Distributed Systems

17. MapReduce

Paul Krzyzanowski

pxk@cs.rutgers.edu

Credit

Much of this information is from the Google Code University:

http://code.google.com/edu/parallel/mapreduce-tutorial.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 equal-size chunks
- Each process can work on a chunk
- Master/worker approach
  - Master:
    - Initializes array and splits it according to # of workers
    - Sends each worker the sub-array
    - Receives the results from each worker
  - Worker:
    - Receives a sub-array from master
    - Perfoms 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

- 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

MapReduce

- Map: (input shard) → intermediate(key/value pairs)
  - Map calls are distributed across machines by automatically partitioning the input data into M "shards".
  - MapReduce library groups together all intermediate values associated with the same intermediate key & passes them to the *Reduce* function
- Reduce: intermediate(key/value pairs) → result files
  - Accepts an intermediate key & a set of values for the key
  - It merges these values together to form a smaller set of values
  - Reduce calls 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

- Map

  Grab the relevant data from the source

  User function gets called for each chunk of input

- Reduce

  Aggregate the results

  User function gets called for each unique key

MapReduce: what happens in between?

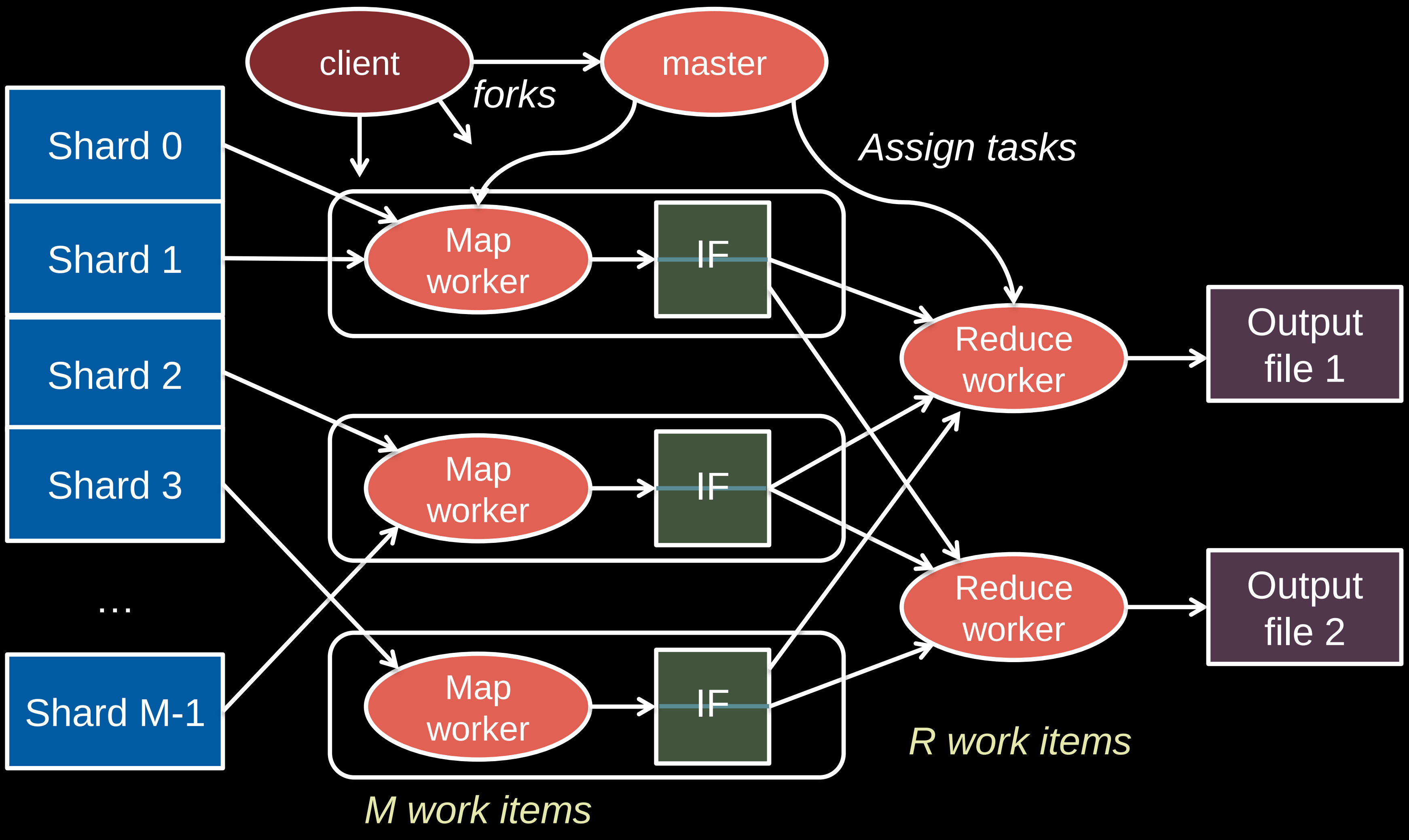
- Map
  - Grab the relevant data from the source (parse into key, value)
  - Write it to an intermediate file
- Partition
  - Partitioning: identify which of *R* reducers will handle which keys
  - Map partitions data to target it to one of *R* Reduce workers based on a partitioning function (both *R*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

Map Worker

- Sort
  - Fetch the relevant partition of the output from all mappers
  - Sort by keys (different mappers may have output 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



Input files

Map workers

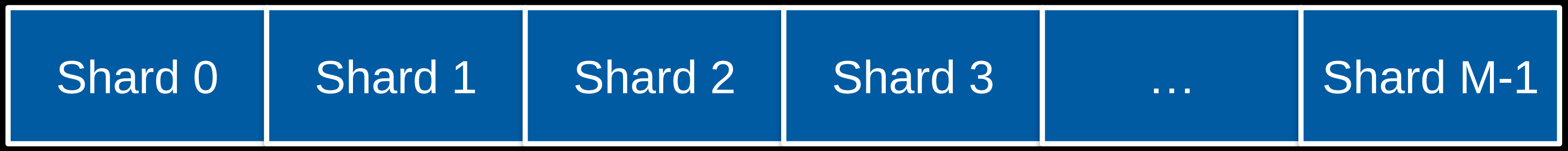
Intermediate files

Reduce workers

Output files

Step 1: Split input files into chunks (shards)

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



Shard 0

Shard 1

Shard 2

Shard 3

…

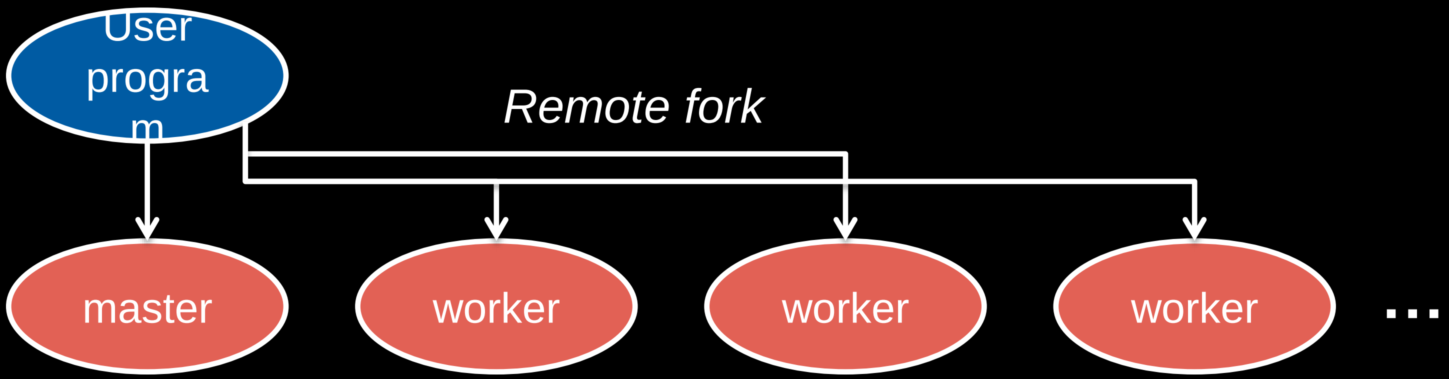
Shard M-1

Input files

Divided into *M* shards

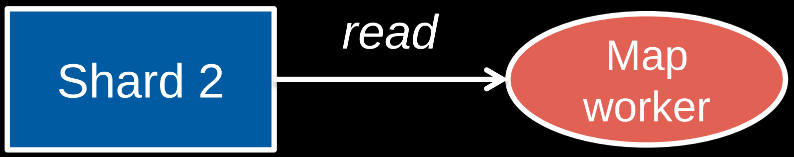
Step 2: Fork processes

- Start up many copies of the program on a cluster of machines
  - 1 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



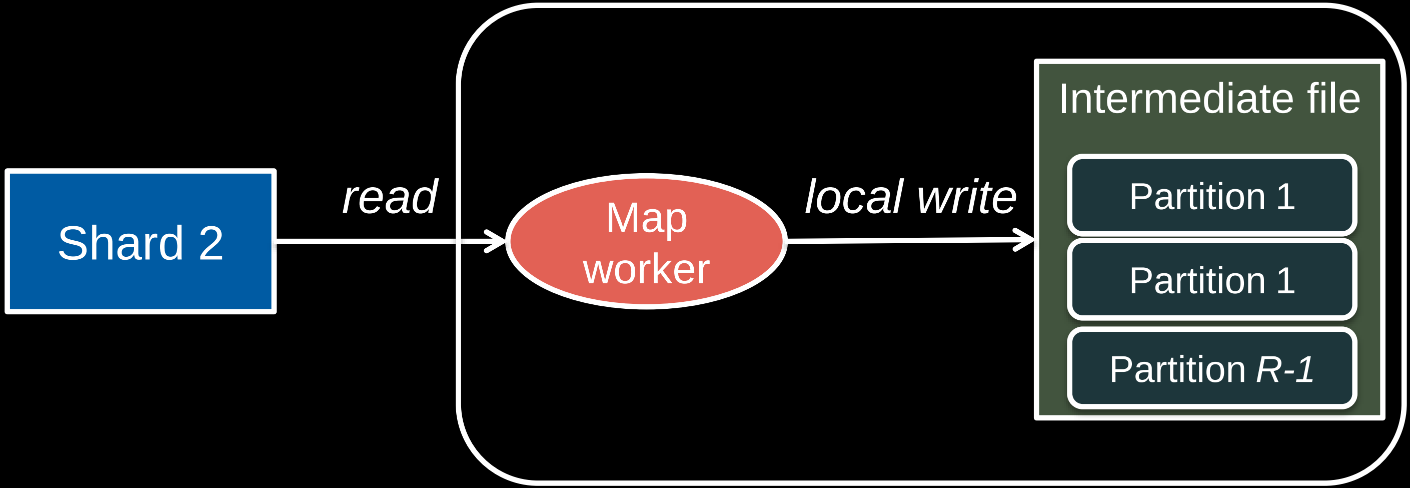
Step 3: Map Task

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

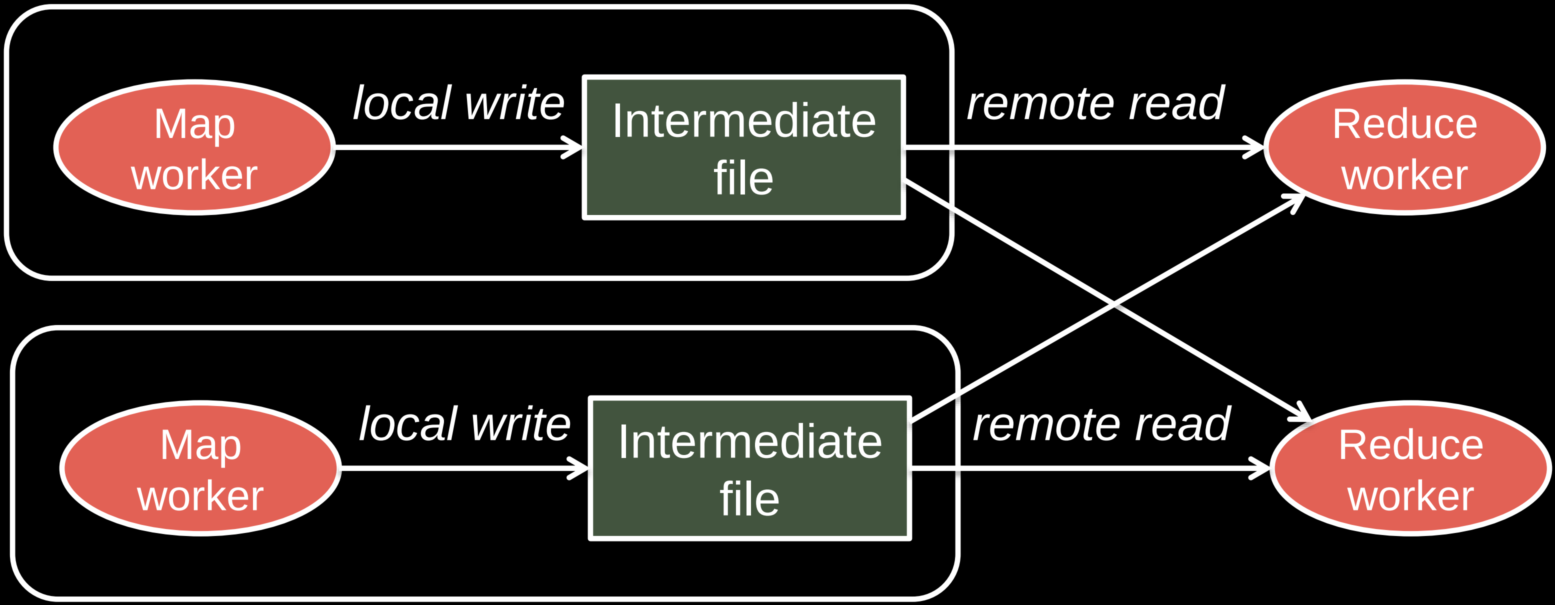


Step 4a. Partitioning

- Map data will be processed by Reduce workers
  - The user's *Reduce* function will be called once per unique key generated by *Map*.
- This means we will need to sort all the (key, value) data by keys and decide which Reduce worker processes which keys – the Reduce worker will do this
- 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 read their partition from every Map worker

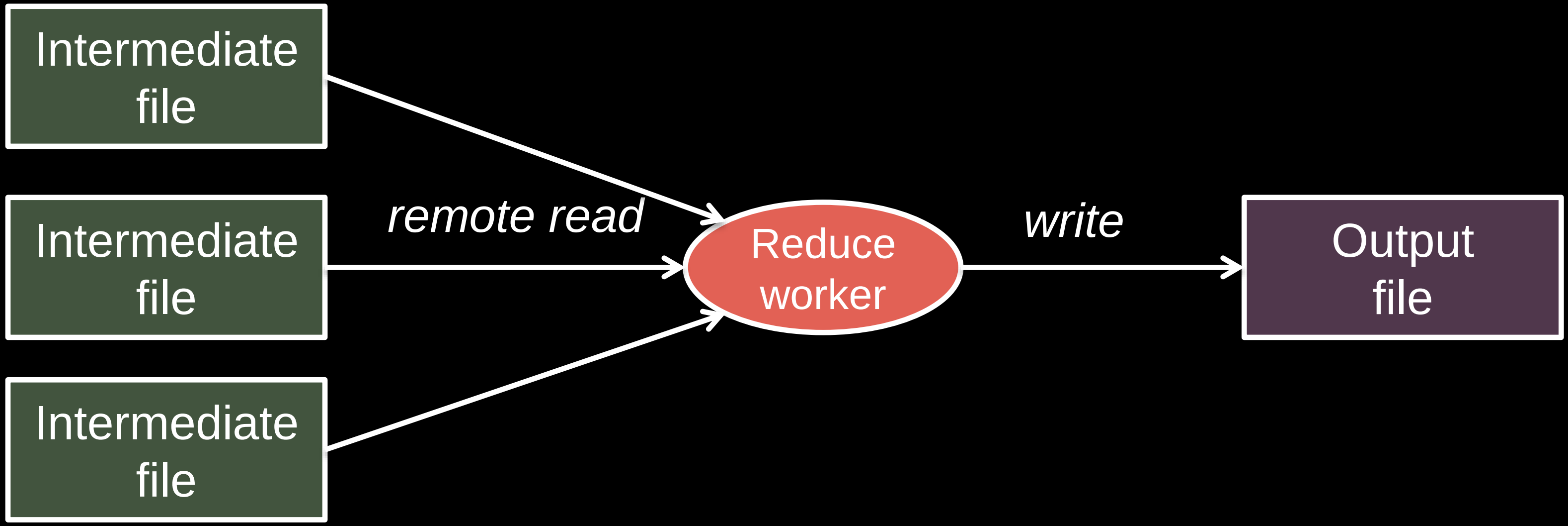
Step 5: Reduce Task: sorting

- Reduce worker gets notified by the master about the location of intermediate files for its partition
- Uses RPCs to read the data from the local disks of the map workers
- 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

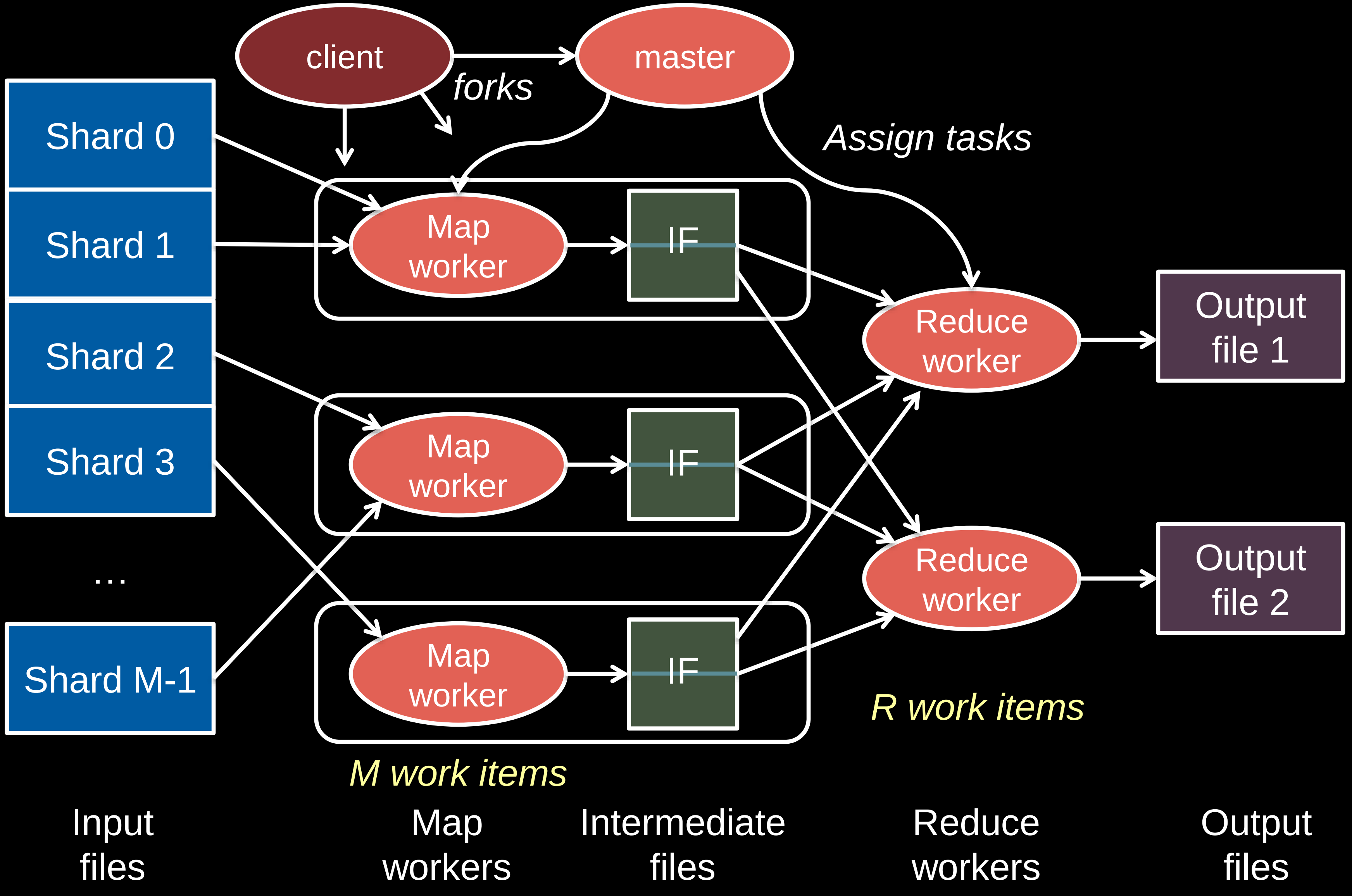


Step 7: Return to user

- 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



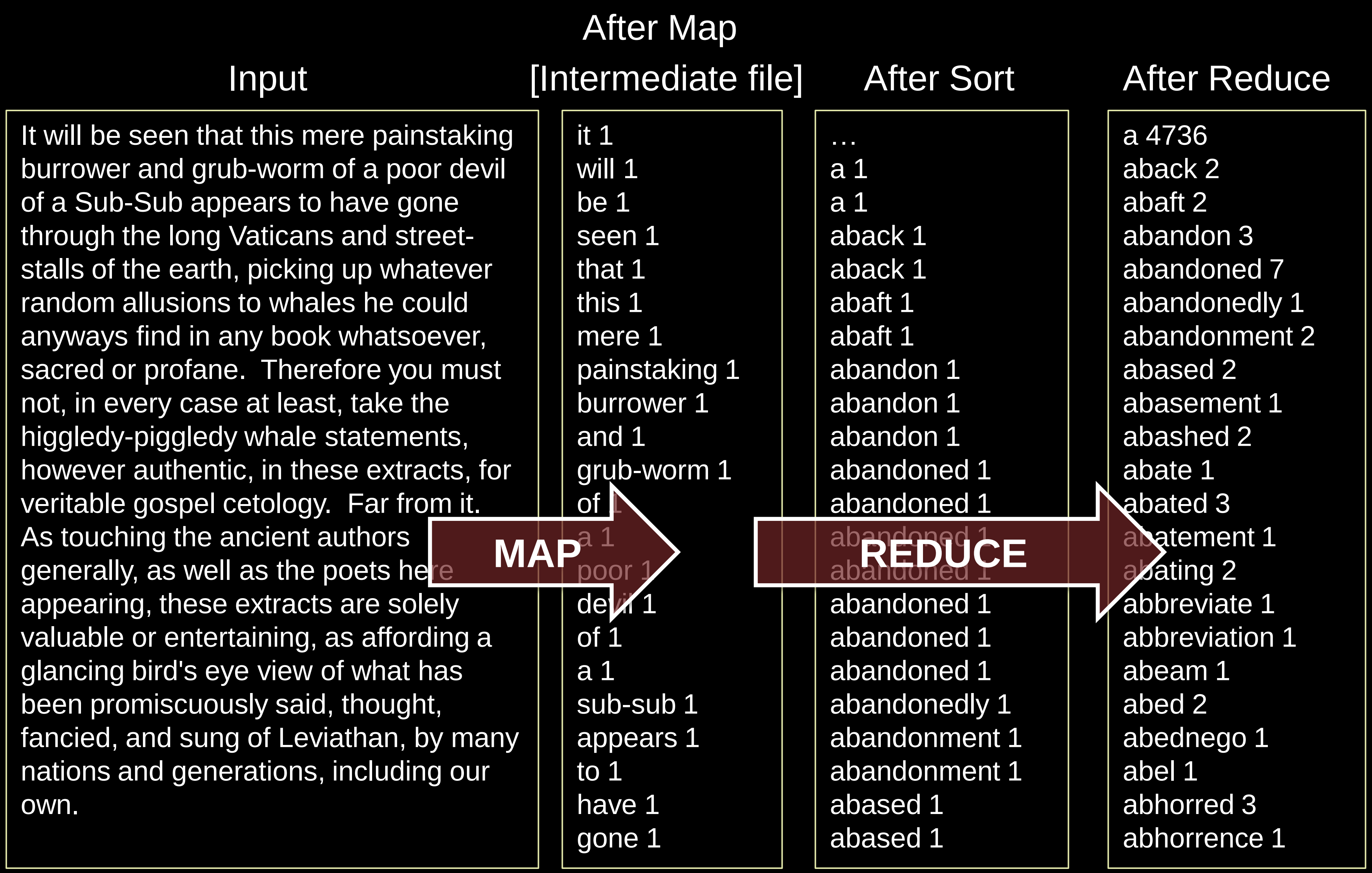
Example

- 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



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 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.

After Map
[Intermediate file]

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

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

After 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 files are on GFS (Google File System)
- MapReduce runs on GFS chunkservers
- 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

- Count URL access frequency
  - *Find the frequency 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

- 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)>

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

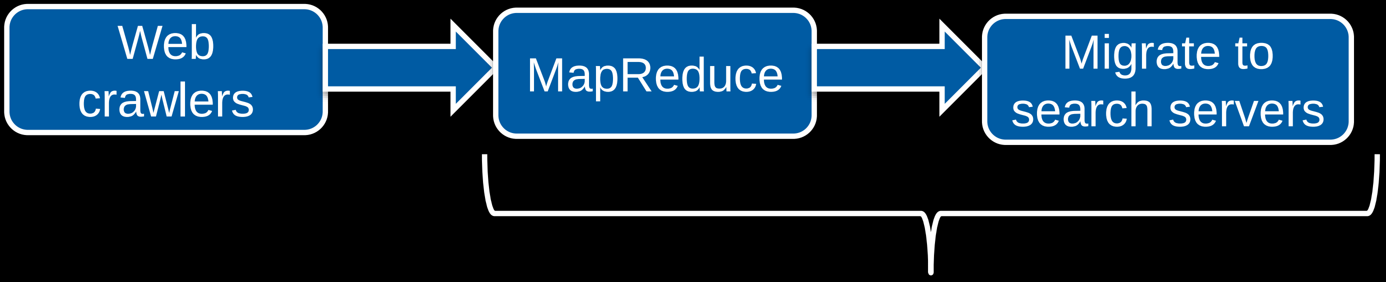
MapReduce Summary

- Get a lot of data
- Map
  - Parse & extract items of interest
- Sort & 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.



~ 8 hours!

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