MapReduce: simplified data processing on large clusters

by Jeffrey Dean and Sanjay Ghemawat

Presented by Israa Alqassem

11.10.2020

Authors

- Google Fellows, i.e., the world's leading experts in their fields
- Google AI and Systems Infrastructure Groups
- Pair programming
- Worked together on:
  - MapReduce
  - Google File System
  - Spanner
  - Bigtable
  - TensorFlow

Roadmap

- ➔ Introduction
- ➔ Examples
- ➔ How it works
- ➔ Fault tolerance
- ➔ Debugging
- ➔ Performance

What is MapReduce

- An automated parallel programming model for processing large datasets
  - User implements Map() and Reduce() functions
- A framework to run on large clusters of machines
- Can also be used to parallelize computations across multiple cores of the same machine
- Libraries take care of the rest
  - Data partition and distribution
  - Parallel computation
  - Fault tolerance
  - Load balancing

Processing of large datasets

- Useful
  - Google
- Very large data sets need to be processed
- Lots of machines
  - Use them efficiently

Examples:

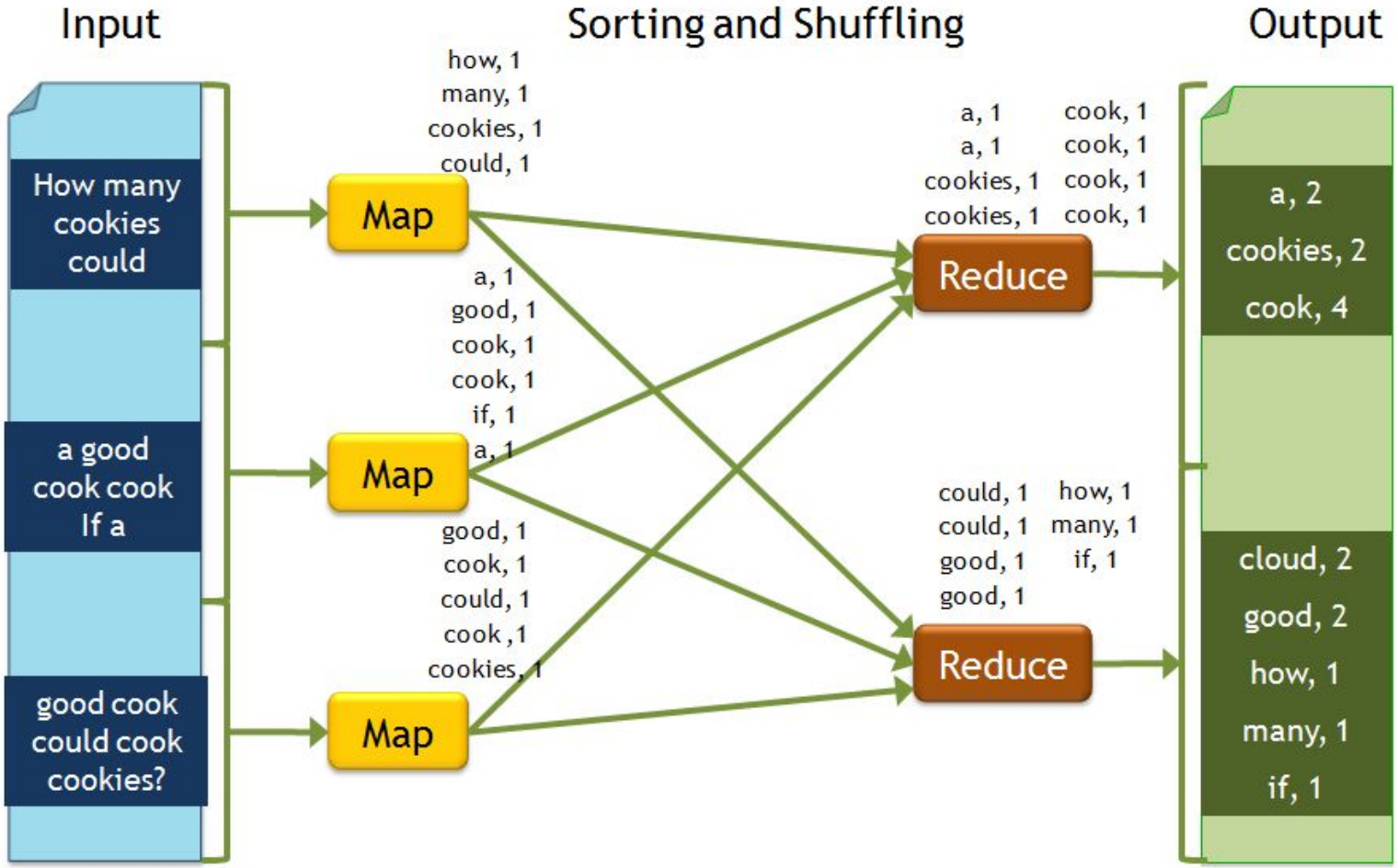
- Counting URL access frequency:
  - Input: list(RequestURL)
  - Output: list(RequestURL, total_number)
- Distributed grep
- Distributed sort

Programming model: Map and Reduce

- Map and Reduce functions borrowed from the functional programming language, LISP
- Map()
  - Process a key/value pair to generate intermediate key/value pairs
  - map (in_key, in_value) -> (out_key, intermediate_value) list
- Reduce()
  - Merge all intermediate values associated with the same key
  - reduce (out_key, intermediate_value list) -> out_value list

Example: Word counting

- Map()
  - Input <filename, file text>
  - Parses file and emits <word, count> pairs
  - e.g., <”hello”, 1>
- Reduce()
  - Sums all the values for the same key and emits <word,TotalCount>
  - e.g., <”hello”, 1,1,1,1> => <”hello”, 4>


Input
Sorting and Shuffling
Output
How many cookies could
a good cook cook If a
good cook could cook cookies?
Map
Map
Map
how, 1
many, 1
cookies, 1
could, 1
a, 1
good, 1
cook, 1
cook, 1
if, 1
a, 1
good, 1
cook, 1
could, 1
cook ,1
cookies, 1
Reduce
Reduce
a, 1
a, 1
cookies, 1
cookies, 1
cook, 1
cook, 1
cook, 1
cook, 1
could, 1
could, 1
good, 1
good, 1
how, 1
many, 1
if, 1
a, 2
cookies, 2
cook, 4
cloud, 2
good, 2
how, 1
many, 1
if, 1

How does it work?

- From user:
  - Input/output files
  - **M**: number of map tasks >> # of worker machines for load balancing
  - **R**: number of reduce tasks
  - **W**: number of machines
  - Write map and reduce functions
  - Submit the job
- Requires no knowledge of parallel or distributed systems
- What about everything else?

Step 1: Data partition and distribution

- Split an input file into M pieces on distributed file system
  - Typically ~ 64 MB blocks
- Intermediate files created from map tasks are written to local disk
- Output files from reduce tasks are written to distributed file system

Step 2: Parallel computation

- Many copies of user program are started
- One instance becomes the Master
- Master finds idle machines and assigns them tasks
  - M map tasks
  - R reduce tasks

Locality

- Tries to utilize data localization by running map tasks on machines with data (the same data that map tasks need to process)
- map() task inputs are divided into 64 MB blocks: same size as Google File System chunks

Step 3: Map execution

- Map workers read in contents of corresponding input partition
- Perform user-defined map computation to create intermediate pairs

Step 4: Output intermediate data

- Periodically buffered output pairs written to local disk
  - Partitioned into R regions by a partitioning function
- Send locations of these buffered pairs on the local disk to the master, who is responsible for forwarding the locations to reduce workers

Partition function

- Partition on the intermediate key
  - Example partition function: hash(key) mod R
- Question: why do we need this?
- Example Scenario:
  - Want to do word counting on 10 documents
  - 5 map tasks, 2 reduce tasks

Step 5: Reduce execution

- The master notifies reduce workers
- Reduce workers iterate over **ordered** intermediate data
  - Data is **sorted** by the keys. Why is sorting needed?
  - For each unique key encountered, values are passed to user's reduce function
  - E.g., <key, [value1, value2, value3,..., valueN]>
  - Output of user's reduce function is written to output file on global file system
- When all tasks have completed, master wakes up user program

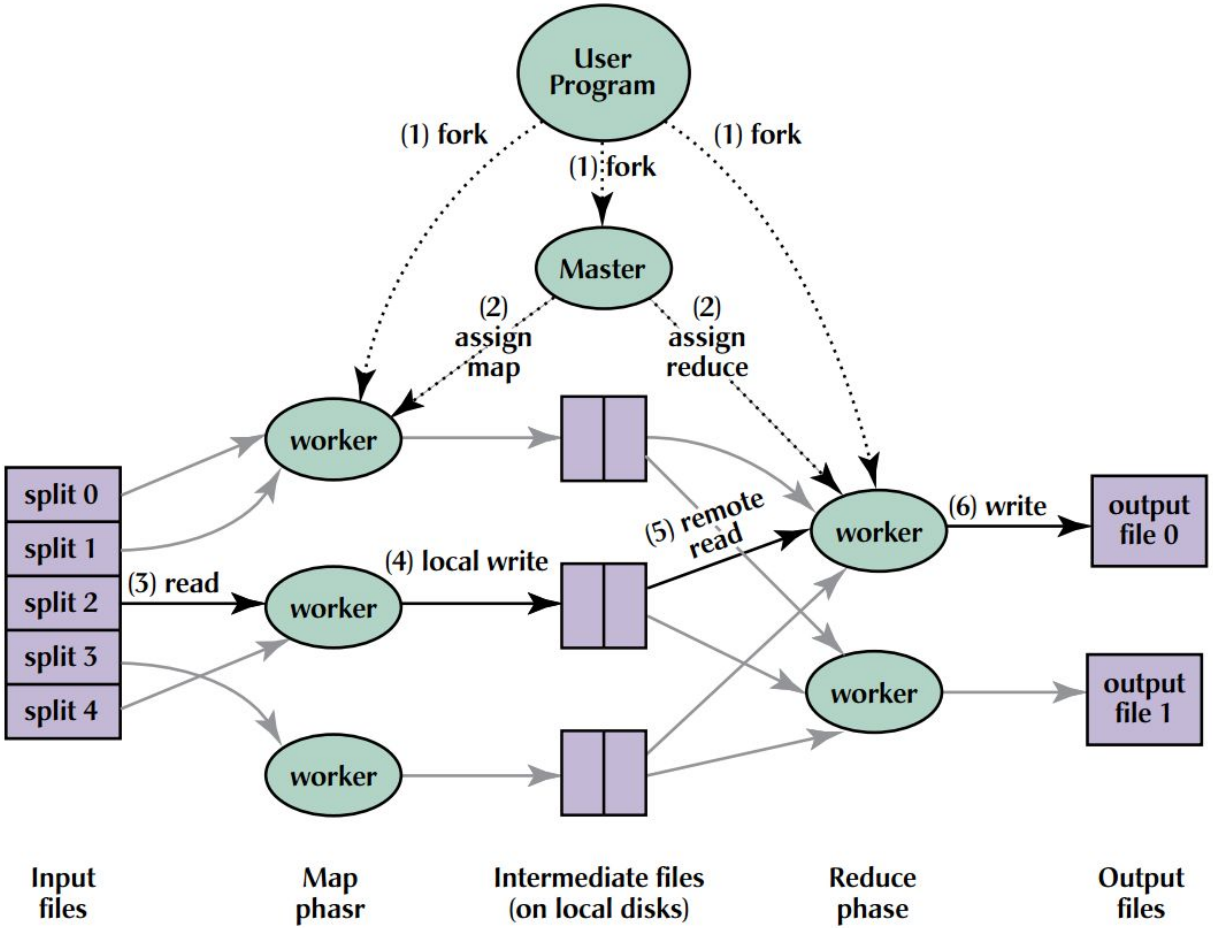


Fig. 1. Execution overview.

Observations

- No reduce can begin until map is complete
  - Why?
- Tasks scheduled based on location of data
- Master must communicate locations of intermediate files
- MapReduce library does most of the hard work

Fault tolerance

- Workers are periodically pinged by master
  - No response = failed worker
- Reassign tasks if worker is dead
- Input file blocks stored on multiple machines

Stragglers and backup tasks

- Problem: often some machines are late in their replies
  - slow disk, background competition, overloaded, bugs, etc
- Solution: backup tasks
  - when only few tasks left to execute, start backup tasks
  - a task completes when either primary or backup completes task
- Performance:
  - without backup, sort (->) takes 44% longer

Skipping bad records

The MapReduce library detects which records cause deterministic crashes

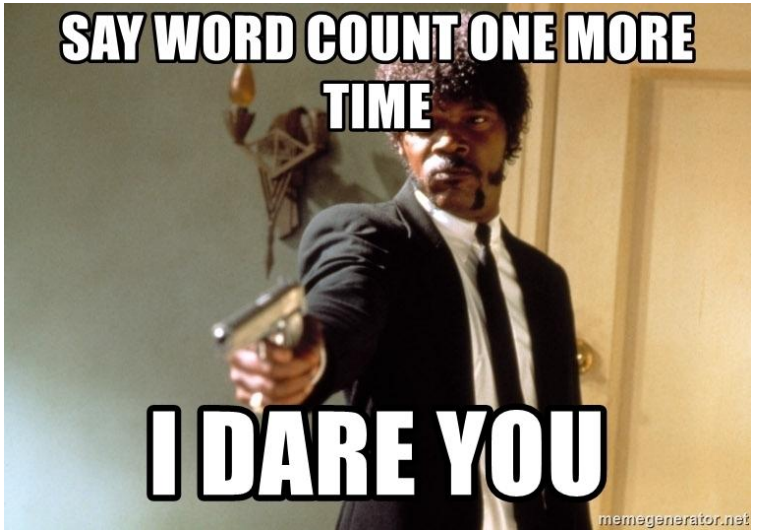
- Each worker process installs a signal handler that catches segmentation violations and bus errors
- Sends a “last gasp” signal to the MapReduce master
- Skip the record

Debugging

- Offers human readable status info on http server
- Users can see jobs completed, in-progress, processing rates, etc.

Conclusions

- Useful abstraction
- Very large variety of problems are easily expressible as MapReduce
- Allows user to focus on the problem without worrying about details
- Computer architecture not very important
    - Portable model



References

- Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51.1 (2008): 107-113.
- https://slideplayer.com/slide/1517884/
- https://slideplayer.com/slide/5229325/
- http://blog.enablecloud.com/2012/06/what-lies-at-core-of-hadoop.html