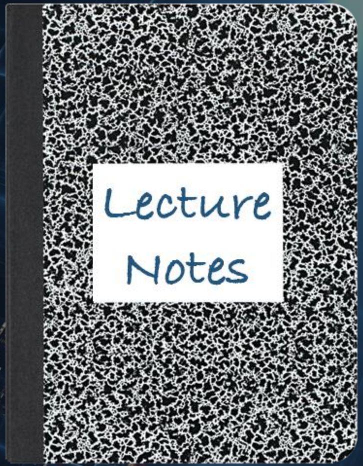
CS 417 – DISTRIBUTED SYSTEMS



Week 10: Large-Scale Data Processing

Part 1: MapReduce

Paul Krzyzanowski

Processing large amounts of data

Suppose you need to perform some computation on a huge amount of data – searching, indexing, generating statistics …

- *Even small amounts of processing can add up*
  - 100ms per data item × 1 billion items = 1157 days of computation!
- What do we do?
  - Break the work up so lots of computers can work on just parts of the data
    - Split the workload among 10,000 computers ⇒ 2.7 hours of computation
- Put the data on a file server?
  - You might have more data than you can fit on one system
  - Disk bandwidth will be an issue: if you read an SSD at 500 MB/s, it becomes a bottleneck before the network
  - And bandwidth is shared, so those 10,000 systems will get data at < 5KB/s

We need to distribute the workload *and* the data

Goals

Traditional programming is serial

Parallel programming in a distributed environment

- Split 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

Dealing with distributed software is a pain!

Interact with the distributed environment

- Split up data
- Allocate servers
- Get programs onto those servers and start them
- Partition the work among the processes
- Monitor for failure
- Restart failed processes
- Collect the results

None of this relates to solving the user's problem –
it's software infrastructure

Simplest environment for parallel processing

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

One

Master

- Initializes data set and splits it according to # of workers
- Sends each worker a sub-array of data
- Receives the results from each worker

Many

Worker

- Receives a sub-array from master
- Performs processing
- Sends results to master

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
```

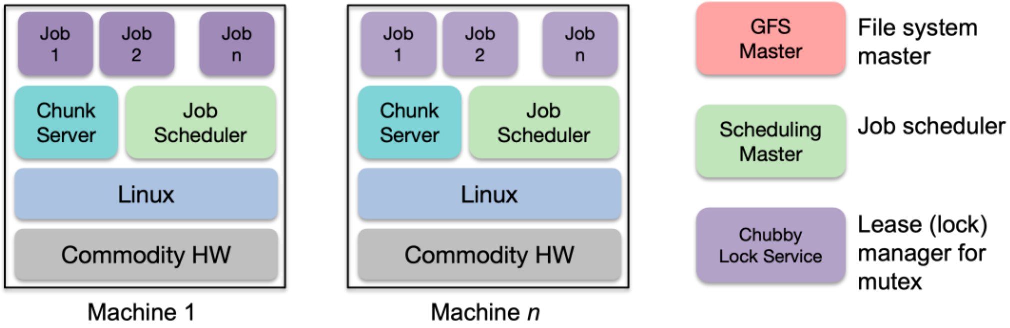
MapReduce

- Framework for parallel computing
- Programmers get simple API
- Don’t have to worry about handling
  - Parallelization
  - Program distribution
  - Data distribution
  - Load balancing
  - Fault tolerance
  - Monitoring

Allows a user to process huge amounts of data (terabytes and petabytes) on thousands of processors

Who has it?

- Google
  - Original proprietary implementation
  - Runs on the same computers as the data storage (GFS, Bigtable)



- Apache Hadoop MapReduce
  - Most common (open-source) implementation
  - Built based on the Google paper
- Others
  - Amazon Elastic MapReduce – uses Hadoop MapReduce running on Amazon EC2
  - Microsoft Azure HDInsight
  - Google Cloud MapReduce for App Engine

MapReduce – a high-level view

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 & passes 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

Keys are distributed to Reduce workers are by partitioning the intermediate key space into *R* pieces using a partitioning function (default = *hash(key) mod R*)

- The user specifies the # of partitions (R) and, optionally, 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 <u>all</u> 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

Step 1: Split input files into chunks (shards/splits)

Break up the input data into *M* pieces
(typically 64 MB to match GFS chunk size)

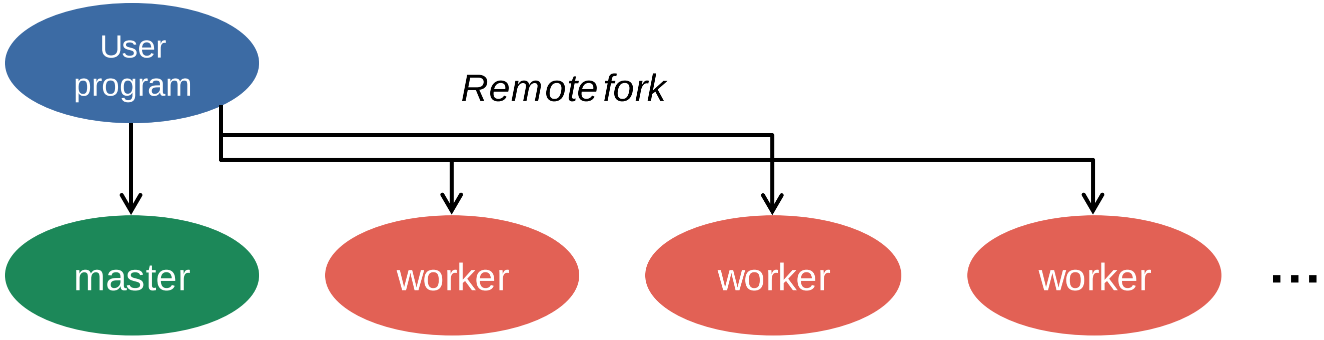
Shard 0	Shard 1	Shard 2	Shard 3	…	Shard M-1

Input data

Divided into *M* shards (splits)

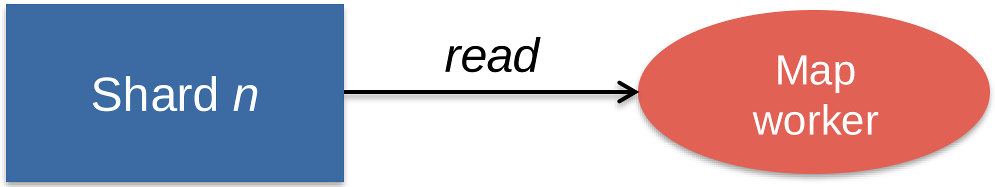
Step 2: Fork processes

- 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* tasks
    - *R* = # partitions, defined by the user



Step 3: Run Map Tasks

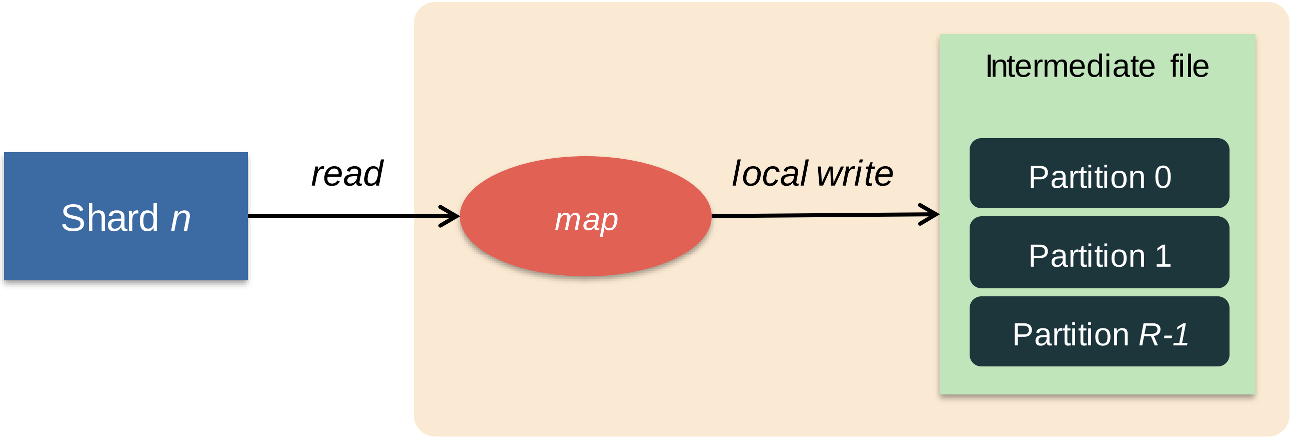
- Reads contents of the input shard assigned to it
- Parses key/value pairs out of the input data
- Passes each pair to a user-defined map function
  - Produces intermediate key/value pairs
  - These are buffered in memory



MapReduce frameworks support multiple input formats:
Input data may be a single file, directories of files, database query results, and various data formats, such as binary, text, line-oriented text, and key-value pairs

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
- Notifies master when complete
  – Passes locations of intermediate data to the master
  – Master forwards these locations to the reduce worker



Step 4a. Partitioning

- Map key-value data will be processed by *Reduce* workers
  – The user's Reduce function will be called once per unique key generated by Map.

- We first need to group all the *(key, value)* data by keys and decide which Reduce worker processes which set of keys
  – The Reduce worker will later sort the values within each keys

Partition function

Decides which of R reduce workers will work on which keys

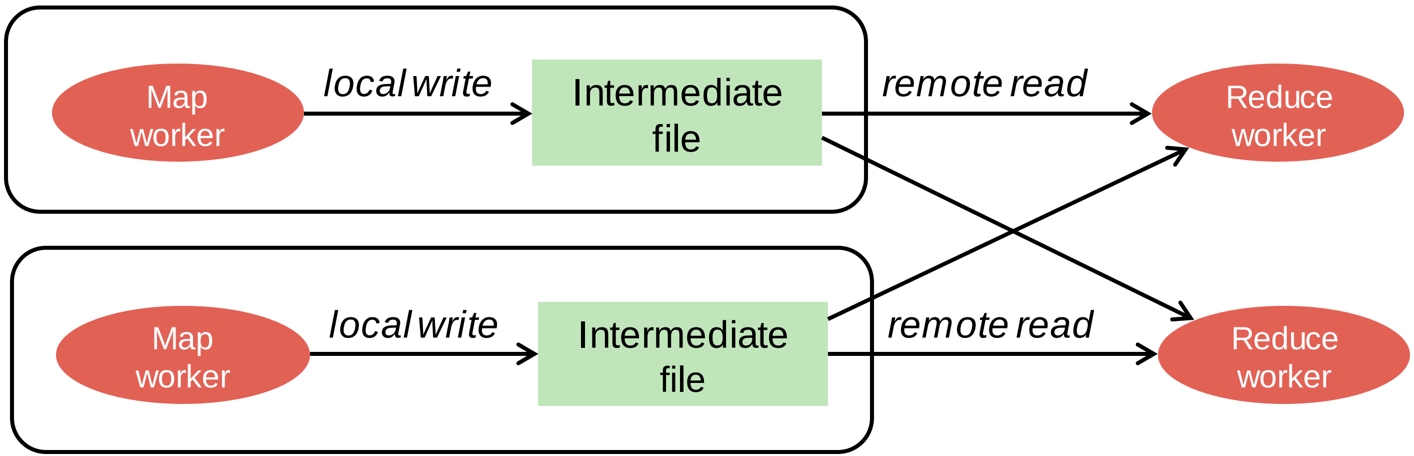
– Default function to identify a reduce worker: *hash(key) mod R*
– Map worker partitions the data by groups of keys for each *Reduce* worker

- Each *Reduce* worker will later read their partition from every *Map* worker

Step 5: Reduce Task: Shuffle & Sort

Reduce worker is 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 gets all the (*key, value*) data for its partition from all workers
  - It sorts the data by the keys
  - All occurrences of the same key are grouped together

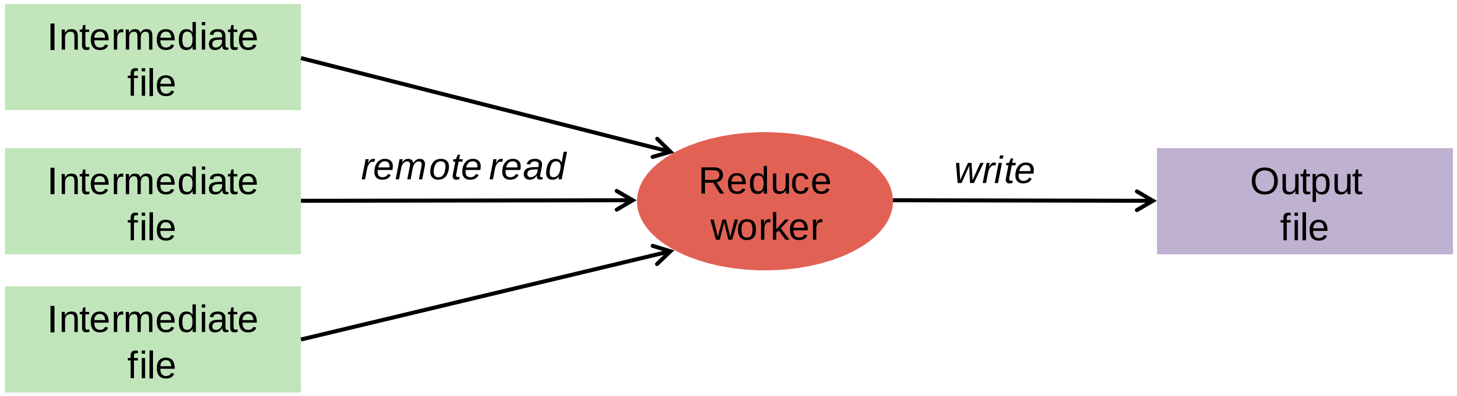


Step 6: Reduce Task: *Reduce*

- The *sort* phase grouped data by keys
  - This makes it easy to identify all the values from all the map workers that are associated with each key
- The user's ***Reduce*** function is given the key and the set of intermediate values for that key

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

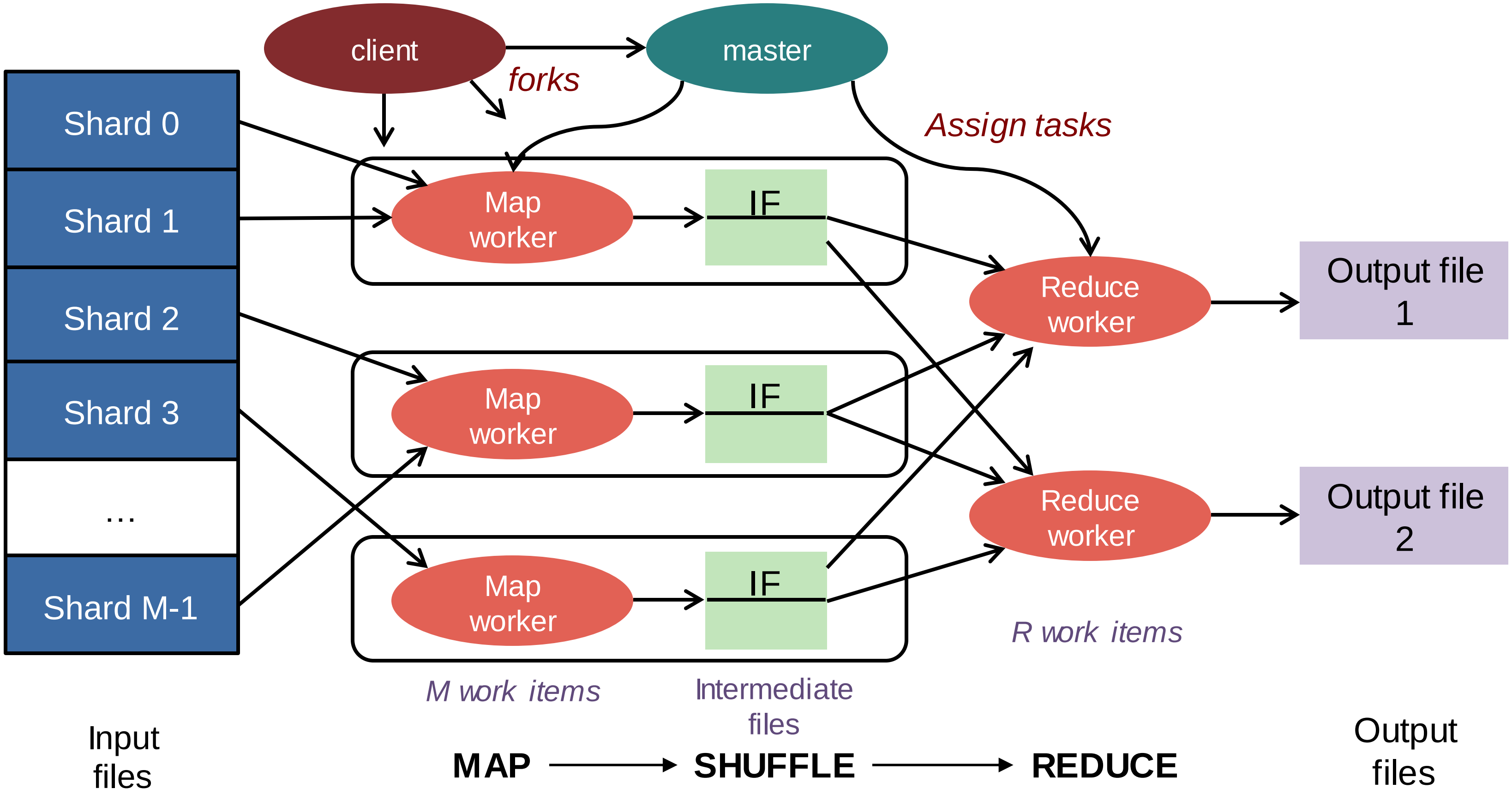
- The output of the *Reduce* function is appended to an output file



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: Word Count

- 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: Word Count

Input

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 …

MAP

After Map
[Intermediate file]

it 1
will 1
be 1
seen 1
that 1
this 1
mere 1
painstaking
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

After Sort

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
abandoned 1
abandonedly 1
abandonment 1
abandonment 1
abased 1
abased 1

REDUCE

Input to *reduce()*

a, (1, 1, 1, …)
aback, (1, 1)
abaft, (1, 1)
abandon, (1, 1, 1)
abandoned, (1 …)
abandonedly, (1)
abandonment, (1, 1)
abased, (1, 1)
abasement, (1)
abashed, (1, 1)
abate, (1)
abated, (1, 1, 1)
abatement 1
abating, (1, 1)
abbreviate, (1)
abbreviation, (1)
abeam, (1)
abed, (1, 1)
abednego , (1)
abel , (1)
abhorred, (1, 1, 1)
abhorrence , (1)

Output from *reduce()*

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

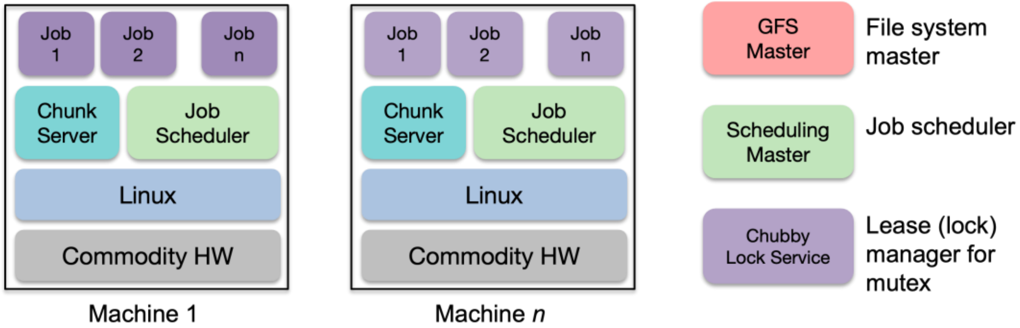
Fault tolerance

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 data comes from:**
  - GFS (Google File System) file or files
  - Bigtable, Spanner



- 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: Search

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

Map

Input: line of text
If pattern matches
Output: ("", line)

Reduce

Input: "", [lines]
Output: lines

Other Examples: URL access counts

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

Map

Input: line from log
Output: (url, 1)

→

Reduce

Input: url, [accesses]
Output: url, sum(accesses)

Other Examples: URL access frequency

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**

	Map		Reduce
Run 1	Input: line from log Output: ("", 1)	→	Input: "", [1, 1, …, 1] Output: total_urls=sum(1, 1, … , 1)
Run 2	Input: line from log Output: (url, 1)	→	Input: url, [accesses] Output: url, sum(accesses)/total_urls

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)>

Map

Input: HTML files
Output: (target, source)

Reduce

Input: target, [sources]
Output: target, [sources]

The *reduce* function does nothing –
it just writes the input to the output!

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

Map

Input: document
Output: (word, doc_id)

→

Reduce

Input: word, [doc_id]
Output: word, [doc_id])

Other Examples

Stock performance summary

- *Find average daily gain of each company from 1/1/2010 – 12/31/2020*
- Data is a set of lines: { date, company, start_price, end_price }

Map

If (date >= "1/1/2010" &&
    date <= "12/31/2020")
        Output: (company, end_price - start_price)

→

Reduce

Input: company, [daily_gains]
Output: word, average([daily_gains])

Other Examples: Two rounds of MapReduce

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 }

Map

Output: (zip, salary)

Reduce

Input: zip, [salary]
Output: zip, average([salary])

Show average salary for each zip code

Map(zip, salary)

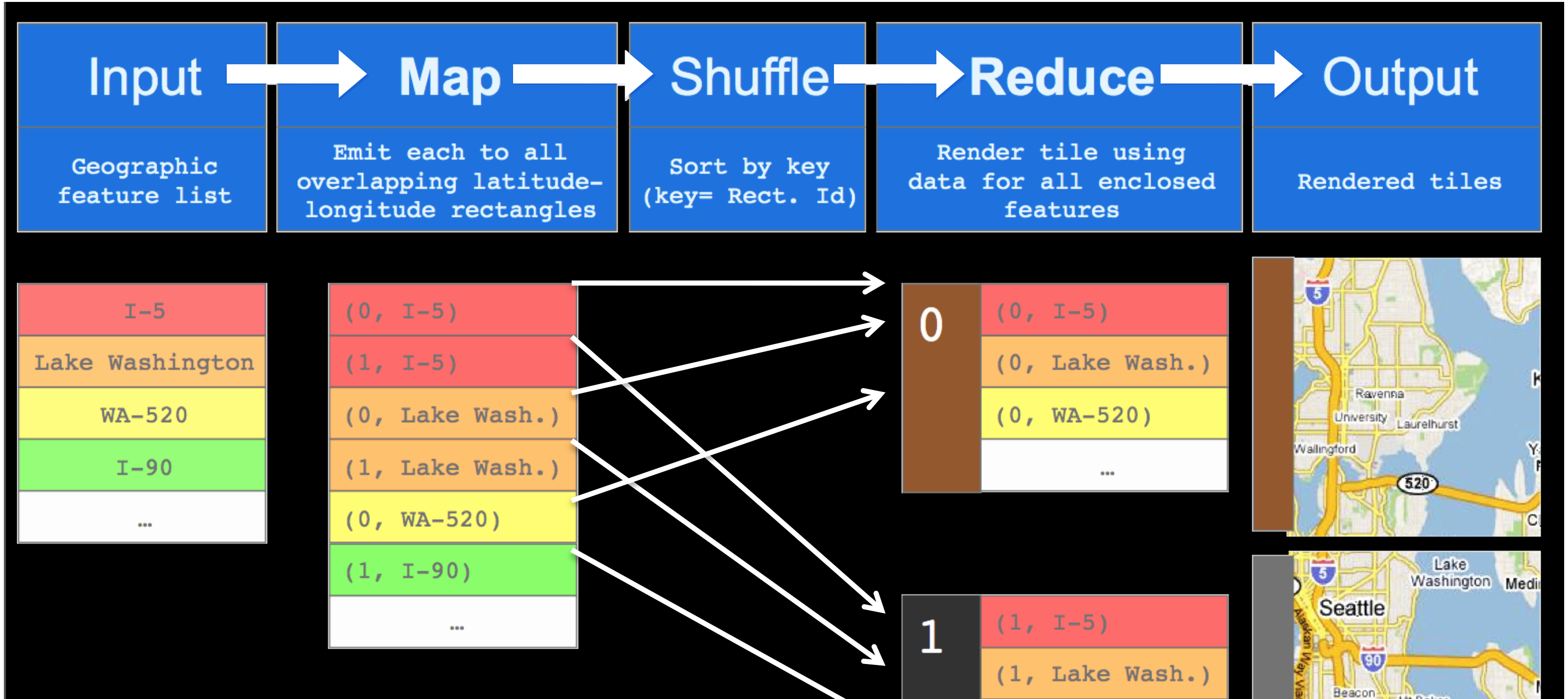
```
if (salary < 100K)
        output("<$100K", zip)
else if (salary > 500K)
        output(">$500K", zip)
else
        output("$100-500K", zip)
```

Reduce

Input: range, [zips]

Output: range
For z in zips
        output(z)

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

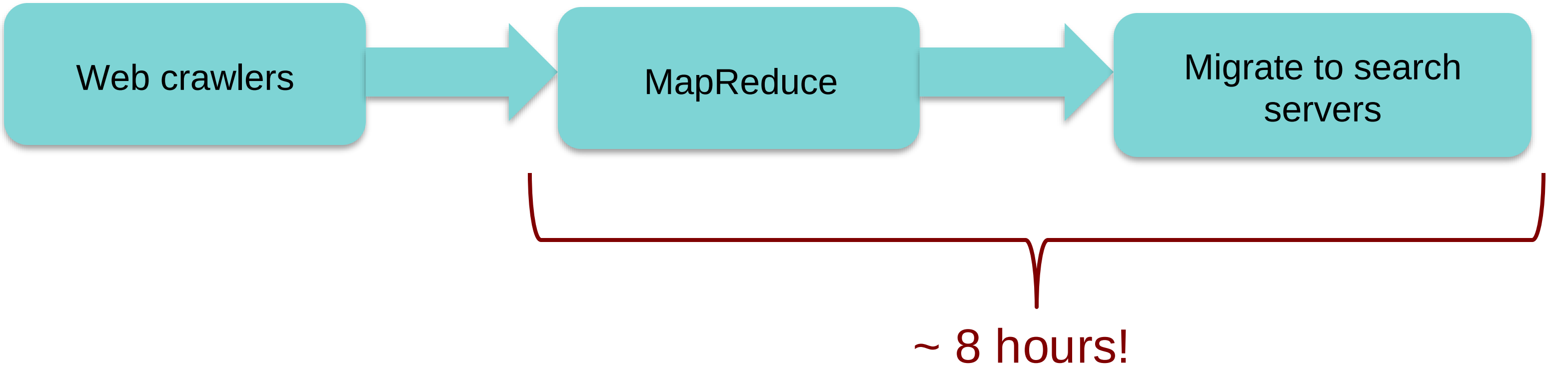
MapReduce Summary

- 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 useful but there are also other options
- Search framework updated in 2009-2010: Caffeine
  - Analyze web in small portions
  - Update index by making direct changes to data stored in Bigtable
  - Process hundreds of thousands of pages per second in parallel – data resides in Colossus (GFS2) instead of GFS

In Practice

- Most data is not stored as simple files
  - B-trees, tables, SQL databases, memory-mapped key-values

- We don't usually use textual data: it's slow & hard to parse
  - Most I/O gets 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