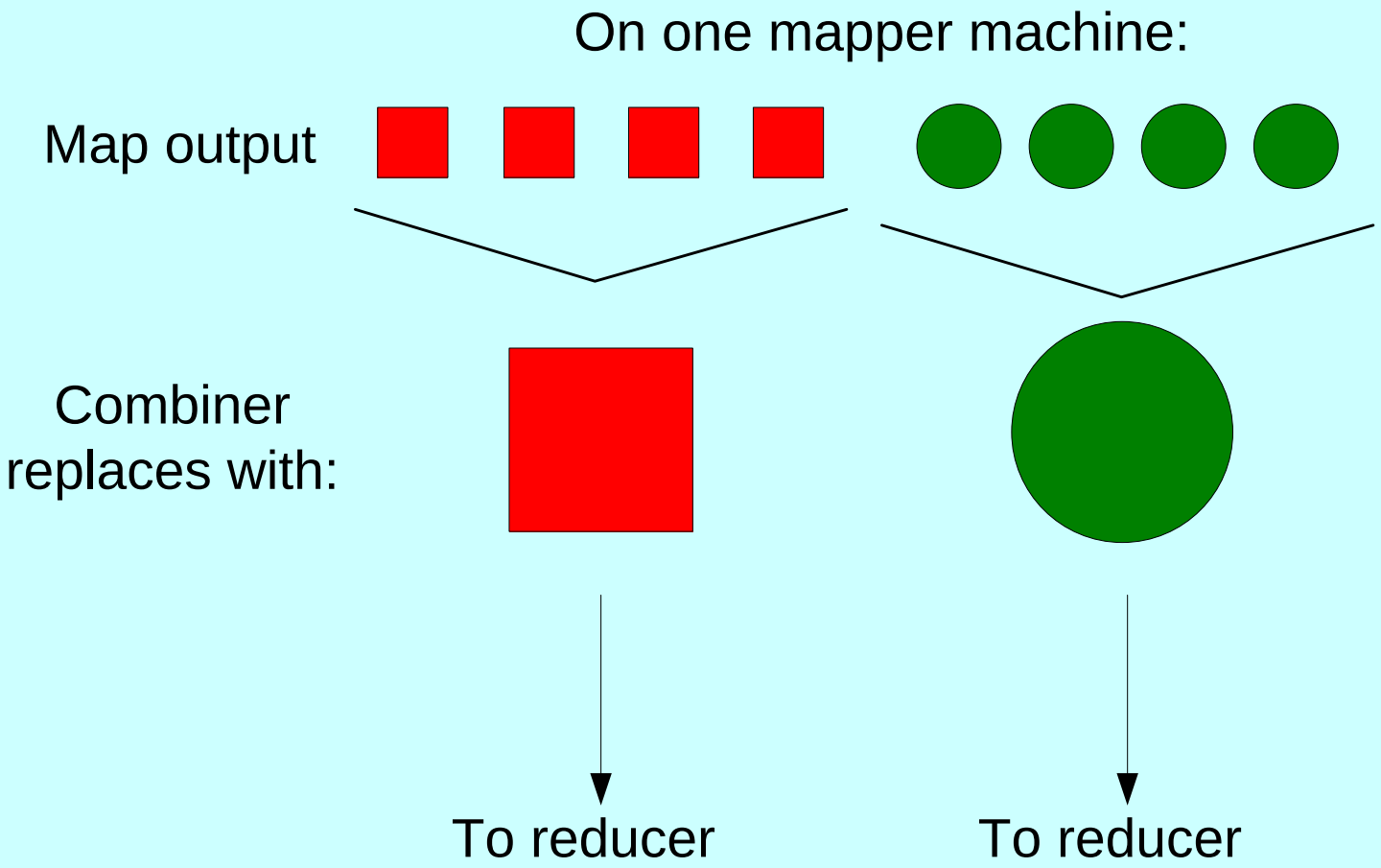
- No reduce can start until map is complete:
  - A single slow disk controller can rate-limit the whole process
- Master redundantly executes "slow-moving" map tasks; uses results of first copy to finish

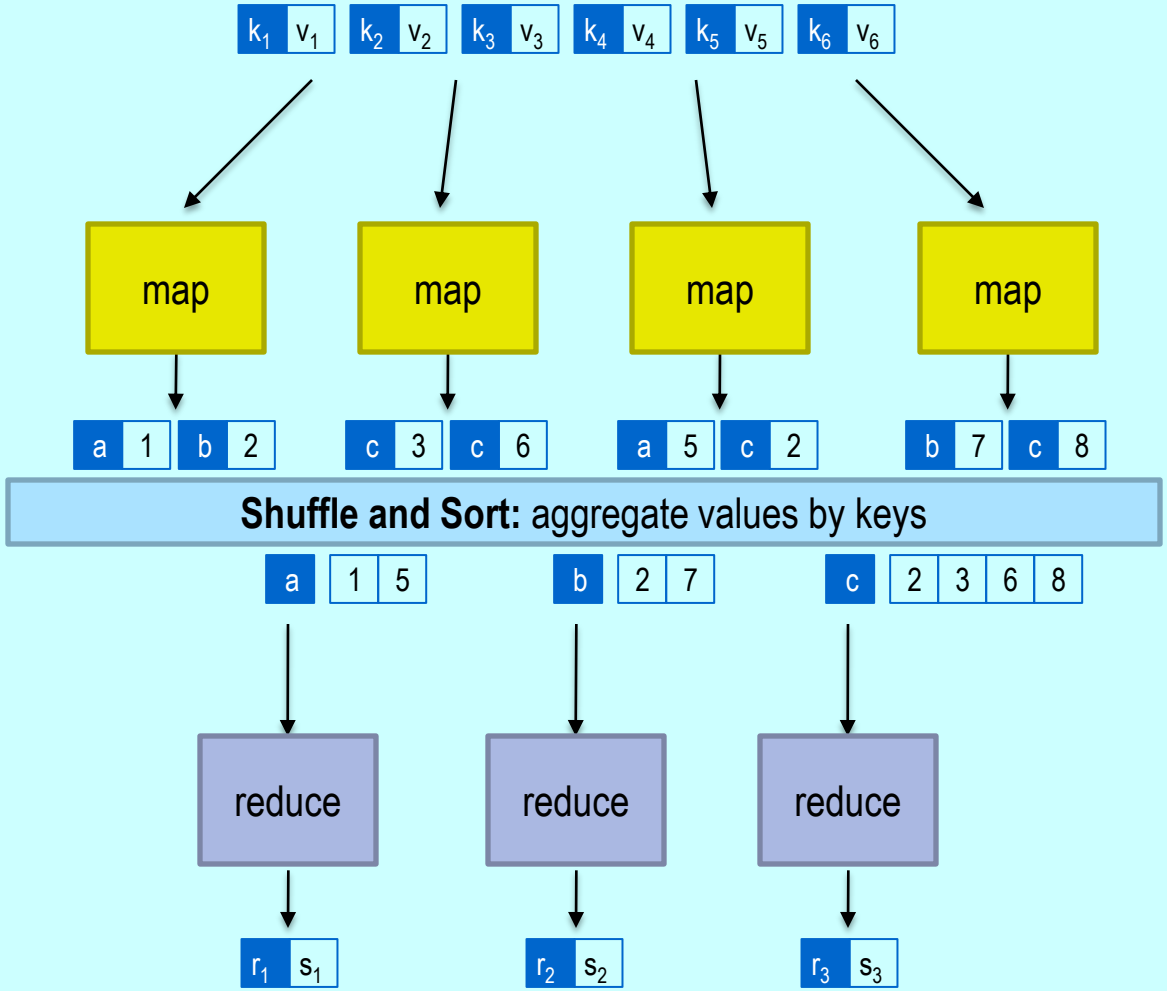
Why is it safe to redundantly execute map tasks? Wouldn't this mess up the total computation?

Combining Phase

- Run on mapper nodes after map phase
- "Mini-reduce," only on local map output
- Used to save bandwidth before sending data to full reducer
- Reducer can be combiner if commutative & associative

Combiner, graphically




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
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

MapReduce

- Programmers specify two functions:

  map (k, v) → <k', v'>*

  reduce (k', v') → <k', v'>*

  - 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
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed FS

MapReduce

- Programmers specify two functions:

  map (k, v) → [(k', v')]

  reduce (k', [v']) → [(k', v')]

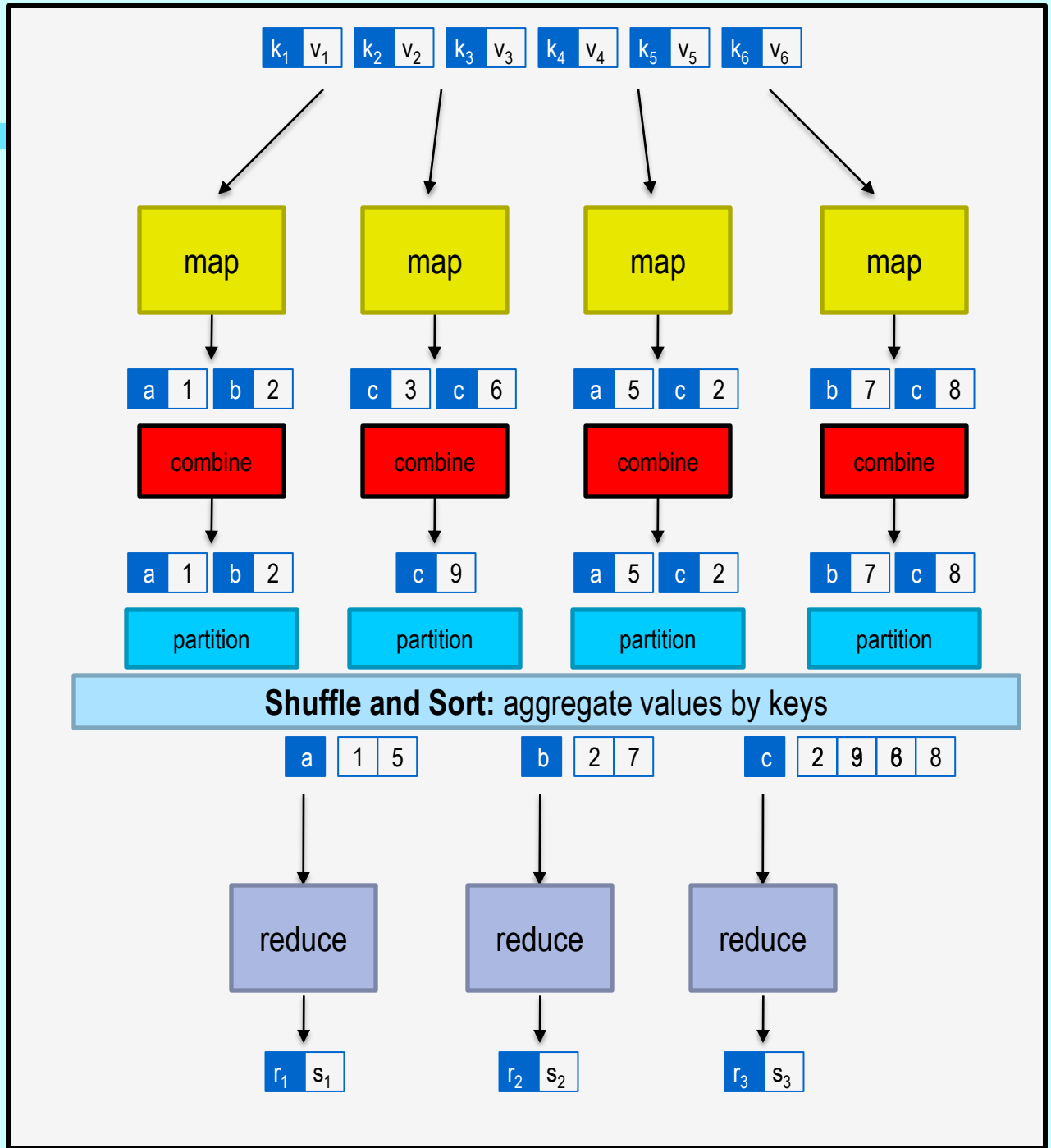
  - All values with the same key are reduced together
- The execution framework handles everything else…
- Not quite…usually, programmers also specify:

  partition (k', number of partitions) → partition for k'

  - Often a simple hash of the key, e.g., hash(k') mod n
  - Divides up key space for parallel reduce operations

  combine (k', [v']) → [(k', v'')]

  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce network traffic


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
combine
combine
combine
combine
a 1
b 2
c 9
a 5
c 2
b 7
c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

Two more details…

- Barrier between map and reduce phases
  - But we can begin copying intermediate data earlier
- 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!

“Hello World”: Word Count

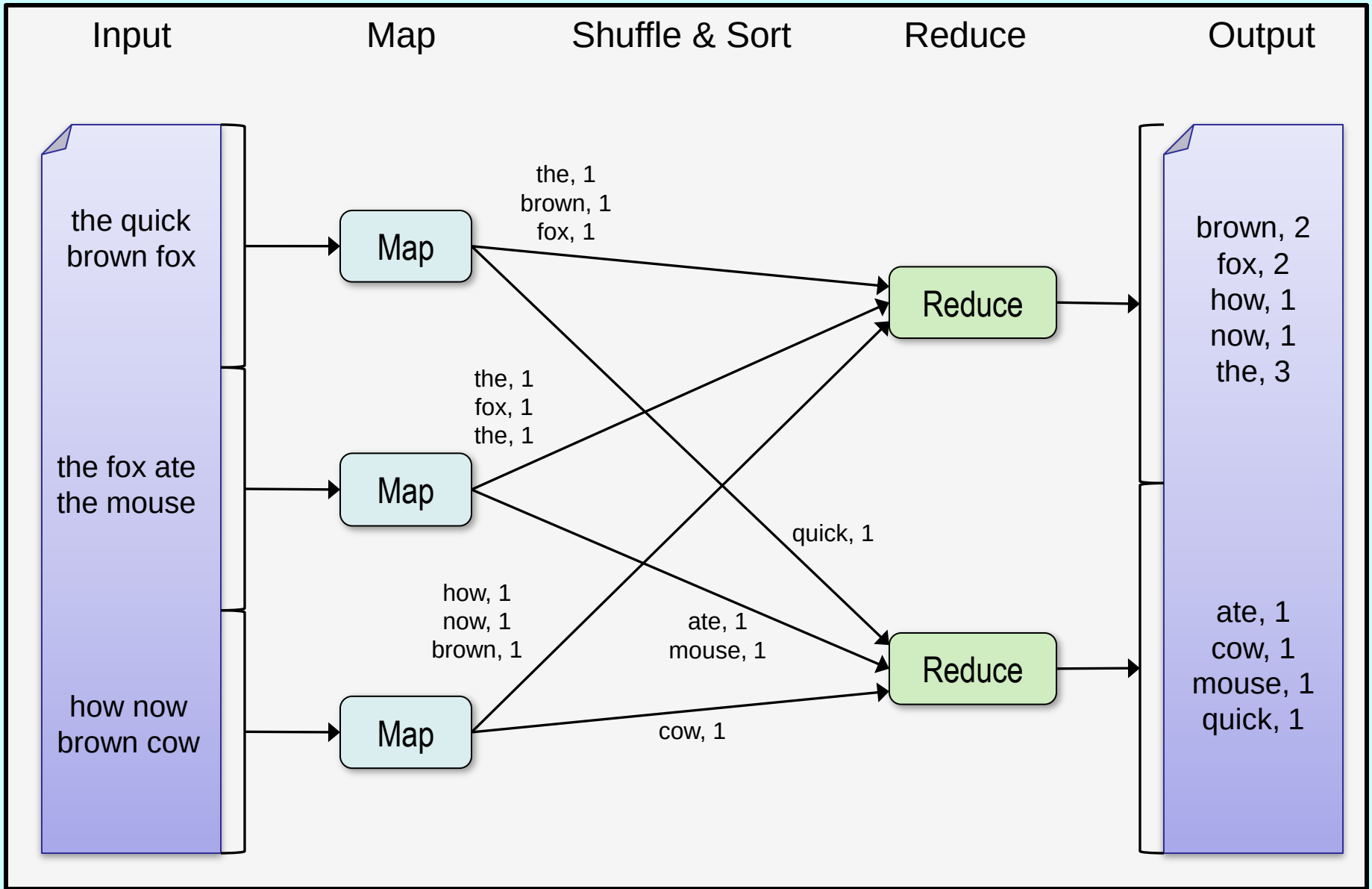
```
Map(String docid, String text):
    for each word w in text:
        Emit(w, 1);

Reduce(String term, Iterator<Int> values):
    int sum = 0;
    for each v in values:
        sum += v;
        Emit(term, value);
```

MapReduce

- Automatic parallelization & distribution
- Fault-tolerant
- Provides status and monitoring tools
- Clean abstraction for programmers

Word Count Execution



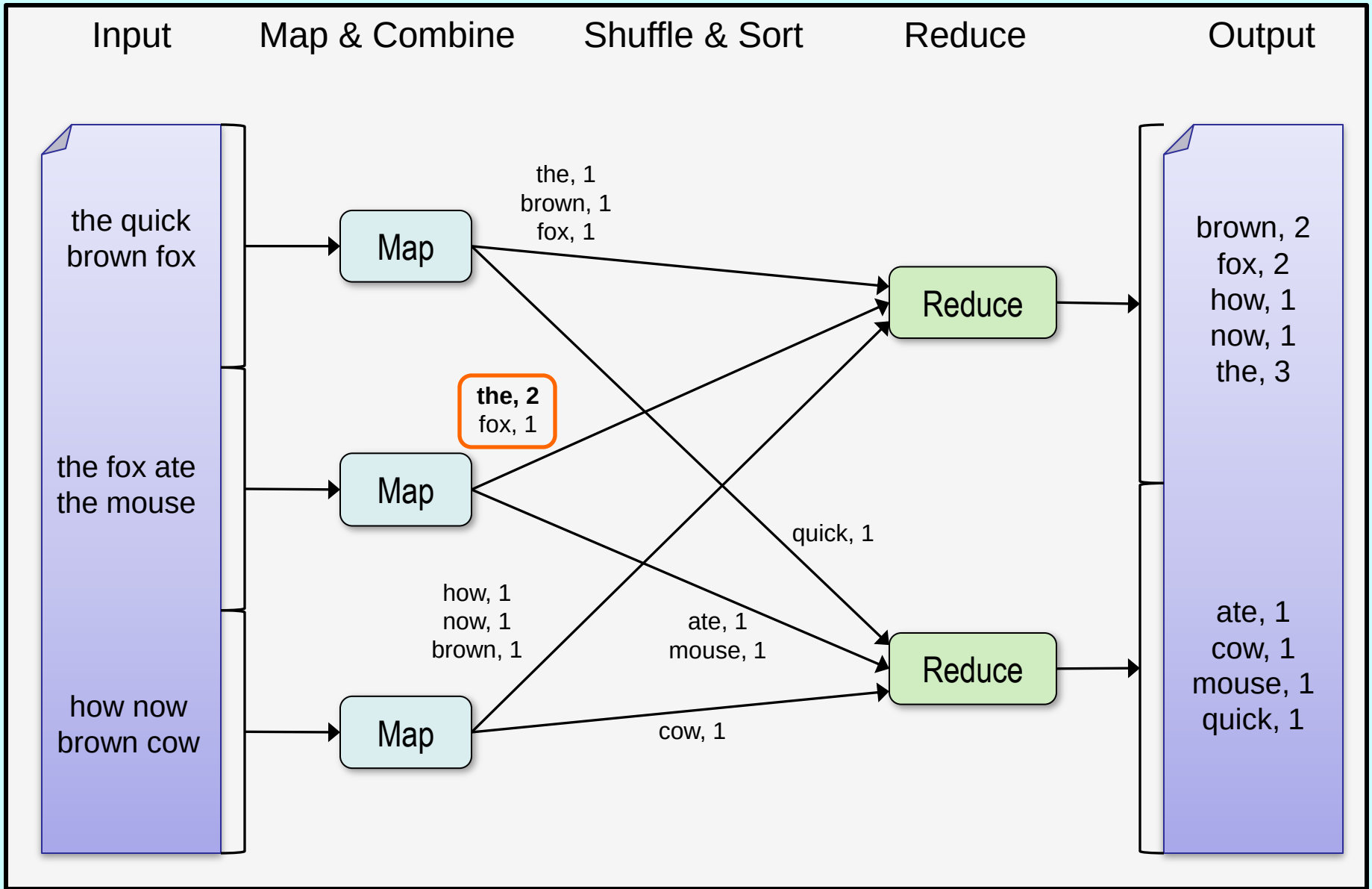
An Optimization: The Combiner

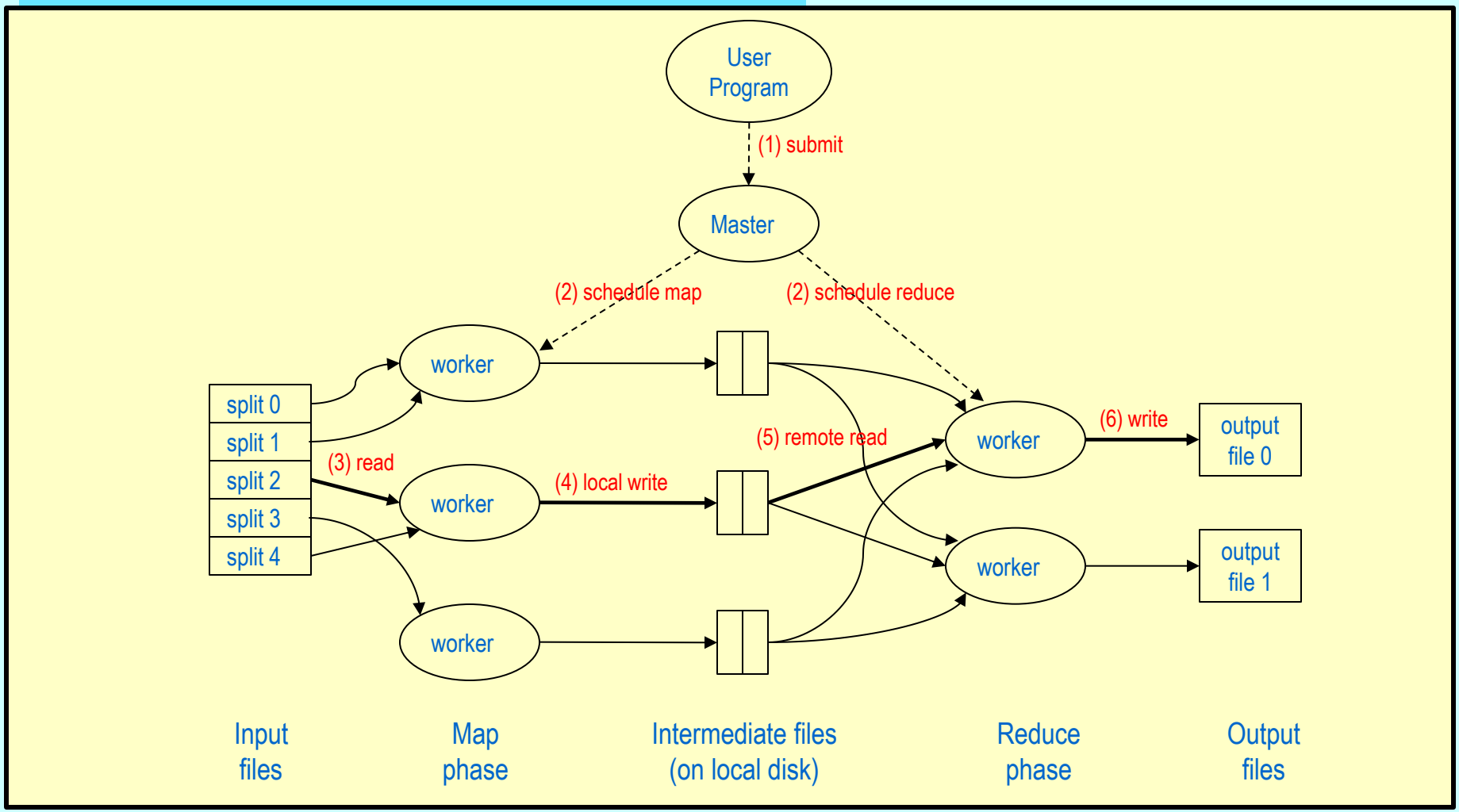
- A combiner is a local aggregation function for repeated keys produced by same map
- For associative ops. like sum, count, max
- Decreases size of intermediate data

- Example: local counting for Word Count:

```
def combiner(key, values):
  output(key, sum(values))
```

Word Count with Combiner




User Program
(1) submit
Master
(2) schedule map
(2) schedule reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disk)
Reduce phase
Output files

MapReduce Conclusions

- MapReduce has proven to be a useful abstraction
- Greatly simplifies large-scale computations at Google
- Functional programming paradigm can be applied to large-scale applications
- Fun to use: focus on problem, let library deal w/ messy details

MapReduce in the Real World

- Implementations
- Who works with this approach

MapReduce Implementations

- Google has a proprietary implementation in C++
  - Bindings in Java, Python
- Hadoop is an open-source implementation in Java
  - Development led by Yahoo, used in production
  - Now an Apache project
  - Rapidly expanding software ecosystem
- Lots of custom research implementations
  - For GPUs, cell processors, etc.

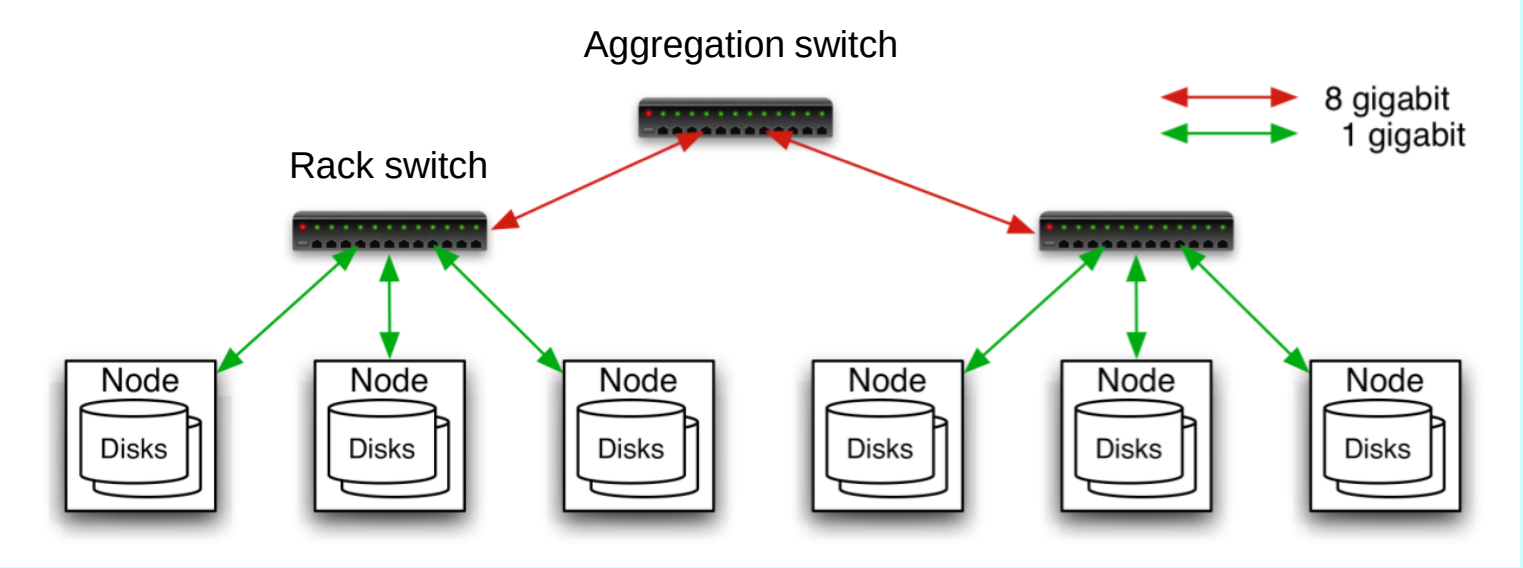
Hadoop History

- Dec 2004 – Google GFS paper published
- July 2005 – Nutch uses MapReduce
- Feb 2006 – Becomes Lucene subproject
- Apr 2007 – Yahoo! on 1000-node cluster
- Jan 2008 – An Apache Top Level Project
- Jul 2008 – A 4000 node test cluster
- Sept 2008 – Hive becomes a Hadoop subproject
- Feb 2009 – The Yahoo! Search Webmap is a Hadoop application that runs on more than 10,000 core Linux cluster and produces data that is now used in every Yahoo! Web search query.
- June 2009 – On June 10, 2009, Yahoo! made available the source code to the version of Hadoop it runs in production.
- In 2010 Facebook claimed that they have the largest Hadoop cluster in the world with 21 PB of storage. On July 27, 2011 they announced the data has grown to 30 PB.

Who uses Hadoop?

- Amazon/A9
- Facebook
- Google
- IBM
- Joost
- Last.fm
- New York Times
- PowerSet
- Veoh
- Yahoo!

Typical Hadoop Cluster



- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 GBps bandwidth in rack, 8 GBps out of rack
- Node specs (Yahoo terasort):
  8 x 2.0 GHz cores, 8 GB RAM, 4 disks (= 4 TB?)

Typical Hadoop Cluster

Image from http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/aw-apachecon-eu-2009.pdf

Example vs. Actual Source Code

- Example is written in pseudo-code
- Actual implementation is in C++, using a MapReduce library
- Bindings for Python and Java exist via interfaces
- True code is somewhat more involved (defines how the input key/values are divided up and accessed, etc.)

Locality

- Master program divvies up tasks based on location of data: tries to have map() tasks on same machine as physical file data, or at least same rack
- map() task inputs are divided into 64 MB blocks: same size as Google File System chunks

Fault Tolerance

- Master detects worker failures
  - Re-executes completed & in-progress map() tasks
  - Re-executes in-progress reduce() tasks
- Master notices particular input key/values cause crashes in map(), and skips those values on re-execution.
  - Effect: Can work around bugs in third-party libraries!