CIS 833 – Information Retrieval and Text Mining

Lecture 3

Introduction to MapReduce

September 1, 2015

Credits for slides: Lin, Hofmann, Mihalcea, Mobasher, Mooney, Schutze.

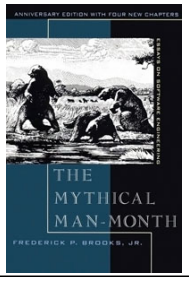
Required Reading

- MapReduce & Hadoop
  - **Data-Intensive Text Processing with MapReduce** (Lin and Dyer) http://www.umiacs.umd.edu/~jimmylin/book.html
    - Chapter 2, MapReduce
    - **MapReduce: Simplified Data Processing on Large Clusters** by Jeffrey Dean and Sanjay Ghemawat from Google Labs
- Next time:
  - **Data-Intensive Text Processing with MapReduce** (Lin and Dyer)
    - Chapter 3, MapReduce Algorithm Design

"Big Ideas"

- Scale "out", not "up"
  - Limits of symmetric multi-processing (SMP) machines and large shared-memory machines
- Move processing to the data
  - Clusters have limited bandwidth
- Process data sequentially, avoid random access
  - Seeks are expensive, disk throughput is reasonable
- Seamless scalability
  - From the mythical man-month to the tradeable machine-hour

"adding manpower to a late software project makes it later"



Typical Large-Data Problem

- Iterate over a large number of records *Map*
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results *Reduce*
- Generate final output

Key idea: provide a functional abstraction for these two operations

(Dean and Ghemawat, OSDI 2004)

Warm up: Word Count

- We have a large file of words, one word to a line
- Count the number of times each distinct word appears in the file
- Sample application: analyze web server logs to find popular URLs

Word Count (2)

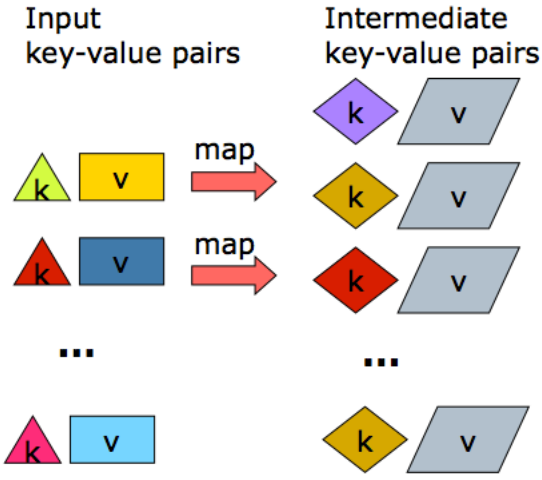
- Case 1: Entire file fits in memory
- Case 2: File too large for memory, but all <word, count> pairs fit in memory
- Case 3: File on disk, too many distinct words to fit in memory
  - `sort datafile | uniq -c`

Word Count (3)

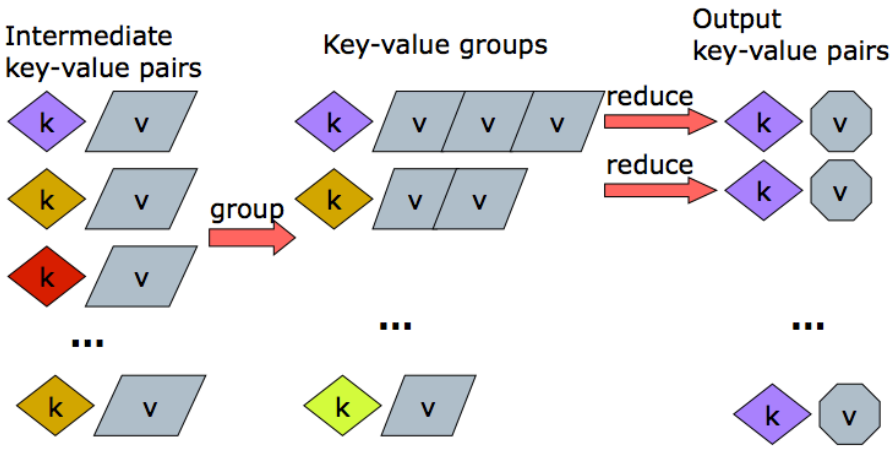
- To make it slightly harder, suppose we have a large corpus of documents
- Count the number of times each distinct word occurs in the corpus
  - `words(docs/*) | sort | uniq -c`

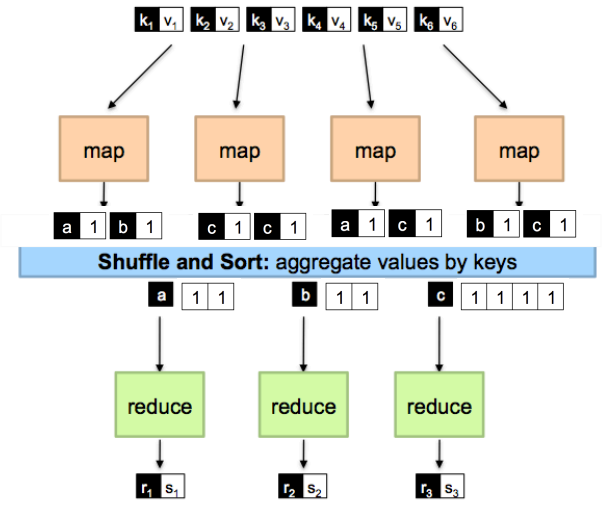
  where `words` takes a file and outputs the words in it, one to a line
- The above captures the essence of MapReduce
  - Great thing is it is naturally parallelizable

MapReduce: the Map Step



MapReduce: the Reduce Step





MapReduce

- Input: a set of key/value pairs
- User supplies two functions:
  - map(k,v) → list(k1,v1)
  - reduce(k1, list(v1)) → v2
- (k1,v1) is an intermediate key/value pair
- All values with the same key are sent to the same reducer
- Output is a set of (k1,v2) pairs

MapReduce

- Programmers specify the map and reduce functions

```
map(in_key, in_value) ->
          (out_key, intermediate_value) list

reduce(out_key, intermediate_value list) ->
          (out_key, out_value) list
```

Example: Count word occurrences

```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
    Emit(w, 1);

reduce(String output_key, Iterator<int>
  intermediate_values):
  // output_key: a word
  // intermediate_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += v;
 Emit(output_key, result);
```

Some MapReduce Terminology

- *Job* – A "full program" - an execution of a Mapper and Reducer across a data set
- *Task* – An execution of a Mapper or a Reducer on a slice of data
  - a.k.a. Task-In-Progress (TIP)
- *Task attempt* – A particular instance of an attempt to execute a task on a machine

Terminology Example

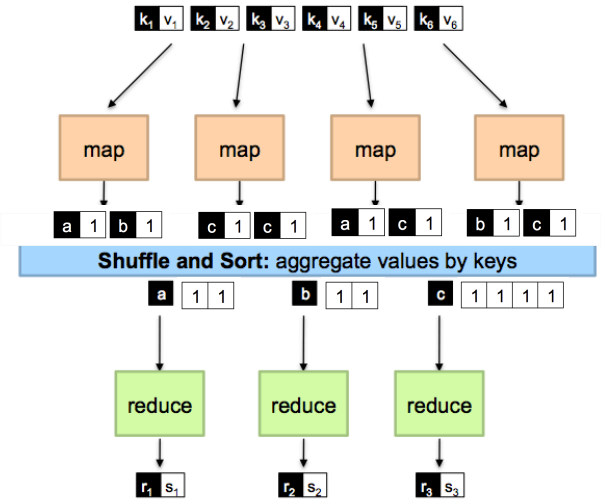
- Running "Word Count" across 20 files/ documents is one *job*
- 20 files to be mapped imply 20 *map tasks* + some number of *reduce tasks*
- At least 20 map *task attempts* will be performed… more if a machine crashes, etc.

MapReduce

- Programmers specify the map and reduce functions.
- The execution framework handles everything else…

- Not quite…usually, programmers also specify:

  combine (k1,list(v1)) →v2
  partition (k1, number of partitions) → partition for k1

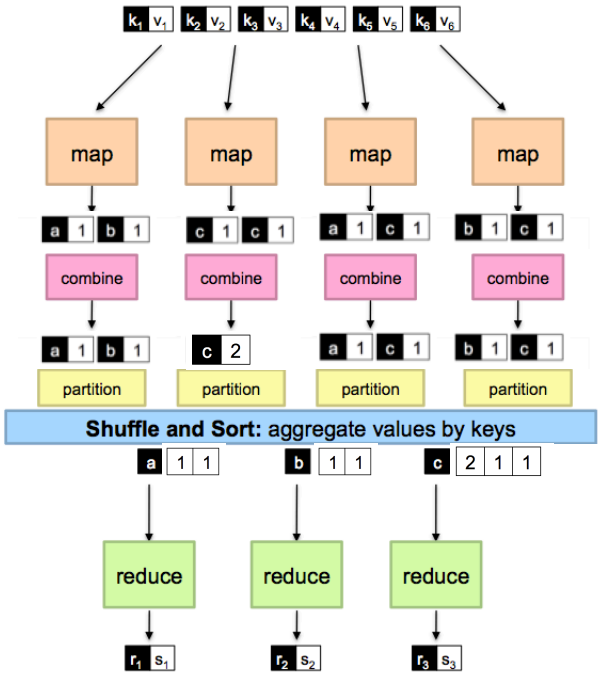


Combiners

- Often a map task will produce many pairs of the form (k,v1), (k,v2), … for the same key k
  - E.g., popular words in Word Count
- Can save network traffic (time) by pre-aggregating at mapper
- Mini-reducers that run in memory after the map phase
  - combine(k1, list(v1)) → v2
  - Usually same as reduce function
- Works only if reduce function is commutative and associative

Partition Function

- Divides up the key space for parallel reduce operations
- Inputs to map tasks are created by contiguous splits of input file
- For reduce, we need to ensure that records with the same intermediate key end up at the same worker
- System uses a default partition function e.g., hash(key) mod R
- Sometimes useful to override
- E.g., hash(hostname(URL)) mod R ensures URLs from a host end up in the same output file



Remember that…

- Barrier between map and reduce phases
  - But we can begin copying intermediate data earlier
- Keys arrive at each reducer in sorted order
  - No enforced ordering *across* reducers

MapReduce

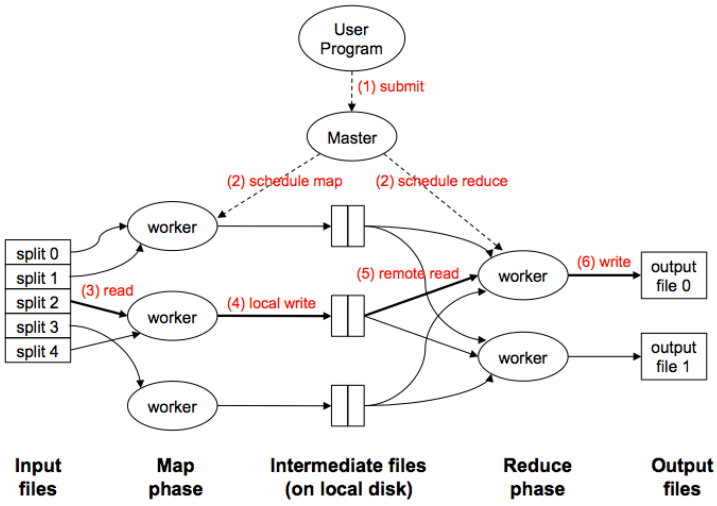
- The execution framework handles everything else…

What's "everything else"?

MapReduce "Runtime"

- Handles scheduling
  - Assigns map and reduce tasks to workers
  - Speculative execution ("stragglers")
- Handles "data distribution"
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed file system (DFS)

Distributed Execution Overview



Adapted from (Dean and Ghemawat, OSDI 2004)

Data flow

- Input, final output files are stored on a distributed file system
  - Scheduler tries to schedule map tasks "close" to physical storage location of input data
- Intermediate results are stored on local FS of map and reduce workers
- Output is often input to another map reduce job

Coordination

- Master data structures
  - Task status: (idle, in-progress, completed)
  - Idle tasks get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its R intermediate files, one for each reducer
  - Master pushes this info to reducers
- Master pings workers periodically to detect failures

Failures

- Map worker failure
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- Reduce worker failure
  - Only in-progress tasks are reset to idle
- Master failure
  - MapReduce job is aborted and client is notified

How many Map and Reduce jobs?

- M map tasks, R reduce tasks
- Rule of thumb:
  - Make M and R much larger than the number of nodes in cluster
  - One DFS chunk (split) per map is common
  - Improves dynamic load balancing and speeds recovery from worker failure
- Usually R is smaller than M, because output is spread across R files

Distributed File System

- Don't move data to workers… move workers to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- Why?
  - Not enough RAM to hold all the data in memory
  - Disk access is slow, but disk throughput is reasonable
- A distributed file system is the answer
  - GFS (Google File System) for Google's MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop