• Want to:
  – Process lots of data ( > 1 TB)
  – Automatically parallelize across hundreds/thousands of CPUs
  – Have status and monitoring tools
  – Provide clean abstraction for programmers
  – Make this easy
“A simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.”

Dean and Ghermawat, “MapReduce: Simplified Data Processing on Large Clusters”, Google Inc.
**Typical Problem**

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

- **Reduce**

**Key idea:** provide an abstraction at the point of these two operations
• Process data using special `map()` and `reduce()` functions

• The `map()` function is called on every item in the input and emits a series of intermediate key/value pairs

• All values associated with a given key are grouped together

• The `reduce()` function is called on every unique key, and its value list, and emits a value that is added to the output
• Borrows from functional programming
• Users implement interface of two functions:
  
  - `map (in_key, in_value) -> (out_key, intermediate_value) list`

  - `reduce (out_key, intermediate_value list) -> out_value list`
Records from the data source (lines out of files, rows of a database, etc) are fed into the map function as key*value pairs: e.g., (filename, line).

map() produces one or more intermediate values along with an output key from the input.
After the map phase is over, all the intermediate values for a given output key are combined together into a list.

reduce() combines those intermediate values into one or more *final values* for that same output key.

(in practice, usually only one final value per key)
• Word frequency
MapReduce Examples

• Distributed grep
  – Map function emits <word, line_number> if word matches search criteria
  – Reduce function is the identity function

• URL access frequency
  – Map function processes web logs, emits <url, 1>
  – Reduce function sums values and emits <url, total>
MapReduce: Programming Model

Input

Map

Reduce

Output

MapReduce Framework

brown 1
cow 1
does 1
How 2
it 1
now 2
work 1

<How,1>
<now,1>
<brown,1>
<cow,1>
<How,1>
<does,1>
<it,1>
<work,1>
<now,1>
<brown,1>
<cow,1>
<does,1>
<it,1>
<work,1>

How now
Brown cow

How does
It work now
1. The user program, via the MapReduce library, shards the input data

* Shards are typically 16-64mb in size
2. The user program creates process copies distributed on a machine cluster. One copy will be the “Master” and the others will be worker threads.
3. The master distributes $M$ map and $R$ reduce tasks to idle workers.
   - $M ==$ number of shards
   - $R ==$ the intermediate key space is divided into $R$ parts
4. Each map-task worker reads assigned input shard and outputs intermediate key/value pairs.
   – Output buffered in RAM.
5. Each worker flushes intermediate values, partitioned into R regions, to disk and notifies the Master process.
6. Master process gives disk locations to an available reduce-task worker who reads all associated intermediate data.
7. Each reduce-task worker sorts its intermediate data. Calls the reduce function, passing in unique keys and associated key values. Reduce function output appended to reduce-task’s partition output file.
8. Master process wakes up user process when all tasks have completed. Output contained in R output files.
MapReduce Runtime System

1. Partitions input data
2. Schedules execution across a set of machines
3. Handles machine failure
4. Manages interprocess communication
• map() functions run in parallel, creating different intermediate values from different input data sets
• reduce() functions also run in parallel, each working on a different output key
• All values are processed independently
• Bottleneck: reduce phase can’t start until map phase is completely finished.
• Master program divvies up tasks based on location of data: tries to have map() tasks on same machine as physical file data, or at least same rack
• map() task inputs are divided into 64 MB blocks: same size as Google File System chunks
• Master detects worker failures
  – Re-executes completed & in-progress map() tasks
  – Re-executes in-progress reduce() tasks
• Master notices particular input key/values cause crashes in map(), and skips those values on re-execution.
  – Effect: Can work around bugs in third-party libraries!
• No reduce can start until map is complete:
  – A single slow disk controller can rate-limit the whole process
• Master redundantly executes “slow-moving” map tasks; uses results of first copy to finish
MapReduce has proven to be a useful abstraction
Greatly simplifies large-scale computations at Google
Functional programming paradigm can be applied to large-scale applications
Fun to use: focus on problem, let library deal w/ messy details
Greatly reduces parallel programming complexity
  – Reduces synchronization complexity
  – Automatically partitions data
  – Provides failure transparency
  – Handles load balancing
• Open source MapReduce implementation

• Uses
  – Hadoop Distributed Filesytem (HDFS)
    • [http://hadoop.apache.org/core/docs/current/hdfs_design.html](http://hadoop.apache.org/core/docs/current/hdfs_design.html)
  – Java
  – ssh