Introduction to MapReduce

Juliana Freire

Some slides borrowed from Jimmy Lin, Jeff Ullman, Jerome Simeon, and Jure Leskovec
Single-node architecture

CPU

Memory

Disk

Data Analysis, Machine Learning, Statistics
How much data?

- Google processes 20 PB a day (2008)
- Wayback Machine has 3 PB + 100 TB/month (3/2009)
- Facebook has 2.5 PB of user data + 15 TB/day (4/2009)
- eBay has 6.5 PB of user data + 50 TB/day (5/2009)
- CERN’s LHC will generate 15 PB a year (??)
What to do with more data?

• “Machine learning algorithms really don’t matter, all that matters is the amount of data you have”, Banko and Brill, 2001
  – More data → higher accuracy
  – More data → accuracy of different algorithms converge
What to do with more data?

• Answering factoid questions
  – Pattern matching on the Web
  – Works amazingly well
  Who shot Abraham Lincoln? → X shot Abraham Lincoln

• Learning relations
  – Start with seed instances
  – Search for patterns on the Web
  – Using patterns to find more instances

Birthday-of(Mozart, 1756)
Birthday-of(Einstein, 1879)

Wolfgang Amadeus Mozart (1756 - 1791)
Einstein was born in 1879

PERSON (DATE – PERSON was born in DATE)
Motivation: Google Example

- 20+ billion web pages x 20KB = 400+ TB
- 1 computer reads 30-35 MB/sec from disk
  - ~4 months to read the web
- ~1,000 hard drives to store the web
- Takes even more to **do** something useful with the data!
- **A standard architecture for such problems is emerging**
  - Cluster of commodity Linux nodes
  - Commodity network (ethernet) to connect them
Cluster Architecture

1 Gbps between any pair of nodes in a rack

2-10 Gbps backbone between racks

Each rack contains 16-64 nodes

In 2011 it was guestimated that Google had 1M machines, [http://bit.ly/Shh0RC](http://bit.ly/Shh0RC)
Divide and Conquer

“Work”

\[ w_1 \]
\[ w_2 \]
\[ w_3 \]

“worker”

\[ r_1 \]
\[ r_2 \]
\[ r_3 \]

“Result”

Partition

Combine
Parallelization Challenges

• How do we assign work units to workers?
• What if we have more work units than workers?
• What if workers need to share partial results?
• How do we aggregate partial results?
• How do we know all the workers have finished?
• What if workers die?

What is the common theme of all of these problems?
Common Theme?

- Parallelization problems arise from:
  - Communication between workers (e.g., to exchange state)
  - Access to shared resources (e.g., data)
- Thus, we need a synchronization mechanism
Managing Multiple Workers

- Difficult because
  - We don’t know the order in which workers run
  - We don’t know when workers interrupt each other
  - We don’t know the order in which workers access shared data

- Thus, we need:
  - Semaphores (lock, unlock)
  - Conditional variables (wait, notify, broadcast)
  - Barriers

- Still, lots of problems:
  - Deadlock, livelock, race conditions...
  - Dining philosophers, sleeping barbers, cigarette smokers...

- Moral of the story: be careful!
Current Tools

- Programming models
  - Shared memory (pthreads)
  - Message passing (MPI)
- Design Patterns
  - Master-slaves
  - Producer-consumer flows
  - Shared work queues
Where the rubber meets the road

• Concurrency is difficult to reason about
• Concurrency is even more difficult to reason about
  – At the scale of datacenters (even across datacenters)
  – In the presence of failures
  – In terms of multiple interacting services
• Not to mention debugging…
• The reality:
  – Lots of one-off solutions, custom code
  – Write your own dedicated library, then program with it
  – Burden on the programmer to explicitly manage everything
What’s the point?

• It’s all about the right level of abstraction
  – The von Neumann architecture has served us well, but is no longer appropriate for the multi-core/cluster environment

• Hide system-level details from the developers
  – No more race conditions, lock contention, etc.

• Separating the what from how
  – Developer specifies the computation that needs to be performed
  – Execution framework (“runtime”) handles actual execution

The datacenter is the computer!
Cloud (Utility) Computing

• What?
  – Computing resources as a metered service ("pay as you go")
  – Ability to dynamically provision virtual machines

• Why?
  – Cost: capital vs. operating expenses
  – Scalability: "infinite" capacity
  – Elasticity: scale up or down on demand

• Does it make sense?
  – Benefits to cloud users
  – Business case for cloud providers

• MapReduce computational model
  abstracts many of the complexities

I think there is a world market for about five computers.
Big Ideas: Scale Out vs. Scale Up

• Scale up: small number of high-end servers
  – Symmetric multi-processing (SMP) machines, large shared memory
  ● Not cost-effective – cost of machines does not scale linearly; and no single SMP machine is big enough

• Scale out: Large number of commodity low-end servers is more effective for data-intensive applications
  – 8 128-core machines vs. 128 8-core machines

“low-end server platform is about 4 times more cost efficient than a high-end shared memory platform from the same vendor”, Barroso and Hölzle, 2009
Why commodity machines?

<table>
<thead>
<tr>
<th></th>
<th>HP INTEGRITY SUPERDOME-ITANUM2</th>
<th>HP PROLIANT ML350 G5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>64 sockets, 128 cores (dual-threaded), 1.6 GHz Itanium2, 12 MB last-level cache</td>
<td>1 socket, quad-core, 2.66 GHz X5355 CPU, 8 MB last-level cache</td>
</tr>
<tr>
<td>Memory</td>
<td>2,048 GB</td>
<td>24 GB</td>
</tr>
<tr>
<td>Disk storage</td>
<td>320,974 GB, 7,056 drives</td>
<td>3,961 GB, 105 drives</td>
</tr>
<tr>
<td>TPC-C price/performance</td>
<td>$2.93/tpmC</td>
<td>$0.73/tpmC</td>
</tr>
<tr>
<td>price/performance (server HW only)</td>
<td>$1.28/transactions per minute</td>
<td>$0.10/transactions per minute</td>
</tr>
<tr>
<td>Price/performance (server HW only) (no discounts)</td>
<td>$2.39/transactions per minute</td>
<td>$0.12/transactions per minute</td>
</tr>
</tbody>
</table>
Big Ideas: Failures are Common

• Suppose a cluster is built using machines with a mean-time between failures (MTBF) of 1000 days.
• For a 10,000 server cluster, there are on average 10 failures per day!
• MapReduce implementation cope with failures
  – Automatic task restarts
Big Ideas: Move Processing to Data

• Supercomputers often have processing nodes and storage nodes
  – Computationally expensive tasks
  – High-capacity interconnect to move data around

• Many data-intensive applications are not very processor-demanding
  – Data movement leads to a bottleneck in the network!
  – New idea: move processing to where the data reside

• In MapReduce, processors and storage are co-located
  – Leverage locality
Big Ideas: Avoid Random Access

• Disk seek times are determined by mechanical factors
  – Read heads can only move so fast and platters can only spin so rapidly
Big Ideas: Avoid Random Access

Example:

- 1 TB database containing $10^{10}$ 100 byte records
- Random access: each update takes ~30ms (seek, read, write)
  - Updating 1% of the records takes ~35 days
- Sequential access: 100MB/s throughput
  - Reading the whole database and rewriting all the records, takes 5.6 hours

- MapReduce was designed for batch processing
  --- organize computations into long streaming operations
Big Ideas: Abstract System-Level Details

• Developing distributed software is very challenging
  – Manage multiple threads, processes, machines
  – Concurrency control and synchronization
  – Race conditions, deadlocks – a nightmare to debug!

• MapReduce isolates developers from these details
  – Programmer defines *what* computations are to be performed
  – MapReduce execution framework takes care of *how* the computations are carried out
MapReduce
Typical Large-Data Problem

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

Key idea: provide a functional abstraction for these two operations

(Dean and Ghemawat, OSDI 2004)
Map and Reduce

• The idea of Map, and Reduce is 40+ year old
  – Present in all Functional Programming Languages.
  – See, e.g., APL, Lisp and ML
• Alternate names for Map: Apply-All
• Higher Order Functions
  – take function definitions as arguments, or
  – return a function as output
• Map and Reduce are higher-order functions.
Map Examples in Haskell

- map (+1) [1,2,3,4,5]  
  == [2, 3, 4, 5, 6]

- map (toLower) "abcDEFG12!@#"  
  == "abcdefg12!@#"

- map (`mod` 3) [1..10]  
  == [1, 2, 0, 1, 2, 0, 1, 2, 0, 1]
Fold-Left in Haskell

Definition
- \( \text{foldl} \ f \ z \ [] = z \)
- \( \text{foldl} \ f \ z \ (x:xs) = \text{foldl} \ f \ (f \ z \ x) \ x s \)

Examples
- \( \text{foldl} \ (+) \ 0 \ [1..5] = 15 \)
- \( \text{foldl} \ (+) \ 10 \ [1..5] = 25 \)
- \( \text{foldl} \ (\text{div}) \ 7 \ [34, 56, 12, 4, 23] = 0 \)
MapReduce

• Programmers specify two functions:
  \[
  \text{map} \ (k, v) \rightarrow <k', v'>^*
  \]
  \[
  \text{reduce} \ (k', v') \rightarrow <k', v'>^*
  \]
  – All values with the same key are sent to the same reducer

• The execution framework handles everything else…
“Hello World”: Word Count

Map(String docid, String text):
   for each word w in text:
      Emit(w, 1);

Reduce(String term, Iterator<Int> values):
   int sum = 0;
   for each v in values:
      sum += v;
      Emit(term, value);
Word Counting with MapReduce

M₁

Documents

Doc1
Financial, IMF, Economics, Crisis

Doc2
Financial, IMF, Crisis

M₂

Documents

Doc3
Economics, Harry

Doc4
Financial, Harry, Potter, Film

Doc5
Crisis, Harry, Potter

© Kyuseok Shim (VLDB 2012 TUTORIAL)
Word Counting with MapReduce

Before reduce functions are called, for each distinct key, the list of its values is generated.
MapReduce

Group by is applied to intermediate keys

Shuffle and Sort: aggregate values by keys

Output written to DFS
MapReduce “Runtime”

- Handles scheduling
  - Assigns workers to map and reduce tasks
- 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 FS (later)
MapReduce

- Programmers specify two functions:
  - `map (k, v) → <k’, v’>*
  - `reduce (k’, v’) → <k’, v’>*
    - All values with the same key are reduced together
- Mappers and reducers can specify arbitrary computations
  - Be careful with access to external resources!
- The execution framework handles everything else…
- Not quite…usually, programmers also specify:
  - `partition (k’, number of partitions) → partition for k’
    - Often a simple hash of the key, e.g., hash(k’) mod n
    - Divides up key space for parallel reduce operations
  - `combine (k’, v’) → <k’, v’>*
    - Mini-reducers that run in memory after the map phase
    - Used as an optimization to reduce network traffic
Before reduce functions are called, for each distinct key, the list of its values is generated.
MapReduce: The Complete Picture

Shuffle and Sort: aggregate values by keys

reduce

r_1 s_1

r_2 s_2

r_3 s_3
Two more details…

• 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 can refer to…

- The programming model
- The execution framework (aka “runtime”)
- The specific implementation

Usage is usually clear from context!
MapReduce Implementations

• Google has a proprietary implementation in C++
  – Bindings in Java, Python

• Hadoop is an open-source implementation in Java
  – Development led by Yahoo, used in production
  – Now an Apache project
  – Rapidly expanding software ecosystem

• Lots of custom research implementations
  – For GPUs, cell processors, etc.
Adapted from (Dean and Ghemawat, OSDI 2004)
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
GFS: Assumptions

• Commodity hardware over “exotic” hardware
  – Scale “out”, not “up”

• High component failure rates
  – Inexpensive commodity components fail all the time

• “Modest” number of huge files
  – Multi-gigabyte files are common, if not encouraged

• Files are write-once, mostly appended to
  – Perhaps concurrently

• Large streaming reads over random access
  – High sustained throughput over low latency
GFS: Design Decisions

- Files stored as chunks
  - Fixed size (e.g., 64MB)
- Reliability through replication
  - Each chunk replicated across 3+ chunkservers
- Single master to coordinate access, keep metadata
  - Simple centralized management
- No data caching
  - Little benefit due to large datasets, streaming reads
- Simplify the API
  - Push some of the issues onto the client (e.g., data layout)

HDFS = GFS clone (same basic ideas)
From GFS to HDFS

• Terminology differences:
  – GFS master = Hadoop namenode
  – GFS chunkservers = Hadoop datanodes

• Functional differences:
  – No file appends in HDFS (planned feature)
  – HDFS performance is (likely) slower

For the most part, we’ll use the Hadoop terminology…
HDFS Architecture

- **HDFS namenode**:
  - File namespace
  - /foo/bar
  - block 3df2

- **HDFS datanode**:
  - Linux file system
  - Block data

- **Application**
  - HDFS Client

- **Instructions to datanode**
  - (block id, byte range)
  - (file name, block id)

- **Datanode state**
Namendoe Responsibilities

• Managing the file system namespace:
  – Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc.

• Coordinating file operations:
  – Directs clients to datanodes for reads and writes
  – No data is moved through the namenode

• Maintaining overall health:
  – Periodic communication with the datanodes
  – Block re-replication and rebalancing
  – Garbage collection
MapReduce Architecture

- namenode
- datanode daemon
- Linux file system
- slave node
- tasktracker
- datanode daemon
- job submission node
- jobtracker
Fault Recovery

• Workers are pinged by master periodically
  – Non-responsive workers are marked as failed
  – All tasks in-progress or completed by failed worker become eligible for rescheduling

• Master could periodically checkpoint
  – Current implementations abort on master failure
Component Overview
• Open source Java
• Scale
  • Thousands of nodes and
  • petabytes of data
Hadoop

- MapReduce and Distributed File System framework for large commodity clusters
- Master/Slave relationship
  - JobTracker handles all scheduling & data flow between TaskTrackers
  - TaskTracker handles all worker tasks on a node
  - Individual worker task runs map or reduce operation
- Integrates with HDFS for data locality
Hadoop Supported File Systems

• HDFS: Hadoop's own file system.
• Amazon S3 file system
  – Targeted at clusters hosted on the Amazon Elastic Compute Cloud server-on-demand infrastructure
  – Not rack-aware
• CloudStore
  – previously Kosmos Distributed File System
  – like HDFS, this is rack-aware.
FTP Filesystem
  – stored on remote FTP servers.
Read-only HTTP and HTTPS file systems.
"Rack awareness"

- Optimization which takes into account the geographic clustering of servers
- Network traffic between servers in different geographic clusters is minimized.
Goals of HDFS

Very Large Distributed File System
  – 10K nodes, 100 million files, 10 PB

Assumes Commodity Hardware
  – Files are replicated to handle hardware failure
  – Detect failures and recovers from them

Optimized for Batch Processing
  – Data locations exposed so that computations can move to where data resides
  – Provides very high aggregate bandwidth

User Space, runs on heterogeneous OS
HDFS: Hadoop DFS

• Designed to scale to petabytes of storage, and run on top of the file systems of the underlying OS.
• Master (“NameNode”) handles replication, deletion, creation
• Slave (“DataNode”) handles data retrieval
• Files stored in many blocks
  – Each block has a block Id
  – Block Id associated with several nodes hostname:port (depending on level of replication)
Distributed File System

Single Namespace for entire cluster

Data Coherency

– Write-once-read-many access model
– Client can only append to existing files

Files are broken up into blocks

– Typically 128 MB block size
– Each block replicated on multiple DataNodes

Intelligent Client

– Client can find location of blocks
– Client accesses data directly from DataNode
NameNode Metadata

Meta-data in Memory
- The entire metadata is in main memory
- No demand paging of meta-data

Types of Metadata
- List of files
- List of Blocks for each file
- List of DataNodes for each block
- File attributes, e.g creation time, replication factor

A Transaction Log
- Records file creations, file deletions. etc
DataNode

A Block Server
- Stores data in the local file system (e.g. ext3)
- Stores meta-data of a block (e.g. CRC)
- Serves data and meta-data to Clients

Block Report
- Periodically sends a report of all existing blocks to the NameNode

Facilitates Pipelining of Data
- Forwards data to other specified DataNodes
Block Placement

Current Strategy
-- One replica on local node
-- Other replicas on remote racks
-- Additional replicas are randomly placed

Clients read from neapont replica
Would like to make this policy pluggable
Data Correctness

Use Checksums to validate data
   – Use CRC32

File Creation
   – Client computes checksum per 512 byte
   – DataNode stores the checksum

File access
   – Client retrieves the data and checksum from DataNode
   – If Validation fails, Client tries other replicas
NameNode Failure

A single point of failure
Transaction Log stored in multiple directories
  – A directory on the local file system
  – A directory on a remote file system (NFS/CIFS)
Data Pipelining

Client retrieves a list of DataNodes on which to place replicas of a block
Client writes block to the first DataNode
The first DataNode forwards the data to the next DataNode in the Pipeline
When all replicas are written, the Client moves on to write the next block in file
Rebalancer

**Goal:** % disk full on DataNodes should be similar
Usually run when new DataNodes are added
Cluster is online when Rebalancer is active
Rebalancer is throttled to avoid network congestion
Command line tool
Hadoop vs. ‘MapReduce’

• MapReduce is also the name of a framework developed by Google
• Hadoop was initially developed by Yahoo and now part of the Apache group.
• Hadoop was inspired by Google's MapReduce and Google File System (GFS) papers.
## MapReduce v. Hadoop

<table>
<thead>
<tr>
<th></th>
<th>MapReduce</th>
<th>Hadoop</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Org</strong></td>
<td>Google</td>
<td>Yahoo/Apache</td>
</tr>
<tr>
<td><strong>Impl</strong></td>
<td>C++</td>
<td>Java</td>
</tr>
<tr>
<td><strong>Distributed File Sys</strong></td>
<td>GFS</td>
<td>HDFS</td>
</tr>
<tr>
<td><strong>Data Base</strong></td>
<td>Bigtable</td>
<td>HBase</td>
</tr>
<tr>
<td><strong>Distributed lock mgr</strong></td>
<td>Chubby</td>
<td>ZooKeeper</td>
</tr>
</tbody>
</table>
References

• http://hadoop.apache.org/
• David DeWitt, Michael Stonebraker, "MapReduce: A major step backwards“, craig-henderson.blogspot.com
• http://scienceblogs.com/goodmath/2008/01/databases_are_hammers_mapreduc.php