# Unit 1: Introduction and Motivation

## Videos:

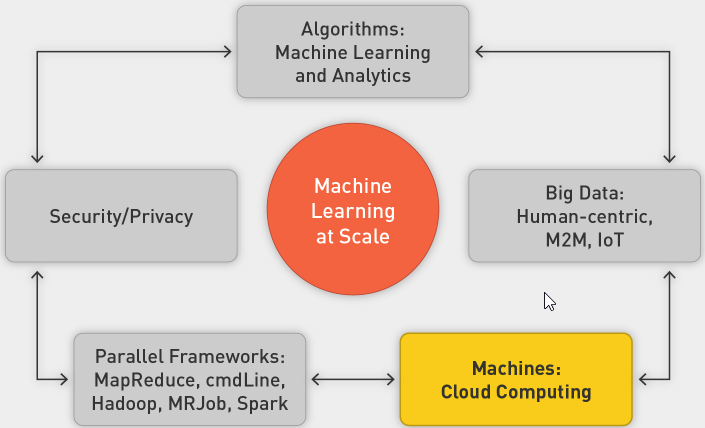
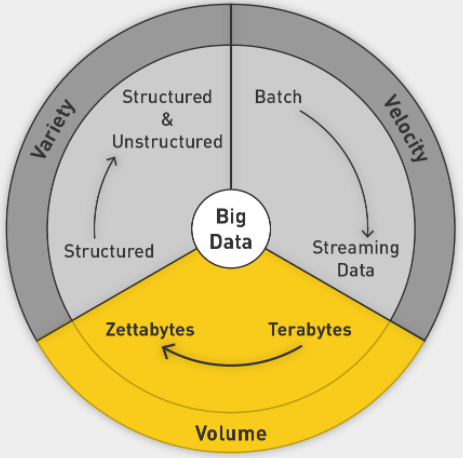
Introduction

1. Intro
2. Big Data
3. Modern-day DS
4. Seven steps in data modelling pipeline
5. Large scale machine learning
6. Tricks of the trade
7. MR using the command line

Readings

1. Doing Data Science Chapter 1 and 2: on Safari
2. An introduction to information retrieval: Chapter 13, downloaded; Info on Naïve Bayes and Text Classification
3. Elements of Statistical Learning, Chapter 7: downloaded and read

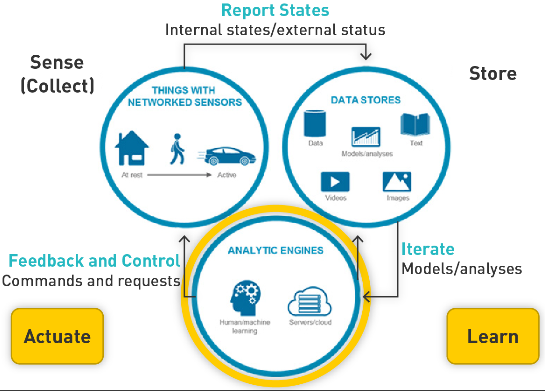
Big Data Definitions

1. ML at Scale
   1. 
2. Big Data Definition
   1. Big data is broad term for data sets so large or complex that traditional data-processing applications are inadequate:
   2. Processing
   3. Laptop (8–16 GB of memory, 1 TB hard drive) overwhelmed with 4–5 GB
   4. Storage
   5. Laptop only 1 TB
   6. Throughput
   7. Three hours to read 1 TB on laptop
   8. Other Challenges
      1. Analysis
      2. Capture
      3. Data curation
      4. Search
      5. Sharing
      6. Storage
      7. Transfer
      8. Visualization
      9. Security
      10. Information privacy
   9. Three Vs
      1. 
      2. Add Veracity: uncertainty of data
3. Sources Driving Big Data
   1. Everything is recorded online
   2. User-generated world, social media
   3. IoT
   4. Scientific Computing
4. Areas of Application
   1. NYSE
   2. FB
   3. CERN
5. Data in Zettabytes (ZB)
   1. 1ZB is 10^21
   2. 2020: 40 ZB

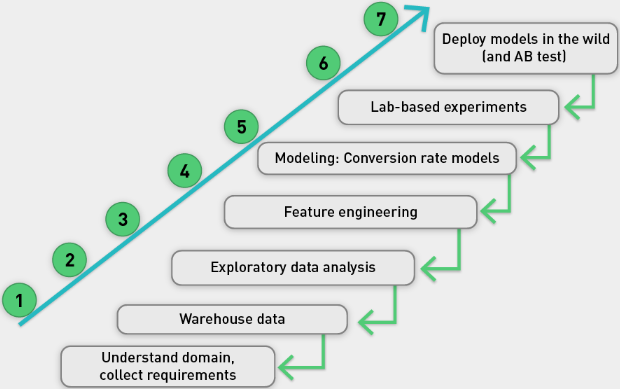
Sources of Big Data

1. Applications of Big Data
2. Societal and Personal Sources of Data
3. Online Society
4. Advertising
   1. Represents 2% of GDP in United States since 1900s
   2. Increased spending online as opposed to on traditional channels
   3. Mobile advertising surpasses others
5. Dating
6. Life Logging
7. Health Care and how tech can help
8. Genomics

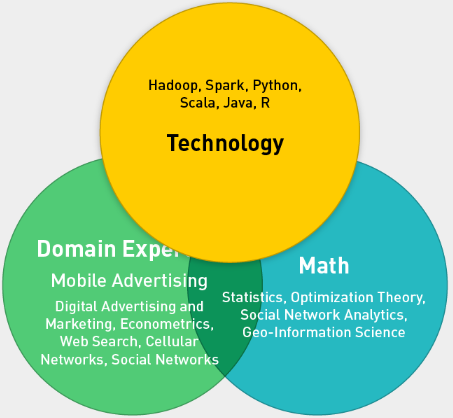
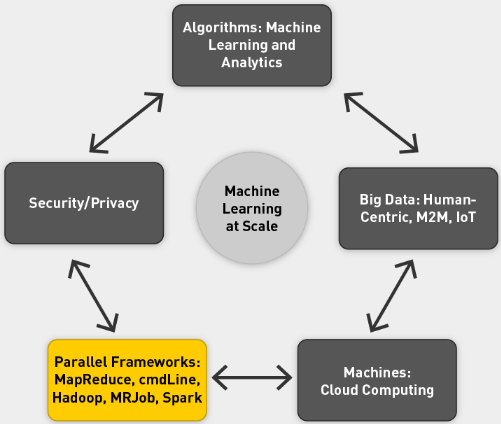
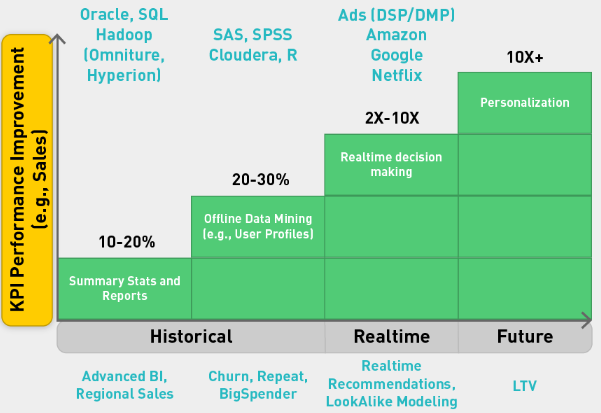
IoT

1. IoT
   1. More data coming from machines than from humans
   2. Machine to machine communication
   3. Attach sensors to all the things
   4. 2020: 26b things on network
   5. Everyone has a unique identifier and data is automatically sent across the internet
2. Japan Earthquake
   1. 14 seconds before, all trains and plants shut down
   2. Mesh of sensors from connected computers to form an early warning system
3. Tracking Nature in the Wild
   1. Sea lions with GPS tracking system
   2. Underwater hub picks up signals from devices
   3. Serengeti
4. Challenges
   1. Battery life
   2. Storage
   3. Network connectivity
   4. Privacy and security
5. Smart homes
6. Smart Cities
   1. 
7. Autonomous Vehicles

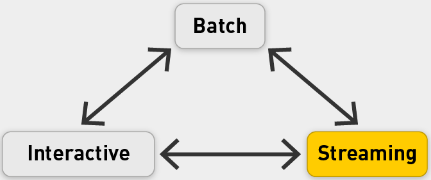
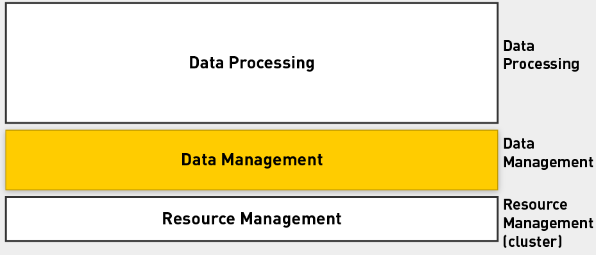
Data Modelling Pipeline

1. Intro
   1. Massive challenge for machine learning
   2. ML is not in a vacuum
   3. ML in real world
2. Modelling
   1. 
   2. Very iterative, might go back at every step
   3. 80% of work is spent in getting, prepping and engineering data
3. Example US insurance
   1. Half of it is property and casualty (auto, home, commercial)
   2. Highly competitive rates
      1. Special rates for good drivers
      2. Score computed through accident history over five years
      3. Static, annual, batch process, easy
4. Progressive’s Snapshot product
   1. Bad driver surveillance through telematics
   2. Small box plugged into steering wheel that records, sends information
   3. Features:
   4. How many miles driven
   5. How many miles driven between midnight and 4 a.m.
   6. Use of sharp or gentle braking
   7. Data used to analyze driving patterns, risk to insurance company
   8. Potential reduced premiums for user
5. Privacy
   1. Huge data generated
   2. Data not transitory
   3. Caution needed with who can access and how is it stored
   4. Who owns the data?

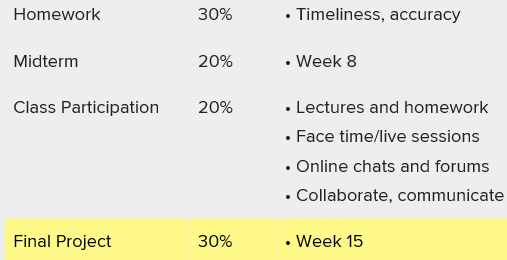
Data Scientist: Post This Class

1. Three skill sets
   1. 
2. Understand Models
3. Understand Security and Privacy
4. ML at scale
   1. 
5. DS Team with different experts in the three fields
6. DS improves KPIs dramatically
   1. 

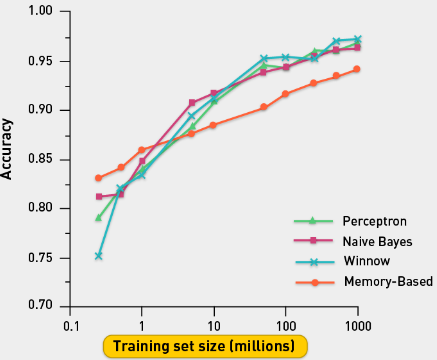
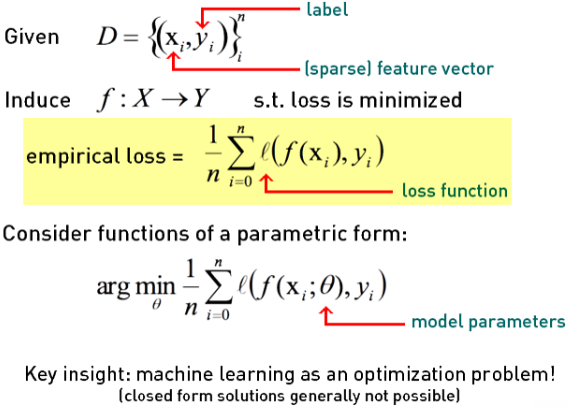
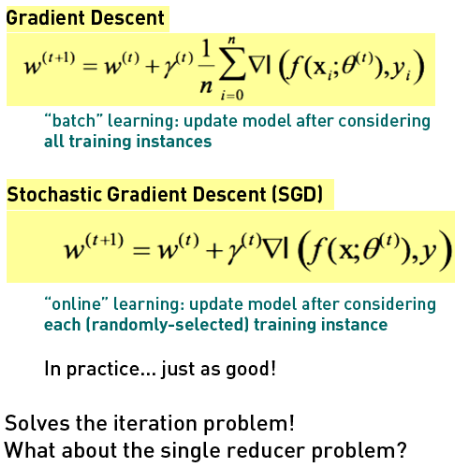
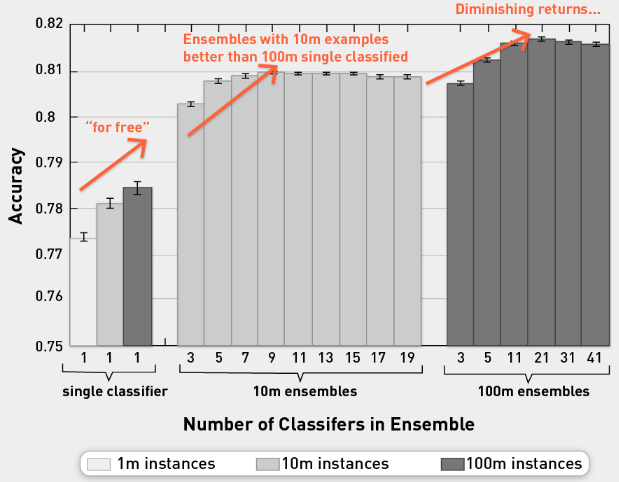
Goals of Class

1. ML at Scale
2. Large-Scale ML
   1. Needs supporting infrastructure
   2. Parallel computing frameworks
3. MapReduce Framework
   1. Allows us to divide and conquer huge data problems
   2. Four frameworks
      1. Cmd
      2. Hadoop
      3. MRJob
      4. Spark
4. Embarrassingly Parallel Problems
   1. Most ML problems
   2. Divide into tasks that can easily be distributed and parallelized
   3. Divided into subproblems
   4. Require little or no communication
   5. Are easily distributed
   6. Have linear computational flow and scale
5. Framework Requirements
   1. Scalable
   2. Fault tolerant
   3. Iterative
   4. Interactive
   5. 
   6. Frameworks that empower us to do above
   7. OS ecosystem
6. Four MR frameworks
   1. CMD based
   2. Hadoop: Storing and manipulating data
   3. MRJob: Convenient Python application programming interface (API) for writing MapReduce programs
   4. Spark: memory backed with integrated framework; Specialized libraries for many things
7. Data analytics stack
   1. 
   2. Populating the stack
   3. 
   4. Add Spark
   5. 
8. Class Phases
   1. Phase 0: CMD
   2. Phase 1: Hadoop/HDFS
   3. Phase 2: MRJob
   4. Phase 3: Spark
9. ML Algorithms
   1. Adopt them for parallel framework
   2. Supervised machine learning (convex optimization, gradient descent, linear regression, decision trees, ensembles of models, support vector machines)
   3. Unsupervised (expectation maximization, matrix multiplication, alternating least squares)
   4. Graphs (random walks, PageRank, graph search algorithms such as breadth-first search, shortest path)
   5. Hybrid algorithms (supervised machine learning and random walks)
   6. Applications (digital advertising, social media, health care, e-commerce, entertainment, metrics, statistics)

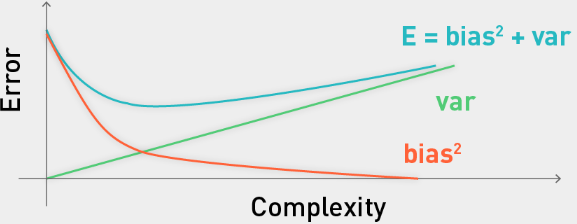
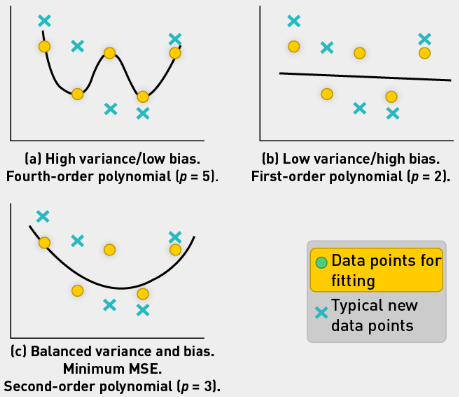
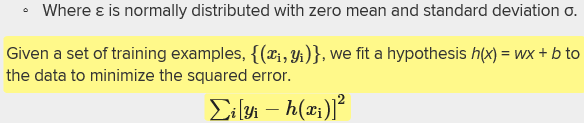
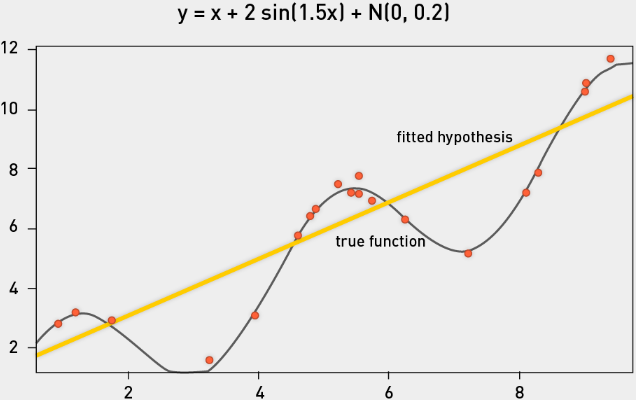
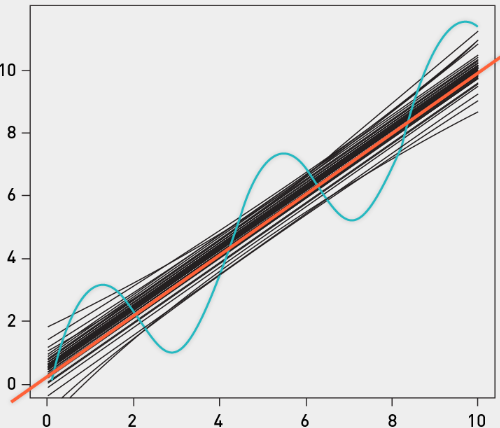
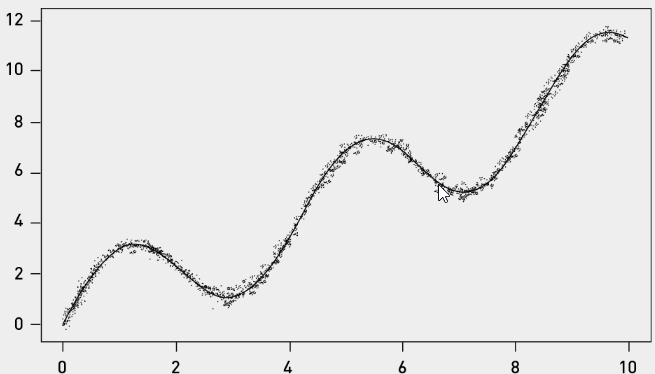
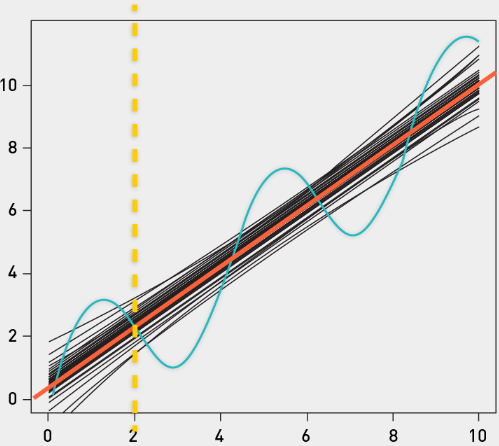
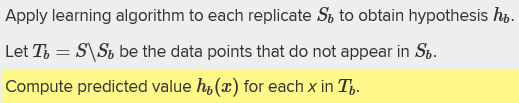
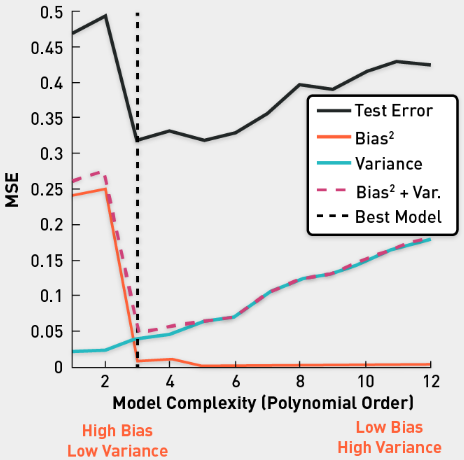
Logistics and Performance Evaluation

1. The more you put in, the more you get back
2. Structure
   1. 13 lectures over 14 weeks
   2. Two exam weeks
   3. Week 8 midterm
   4. Week 15 final project
   5. Performance evaluation
   6. 

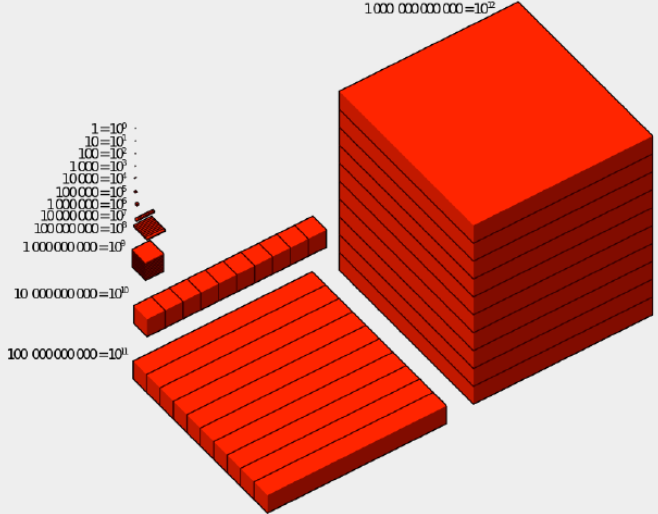
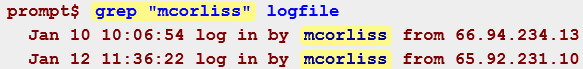
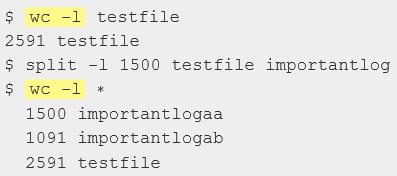
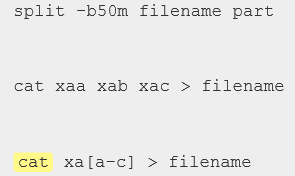
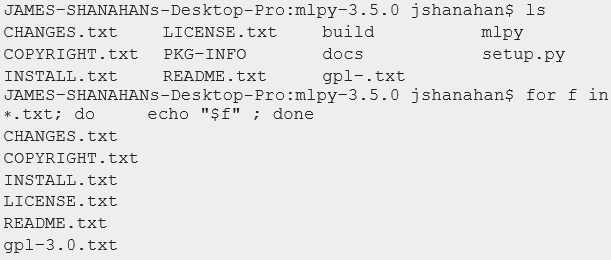
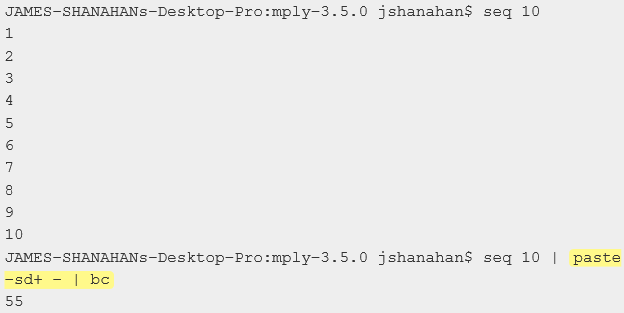
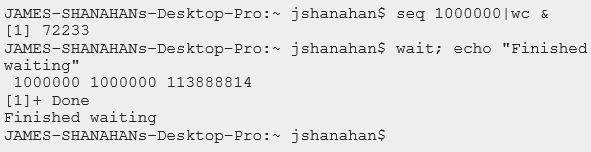
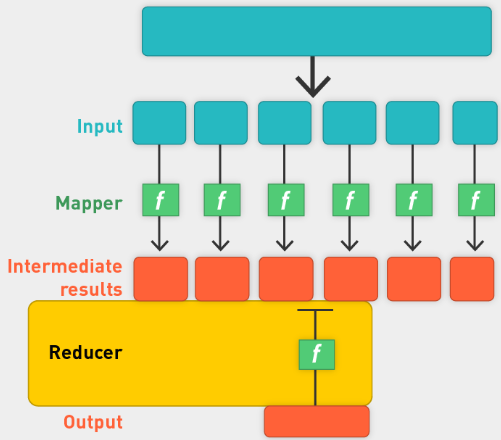
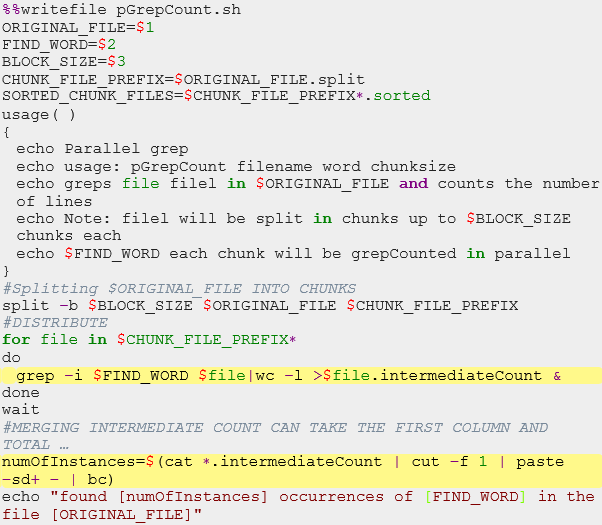
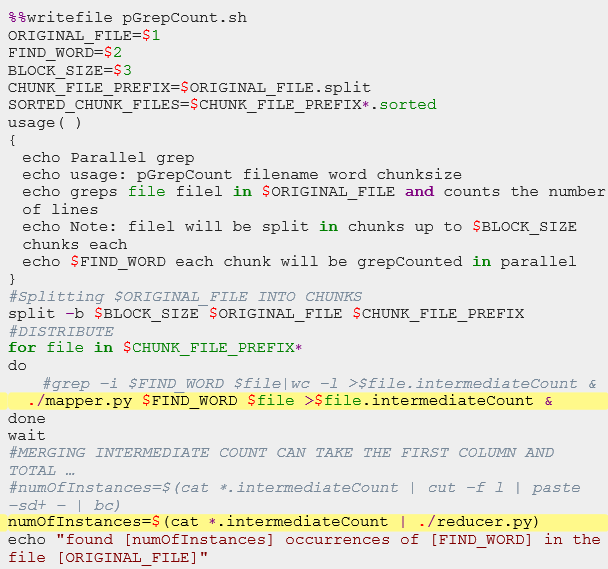
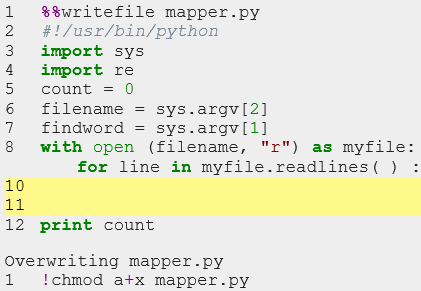
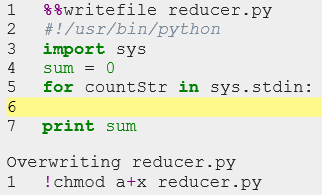
Large Scale ML

1. Do you need more data scientists or do you need more data?
2. ML and Data Study
   1. Filling in confusable words in sentences
   2. Use ML
   3. Vary size of training data
   4. 
   5. Different algorithms perform differently if they have small number of training examples, but similarly with large number of examples
   6. Conclusion: more data leads to 10-20% boost in performance
3. Supervised Classification in a nutshell
   1. 
   2. 
4. Complexity of models
   1. 
   2. Moving from single to ensemble, you get huge boost in performance
   3. However, if you use 100m examples, you get further boost of 1-2%.
   4. Using more data we get amazing improvements, but we hit diminishing returns
5. More data or more DS?
   1. Trade-off between bias and variance
   2. Bias from data scientists
   3. Variance from more data
   4. With more data can reduce variance

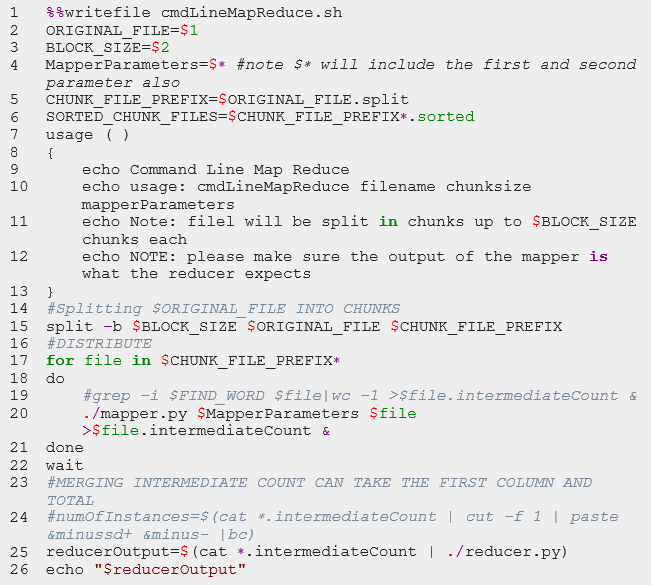
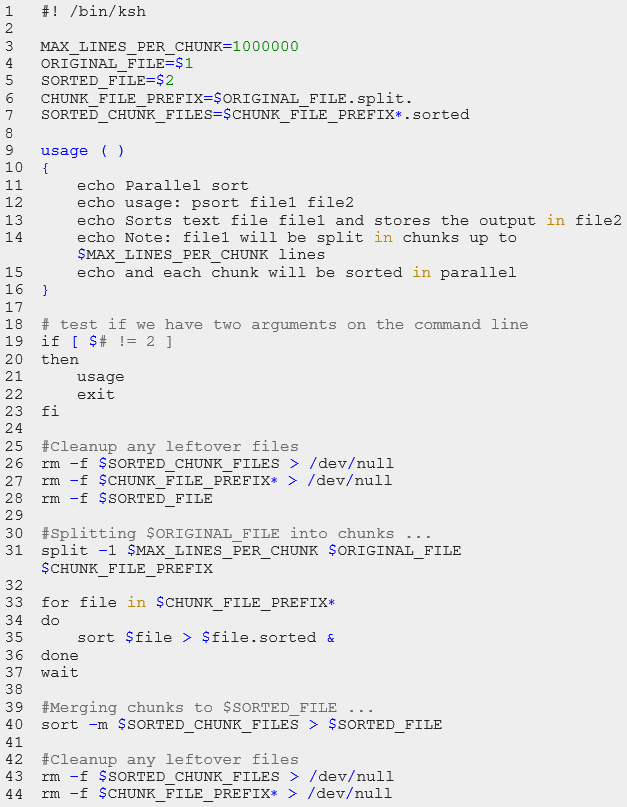
Bias-Variance Background

1. B-V
   1. Empirical studies: more data leads to big improvements
   2. More data or more DS?
   3. Bias-variance trade-off
2. ML objectives
   1. Minimize error term
   2. Minimize loss function
   3. Reduce model complexity
   4. Better generalization
   5. For linear regression, we use mean squared error
3. Loss: Irreducible and reducible error
   1. Irreducible:
      1. Has inherent uncertainty
      2. Associated with natural variability in system
      3. Noisy sensors
   2. Reducible:
      1. Can and should be minimized further to max accuracy
4. Reducible error
   1. Includes both "error due to squared bias" and "error due to variance"
   2. Goal: Simultaneously reduce bias and variance as much as possible in order to obtain as accurate a model as is feasible
   3. Trade-offs in selecting models:
      1. Flexibility and complexity
      2. Selecting appropriate training sets
5. Complexity of model
   1. 
   2. Try to minimize error
   3. E is from two components: bias^2 and variance (noise factor as well)
   4. Bias decreasing as complexity increased
   5. Usually, bias is a decreasing function of complexity, while variance is an increasing function of complexity.
6. BV tradeoff in model selection
   1. 
   2. First plot shows high variance as we fit the data too well.
   3. Second plot shows **high bias as we cannot fit the training data well**.
7. Error due to squared bias
   1. Definition: Amount by which expected model prediction differs from the true value, **over the training data**
   2. Bias is introduced at model selection
   3. Repeat model-building process (through resampling) to obtain average of prediction values
   4. If average prediction values are substantially different than true value, bias will be high
8. Error due to variance
   1. Definition: Amount by which prediction **over one training set differs from expected predicted value over all training sets**
   2. Repeat modeling, measure inconsistency over different training data sets
   3. Variance measures how inconsistent predictions are from one another, over different training sets, not whether or not they are accurate (Manning et al., 2008)
9. BV Analysis in Regression
   1. 
   2. 
10. Example
    1. 
    2. We fit linear model
    3. If we resample 50 times and fit 50 linear models and plot them, we get:
    4. 
    5. Use bootstrap sampling, see average model
    6. 
    7. Avg model introduces a bias. Prediction for x=3 is way above actual value for 3.
    8. From variance perspective, models are variable from training set to training set
    9. Noise: low since artificial
    10. 
11. Variance, Bias and Noise
    1. Variance: 
    2. Model prediction minus average model prediction
    3. Bias: 
    4. Average model prediction vs true function value
    5. Noise: 
    6. Observed target value and true function value
12. Distribution of Predictions at x=2.0
    1. 
    2. High variance between 50 models in this case, but true and mean prediction aren’t very different.
    3. X=5 has big difference between true prediction and mean prediction.
13. Measuring Bias and Variance
    1. Simulate multiple training sets by bootstrap replicates.
    2. Construct B bootstrap replicates of S (e.g., B = 200).
    3. 
    4. Use to est bias and variance
14. Model Complexity
    1. 
    2. Experiment where you take training data and use different models: from linear to polynomial regression (go from 1 to 12 polynomial order)
    3. Linear models have very high bias (don’t fit data well)
    4. High polynomial models fit data precisely, lowering bias, but they increase variance (the green line)
    5. Polynomial regression model of order 3 is the best here
15. BV Decomposition
    1. Can be extended to classification problems
    2. Pedro Domingos (2000a; 2000b): Developed unified decomposition that covers both regression and classification
16. BV Tradeoff code:
    1. [Model Selection: Underfitting, Overfitting, and the Bias-Variance Tradeoff](https://theclevermachine.wordpress.com/2013/04/21/model-selection-underfitting-overfitting-and-the-bias-variance-tradeoff/)
    2. [Ask a Data Scientist: The Bias vs. Variance Tradeoff](http://insidebigdata.com/2014/10/22/ask-data-scientist-bias-vs-variance-tradeoff/)
    3. [polyfit: Polynomial curve fitting](http://www.mathworks.com/help/matlab/ref/polyfit.html)
17. Model Selection
    1. Trade off between b and v
    2. Practical method for selecting model
       1. Minimize error
       2. Function of model complexity
18. Effect of algorithm parameters
    1. K-Nearest Neighbor: Increasing k typically increases bias, reduces variance
    2. Decision trees of depth D: Increasing D typically increases variance, reduces bias
    3. Radial basis function (RBF) support vector machines (SVM) with parameter σ: Increasing σ increases bias, reduces variance
    4. Bagging tends to reduce variance without increasing bias
19. Wrap-up
    1. Bias and variance of an estimator are related to squared prediction error.
    2. These concepts can be applied to classification problems.
    3. An optimal estimator will have both low variance and low bias.

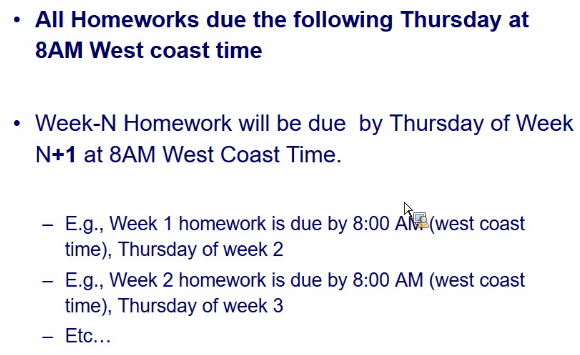
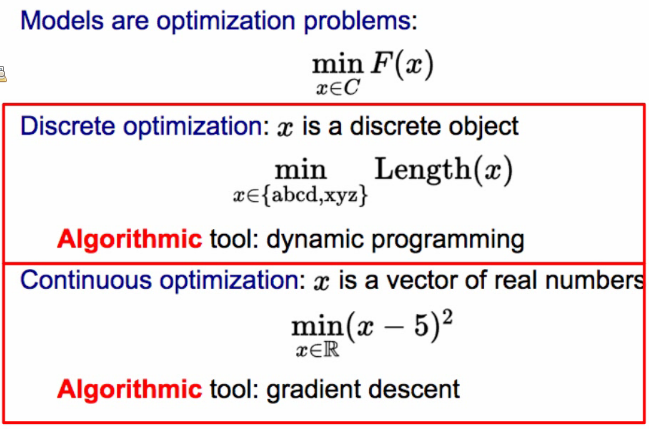
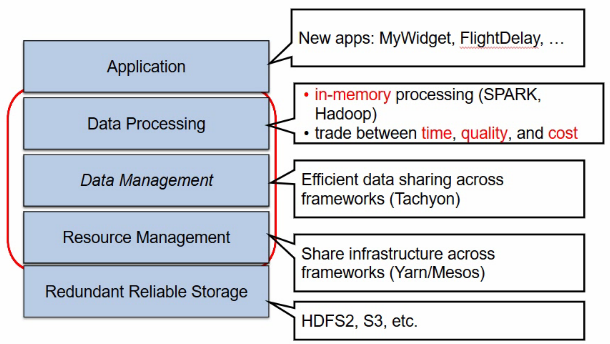
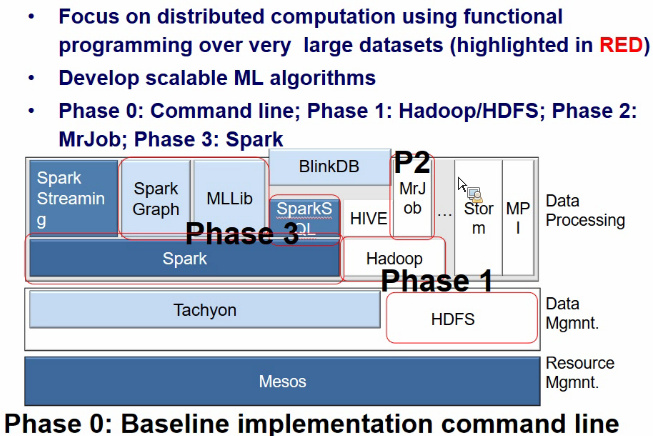
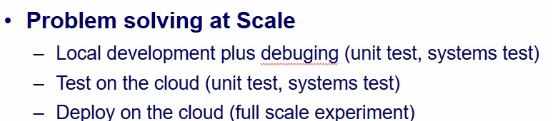
MR with CMD

1. Overview
   1. Aim for ML on billions of examples
   2. Use parallel computational framework based on cmd
   3. Divide and conquer strategy
2. Goal
   1. Large scale problem
   2. Decompose into nonoverlapping subproblems
   3. Non-ML problem
   4. Related to NB
3. Task
   1. Find all occurrences of the word apple (by date) in Facebook posts log file.
   2. Parallel grep for FB posts available in a log file
4. Back-of-the-Envelope Calculations
   1. 10^9 active FB users on a daily basis
   2. One post per day = 10^9 posts per day
   3. 
   4. Each post = 1000 bytes (text and metadata)
   5. 1TB of data (1000 \* 10^9 = 10^12 bytes)
5. Processing Times
   1. On one node:
      1. if you scan at 100 MB/s (10^8), it will take you
      2. 
      3. This is way too long
   2. On a 96 multicore machine
      1. Scanning at 100MB/s, assume 100 cores has no latency
      2. 
      3. How do you write code to leverage this power?
6. Strategy: Calculate Term Popularity
   1. Unix-based divide and conquer strategy
   2. Poor man’s distributed computational perspective
7. Ground Rules
   1. Use only below commands
   2. split, grep, wc, merge, cat, for, echo
   3. cut, paste, bc
   4. |, &, wait
8. GREP command
   1. "grep <string> <filename>" returns all lines in file that contain string
   2. Each line corresponds to one record (Facebook post)
   3. Date, time, body of post
   4. 
9. Pipe command: |
   1. Chain together various commands
   2. 
   3. Find lines that contain all three words
10. Split command
    1. Split by number of lines: "split -l 30" splits file into chunks of 30 lines or less
    2. Split by number of MB: ◦"split -b 24m" splits file into chunks of 24MB or less
    3. Example Split command:
       1. 
       2. Wc counts number of lines in this case
       3. 
       4. Cat displays the data or feeds it into another file
11. For loop and Echo
    1. 
12. Cut, Paste, bc
    1. Seq just generates list of numbers
    2. 
    3. Paste takes the numbers produced by the sequence and creates a string with a plus sign between each of the numbers.
    4. Bc then takes that string and executes it as a calculator, summing up the count
13. Parallel Computing with & and Wait
    1. "&" causes parent process to spawn off parallel processes
    2. "wait" will cause parent process to wait until child processes have finished
    3. 
    4. "seq 1000000|wc &" causes the parent terminal process to spawn line-count task off as child process
    5. "wait" causes the shell to freeze until child process finishes
    6. It will print Finished waiting when wait is over
    7. Once child command finishes, shell can continue and run echo command
14. Schematic of Parallel Processing
    1. Split file in chunks of 10G each
    2. Take grep function and grep each chunk | wc to count how many matches the grep generated in each chunk. Output is number of matches per chunk
    3. Merge these counts together
    4. 
    5. Unix Commands
    6. 
    7. MapReduce Pattern
    8. 
15. MapReduce Framework
    1. 
    2. How about using python?
    3. 
    4. Write mapper.py and reducer.py
    5. Change execution privileges
16. Python: Mapper for Grep
    1. 
    2. If findword in line
    3. Count = count + 1
17. Python Reducer for Grep
    1. Barrier pattern: we cannot start reduce function until all mapper processes are done
    2. 
    3. Sum = sum + int(countStr)

Summary

1. Keep Unix MR framework, swap out map and reduce files for other frameworks
   1. 
   2. Parallel Sort
   3. 
2. Course Overview
   1. How to quantify Data
   2. How to estimate Time
   3. Powers of 10
   4. Parallelism
   5. Scale mundane tasks like grep, sort
3. Challenges in MR Framework
   1. Data management
   2. Task management
   3. Fault tolerance
   4. Lack of interactivity
4. A better framework
   1. Scale Out, not up
   2. Fault tolerance
   3. Minimise data movement
   4. Move processing of the data
   5. Hide file system details from application developer
   6. Seamless scalability
5. Class Phases
   1. Phase 0: Command line
   2. Phase 1: Hadoop/HDFS
   3. Phase 2: MRJob
   4. Phase 3: Spark
6. Summary
   1. Big data (what, why, where, who, how)
   2. Role of data scientist
   3. Data modeling pipeline
   4. Bias variance as a means of understanding more data
   5. MapReduce framework using command-line utility

## Lecture:

* Long/Short slides, follow short
* Lecturer: Mike Bowles
* Homework, normally due Thursday after next class; HW1 should be done before the class however.
* 
* HW1:
* Saturday: OH for setting up Docker container; check on slack when it is.
* Docker and HW: look at UCB-w261
* Slides:
* Data science is everywhere
* Steps:
  + Raw data
  + Feature extraction
  + Classifier is run
  + Result
* What can make use of MR format, what cannot?
* Optimization
* 
* ML allows us to shift the complexity from the program to the data.
* Deep learning
* Data Analytics Stack
* 
* Class Phases
* 
* Scaling solutions
* 

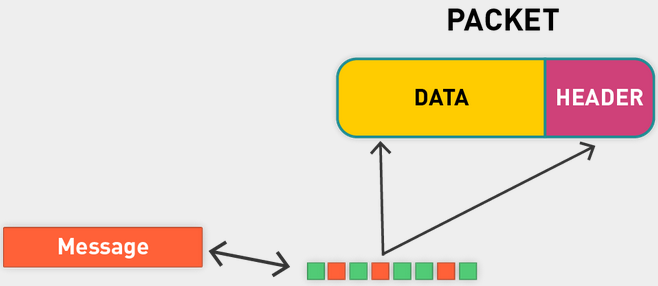
# Unit 2: Parallel Computing, MapReduce, and Hadoop (Data Storage and Algorithms)

## Videos:

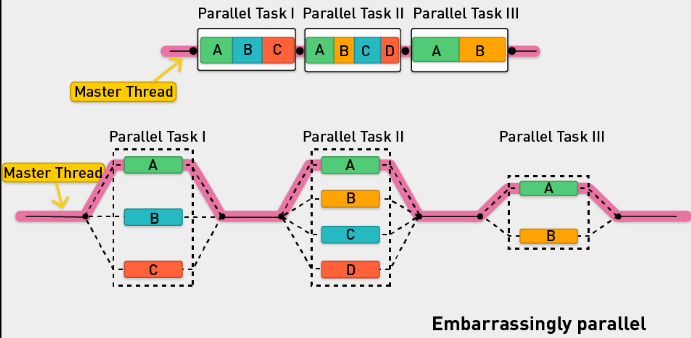
Introduction

1. Gigabytes of data? Can fit on PC memory
2. Petabytes? Need cluster of server
3. MapReduce
4. Embarrassingly Parallel
5. Hadoop

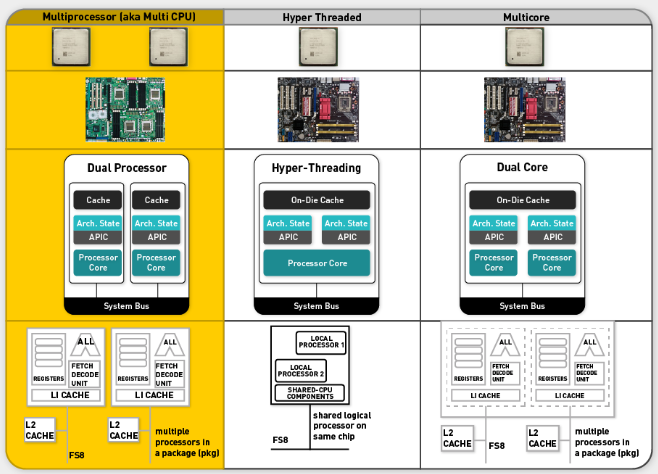
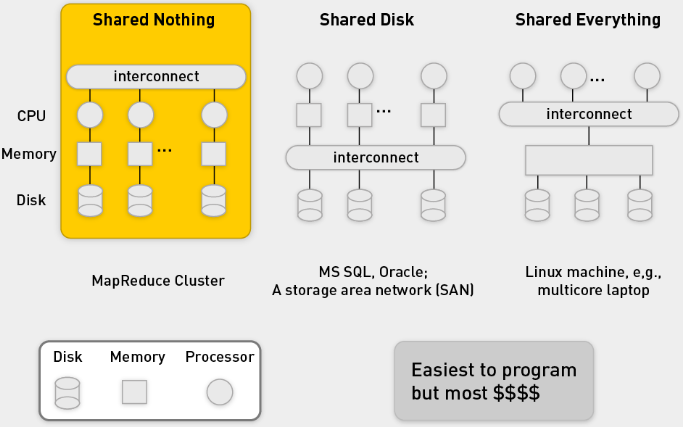
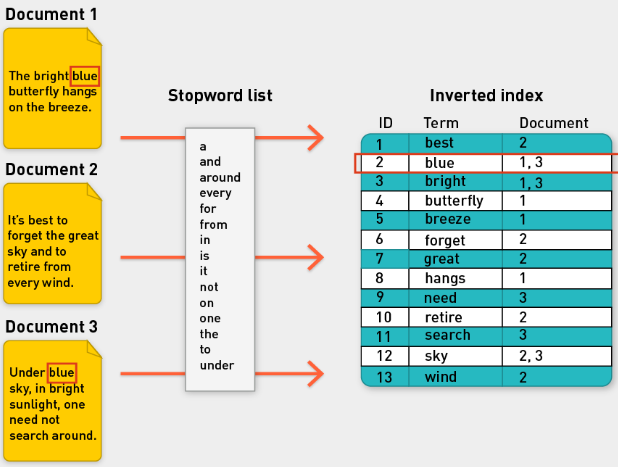
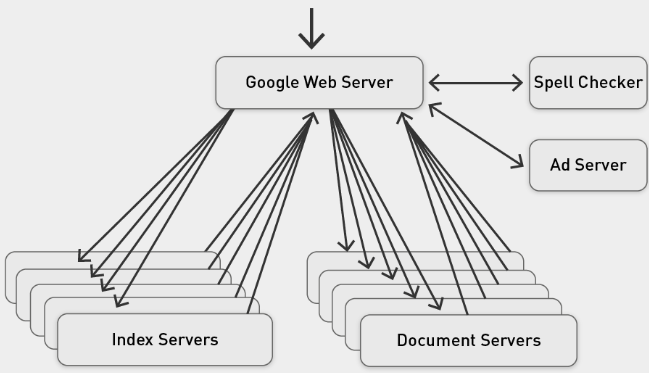
Motivation for Parallel Computing

1. Assume 100b web pages
   1. How to store them?
   2. 10,000 characters per page x10^11 = 10^15 (1 petabyte of raw data)
   3. How can we scan them?
   4. How to do sth useful with them?
   5. Document frequency of term
   6. Extract out-links and count in-links for page
2. Why Parallelism?
   1. Data size is increasing
   2. Scan 1000 TB on one node at 100 MB/s = 120 days
   3. Store that amount of data?
   4. Growing demand
   5. Parallelism: Divide and Conquer
   6. Standard or commodity and affordable architecture emerging
   7. Clusters of commodity linux nodes (CPU, memory, disk)
   8. Gigabit internet interconnectivity
3. Design Goals for Parallelism
   1. Scalability to large data volumes
   2. Scan 1,000 TB (1PB) on one node at 50 MB/s = 120 days
   3. Scan on 10,000-node cluster = 16 minutes!
   4. Cost efficiency: commodity nodes and network
   5. Fault tolerance is automatic
   6. Easy to use framework
4. D&C Is common: Packet switching
   1. Packet switching is a digital networking communications method that transmits messages in chunks or blocks, called packets.
   2. Packets are transmitted via a medium that may be shared by multiple simultaneous communication sessions.
   3. Packet switching increases network efficiency and robustness and enables technological convergence of many applications operating on the same network.
   4. 
   5. Split message into packets, each packet has data (payload) and a header that tells where it goes, where it comes from, etc.
   6. Sender sends chunks across network. Each packet may take different path through the network.
   7. Barrier sync in receiver says: hold until all packages have been received and combined
   8. Individual packets get transmitted in isolation. There is no communication between the packets, and they synchronize only at the destination node; this is known as barrier-based synchronization in parallel computing.
   9. Handle process synchronization through devices such as barriers.
5. Conclusion
   1. The key to success here was divide and conquer
   2. Decompose a large task into smaller ones
   3. We came up with a very nice framework for parallelizing tasks on the command line!
      1. But it is limited
      2. Granularity of task is somewhat coarse
      3. No fault tolerance
      4. No control over file space
   4. Divide and conquer does not come for free; there are obligations in terms of communication, synchronization, and fault tolerance

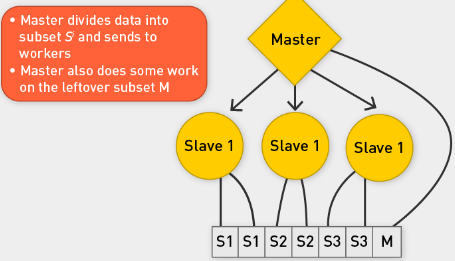
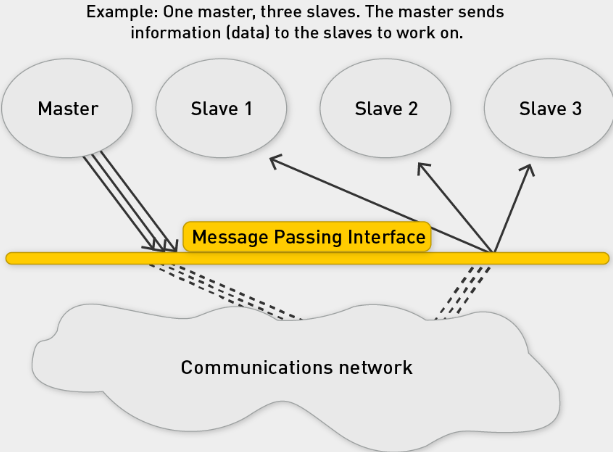
ll Definition and Communication Synchronization Types of PC Tasks

1. Serial vs Parallel Computation
   1. Traditionally, computer software has been written for serial computation.
   2. To solve a problem, an algorithm is constructed and implemented as a serial stream or list of instructions.
   3. These instructions are executed on a central processing unit on one computer. Only one instruction may execute at a time—after that instruction is finished, the next is executed.
   4. Parallel computing, in contrast, uses multiple processing elements simultaneously to solve a problem.
   5. This is accomplished by breaking the problem into independent parts so that each processing element can execute its part of the algorithm simultaneously with the others.
   6. The processing elements can be diverse and include resources such as a single computer with multiple processors, several networked computers, specialized hardware, or a combination of the above.
2. Parallel computing
   1. Parallel computing is a form of computation in which many calculations are carried out simultaneously.
   2. It operates on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel").
3. Parallel computers
   1. How much does HW support parallelism?
   2. Single computer: Multicore or multiprocessor
   3. Clusters: multiple computers
   4. Specialized HW: GPUs
4. Challenges of Group Work
   1. Division of work
   2. Coordination costs (partition problem, communication)
   3. Coordination costs represent time and energy that group work consumes that individual work does not
   4. They include the time it takes to coordinate schedules, arrange meetings, meet, correspond, make decisions collectively
   5. Integrate the contributions of group members, etc.
   6. The time spent on each of these tasks may be small, but together they are significant
   7. Coordination costs cannot be eliminated
5. Parallel programs are challenging
   1. Single node:
      1. Parallel computer programs are more difficult to write than sequential ones, because concurrency introduces several new classes of potential software bugs, of which **race conditions** are the most common
      2. Communication and synchronization between the different subtasks are typically some of the greatest obstacles to getting good parallel program performance
   2. Clusters: similar issues, amplified
6. Dev responsibilities
   1. Explicitly address above issues
   2. Shared-memory programming: Explicitly coordinate access to shared data structures through synchronization primitives such as mutexes. Also, explicitly handle process synchronization through devices such as barriers. Be vigilant against deadlocks and race conditions
7. Without Sync
   1. Thread A
      1. Read variable X
      2. Compute X + 1
      3. Assign to X
   2. Thread B
      1. Read variable X
      2. Compute X + 1
      3. Assign to X
   3. We don’t know which thread will run first. Depending on order, final value might be different.
   4. Case 1: X = X + 1
      1. A and B could both read the initial value of X and then overwrite each other
      2. Final result: X = X + 1
   5. Case 2: X = X + 2
      1. A reads and writes, then B reads and writes
      2. Final result: X = X + 2
   6. Don’t know whether the right one (second will occur)
   7. Case 1 is a race condition
8. Use Sync to avoid Race condition
   1. There are several ways to synchronize and avoid race conditions
   2. Depending on the level of parallelism possible for the problem at hand
   3. Mutex:
      1. In computer science, mutual exclusion refers to the requirement of ensuring that no two concurrent processes are in their critical section at the same time; it is a basic requirement in concurrency control, to prevent race conditions
      2. In computer programming, a mutex is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously
      3. When a program is started, a mutex is created with a unique name
      4. Usually a mutex is associated with a variable or code block
      5. 
   4. Barrier
9. Deadlock from Mutex
   1. Next issue: Mutex base syncing can cause deadlocks.
   2. Threads A and B both need locks for variables X and Y.
   3. A acquires mutex\_X.
   4. B acquires mutex\_Y.
   5. A waits for mutex\_Y to be free.
   6. B waits for mutex\_X to be free.
10. Join Pattern for Sync
    1. Those problems that can be decomposed into independent subtasks, requiring no communication or synchronization between the subtasks except a join or barrier at the end
    2. 
    3. Master thread with multiple zones, each with multiple tasks (A,B,C). Tasks don’t need to communicate with each other, can be done in parallel, need to be joined together
11. Barrier
    1. Another popular way of syncing is the barrier method
    2. In parallel computing, a barrier is a type of synchronization method
    3. Explicitly handle process synchronization through devices such as barriers
    4. It is pretty effective and very coarse grained and can be great in certain types of problems
    5. A barrier for a group of threads or processes in the source code means any thread or process must stop at this point and cannot proceed until all other threads or processes reach this barrier
12. Summary
    1. Those problems that can be decomposed into independent subtasks, requiring no communication or synchronization between the subtasks except a join or barrier at the end, are very parallelizable ("embarrassingly parallel")
    2. Mutex, Join, Barrier patterns can be used to sync
13. Categories of Computational Tasks
    1. Applications are often classified according to how often their subtasks need to synchronize or communicate with each other
    2. Fine-grained parallelism: An application exhibits fine-grained parallelism if its subtasks must communicate many times per second (share memory programming)
    3. Coarse-grained parallelism: It exhibits coarse-grained parallelism if they do not communicate many times per second
    4. Embarrassingly parallel: An application is embarrassingly parallel if it rarely or never has to communicate: divide and conquer; summing a list of numbers
    5. Such applications are considered the easiest to parallelize
    6. Can be realized on a shared nothing architecture (see this shortly)
14. Examples of Embarrassingly Parallel problems
    1. An application is embarrassingly parallel if it rarely or never has to communicate
    2. Summing a list of numbers
    3. Matrix multiplication
    4. Lots of machine learning algorithms
    5. Tree growth step of the random forest machine learning technique
    6. Genetic algorithms and other evolutionary computation metaheuristics
    7. Serving static files on a Web server to multiple users at once
    8. Computer simulations comparing many independent scenarios, such as climate models
    9. Ensemble of numerical weather predictions
    10. Discrete Fourier transform, where each harmonic is independently calculated

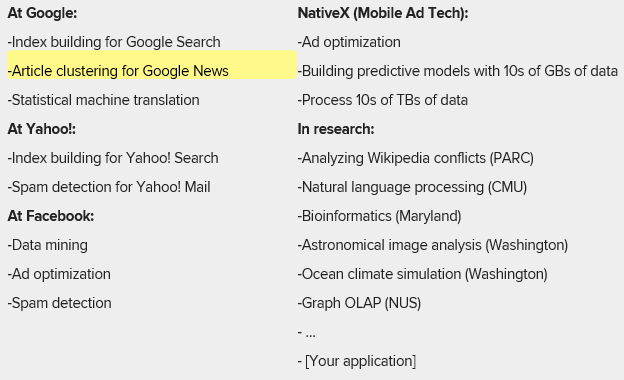
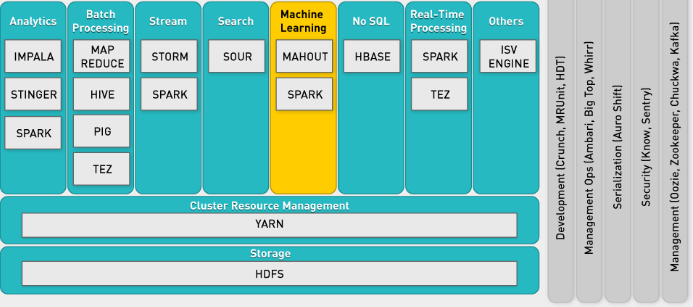
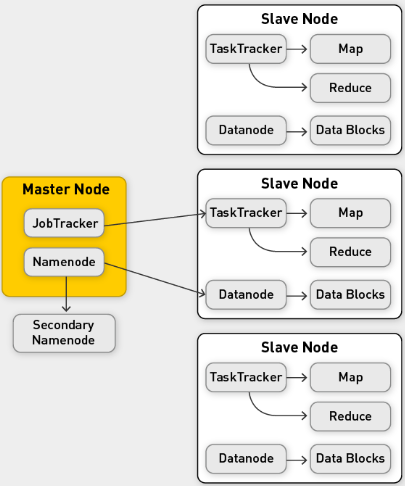
Architectures for Parallel computing

1. CPUs: Multiprocessor vs Multicore
   1. We think about computation in terms of CPU, memory, and hard disks (and possibly network bandwidth)
   2. A computer (motherboard) has multiple components for computation
   3. A CPU, or simply "processor," contains many discrete parts, such as one or more memory caches for instructions and data, instruction decoders, and various types of execution units for performing arithmetic or logical operations
   4. GPU (graphics processor unit)
   5. Multicore CPU: The CPU is not always busy; e.g., it may be idle while the GPU is performing a task. A CPU can be better utilized by sharing with many tasks. In 2000, IBM first introduced a dual-core CPU. A multicore CPU has multiple execution cores on one CPU; also shares memory; may have its LX caches
   6. 
2. iPhone 6 Single CPU with Dual Cores
   1. iPhone has single CPU with dual cores
   2. And a quad-core GPU
   3. iPad: three-core CPU; eight-core GPU
3. Parallel Systems
   1. Resources: CPU, Memory, Disk, Network
   2. Many parallel architectures
      1. Shared Nothing
      2. Shared disk
      3. Shared All
   3. 
4. Shared Memory Programming
   1. Onus on the programmer
   2. In shared memory programming, the developer needs:
   3. To explicitly coordinate access to shared data structures through synchronization primitives such as mutexes
   4. To explicitly handle process synchronization through devices such as barriers
   5. To remain ever vigilant for common problems such as deadlocks and race conditions
5. Shared Nothing
   1. Shared nothing: The machines are connected only by the network (scale out).
   2. A shared nothing architecture (SN) is a distributed computing architecture in which each node is independent and self-sufficient, and there is no single point of contention across the system.
   3. A "shared nothing" architecture means that each computer system has its own private memory and private disk.
   4. The "shared nothing" architecture shares network bandwidth because it must transfer data from machines doing the map tasks to machines doing the reduce tasks over the network.
   5. In order to achieve a good workload distribution, MPP systems have to use a hash algorithm to distribute (partition) data evenly across available CPU cores.
   6. It is very popular because it is highly scalable
   7. Shared nothing is popular for Web development because of its scalability
   8. As Google has demonstrated, a pure SN system can scale almost infinitely simply by adding nodes in the form of inexpensive computers, since there is no single bottleneck to slow the system down
   9. Shard or partitioned data (e.g., distribute documents)
   10. Google calls this sharding
   11. An SN system typically partitions its data among many nodes on different databases (assigning different computers to deal with different users or queries) or may require every node to maintain its own copy of the application's data, using some kind of coordination protocol
   12. This is often referred to as database sharing
6. Shared Everything
   1. Only way to scale is to scale up
   2. Very expensive to add more memory and disk
   3. Oracle RAC does not run on a shared nothing system. It was built many years ago to run on a "shared disk" architecture.
   4. In this world, a computer system has private memory but shares a disk system with other computer systems.
   5. Such a "disk cluster" was popularized in the 1990s by Sun and HP, among others. In the 2000s, this architecture has been replaced by "grid computing," which uses shared nothing.
   6. Shared disk has well-known scalability problems when applied to database management systems (DBMSs) and is super-expensive.
   7. A storage area network (SAN) can cost $500,000 and still struggle with terabytes of data.
7. Web Search:
   1. 
   2. Sharding: partition data
   3. Google web search in the past
   4. 
   5. Index servers have index of the documents
   6. Doc servers have contents of document
   7. Process query locally on each index server to figure out what is relevant for a query.
   8. Web server takes those results from each server, merge them, create final ranking
8. Shared Nothing Architecture
   1. 

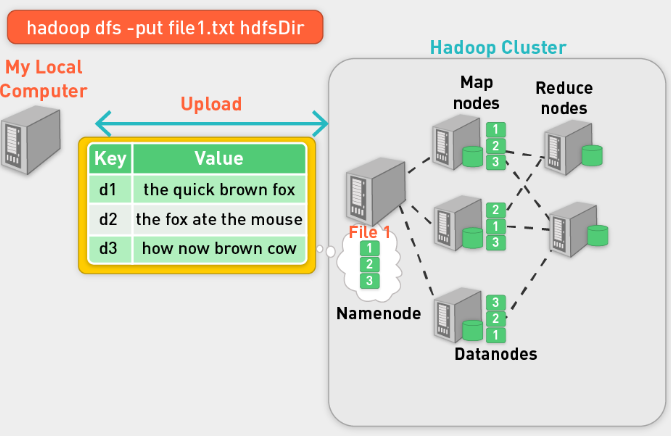
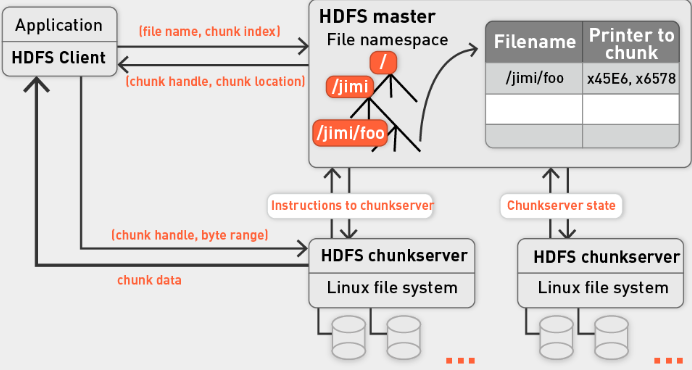
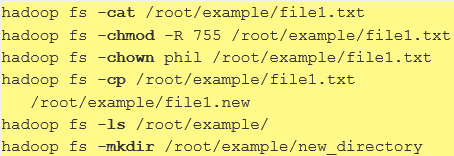
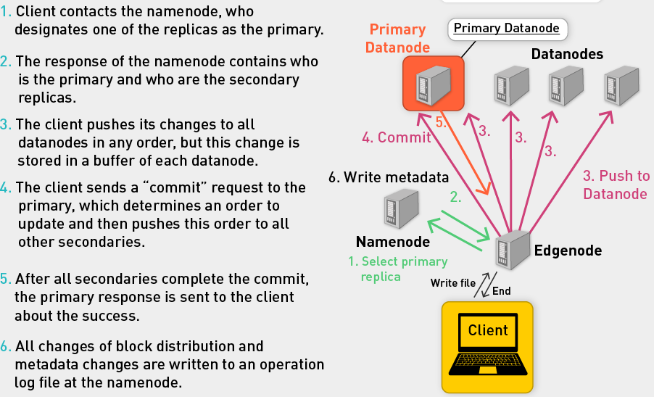
Dev frameworks for Parallel Computation

1. Frameworks for Parallel Computation
   1. Onus is on the programmer, they can do it themselves if they want
   2. Language extensions such as OpenMP (Open Multi-Processing) can be used for shared-memory parallelism
   3. OpenMP has predefined locking paradigms
      1. Reduce the chance of deadlock
      2. Not foolproof (though possibly fool resistant)
   4. Cluster and grid computing
      1. Provide logical abstractions that hide details of operating system synchronization and communications primitives
      2. MPI: Libraries implementing the Message Passing Interface (MPI) for cluster-level parallelism (same local network)
      3. MapReduce: MapReduce is a framework processing parallelizable problem across huge data sets, using a large number of computers (nodes); cluster or grid (nodes are shared across geographically or heterogeneous hardware)
2. MPI
   1. Message Passing Interface (MPI) provides a powerful, efficient, and portable way to express parallel programs.
   2. MPI is a specification for an API that allows many computers to communicate with one another.
   3. It is used in computer clusters and supercomputers.
   4. MPI was created by William Gropp, Ewing Lusk, and others.
   5. MPI, in short:
      1. Hides the details of architecture.
      2. Hides the details of message passing and buffering.
      3. Provides message management services:
      4. Packaging
      5. Send, receive
      6. Broadcast, reduce, scatter, gather message modes
   6. MPI defines an environment where programs can run in parallel and communicate with each other by passing messages to each other. There are two major parts to MPI:
      1. The environment programs run in.
      2. The library calls programs use to communicate.
   7. MPI is so preferred by users because it is so simple to use. Programs just use messages to communicate with each other. Can you think of another successful system that functions using message passing? (The Internet!)
      1. Each program has a mailbox (called a queue).
      2. Any program can put a message into another program's mailbox.
      3. When a program is ready to process a message, it receives a message from its own mailbox and does something with it.
3. MPI: Master Divides and sends data to workers
   1. 
   2. Example task: Sum an array of numbers.
   3. In this program, the master divides the array into subarrays Si and sends these to each slave.
   4. Each slave will then calculate the sum of its subarray. In the case where the size of the array is not an even multiple of the number of slaves, the master will finish the remaining work by calculating the sum of the last clause of the array.
   5. 
4. MPI for Big Data?
   1. Data transfer is huge problem: transmit array of numbers to slaves
   2. Requires cooperation of sender and receiver
   3. Cooperation not always apparent in code
   4. Hard for programming
   5. Explicitly communicate between nodes via message passing
   6. Synchronization is hard and error prone
   7. Not designed for fault tolerance
   8. Not designed for data locality
   9. Move data to computation nodes
   10. Network bandwidth is limited
5. Data Locality and Isolation
   1. Various patterns for parallel computation have been proposed over the years.
   2. These patterns, when composed, can be used to express computations in a wide variety of domains.
   3. Scalable approaches to parallel computation have to take two things into account: Tasks and data.
   4. Many approaches to parallel computation overemphasize the former (task) at the expense of the latter (e.g., MPI).
   5. However, in order to achieve scalable parallel performance, algorithms need to be designed in a way that emphasizes data locality and isolation.
6. MapReduce
   1. The MapReduce pattern provides a simple way for the software developer to express both data and locality in isolation.
7. Extending for Loops to Maps
   1. A set of numbers
   2. A common pattern in sequential programming is iteration
   3. If we disallow dependencies between loop iterations, we do get a type of computation that can be parallelized easily: The map pattern
   4. In the map pattern, a set of loop indices are generated, and independent computations are performed for every unique index
   5. Map is a Design pattern
   6. Map is a design pattern in parallel computing where a simple operation is applied to all elements of a sequence, potentially in parallel.
   7. It is used to solve embarrassingly parallel problems:
   8. Those problems that can be decomposed into independent subtasks, requiring no communication or synchronization between the subtasks except a join or barrier at the end
   9. The map pattern is typically combined with other parallel design patterns. E.g., map combined with category reduction gives the MapReduce pattern:
   10. A straightforward way to express both data locality and isolation.
8. MR frameworks
   1. Some parallel programming systems, such as OpenMP and Cilk, have language support for the map pattern in the form of a parallel for loop; languages such as OpenCL (Open Computing Language) and CUDA (Complete Unified Device Architecture) support elemental functions (as "kernels") at the language level
   2. Limitations of these frameworks:
      1. Developers are still burdened to keep track of how resources are made available to workers
      2. Additionally, these frameworks are mostly designed to tackle processor-intensive problems and have only rudimentary support for dealing with very large amounts of input data
   3. MapReduce-Hadoop-Spark to the rescue!
   4. Distributed data handling, and distributed computation, framework that hides system-level details
9. MR: Move code to the data
   1. Distributed data handling and storage
      1. Terabytes and petabytes in size
      2. Large-data processing by definition requires bringing data and code together for computation to occur; no small feat for huge data sets
      3. MapReduce addresses this challenge by providing a simple abstraction for the developer, transparently handling most of the details behind the scenes in a scalable, robust, and efficient manner
   2. Distributed computation: Move the code to the data
      1. Like OpenMP and MPI, MapReduce provides a means to distribute computation without burdening the programmer with the details of distributed computing (but at a different level of granularity)
      2. But instead of moving large amounts of data around, it is far more efficient, if possible, to move the code to the data
      3. This is operationally realized by spreading data across the local disks of nodes in a cluster and running processes on nodes that hold the data
10. MR + Scalability and Fault Tolerance
    1. Inspired by functional programming: The model is inspired by the map and reduce functions commonly used in functional programming, although their purpose in the MapReduce framework is not the same as in their original forms
    2. Scalability and fault tolerance: The key contributions of the MapReduce framework are not the actual map and reduce functions but the scalability and fault tolerance achieved for a variety of applications by optimizing the execution engine once
    3. Optimizing the communication: Optimizing the communication cost is essential to a good MapReduce algorithm
    4. The use of this model is beneficial only when the optimized distributed shuffle operation (which reduces network communication cost) and fault-tolerance features of the MapReduce framework come into play
11. Why use MR?
    1. Increase computational power: linear scalability
    2. Must be able to split up the data in chunks for processing, which are then recombined later
    3. Requires a constant flow of data from one simple state to another
    4. The Hadoop framework handles the processing details, leaving developers free to focus on application logic
    5. Within the Hadoop Core framework, MapReduce is often referred to as mapred, and HDFS is often referred to as dfs
    6. MRs ideal for "embarrassingly parallel" problems
    7. Very little communication
    8. Easily distributed
    9. Linear computational flow

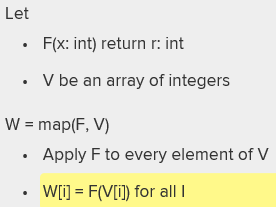
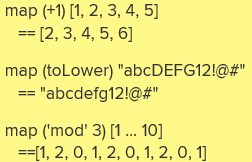
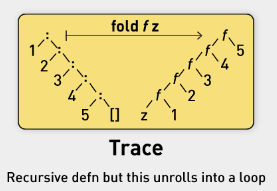
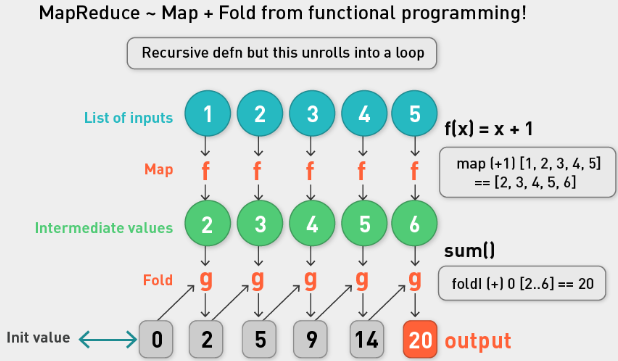
Hadoop Background and History

1. MR
   1. MapReduce is a programming model for processing and generating large data sets with a parallel, distributed algorithm on a cluster
   2. Pioneered by Google
   3. Processes 20 PB of data per day
   4. Popularized by open-source Hadoop project
   5. Used by Yahoo!, Facebook, Amazon, NativeX, …
   6. Distributed file system (HDFS)
      1. Single namespace for entire cluster
      2. Replicates data three times for fault tolerance
   7. MapReduce implementation
      1. Executes user jobs specified as "map" and "reduce" functions
      2. Manages work distribution and fault tolerance
2. What is it used for?
   1. 
   2. And for ML
3. Hadoop Ecosystem
   1. 
4. What is Hadoop?
   1. The Apache Hadoop project develops open-source software for reliable, scalable, distributed computing. Hadoop includes:
   2. Hadoop Common utilities.
      1. Avro, a data serialization system with scripting languages.
      2. HDFS, a distributed file system.
      3. Hive, data summarization and ad hoc querying.
      4. MapReduce, distributed processing on compute clusters.
5. Master slave architecture
   1. Hadoop follows the master-slave architecture described in the original MapReduce paper.
   2. Since Hadoop consists of two parts—data storage (HDFS) and processing engine (MapReduce)—there are two types of master node.
   3. Types of node:
   4. For HDFS, the master node is namenode, and slave is datanode.
   5. For MapReduce in Hadoop, the master node is jobtracker, slave node is task tracker
   6. 

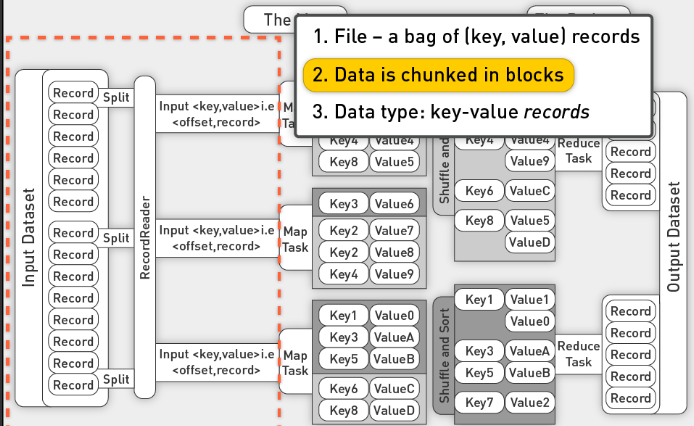
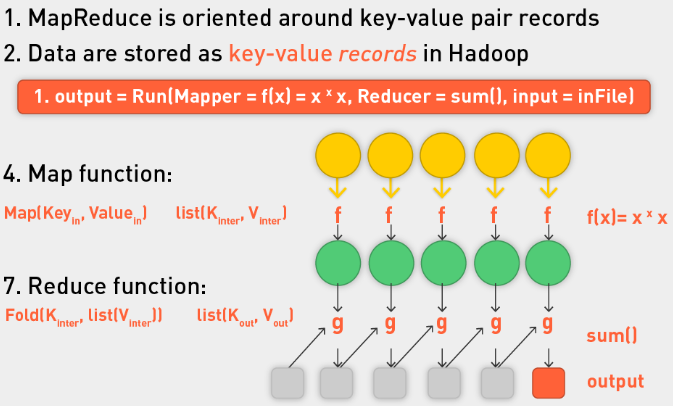
HDFS

1. HDFS
   1. Big files (Petabytes)
   2. Typical usage patterns
      1. Append only
      2. Rarely update in place
      3. Reads common
   3. Optimized for large files, sequential reads
   4. 
   5. Files split into blocks (chunks): 64-128MB, today 1-2GB recommended
   6. Blocks are replicated (usually three times) across several datanodes (called chunk or slave nodes)
   7. Computation is done at chunk node (close to data)
   8. Single namenode (master node) stores metadata (file names, block locations, etc.)
   9. May be replicated also
   10. Datanodes provide redundant store for the data blocks
   11. Client library provides file access
   12. Talks to master to find chunk servers
   13. Connects directly to chunk servers to access data
   14. Master nodes not a bottleneck
2. Hadoop Cluster
   1. 
   2. Take data from local computer, put it in HDFS
   3. We upload three files, they are distributed by namenode to datanodes
   4. 
   5. Default: 3x replication
   6. HDFS is Rack-aware when placing blocks
   7. Dynamic control for replication factor
   8. Balancer application to rebalance cluster in background
3. HDFS commands
   1. 
4. Write file in HDFS
   1. 
5. Summary:
   1. Hadoop file system
   2. Master-slave architecture
   3. Organized data
   4. How to get data in
   5. How to get data out
   6. HDFS goals
   7. Store large data sets
   8. Deal with hardware failures
   9. Emphasize streaming data access

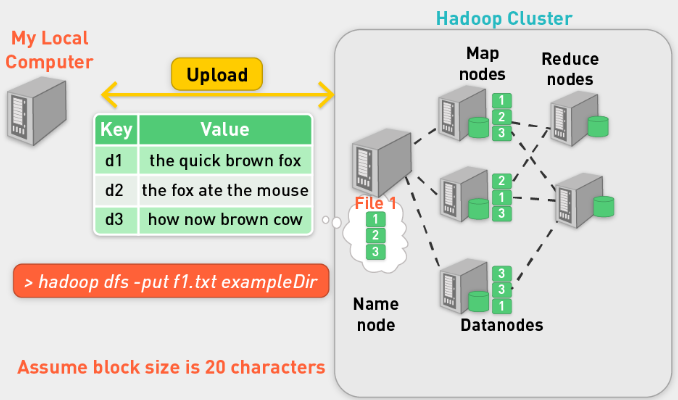
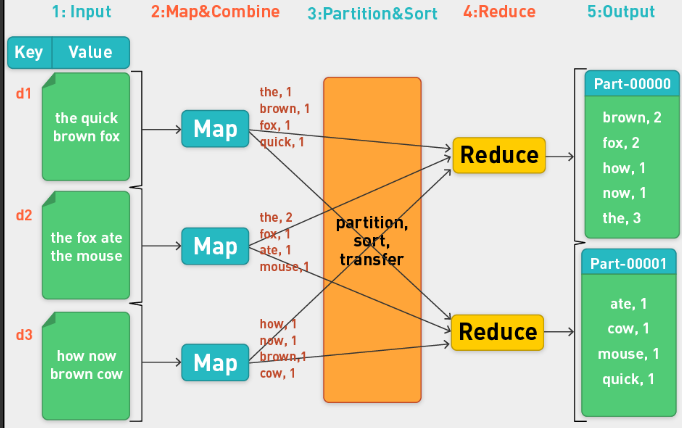
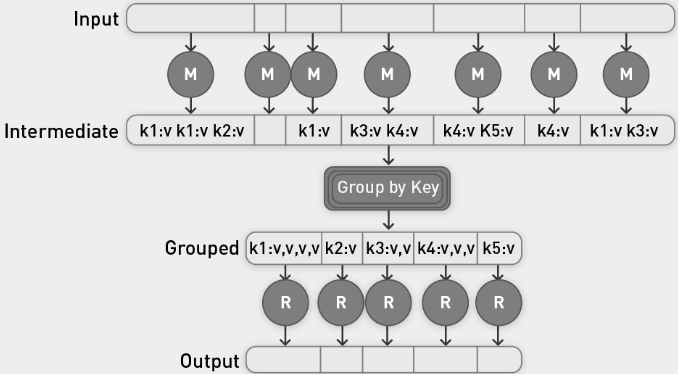
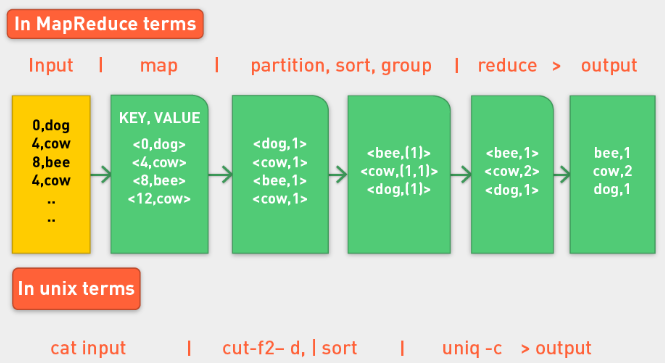
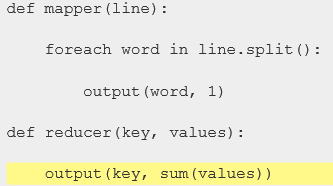
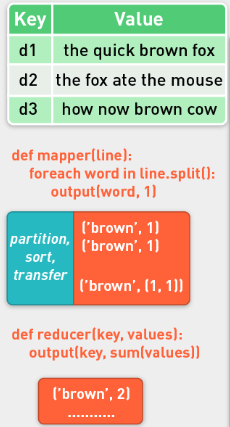
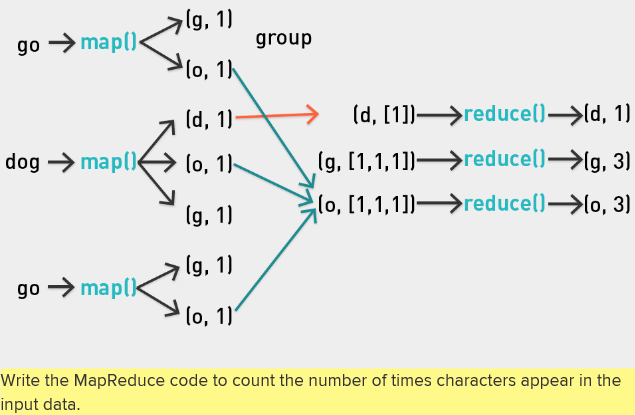
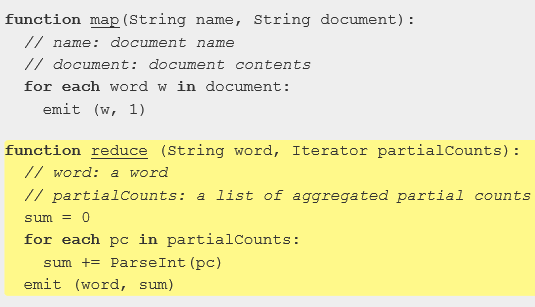
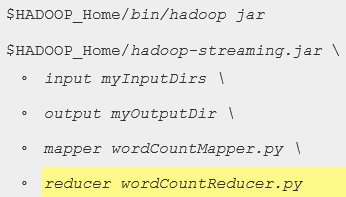
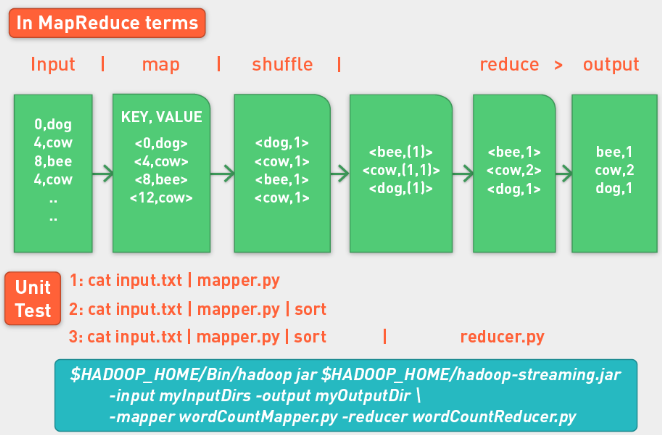
MR: Functional Programming

1. Functional Programming: M and R
   1. Key feature: higher-order functions that accept other functions as arguments
   2. Map and reduce are such functions
2. Map: A higher order function
   1. 
   2. Example in Haskell
   3. 
3. Reduce
   1. 
   2. 
   3. You need to pass a starting point, an operator and the array
   4. Fold left type of reduce functions can be defined recursively
4. MR from FP
   1. 
5. FP
   1. FP is all about the evaluation of mathematical functions and avoids changing state and mutable data.
   2. It is a declarative programming paradigm, which means programming is done with expressions.
   3. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function f twice with the same value for an argument x will produce the same result f(x) each time.
   4. This is an abstract diagram; despite this, you can see the potential for parallelism, especially in the map phase.
   5. It is lazy: only works when it requires an answer. Hadoop isn’t lazy.

Hadoop: MapReduce

1. From FP to MR
   1. Map corresponds and reduce corresponds approximately.
   2. The map phase in MapReduce roughly corresponds to the map operation in functional programming, whereas the reduce phase in MapReduce roughly corresponds to the fold operation in functional programming.
   3. MR framework takes care of the execution, data processing, storage (HDFS). MR does the rest.
2. Data
   1. 
   2. Decompose large file into records
   3. Delimited by carriage return
   4. Split into records that are fed into a map task
   5. The map task creates a key and value pair
   6. Mapper operates on each record, produces output that is shipped off to reducers
   7. Reducer accepts a key and a list of values with the same key
   8. Process list of values and key, stores result on disk
   9. What is restriction of key/val pair: Each pair must be small enough to fit into one single machine; e.g., a document should be fine, an image should be fine, but 1PB value is probably not.
3. Mapper
   1. takes key-value pair
   2. produces key-value pair
   3. framework uses protocol to move data from mapper to reducer: each record with same key value is routed to same reducer
   4. divide and conquer in both mapper and reducer
4. Reducer
   1. Gets list of values associated with a key
   2. Produces output as result.
5. MR Example Sum (X^2)
   1. 
   2. We can reduce to multiple key values, not just one
6. MR
   1. Framework for big data
   2. Codify generic recipe for processing large data sets that consist of two stages
   3. Programmer and execution framework synergy
   4. Just provide the mapper and reducer functions
   5. The programmer defines these two types of computations, and the execution framework coordinates the actual processing (very loosely, MapReduce provides a functional abstraction)
   6. Very powerful: Many interesting algorithms can be expressed quite concisely
   7. Although such a two-stage processing structure may appear to be very restrictive, many interesting algorithms can be expressed quite concisely, especially if one decomposes complex algorithms into a sequence of MapReduce jobs
7. Summary
   1. MR cluster of machines
   2. Each machine is storage node and computational node
   3. The framework will convert each record of input into a key-value pair.
   4. The framework will convert each record of input into a key-value pair, and each pair will be input to the map function once.
   5. The map output pairs are grouped and sorted by key.
   6. The map output is a set of key-value pairs—nominally, one pair that is the transformed input pair, but it is perfectly acceptable to output multiple pairs.
   7. The reduce function is called one time for each key, in sort sequence, with the key and the set of values that share that key.
   8. Reduce outputs to file:
   9. The reduce method may output an arbitrary number of key-value pairs, which are written to the output files in the job output directory.
   10. If the reduce output keys are unchanged from the reduce input keys, the final output will be sorted.

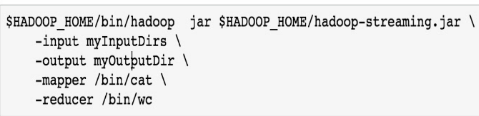
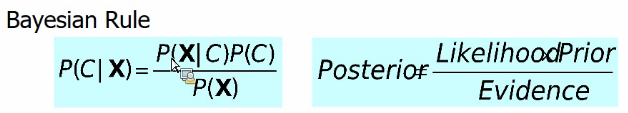
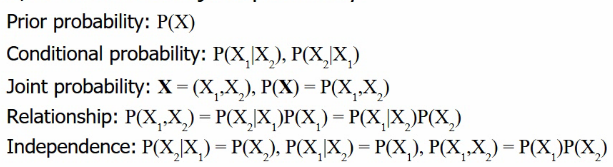
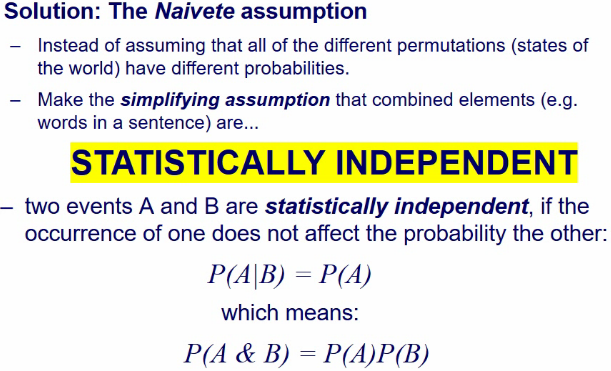
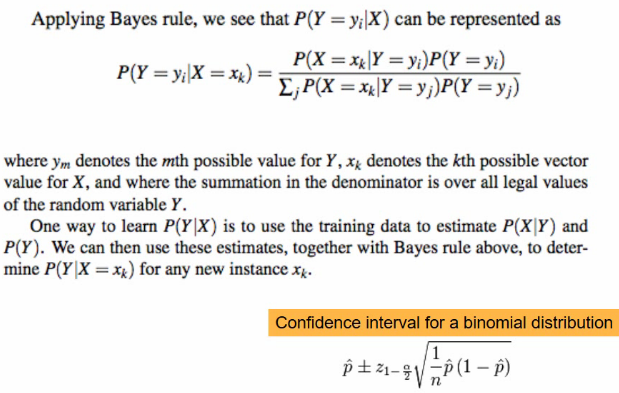
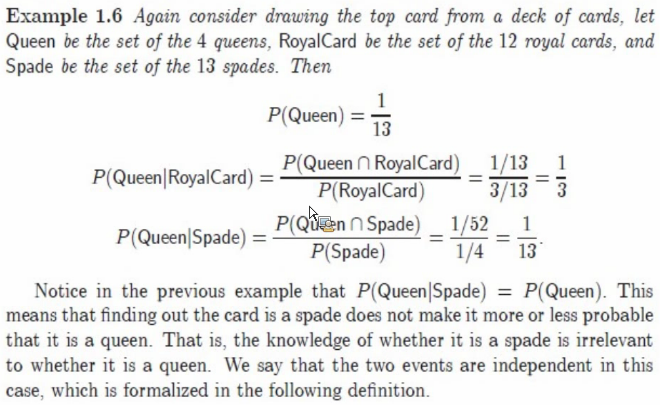
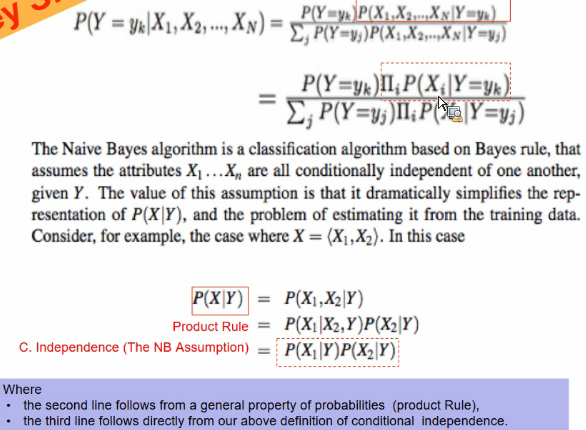
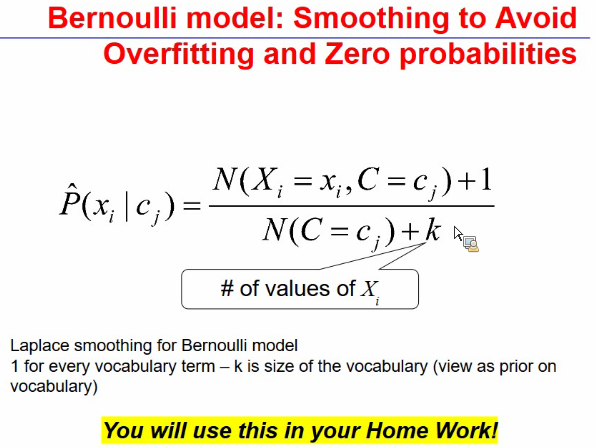
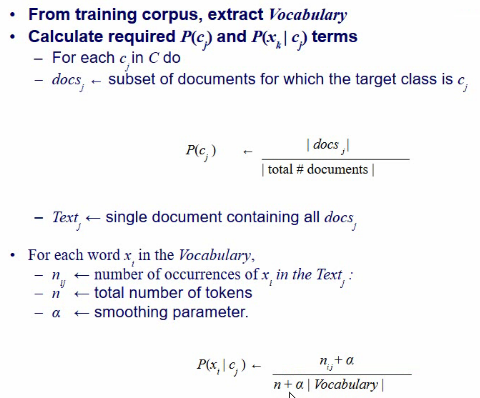
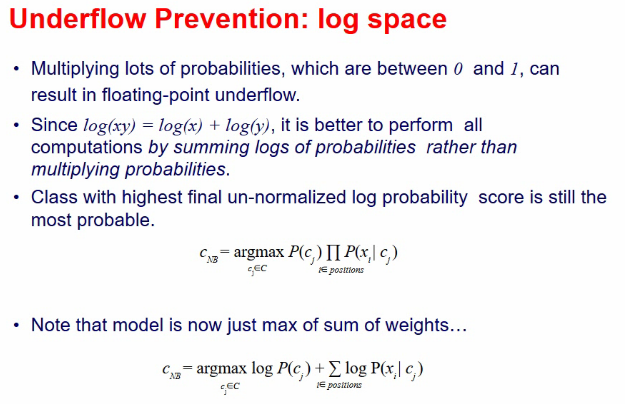
Animated Examples

1. Count example
   1. Count words in each example
   2. We have three documents
   3. 
   4. Use hdfs put to upload data, each file gets its own node in this case
   5. 
   6. Key is d1, d2, d3 identifiers
   7. Map produces (key,val) as (word,intermediateCount)
   8. Framework partitions data so that records with the same key routed to the same reducer.
   9. Take key (brown) and sum up individual values for that word.
   10. Two parts of the output file, one for each reducer.
   11. There is a record for each word
2. High Level
   1. Specify
   2. Mapper
   3. Reducer
   4. Where data is
   5. Where output should be
   6. 
   7. Group by the key of the key,value pairs.
3. MR: Word Count from a Word Stream
   1. 
   2. Unix commands replicate functionality of MR world
4. Word Count Pseudo Code
   1. 
   2. Python:
   3. 
   4. Java:
   5. 
5. Contract with the programmer
   1. The frozen part of the MapReduce framework is a large distributed sort. The hot spots, which the application defines, are:
      1. An input reader
      2. A map function\*
      3. A partition function
      4. A compare function
      5. A reduce function\*
      6. An output writer
      7. \*Required from programmer (generally)
6. Character count
   1. 
7. Hadoop: Native Java or Streaming
   1. Although the Hadoop framework is implemented in Java, MapReduce applications need not be written in Java.
   2. Hadoop provides an API to MapReduce that allows you to write your map and reduce functions in languages other than Java (e.g., Python, Ruby, Perl, etc.).
   3. **Hadoop Streaming** uses Unix standard streams as the interface between Hadoop and your program, so you can use any language that can read standard input and write to standard output to write your MapReduce program.
   4. Hadoop Streaming is a utility that allows users to create and run jobs with any executables (e.g., shell utilities) as the mapper or the reducer.
8. Python example with Hadoop Streaming
   1. 
   2. 
   3. Hadoop Streaming is a utility that comes with the Hadoop distribution.
   4. The utility allows you to create and run MapReduce jobs with any executable or script as the mapper or reducer.
   5. 
9. Word count stream
   1. 
   2. Unit test mapper and reducer

Summary

1. Parallel Computing
   1. Motivation for parallel computing
   2. Parallel computing (PC)
   3. Definition, communication and synchronization, types of PC tasks
   4. Architectures for parallel computation
   5. Developer frameworks for parallel computation
2. Hadoop
   1. Background and history
   2. Hadoop Distributed File System (HDFS)
   3. MapReduce
   4. Functional programming
   5. MapReduce
   6. Animated examples
   7. Hadoop in practice
   8. Full-code examples
   9. Word count example on local machine
   10. Word count example on cluster
   11. MapReduce: Runtime environment

## Lecture:

* Treat function like a parameter or variable, input to another function
* Proximity function: like Euclidean distance, for example; you can pass in different proximity functions to other functions
* Substitute in different functions for the mapper and the reducer.
* Map applies same function to each element that is iterated
* Reducer: iterates over fixed values for a single key
* Python apply(func, args) applies a function to any number of args
* Python filter() function takes function and iterable and returns a list. Function must return true or false for each element of the iterable. If true, element enters list that will be returned
* Reduce(function, iterable[,initializer]): initializer can sit as the first argument in the sum, function application. We get a single output
* Shortly after Google MR paper, a lot of people release paper for MR and ML.
* Classifying spam
  + We have email that’s classified as spam or ham
  + Divide them in two piles
  + Count frequency of words in spam pile and frequency of words in ham pile and compare
* Hadoop Streaming
  + Comes from Hadoop distribution
  + 
  + We need to define mapper and reducer (python functions)
  + Shuffle phase sends data to reducer in right order. Hadoop really good at sorting, let it do it.
* Bayes Rule
* 
* 
* Interested in classification, so probability of C=spam or ham given the example.
* Easier to count probability of seeing these words, given the classification, we know this because we have labelled data. P© just count examples with this classification. P(X) count occurrence of these words. To get it for a list of words, we make bayes assumption of independence. Therefore P(X|C)=P(word1|C)\*P(word2|C)
* 
* Learning classifiers based on Bayes Rule
* 
* To get P(X=x1|Y=y1), count number of time the word x1 appears in emails classified as y1 and divide by the total number of words in emails classified as y1
* 
* 
* 
* After you do counts but before doing multiplications, take log of values.
* 
* Will this normalizer work the right way?
* This stops you multiplying by zero for words that don’t appear often. This also stops overtraining.
* 
* Alpha equal to one, we get previous slide
* 
* Best to sum logs of probabilities than multiplying probabilities.
* See rest of slides for more details on NB and HW2.

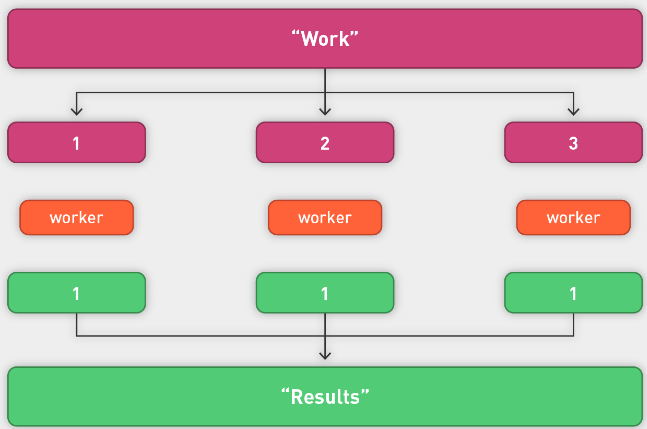
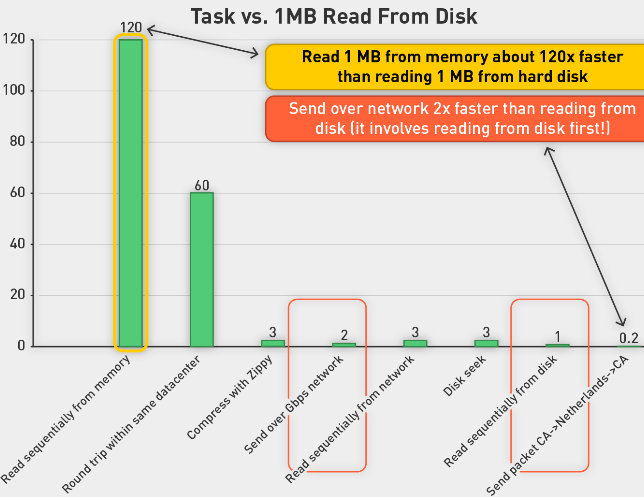
# Unit 3: MapReduce Algorithm Design

## Videos:

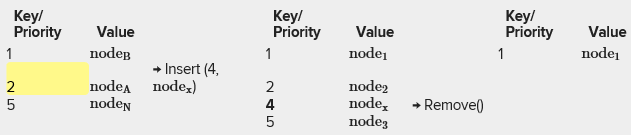
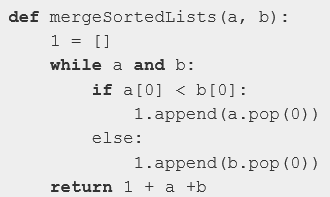
Introduction

1. Contract with programmer
   1. Data
   2. Mapper
   3. Reducer
   4. Optional: combiner, partitioner
2. All else is handled transparently by framework
3. Design Patterns for MapReduce
   1. Design mappers
   2. Design reducers
   3. Combiners if we need
4. When and how to partition data
5. Useful for ML
6. Relationship between
   1. Memory
   2. Disk
   3. Bandwidth
7. Data structures
8. Hadoop Shuffle
9. Local aggregation
10. Pairs and stripes
11. Secondary sorting
12. Scaling tricks

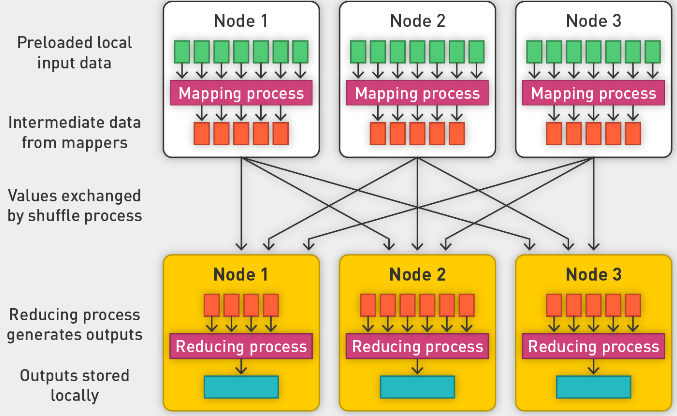
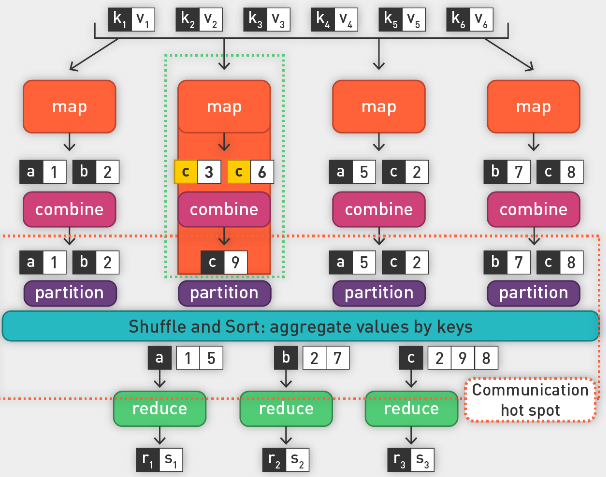
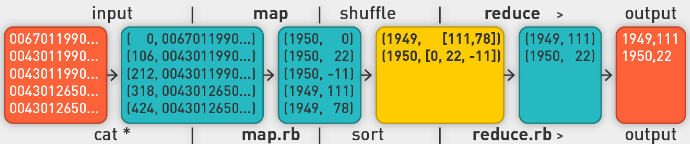
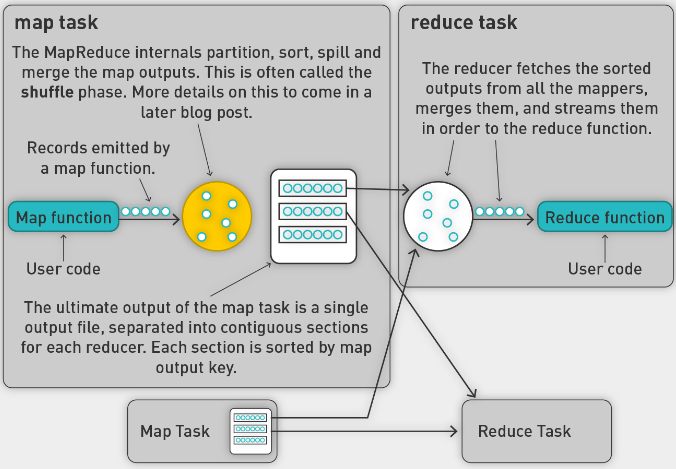
Background: RAM vs Disk vs Bandwidth

1. Hadoop Workflow
2. Divide and conquer
   1. 
   2. Adds burden to sync and communicate results
   3. Hadoop takes care of a lot of this sync and communication, as well as fault tolerance
   4. Other systems are compute-centric, but don’t cater for big data
3. Sync and coordinate
   1. Modeled after Lisp primitives:
   2. Map (apply function to all items in a collection)
   3. Reduce (apply function to set of items with a common key)
   4. Synchronization and coordination in distributed tasks is key
   5. E.g., reducer is a big target of communication and synchronization (barrier between map and reduce)
   6. Need to get all data for a particular key to the same location
   7. How to achieve sync?
      1. Keys
      2. Sorting and secondary sorting
      3. Partitioning
      4. Framework or in-memory mappers or reducers
4. Communicate through files and streams
   1. Abstraction: Instead of sending messages, different pieces of code communicate through files
   2. Code is going to take a very "stylized" form; at each stage each machine will get input from files, send output to files
   3. Files are generally persistent, nameable (in contrast to distributed hash table messages, which are transient)
   4. Files consist of blocks, which are the basic unit of partitioning (in contrast to object or data item IDs)
   5. Server is stateless, so clients must send all context (including position to read from) in each request
5. Reading data from Disk
   1. Also transfer data over network
   2. Sometimes quickly read from memory (100x faster)
   3. 
   4. Transferring is faster than disk, but usually followed by write to disk.
   5. Try to put in memory

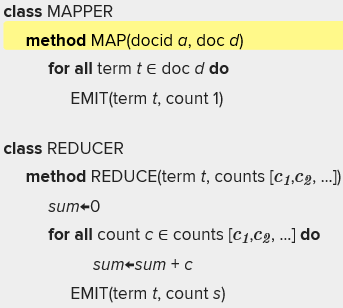
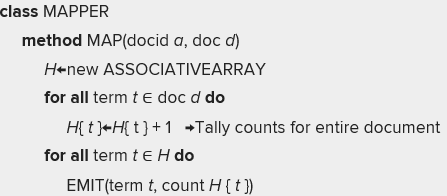
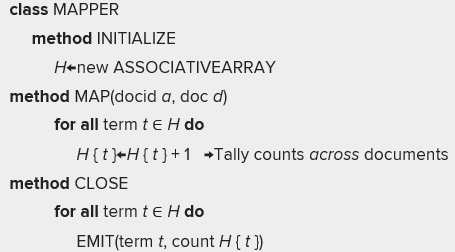
Background: Priority Queues and Merge Sort

1. Priority queue
   1. Queue: list of elements to be consumed by a thread
   2. LIFO, FIFO
   3. Each element has priority in priority queue
   4. 
   5. First element of queue has highest priority
   6. Main methods of the priority queue abstract data type:
   7. insert(k, x): Inserts an entry with key k and value x
   8. removeMin(): Removes and returns the entry with smallest key
   9. Maintained priority queue always sorted
   10. A priority queue stores a collection of entries, where each entry is a pair (key, value), where the key is priority; these entries are sorted in order of decreasing priority
   11. Priority queues used throughout Hadoop
2. Merge Two Sorted Lists or Arrays
   1. We have three pointers:
      1. Start of first sorted list
      2. Start of second sorted list
      3. Start of merged list
   2. Keep picking the smallest element and move it to a temp array, incrementing corresponding indices.
   3. 
3. Merge algorithms
   1. Complexity of O(mlogn)
   2. Given n sorted lists, let m= sum of lengths of the lists
   3. Merge can use a heap-based priority queue : Merge algorithms that operate on large numbers of lists at once will multiply the sum of the lengths of the lists by the time to figure out which of the pointers points to the lowest item, which can be accomplished with a heap-based priority queue in O(log n) time, for O(m log n) time, where n is the number of lists being merged and m is the sum of the lengths of the lists. When merging two lists of length m, there is a lower bound of 2m − 1 comparisons required in the worst case.

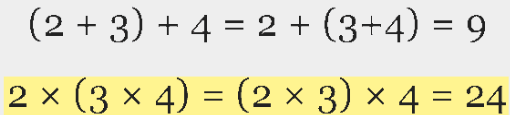
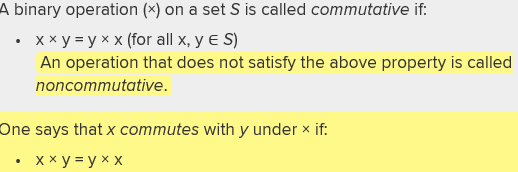
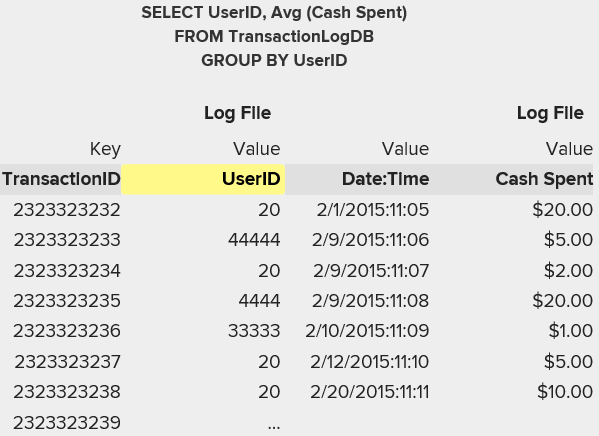
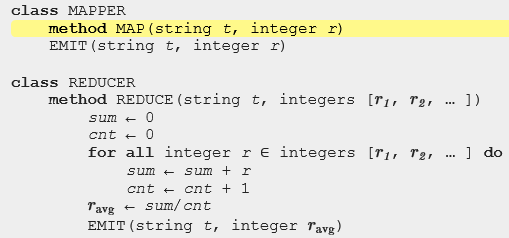
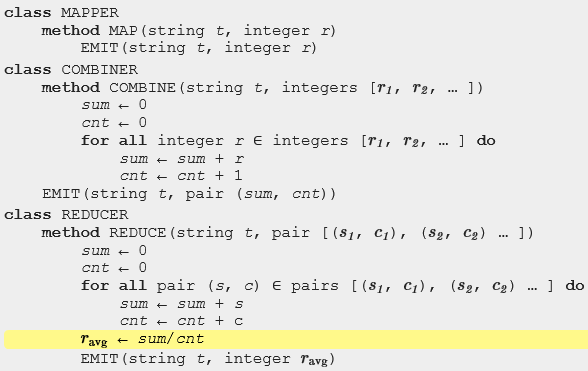
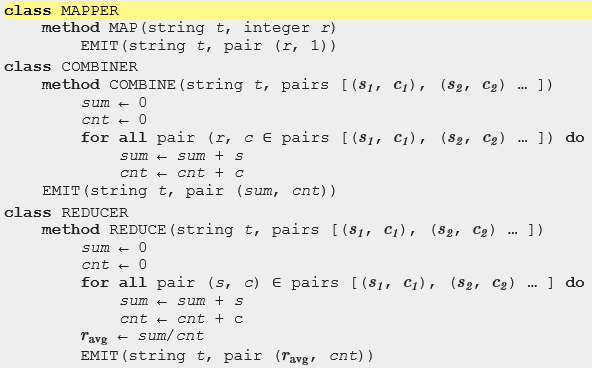
Background: Internals of Hadoop Shuffle

1. Recap of MR
   1. Specify map, reduce, partition and combine function
   2. Partition (k', number of partitions) → partition for k'
      1. Often a simple hash of the key, e.g., hash(k') mod n
      2. Divides up key space for parallel reduce operations
   3. Combine (k', v') → <k', v'>\*
      1. Mini-reducers that run in memory after the map phase
      2. Used as an optimization to reduce network traffic
   4. Framework deals with
      1. Scheduling: Assigns workers to map and reduce tasks
      2. "Data distribution": Moves processes to data
      3. Synchronization: Gathers, sorts, and shuffles intermediate data
      4. Errors and faults: Detects worker failures and restarts
      5. Limited control over data and execution flow
      6. All algorithms must be expressed in m, r, c, p
   5. You don't know:
      1. Where mappers and reducers run
      2. When a mapper or reducer begins or finishes
      3. Which input a particular mapper is processing
      4. Which intermediate key a particular reducer is processing
   6. A lot going on under the covers
2. Communication Hot spot: transferring between M and R
   1. 
   2. Records with same key must be processed by same reducer
   3. Incur a lot of communication traffic.
3. In-Mapper combiner
   1. A mapper-side function allows us to combine records with the same key
   2. 
   3. Reducing the number of tuples to be processed by shuffle and reducers.
4. Shuffle: Heart of MR
   1. Makes sure input of every reducer is sorted by key
   2. We can take control of picking the key
   3. Performing the sort and transferring the data is know as the shuffle
   4. Partition, sort, combine: do in memory if possible, otherwise on disk
   5. 
5. Partition, sort and combine
   1. 
   2. In memory:
      1. Partition output of mapper into separate files (one for each reducer)
      2. Can look at first letter of key and put in its own bucket: 26 buckets.
      3. MR has internal partition functions based on hashing
6. Shuffle on disk
   1. Map side: Map outputs are buffered in memory in a circular buffer (in-memory shuffle, i.e., partition, sort, combine). First task is to assign output record to partition. Records in same partition are sorted. We use combine function to combine records of the same key. Then pass to reducer.
   2. When buffer reaches threshold, sorted contents are spilled to disk
   3. Spills (sorted) are merged in a single, partitioned file (sorted within each partition)
   4. Combiner runs on the sorted partition file
   5. We do this continually.
   6. Make sure all spills are sort-merged.
   7. We can transfer sorted partition file to reducer
   8. Reducer side: receive multiple sorted files from multiple mappers. "Sort" is a multipass merge (merge-sort) of map outputs (happens in memory and on disk); combiner runs here.
   9. Produce stream of values which can be put in the reducer input stream.
7. Things we control
   1. When the contents of the buffer reach a certain threshold size (mapreduce.map.sort.spill.percent, which has the default 0.80, or 80%), a background thread will start to spill the contents to disk.
   2. Map outputs will continue to be written to the buffer while the spill takes place, but if the buffer fills up during this time, the map will block until the spill is complete.
8. Summary
   1. 

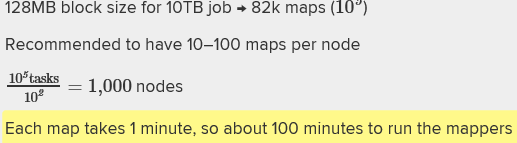
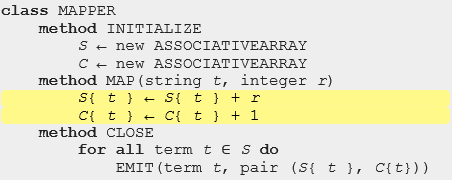
Local Aggregation Combiners and In-Mapper Combining

1. Divide and Conquer
   1. Ideal scaling characteristics (linear scaling):
      1. Twice the data, twice the running time
      2. Twice the resources, half the running time
   2. Why can't we achieve this?
      1. Communication kills performance
      2. Synchronization requires communication
   3. Thus … avoid communication?
      1. Reduce intermediate data via local aggregation
      2. Combiners, in-memory mappers, and reducers can help
2. Individual M and R
   1. All mappers and reducers have a common goal.
   2. To solve a job, they need to work as a team.
   3. Teamwork requires communication and synchronization.
   4. Mappers and reducers run in isolation without any mechanisms for direct communication.
   5. Synchronization or merging of results can be the most tricky aspect of designing MapReduce algorithms (or, for that matter, parallel and distributed algorithms in general).
   6. Within a single MapReduce job, there is only one opportunity for cluster-wide synchronization—during the shuffle and sort stage where intermediate key-value pairs are copied from the mappers to the reducers and grouped by key.
3. Local Aggregation
   1. In the context of data-intensive distributed processing, the single most important aspect of synchronization is the exchange of intermediate results from the processes that produced them to the processes that will ultimately consume them.
4. Tools for Sync
   1. Cleverly constructed data structures for bringing partial results together
   2. Sort order of intermediate keys: control order in which reducer processes keys
   3. Partitioner: control which reducer processes which keys
   4. Preserving state in M and R: captures dependencies across multiple keys and values
5. Word count: Baseline
   1. 
   2. Emit each word … heavy disk storage requirements on mapper side; big sorts; big transfer from mapper to reducer
   3. What's the impact of combiners?
   4. Can reduce storage and transfer requirements by 50%
6. Other forms of local optimization
   1. Data has gravity, heavy to move around network
   2. In-mem mappers and reducers?
7. Mappers are Java Objects with Init and State
   1. In Hadoop, mappers are Java objects with a map method (among others).
   2. A mapper object is instantiated for every map task by the task tracker.
   3. The life cycle of this object begins with instantiation, where a hook is provided in the API to run programmer-specified code.
   4. This means that mappers can read in "side data," providing an opportunity to load state, static data sources, dictionaries, etc.
   5. After initialization, the map method is called (by the execution framework) on all key-value pairs in the input split.
   6. Since these method calls occur in the context of the same Java object, it is possible to preserve state across multiple input key-value pairs within the same map task. This is an important property to exploit in the design of MapReduce algorithms, as we will see in the next chapter.
   7. After all key-value pairs in the input split have been processed, the mapper object provides an opportunity to run programmer-specified termination code. This, too, will be important in the design of MapReduce algorithms.
   8. The actual execution of reducers is similar to that of the mappers.
8. WC V1: Record-level combiner
   1. Keep local tally of values that can be emitted
   2. 
   3. Issues:
      1. Disk
      2. CPU
      3. Memory
   4. We still need combiners to combine instances from different mappers.
9. WC V2: In-memory mapper combiner
   1. 
   2. For each mapper, we create an associative array
   3. Each document that is processed increments counts in the hash table
   4. Resolves: Disk, CPU and network issues, but has memory issues as HT may grow to be very big for a huge corpus of words
   5. Combining has small footprint on disk and on network, but memory issues.
   6. You still need combiners on reducer side
   7. Combiners can be run to consolidate results before putting them into the REDUCE stream.
10. Design pattern for local aggregation
    1. "In-mapper combining"
    2. Fold the functionality of the combiner into the mapper by preserving state across multiple map calls
    3. Advantages
    4. Speed
    5. Why is this faster than actual combiners?
    6. Disadvantages
    7. Explicit memory management required
    8. Potential for order-dependent bugs

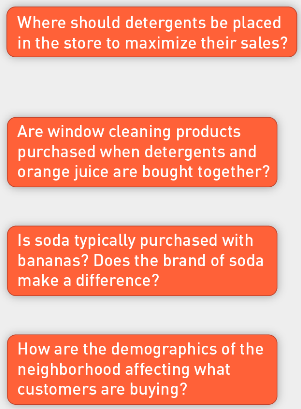
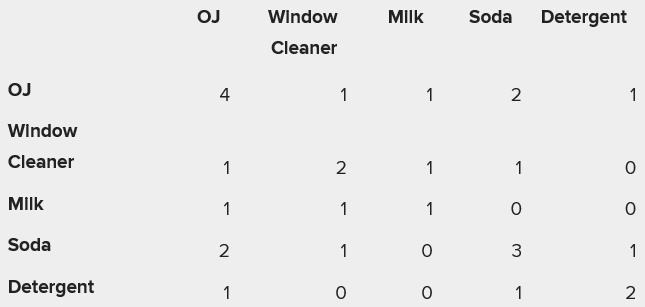
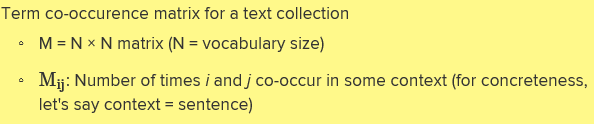
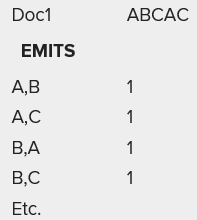
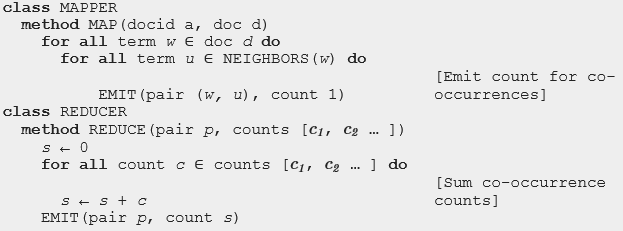
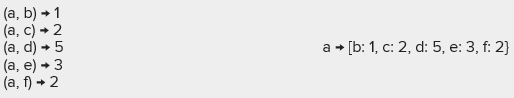
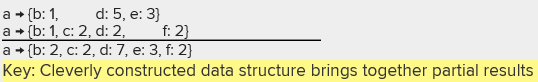
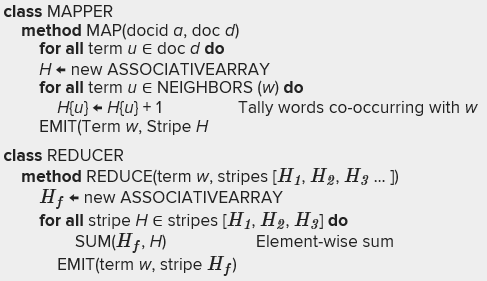
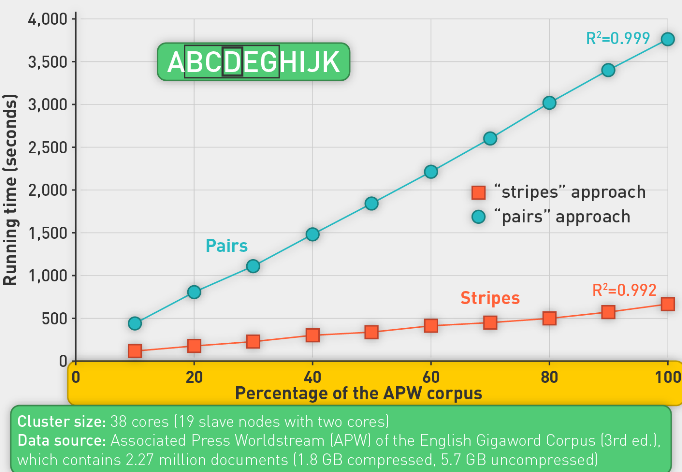
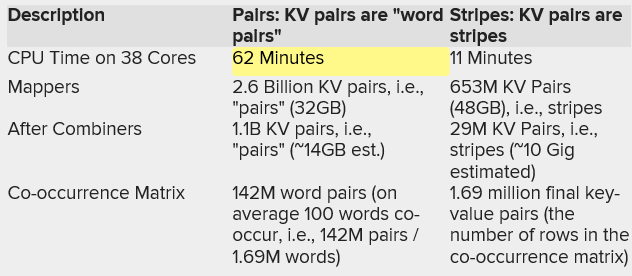
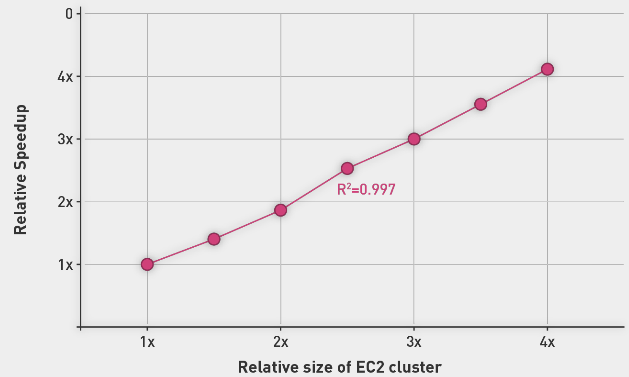
Local Aggregation: Algorithmic correctness

1. Associative (Grouping) property
   1. Within an expression containing two or more occurrences in a row of the same associative operator, the order in which the operations are performed does not matter as long as the sequence of the operands is not changed.
   2. That is, rearranging the parentheses in such an expression will not change its value.
   3. 
2. Commutative Property
   1. In mathematics, a binary operation is commutative if changing the order of the operands does not change the result. The term commutative is used in several related senses.
   2. 
3. Combiner design
   1. Both associative and commutative
   2. Combiners and reducers share the same method signatures
   3. They are optional optimizations, so should not affect algorithm correctness
   4. Hadoop makes no guarantees; could be run 0,1 or multiple times
   5. Combiners need to be associative and commutative
   6. Example: Find average of all integers associated with the same key
4. Simple Problem
   1. We have a large dataset where input keys are strings and input values are numeric, and we wish to compute the mean of all numerics associated with the same key.
   2. Customer log file (e.g., purchase transactions)
   3. A real-world example might be a large user log from a popular website, where keys represent user ids and values represent some measure of activity, such as elapsed time for a particular session. The task would correspond to computing the mean session length on a per-user basis, which would be useful for understanding user demographics.
   4. 
5. Computing the Mean: V1, no combiner
   1. 
   2. Can we use reducer as combiner? No, mean is not associative operation:
   3. 
   4. Challenge: How can we transform a nonassociative operation (mean of numbers) into an associative operation?
   5. (Element-wise, sum of a pair of numbers, and a count, with an additional division at the very end)
6. Computing the Mean: V1, combiner
   1. 
   2. Combiner is mirror image of reducer without the division step. We are doing a local accumulation for transactions for each user ID.
   3. However, there is no guarantee that we get combiner at all. If we don’t call it, there will be a mismatch between mapper output and reducer input.
7. Computing the Mean: Version 3
   1. 
   2. Combiners and reducers share the same method signature.
   3. Remember: Hadoop makes no guarantees; could be run zero, one, or multiple times.
   4. Combiners need to be associative and commutative.
   5. Note that although the output key-value type of the combiner must be the same as the input key-value type of the reducer, the reducer can emit final key-value pairs of a different type.
8. Solution to our challenge
   1. How can we transform a nonaassociative operation (mean of numbers) into an associative operation?
   2. Choose a combiner operation that is commutative and associative (addition/sum is!)
   3. Element-wise sum of a pair of numbers, with an additional division at the very end

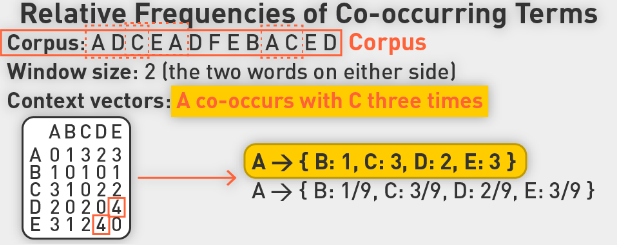
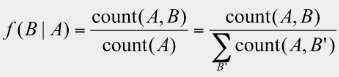
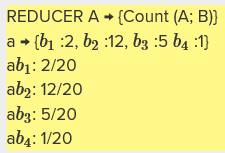
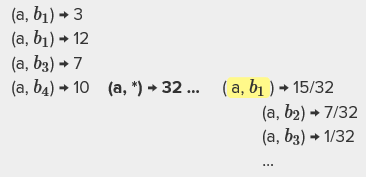
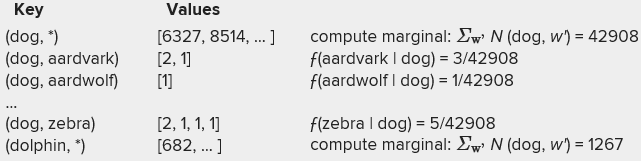
How Many Maps and Reduces?

1. Hadoop Wiki: <http://wiki.apache.org/hadoop/HowManyMapsAndReduces>
2. How many Maps and Reduces in cluster?
   1. Picking the appropriate size for the tasks for your job can radically change the performance of Hadoop.
   2. Increasing the number of tasks increases the framework overhead but increases load balancing and lowers the cost of failures.
   3. At one extreme is the one map/one reduce case, where nothing is distributed. The other extreme is to have 1,000,000 maps/1,000,000 reduces, where the framework runs out of resources for the overhead.
   4. The number of reduce tasks can also be increased in the same way as the map tasks, via **JobConf's conf.setNumMapTasks(int num).**
   5. The number of maps is usually driven by the number of DFS blocks in the input files.
   6. Although that causes people to adjust their DFS block size to adjust the number of maps.
   7. The right level of parallelism for maps seems to be around 10–100 maps/node, although we have taken it up to 300 or so for very CPU-light map tasks. Task setup takes a while, so it is best if the maps take at least a minute to execute.
   8. E.g., if you expect 10TB of input data and have 128MB DFS blocks, you'll end up with 82k maps, unless your mapred.map.tasks is even larger.
   9. 128MB block size for 10TB job → 82k maps 10–100 maps per node; each map takes one minute
3. Back-of-the-Envelope Estimates: 10TB
   1. 
   2. Number of reducers
   3. The ideal reducers should be the optimal value that gets them closest to:
   4. A multiple of the number of blocks
   5. A task time between 5 and 15 minutes
   6. Creating the fewest files possible
   7. The number of reduce tasks can also be increased in the same way as the map tasks, via JobConf's conf.setNumReduceTasks(int num).
4. How many?
   1. Empirical question best solved by taking this guidance and experimenting with some small jobs and establishing the length of map and reduce tasks at different block sizes, etc.
   2. Block sizes of 128MB to 1GB
5. Computing the mean: V4, In-memory mapper
   1. 
   2. Super-efficient: no shuffling!
   3. Inside the mapper, the partial sums and counts associated with each string are held in memory across input key-value pairs. Intermediate key-value pairs are emitted only after the entire input split has been processed; similar to before, the value is a pair consisting of the sum and count. The reducer is exactly the same as above.
   4. We can run out of memory easily. Do we need a combiner? Yes, I think they are still needed.

Pairs and Stripes

1. Sync in streams!
   1. Change example and change gears here to introduce some more design patterns for communication efficiency and for synchronization
   2. Previous example: Grouping by customer ID to get average amount spent by customer
   3. Next we want to explore more challenging examples that are combinatorially explosive before synchronizing down to a solution space
   4. One common approach for synchronization in MapReduce is to construct complex keys and values in such a way that data necessary for a computation are naturally brought together by the execution framework
   5. More complex value
   6. We first touched on this technique in the previous section, in the context of packaging partial sums and counts in a complex value (i.e., pair) that is passed from mapper to combiner to reducer
2. Market basket analysis
   1. 
   2. Correlations and patterns are interesting
3. Co-occurrence matrix
   1. 
   2. In how many baskets did window cleaner and OJ appear together? Only one.
4. Algorithm design: running example
   1. 
   2. Corpus: A D C E A D F E B A C E D
   3. Window size: 2 (the two words of either side)
   4. 
   5. Using context vectors, you can get co-occurrences very easily
   6. Co occurrence of D and E is D[E]=4
   7. Why build it? Distributional profiles as a way of measuring semantic distance, which is useful for many language-processing tasks
5. MR and large counting problems
   1. Term co-occurrence matrix for text collection = specific instance of a large counting problem
   2. A large event space (number of terms)
   3. A large number of observations (the collection itself)
   4. Goal: Keep track of interesting statistics about the events
   5. Basic approach
   6. Mappers generate partial counts
   7. Reducers aggregate partial counts
   8. Combinatorial event space and large number of observations
   9. How do we aggregate partial counts efficiently?
6. CO Matrix: V1 Pairs
   1. Each pair corresponds to a cell in the word co-occurrence matrix. This algorithm illustrates the use of complex keys in order to coordinate distributed computations.
   2. Each mapper takes a sentence:
   3. Generate all co-occurring term pairs
   4. For all pairs, emit (a, b) → count
   5. 
   6. Reducers sum up counts associated with these pairs.
   7. Use combiners!
   8. Each pair corresponds to a cell in the word co-occurrence matrix. This algorithm illustrates the use of complex keys in order to coordinate distributed computations.
   9. 
   10. Outer loop iterates over all words, inner loop iterates over neighbours of current word
7. Above is Pairs analysis
   1. Advantages
   2. Easy to implement, easy to understand
   3. Can use reducer as a combiner!
   4. Disadvantages
   5. Lots of pairs to sort and shuffle around (upper bound?)
   6. Not many opportunities for combiners to work
   7. How can we overcome shuffle overhead?
8. Stripes approach
   1. Change mapper
   2. Group together pairs in associative array
   3. 
   4. Each mapper takes a sentence:
   5. Generates all co-occurring term pairs
   6. For each term, emits a → { b: ountb , c: countc , d: countd … }
   7. Reducers perform element-wise sum of associative arrays
   8. 
   9. 
   10. Memory needs to be managed very carefully
9. Performance
   1. Stripes has a bigger value per key.
   2. Pairs has more partition and sort overhead.
   3. Stripes has
      1. Less sorting and shuffling
      2. Better use of combiners
   4. Stripers is limited by
      1. More difficult to implement
      2. Underlying object more heavyweight
      3. Fundamental limitation in terms of size of event space
10. Experimental Data Set
    1. Associated Press Worldstream (APW) of the English Gigaword Corpus (3rd ed.), which contains:
    2. 2.27 million; 1.8 GB compressed; 5.7 GB uncompressed
    3. Would push any laptop to its limits
    4. Preprocessing
    5. All XML markup was removed, followed by tokenization and stop-word removal using standard tools from the Lucene search engine
    6. All tokens were then replaced with unique integers for a more efficient encoding
    7. Goal to generate co-occurrence matrix
11. Performance
    1. 
    2. Stripes approach outperformed pairs approach in running time
    3. Six times better
12. Analysis
    1. 
    2. Pairs: The mappers in the pairs approach generated 2.6 billion intermediate key-value pairs totaling 31.2 GB. After the combiners, this was reduced to 1.1 billion key-value pairs, which quantifies the amount of intermediate data transferred across the network. In the end, the reducers emitted a total of 1.42 million final key-value pairs (the number of nonzero cells in the co-occurrence matrix).
    3. Stripes
       1. Mappers emitted 653M KV Pairs (i.e., stripes) (48GB) but after combiners this reduced to 29 Million stripes (extrapolate that the amount of data transferred from mappers to reducers is less 10GB).
       2. As expected, the stripes approach provided more opportunities for combiners to aggregate intermediate results (only a simple key consisting of a single-word (and not two words as is the case in the "pairs" approach), thus greatly reducing network traffic in the shuffle and sort phase.
    4. As expected, the stripes approach provided more opportunities for combiners to aggregate intermediate results, thus greatly reducing network traffic in the shuffle and sort phase.
13. Running time of stripes algorithm
    1. 
    2. Here we vary size of ec2 cluster (number of slave instances)
    3. We are seeing a linear reduction of time required
    4. 
    5. RHS recasts the same results to illustrate scaling characteristics. The circles plot the relative size and speedup of the EC2 experiments, with respect to the 20-instance cluster.
    6. These results show highly desirable linear scaling characteristics (i.e., double the cluster size makes the job twice as fast).
    7. This is confirmed by a linear regression with an R2 value close to 1.
    8. Caution: use it with care

Computing Relative Frequencies

1. Relative frequencies
   1. 
   2. 
   3. This gets us relative frequencies
2. Stripes
   1. 
   2. Easy to do apparently
   3. One loop over all value terms (co-occurring words) to compute (a, \*). Another pass to directly compute the relative frequency f(B|A)
3. Pairs approach
   1. Note: In the pairs approach, the reducer receives (wi;wj) as the key and the count as the value. From this alone, it is not possible to compute ƒ(wj jwi), since we do not have the marginal.
   2. Synchronization: Need to reconstruct the list of co-occurring terms with the term of interest
   3. Fortunately, as in the mapper, the reducer can preserve state across multiple keys
   4. Inside the reducer, we can buffer in memory all the words that co-occur with wi and their counts, in essence building the associative array in the stripes approach
4. Custom partitioner: sync word counts for word of interest
   1. Repartition data according to primary token (first word) only.
   2. We must ensure that all pairs with the same left word are sent to the same reducer.
   3. This, unfortunately, does not happen automatically. Recall that the default partitioner is based on the hash value of the intermediate key, modulo the number of reducers.
   4. For a complex key, the raw byte representation is used to compute the hash value. As a result, there is no guarantee that, for example, (dog, aardvark) and (dog, zebra) are assigned to the same reducer.
   5. To produce the desired behavior, we must define a custom partitioner that pays attention only to the left word
   6. Must make sure all a's get sent to same reducer (use custom partitioner)
   7. Must make sure (a, \*) comes first (define sort order)
   8. Must hold state in reducer across different key-value pairs
   9. 
   10. Partition based on primary token, but sort on entire key. This gives us a stream that has same primary token and same co-occurring token appearing together.
   11. 
   12. Reducer has co-occurring words and frequencies in memory
   13. How can we avoid the memory issues?
5. Order Inversion pattern
   1. Use framework to count the total occurrences in the primary token. We want it to come first in the reducer stream.
   2. First emit (dog,\*), so reducer has the total count for primary token without having to accumulate all records in memory.
   3. 
   4. Example of the sequence of key-value pairs presented to the reducer in the pairs algorithm for computing relative frequencies; this illustrates the application of the order inversion design pattern
   5. It is so named because through proper coordination, we can access the result of a computation in the reducer (for example, an aggregate statistic) before processing the data needed for that computation
   6. The key insight is to convert sequencing of computations into a sorting problem.

Summary

1. Ram vs Disk vs bandwidth
2. Back of the envelope calculations
3. Priority queues and merge sorts
4. Internals of Hadoop Shuffle operation
5. Patterns
6. Local aggregation:
   1. Combiners and in-mapper combining
   2. Algorithmic correctness with local aggregation
   3. Number of tasks and in-memory mapper
7. Pairs and stripes
8. Computing relative frequencies
9. Order inversion pattern
10. Secondary Sorts
11. Sync and Communication are key:
    1. Constructing complex keys and values that bring together data necessary for a computation. This is used in all of the above design patterns.
    2. Controlling the partition of the intermediate key space. This is used in order inversion and value-to-key conversion.
    3. Controlling the sort order of intermediate keys. This is used in order inversion and secondary sorting.
12. Patterns apply to Hadoop MR and MRJob and Spark