blue heeler
• 378,000 results in 0.17 seconds
  • including images and video

• communicates with 1000s of machines
  • web server
  • index servers
  • document servers
  • spell checker
  • ad system
  • news, book, images, maps, ...
How?

- Indexing

- Google has a copy of the web on its servers

- Google computes an index mapping search terms to URLs
  - blue -> { www.foo.com, www.bar.com, ... }
  - heeler -> { www.bar.com, ... }
How big is the web?
How big is the web?

- Many billions of documents
- One trillion URLs (2008)
- Google has a copy of the web on its servers
How many servers?
How many servers?

- Google has ~450,000 servers (2007)
- 20 megawatts (~12500 homes)
- $2M / month for electricity
- 20–100 petaflops
Indexing the web

• Want to index billions of documents stored on thousands of machines
• ... in a reasonable time
Problem

- Large amounts of raw data — petabytes
  - web documents
  - logs
  - indices
  - the web graph
  - web crawl metadata
  - queries
- More data than can fit on single machine
Solution
Solution

• Functional programming!
Functional programming

• Programming with pure functions
  • output of the function depends only on the input
  • no side effects (I/O, assign to shared variables)
• functions are values
  • can pass a function as argument to another function
  • can return a function from another function

• Functional languages promote functional programming
• But, can do functional programming in many languages
MapReduce

• Framework for distributed computing on large data sets

• Introduced by Google
  • Jeffrey Dean and Sanjay Ghemawat, OSDI 2004

• Many other implementations
  • for: C++, Python, Erlang,Scala, Java, ...

• Based on functional programming
• Users provide two functions: map and reduce
• 2008:
  • 100000 MapReduce jobs per day
  • Average of 400 machines per job
  • Average of 5–10 minutes per job
MapReduce

• Goal: reduce complexity of distributed computation

• Programmers just write map and reduce functions

• Runtime system takes care of:
  • parallelizing the computation
  • data partitioning
  • scheduling
  • failures
  • communication
  • executing on a cluster
Programming model

• Computation takes a set of key–value pairs
• Outputs a set of key–value pairs

• Map:
  • takes a key–value pair, generates intermediate pairs
• Reduce:
  • merges intermediate values associated with same key
Map

- \(\text{map}: (\text{Kin}, \text{Vin}) \Rightarrow \text{Seq}[(\text{Kout}, \text{Vtmp})]\)

- Takes a key–value pair of one type
- Returns a list of intermediate key–value pairs of another type

- Applied in parallel to every item in the input set
  - produces a list for each call
Reduce

• `reduce : (Kout, Seq[Vtmp]) => Seq[Vout]`

• Applied in parallel to each group
• Typically returns an option type (some value or none)
def map(name: String, doc: String): Seq[(String,Int)] = {
    doc.split(" ").map(w => (w, 1))
}

def reduce(word: String, counts: Seq[Int]): (String,Int) = {
    val count = counts.foldLeft(0)(_ + _)
    (word, count)
}
grav.txt,
“A screaming comes across the sky.”

1984.txt,
“It was a bright cold day in April, and the clocks were striking thirteen.”

neuro.txt,
“The sky above the port was the color of television, tuned to a dead channel.”
It was a bright cold day in April and the clocks were striking thirteen.

The screaming came across the sky above the port, where the color of the television was tuned to a dead channel.
Parallelism

• If each map operation is independent, all mappers can be done in parallel

• reduce requires all outputs of map with same key presented to same reducer at same time
Examples

- grep
  - map emits a line if it matches a pattern
  - reduce copies the inputs to output
Examples

- count url access frequency
- map outputs (url, 1) for each url in log
- reduce adds counts and emits (url, sum)
Examples

• reverse web-link graph
  • map outputs (target,source) pairs for each link to target in url named source
  • reduce cats the list of sources and returns (target, list(source))
Examples

- term-vector per host
  - map emits (hostname, term-vector)
  - reduce adds term vectors, discarding infrequent terms
• inverted index
  • map parses each doc and emits (word, url)
  • reduce sorts url list: (word, list(url))
• sort
  • map extracts key from each record, emits (key, record)
• reduce emits pairs unchanged
Implementation

- **map** distributed across multiple machines by partitioning input data into a set of M *splits*
  - input splits processed in parallel by different nodes
- **reduce** distributed by partitioning intermediate key space into R pieces using partitioning function hash(key) mod R
  - number of partitions R specified by user
Execution
reduce
tation
from

In this document, to implement MapReduce, the input is split into multiple files, and each file is read by a worker process. Each worker process runs the map function on the split data, which emits intermediate key-value pairs. These pairs are then passed to the reduce function, which aggregates and outputs the final result.

The process can be broken down as follows:

1. **Input files**: The input data is split into smaller files that are readable by the workers.
2. **Map phase**: Each worker reads a file, parses its content, and applies the map function to emit intermediate key-value pairs.
3. **Intermediate files**: The emitted pairs are written to intermediate files, which are stored on local disks.
4. **Reduce phase**: The master worker distributes the intermediate files to the workers, which then apply the reduce function to the pairs, emitting final key-value pairs.
5. **Output files**: The final pairs are written to output files, which are typically stored on a more durable storage system.

This process is designed to handle large-scale data processing efficiently, leveraging parallelism on multi-processor systems.
Map implementation

- map worker reads its input split
- parses $K_{in},V_{in}$ pairs out of the data and passes each to a user-defined map function
- buffers intermediate $K_{out},V_{tmp}$ pairs in memory
Partitioning

- periodically, buffered intermediate pairs written to local disk
- partitioned into $R$ returns by partitioning function
- locations of these buffered pairs passed back to the master
- master forwards to reduce worker
Reduce implementation

- reduce worker uses RPC to read data from local disks of map workers
- when reduce worker has read all intermediate data, it sorts it by key so all occurrences of the same key are grouped together
- reduce worker iterates over sorted intermediate data for each key
  - passes key and set of values to reduce function
- output of reduce function appended to final output file for this partition
Completion

- when all tasks complete, master notifies user program
- results available in R output files
Parallel execution

Map Task 1

Map Task 2

Map Task 3

Sort and Group

Reduce Task 1

Reduce Task 2
Limitations

• For maximum parallelism, you need the Maps and Reduces to be stateless, to not depend on any data generated in the same MapReduce job. You cannot control the order in which the maps run, or the reductions.
Limitations

• It is very inefficient if you are repeating similar searches again and again. A database with an index will always be faster than running an MR job over unindexed data.

• However, if that index needs to be regenerated whenever data is added, and data is being added continually, MR jobs may have an edge. That inefficiency can be measured in both CPU time and power consumed.
Implementation

- split into M pieces 16-64 MB each
- starts up many copies of the program
- one copy designated the **master**
- rest of the workers assigned work by the master
- M map tasks, R reduce tasks
- master picks idle workers and assigns one a map task or a reduce task
Task granularity and pipelining

- Many more map tasks than machines
  - minimizes time for fault recovery
  - can pipeline shuffling with map execution
  - better dynamic load balancing
- Often (2004): 200,000 map/5,000 reduce tasks with 2,000 machines
Fault tolerance

• Handled via re-execution

• on worker failure:
  • detect failure via periodic heartbeats
  • re-execute completed and in-progress map tasks
  • re-execute in-progress reduce tasks
  • task completion committed through master

• on master failure
  • don’t handle (could, but too rare to be worth it)

• robust:
  • once lost 1600-1800 machines, but finished fine
Locality

• 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
 Fault tolerance

• 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!
Skipping bad records

- On crash of a task, send record number to master
- If two failures for same record, drop the record
Optimizations

• No reduce can start until map is complete:
  • A single slow disk controller can rate-limit the whole process
Redundant execution

• Slow workers lengthen completion time
  • other jobs consuming resources on machine
  • bad disks with soft errors transfer data very slowly
  • weird things: machine misconfiguration, processor caches disabled

• Solution:
  • near end of phase, spawn backup copies of tasks
  • first one to finish “wins”

• Why is this safe?
Optimizations

- “Combiner” functions can run on same machine as a mapper
- Causes a mini-reduce phase to occur before the real reduce phase, to save bandwidth
Google indexer

- 2004
- 20 terabytes of data to index
- runs as a sequence of 5–10 MapReduce operations

Results:
- simpler, smaller indexing code
  - FT, distribution, parallelization hidden in library
- 3800 LOC -> 700 LOC
- as fast
- more fault tolerant
- easier to change: days vs. months
Conclusions

• 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
• Focus on problem, let library deal with messy details