Finding Similar Sets

Applications

Shingling

Minhashing

Locality-Sensitive Hashing

A Common Metaphor

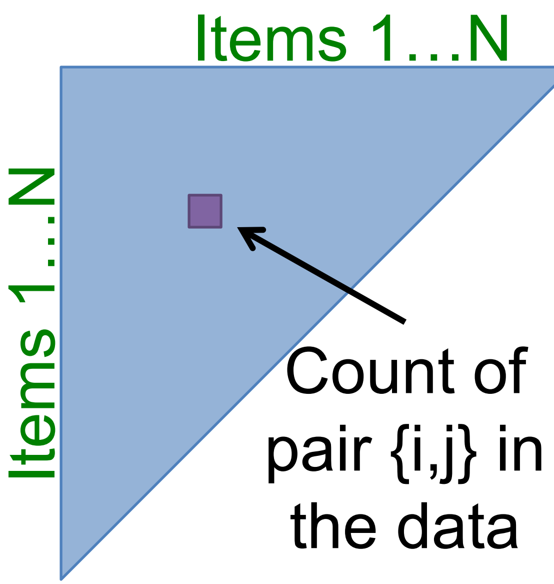
- **Many problems can be expressed as finding "similar" sets:**
  - **Find near-neighbors in high-dimensional space**
- **Examples:**
  - **Pages with similar words**
    - For duplicate detection, classification by topic
  - **Customers who purchased similar products**
    - Products with similar customer sets
  - **Images with similar features**
    - Users who visited similar websites

Problem to solve

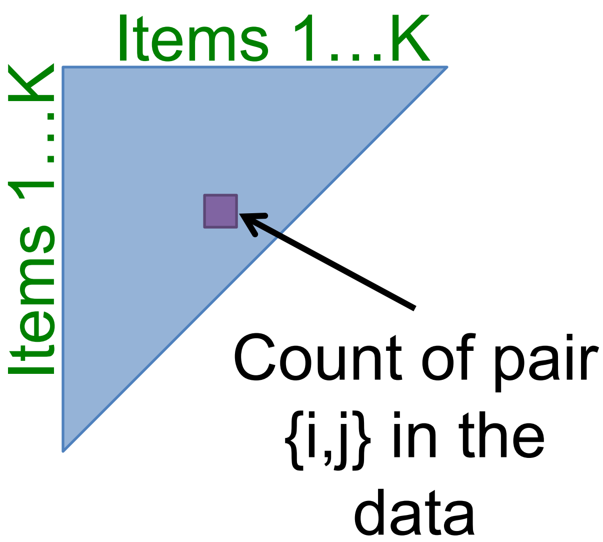
- Given high-dimensional data points
- And a distance function
  - That quantifies the distance between points
- **Find all pairs of points that are within some distance threshold**
- Naïve solution would take $O(N^2)$ for N points
- We’ll look at O(N) solutions

Relation to Previous Lectures

◆ **Ch. 6:** Finding frequent pairs



Naïve solution:
Single pass but requires space quadratic in the number of items $O(N^2)$

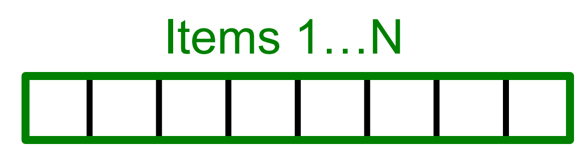


A-Priori:
First pass: Find frequent singletons
For a pair to be **a frequent pair candidate**, its singletons have to be frequent!
Second pass:
Count only candidate pairs!

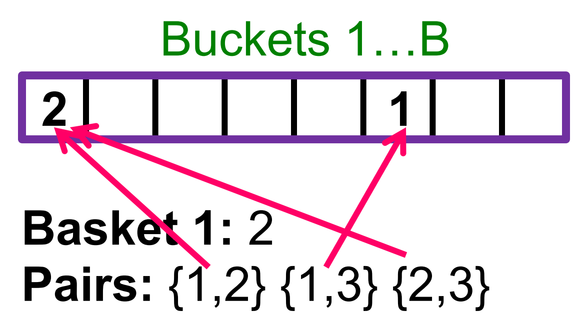
N … number of distinct items
K … number of items with support $\geq s$

Relation to Previous Lectures

- **Ch. 6:** Finding frequent pairs
- **Further improvement: PCY**
  - **Pass 1:**
    - Count exact frequency of each item:

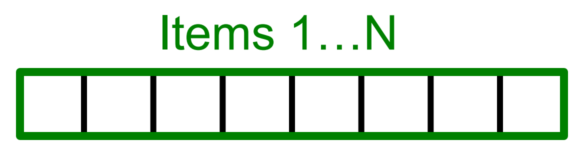


    - Take pairs of items {i,j}, hash them into *B* buckets and count of the number of pairs that hashed to each bucket:

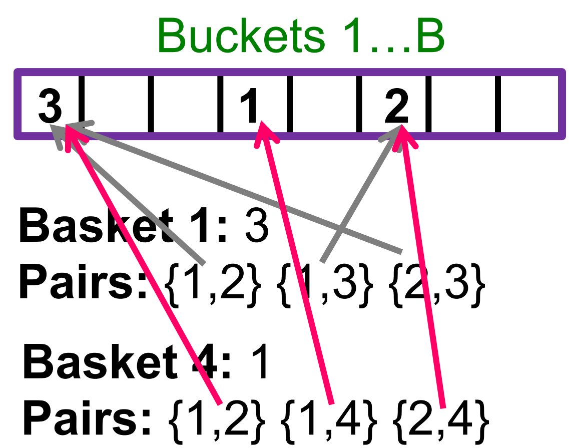


Relation to Previous Lecture

- **Ch. 6:** Finding frequent pairs
- **Further improvement: PCY**
  - **Pass 1:**
    - Count exact frequency of each item:



    - Take pairs of items {i,j}, hash them into *B* buckets and count of the number of pairs that hashed to each bucket:
  - **Pass 2:**
    - For a pair {i,j} to be a **candidate for a frequent pair,** its singletons {i}, {j} have to be frequent and the pair has to hash to a frequent bucket!



Relation to Previous Lecture

- **Last time:** Finding frequent pairs
- **Fu**

Ch. 6: A-Priori

Main idea: Candidates

Instead of keeping a count of each pair, only keep a count of candidate pairs!

Today's lecture: Find pairs of similar docs

Main idea: Candidates

-- Pass 1: Take documents and hash them to buckets such that documents that are similar hash to the same bucket

-- Pass 2: Only compare documents that are **candidates** (i.e., they hashed to a same bucket)

Benefits: Instead of O(N²) comparisons, we need O(N) comparisons to find similar documents

Finding Similar Items

Problem Formulation

- <u>Item</u> represented as a set of <u>objects</u>
  - “baskets”=?
- Problem becomes: find similar sets
  - ”finding similar items” = “finding items having similar objects”
- Challenges:
  - Large sets
  - Large number of items/sets

Distance Measures

- **Goal: Find near-neighbors in high-dimensional space**
  - We formally define “near neighbors” as points that are a “small distance” apart
- For each application, we first need to define what “**distance**” means
- **Today: Jaccard distance/similarity**

Task: Finding Similar Documents

- **Goal: Given a large number (in the millions or billions) of documents, find "near duplicate" pairs**
- **Applications:**
  - Mirror websites, or approximate mirrors
    - Don't want to show both in search results
  - Similar news articles at many news sites
    - Cluster articles by "same story"
- **Problems:**
  - Many small pieces of one document can appear out of order in another
  - Too many documents to compare all pairs
  - Documents are so large or so many that they cannot fit in main memory

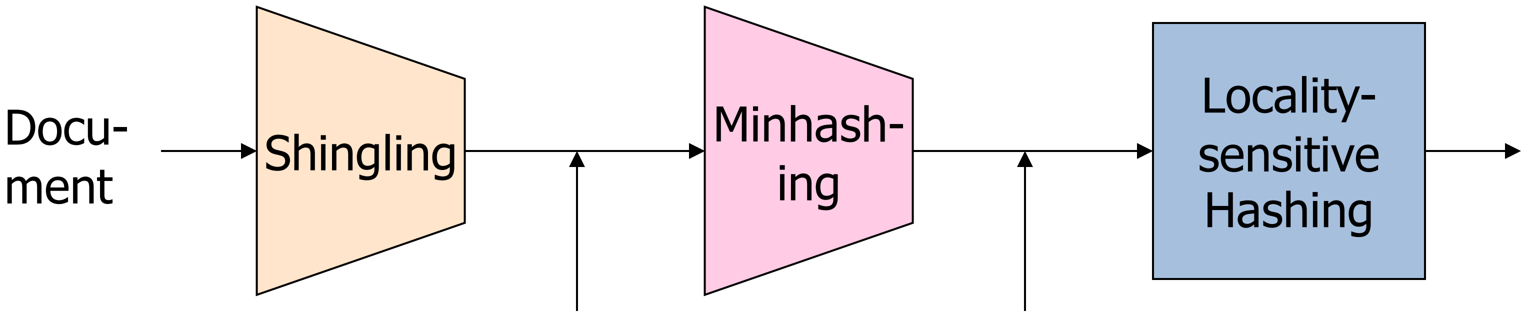
3 Essential Steps for Finding Similar Docs

1. ***Shingling:*** Convert documents to sets

2. ***Min-Hashing:*** Convert large sets to short signatures, while preserving similarity

3. ***Locality-Sensitive Hashing:*** Focus on pairs of signatures likely to be from similar documents
   - **Candidate pairs!**

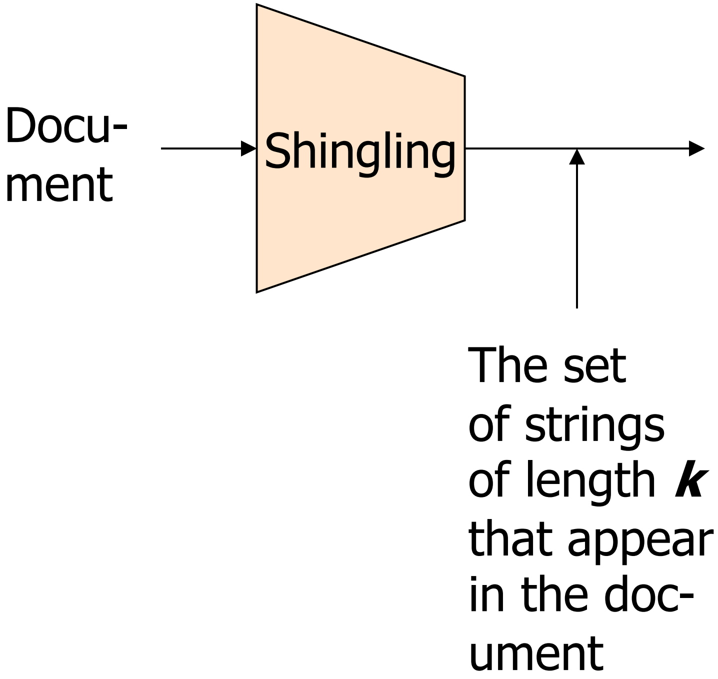
The Big Picture



The set of strings of length *k* that appear in the doc-ument

Signatures : short integer vectors that represent the sets, and reflect their similarity

Candidate pairs : those pairs of signatures that we need to test for similarity



Shingling

Step 1: ***Shingling:*** Convert documents to sets

Documents as High-Dimensional Data

- **Step 1: *Shingling:* Convert documents to sets**

- **Simple approaches:**
  - Document = set of words appearing in document
  - Document = set of “important” words
  - Don’t work well for this application. Why?

- **Need to account for ordering of words!**
- A different way: **Shingles!**

Define: 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 = characters, for examples

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

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

- May want to compress long shingles

Compressing Shingles

- To **compress long shingles**, we can **hash** them to numbers
  - Each number may be represented as (say) 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-values were shared
- **Example:** **k=2**; document $\mathbf{D_1}$= abcab
  Set of 2-shingles: $\mathbf{S(D_1)}$ = {ab, bc, ca}
  Hash the singles: $\mathbf{h(D_1)}$ = {1, 5, 7}

Why is compression needed?

- How many k-shingles?
  - Rule of thumb: imagine 20 characters in alphabet
  - Estimate of number of k-shingles is 20^k
  - 4-shingles: 20^4 or 160,000 or $2^{17.3}$
  - 9-shingles: 20^9 or 512,000,000,000 or 2^{39}
- How many buckets?
  - Assume we use 4 bytes to represent a bucket
  - Assume buckets are numbered in range 0 to $2^{32} - 1$
  - This is much smaller than possible number of 9-shingles
  - Compression
    - Represent each shingle with 4 bytes, not 9 bytes

Thought Question

- Why is it better to hash 9-shingles (say) to 4 bytes than to use 4-shingles?
- Hint: How random are the 32-bit sequences that result from 4-shingling?

Why hash 9-shingles to 4 bytes rather than use 4-shingles?

- With 4-shingles, $2^{17.3}$ possible singles
  - Most sequences of four bytes are unlikely or impossible to find in typical documents
  - Effective number of different shingles is much less than the number of buckets $2^{32} - 1$
    - Not efficient use of memory
- With 9-shingles, 2^{39} possible shingles
  - Many more than 2^{32} buckets
  - After hashing, may get any sequence of 4 bytes
    - Effectient use of memory

Similarity Metric for Shingles

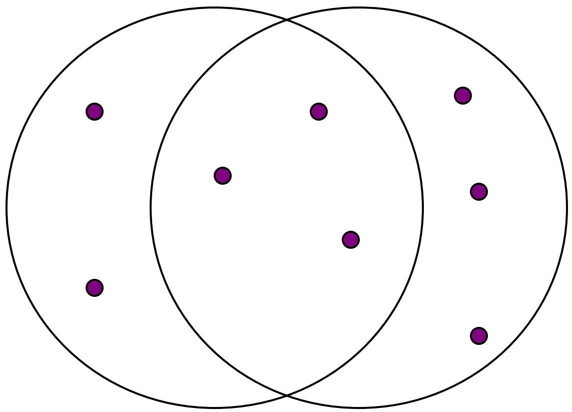
- **Document D_1 is a set of its k-shingles C_1=S(D_1)**
- Equivalently, each document is a vector of 0s,1s in the space of *k*-shingles
  - Each unique shingle is a dimension
  - Vectors are very sparse
- **A natural similarity measure is the Jaccard similarity**

Jaccard Similarity of Sets

- The *Jaccard similarity* of two sets is the size of their intersection divided by the size of their union

$$\boldsymbol{Sim}\ (C_1, C_2) = |C_1 \cap C_2| / |C_1 \cup C_2|$$

Example: Jaccard Similarity



3 in intersection
8 in union

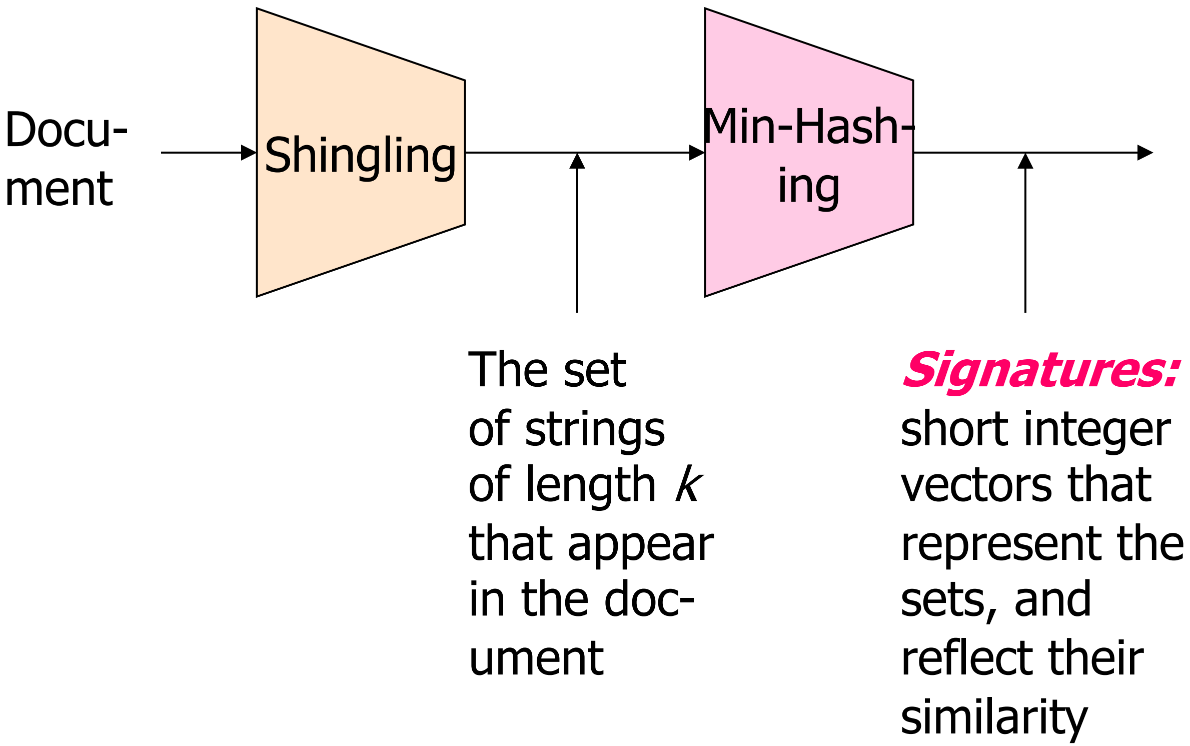
Jaccard similarity = 3/8

- **Jaccard distance** = 1 – Jaccard Similarity or 5/8 in this example

Motivation for Minhash/LSH

Use k-shingles to create Signatures: short integer vectors that represent sets and reflect their similarity

- **Suppose we need to find near-duplicate documents among million documents**
- Naïvely, we would have to compute **pairwise Jaccard similarities** for **every pair of docs**
  - **≈ $5*10^{11}$** comparisons
  - At 10^5 secs/day and 10^6 comparisons/sec, it would take **5 days**
- For million, it takes more than a year…



MinHashing

Step 2: ***Minhashing:*** Convert **large sets** to **short signatures**, while **preserving similarity**

From Sets to Boolean Matrices

- Rows = elements of the universal set
- Columns = sets
- 1 in row *e* and column *S* if and only if element *e* is a member of set *S*
- Column similarity is the Jaccard similarity of the sets of their rows with 1: intersction/union of sets
- Typical matrix is sparse (many 0 values)
  - May not really represent the data by a boolean matrix
  - Sparse matrices are usually better represented by the list of non-zero values (e.g., triples)
  - But the matrix picture is conceptually useful

Example 3.6

Element	S_1	S_2	S_3	S_4
a	1	0	0	1
b	0	0	1	0
c	0	1	0	1
d	1	0	1	1
e	0	0	1	0

- Universal set: {a, b, c, d, e}
- Matrix represents sets chosen from universal set
- S1 = {a, d}, S2 = {c], S3 = {b, d, e} and S4 = {a, c, d}
- Example: rows are products and columns are customers, represented by set of items they bought
- Jacquard similarity of S1, S4: intersection/union = 2/3

Example: Jaccard Similarity of Columns

C_1	C_2		
0	1		*
1	0		*
1	1	*	*
0	0		
1	1	*	*
0	1		*

Sim (C_1, C_2) =

2/5 = 0.4

When Is Similarity Interesting?

1. When the sets are so large or so many that they cannot fit in main memory
2. Or, when there are so many sets that comparing all pairs of sets takes too much time
3. Or both

Outline: Finding Similar Columns

1. Compute signatures of columns = small summaries of columns
2. Examine pairs of signatures to find similar signatures
   - Essential: “similarities of signatures” and “similarities of columns” are related
3. Optional: check that columns with similar signatures are really similar

Warnings

1. Comparing all pairs of signatures may take too much time, even if not too much space
   - A job for Locality-Sensitive Hashing
2. These methods can produce false negatives, and even false positives (if the optional check is not made)

Signatures

- Key idea: "hash" each column *C* to a small *signature* *Sig*(C), such that:
  1. *Sig* (C) is small enough that we can fit a signature in main memory for each column
  2. *Sim* (C_1, C_2) is the same as the "similarity" of *Sig* (C_1) and *Sig* (C_2)

Four Types of Rows

- Given columns C_1 and C_2, there are 4 types of rows and may be classified as:

	C_1	C_2	
a	1	1	← type "a"
b	1	0	← type "b"
c	0	1	← type "c"
d	0	0	← type "d"

- Also, a = "# rows of type a" , etc.
- Note $Sim(C_1, C_2) = a/(a+b+c)$
  - Jacquard similarity: intersection/union
  - "a" is intersection, "$a+b+c$" is union

Minhashing

- To ***minhash*** a set represented by a column of the matrix, **pick a random permutation of the rows**
- **Define "hash" function *h* (*C*) = the number of the first (in the permuted order) row in which column *C* has 1**
- Use several (e.g., 100) independent hash functions to create a signature

Minhashing Example (3.7)

Element	S_1	S_2	S_3	S_4
a	1	0	0	1
b	0	0	1	0
c	0	1	0	1
d	1	0	1	1
e	0	0	1	0

Permute

Element	S_1	S_2	S_3	S_4
b	0	0	1	0
e	0	0	1	0
a	1	0	0	1
d	1	0	1	1
c	0	1	0	1

- To minhash a set represented by a column of the characteristic matrix, pick a permutation of the rows
- The minhash value of any column is the "index" number of the first row, in the permuted order, in which that column has a 1
- For set S1, first 1 appears in row a, so:

h(S1) = a

h(S2) = c

h(S3) = b

h(S4) = a

Minhashing Example

1	4	3
3	2	4
7	1	7
6	3	6
2	6	1
5	7	2
4	5	5

Input matrix

1	0	1	0
1	0	0	1
0	1	0	1
0	1	0	1
0	1	0	1
1	0	1	0
1	0	1	0

Signature matrix *M*

2	1	2	1
2	1	4	1
1	2	1	2

Surprising Property: Connection between Minhashing and Jaccard Similarity

- The probability that minhash function for a random permutation of rows produces same value for two sets equals Jaccard similarity of those sets
  - **"The probability that *h*(C_1)=*h*(C_2)" is the same as "*Sim* (C_1, C_2)"**
- Recall four types of rows:

	C_1	C_2
a	1	1
b	1	0
c	0	1
d	0	0

- **Sim(C_1, C_2) for both Jacquard and Minhash are $a/(a+b+c)$!**
  - Why? Look down the permuted columns C_1 and C_2 until we see a 1
  - If it's a type-*a* row, then $h(C_1) = h(C_2)$. If a type-*b* or type-*c* row, then not. (Don't count the *type-d* rows)

Similarity for Signatures

- Sets represented by characteristic matrix M
- To represent sets: pick at random some number n of permutations of the rows of M
  - 100 permutations or several hundred
- Call minhash functions determined by these permutations $h_1, h_2, \ldots, h_n$
- From column representing set S, **construct minhash signature for S**:
  - vector $[h_1(S), h_2(S), \ldots, h_n(S)]$, usually represented as column
- Construct a ***signature matrix***: i^{th} column of M replaced by minhash signature for i^{th} column
- **The *similarity of signatures* is the fraction of the hash functions in which they agree**

Min Hashing – Example

1	4	3
3	2	4
7	1	7
6	3	6
2	6	1
5	7	2
4	5	5

Input matrix

1	0	1	0
1	0	0	1
0	1	0	1
0	1	0	1
0	1	0	1
1	0	1	0
1	0	1	0

Signature matrix *M*

2	1	2	1
2	1	4	1
1	2	1	2

Similarities:

	1-3	2-4	1-2	3-4
Col/Col	0.75	0.75	0	0
Sig/Sig	0.67	1.00	0	0

Minhash Signatures

- Pick (say) 100 random permutations of the rows
- Think of *Sig* (C) as a column vector
- Let *Sig* (C)[i] =

  according to the *i* th permutation, the number of the first row that has a 1 in column *C*

Implementation – (1)

- Not feasible to permute a large characteristic matrix explicitly
  - Suppose 1 billion rows
  - Hard to pick a random permutation from 1…billion
  - Representing a random permutation requires 1 billion entries
  - Accessing rows in permuted order leads to thrashing

- **Can simulate the effect of a random permutation by a random hash function**
  - Maps row numbers to as many buckets as there are rows
  - May have collisions on buckets
  - Not important as long as number of buckets is large

Implementation – (2)

- A good approximation to permuting rows: pick around 100 hash functions
- For each:
  - column *c* (set representing a document)
  - hash function h_i
- Keep a "slot" in signature matrix $M(i,c)$
- Intent: **$M(i,c)$ will become the smallest value of $h_i(r)$ for which column c has 1 in row r**
  - $h_i(r)$ gives order of rows for i^{th} permuation

Implementation – (3)

for each row r

for each column c

if c has 1 in row r

for each hash function h_i **do**

if $h_i(r)$ is a smaller value than $M(i, c)$ **then**

$M(i, c) := h_i(r)$;

Computing Minhash Signatures: Example 3.8

Row	S_1	S_2	S_3	S_4	$x+1 \mod 5$	$3x+1 \mod 5$
0	1	0	0	1	1	1
1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

Two hash functions give permutations of rows:
h1 = x+1 mod 5, h2 = 3x +1 mod 5

	S_1	S_2	S_3	S_4
h_1	∞	∞	∞	∞
h_2	∞	∞	∞	∞

Initial signature matrix

	S_1	S_2	S_3	S_4
h_1	1	∞	∞	1
h_2	1	∞	∞	1

For row 0: Replace existing signature values with lower hash values for S1 and S4, since both have 1 in row

Computing Minhash Signatures: Example 3.8 (part 2)

Row	S_1	S_2	S_3	S_4	$x+1 \mod 5$	$3x+1 \mod 5$
0	1	0	0	1	1	1
1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

	S_1	S_2	S_3	S_4
h_1	1	∞	2	1
h_2	1	∞	4	1

For row 1: replace h1 and h2 values for S3, since row has a 1 and values are lower

	S_1	S_2	S_3	S_4
h_1	1	3	2	1
h_2	1	2	4	1

For row 2: replace values for S2 since set has a 1 value. Do not replace values for S4, because existing values are lower

Computing Minhash Signatures: Example 3.8 (part 3)

Row	S_1	S_2	S_3	S_4	$x+1 \mod 5$	$3x+1 \mod 5$
0	1	0	0	1	1	1
1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

	S_1	S_2	S_3	S_4
h_1	1	3	2	1
h_2	0	2	0	0

For row 3: don't replace h1 values--all are below 4; replace h2 values with 0 for S1, S3, S4

	S_1	S_2	S_3	S_4
h_1	1	3	0	1
h_2	0	2	0	0

For row 4: replace h1 value for S3, don't replace h2 value since current value is lower
Note: result is same as permutations to find first 1