Processing Platforms
# Big Data Technology & Analytics

## Data Ingestion
- ETL, Distcp, Kafka, OpenRefine, ...

## Storage Systems
- HDFS, RDBMS, Column Stores, Graph Databases

## Batch Processing Platforms
- MapReduce, SparkSQL, BigQuery, Hive, Cypher, ...

## Stream Processing Platforms
- Storm, Spark, ..

## Query & Exploration
- SQL, Search, Cypher, ...

## Data Definition
- SQL DDL, Avro, Protobuf, CSV

## Data Serving
- BI, Cubes, RDBMS, Key-value Stores, Tableau, ...

## Computing Platforms
- Distributed Commodity, Clustered High-Performance, Single Node
Computing Platforms
Distributed Commodity, Clustered High-Performance, Single Node

Data Ingestion
ETL, Distcp, Kafka, OpenRefine, ...

Query & Exploration
SQL, Search, Cypher, ...

Stream Processing Platforms
Storm, Spark,..

Batch Processing Platforms
MapReduce, SparkSQL, BigQuery, Hive, Cypher, ...

Data Definition
SQL DDL, Avro, Protobuf, CSV

Storage Systems
HDFS, RDBMS, Column Stores, Graph Databases

Data Serving
BI, Cubes, RDBMS, Key-value Stores, Tableau, ...

Big Data Technology & Analytics
Big Data Architectures
Elements of DAV Architecture

1. Collect/Ingest Data
2. Store Raw Data
3. Clean Transform Data
4. Query Data
5. Compute, Join, Aggregate data
6. Analyze Report
Difference in Approach

Traditional Approach
Structured & Repeatable Analysis

Business Users
Determine what question to ask

IT
Structures the data to answer that question
Monthly sales reports
Profitability analysis
Customer surveys

Big Data Approach
Iterative & Exploratory Analysis

IT
Delivers a platform to enable creative discovery

Business
Explores what questions could be asked
- Brand sentiment
- Product strategy
- Maximum asset utilization
- Preventative care

Notice the difference!
Traditional Business Warehouse

New transactions and facts

Transaction data and Fact based data model

Analytics Data
Big Data Analytics Architecture

Example:
Lambda Architecture

Other examples:
Kappa Architecture
Netflix Architecture
Ingest

Key points:
- Keep your data in Kafka and HDFS
- Low latency processing as a stream
- Re-process and batch processing in Hadoop
Key points:
- Keep your data in Kafka
- Treat everything as a Stream
- Re-process stream by resetting offset
- Advantage: simplified architecture, everything is a stream
Netflix Architecture

ETL: Enterprise Scheduler, Ursula, Aegisthus
Viz tools
Web UIs

Hadoop Platform as a Service (Genie)
Job Execution
Resource Configuration & Management

Traditional Gateways
CLIs

Tools
Hadoop (EMR) Clusters

Cloud Data Warehouse

Analytics Data

New transactions and facts
Transaction data and Fact based data model
MapReduce, Spark, BigQuery, ...
Processing Platforms

- **Batch Processing**
  - Google GFS/MapReduce (2003)
  - Apache Hadoop HDFS/MapReduce (2004)

- **SQL**
  - BigQuery (based on Google Dremel, 2010)
  - Apache Hive (HiveQL) (2012)

- **Streaming Data**

- **Unified Engine (Streaming, SQL, Batch, ML)**
  - Apache Spark (2012)
Map-Reduce and the New Software Stack

Mining of Massive Datasets
Jure Leskovec, Anand Rajaraman, Jeff Ullman
Stanford University
http://www.mmds.org
Single Node Architecture

Machine Learning, Statistics

“Classical” Data Mining
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!
Today, a standard architecture for such problems is emerging:
  Cluster of commodity Linux nodes
  Commodity network (ethernet) to connect them
Cluster Architecture

Each rack contains 16-64 nodes

In 2011 it was guestimated that Google had 1M machines, [http://bit.ly/Shh0RO](http://bit.ly/Shh0RO)
Large-scale computing for data mining problems on commodity hardware

Challenges:

- How do you distribute computation?
- How can we make it easy to write distributed programs?
- Machines fail:
  - One server may stay up 3 years (1,000 days)
  - If you have 1,000 servers, expect to loose 1/day
  - People estimated Google had ~1M machines in 2011
    - 1,000 machines fail every day!
Idea and Solution

- **Issue:** Copying data over a network takes time
- **Idea:**
  - Store files multiple times for reliability
  - Bring computation close to the data
- **Storage Infrastructure:** Distributed File system
  - Google: GFS. Hadoop: HDFS
- **Programming Model:** Map-Reduce
  - Google’s computational/data manipulation model
  - Elegant way to work with big data
Problem:
- If nodes fail, how to store data persistently?

Answer:
- Distributed File System:
  - Provides global file namespace
  - Google GFS; Hadoop HDFS;

Typical usage pattern
- Huge files (100s of GB to TB)
- Data *reads* and *appends* are common
- Data is rarely *updated* in place
Distributed File System

- **Chunk servers**
  - File is split into contiguous chunks
  - Typically each chunk is 16-64MB
  - Each chunk replicated (usually 3x)
  - Try to keep replicas in different racks

- **Master node**
  - a.k.a. Name Node in Hadoop’s HDFS
  - Stores metadata about where files are stored
  - Might be replicated

- **Client library for file access**
  - Talks to master to find chunk servers
  - Connects directly to chunk servers to access data
Distributed File System

- Reliable distributed file system
- Data kept in “chunks” spread across machines
- Each chunk replicated on different machines
  - Seamless recovery from disk or machine failure

Bring computation directly to the data!

Chunk servers also serve as compute servers
Warm-up task:

- We have a huge text document

- Count the number of times each distinct word appears in the file

Sample application:

- Analyze web server logs to find popular URLs
Case 1:
- File too large for memory, but all \(<\text{word, count}>\) pairs fit in memory

Case 2:
- Count occurrences of words:
  - \texttt{words(doc.txt) | sort | uniq -c}
    - where \texttt{words} takes a file and outputs the words in it, one per a line
- Case 2 captures the essence of \texttt{MapReduce}
  - Great thing is that it is naturally parallelizable
MapReduce: Overview

- Sequentially read a lot of data
- **Map**: Extract something you care about
- **Group by key**: Sort and Shuffle
- **Reduce**: Aggregate, summarize, filter or transform
- Write the result

Outline stays the same, **Map** and **Reduce** steps change to fit the problem
MapReduce: The Map Step

Input key-value pairs

Intermediate key-value pairs

Map

...
MapReduce: The **Reduce** Step

- **Intermediate key-value pairs**
- **Group by key**
- **Key-value groups**
- **Output key-value pairs**

reduce
More Specifically

- **Input**: a set of key-value pairs
- Programmer specifies two methods:
  - **Map**($k, v$) $\rightarrow$ $<k', v'>^*$
    - Takes a key-value pair and outputs a set of key-value pairs
    - E.g., key is the filename, value is a single line in the file
    - There is one Map call for every $(k,v)$ pair
  - **Reduce**($k', <v'>^*$) $\rightarrow$ $<k', v''>^*$
    - All values $v'$ with same key $k'$ are reduced together and processed in $v'$ order
    - There is one Reduce function call per unique key $k'$
The crew of the space shuttle Endeavor recently returned to Earth as ambassadors, harbingers of a new era of space exploration. Scientists at NASA are saying that the recent assembly of the Dextre robot is the first step in a long-term space-based man/mae partnership. "The work we're doing now -- the robotics we're doing -- is what we're going to need.....................
map(key, value):
  // key: document name
  // value: text of the document
  for each word w in value:
    emit(w, 1)

reduce(key, values):
  // key: a word
  // value: an iterator over counts
  result = 0
  for each count v in values:
    result += v
  emit(key, result)
Map-Reduce: Environment

Map-Reduce environment takes care of:

- **Partitioning** the input data
- **Scheduling** the program’s execution across a set of machines
- Performing the **group by key** step
- Handling machine **failures**
- Managing required inter-machine communication
Map-Reduce: A diagram

**MAP:**
Read input and produces a set of key-value pairs

**Group by key:**
Collect all pairs with same key (Hash merge, Shuffle, Sort, Partition)

**Reduce:**
Collect all values belonging to the key and output
All phases are distributed with many tasks doing the work
Programmer specifies:
- Map and Reduce and input files

Workflow:
- Read inputs as a set of key-value-pairs
- **Map** transforms input kv-pairs into a new set of k'v'-pairs
- Sorts & Shuffles the k'v'-pairs to output nodes
- All k’v’-pairs with a given k’ are sent to the same **reduce**
- **Reduce** processes all k'v'-pairs grouped by key into new k''v''-pairs
- Write the resulting pairs to files

All phases are distributed with many tasks doing the work
Input and final output are stored on a distributed file system (FS):
- 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 MapReduce task
**Coordination: Master**

- **Master node takes care of coordination:**
  - **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**
Dealing with 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
  - Reduce task is restarted
- **Master failure**
  - MapReduce task is aborted and client is notified
How many Map and Reduce jobs?

- $M$ map tasks, $R$ reduce tasks
- Rule of a thumb:
  - Make $M$ much larger than the number of nodes in the cluster
  - One DFS chunk per map is common
  - Improves dynamic load balancing and speeds up recovery from worker failures
- Usually $R$ is smaller than $M$
  - Because output is spread across $R$ files
**Fine granularity tasks:** map tasks >> machines
- Minimizes time for fault recovery
- Can do pipeline shuffling with map execution
- Better dynamic load balancing
Refinements: Backup Tasks

- **Problem**
  - Slow workers significantly lengthen the job completion time:
    - Other jobs on the machine
    - Bad disks
    - Weird things

- **Solution**
  - Near end of phase, copy and run poorly performing tasks (*stragglers*) on another machine
    - Called *speculative execution* (tasks called “backup tasks”)
    - Whichever copy finishes first “wins”

- **Effect**
  - Dramatically shortens job completion time
Refinement: Combiners

- Often a Map task will produce many pairs of the form \((k,v_1), (k,v_2), \ldots\) for the same key \(k\)
  - E.g., popular words in the word count example
- Can save network time by pre-aggregating values in the mapper:
  - \(\text{combine}(k, \text{list}(v_1)) \Rightarrow v_2\)
  - Combiner is usually same as the reduce function
- Works only if reduce function is \text{commutative} and \text{associative}
Back to our word counting example:

- Combiner combines the values of all keys of a single mapper (single machine):
  - Much less data needs to be copied and shuffled!
Want to control how keys get partitioned

- Inputs to map tasks are created by contiguous splits of input file
- Reduce needs to ensure that records with the same intermediate key end up at the same worker

System uses a default partition function:

- hash(key) mod $R$

Sometimes useful to override the hash function:

- E.g., hash(hostname(URL)) mod $R$ ensures URLs from a host end up in the same output file
Problems Suited for Map-Reduce
Example: Host size

- Suppose we have a large web corpus
- For each host, find the total number of bytes
  - That is, the sum of the page sizes for all URLs from that particular host

Other examples:
- Link analysis and graph processing
- Machine Learning algorithms
Example: Language Model

- **Statistical machine translation:**
  - Need to count number of times every 5-word sequence occurs in a large corpus of documents

- **Very easy with MapReduce:**
  - **Map:**
    - Extract (5-word sequence, count) from document
  - **Reduce:**
    - Combine the counts
Example: Join By Map-Reduce

- Compute the natural join $R(A,B) \bowtie S(B,C)$
- $R$ and $S$ are each stored in files
- Tuples are pairs $(a,b)$ or $(b,c)$
Map-Reduce Join

- Use a hash function $h$ from $B$-values to $1...k$
- **A Map process turns:**
  - Each input tuple $R(a,b)$ into key-value pair $(b,(a,R))$
  - Each input tuple $S(b,c)$ into $(b,(c,S))$
- **Map processes** send each key-value pair with key $b$ to Reduce process $h(b)$
  - Hadoop does this automatically; just tell it what $k$ is.
- Each **Reduce process** matches all the pairs $(b,(a,R))$ with all $(b,(c,S))$ and outputs $(a,b,c)$. 

Cost Measures for Algorithms

- In MapReduce we quantify the cost of an algorithm using:
  1. *Communication cost* = total I/O of all processes
  2. *Elapsed communication cost* = max of I/O along any path
  3. *(Elapsed) computation cost* analogous, but count only running time of processes

Note that here the big-O notation is not the most useful (adding more machines is always an option)
For a map-reduce algorithm:

- **Communication cost** = input file size + 2 \times (sum of the sizes of all files passed from Map processes to Reduce processes) + the sum of the output sizes of the Reduce processes.

- **Elapsed communication cost** is the sum of the largest input + output for any map process, plus the same for any reduce process.
What Cost Measures Mean

- Either the I/O (communication) or processing (computation) cost dominates
  - Ignore one or the other

- Total cost tells what you pay in rent from your friendly neighborhood cloud

- Elapsed cost is wall-clock time using parallelism
Cost of Map-Reduce Join

- **Total communication cost**
  \[ O(|R| + |S| + |R \bowtie S|) \]

- **Elapsed communication cost**
  \[ = O(s) \]

  - We’re going to pick \( k \) and the number of Map processes so that the I/O limit \( s \) is respected.
  - We put a limit \( s \) on the amount of input or output that any one process can have. \( s \) **could be**:
    - What fits in main memory
    - What fits on local disk

- **With proper indexes, computation cost is linear in the input + output size**
  - So computation cost is like communication cost
Pointers and Further Reading
Implementations

- **Google**
  - Not available outside Google

- **Hadoop**
  - An open-source implementation in Java
  - Uses HDFS for stable storage

- **Aster Data**
  - Cluster-optimized SQL Database that also implements MapReduce
Cloud Computing

- Ability to rent computing by the hour
  - Additional services e.g., persistent storage
- Amazon’s “Elastic Compute Cloud” (EC2)
- Aster Data and Hadoop can both be run on EC2
Jeffrey Dean and Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung: The Google File System
Hadoop Wiki
- Introduction
- Getting Started
- Map/Reduce Overview
  - http://wiki.apache.org/lucene-hadoop/HadoopMapReduce
  - http://wiki.apache.org/lucene-hadoop/HadoopMapRedClasses
- Eclipse Environment
- Javadoc
  - http://lucene.apache.org/hadoop/docs/api/
Resources

- Releases from Apache download mirrors
  - http://www.apache.org/dyn/closer.cgi/lucene/hadoop/
- Nightly builds of source
- Source code from subversion
Further Reading

- Programming model inspired by functional language primitives
- Partitioning/shuffling similar to many large-scale sorting systems
  - NOW-Sort ['97]
- Re-execution for fault tolerance
  - BAD-FS ['04] and TACC ['97]
- Locality optimization has parallels with Active Disks/Diamond work
  - Active Disks ['01], Diamond ['04]
- Backup tasks similar to Eager Scheduling in Charlotte system
  - Charlotte ['96]
- Dynamic load balancing solves similar problem as River's distributed queues
  - River ['99]