**Lecture 1**

Items = products; Baskets = sets of products someone bought in one trip to the store

Baskets = sentences; Items = documents containing those sentences

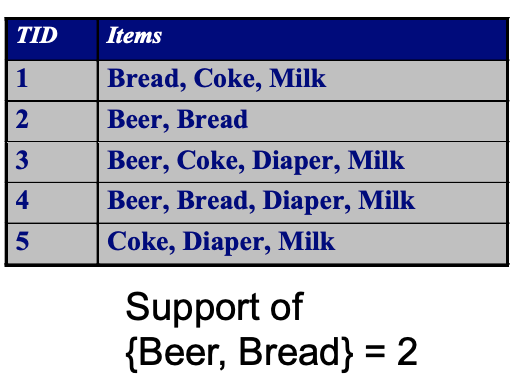
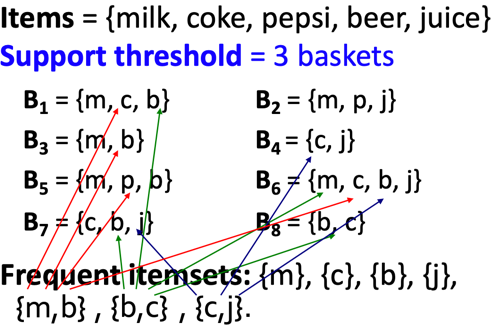
Baskets = patients; Items = drugs & side-effects

There are also many-to-many mapping association between two kinds of things.

Baskets = patients; Items = drugs & side-effects

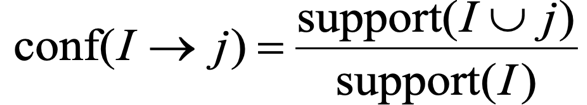
**Support** for itemset **I**: Number of baskets containing all items in **I.**

Given a **support threshold s**, then sets of items that appear in at least **s** baskets are called ***frequent itemsets***

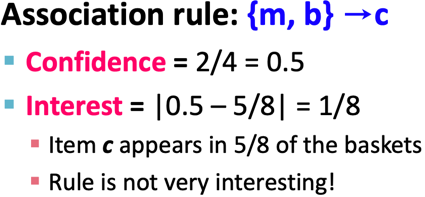
**ASSOCIATION RULES:** If-then rules about the contents of baskets

* **{i1, i2,…,ik}** → j means: “if a basket contains all of i1,…,ik then it is likely to contain j”
* In practice there are many rules, want to find significant/interesting ones!
* **Confidence** of this association rule is the probability of j given **I = {i1,…,ik}**



* **Interest** of an association rule **I → j:** difference between its confidence and the fraction of baskets that contain





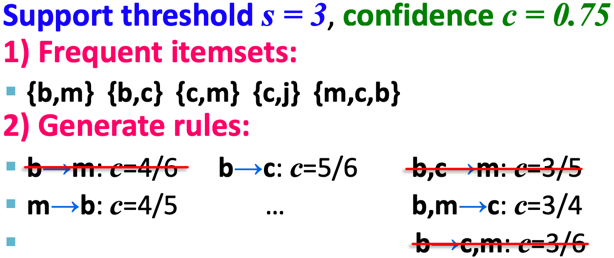
MINING ASSOCIATION RULES:

Step 1: find all frequent itemsets I

Step 2: Rule generation



Example



The true cost of mining disk-resident data is usually the number of disk I/Os.

In practice, association-rule algorithms read the data in passes – all baskets read in turn

We measure the cost by the number of passes an algorithm makes over the data.

For many frequent itemst algorithms, main-memory is the critical resource – As we read baskets, we need to count something, e.g., occurrences of pairs of items.

The hardest problem often turns out to be finding the frequent pairs of items {i1, i2,} –

why? Because frequent pairs are common, frequent triples are rare.

why? Because probability of being frequent drops exponentially with size.

Naïve Algorithm – Naïve approach to finding frequent pairs

* read file once
* counting in main memory the occurrences of each pair.
* From each basket of **n** items, generate its **n(n-1)/2** pairs by two nested loops
* Fails if (#items)2 exceeds main memory

Counting Pairs in Memory

Two approaches:

Approach 1: count all pairs using a matrix – only requires 4 bytes per pair

* Triangular Matrix
* n = total number items
* count pair of items {i, j} only if i < j
* keep pair counts in lexicographic order:

- {1,2}, {1,3},…, {1,n}, {2,3}, {2,4},…,{2,n}, {3,4},…

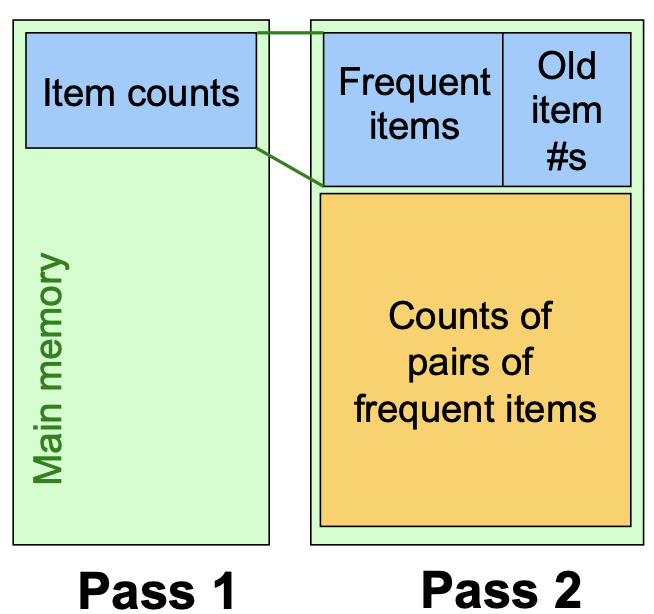
* pair {i, j} is at position (i – 1)(n – i/ 2) + j -1
* Total number of pairs **n(n-1)/2**; total bytes = **2n2**

Approach 2: keep triples [i, j, c] = “the count of the pair of items {i, j} if c – uses 12 bytes per pair (but only for pairs with count > 0)

**A-Priori Algorithm** – A two-pass approach called A-Priori limits the need for main memory

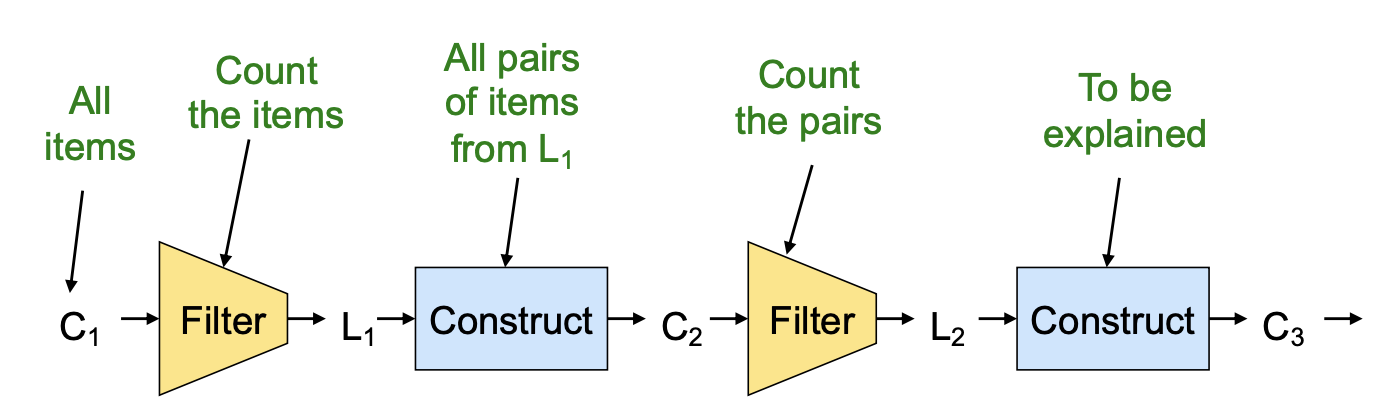
Key idea: monotonicity – if a set of items **I** appears at least **s** times, so does every **subset J** of **I**

Contrapositive for pairs: if item **I** does not appear in **s** baskets, then no pair including **i** can appear in **s** baskets.

* 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)
* **
* For each k, we construct

§ **Ck** = candidate = those that might be frequent sets (support **>= s**) based on information from the pass for **k–1**

§ **Lk** = the set of truly frequent **k**-tuples

* **

Hypothetical steps of the A-Priori algorithm

* C1 = { {b} {c} {j} {m} {n} {p} }
* Count the support of items in C1
* Prune non-frequent: L1 = { b, c, j, m }
* Generate C2 = { {b,c} {b,j} {b,m} {c,j} {c,m} {j,m} }
* Count the support of items in C2
* Prune non-frequent: L2 = { {b,m} {b,c} {c,m} {c,j} }
* Generate C3 = { {b,c,m} {b,c,j} {b,m,j} {c,m,j} }
* Count the support of items in C3
* Count the support of itemsets in C3
* Prune non-frequent: L3 = { {b,c,m} }

**PCY (Park-Chen-Yu) Algorithm** – In A-priori, most memory is idle in pass 1, so PCY questions if that idle memory can be used to reduce memory required in pass 2.

* Pass 1 of PCY: 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
* That is, for each bucket just keep the count, not the actual pairs that hash to the bucket!
* FOR (each basket) :

FOR (each item in the basket):

add 1 to item’s count;

FOR (each pair of items):

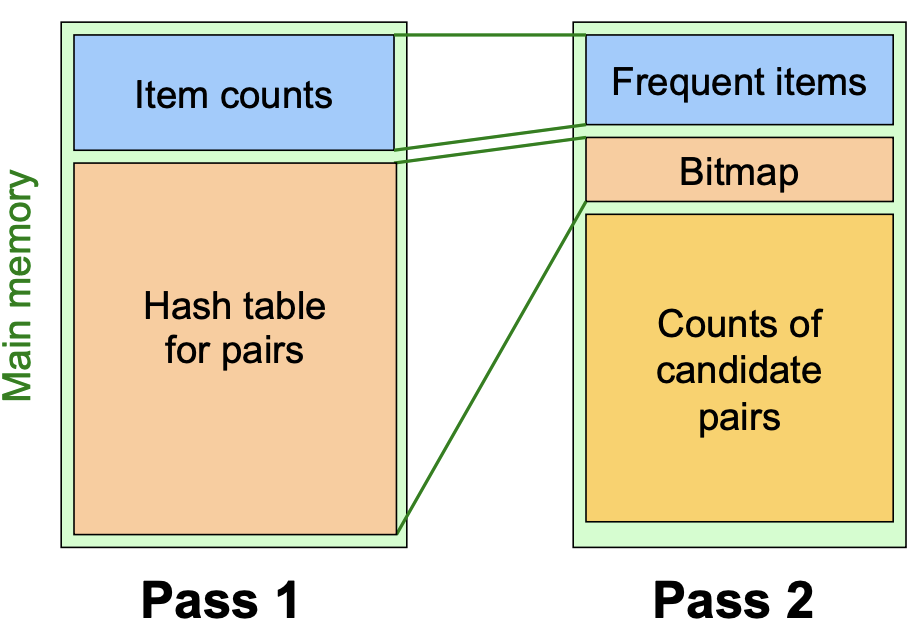
hash the pair to a bucket;

add 1 to the count for that bucket

* NOTE 1
* Pairs of items are not present in file. They need to be generated from the input file.
* We are not interested in the presence of a pair, but we need to see whether it is present at least **s**(support) times.
* NOTE 2
* A bucket with total count less than **s**(support), none of its pairs can be frequent.
* A bucket can be frequent without any frequent pair, but is surely frequent if it contains frequent pairs.
* Pass 2 of PCY: Only count pairs that hash to frequent buckets

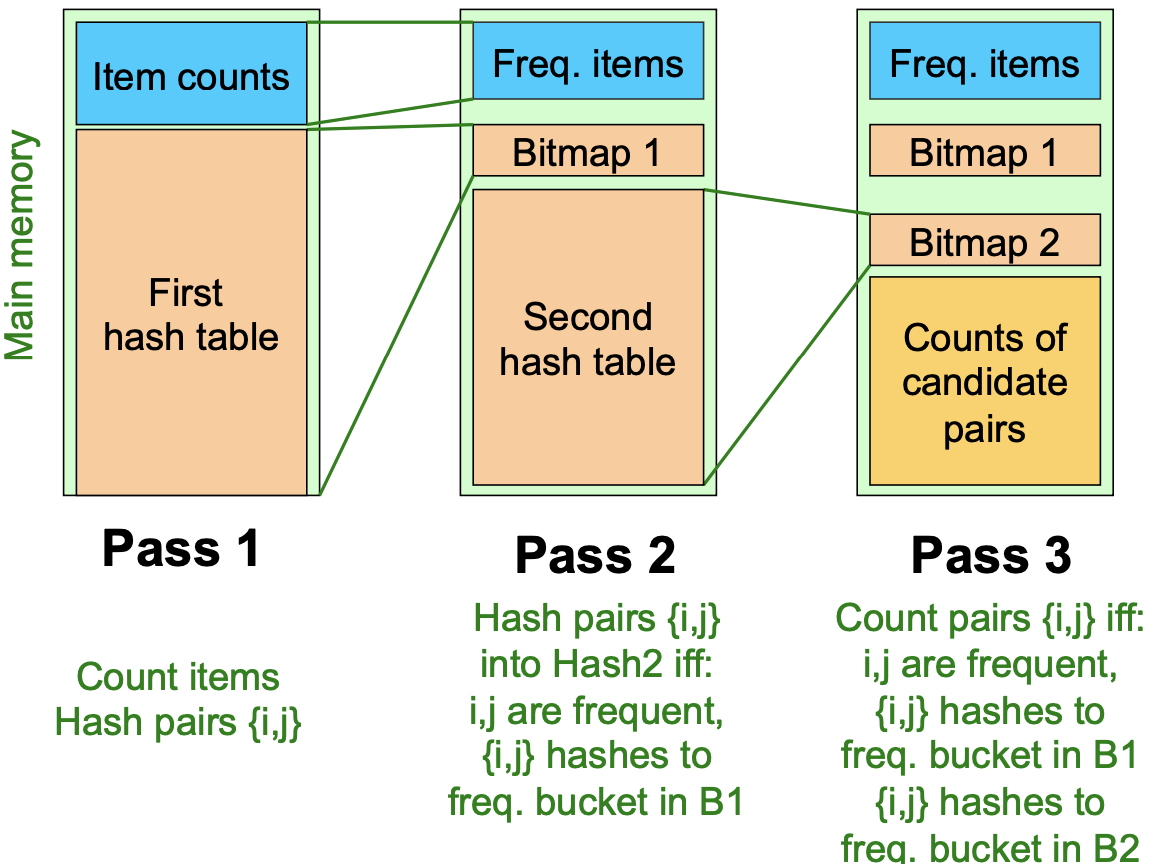
Count all pairs {i, j} that meet the conditions for being a candidate pair:

1. Both i and j are frequent items
2. The pair {i, j} hashes to a bucket whose bit in the bit vector is 1 (i.e., a frequent bucket)

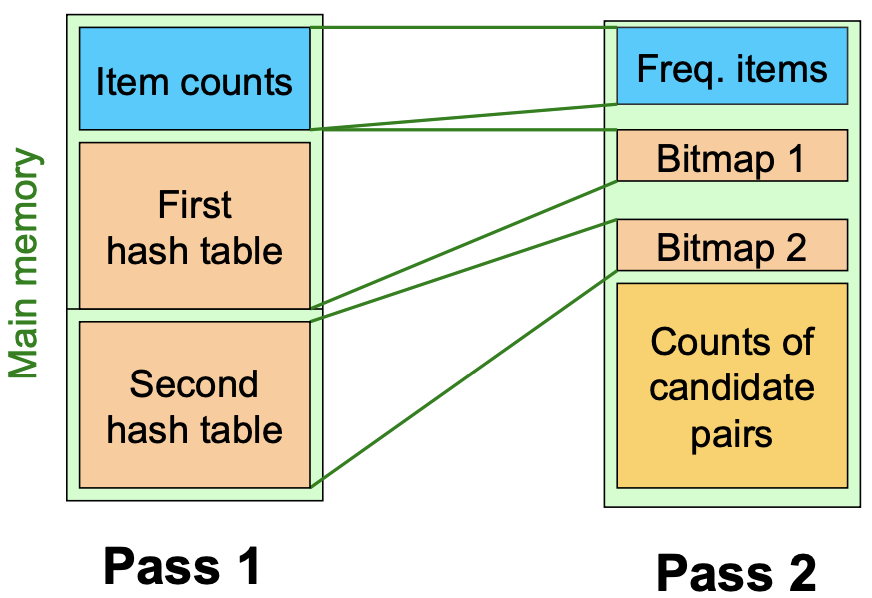


* Key idea: After Pass 1 of PCY, rehash only those pairs that qualify for Pass 2 of PCY.
* (Multistage)Pass 3 of PCY: Count only those pairs {i, j} that satisfy these candidate pair conditions:

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



Refinement : (Mutlihash)

****

**PCY: Extensions**

Either multistage or multihash can use more than two hash functions;

1. Multistage - there is a point of diminishing returns, since the bit-vectors eventually consume all of main memory
2. Multihash – the bit-vectors occupy exactly what one PCY bitmap does, but too many hash functions makes all counts **>= s**

**Randomized Algorithms (Sampling) (1)**

* Take a random sample of the market baskets
* Run a-priori or one of its improvements
* Main memory is half (copy of sample baskets) and half (space for counts).

**Randomized Algorithms (Sampling) (2)**

* Avoid false positives
* Smaller threshold helps catch more truly frequent itemsets

**SON Algorithm (Segments) (1)**

* Repeatedly read small subsets of the baskets into main memory and run an in memory algorithm to find all frequent itemsets. (processing entire file in memory-sized chunks not sampling)
* An itemset becomes a candidate if it is found to be frequent in any one or more subsets of the baskets.

**SON Algorithm (Segments) (2)**

* On a second pass
* Count all the candidate items
* Determine which are frequent in the entire set
* **Key “monotonicity” idea:** an itemset cannot be frequent in the entire set of baskets unless it is frequent in at least on subset.

**SON – Distributed Version**

* SON lends itself to distributed data mining
* Baskets distributed among many nodes
* **Compute** frequent itemsets at each node
* **Distribute** candidates to all nodes
* **Accumulate** the counts of all candidates

**Lecture 2 – Finding Similar Items**

Find near-neighbors in high-dimensional space is a case where a problem is expressed in finding similar sets. Examples are;

* Pages with similar words :- Duplicate detection(Mirror Pages, Plagiarism)
* Customers who purchased similar products :- Online Purchases(Amazon)

Problems:

* Too many documents to compare all pairs
* Documents are so large or so many that they cannot fit in main memory
* Many small pieces of one document can appear out of order in another

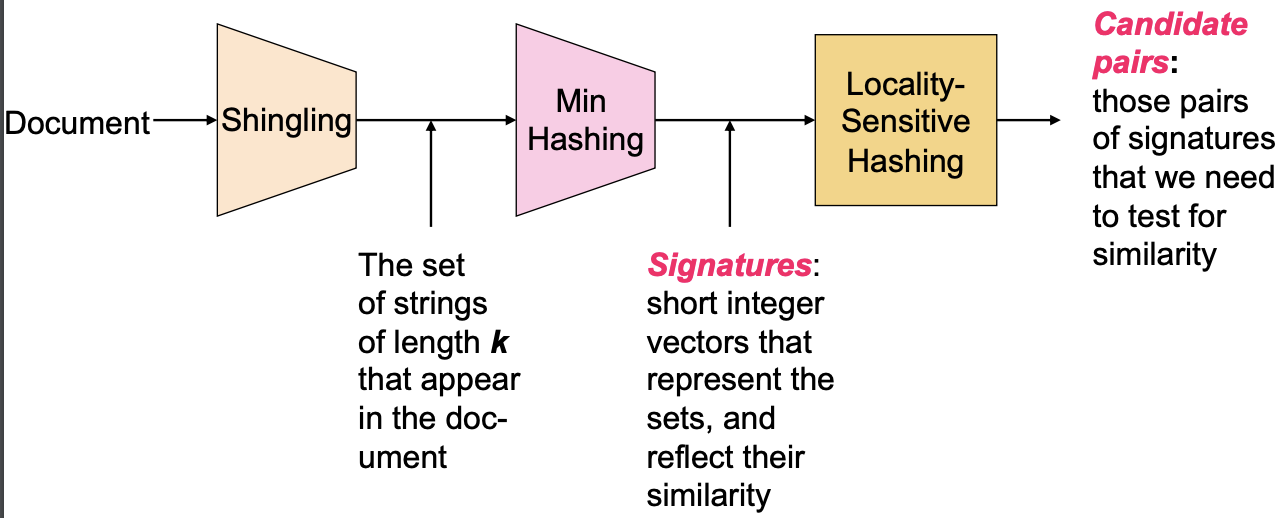
**Jaccard Similarity –** The Jaccard similarity of two sets is the size of their intersection divided by the size of their union.

****

**Jaccard Distance –** 1 minus the Jaccard Distance



**3 Essential Steps for Similar Docs**



**Shingles –** convert documents to sets

* Documents as High-Dim Data
* Definition of Shingles

Example: **k=2**; document **D1** = abcab

Set of 2-shingles: **S(D1)** = {ab, bc, ca}

**Option:** Shingles as a bag (multiset), count ab twice: **S’(D1)** = {ab, bc, ca, ab}

* Compressing shingles

To compress long shingle, we can hash (or simply map) them to 4 bytes

* Similarity Metric for shingles
* Document D1 is a set of its k-shingles C1=S(D1)
* Equivalently, each document is a 0/1 vector in the space of k-shingles
* Each unique shingle is a dimension
* Working Assumption

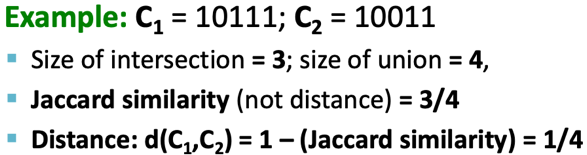
Documents that have lots of shingles in common have similar text, even if the text appears in different order.

Caveat: You must pick k large enough, or most documents will have most shingles

* k = 5 is OK for short documents.
* k = 10 is better for long documents
* Motivation for Minhash/LSH

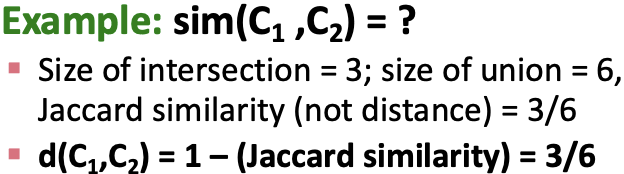
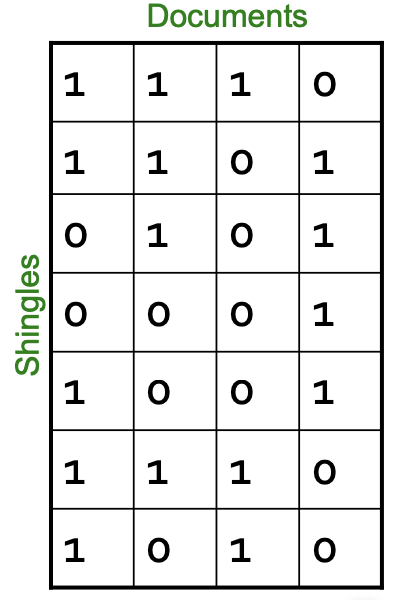
**Minhashing -**  convert large sets to short signatures, while preserving similarity

* Encoding Sets as Bit Vectors
* Interpret set intersection as bitwise **AND**, and set union as bitwise **OR**

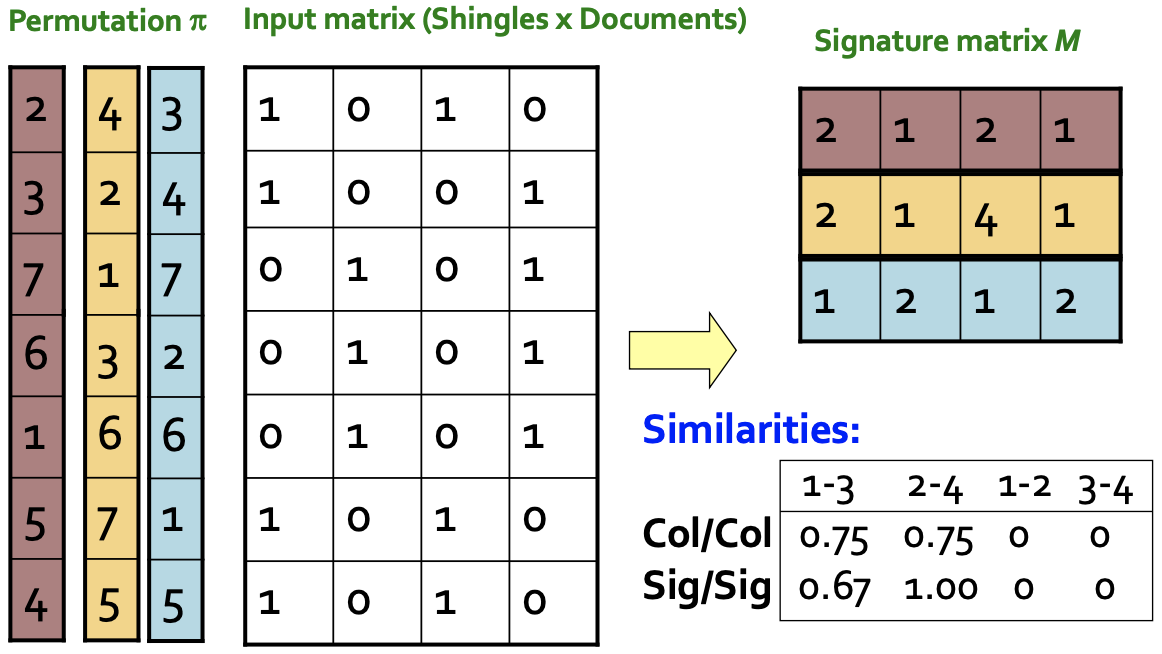


* From Sets to Boolean Matrices
* Rows = elements(shingles). Columns = sets(documents)

C1 C2 C3 C4



* Outline: Finding Similar Columns
* Documents -> sets of shingles
* **Similarity of columns == similarity of signatures**
* Hashing Columns(Signatures)
* **h(C)** is small enough that the signature fits in RAM
* **sim(C1, C2)** is the same as the “similarity” of signatures **h(C1)** and **h(C2)**
* Goal: Find a hash function h(·) such that:
* If sim(C1,C2) is high, then with high prob. h(C1) = h(C2)
* If sim(C1,C2) is low, then with high prob. h(C1) ≠ h(C2)
* Min-Hashing Example
* Check if the index of lowest number(0) in the permutation column is equals to 0 in the input matrix (Shingles & Documents matrix)
* if it is check if the index of the next lowest number(1) in the permutation column is equals to 0. Else if it is equals to 1 in the input matrix (Shingles & Documents matrix) add that number from the permutation to the signature matrix
* if it is
* check if the index of the next lowest number(2) in the permutation column is equal to 0
* if it is check if the index of the next lowest number(1) in the permutation column is equals to 0. Else if it is equals to 1 in the input matrix (Shingles & Documents matrix) add that number from the permutation to the signature matrix
* repeat until you get a complete signature matrix.



* Similarity for Signatures

We know: **Pr[h(C1) = h(C2)] = sim(C1, C2)**

The similarity of two signatures is the fraction of the hash functions in which they agree

**Locality-Sensitive Hashing –** Focus on pairs of signatures likely to be from similar document

* (candidate pairs)

**LSH**

False positives

False negatives

**b – bands , r – rows, s – similarity**

Columns C1 and C2 have similarity **s**

Pick any band (r rows)

Prob. that all rows in band equal = **sr**

Prob. that some row in band unequal = **1 - sr**

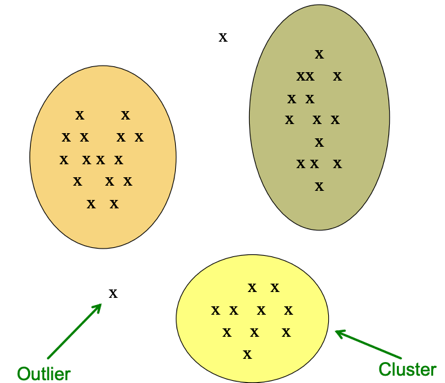
Prob. that no band identical = **(1 - sr )b**

Prob. that at least 1 band identical = **1 - (1 - sr )b**

**Lecture 3 – Clustering**

Hierarchical Clustering

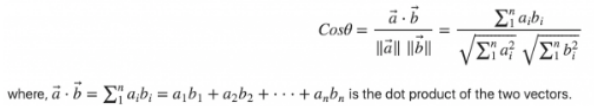
* The problem of clustering - Given a set of points, with a notion of distance between points, group the points into some number of clusters, so that;
* Members of a cluster are close/similar to each other
* Members of different clusters are dissimilar
* Example: Clusters & Outliers

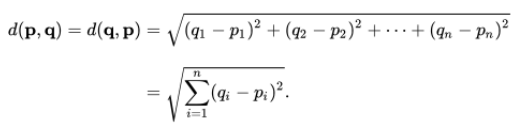


* Why is it hard?
* Clustering in two dimensions looks easy
* Clustering small amounts of data looks easy
* And in most cases, looks are not deceiving
* Many applications involve not 2, but 10 or 10,000 dimensions
* High-dimensional spaces look different: Many pairs of points are at about the same distance.
* Clustering Problem: Documents

Represent a document by a vector (x1, x2,…, x3), where xi = 1 iff the i th word (in some order) appears in the document.

* Sets as vectors: Measure similarity by the **Cosine Distance**



* Sets as sets: Measure similarity by the **Jaccard Distance**
* Sets as points: Measure similarity by **Euclidean Distance**
* Hierarchial
* Agglomerative (bottom up)
* Divisive (top down)
* Key operation: Repeatedly combine two nearest clusters
* Three important questions

1. How do you represent a cluster of more than one point?
2. How do you determine the “nearness” of clusters
3. When to stop combing clusters

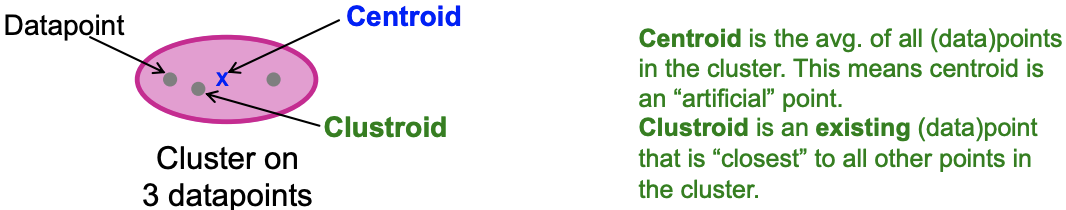
* Euclidean case:

1. How do you represent a cluster of more than one point? – **centroid** = average of its data points
2. How do you determine the “nearness” of clusters? – measure cluster distance by distances of centroids
3. When to stop combing clusters

* Non-Euclidean case:
* APPROACH 1

1. How do you represent a cluster of more than one point? – **clustroid** = data point “**closest**” to other points. Where “closest” means:

- Smallest maximum distance to other points

- Smallest average distance to other points

1. How do you determine the “nearness” of clusters? – measure cluster distance by distances of clustroids as if they were centroids.
2. When to stop combing clusters

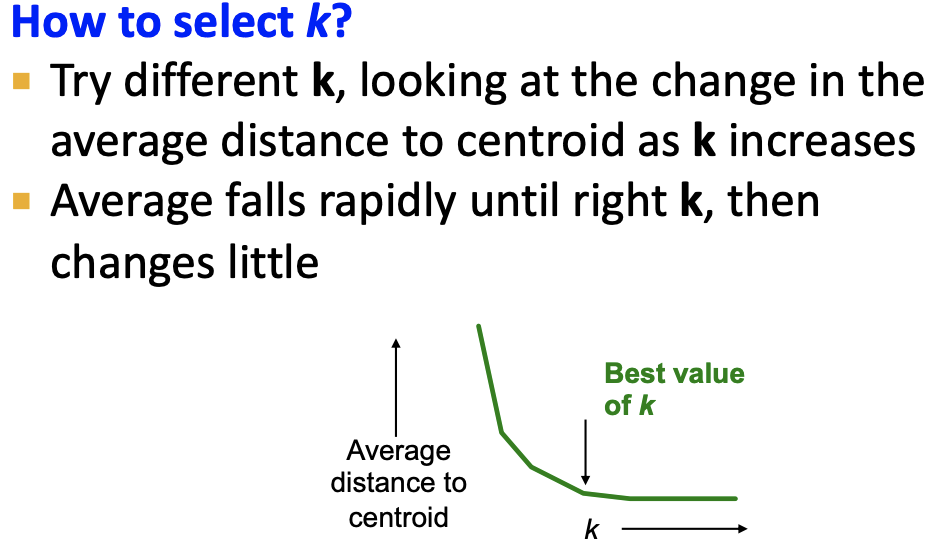
* APPROACH 2

K-Means Clustering Initialization

* Assumes Euclidean space/distance
* Start by picking **k,** the number of clusters
* 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
4. Repeat 2 and 3 until convergence. Convergence: Points don’t move between clusters and centroids stabilize



BFR Algorithm

Bradley-Fayyad-Reina is a variant of k-means designed to handle **very large** (disk-resident) data sets. Clusters are axis aligned ellipses.

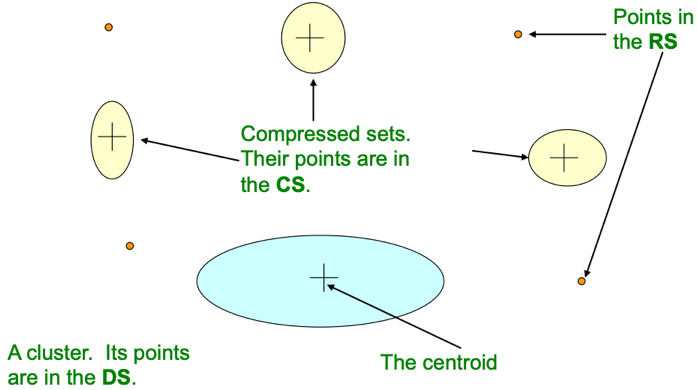
* Algorithm
* Points are read from disk into main memory in chunks.
* Most points from previous memory loads are summarized by simple statistics
* To begin, from the initial load we select the initial k centroids by some sensible approach:

1. Take k random points
2. Take a small random sample and cluster optimally
3. Take a sample; pick a random point, and then k–1 more points, each as far from the previously selected points as possible

* Three Classes of Points

3 sets of points which we keep track of:

* 1. Discard set (DS): Close enough to a centroid to be summarized. Only summaries are kept in main memory
  2. Compression set (CS): Summarized, but not assigned to a cluster. Only summaries are kept in main memory
  3. Retained set (RS): Isolated points to be assigned to a compression set. Held in main memory exactly as they are.



**Summarizing Sets of Points**

* The number of points, **N**
* The vector **SUM**, whose ith component is the sum of the coordinates of the points in the ith dimension
* The vector **SUMSQ**: ith component = sum of squares of coordinates in ith dimension
* **2d + 1** values represent any size cluster
* **d** = number of dimensions
* Average in each dimension (the centroid) can be calculated as **SUMi / N**
* **SUMi** = ith component of SUM
* Variance of a cluster’s discard set in dimension ***i*** is: **(SUMSQi / N) – (SUMi / N)2**
* Standard deviation is the square root of variance

Initialization in CURE

The CURE Algorithm is a 2 pass algorithm.

* Problem with BFR:
* Assumes clusters are normally distributed in each dimension
* And axes are fixed – ellipses at an angle are not **OK**
* CURE(Clustering Using REpresentatives):
* Assumes a Euclidean distance
* Allows clusters to assume any shape
* Use a collection of representative points to represent clusters

**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**

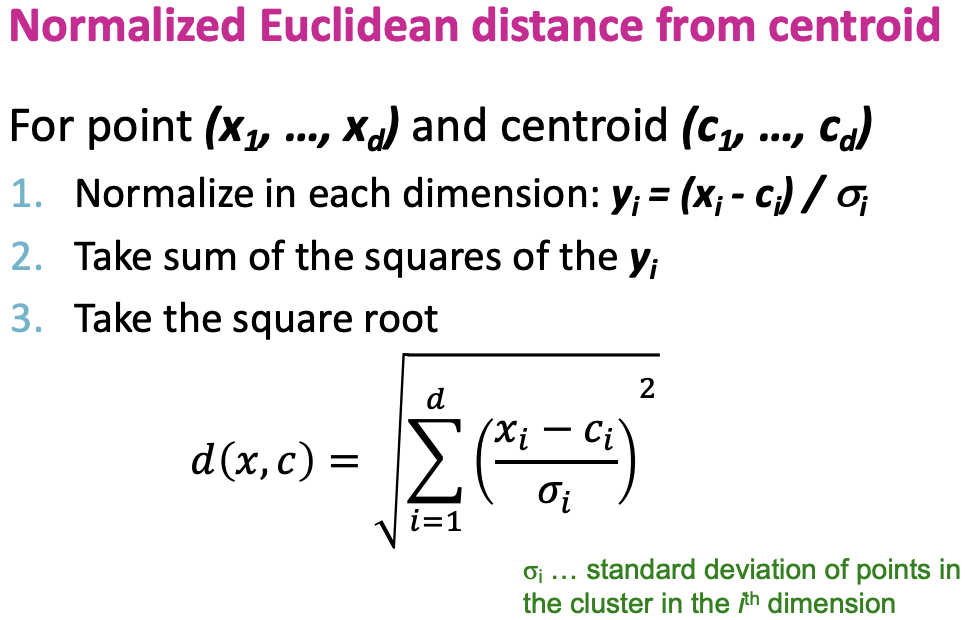
Now, rescan the whole dataset and visit each point p in the data set.

Place it in the “**closest cluster**”

* Normal definition of “closest”: Find the closest representative to **p** and assign it to representative’s cluster

Mahalanobis

* **Q1) We need a way to decide whether to put a new point into a cluster (and discard)**
* e.g, BFR suggests: The Mahalanobis distance is less than a threshold



* Q2) Should 2 CS subclusters be combined?
* Compute the variance of the combined subcluster
* N, SUM, and SUMSQ allow us to make that calculation quickly (oval/ ellipse shape)
* Combine if the combined variance is below some threshold(circular shape)

**RESEARCH**

