Distributed Systems

18. MapReduce

Paul Krzyzanowski

Rutgers University

Fall 2018

Credit

Much of this information is from Google:

- Google Code University [no longer supported]
  http://code.google.com/edu/parallel/mapreduce-tutorial.html
- MapReduce: The programming model and practice
  research.google.com/pubs/pub36249.html

See also: http://hadoop.apache.org/common/docs/current/

for the Apache Hadoop version

Read this (the definitive paper):

http://labs.google.com/papers/mapreduce.html

Background

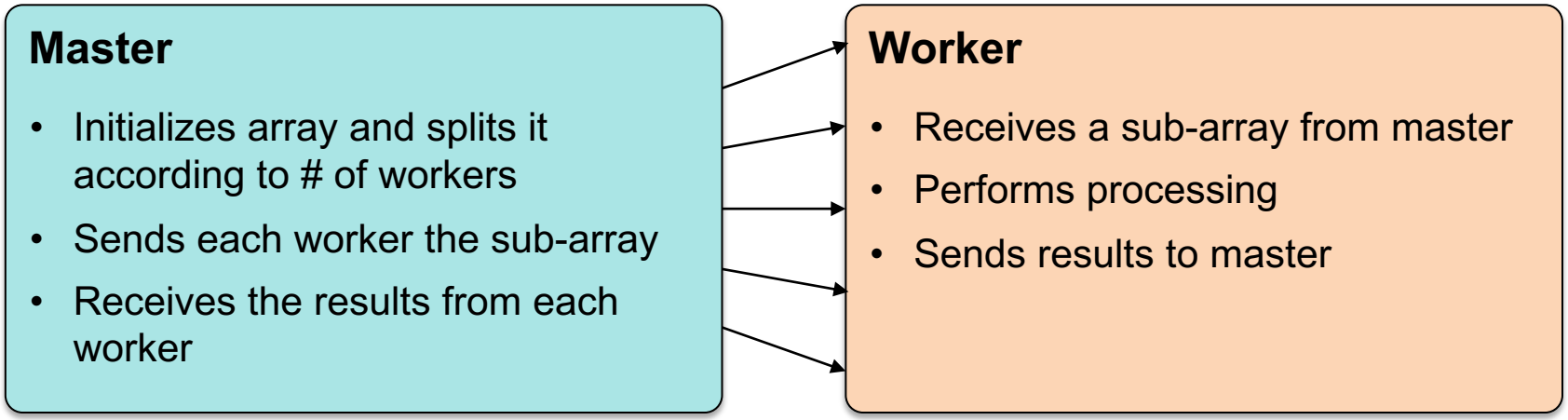
- Traditional programming is serial

- Parallel programming
  - Break processing into parts that can be executed concurrently on multiple processors

- Challenge
  - Identify tasks that can run concurrently
    and/or groups of data that can be processed concurrently
  - Not all problems can be parallelized

Simplest environment for parallel processing

- No dependency among data
- Data can be split into lots of smaller chunks - **shards**
- Each process can work on a chunk
- Master/worker approach



MapReduce

- Created by Google in 2004
  - Jeffrey Dean and Sanjay Ghemawat

- Inspired by LISP
  - Map(function, set of values)
    - Applies function to each value in the set

      (map 'length '(() (a) (a b) (a b c))) ⇒ (0 1 2 3)
  - Reduce(function, set of values)
    - Combines all the values using a binary function (e.g., +)

      (reduce #'+ '(1 2 3 4 5)) ⇒ 15

MapReduce

- MapReduce
  - Framework for parallel computing
  - Programmers get simple API
  - Don’t have to worry about handling
    - parallelization
    - data distribution
    - load balancing
    - fault tolerance

- Allows one to process huge amounts of data (terabytes and petabytes) on thousands of processors

Who has it?

- Google
  - Original proprietary implementation
- Apache Hadoop MapReduce
  - Most common (open-source) implementation
  - Built to specs defined by Google
- Amazon Elastic MapReduce
  - Uses Hadoop MapReduce running on Amazon EC2
    … or Microsoft Azure HDInsight
    … or Google Cloud MapReduce for App Engine

MapReduce

- Map
  - Grab the relevant data from the source
  - User function gets called for each chunk of input
  - Spits out (key, value) pairs
- Reduce
  - Aggregate the results
  - User function gets called for each unique key with all values corresponding to that key

MapReduce

- Map: (input shard) → intermediate(key/value pairs)
  - Automatically partition input data into *M* shards
  - Discard unnecessary data and generate (key, value) sets
  - Framework groups together all intermediate values with the same intermediate key & pass them to the *Reduce* function
- Reduce: intermediate(key/value pairs) → result files
  - Input: key & set of values
  - Merge these values together to form a smaller set of values

  Reduce workers are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., *hash(key) mod R*)

  The user specifies the # of partitions (R) and the partitioning function

MapReduce: what happens in between?

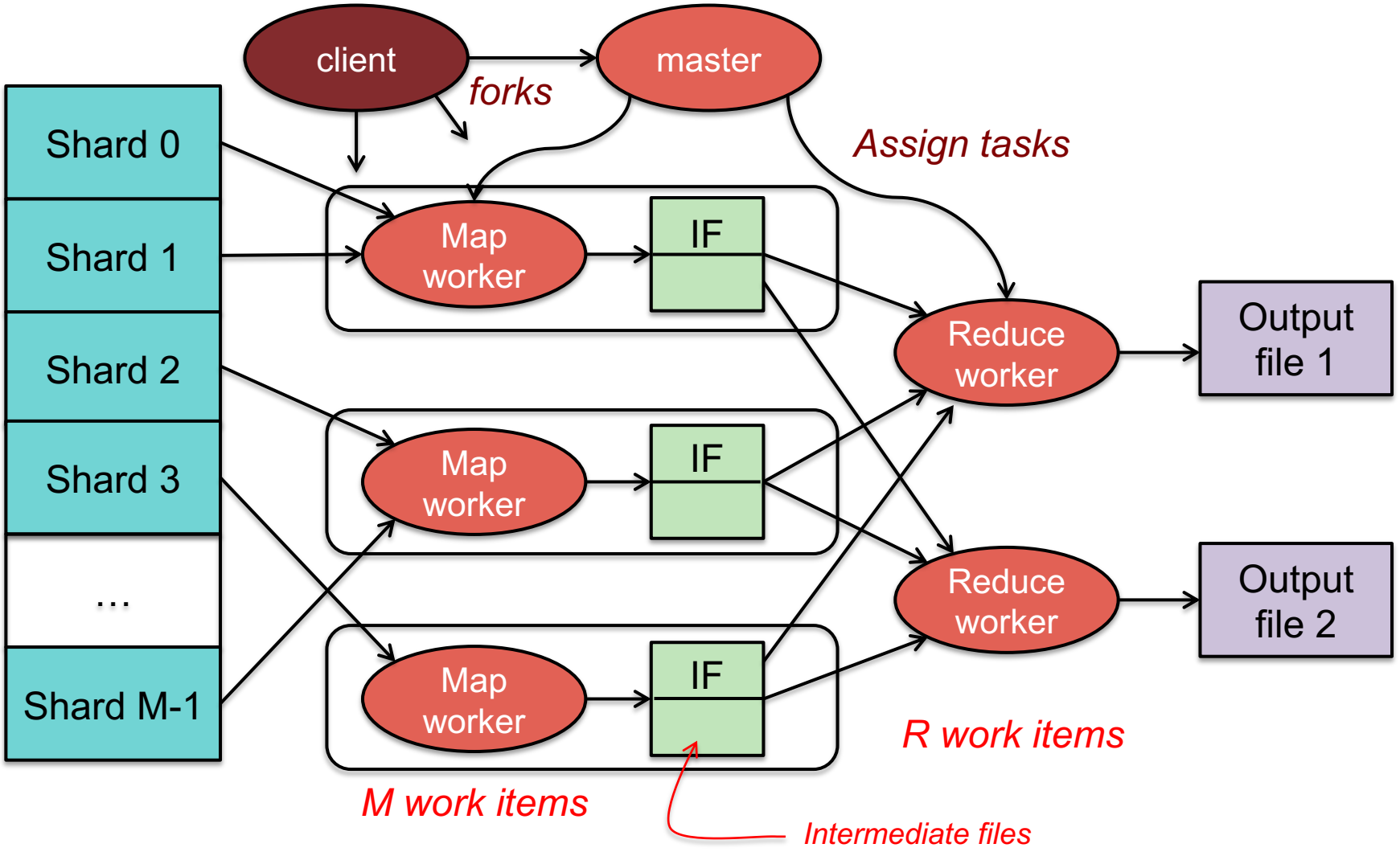
- Map
  - Grab the relevant data from the source (parse into key, value)
  - Write it to an intermediate file
- Partition
  - Partitioning: identify which of *R* reducers will handle which keys
  - Map partitions data to target it to one of *R* Reduce workers based on a partitioning function (both *R* and partitioning function user defined)

Map Worker

- Shuffle & Sort
  - Shuffle: Fetch the relevant partition of the output from all mappers
  - Sort by keys (different mappers may have sent data with the same key)
- Reduce
  - Input is the sorted output of mappers
  - Call the user *Reduce* function per key with the list of values for that key to aggregate the results

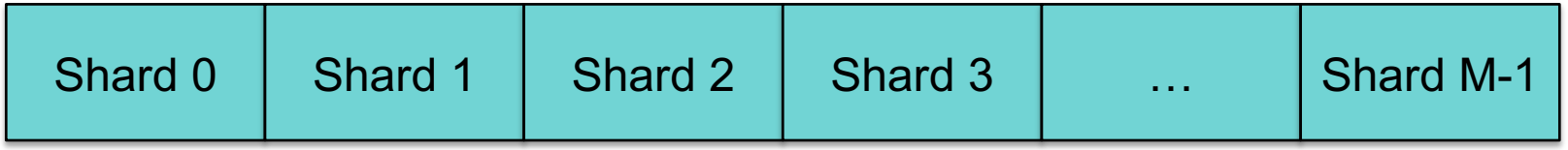
Reduce Worker

MapReduce: the complete picture



Step 1: Split input files into chunks (shards)

- Break up the input data into *M* pieces (typically 64 MB)

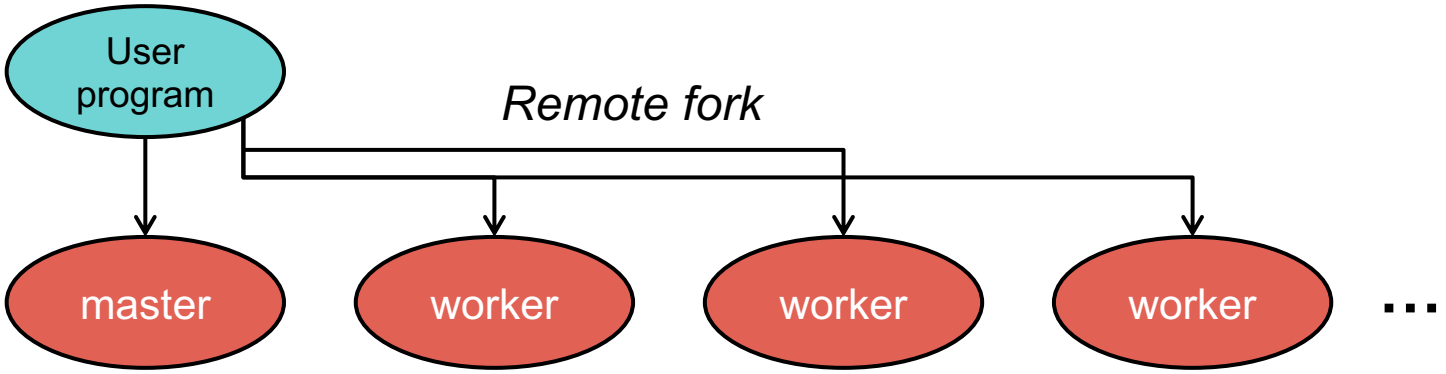


Input files

Divided into *M* shards

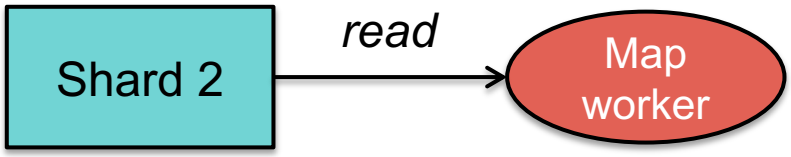
Step 2: Fork processes

- Start up many copies of the program on a cluster of machines
  - **One master**: scheduler & coordinator
  - Lots of workers
- Idle workers are assigned either:
  - map tasks (each works on a shard) – there are *M* map tasks
  - reduce tasks (each works on intermediate files) – there are *R*
    - *R* = # partitions, defined by the user



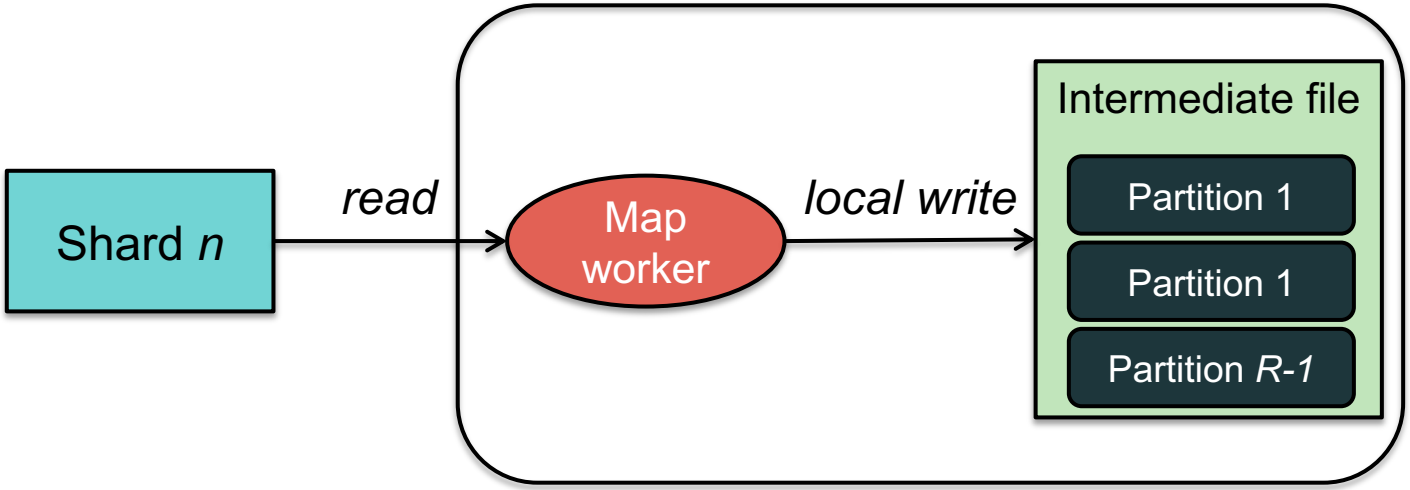
Step 3: Run Map Tasks

- Reads contents of the input shard assigned to it
- Parses key/value pairs out of the input data
- Passes each pair to a user-defined *map* function
  - Produces intermediate key/value pairs
  - These are buffered in memory



Step 4: Create intermediate files

- Intermediate key/value pairs produced by the user's *map* function buffered in memory and are periodically written to the local disk
  - Partitioned into *R* regions by a partitioning function

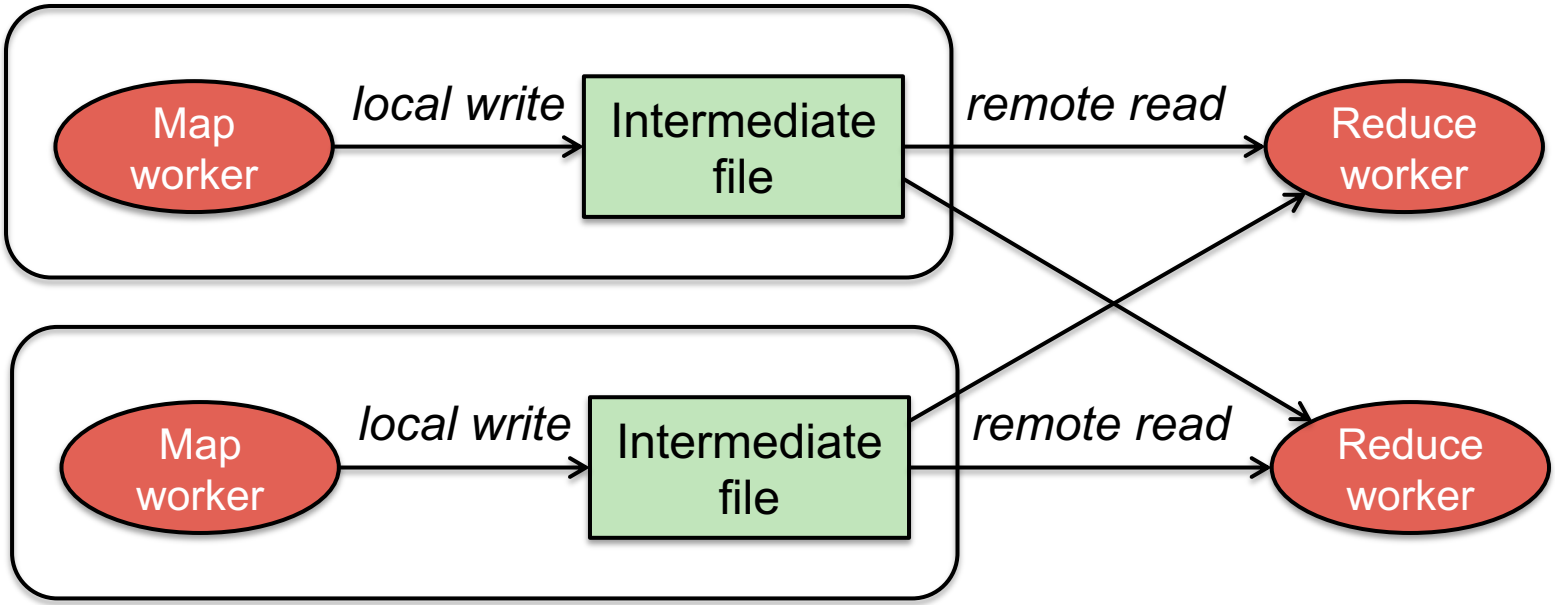


Step 4a. Partitioning

- Map data will be processed by Reduce workers
  - User's *Reduce* function will be called once per unique key generated by *Map*.
- We first need to sort all the (*key, value*) data by keys and decide which Reduce worker processes which keys
  - The Reduce worker will do the sorting
- **Partition function**
  Decides which of *R* reduce workers will work on which key
  - Default function: *hash(key) mod R*
  - Map worker partitions the data by keys
- Each Reduce worker will later read their partition from every Map worker

Step 5: Reduce Task: sorting

- Reduce worker gets notified by the master about the location of intermediate files for its partition
- Shuffle: Uses RPCs to read the data from the local disks of the map workers
- Sort: When the *reduce* worker reads intermediate data for its partition
  - It sorts the data by the intermediate keys
  - All occurrences of the same key are grouped together

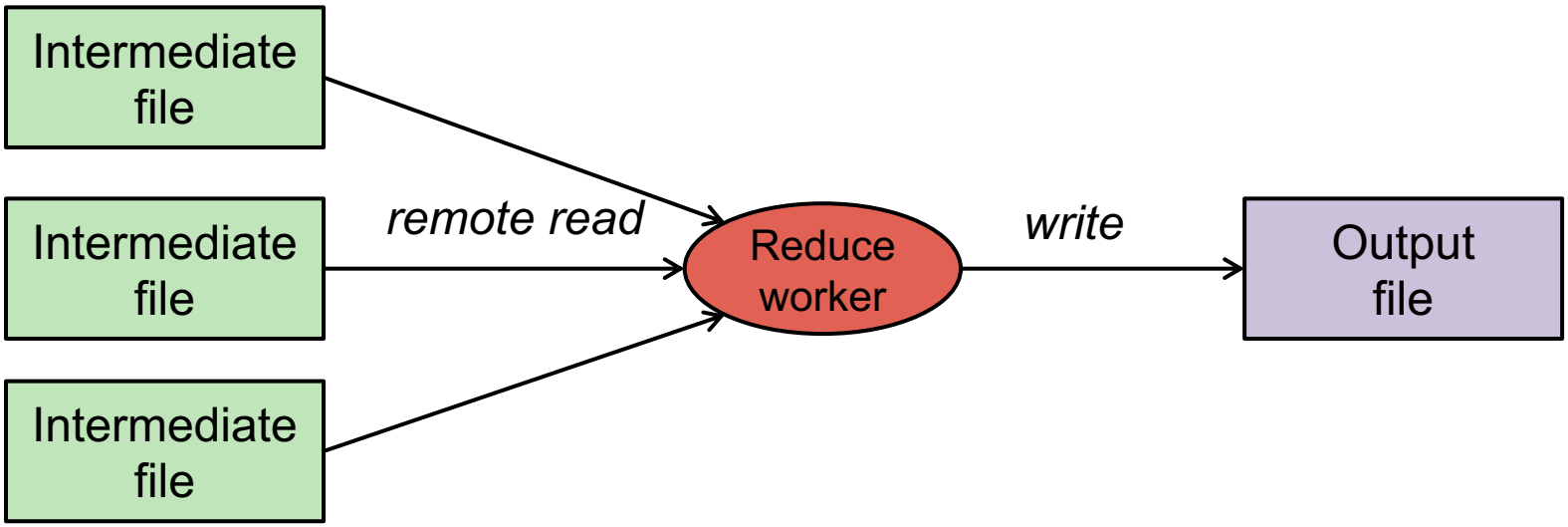


Step 6: Reduce Task: *Reduce*

- The sort phase grouped data with a unique intermediate key
- User's ***Reduce*** function is given the key and the set of intermediate values for that key

  < key, (value1, value2, value3, value4, …) >

- The output of the *Reduce* function is appended to an output file

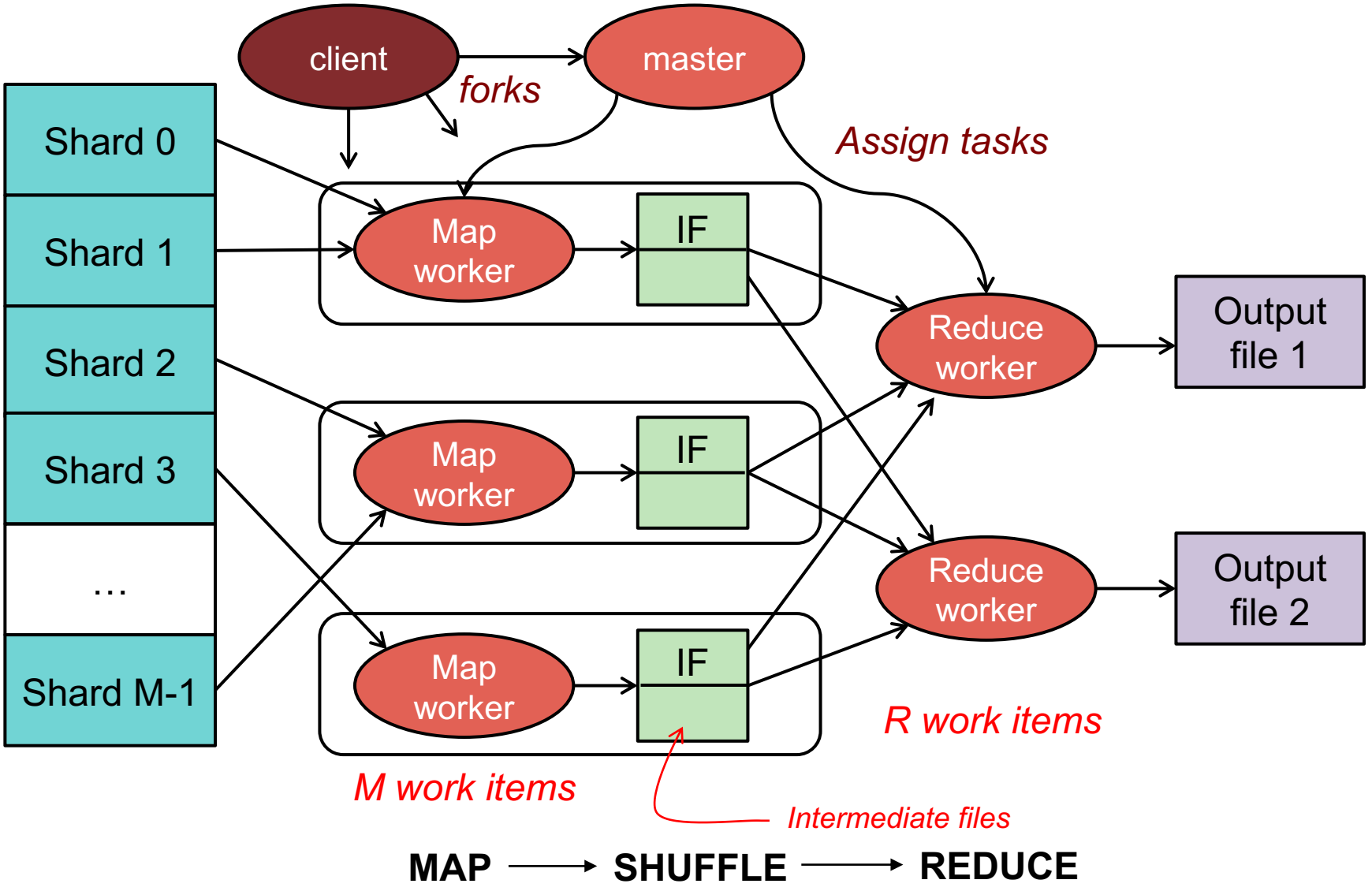


Step 7: Return to user

- When all *map* and *reduce* tasks have completed, the master wakes up the user program

- The *MapReduce* call in the user program returns and the program can resume execution.
  - Output of *MapReduce* is available in *R* output files

MapReduce: the complete picture



Example

- Count # occurrences of each word in a collection of documents
- Map:
  - Parse data; output each word and a count (1)
- Reduce:
  - Sort: sort by keys (words)
  - Reduce: Sum together counts each key (word)

```
map(String key, String value):
// key: document name,  value: document contents
for each word w in value:
  EmitIntermediate(w, "1");

reduce(String key, Iterator values):
// key: a word;  values: a list of counts
int result = 0;
for each v in values:
  result += ParseInt(v);
Emit(AsString(result));
```

Example

It will be seen that this mere painstaking burrower and grub-worm of a poor devil of a Sub-Sub appears to have gone through the long Vaticans and street-stalls of the earth, picking up whatever random allusions to whales he could anyways find in any book whatsoever, sacred or profane. Therefore you must not, in every case at least, take the higgledy-piggledy whale statements, however authentic, in these extracts, for veritable gospel cetology. Far from it. As touching the ancient authors generally, as well as the poets here appearing, these extracts are solely valuable or entertaining, as affording a glancing bird's eye view of what has been promiscuously said, thought, fancied, and sung of Leviathan, by many nations and generations, including our own.

MAP

it 1
will 1
be 1
seen 1
that 1
this 1
mere 1
painstaking 1
burrower 1
and 1
grub-worm 1
of 1
a 1
poor 1
devil 1
of 1
a 1
sub-sub 1
appears 1
to 1
have 1
gone 1

REDUCE

…
a 1
a 1
aback 1
aback 1
abaft 1
abaft 1
abandon 1
abandon 1
abandon 1
abandoned 1
abandoned 1
abandoned 1
abandoned 1
abandoned 1
abandoned 1
abandonedly 1
abandonment 1
abandonment 1
abased 1
abased 1

a 4736
aback 2
abaft 2
abandon 3
abandoned 7
abandonedly 1
abandonment 2
abased 2
abasement 1
abashed 2
abate 1
abated 3
abatement 1
abating 2
abbreviate 1
abbreviation 1
abeam 1
abed 2
abednego 1
abel 1
abhorred 3
abhorrence 1

Fault tolerance

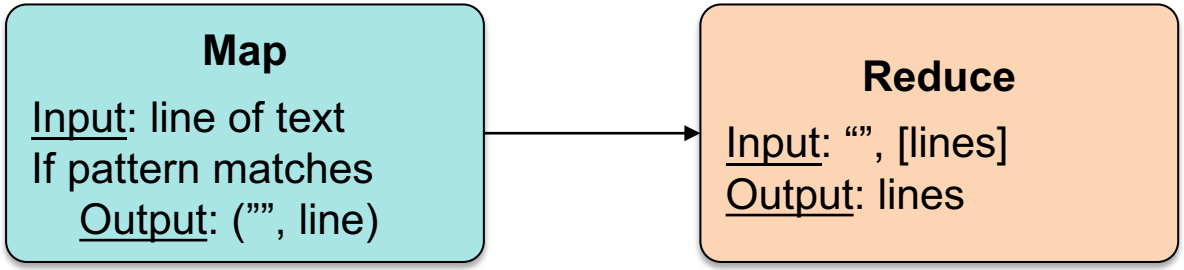
- Master pings each worker periodically
  - If no response is received within a certain time, the worker is marked as *failed*
  - *Map* or *reduce* tasks given to this worker are reset back to the initial state and rescheduled for other workers.

Locality

- **Input and Output files**
  - GFS (Google File System)
  - Bigtable
- MapReduce (often) **runs on** GFS chunkservers
  - Keep computation close to the files if possible
- Master tries to schedule *map* worker on one of the machines that has a copy of the input chunk it needs.

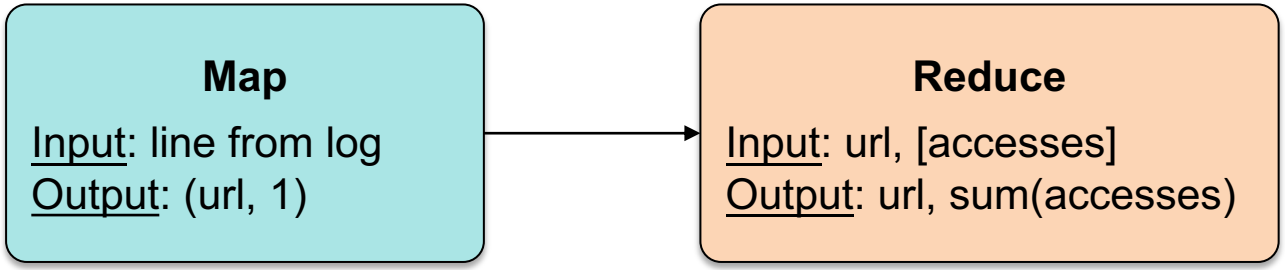
Other Examples

- Distributed grep (search for words)
  - *Search for words in lots of documents*
  - Map: emit a line if it matches a given pattern
  - Reduce: just copy the intermediate data to the output



Other Examples

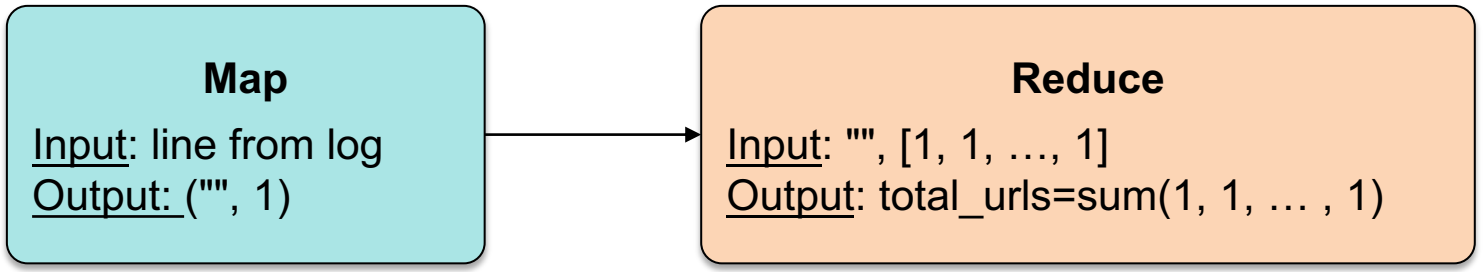
- List URL access counts
  - *Find the count of each URL in web logs*
  - Map: process logs of web page access; output <URL, 1>
  - Reduce: add all values for the same URL



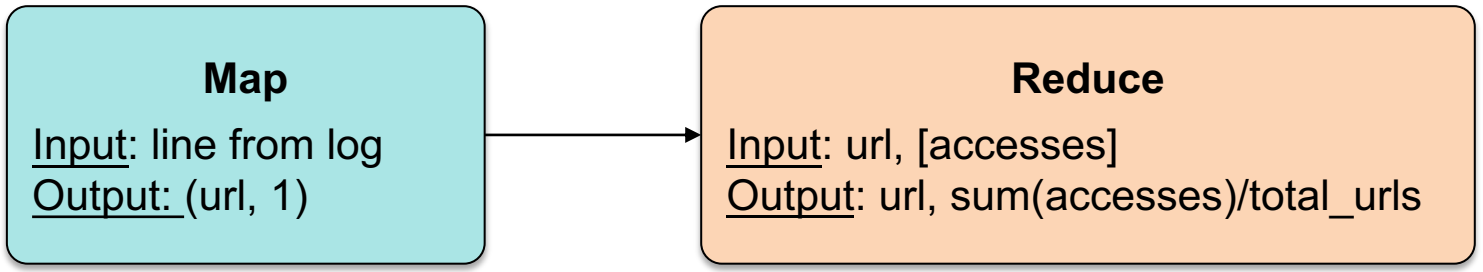
Other Examples

- Count URL access **frequency**
  - *Find the frequency of each URL in web logs*
  - Run 1: just count total URLs
  - Run 2: just like URL count but now we stored **total_urls**

Run 1

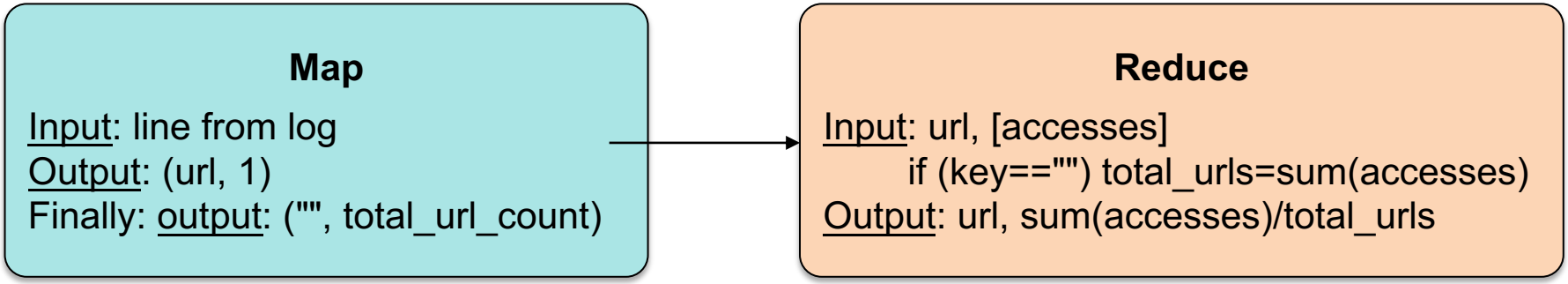


Run 2



Other Examples

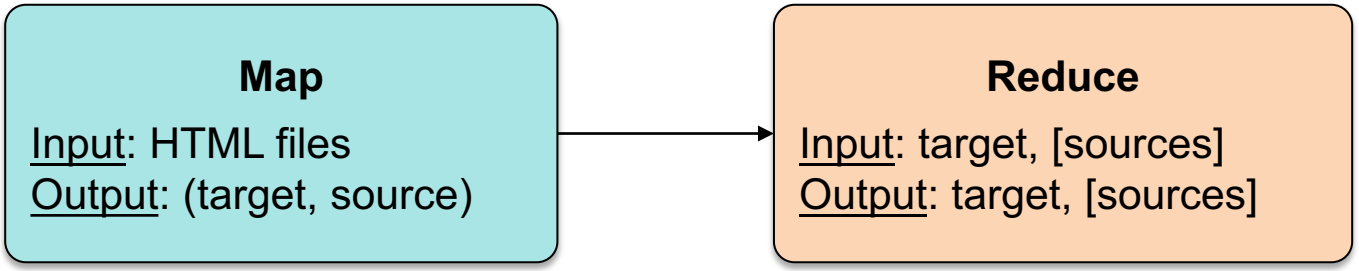
- Count URL access **frequency**
  - *Can we avoid processing the input data twice?*
  - Technically, no, but there's a hack
    - Map task sends its total URL count to *all* reducers with **key=""**
    - The key **""** happens to be the one that gets processed first
  - **Map**: process logs of web page access
    - Output <URL, 1>
    - Output total URL count to all reducers
  - **Reduce**: add all values for the same URL



Other Examples

- Reverse web-link graph
  - *Find where page links come from*
  - Map: output <target, source> for each link to *target* in a page *source*
  - Reduce: concatenate the list of all source URLs associated with a target.

    Output <target, list(source)>

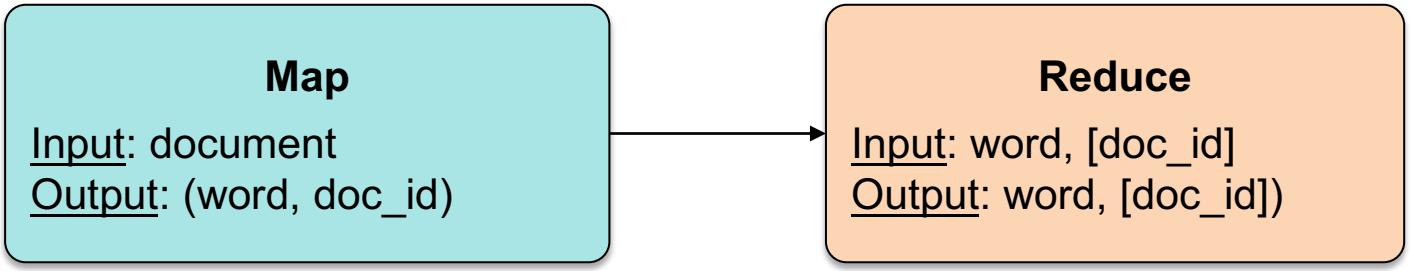


The *reduce* function does nothing!

Other Examples

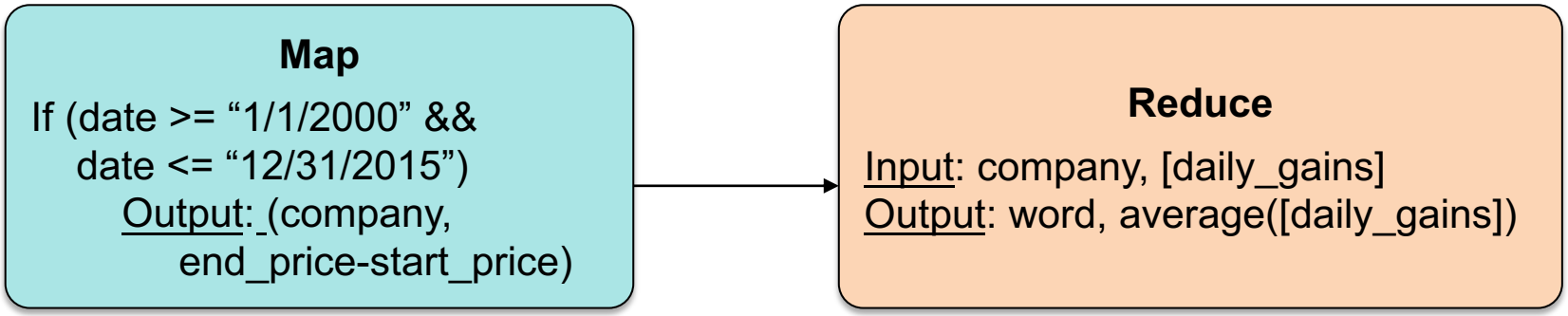
- Inverted index
  - *Find what documents contain a specific word*
  - Map: parse document, emit <word, document-ID> pairs
  - Reduce: for each word, sort the corresponding document IDs

    Emit a <word, list(document-ID)> pair
    The set of all output pairs is an inverted index



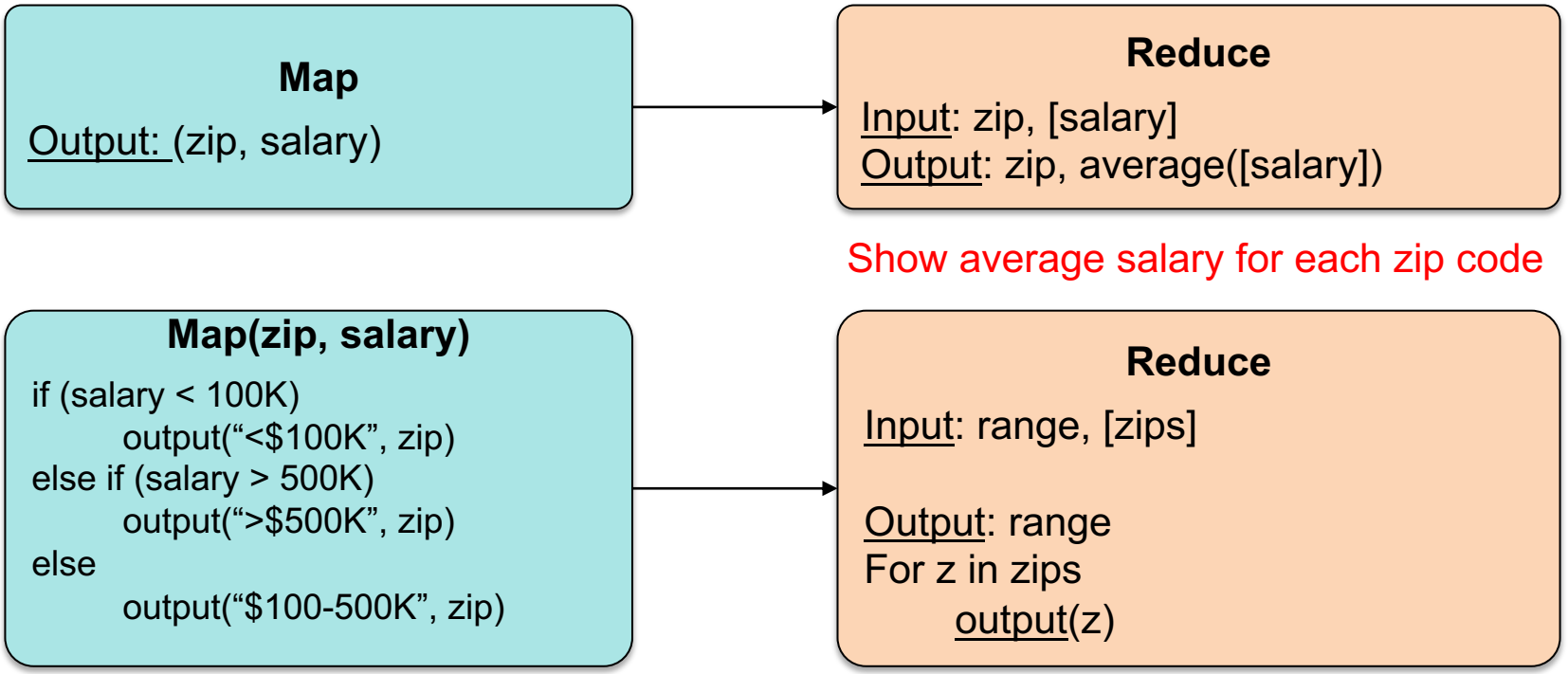
Other Examples

- Stock summary
  - *Find average daily gain of each company from 1/1/2000 – 12/31/2015*
  - Data is a set of lines: { date, company, start_price, end_price }

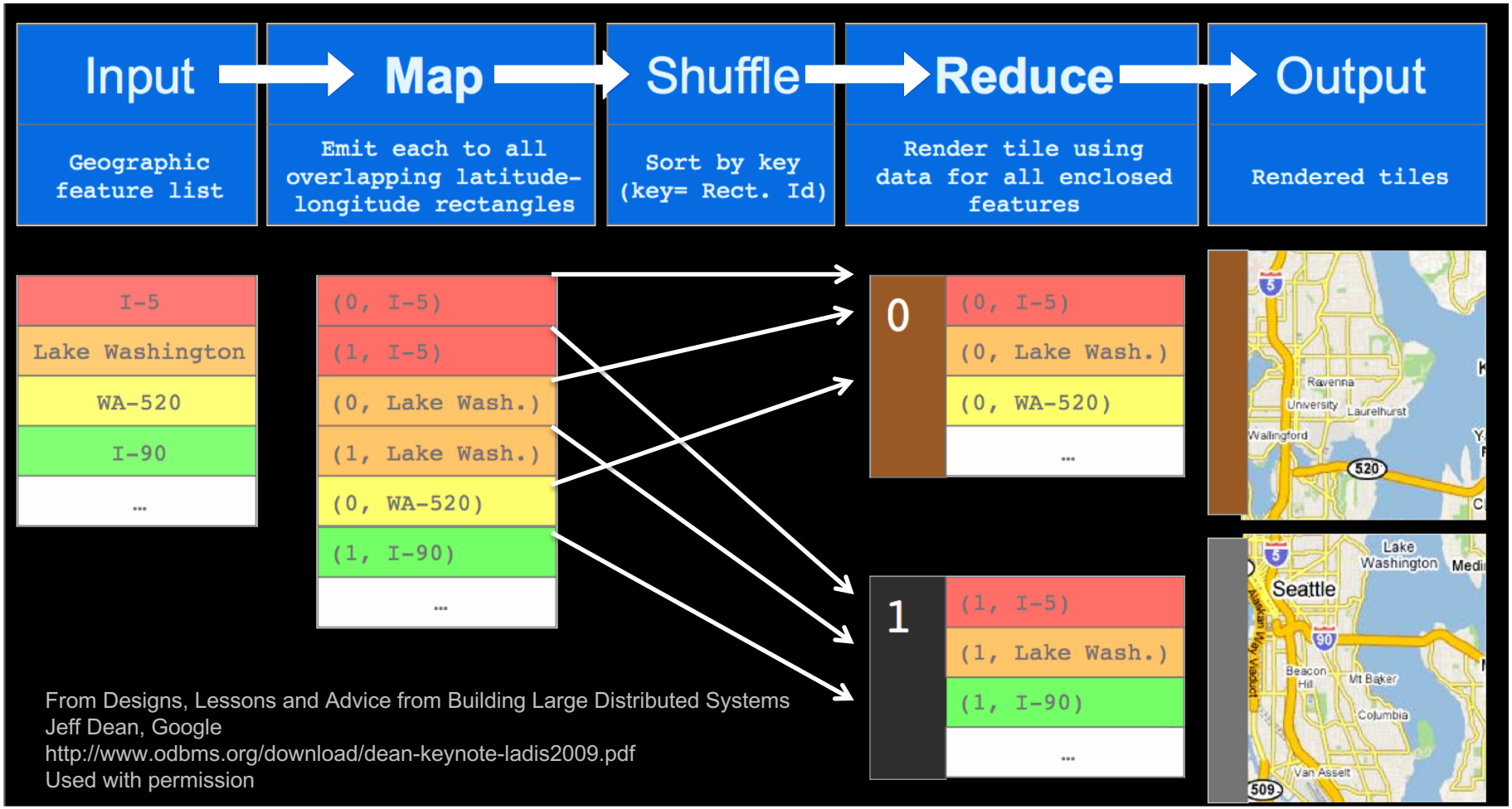


Other Examples: Two rounds

- Average salaries in regions
  - *Show zip codes where average salaries are in the ranges:*
    (1) < $100K (2) $100K … $500K (3) > $500K
  - Data is a set of lines: { name, age, address, zip, salary }



MapReduce for Rendering Map Tiles



From Designs, Lessons and Advice from Building Large Distributed Systems
Jeff Dean, Google
http://www.odbms.org/download/dean-keynote-ladis2009.pdf
Used with permission

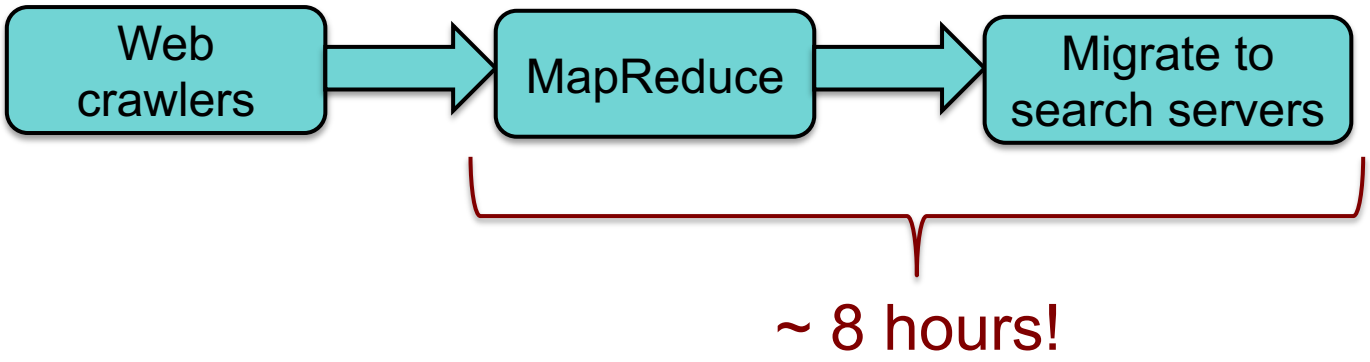
MapReduce Summary

- Get a lot of data
- Map
  - Parse & extract items of interest
- Sort (shuffle) & partition
- Reduce
  - Aggregate results
- Write to output files

All is not perfect

- MapReduce was used to process webpage data collected by Google's crawlers.
  - It would extract the links and metadata needed to search the pages
  - Determine the site's PageRank

- The process took around eight hours.
  - Results were moved to search servers.
  - This was done continuously.



All is not perfect

- Web has become more dynamic
  - an 8+ hour delay is a lot for some sites
- Goal: refresh certain pages within seconds
- MapReduce
  - Batch-oriented
  - Not suited for near-real-time processes
  - Cannot start a new phase until the previous has completed
    - Reduce cannot start until all Map workers have completed
  - Suffers from “stragglers” – workers that take too long (or fail)
  - This was done continuously
- MapReduce is still used for many Google services
- Search framework updated in 2009-2010: Caffeine
  - Index updated by making direct changes to data stored in Bigtable
  - Data resides in Colossus (GFS2) instead of GFS

In Practice

- Most data not simple files
  - B-trees, tables, SQL databases, memory-mapped key-values
- Hardly ever use textual data: slow & hard to parse
  - Most I/O encoded with Protocol Buffers

More info

- Good tutorial presentation & examples at:

  http://research.google.com/pubs/pub36249.html

- The definitive paper:

  http://labs.google.com/papers/mapreduce.html

The End