Lecture 36: MapReduce

[Adapted from slides by John DeNero and
http://research.google.com/archive/mapreduce-osdi04-slides]

Frameworks

MapReduce is a framework for batch processing of Big Data:

- *Framework:* A system used by programmers to build applications
- *Batch processing:* All the data is available at the outset and results aren't consumed until processing completes
- *Big Data:* A buzzword used to describe datasets so large that they reveal facts about the world via statistical analysis

The big ideas that underly MapReduce:

- Datasets are too big to be stored or analyzed on one machine.
- When using multiple machines, systems issues abound.
- *Pure functions* enable an abstraction barrier between data processing logic and distributed system administration.

Systems

Systems research enables the development of applications by defining and implementing abstractions:

- Operating systems provide a stable, consistent interface to unreliable, inconsistent hardware
- Networks provide a simple, robust data transfer interface to constantly evolving communications infrastructure
- Databases provide a declarative interface to software that stores and retrieves information efficiently
- Distributed systems provide a single-entity-level interface to a cluster of multiple machines

Unifying property of effective systems:

Hide complexity, but retain flexibility

Unix Pipes as a Framework

(Review) Unix embeds a framework for composing processes:

- Each process has a standard input stream (of characters) and a standard output stream (plus a standard error stream "on the side.")
- Programming languages provide functions to read and write to these streams, just as for ordinary files.
- In Python, these streams are called `sys.stdin`, `sys.stdout`, and `sys.stderr`.
- They are objects whose interface provides read and write operations and iterators.
- The OS allows one to string together (*compose*) sequences of programs into a pipeline, enabled by the common stream interface. E.g., (from lecture 9):

```
tr -c -s '[:alpha:]' '[\n*]' < FILE | sort | uniq -c | \
sort -n -r -k 1,1 | sed 20q
```

  prints the 20 most frequent words in *FILE*, with their counts.

MapReduce Idea

A MapReduce job takes a *mapper* program and a *reducer* program from a user and applies them to a set of data.

- *Map phase:* Apply a mapper function to inputs, emitting a set of intermediate key-value pairs.
- *Reduce phase:* For each distinct intermediate key, apply a reducer function to accumulate all the values with that key. Return a list of accumulated values for the key.

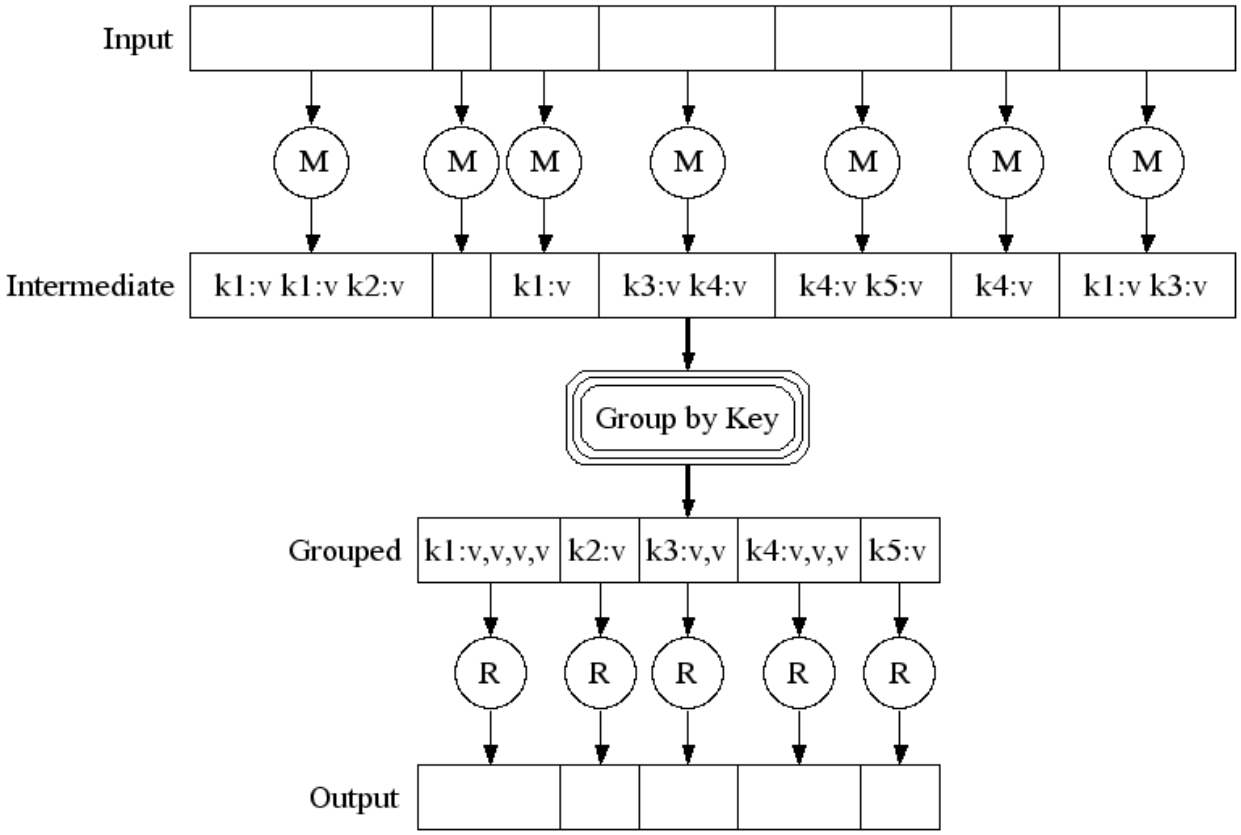
Example I: Counting Occurrences

Counting occurrences of words in a large collection of documents:

- Input to map operation: pairs `(name of document, text of document)`.
- Output from map operation: pairs `(word, 1)` (the 1 represents a count).
- Input to reduce operation: `(word, iterator over all counts for that word)`
- Output from reduce operation: sum of all counts for one word.
- (Could make things more efficient by having the map operation do some counting and return just one count for each distinct word).

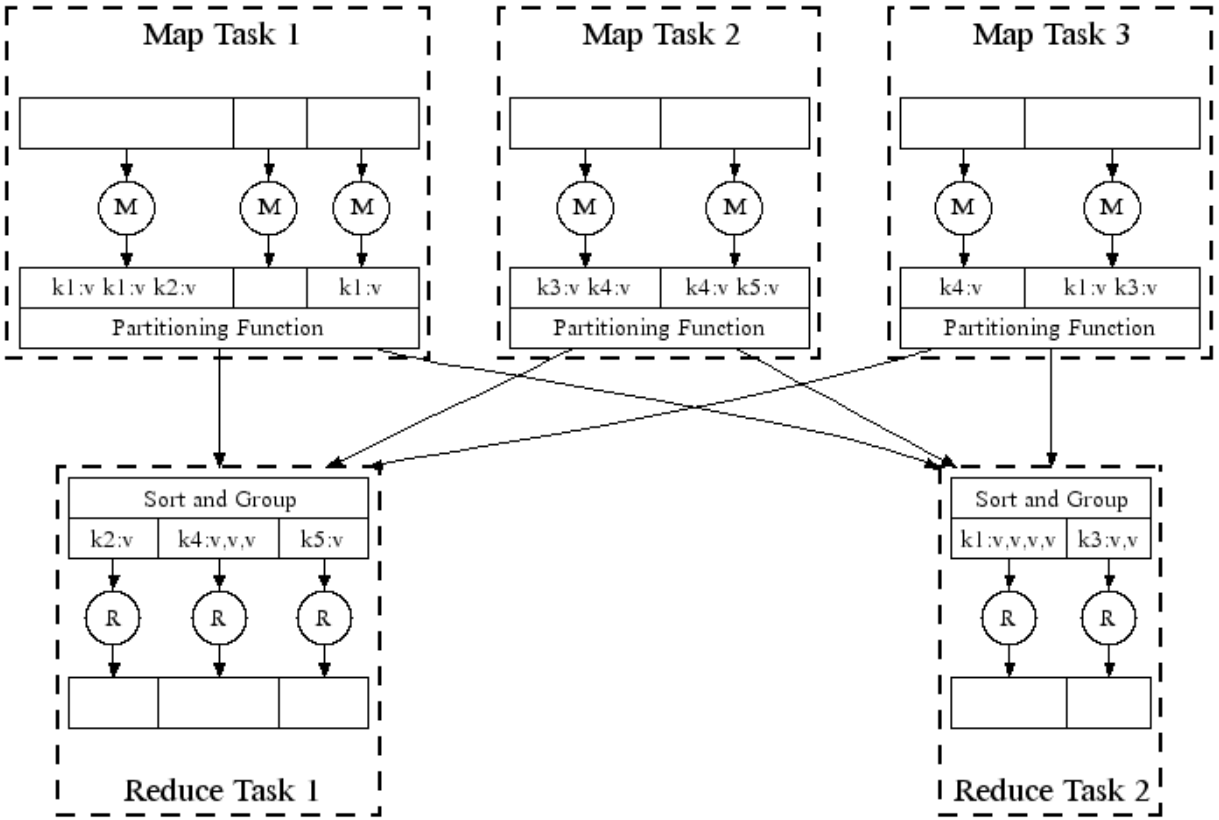
Abstract Execution Model

Execution



Parallel Execution

Parallel Execution



Example II: Distributed Grep

- Input to map: Pairs `(name of document, text of document)`.
- Output from map: pairs `(name of document, line matching target pattern)`
- Output from reduce: the list of matching lines from each document.
- (Reduce is trivial here; we're just using map).

Example III: Reverse Web-Link Graph

- Input to map: Pairs `(source URL, webpage content of URL)`
- Output from map: Pairs `(target URL, source URL)` for each hyperlink target on the input webpage.
- Output from reduce: `(target URL, list of source URLs)`.
- The work here is mostly in gathering up and sorting the results of map.

Inverted Index

- Input to map: Pairs `(document name, document contents)`.
- Output from map: Pairs `(word from document, document name)`
- Output from reduce: for each word, list of all documents it came from.

Scale

Way back in August 2004, MapReduce at Google processed this much data in using MapReduce:

Number of jobs	29,423	
Average job completion time	634	secs
Machine days used	79,186	days
Input data read	3,288	TB
Intermediate data produced	758	TB
Output data written	193	TB
Average worker machines per job	157	
Average worker deaths per job	1.2	
Average map tasks per job	3,351	
Average reduce tasks per job	55	
Unique map implementations	395	
Unique reduce implementations	269	
Unique map/reduce combinations	426	

What the Framework Provides

- *Fault tolerance:* A machine or hard drive might crash.
  - The MapReduce framework automatically re-runs failed tasks.
- *Speed:* Some machine might be slow because it's overloaded or failing.
  - The framework can run multiple copies of a task and keep the result of the one that finishes first.
- *Network locality:* Data transfer is expensive.
  - The framework tries to schedule map tasks on the machines that hold the data to be processed.
- *Monitoring:* Will my job finish before dinner?!?
  - The framework provides a web-based interface describing jobs.