1. Challenges for big data computing:
    1. Scalability
        a. Volume, huge amount of data.
        b. Velocity, the speed of accumulation of data.
    2. Reliability
        a. Veracity, inconsistencies and uncertainty in data
    3. Productivity
    4. Variety, different formats of data from various sources
    5. Value, extract useful data
2. Three perspectives in Big Data Computing:
    a. Architecture:
    b. algorithms:
        i. parallelism
        ii. Scalability
        iii. distributed data
    c. Programming: Map reduce
3. Map reduce is to extract something you care about and group by key, and then aggregate, summarize, filter or transform.
    a. Environment
        i. Partitioning the input data
        ii. Scheduling the program's execution across a set of machines
        iii. Performing the group by key step
        iv. Handling machine failures
        v. Managing required inter-machine communication
    b. Resilient distributed dataset (RDD)

This a fault-tolerant collection of elements that can be operated on in parallel.

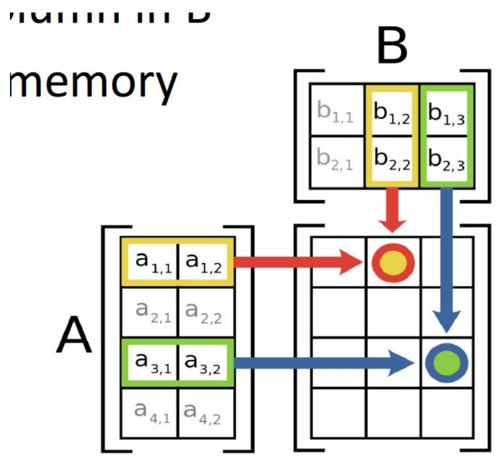
    i. Two ways to create it
        o Parallelizing an existing collection in your driver program.
        o Referencing a dataset in an external storage system.

- Joins: RDD + RDD -> RDD

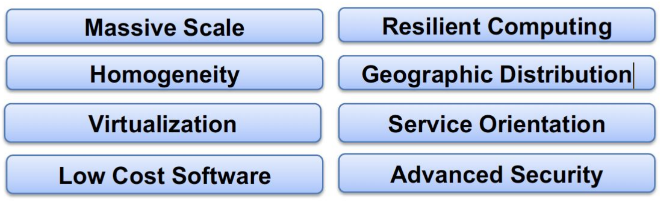
    ii. A Map process turns:
        o Each input tuple R(a, b) into key-value pair (b,(a, R))
        o Each input tuple S(b, c) into (b,(c, S))
    iii. Map processes send each key-value pair with key b to Reduce process h(b).

iv. Each Reduce process matches all the pairs (b, (a, R)) with all (b, (c, S)) and outputs (a, b, c).

4. Algorithm in Spark:
   1. Selection, Union, Intersection, Difference, Grouping, Joins,
   2. https://nbviewer.jupyter.org/github/jkthompson/pyspark-pictures/blob/master/pyspark-pictures.ipynb
   3. Matrix-Vector is to compute the product of matrix M with vector v
   4. $$(Mv)_i = \sum_j m_{ij} v_j$$
      a. Storage: The matrix and vectors are stored in a sparse form:
         i. Triplets of the form $(i,\ j,\ m_{ij})$ for the non-zero entries of the matrix.
         ii. Pairs of the form $(i,\ v_i)$ for the elements of the vector.
      b. Case 1: the vector fits in memory
         i. In this case the vector that we want multiply is loaded in memory at each mapper.
         ii. The mapper reads a chunk of the matrix M, and for each entry
         iii. $(i,\ j,\ m_{ij})$ it outputs the key-value pair $(i,\ m_{ij} v_j)$.
         iv. The reducer takes the sum of all values that are associated with row.
      c. Case 2: the vector does not fit in memory
         i. We split the matrix and the vector into stripes and then perform the computation for each stripe of the matrix, where the vector can fit into memory.
      - Matrix-Matrix is to repeat matrix-vector multiplication across each column in B.

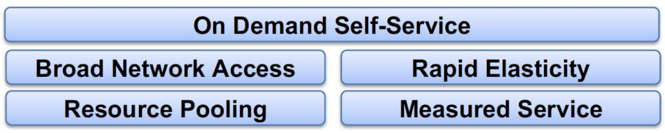


5. Cloud Computing is a general term used to describe a new class of network based computing that takes place over the Internet.
   a. Characteristics:
      i. Common Characteristics:



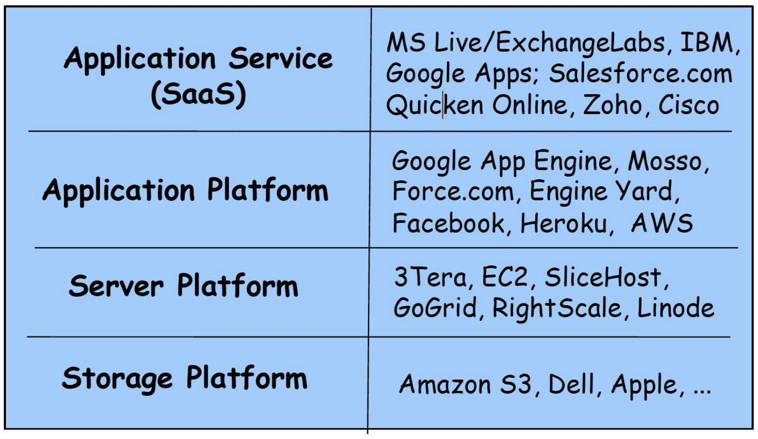
Massive Scale	Resilient Computing
Homogeneity	Geographic Distribution
Virtualization	Service Orientation
Low Cost Software	Advanced Security

ii. Essential Characteristics



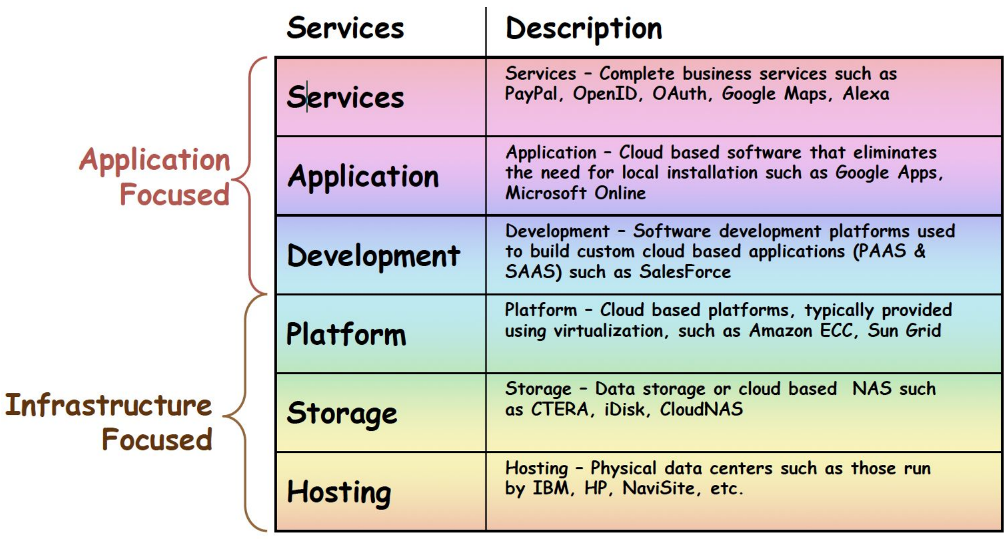
On Demand Self-Service	
Broad Network Access	Rapid Elasticity
Resource Pooling	Measured Service

b. Different cloud computing layers



Application Service (SaaS)	MS Live/ExchangeLabs, IBM, Google Apps; Salesforce.com Quicken Online, Zoho, Cisco
Application Platform	Google App Engine, Mosso, Force.com, Engine Yard, Facebook, Heroku, AWS
Server Platform	3Tera, EC2, SliceHost, GoGrid, RightScale, Linode
Storage Platform	Amazon S3, Dell, Apple, ...

c. cloud computing service layers


Application Focused
Infrastructure Focused

Services	Description
Services	Services - Complete business services such as PayPal, OpenID, OAuth, Google Maps, Alexa
Application	Application - Cloud based software that eliminates the need for local installation such as Google Apps, Microsoft Online
Development	Development - Software development platforms used to build custom cloud based applications (PAAS & SAAS) such as SalesForce
Platform	Platform - Cloud based platforms, typically provided using virtualization, such as Amazon ECC, Sun Grid
Storage	Storage - Data storage or cloud based NAS such as CTERA, iDisk, CloudNAS
Hosting	Hosting - Physical data centers such as those run by IBM, HP, NaviSite, etc.

d. Basic cloud characteristics:

i. No need to know

Underlying the details of infrastructure, application interface with the infrastructure via the APIs.

ii. flexibility and elasticity are to allow these systems to scale up and down at will.

iii. pay as much as used and needed

iv. always on! anywhere and any place

e. Cloud storage

f. Advantages of cloud computing

   i. Easier group collaboration

   ii. Device independence

g. Disadvantages of cloud computing

   i. Requires a constant internet connection

   ii. Does not work will with low speed connections

   iii. Features might be limited

   iv. Can be slow by lots of reasons.

   v. Stored data might not be secure and it may be lost.

6. Frequency Item

   a. The market-basket model

      - ♥ A large set of items
      - ♥ A large set of baskets
      - ♥ Each basket is a small subset of items

   b. Let

      $\{\backslash displaystyle\, X, Y\}$ $INCLUDEPICTURE$ "$https : //wikimedia.org/api/rest\_v1/media/math/render/$

      ,

      $\{\backslash displaystyle\, X, Y\}$ $INCLUDEPICTURE$ "$https : //wikimedia.org/api/rest\_v1/media/math/render/$

      be itemsets, $X \Rightarrow Y$ $XxXxx$ an association rule and

      $\{\backslash displaystyle\, X, Y\}$ $INCLUDEPICTURE$ "$https : //wikimedia.org/api/rest\_v1/media/math/render/$

      a set of transactions of a given database.

      i. Frequent Itemset

         o **Support** for itemset *I*: number of baskets containing all items in *I*

            ▪ an indication of how frequently the itemset appears in the dataset.

            ▪ $supp\,(X) = \frac{|\{t \in T; X \subseteq t\}|}{|T|}$

         o **Confidence** is an indication of how often the rule has been found to be true.

            ▪ $conf\,(X \Rightarrow Y) = \frac{supp(X \cup Y)}{supp(X)}$

               ♥ $supp\,(X \cup Y)$ means the support of the union of the items in X and Y.

               ♥ $supp\,(X \cup Y)$ can be rewrite as probability $P\,(E_X \cap E_Y)$, where E_X and E_Y are the events that a transaction contains itemset X and

$\{\backslash displaystyle\, X, Y\}$ *INCLUDEPICTURE "https : //wikimedia.org/*

, respectively.

- Given a support threshold s, then sets of items that appear in at least s baskets are called frequent itemsets.
  - **Frequent itemset** is an itemset whose support is greater than some user-specified minimum support (denoted L_k , where k is the size of the itemset)
- **Interest** of an association rule $I \rightarrow j$: difference between its confidence and the fraction of baskets that contain j
  - $Interest(I \rightarrow j) = conf(I \rightarrow j) - Pr(j)$
    - ♥ $Pr(j)$ is how many time of j appears in the number of baskets
  - Interesting rules are those with high positive or negative interest values (usually above 0.5)

ii. Compacting the output

- To reduce the number of rules we can post-process them and only output:
  - Maximal frequent itemsets:

An itemset is maximal frequent if none of its immediate supersets is frequent and itemset must be a frequent itemset.

Note: immediate supersets: if A is a subset of B and next to B, B will be immediate superset.

- Closed itemsets:

An itemset is closed if none of its immediate supersets has the same support as the itemset.

c. Find frequent itemsets

i. Main-memory is the critical resource.

d. Find frequent Pairs

i. Approach:

- Generate all the itemsets
- Count (keep track) of those itemsets that in the end turn out to be frequent

ii. Naïve algorithm

- From each basket of n items, generate its $\frac{n(n-1)}{2}$ pairs by two nested loops.
- Fails if $(\#items)^2$ exceeds main memory

iii. Counting Pairs in memory

- Approach

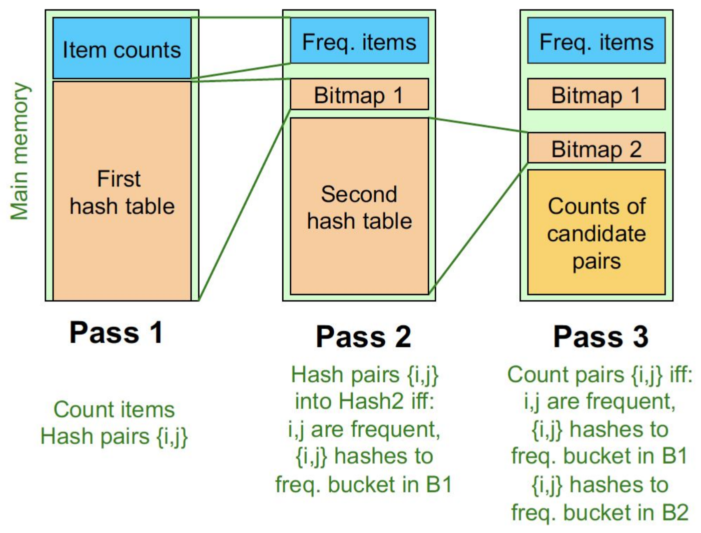
        - Count all pairs using a matrix which only requires 4 bytes per pair.
        - Keep a table of triples [i, j, c] = the count of the pair of items {i, j} is c. This uses 12 bytes per pair (but only for pairs with count>0)

iv. A-Priori Algorithm
    - Pass 1: read baskets and count in main memory the occurrences of each individual item. (items that appear >= s times are the frequent items)
    - Pass 2: read baskets again and count in main memory only those pairs where both elements are frequent (from pass 1)

v. PCY algorithm
    - Pass 1: in addition to item counts, maintain a hash table with as many buckets as fit in memory. Keep a count for each bucket into which pairs of items are hashed.
        - Note:
            - Pairs of items need to be generated from the input file; they are not present in the file.
            - We are not just interested in the presence of a pair, but we need to see whether it is present at least s (support) times.
            - Bucket contains a frequent pair then the bucket is surely frequent.
            - However, even without any frequent pair, a bucket can still be frequent.
            - But, for a bucket with total count less than s, none of its pairs can be frequent.
    - Pass 2: only counts pairs that hash to frequent buckets
        - Count all pairs {i, j} that meet the condition for being a candidate pair
            - Both i and j are frequent items
            - The pair {i, j} hashes to a bucket whose bit in the bit vector is 1.
        - Both conditions are necessary for the pair to have a chance of being frequent.

vi. Multistage algorithm
    - Limit number of candidates to be counted
    - Key idea: after Pass 1 of PCY, rehash only those pairs that qualify for Pass 2 of PCY.
        - i and j are frequent, and {i, j} hashes to a frequent bucket from Pass 1
    - on middle pass, fewer pairs contribute to buckets, so fewer false positives.
    - requires 3 passes over the data
    - Pass 3: count only those pairs {i, j} that satisfy these candidate pair conditions:

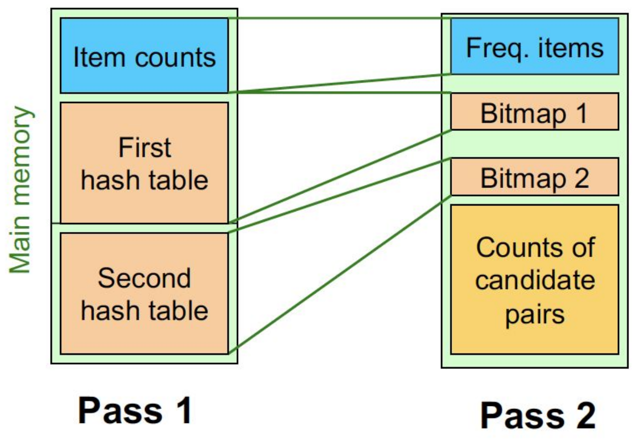
        - Both i and j are frequent items
        - Using the first hash function, the pair hashes to a bucket whose bit in the first bit-vector is 1.
        - Using the second has function, the pair hashes to a bucket whose bit in the second bit-vector is 1.



Main memory
Item counts
First hash table
Freq. items
Bitmap 1
Second hash table
Freq. items
Bitmap 1
Bitmap 2
Counts of candidate pairs
Pass 1
Pass 2
Pass 3
Count items Hash pairs {i,j}
Hash pairs {i,j} into Hash2 iff: i,j are frequent, {i,j} hashes to freq. bucket in B1
Count pairs {i,j} iff: i,j are frequent, {i,j} hashes to freq. bucket in B1 {i,j} hashes to freq. bucket in B2

    - Note:
        - The two hash functions have to be independent
        - We need to check both hashes on the third pass

vii. Multihash
    - Key idea: use several independent has tables on the first pass
    - Risk: halving the number of buckets doubles the average count. (make sure most buckets less than count s)
    - If so, we have multihash.



viii. Others algorithm with less than 2 Passes for finding frequent itemsets:
    - It is to use 2 or fewer passes for all sizes but may miss some frequent itemsets.
        - Random sampling

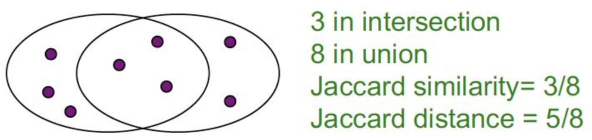
        - SON
        - Toivonen

7. High dimensional data
    a. Distance Measures
        i. Jaccard distance/similarity
            - The Jaccard similarity of two sets is the size of their intersection divided by the size of their union:

$$sim(C_1, C_2) = |C_1 \cap C_2| / |C_1 \cup C_2|$$

            - Jaccard distance:

$$d(C_1, C_2) = 1 - |C_1 \cap C_2| / |C_1 \cup C_2|$$



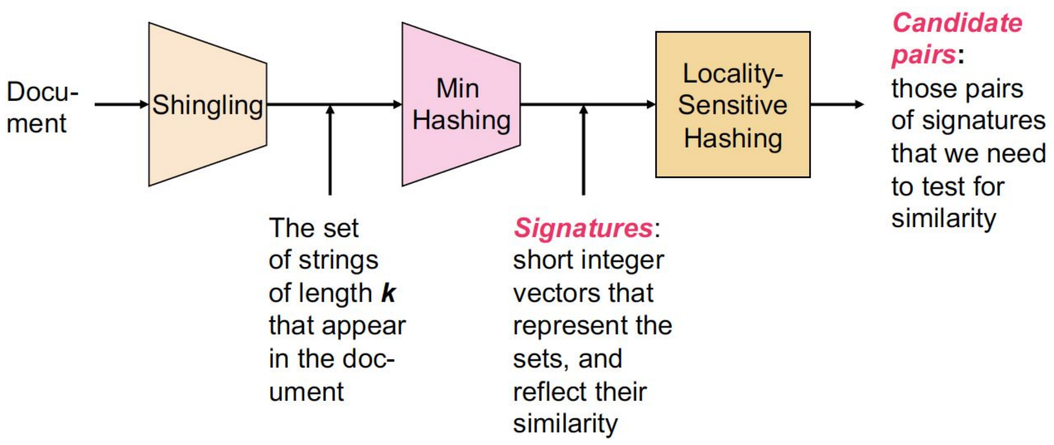
        ii. Axioms of Distance Measures
            - D is a distance measure if it a function from pairs of points to real numbers such that:
                - $d(x,y)>0$.
                - $d(x,y)=0$ iff $x=y$
                - $d(x,y)=d(y,x)$
                - $d(x,y)<d(x,z)+d(z,y)$ (aka triangleinequality)
            - Cosine distance
                - Two points' vectors make an angle, whose cosine is the normalized dot-product of the vectors

$$\cos cos\ (\theta) = \frac{p_1 p_2}{|p_1|*|P_2|}$$

        iii. Other distance measures
            - L_1 norm = sum of the differences in each dimension. (aka Manhattan distance)
            - L_2 norm = square root of the sum of the squares of the differences between x and y in each dimension
            - L_∞ norm = the maximum of the differences between x and y in any dimension
            - Hamming distance = number of positions in which bit vectors differ
        iv. Distance measure for different sets
            - Sets as vectors: measure similarity by cosine distance

- Sets as sets: measure similarity by the Jaccard distance
- Sets as points: measure similarity by Euclidean distance

b. Documents as high-dim data


Docu-
ment
Shingling
Min
Hashing
Locality-
Sensitive
Hashing
Candidate pairs: those pairs of signatures that we need to test for similarity
The set of strings of length k that appear in the document
Signatures: short integer vectors that represent the sets, and reflect their similarity

i. Step 1: shingling: convert documents to sets

We used hashing to assign each shingle an ID

- Simple approaches:
    - Document = set of words appearing in document
    - Document = set of ‘important’ words
    - Need to account for ordering of words
- Shingles: A k-shingle (or k-gram) for a document is a sequence of k tokens that appears in the doc.
    - Tokens can be characters, words or something else, depending on the application
    - Assume tokens = character for examples

- **Example:** **k=2**; document **D_1** = abcab
  Set of 2-shingles: **$S(D_1)$** = {ab, bc, ca}
    - **Option:** Shingles as a bag (multiset), count ab twice: **$S'(D_1)$** = {ab, bc, ca, ab}

    - To compress long shingles, we can hash them to 4 bytes
        - Represent a document by the set of hash values of its k-shingles
        - Idea: two documents could (rarely) appear to have shingles in common, when in fact only the hash-value were shared.

- **Example:** **k=2**; document **D_1**= abcab
  Set of 2-shingles: **$S(D_1)$** = {ab, bc, ca}
  Hash the singles: **$h(D_1)$** = {1, 5, 7}

- Similarity Metric for Shingles
  - Document D_1 is a set of its k-shingles $C_1 = S(D_1)$
  - Equivalently, each document is 0/1 vector in the space of k-shingles
    - Each unique shingle is a dimension
    - Vectors are very sparse
  - A natural similarity measure is the Jaccard similarity
- You must pick k large enough, or most documents will have most shingles.

ii. Step 2: Minhashing: Concert large set to short signatures, while preserving similarity

We used similarity preserving hashing to generate signatures with property

$$Pr[h_\pi(C_1) = h_\pi(C_2)] = sim(C_1, C_2)$$

We used hashing to get around generating random permutations

- Represent sets as Boolean vectors in a matrix
- Find similar columns while computing small signatures

Note: Similarity of columns == similarity of signatures

- Naïve approach will take too much time
  - Signatures of columns: small summaries of columns
  - Examine pairs of signatures to find similar columns
  - Check that columns with similar signatures are really similar
- Hashing columns (signatures)
  - Key idea: "hash" each column C to a small signature $h(C)$, such that:
    - $h(C)$ is small enough that the signature fits in RAM
    - $sim(C_1, C_2)$ is the same as the similarity of signatures $h(C_1)$ and $h(C_2)$
  - Goal: find a hash function $h(\cdot)$ such that
    - If $sim(C_1, C_2)$ is high, then with high prob. $h(C_1) = h(C_2)$
    - It $sim(C_1, C_2)$ is low, then with high prob. $h(C_1) \neq h(C_2)$

- Min-hashing: a suitable hash function for the Jaccard similarity
  - Define a hash function $h_\pi(C) =$ the index of the first (in the permuted order π) row in which column C has value 1:

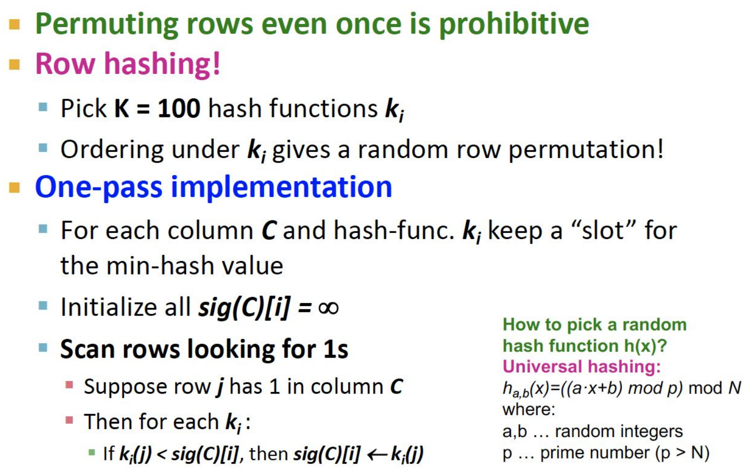
    $$h_\pi(C) = Min_\pi\, \pi(C)$$
  - Use several independent hash functions to create a signature of a column
  - Property
    - Choose a random permutation π

- ◆ $\Pr Pr\left[h_\pi(C_1) = h_\pi(C_2)\right] = sim(C_1, C_2)$ by

  $$\Pr[\min(\pi(C_1))=\min(\pi(C_2))]=|C_1 \cap C_2|/|C_1 \cup C_2| = sim(C_1, C_2)$$

- ▪ Now generalize to multiple hash functions
- ▪ The similarity of two signatures is the fraction of the hash functions in which they agree.
  - ◆ Note: similarity of columns is the same as the expected similarity of their signatures because of min-hashing property
  - ◆ Compress long bit vector into short signatures
    - Pick k = 100 random permutations of the rows
    - Think of sig(C) as a column vector
    - Sig(C)[i] = according to the i-th permutation, the index of the first row that has a 1 in column C

    $sig(C)[i] = Min(\pi_i(C))$
  - ◆ Implementation Trick



- **Permuting rows even once is prohibitive**
- **Row hashing!**
  - Pick **K = 100** hash functions $\boldsymbol{k_i}$
  - Ordering under $\boldsymbol{k_i}$ gives a random row permutation!
- **One-pass implementation**
  - For each column ***C*** and hash-func. $\boldsymbol{k_i}$ keep a "slot" for the min-hash value
  - Initialize all ***sig(C)[i] = ∞***
  - **Scan rows looking for 1s**
    - Suppose row ***j*** has 1 in column ***C***
    - Then for each $\boldsymbol{k_i}$:
      - If $k_i(j) < sig(C)[i]$, then $sig(C)[i] \leftarrow k_i(j)$

How to pick a random hash function h(x)?
Universal hashing:
$h_{a,b}(x)=((a \cdot x+b) \bmod p) \bmod N$
where:
a,b … random integers
p … prime number (p > N)

iii. Step 3: Locality-Sensitive Hashing: Focus on pairs of signatures likely to be similar documents

We used hashing to find candidate pairs of similarity $\geq s$

- o Goal: find documents with Jaccard similarity at least s
- o General idea of LSH:
  - ▪ Use a function $f(x, y)$ that tells whether x and y is a **candidate pair**: a pair of elements whose similarity must be evaluated.
- o For min-hash matrices:
  - ▪ Hash columns of signature matrix M to many buckets.
  - ▪ Each pair of documents that hashes into the same bucket is a **candidate pair.**
    - ◆ Candidates:

            - Pick a similarity threshold s $(0 < s < 1)$
            - Columns x and y of M are a candidate pair if their signatures agree on at least fraction s of their rows:

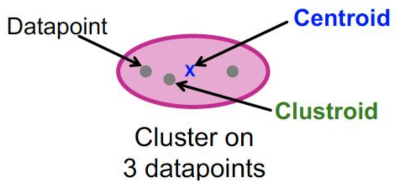
              $M(i, x) = M(i, y)$ for at least frac. s values of i

              $\Rightarrow$ We expect documents x and y to have the same (Jaccard) similarity as their signatures
        - LSH:
            - Big idea: Hash columns of signature matrix M several times.
            - Arrange that (only) **similar columns** are likely to **hash to the same bucket**, with high probability.
            - Candidate pairs are those that hash to the same bucket.
    - Partition M into Bands
        - Divide matrix M into b bands of r rows.
        - For each band, hash its portion of each column to a hash table with k buckets.
            - Make k as large as possible
        - **Candidate** column pairs are those that hash to the same bucket for ≥ 1 band.
        - Tune b and r to catch most similar pairs, but few non-similar pairs.
- Simplifying Assumption
    - There are **enough buckets** that columns are unlikely to hash to the same bucket unless they are **identical** in a particular band.
    - Hereafter, we assume that "**same bucket**" means "**identical in that band**".
    - Assumption needed only to simplify analysis, not for correctness of algorithm.
- LSH summary:
    - Tune M, b, r to get almost all pairs with similar signatures, but eliminate most pairs that do not have similar signatures
    - Check in main memory that **candidate pairs** really do have similar signatures
    - Optional: In another pass through data, check that the remaining candidate pairs really represent similar documents

iv. Clustering is given a **set of points**, with a notion of **distance** between points, **group the points** into some number of **clusters**.

- Methods
  - Hierarchical:
    - agglomerative (bottom up)
    - Divisive (top down)
  - Point assignment
    - Maintain a set of clusters
    - Points belong to nearest cluster
- Hierarchical clustering
  - Key operation: repeatedly combine two nearest clusters
  - Problems:
    - How to represent a cluster of many points?
      - Euclidean case:
        - ⇒ each cluster has a centroid = average of its point
      - Non-Euclidean case:
        - ⇒ clustroid = (data) point "closest" to other points
        - ⇒ For distance metric d clustroid c of cluster C is

$$\sum_{x \in C} d(x, c)^2$$



        - ⇒ Meanings of "closest"
          - ❖ Smallest maximum distance to other points
          - ❖ Smallest average distance to other points
          - ❖ Smallest sum of squares of distances to other points
    - How to determine "nearness" of clusters?
      - Euclidean case:
        - ⇒ Measure cluster distances by distances of centroids.
      - Non-Euclidean case:
        - ⇒ Approach 1:

          Treat clustroid as if it were centroid, when computing inter-cluster distances.
        - ⇒ Approach 2:

          Inter-cluster distance = minimum of the distances between any two points, one from each cluster
        - ⇒ Approach 3:

Pick a notion of “cohesion” of clusters, e.g., maximum distance from the clustroid.

Merge clusters whose union is most cohesive.

❖ Approach 3.1:

Use the diameter of the merged cluster = maximum distance between points in the cluster

❖ Approach 3.2:

Use the average distance between points in the cluster

❖ Approach 3.3:

Use a density-based approach

Take the diameter or avg. distance, e.g., and divide by the number of points in the cluster

- K-means clustering
  - Algorithms
    - ♦ Assumes Euclidean space/distance
    - ♦ Start by picking k, the number of clusters
      - How to select k?
        - ⇒ Try different k, looking at the change in the average distance to centroid as k increases.
        - ⇒ Average falls rapidly until right k, then changes little.
    - ♦ Initialize clusters by picking one point per cluster
      - Populating clusters

        1)For each point, place it in the cluster whose current centroid it is nearest.

        2) After all points are assigned, update the locations of centroids of the k clusters.

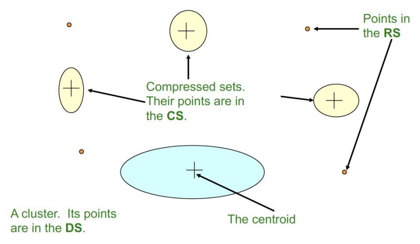
        3) Reassign all points to their closest centroid.

        ❖ Sometimes moves points between clusters.

        Repeat 2 and 3 until convergence.

        ❖ Convergence: Points don’t move between clusters and centroids stabilize.
- The BFR algorithm (for large data)
  - BFR [Bradley-Fayyad-Reina] is a variant of k-means designed to handle very large (disk-resident) data sets.

- Assumes that clusters are normally distributed around a centroid in a Euclidean space.
  - Standard deviations in different dimensions may vary
    - Clusters are axis-aligned ellipses
- Efficient way to summarize clusters (want memory required $O(clusters)$ and not $O(data)$)
  - Points are read from disk one main-memory-full at a time
  - Most points from previous memory loads are summarized by simple statistics
    - Processing the "Memory-Load" of points
      - ⇒ Find those points that are "sufficiently close" to a cluster centroid and add those points to that cluster and the DS
      - ⇒ Use any main-memory clustering algorithm to cluster the remaining points and the old RS. (Clusters go to the CS; outlying points to the RS)
      - ⇒ DS set: Adjust statistics of the clusters to account for the new points --- Add Ns, SUMs, SUMSQs
      - ⇒ Consider merging compressed sets in the CS
      - ⇒ If this is the last round, merge all compressed sets in the CS and all RS points into their nearest cluster
  - To begin, from the initial load we select the initial k centroids by some sensible approach:
    - Take k random points
    - Take a small random sample and cluster optimally
    - Take a sample; pick a random point, and then k–1 more points, each as far from the previously selected points as possible
  - 3 sets of pints which we keep track of:
    - Discard set (DS):
      - ⇒ Points close enough to a centroid to be summarized
    - Compression set (CS):
      - ⇒ Groups of points that are close together but not close to any existing centroid
      - ⇒ These points are summarized, but not assigned to a cluster


Points in the RS
Compressed sets. Their points are in the CS.
A cluster. Its points are in the DS.
The centroid

            - Retained set (RS):
                - ⇒ Isolated points waiting to be assigned to a compression set
    - How to decide whether to put a new point into a cluster
        - The Mahalanobis distance is less than a threshold
        - high likelihood of the point belonging to currently nearest centroid
            - Normalized Euclidean distance from centroid
            - For point $(x_1, \dots, x_d)$ and centroid $(c_1, \dots, c_d)$

$$d(x,c) = \sqrt{\sum_{i=1}^{d} \left(\frac{x_i - c_i}{\sigma_i}\right)^2}$$

            - If clusters are normally distributed in d dimensions, then after transformation, one standard deviation $= \sqrt{d}$
            - Accept a point for a cluster if its M.D. is < some threshold, e.g. 2 standard deviations
    - Should 2 CS subclusters be combined?
        - Compute the variance of the combined subclusters
            - N, SUM, and SUMSQ allow us to make that calculation quickly
        - Combine if the combined variance is below some threshold
        - Many alternatives: Treat dimensions differently, consider density.

- Problem with BFR/k-means:
    - Assumes clusters are normally distributed in each dimension
    - And axes are fixed – ellipses at an angle are not OK
- The CURE (Clustering Using REpresentatives) algorithm (to clusters of arbitrary shapes)
    - Methods:
        - Assumes a Euclidean distance
        - Allows clusters to assume any shape
        - Uses a collection of representative points to represent clusters
    - 2 Pass algorithm
        - Pass 1:

        0) Pick a random sample of points that fit in main memory

        1) Initial clusters:

            ⇒ Cluster these points hierarchically – group nearest points/clusters

2) Pick representative points:

⇒ For each cluster, pick a sample of points, as dispersed as possible

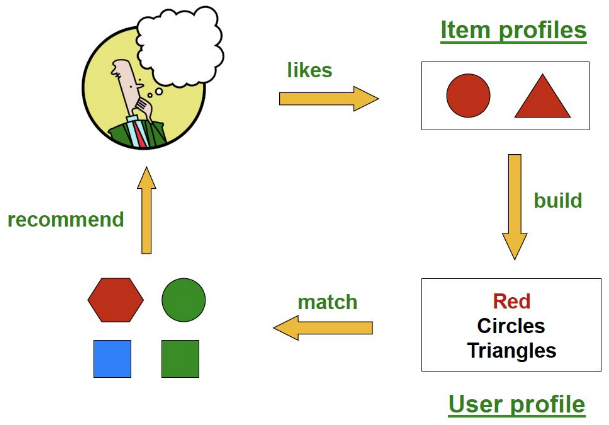
⇒ From the sample, pick representatives by moving them (say) 20% toward the centroid of the cluster

♦ Pass 2:

- Now, rescan the whole dataset and visit each point p in the data set
- Place it in the "closest cluster"
  - ⇒ Normal definition of "closest":
    - ❖ Find the closest representative to p and assign it to representative's cluster

8. Recommender systems
   a. Formal Model
      i. X = set of customers
      ii. S = set of items
      iii. Utility function $u : X \times S \rightarrow R$
         - o $R =$ set of ratings
         - o R is a totally ordered set. E.g.,0-5 stars, real number in [0, 1]
      iv. Utility Matrix
   b. Key Problems
      i. Gathering known ratings for matrix (How to collect the data in the utility matrix)
         - o Explicit
           - ▪ Ask people to rate items
           - ▪ Doesn't work well in practice – people can't be bothered
         - o Implicit
           - ▪ Learn ratings from user actions
           - ▪ Low ratings?
      ii. Extrapolate unknown ratings from the known ones (mainly interested in high unknown ratings)
         - o Key problems: utility matrix U is sparse
           - ▪ Most people have not rated most items
         - o Approaches

- Content-based: recommend items to customer x similar to previous items rated highly by x



- For each item, create an item profile which is a set (vector) of features.
- Use TF-IDF to pick important features
  - f_{ij} = frequency of term (feature) i in doc (item) j

    $TF_{ij} = \frac{f_{ij}}{max_k f_{kj}}$
  - n_i = number of docs that mention term i

    N = total number of docs

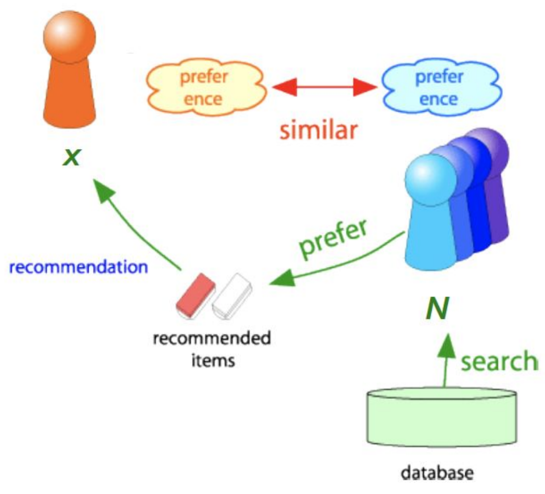
    $IDF_i = \log log \frac{N}{n_i}$
  - TF-IDF score: $w_{ij} = TF_{ij} \times IDF_i$
  - Doc profile = set of words with highest TF-IDF scores, together with their scores
- User profile possibilities:
  - Weighted average of rated item profiles
  - Variations: weight by difference from average rating for item
- Prediction heuristic: Given user profile x and item profile I, estimate

  $u(x, i) = \cos cos \ (x, i) \ = \frac{x \cdot i}{||x|| \cdot ||i||}$
- Pros:
  - No need for data on other users
  - Able to recommend to users with unique tastes
  - Able to recommend new & unpopular items
  - Able to provide explanations
- Cons:
  - Finding the appropriate features is hard
  - Recommendations for new users

- Overspecialization (Unable to exploit quality judgments of other users)

- Collaborative
  - User-user collaborative filtering

    Consider user x. Find set N of other users whose ratings are similar to x's ratings. Estimate x's ratings based on ratings of users in N.



- Finding "similar" users
  - ⇒ Let r_x be the vector of user x's ratings
  - ⇒ Jaccard similarity measure

    (ignores the value of the rating)

r_x, r_y as sets:
$r_x = \{1, 4, 5\}$
$r_y = \{1, 3, 4\}$

  - ⇒ Cosine similarity measure

    (treats missing rating as negative)

$$sim(x, y) = \cos cos\ (r_x, r_y) = \frac{r_x \bullet r_y}{\|r_x\| \cdot \|r_y\|}$$

r_x, r_y as points:
$r_x = \{1, 0, 0, 1, 3\}$
$r_y = \{1, 0, 2, 2, 0\}$

  - ⇒ Pearson correlation coefficient

    s_{xy} = items rated by both user x and y

$$\boldsymbol{sim(x, y)} = \frac{\sum_{s \in S_{xy}} (\boldsymbol{r_{xs}} - \overline{\boldsymbol{r_x}})(\boldsymbol{r_{ys}} - \overline{\boldsymbol{r_y}})}{\sqrt{\sum_{s \in S_{xy}} (\boldsymbol{r_{xs}} - \overline{\boldsymbol{r_x}})^2} \sqrt{\sum_{s \in S_{xy}} (\boldsymbol{r_{ys}} - \overline{\boldsymbol{r_y}})^2}}$$

- Rating predictions

  From similarity metric to recommendations:
  - ⇒ Let r_x be the vector of user x's ratings
  - ⇒ Let N be the set of k users most similar to x who have rated item i

- Prediction for item s of user x:

$$r_{xi} = \frac{1}{k} \sum_{y \in N} r_{yi}$$

$$r_{xi} = \frac{\sum_{y \in N} S_{xy} \cdot r_{yi}}{\sum_{y \in N} S_{xy}} \quad \text{with} \quad S_{xy} = sim(x, y)$$

- ♦ Item-item collaborative filtering
  - For item i, find other similar items
  - Estimate rating for item i based on ratings for similar items
  - It can use same similarity metrics and prediction functions as in user-user model

$$\boxed{r_{xi} = \frac{\sum_{j \in N(i;x)} s_{ij} \cdot r_{xj}}{\sum_{j \in N(i;x)} s_{ij}}}$$

s_{ij}… similarity of items i and j
r_{xj}…rating of user u on item j
$N(i;x)$… set items rated by x similar to i

  - Common Practice
    - ⇒ Define similarity s_{ij} of items i and j
    - ⇒ Select k nearest neighbors $N(i;x)$

(items most similar to i, that were rated by x)

    - ⇒ Estimate rating r_{xi} as the weighted average:

$$r_{xi} = b_{xi} + \frac{\sum_{j \in N(i;x)} s_{ij} \cdot (r_{xj} - b_{xj})}{\sum_{j \in N(i;x)} s_{ij}}$$

baseline estimate for r_{xi}

$$b_{xi} = \mu + b_x + b_i$$

- μ = overall mean movie rating
- b_x = rating deviation of user x = (*avg. rating of user* x) $- \mu$
- b_i = rating deviation of movie i

- ♦ Pros: No feature selection needed
- ♦ Cons:
  - need enough users in the system to find a match
  - Sparsity
  - Can't recommend an item that has not been previously rated
  - Popularity bias.
- ♦ Hybrid methods
  - Add content-based methods to collaborative filtering

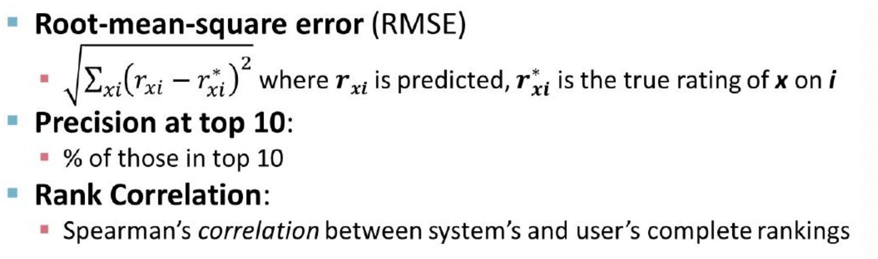
▪ Latent factor based

iii. Evaluating extrapolation methods

(how to measure success / performance of recommendation methods)

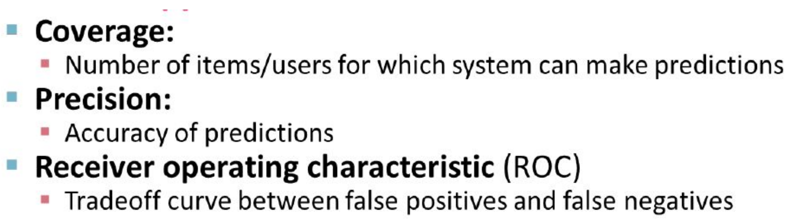
o Evaluating predictions

▪ Compare predictions with known ratings



- **Root-mean-square error** (RMSE)
  - $\sqrt{\sum_{xi}(r_{xi} - r^*_{xi})^2}$ where $\boldsymbol{r_{xi}}$ is predicted, $\boldsymbol{r^*_{xi}}$ is the true rating of ***x*** on ***i***
- **Precision at top 10**:
  - % of those in top 10
- **Rank Correlation**:
  - Spearman's *correlation* between system's and user's complete rankings

- 0/1 model



- **Coverage:**
  - Number of items/users for which system can make predictions
- **Precision:**
  - Accuracy of predictions
- **Receiver operating characteristic** (ROC)
  - Tradeoff curve between false positives and false negatives

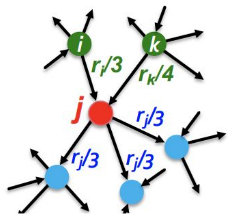
- Problems with error measures
  - Narrow focus on accuracy sometimes misses the point
    - Prediction diversity
    - Prediction context
    - Order of predictions
- In practice, we care only to predict high ratings
  - RMSE might penalize a method that does well for high ratings and badly for others
- In collaborative filtering: complexity
  - Expensive step is finding k most similar customers: $O(|X|)$
  - Too expensive to do at runtime
  - Naïve pre-computation takes time $O(k \cdot |X|)$
    - Near-neighbor search in high dimensions (LSH)
    - Clustering
    - Dimensionality reduction

9. Analysis of Large graphs:
   a. Link Analysis Algorithms: to compute importance of nodes in a graph
      i. Page Rank
      ii. Topic-specific (personalized) Page Rank
      iii. Web spam detection algorithms
   b. PageRank:
      i. The "Flow" formulation
         - Links as votes:
           - Page is more important if it has more links.

        - Links from important pages count more
    - Each link's vote is proportional to importance of its source page.

        - If page *j* with importance r_j has *n* out-links, each link gets r_j / n votes
        - Page *j*'s own importance is the sum of the votes on its in-links

$$r_j = r_i/3 + r_k/4$$

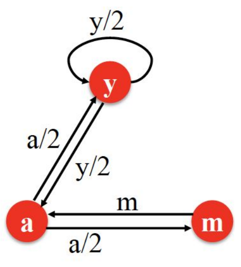


- The "Flow" Model
    - A "vote" from an important page is worth more
    - A page is important if it is pointed to by other important pages
    - Define a "rank" r_j for page j

$$r_j = \sum_{i \to j} \frac{r_i}{d_i}$$

$d_i \ldots$ out-degree of node i

    - Example:



"Flow" equations:

$$r_y = r_y/2 + r_a/2$$
$$r_a = r_y/2 + r_m$$
$$r_m = r_a/2$$

    - Additional constraint forces uniqueness:
        - $r_y + r_a + r_m = 1$
        - we can have unique solutions, but it only works in small example
    - Matrix Formulation
        - Stochastic adjacency matrix M
            - Let page i has d_i out-links
            - If $i \to j$, them $M_{ji} = \frac{1}{d_i}$ else $M_{ji} = 0$

$\Rightarrow$ M is a column stochastic matrix and columns sum to 1.

        - Rank vector r: vector with an entry per page
            - r_i is the importance score of page i
            - $\sum_i r_i = 1$
        - The flow equations can be written

$$r = M \cdot r$$

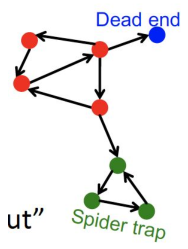
- Random Walk interpretation
    - r is a stationary distribution for the random walk.

- For graphs that satisfy certain conditions, the stationary distribution is unique and eventually will be reached no matter what the initial probability distribution at time t = 0.

ii. The google formulation

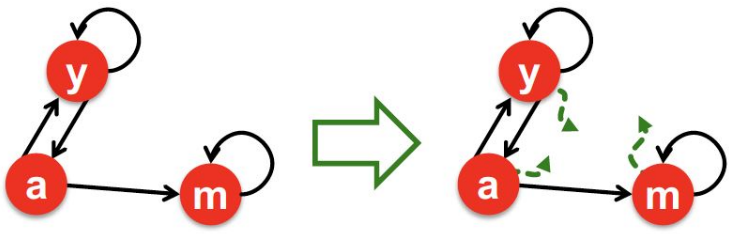
o Problems

- Some pages are dead ends (have no out-links)
  - The matrix is not column stochastic, so our initial assumptions are not met.
- Spider traps: all out links are within the group
  - With traps PageRank scores are not what we want.

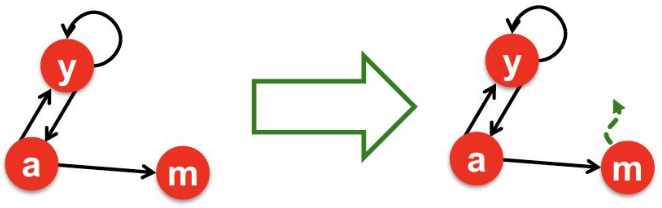


o Solution: Teleports

- The google solution for spider traps: at each time step, the random surfer has two options
  - With prob. β, follow a link at random
  - With prob. $1-\beta$, jump to some random page
  - Common values for β are in the range 0.8 to 0.9.
  - Surfer will teleport out of spider trap within a few time steps



- For dead-ends, teleport can also work: follow random teleport links with probability 1.0 from dead-ends



	y	a	m
y	½	½	0
a	½	0	0
m	0	½	0

	y	a	m
y	½	½	⅓
a	½	0	⅓
m	0	½	⅓

- PageRank equation

$$r_j = \sum_{i \to j} \beta \frac{r_i}{d_i} + (1-\beta)\frac{1}{N}$$

d_i … out-degree of node i

This formulation assumes that ***M*** has no dead ends. We can either preprocess matrix ***M*** to remove all dead ends or explicitly follow random teleport links with probability 1.0 from dead-ends.

- The google matrix A

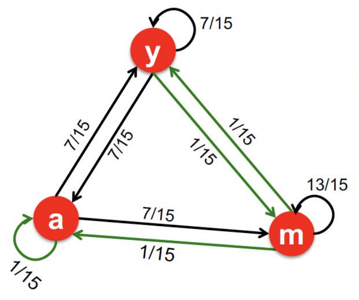
  $A = \beta M + (1-\beta)\left[\frac{1}{N}\right]_{N\times N}$

  $[1/N]_{NxN}$…N by N matrix where all entries are 1/N

  In practice $\beta = 0.8,\ 0.9$ (make 5 steps on avg., jump)

- We have a recursive problem: $r = A \cdot r$

- Example: $\beta = 0.8$



$$0.8 \overset{\textbf{M}}{\begin{array}{|ccc|}\hline 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 1 \\ \hline\end{array}} + 0.2 \overset{[\mathbf{1/N}]_{\mathbf{NxN}}}{\begin{array}{|ccc|}\hline 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \\ \hline\end{array}}$$

$$\underset{\textbf{A}}{\begin{array}{c|ccc|} & & & \\ \hline y & 7/15 & 7/15 & 1/15 \\ a & 7/15 & 1/15 & 1/15 \\ m & 1/15 & 7/15 & 13/15 \\ \hline\end{array}}$$

$$\begin{array}{l} y \\ a \\ m \end{array} = \begin{array}{c} 1/3 \\ 1/3 \\ 1/3 \end{array} \quad \begin{array}{c} 0.33 \\ 0.20 \\ 0.46 \end{array} \quad \begin{array}{c} 0.24 \\ 0.20 \\ 0.52 \end{array} \quad \begin{array}{c} 0.26 \\ 0.18 \\ 0.56 \end{array} \quad \dots \quad \begin{array}{c} 7/33 \\ 5/33 \\ 21/33 \end{array}$$

iii. Computing PageRank

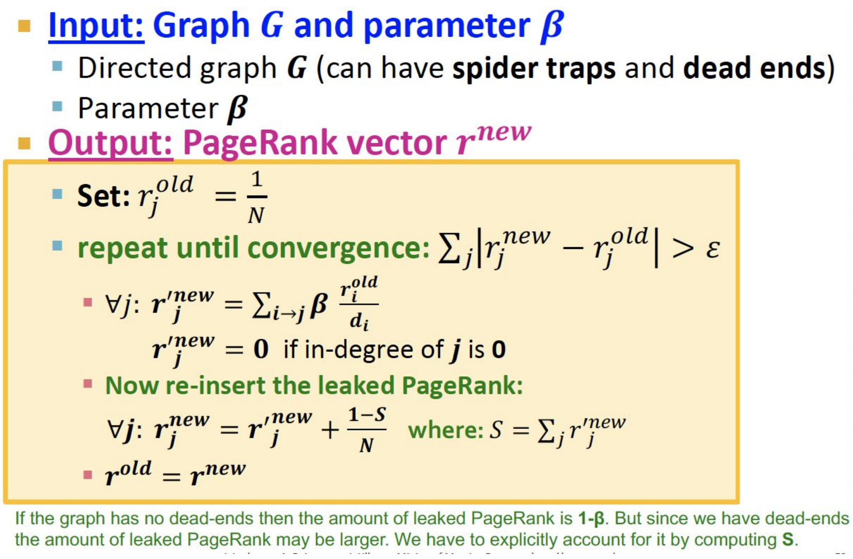
- o Key step is matrix-vector multiplication $r^{new} = A \cdot r^{old}$
- o Easy if we have enough main memory to hold A, r^{old}, r^{new}
  - We just rearrange the PageRank equation

    $$r = \beta M \cdot r + \left[\tfrac{1-\beta}{N}\right]_N$$

    where $\left[\frac{1-\beta}{N}\right]_N$ is a vector with all N entries $\frac{1-\beta}{N}$.
  - M is a sparse matrix (with no dead-ends)
  - So, in each iteration, we need to
    - ♦ Compute $r^{new} = \beta M \cdot r^{old}$
    - ♦ Add a constant value $\frac{1-\beta}{N}$ to each entry in r^{new}

Note: if M contains dead-ends then $\sum_i r_i^{new} < 1$ and we also have to renormalize r^{new} so that it sums to 1.

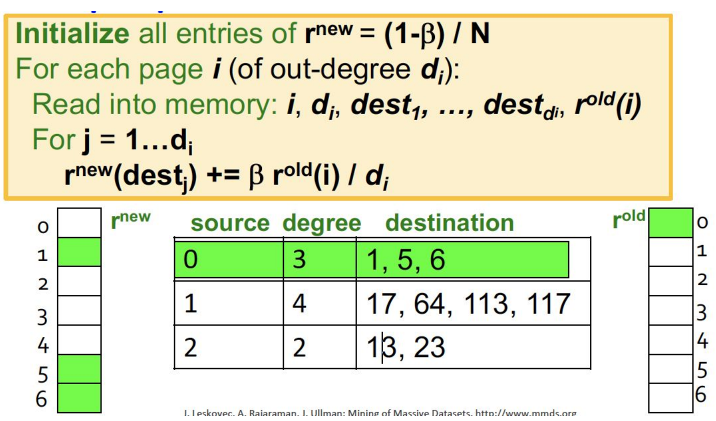
  - The complete algorithm



- **Input: Graph G and parameter β**
  - Directed graph G (can have **spider traps** and **dead ends**)
  - Parameter β
- **Output: PageRank vector r^{new}**
  - **Set:** $r_j^{old} = \frac{1}{N}$
  - **repeat until convergence:** $\sum_j \left| r_j^{new} - r_j^{old} \right| > \varepsilon$
    - $\forall j$: $r'^{new}_j = \sum_{i \to j} \beta \frac{r_i^{old}}{d_i}$
      $r'^{new}_j = 0$ if in-degree of j is 0
    - **Now re-insert the leaked PageRank:**
      $\forall j$: $r_j^{new} = r'^{new}_j + \frac{1-S}{N}$ where: $S = \sum_j r'^{new}_j$
    - $r^{old} = r^{new}$

If the graph has no dead-ends then the amount of leaked PageRank is **1-β**. But since we have dead-ends the amount of leaked PageRank may be larger. We have to explicitly account for it by computing **S**.

- o Assume enough RAM to fit r^{new} into memory, store r^{old} and matrix M on disk
  - ▪ 1 step of power-iteration is:



```
Initialize all entries of r^new = (1-β) / N
For each page i (of out-degree d_i):
  Read into memory: i, d_i, dest_1, …, dest_{d_i}, r^old(i)
  For j = 1…d_i
    r^new(dest_j) += β r^old(i) / d_i
```

r^{new}	
0	
1	🟩
2	
3	
4	
5	🟩
6	🟩

source	degree	destination
0	3	1, 5, 6
1	4	17, 64, 113, 117
2	2	13, 23

r^{old}	
0	🟩
1	
2	
3	
4	
5	
6	

J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, http://www.mmds.org

  - ▪ In each iteration, we have to:
    - ◆ Read r^{old} and matrix M
    - ◆ Write r^{new} back to disk
    - ◆ Cost per iteration of power method: $= 2|r| + M$
- o If we couldn't fit r^{new} in memory?
  - ▪ Break r^{new} into k blocks that fit in memory.
  - ▪ Scan M and r^{old} once for each block.

iv. Topic-Specific PageRank (measure popularity within a topic from a page)

- o Measures generic popularity of a page
- o Biased against topic-specific authorities
- o Allows each query to be answered based on interests of the user
- o Goal: Evaluate Web pages not just according to their popularity, but by how close they are to a particular topic, e.g. sports or history
  - ▪ Teleport go to a topic-specific set of relevant pages (teleport set)
  - ▪ SimRank: random walks from a fixed node on k-partite graphs

- Setting k-partite graph with k types of nodes
- Topic Specific PageRank from node u: teleport set = {u}
- Resulting scores measures similarity to node u
- Problem:
  - Must be done once for each node u
  - Suitable for sub-web-scale applications

v. Summary

- Normal PageRank:
  - Teleports uniformly at random to any node
  - All nodes have the same probability of surfer landing there:

  S = [0.1, 0.1, 0.1]
- Topic-Specific PageRank (Personalized PageRank)
  - Teleports to a topic specific set of pages
  - Nodes can have different probabilities of surfer landing there:

  S = [0.1, 0, 0.2]
- Random Walk with Restarts
  - Topic-Specific PageRank where teleport is always to the same node.

  S = [0, 1, 0]

vi. Hubs-and-authorities

- Uses a single measure of importance

c. Web spam detection algorithms

i. Spamming: any deliberate action to boost a web page's position in search engine results, incommensurate with page's real value

ii. Spam: web pages that are result of spamming.

iii. TrustRank = topic-specific PageRank with a teleport set of trusted pages

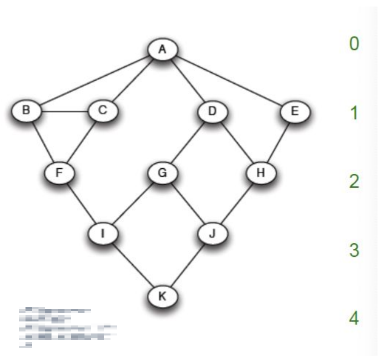
- Artificial link topographies crated in order to boost page rank
- Combating link spam
  - Detection and blacklisting of structures that look like spam farms
- Combating term spam
  - Analyse text using statistical methods
  - Similar to email spam filtering
  - Also, useful to detect approximate duplicate page
- Basic principle: approximate isolation
  - Sample a set of seed pages from the web

  - Have an oracle (human) to identify the good pages and the spam pages in the seed set

d. Community detection

  i. Method 1: Strength of weak ties
    - Edge betweenness: Number of shortest paths passing over the edge

  ii. Girvan-Newman
    - Divisive hierarchical clustering based on the notion of edge betweenness
      - Number of shortest paths passing through the edge
    - Algorithm
      - Repeat until no edges are left:
        - Calculate betweenness of edges
        - Remove edges with highest betweenness
      - Connected components are communities
      - Gives a hierarchical decomposition of network



    - How to compute betweenness?
      - Find start node. e.g. node A
      - Breath first search starting from A (Figure 1)
      - Count the number of shortest paths from A to all other nodes of the network (Figure 2)

Figure 2

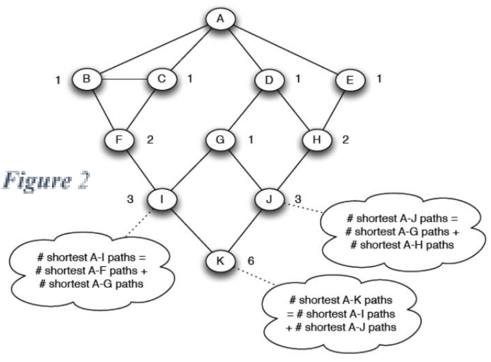
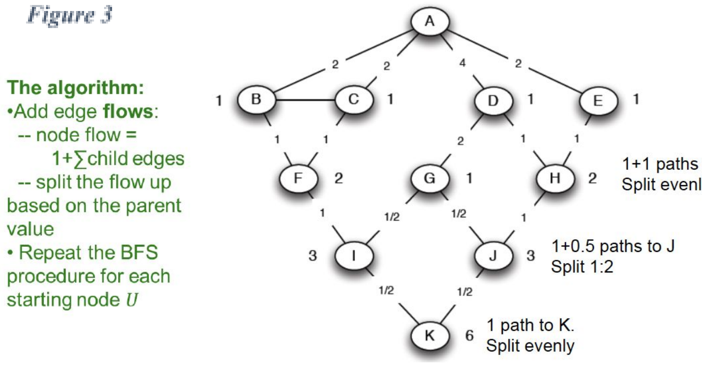

A
1 B
C 1
D 1
E 1
F 2
G 1
H 2
3 I
J 3
K 6
# shortest A-I paths = # shortest A-F paths + # shortest A-G paths
# shortest A-J paths = # shortest A-G paths + # shortest A-H paths
# shortest A-K paths = # shortest A-I paths + # shortest A-J paths

Figure 3


The algorithm:
•Add edge flows:
-- node flow = 1+∑child edges
-- split the flow up based on the parent value
• Repeat the BFS procedure for each starting node U

      - Compute betweenness by working up the tree: If there are multiple paths count them fractionally (Figure 3)
    - How to select the number of clusters?
      - Communities: set of tightly connected nodes
      - Modularity Q is useful for selecting the number of clousters.
        - Definition:

- A measure of how well a network is partitioned into communities
- Given a partitioning of network into groups $s \in S$:

Marked part need a null model

$$Q \propto \sum_{s \in S} [(\#edges\ within\ group\ s) - (expected\ \#\ edges\ within\ group\ s)]$$

- Modularity of partitioning S of graph G:

$$Q(G,S) = \underbrace{\frac{1}{2m}}_{\text{Normalizing cost.: -1<Q<1}} \sum_{s \in S} \sum_{i \in s} \sum_{j \in s} \left(A_{ij} - \frac{k_i k_j}{2m} \right)$$

A_{ij} = 1 if i→j, 0 else

- Modularity values take range [-1, 1]
  - It is positive if the number of edges within groups exceeds the expected number
  - $0.3 - 0.7 < Q$ means significant community structure
- Null model: configuration model
  - Given real G on n nodes and m edges, construct rewired network G'
    - Same degree distribution but random connections
    - Consider G' as a multigraph
    - The expected number of edges between nodes i and j of degrees k_i and k_j equals to:

$$k_i \bullet \frac{k_j}{2m} = \frac{k_i k_j}{2m}$$

    - The expected number of edges in multigraph G':

Note: $\sum_{u \in N} k_u = 2m$

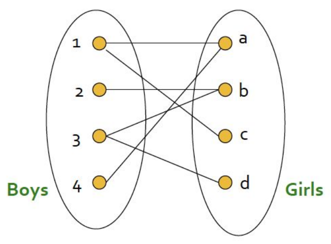
$$= \frac{1}{2} \sum_{i \in N} \sum_{j \in N} \frac{k_i k_j}{2m} = \frac{1}{2} \bullet \frac{1}{2m} \sum_{i \in N} k_i \left(\sum_{j \in N} k_j \right) = \frac{1}{4m} 2m \cdot 2m = m$$

e. Advertising on the Web

i. Online Algorithms is similar to the data stream model

  o You get to see the input one piece at a time and need to make irrevocable decisions along the way

ii. Online Bipartite Matching



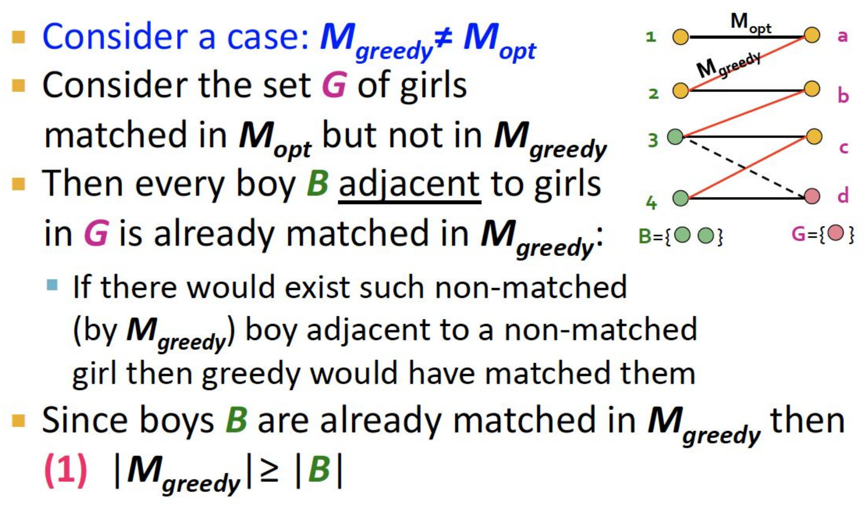
Nodes: Boys and Girls; Edges: Preferences

Goal: Match boys to girls so that maximum number of preferences is satisfied

- Perfect matching: all vertices of the graph are matched
- Maximum matching: a matching that contains the largest possible number of matches
- Greedy algorithm for online graph matching problem:
  - Pair the new girl with any eligible boy (if there is none, do not pair girl)
  - Evaluation:
    - Competitive ratio: for input I, support greedy produces matching M_{greedy} while an optimal matching is M_{opt}

$$Competitive\ ratio = min_{all\ possible\ inputs\ I}(\frac{|M_{greedy}|}{|M_{opt}|})$$

  - Analysing:



- Consider a case: $M_{greedy} \neq M_{opt}$
- Consider the set G of girls matched in M_{opt} but not in M_{greedy}
- Then every boy B adjacent to girls in G is already matched in M_{greedy}:
  - If there would exist such non-matched (by M_{greedy}) boy adjacent to a non-matched girl then greedy would have matched them
- Since boys B are already matched in M_{greedy} then (1) $|M_{greedy}| \geq |B|$

M_{opt}
M_{greedy}
1
2
3
4
a
b
c
d
B={● ●}
G={●}

  - Summary:
    - Girls G matched in M_{opt} but not in M_{greedy}

$$|M_{greedy}| \geq |B|$$

    - There are at least $|G|$ such boys $|G| \leq |B|$ otherwise the optimal algorithm couldn't have matched all girls in $|G|$,

so $|G| \leq |B| \leq |M_{greedy}|$

    - By definition of G also: $|M_{opt}| \leq |M_{greedy}| + |G|$

□ Worst case is when $|G| = |B| = |M_{greedy}|$

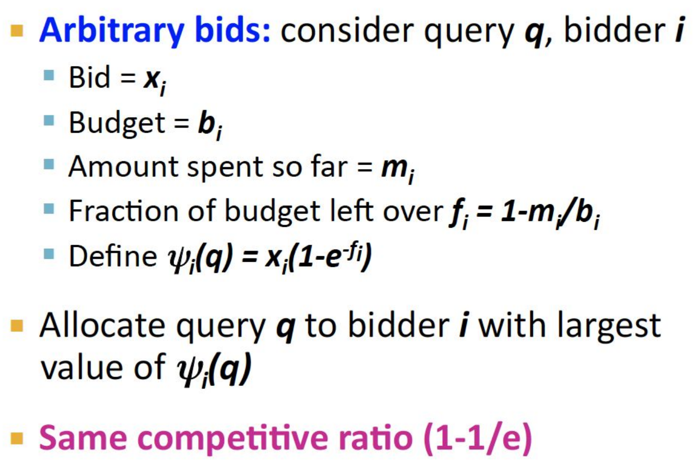
- ♦ $|M_{opt}| \leq 2|M_{greedy}|$ then $\frac{|M_{greedy}|}{|M_{opt}|} \geq \frac{1}{2}$

iii. Web advertising

- o Adwords:
  - ▪ advertisers bid on search keywords
  - ▪ When someone searches for that keyword, the highest bidder's ad is shown
  - ▪ Advertiser is charged only if the ad is clicked on
- o Goal: maximize search engine's revenues
  - ▪ Simple solution: use the expected revenue per click

(i.e., bid*CTR [click through rate])

  - ▪ Complications
    - ♦ Budget: each advertiser has a limited budget (guarantees not charge more than their daily budget)
    - ♦ CTR (click through rate) of an is unknown.
      - ⍰ Each ad has different likelihood of being clicked
      - ⍰ CTR is measured historically
  - ▪ Greedy algorithm
  - ▪ Balance algorithm
    - ♦ For each query, pick the advertiser with the largest unspent budget
    - ♦ In general: revenue = budget * number_of_advertiser (1-1/e)
    - ♦ Competitive ratio = 1-1/e
    - ♦ Generalized balance



- **Arbitrary bids:** consider query ***q***, bidder ***i***
  - Bid = x_i
  - Budget = b_i
  - Amount spent so far = m_i
  - Fraction of budget left over $f_i = 1 - m_i/b_i$
  - Define $\psi_i(q) = x_i(1-e^{-f_i})$
- Allocate query ***q*** to bidder ***i*** with largest value of $\psi_i(q)$
- **Same competitive ratio (1-1/e)**

10. Parallel Computing

a. Data Parallelism: independent tasks apply same operation to different elements of a data set.

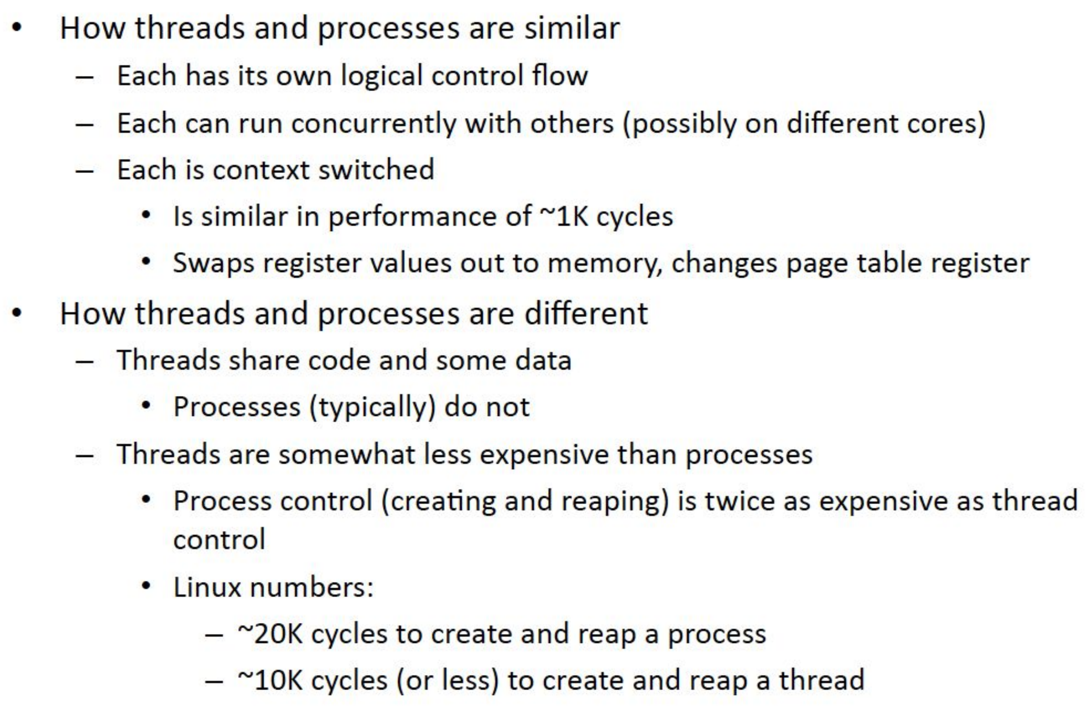
b. Functional parallelism: independent tasks apply different operations to different data elements.

c. Pipelining:

   i. Divide a process into independent stages
   ii. Move objects through stages in sequence
   iii. At any given times, multiple objects being processed

d. Dependences and hazards
   i. If two instructions are data dependent, they cannot execute simultaneously.
   ii. Whether a dependence results in a hazard and whether that hazard actually causes a stall are properties of the pipeline organization.
   iii. Data dependences may occur through registers ore memory.

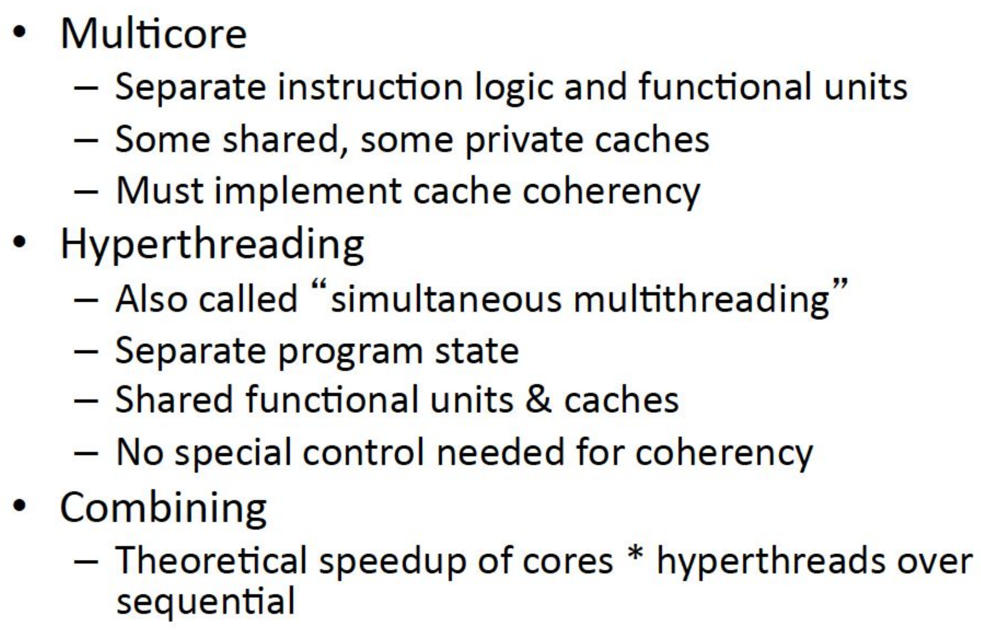
e. Threads vs. Processes



- How threads and processes are similar
  - Each has its own logical control flow
  - Each can run concurrently with others (possibly on different cores)
  - Each is context switched
    - Is similar in performance of ~1K cycles
    - Swaps register values out to memory, changes page table register
- How threads and processes are different
  - Threads share code and some data
    - Processes (typically) do not
  - Threads are somewhat less expensive than processes
    - Process control (creating and reaping) is twice as expensive as thread control
    - Linux numbers:
      - ~20K cycles to create and reap a process
      - ~10K cycles (or less) to create and reap a thread

   i. Processes:
      - o hard to share resources: easy to avoid unintended sharing
      - o High overhead in adding/removing clients
   ii. Threads:
      - o Easy to share resources: perhaps too easy
      - o Medium overhead
      - o Not much control over scheduling policies
      - o Difficult to debug
   iii. Thread-based designs
      - o Easy to share data structures between threads
      - o Threads are more efficient than processes
      - o Unintentional sharing can introduce subtle and hard to reproduce errors

f. Creating parallel machines



- Multicore
  - Separate instruction logic and functional units
  - Some shared, some private caches
  - Must implement cache coherency
- Hyperthreading
  - Also called "simultaneous multithreading"
  - Separate program state
  - Shared functional units & caches
  - No special control needed for coherency
- Combining
  - Theoretical speedup of cores * hyperthreads over sequential

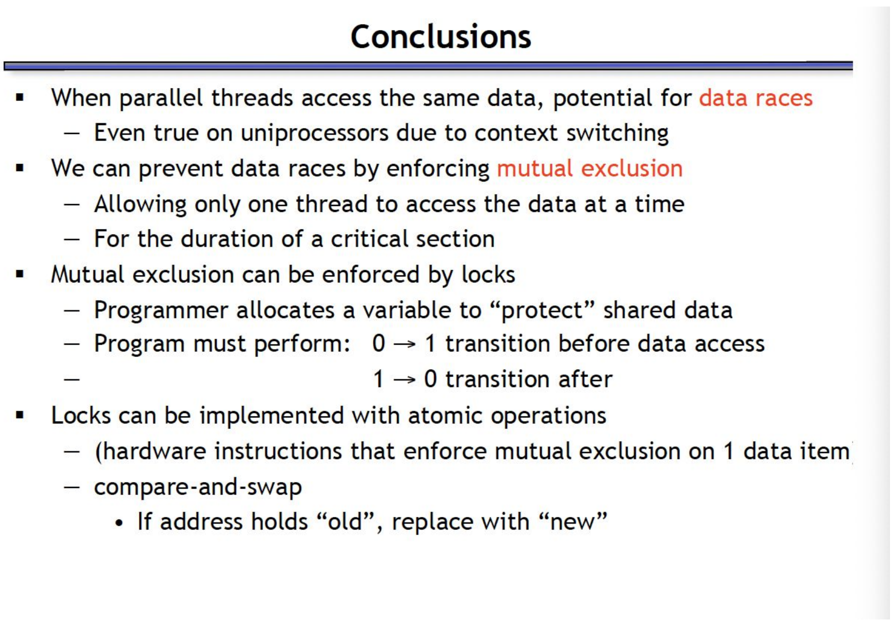
g. Single instruction stream, multiple data streams (SIMD) Parallelism

i. Exploit significant data level parallelism for
   - o Matrix- oriented scientific computing
   - o Media-oriented image an sound processors

ii. More energy efficient than MIMD

iii. Allows programmer to continue to think sequentially

iv. Contains:
   - o Vector architectures
     - ▪ Basic idea
       - ♦ Read sets of data elements into 'vector registers'
       - ♦ Operate on those registers
       - ♦ Disperse the results back into memory
     - ▪ Register are controlled by compiler
       - ♦ Used to hide memory latency
       - ♦ Leverage memory bandwidth
   - o Multimedia extensions.



Conclusions

- When parallel threads access the same data, potential for data races
  - Even true on uniprocessors due to context switching
- We can prevent data races by enforcing mutual exclusion
  - Allowing only one thread to access the data at a time
  - For the duration of a critical section
- Mutual exclusion can be enforced by locks
  - Programmer allocates a variable to “protect” shared data
  - Program must perform: 0 → 1 transition before data access
  - 1 → 0 transition after
- Locks can be implemented with atomic operations
  - (hardware instructions that enforce mutual exclusion on 1 data item
  - compare-and-swap
    - If address holds “old”, replace with “new”

- Graphics processor units
  - Memory wall: the growing disparity of speed between the chip and performing off-chip memory transactions.
  - Memory latency is a barrier to performance improvements
    - Latency: the amount of time it takes for an operation to complete
  - Memory wall: due to latency and limited communication bandwidth beyond chip boundaries.
    - Bandwidth: how much data can be transferred per second
  - Writing CUDA software:
    - High Priority Recommendations:
      - To get the maximum benefit from CUDA, focus first on finding ways to parallelize your solution.
      - Use the effective bandwidth of your computation as a metric when measuring performance and optimization benefits.
      - Minimize data transfer between the host and the device, even if it means running some kernels on the device that do not show performance gains when compared with running them on the host CPU.
      - Ensure global memory accesses are coalesced whenever possible.
      - Minimize the use of global memory. Prefer shared memory access where possible.
      - Avoid different execution paths within the same warp.
    - Medium Priority Recommendations:

            - Accesses to shared memory should be designed to avoid serializing requests due to bank conflicts.
            - To hide latency arising from register dependencies, maintain sufficient numbers of active threads per multiprocessor (i.e., sufficient occupancy).
            - The number of threads per block should be a multiple of 32 threads because this provides optimal computing efficiency and facilitates coalescing.
            - Use the fast math library whenever speed is important and you can live with a tiny loss of accuracy.
            - Prefer faster, more specialized math functions over slower, more
            - general ones when possible
        - ♦ Low Priority Recommendations:
            - For kernels with long argument lists, place some arguments into constant memory to save shared memory.
            - Use shift operations to avoid expensive division and modulo calculations.
            - Avoid automatic conversion of doubles to floats.
            - Make it easy for the compiler to use branch predication in lieu of loops or control statements.
    - ▪ The common pattern to CUDA Programming
        - ♦ Phase 1: Allocate memory on the device and copy to the device the data required to carry out computation on the GPU.
        - ♦ Phase 2: GPU crunches numbers based on the kernel you defined.
        - ♦ Phase 3: Bring back the results from the GPU. Free memory on the device (clean up…)
    - ▪ Rules of Thumb for Efficient GPU Computing:
        - ♦ 1. Get the data on the GPU and keep it there
        - ♦ 2. Give the GPU enough work to do
        - ♦ 3. Focus on data reuse within the GPU to avoid memory bandwidth limitations

h. Multiple instruction streams, single data streams (MISD) – no commercial implementation

i. Multiple instruction streams, multiple data streams (MIMD)

11. Python MPI https://nyu-cds.github.io/python-mpi/
12. The raft consensus algorithm (http://raftconsensus.github.io)
    a. Consensus
        i. Agreement on shared state (single system image)
        ii. Recovers from server failures autonomously
        iii. Key to building consistent storage systems
    b. Raft overview
        i. Leader election
            - Select one of the servers to act as cluster leader
            - Detect crashes, choose new leader
        ii. Log replication (normal operation)
            - Leader takes commands from clients, appends them to its log
            - Leader replicates its log to other servers (overwriting inconsistencies)
        iii. Safety
            - Only a server with an up-to-date log can become leader
    c. RaftScope visualization
        i. Core Raft Review
            - Leader election
                - Heartbeats and timeouts to detect crashes
                - Randomized timeouts to avoid split votes
                - Majority voting to guarantee at most one leader per term
            - Log replication (normal operation)
                - Leader takes commands from clients, appends them to its log
                - Leader replicates its log to other servers (overwriting inconsistencies)
                - Built-in consistency check simplifies how logs may differ
            - Safety
                - Only elect leaders with all committed entries in their logs
                - New leader defers committing entries from prior terms
    d. Conclusions
        i. Consensus widely regarded as difficult
        ii. Raft designed for understandability
            - Easier to teach in classrooms
            - Better foundation for building practical systems
        iii. Paper/thesis covers much more
            - Cluster membership changes (simpler in thesis)

- Log compaction (expanded tech report/thesis)
- Client interaction (expanded tech report/thesis)
- Evaluation (thesis)

Point to point communication in MPI is a message sent directly from Process A to Process B
In MPI, deadlock occurs when at least one process is blocked waiting for a communication it will never receive

```
[('John', 'Sarah'), ('John', 'Malcolm'), ('John', 'Winnie'),
('Malcolm', 'Sarah'), ('Sarah', 'Winnie'),
 ('Malcolm', 'Winnie'),
 ('Malcolm', 'Sarah'), ('Sarah', 'Winnie'), ('Fred', 'Sarah'),
('Malcolm', 'Winnie'), ('Fred', 'Malcolm'),
 ('Fred', 'Winnie'),
('Sarah', 'Winnie'), ('Fred', 'Sarah'), ('Charles', 'Sarah'),
 ('Fred', 'Winnie'), ('Charles', 'Winnie'),
'Charles', 'Fred'),

('Malcolm', 'Sarah'), ('Sarah', 'Winnie'), ('Charles', 'Sarah'),
('Malcolm', 'Winnie'), ('Charles', 'Malcolm'),
('Charles', 'Winnie')]
```