The MapReduce Paradigm

Michael Kleber

with most slides shamelessly stolen from Jeff Dean and Yonatan Zunger

Google, Inc.

Jan. 14, 2008

Do We Need It?

If distributed computing is so hard, do we really need to do it?

Yes: Otherwise some problems are too big.

Example: 20+ billion web pages x 20KB = 400+ terabytes

- One computer can read 30-35 MB/sec from disk

  ~four months to read the web

- ~1,000 hard drives just to store the web
- Even more to *do* something with the data

Yes, We Do

Good news: same problem with 1000 machines, < 3 hours

Bad news: programming work

- communication and coordination
- recovering from machine failure (all the time!)
- status reporting
- debugging
- optimization
- locality

Bad news II: repeat for every problem you want to solve

How can we make this easier?

MapReduce

A simple programming model that applies to many large-scale computing problems

Hide messy details in MapReduce runtime library:

- automatic parallelization
- load balancing
- network and disk transfer optimization
- handling of machine failures
- robustness
- **improvements to core library benefit all users of library!**

Typical problem solved by MapReduce

Read a lot of data

Map: extract something you care about from each record

Shuffle and Sort

Reduce: aggregate, summarize, filter, or transform

Write the results

Outline stays the same,

Map and Reduce change to fit the problem

MapReduce Paradigm

Basic data type: the key-value pair (k,v).

For example, key = URL, value = HTML of the web page.

Programmer specifies two primary methods:

- Map: $(k, v) \mapsto <(k_1,v_1), (k_2,v_2), (k_3,v_3),\ldots,(k_n,v_n)>$
- Reduce: $(k', <v'_1, v'_2,\ldots,v'_{n'}>) \mapsto <(k', v''_1), (k', v''_2),\ldots,(k', v''_{n''})>$

All v' with same k' are reduced together.

(Remember the invisible “Shuffle and Sort” step.)

Example: Word Frequencies in Web Pages

A typical exercise for a new Google engineer in his or her first week

Input: files with one document per record

Specify a *map* function that takes a key/value pair
key = document URL
value = document contents

Output of map function is (potentially many) key/value pairs.
In our case, output (word, “1”) once per word in the document

“document1”, “to be or not to be”

↓

“to”, “1”
“be”, “1”
“or”, “1”
…

Example: Word Frequencies in Web Pages

MapReduce library gathers together all pairs with the same key (shuffle/sort)

Specify a *reduce* function that combines the values for a key
In our case, compute the sum

key	values	reduce output
key = "be"	values = "1", "1"	"2"
key = "not"	values = "1"	"1"
key = "or"	values = "1"	"1"
key = "to"	values = "1", "1"	"2"

Output of reduce (usually 0 or 1 value) paired with key and saved

"be", "2"
"not", "1"
"or", "1"
"to", "2"

Google™

Under the hood: Scheduling

One master, many workers

- Input data split into *M* map tasks (typically 64 MB in size)
- Reduce phase partitioned into *R* reduce tasks (= # of output files)
- Tasks are assigned to workers dynamically
- Reasonable numbers inside Google: *M*=200,000; *R*=4,000; workers=2,000

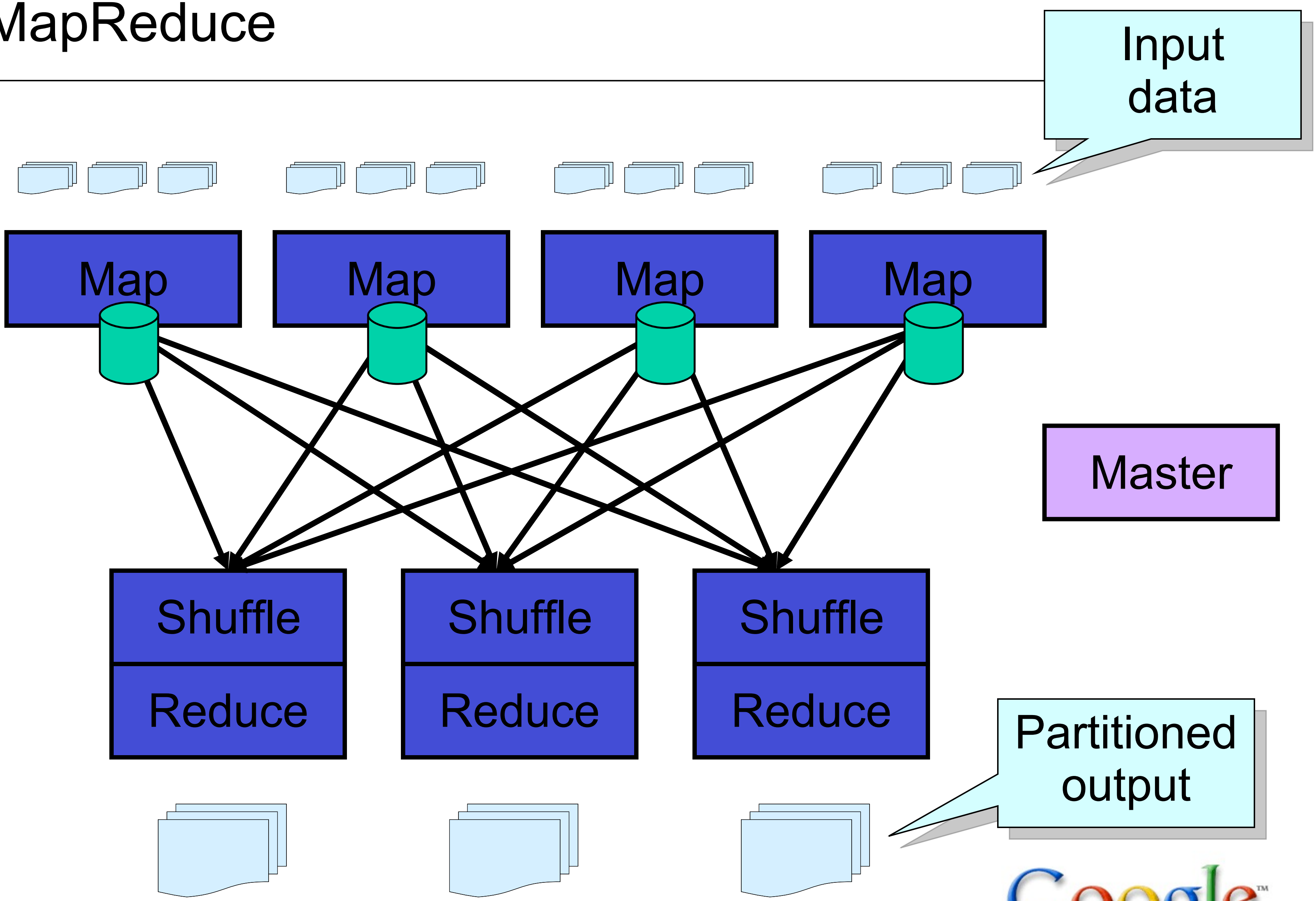
Master assigns each map task to a free worker

- Considers locality of data to worker when assigning task
- Worker reads task input (often from local disk!)
- Worker produces R **local files** containing intermediate (k,v) pairs

Master assigns each reduce task to a free worker

- Worker reads intermediate (k,v) pairs from map workers
- Worker sorts & applies user's Reduce op to produce the output
- User may specify Partition: which intermediate keys to which Reducers

MapReduce

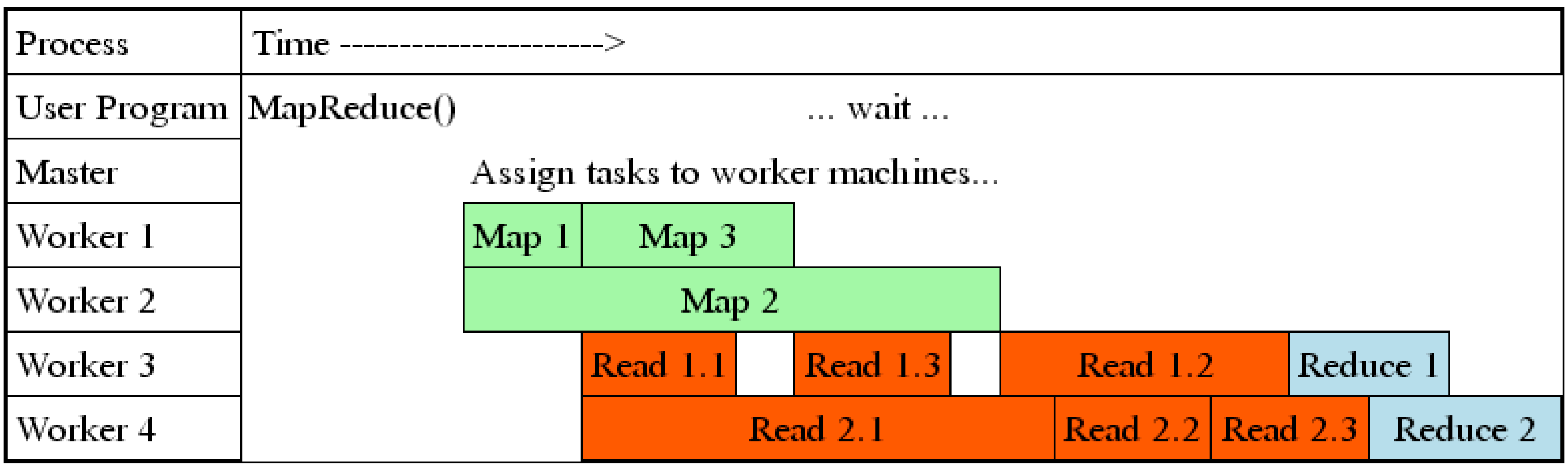


Input data
Map
Map
Map
Map
Master
Shuffle
Reduce
Shuffle
Reduce
Shuffle
Reduce
Partitioned output

MapReduce: Granularity

Fine granularity tasks: many more map tasks than machines

- Minimizes time for fault recovery
- Can pipeline shuffling with map execution
- Better dynamic load balancing



MapReduce: Fault Tolerance via Re-Execution

Worker failure:

- Detect failure via periodic heartbeats
- Re-execute completed and in-progress map tasks
- Re-execute in-progress reduce tasks
- Task completion committed through master

Master failure:

- State is checkpointed to replicated file system
- New master recovers & continues

Very Robust: lost 1600 of 1800 machines once, but finished fine

MapReduce: A Leaky Abstraction

MR insulates you from many concerns, but not all of them.

- Don't overload one reducer
- Don't leak memory, even a little!
- Static and global variables probably don't do what you expect

  (They can sometimes be useful, though!)
- Mappers might get rerun -- maybe on different data!

  Careful with side-effects: must be atomic, idempotent.

  Different reducers might see different versions!