Philipps Universität Marburg

Summer 2023

Data Integration

Thorsten Papenbrock

Duplicates

Duplicate: Two representations (e.g. records, objects, XML structures) that represent the same real-world entity or concept

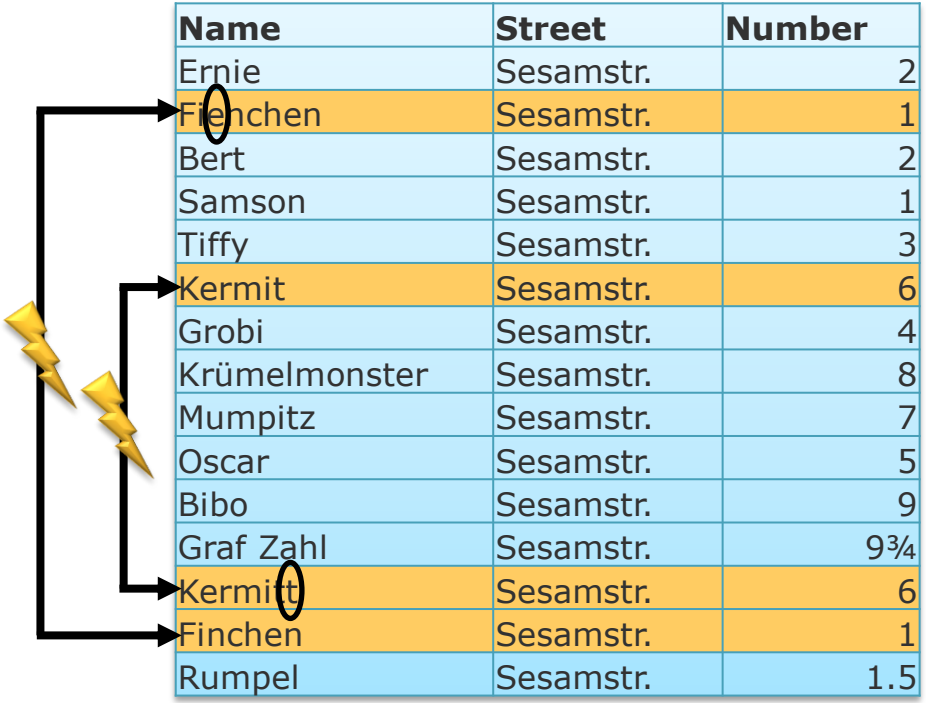
- Example:

Name	Street	Number
Ernie	Sesamstr.	2
Fienchen	Sesamstr.	1
Bert	Sesamstr.	2
Samson	Sesamstr.	1
Tiffy	Sesamstr.	3
Kermit	Sesamstr.	6
Grobi	Sesamstr.	4
Krümelmonster	Sesamstr.	8
Mumpitz	Sesamstr.	7
Oscar	Sesamstr.	5
Bibo	Sesamstr.	9
Graf Zahl	Sesamstr.	9¾
Kermitt	Sesamstr.	6
Finchen	Sesamstr.	1
Rumpel	Sesamstr.	1.5

Duplicates

Duplicate: Two representations (e.g. records, objects, XML structures) that represent the same real-world entity or concept

- Example:



Name	Street	Number
Ernie	Sesamstr.	2
Fienchen	Sesamstr.	1
Bert	Sesamstr.	2
Samson	Sesamstr.	1
Tiffy	Sesamstr.	3
Kermit	Sesamstr.	6
Grobi	Sesamstr.	4
Krümelmonster	Sesamstr.	8
Mumpitz	Sesamstr.	7
Oscar	Sesamstr.	5
Bibo	Sesamstr.	9
Graf Zahl	Sesamstr.	9¾
Kermitt	Sesamstr.	6
Finchen	Sesamstr.	1
Rumpel	Sesamstr.	1.5

Duplicates

Duplicate: Two representations (e.g. records, objects, XML structures) that represent the same real-world entity or concept

- Typically not equal but similar (w.r.t. values, attribute sets, references)
- Example:

- Origin examples:
  - Data integration: data silos, data sharing, data discovery
  - Errors in data entry: typos, redundancies, transmission
  - Fraudulent actions: manipulation, human mistakes

Duplicates

Duplicate: Two representations (e.g. records, objects, XML structures) that represent the same real-world entity or concept

- Typically not equal but similar (w.r.t. values, attribute sets, references)
- Effects:
  - Wrong decision making
    - Actions are carried out multiple times per entity
  - Inaccurate statistics
    - Number of entities is counted to high
  - Poorly trained machine learning models
    - Duplicate entries introduce bias and misclassifications
  - Software failures
    - Unique constraints may be violated
  - …

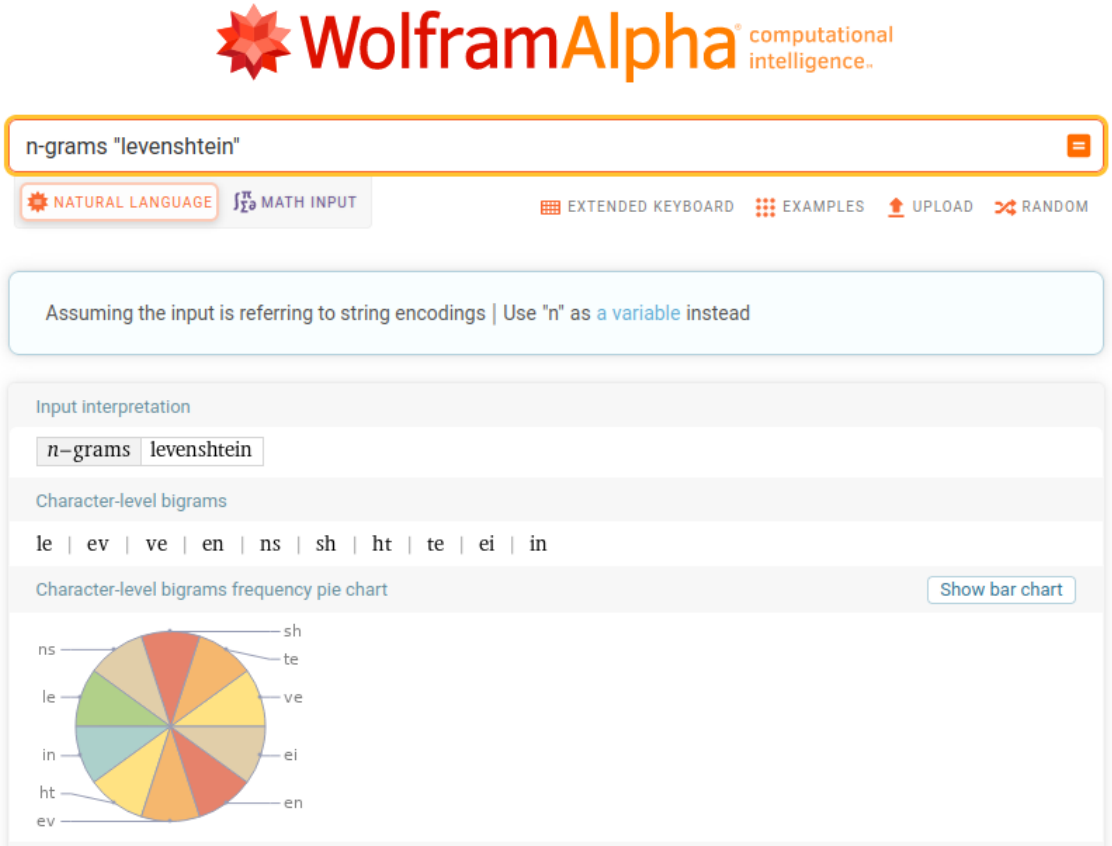
All this costs a lot of money in practice!

Token-based Similarity: n-grams

- The function $to^k(x)$ splits the string *x* into short substrings of length n by sliding a window of size n over *x*; every slide creates an n-gram.
  - n=2: Bigrams; n=3: Trigrams
  - Number of n-grams = |*x*| – n + 1
- Variation 1: Pad with n – 1 special characters
  - ➢ Emphasizes beginning and end of string
- Variation 2: Include positional information
  - ➢ Useful to weight tokens by their positions

String	Bigrams	Padded bigrams	Positional bigrams	Trigrams
gail	ga, ai, il	⊙g, ga, ai, il, l⊗	(ga,1), (ai,2), (il,3)	gai, ail
gayle	ga, ay, yl, le	⊙g, ga, ay, yl, le, e⊗	(ga,1), (ay,2), (yl,3), (le,4)	gay, ayl, yle
peter	pe, et, te, er	⊙p, pe, et, te, er, r⊗	(pe,1), (et,2), (te,3), (er,4)	pet, ete, ter
pedro	pe, ed, dr, ro	⊙p, pe, ed, dr, ro, o⊗	(pe,1), (ed,2), (dr,3), (ro,4)	ped, edr, dro

Token-based Similarity: n-grams



Levenshtein Distance

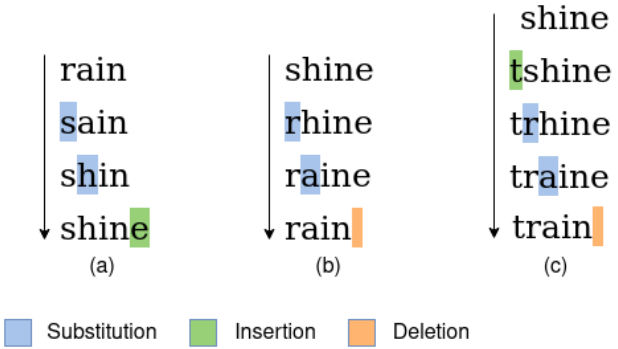
This is the challenge!

Definition:

- $dist_{levenshtein}(x,y)$ = minimum number of edits (insert, delete, replace) that transform the string x into string y
- The most popular metric for describing the edit-distance of strings

Operations:

1. insert a character into the string
2. delete a character from the string
3. replace (substitute) a character with a different character



Examples:

- $dist_{levenshtein}$('table', 'cable') = 1
  (1 replace)
- $dist_{levenshtein}$('Thorsten Papenbrock', ' Papenbrock, Thorsten') = 19
  (9 deletes + 10 inserts)

Levenshtein Distance

Calculation:

- Calculating the minimum edit-distance is a case for dynamic programming
- Optimality principle:
  - Any minimum edit-distance of two substrings must be part of the best overall solution.
- Dynamic programming algorithm:
  1. Initialize a matrix M of size (|x|+1) x (|y|+1)
  2. Fill matrix: $M_{i,0} = i$ and $M_{0,j} = j$
  3. Recursion: $M_{i,j} = \begin{cases} M_{i-1,j-1} & if\ x[i] = y[j] \\ 1 + \min(M_{i-1,j}, M_{i,j-1}, M_{i-1,j-1}) & else \end{cases}$
  4. Distance: $dist_{levens}(x, y) = M_{|x|,|y|}$

Levenshtein Distance

Calculation example:

		J	O	N	E	S
	0	1	2	3	4	5
J	1					
O	2					
H	3					
N	4					
S	5					
O	6					
N	7					

		J	O	N	E	S
	0	1	2	3	4	5
J	1	0	1	2	3	4
O	2					
H	3					
N	4					
S	5					
O	6					
N	7					

		J	O	N	E	S
	0	1	2	3	4	5
J	1	0	1	2	3	4
O	2	1	0	1	2	3
H	3	2	1	1	2	3
N	4	3	2	1	2	3
S	5	4	3	2	2	2
O	6	5	4	3	3	3
N	7	6	5	4	4	4

Every path leads to the same optimal solution!

$M_{i,0} = i$ and $M_{0,j} = j$

$$M_{i,j} = \begin{cases} M_{i-1,j-1} & if\ x[i] = y[j] \\ 1 + \min\left(M_{i-1,j}, M_{i,j-1}, M_{i-1,j-1}\right) & else \end{cases}$$

Levenshtein Distance

Similarity:

- $sim_{levenshtein}(x,y) = 1 - dist_{levenshtein}(x,y) / max(|x|,|y|)$

x	y	Lev. Distance	Lev. Similarity
Jones	Johnson	4	0.43
Paul	Pual	2	0.5
Paul Jones	Jones, Paul	11	0

Discussion:

- Robust against different forms and positions of typos
- Easy to understand and implement
- Unsupervised similarity metric
- Sensitive to word order changes
- Expensive to calculate

Levenshtein Distance

Complexity:

- Time: $O(|x| \cdot |y|)$ (calculate entire matrix)
- Space: $O(\min(|x|, |y|))$ (store only two rows of the matrix)

Properties:

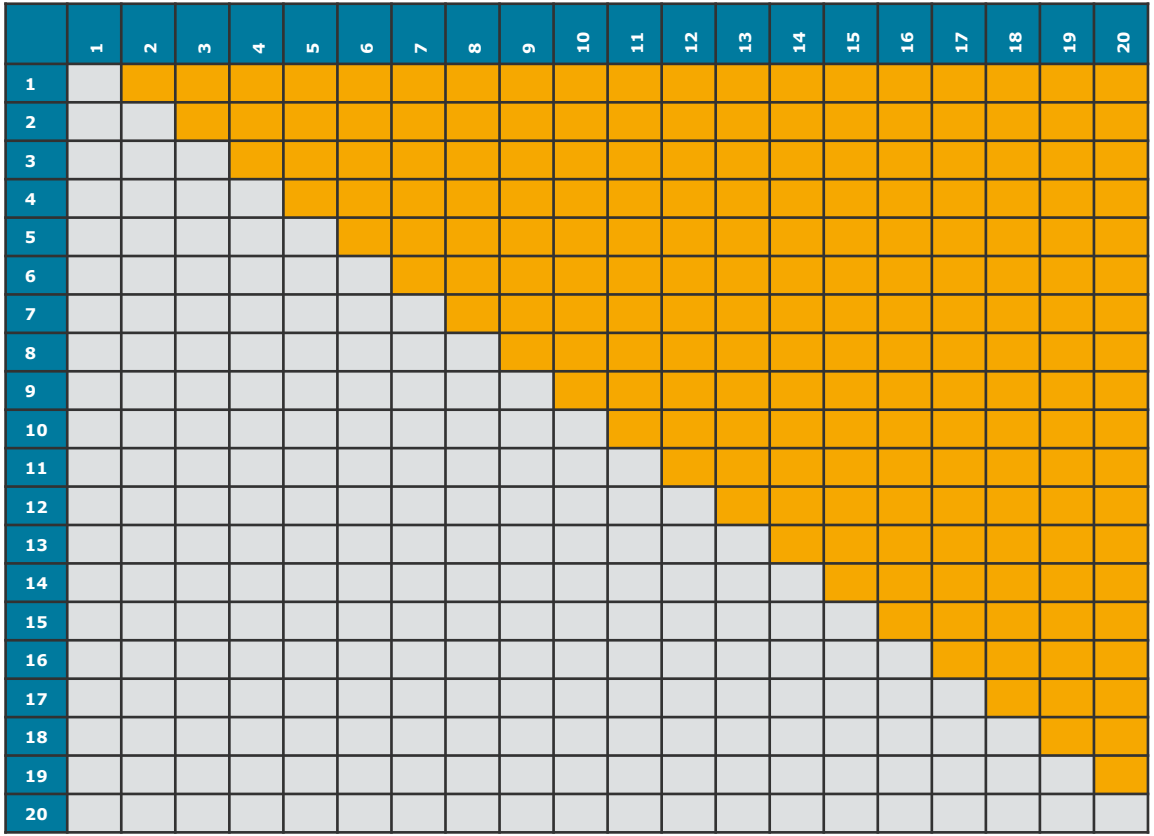
- $0 \leq dist_{levenshtein}(x,y) \leq max(|x|,|y|)$
- $||x|-|y|| \leq dist_{levenshtein}(x,y)$

Good lower-bound estimate to possibly skip the exact distance computation!

Cost models (extension): Assign different costs to edit-operations.

- By operation:
  - E.g. replace costs 0.5 but insert/delete cost 1.0 to punish string length changes.
- By character:
  - E.g. OCR (m ≃ n, 1 ≃ l) or keyboard (a ≃ s) or brain (6 ≃ 9) or biology (a ≃ t)

Sorted Neighborhood Method (SNM)

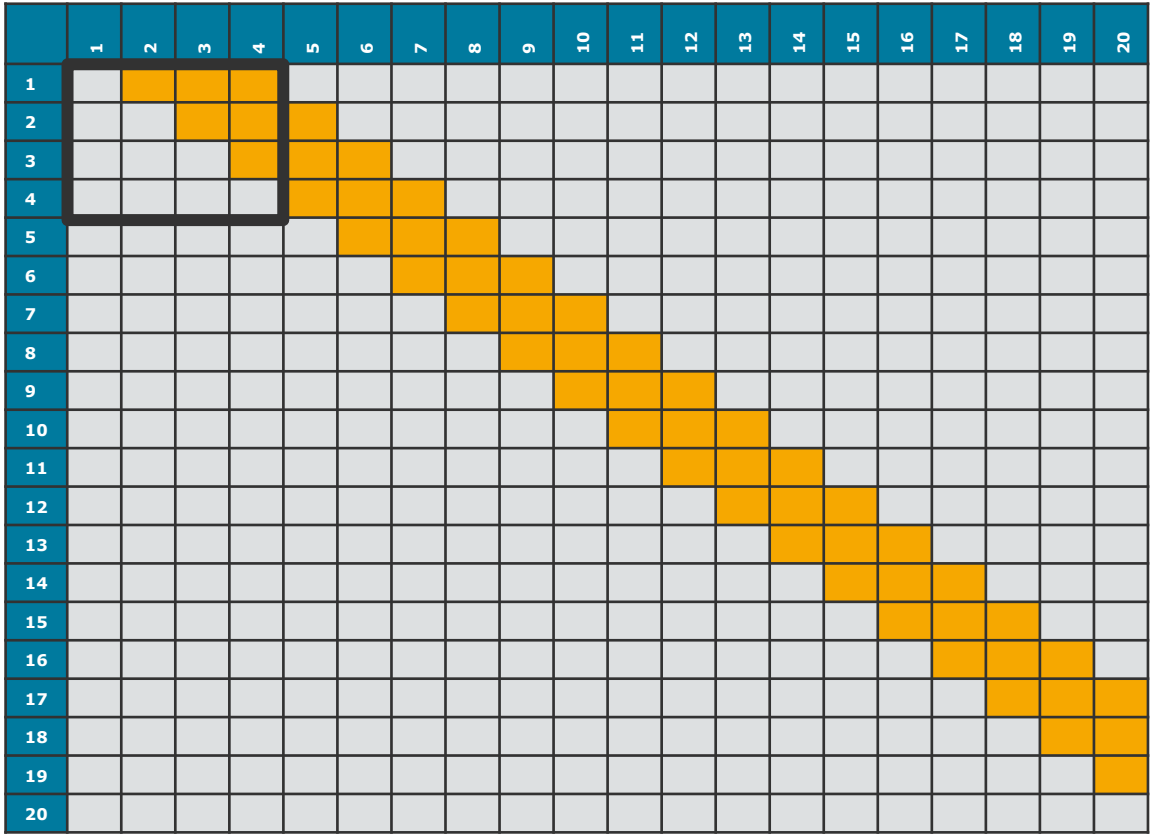


Sorted Neighborhood Method

- Sort tuples so that similar tuples are close to each other.
- Compare tuples only within a small neighborhood (= window).

Sorted Neighborhood Method (SNM)

M. A. Hernández, S. J. Stolfo
Real-world Data is Dirty: Data Cleansing and The Merge/Purge Problem. 1998

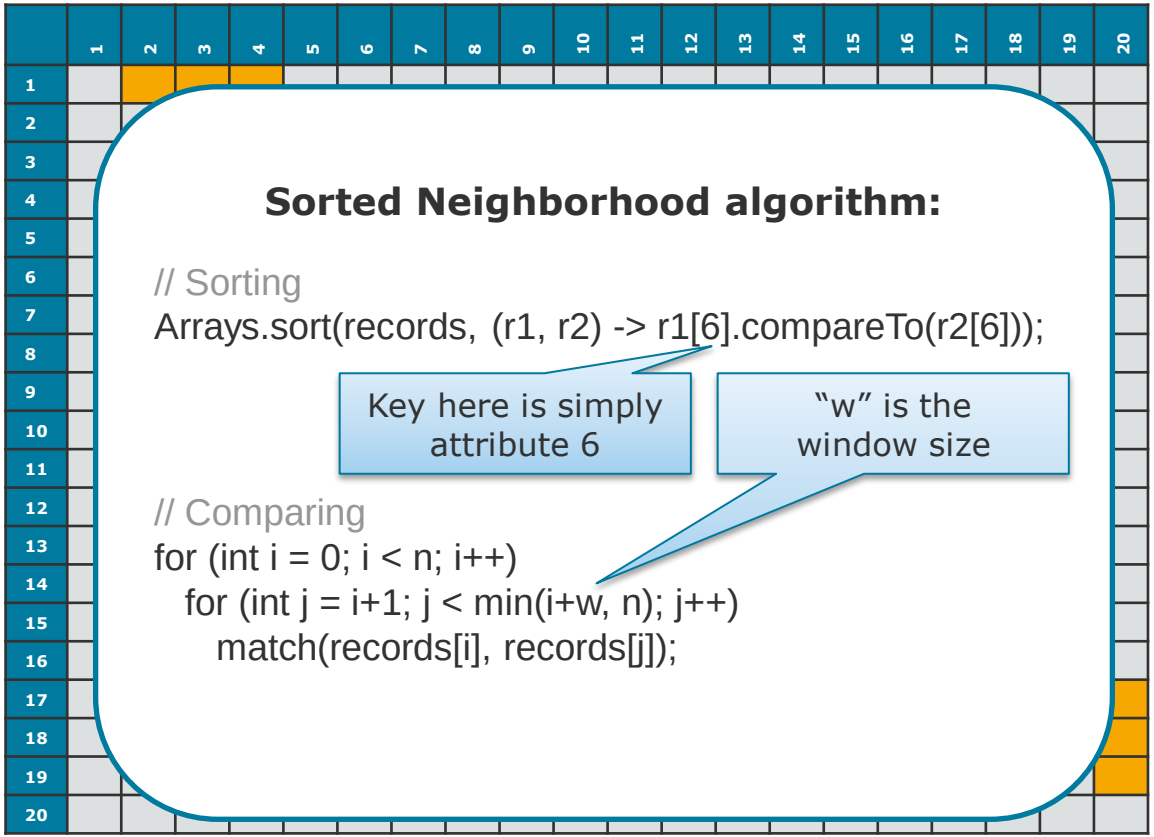


Sorted Neighborhood Method

- Sort tuples so that similar tuples are close to each other.
- Compare tuples only within a small neighborhood (= window).

1. Generate a key
   - E.g. SSN
   - E.g. Name[1-3] + Age + …
2. Sort entire relation by the key
   - Sort records physically or create a sorted index.
3. Slide a window over sorted tuples
   - Compare all pairs of tuples within the sliding window.

Sorted Neighborhood Method (SNM)

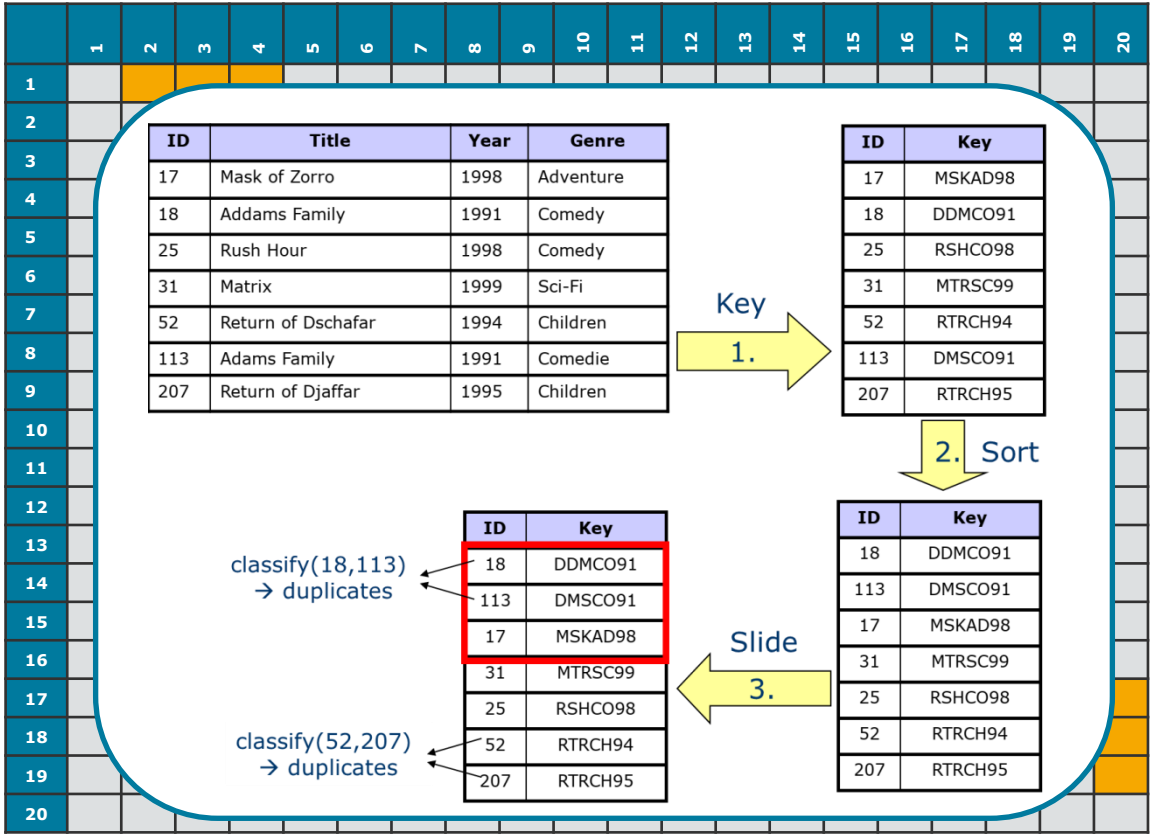


Sorted Neighborhood Method

- Sort tuples so that similar tuples are close to each other.
- Compare tuples only within a small neighborhood (= window).

1. Generate a key
   - E.g. SSN
   - E.g. Name[1-3] + Age + …
2. Sort entire relation by the key
   - Sort records physically or create a sorted index.
3. Slide a window over sorted tuples
   - Compare all pairs of tuples within the sliding window.

Sorted Neighborhood Method (SNM)

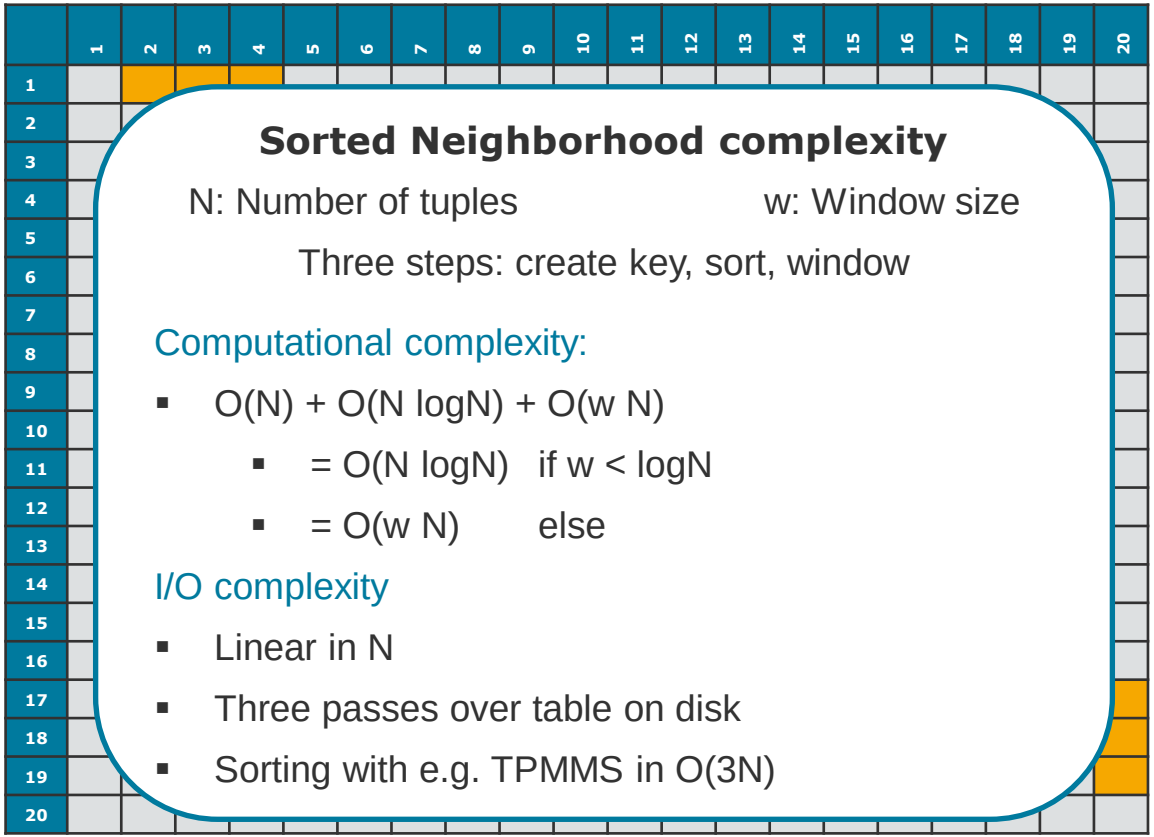


Sorted Neighborhood Method

- Sort tuples so that similar tuples are close to each other.
- Compare tuples only within a small neighborhood (= window).

1. Generate a key
   - E.g. SSN
   - E.g. Name[1-3] + Age + …
2. Sort entire relation by the key
   - Sort records physically or create a sorted index.
3. Slide a window over sorted tuples
   - Compare all pairs of tuples within the sliding window.

Sorted Neighborhood Method (SNM)



Sorted Neighborhood complexity

N: Number of tuples w: Window size

Three steps: create key, sort, window

Computational complexity:

- O(N) + O(N logN) + O(w N)
  - = O(N logN) if w < logN
  - = O(w N) else

I/O complexity

- Linear in N
- Three passes over table on disk
- Sorting with e.g. TPMMS in O(3N)

Sorted Neighborhood Method

- Sort tuples so that similar tuples are close to each other.
- Compare tuples only within a small neighborhood (= window).

1. Generate a key
   - E.g. SSN
   - E.g. Name[1-3] + Age + …
2. Sort entire relation by the key
   - Sort records physically or create a sorted index.
3. Slide a window over sorted tuples
   - Compare all pairs of tuples within the sliding window.

Locality-Sensitive Hashing (LSH)

- Given: Two **large** sets of values X and Y
  (e.g. two columns in a relational table;
  or token sets if X and Y are strings)
- Task: Calculate the similarity of the two sets

In particular, token-based and hybrid algorithms!

- Before: Similarity functions with a quadratic number of value comparisons
- Now: A technique that avoids comparing all values!

- Idea:
  - Hash the value sets using different locality-sensitive hash functions.

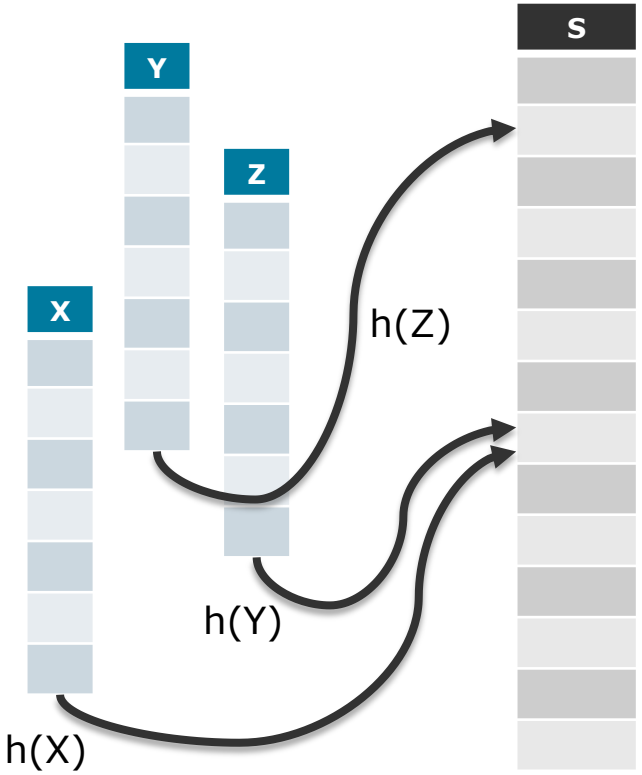
Hash collisions should be maximized (not minimized as in conventional hashing)!

    - ➢ Hash functions with the property that they hash similar values (or value sets) into same buckets with high probability.
  - Compare X and Y based on the buckets they have been hashed into.

Locality-Sensitive Hashing (LSH)

Locality-sensitive hash function

- Tries to hash similar value sets to same buckets.
- Value sets in the same bucket are candidates for matches.
- Use cases:
  - Similarity search
  - Clustering
  - Finger printing
  - Blocking
  - …



Locality-Sensitive Hashing (LSH)

Locality-sensitive hash function

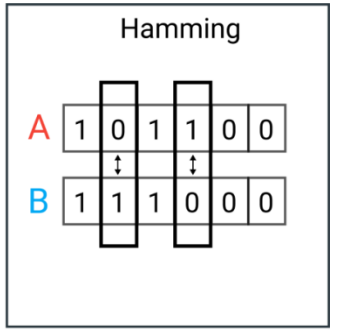
- Based on a metric space $\mathcal{M}$ = (M, d)
  - M: set of values x (or value sets X)
  - d: metric function on M (e.g. Euclidean, Levenshtein, Hamming, …)
  - Properties (see distance measures):
    - Reflexivity d(*x,x*) = 0
    - Symmetry d(*x,y*) = d(*y,x*)
    - Triangular inequation d(*x,z*) ≤ d(*x,y*) + d(*y,z*)
- Specifies a hash function h: M → S that maps points x ∈ M from $\mathcal{M}$ to a bucket s ∈ S such that for all x,y,z ∈ M:
  - If d(x,y) < d(x,z) then Pr(h(x) = h(y)) > Pr(h(x) = h(z))
  - ➢ The concrete hash function depends on the metric space $\mathcal{M}$!

Locality-Sensitive Hashing (LSH)

Locality-sensitive hash function

$$dist_{ham}(o,o') = \sum_{i=1..n} 1(c_i \neq c'_i)$$

- Example: Hamming (Bit sampling)
  - Hash every value x = $\{0,1\}^d \in M$ to the bit at position i:
    - $h(x) = x_i$ for some $i \in \{1,\ldots,d\}$

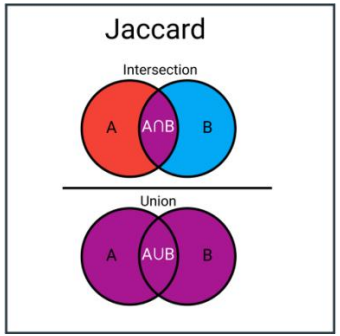


- Example: Jaccard (MinHash)

For sets X: tok(x) = X

$$sim_{jaccard}(x,y) = \frac{|tok(x) \cap tok(y)|}{|tok(x) \cup tok(y)|}$$

  - Every value x is a set of tokens tok(x)
  - Assume $\pi(tok(x))$ is some sortation function for tok(x) (e.g. w.r.t. a random permutation of the entire token universe)
  - Hash every value x to the smallest token of x:
    - $h(x) = \min\{\pi(tok(x))\}$ for some π

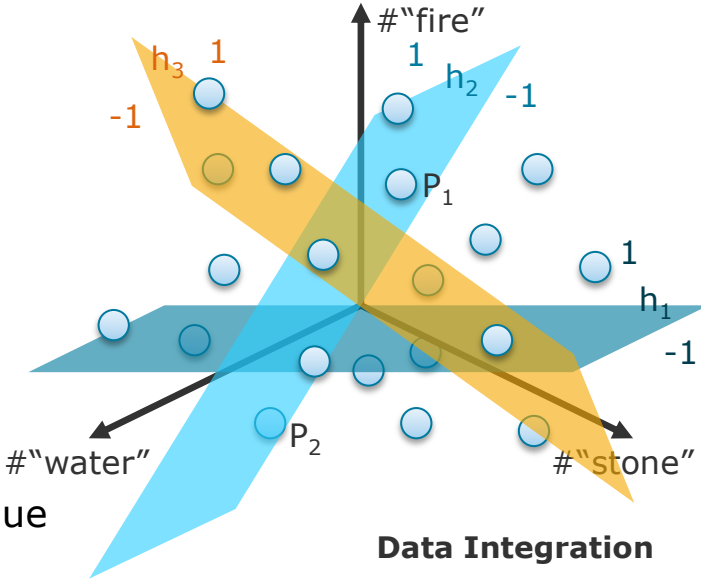


- Both fulfill: If d(x,y) < d(x,z) then Pr(h(x) = h(y)) > Pr(h(x) = h(z))

Locality-Sensitive Hashing (LSH)

Locality-sensitive hash function

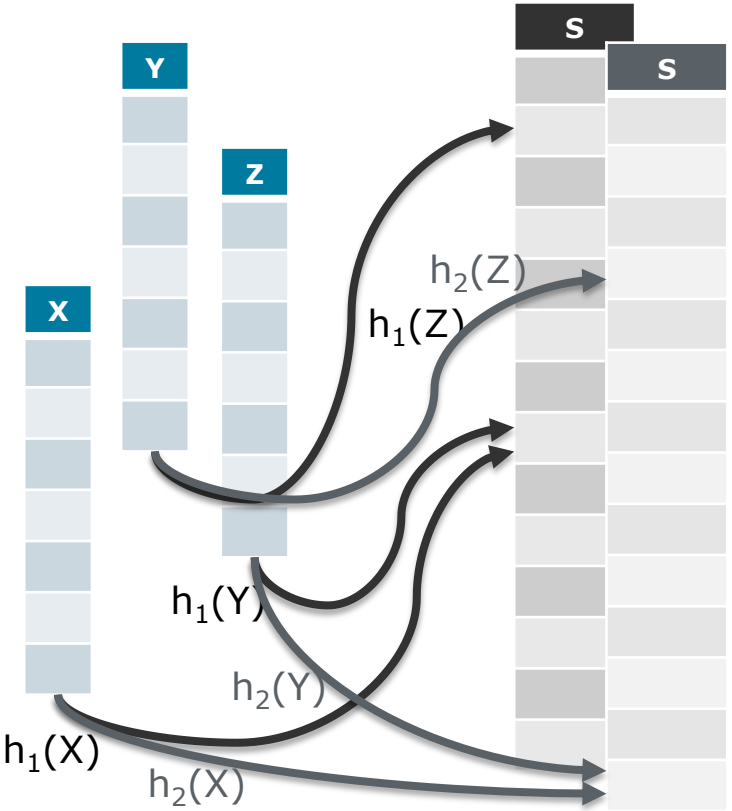
- Example: Embeddings (Space cutting)
  - Every value X is a set of tokens, words or strings that we can map into a vector space
  - Example:
    - each element X is a relational column of strings
    - each dimension of the space is a string from the entire string universe
    - the coordinate of X in each dimension is the value count of that dimension's string in X
  - Generate random hyperplanes (h_1, h_2, h_3)
  - Hash every value X by the sector it falls into:
    - $h(X) = [\text{sign}(X^T h_1), \text{sign}(X^T h_2), \text{sign}(X^T h_3)]$
    - Example: $h(P_1) = [1,-1,1]$ and $h(P_2) = [-1,1,-1]$



Locality-Sensitive Hashing (LSH)

LSH family

- Hashing is approximate:
  - Hash collisions lead to false positives.
  - Small differences with (even just a little) hash impact lead to false negatives.
- ➢ One function alone cannot capture the similarity of large sets sufficiently well.
- Solution:
  - Use multiple functions!



Locality-Sensitive Hashing (LSH)

LSH family

- An LSH family is a set of hash functions H in a metric space $\mathcal{M}$ that fulfills the property $Pr_{h \in H}(h(X) = h(Y)) = sim(X,Y)$.
  - Interpretation: "The probability that any h ∈ H hashes X and Y into the same bucket is equal to their similarity"
  - H is formed by e.g. using many random indexes i for $\mathcal{M}_{\text{Hamming}}$ or using many random sortations π for $\mathcal{M}_{\text{Jaccard}}$
- Calculating $H(X) = [h_1(X), \ldots, h_k(X)]$ generates a (e.g. MinHash) signature for the value set X, which is a vector hash that preserves the similarity properties of X w.r.t. $\mathcal{M}$.
- The hash functions (and their signature) work on tokens of a string x but also on the values of a string set X (i.e. columns)!
  - Especially on columns, the signature is much shorter than the input!
    - ➢ Similarity calculations on H(X) are more efficient than on X!

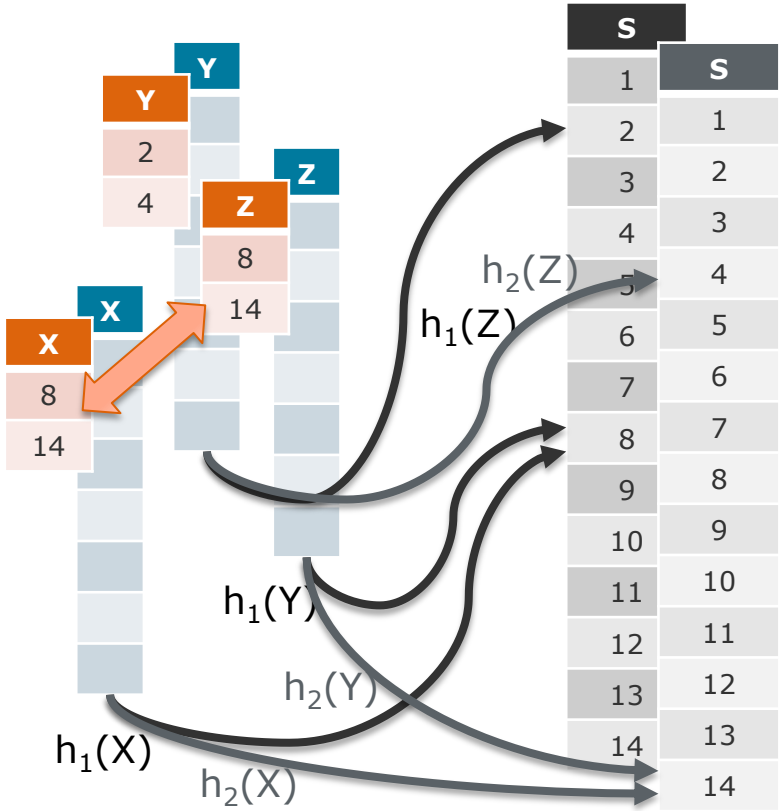
Locality-Sensitive Hashing (LSH)

LSH family

- The buckets of a value set X generate signatures.

- Signature use:
  - a) Blocking:
    Compare all X and Y that share at least one (or a certain amount of) buckets with one of the expensive measures.
  - b) Hashing:
    Compare X and Y by their bucket signature.

now



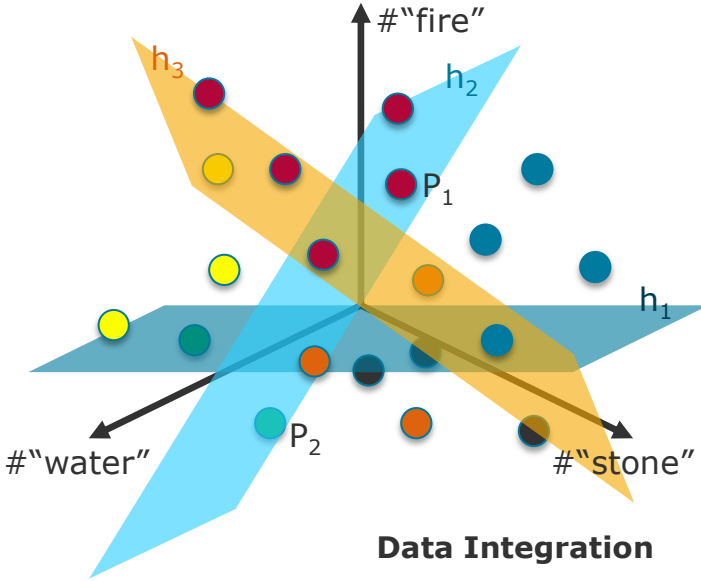
Locality-Sensitive Hashing (LSH)

LSH family and blocking with embeddings

- Example LSH hashes with 3 hyperplanes:
  - $h_1(P_1)$ = [1,-1,1] and $h_1(P_2)$ = [-1,1,-1]

- Example signatures with 4 LSH functions (= 4 randomizations of the 3 hyperplanes):
  - $H(P_1)$ = ([1,-1,1], [1,1,1], [1,-1,-1], [1,-1,1])
  - $H(P_2)$ = ([-1,1,-1], [-1,1,1], [1,-1,1], [1,-1,-1])

- Compare P_1 and P_2 only if $h_i(P_1) = h_i(P_2)$ for any i.
  - E.g. with an expensive $sim_{MongeElkan}(P_1,P_2)$

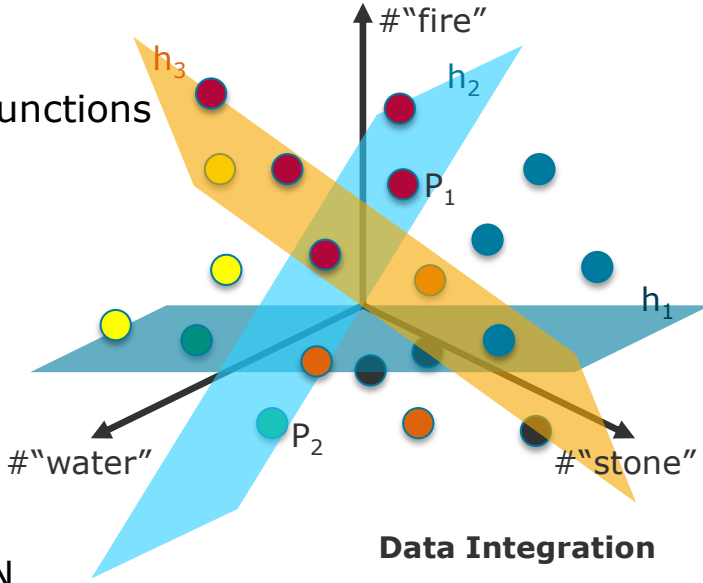


Locality-Sensitive Hashing (LSH)

LSH family and blocking with embeddings

- Given: N points, P hyperplanes, S similarity cost, K LSH functions
- Computational costs (naïve):
  - N*N*S
    - O(N²)
- Computational costs (blocking via LSH):
  - $N*N/2^P*S*K$ on average $N/2^P$ points per bucket
    - O(N*log(N)) if we increase P proportionally to N
- Computational costs (indexing)
  - $N*N/\sqrt{N}*S$ on average $N/\sqrt{N}$ points per leaf
    - $O(N*\sqrt{N})$

Not considering the index lookup costs for LSH and indexing!



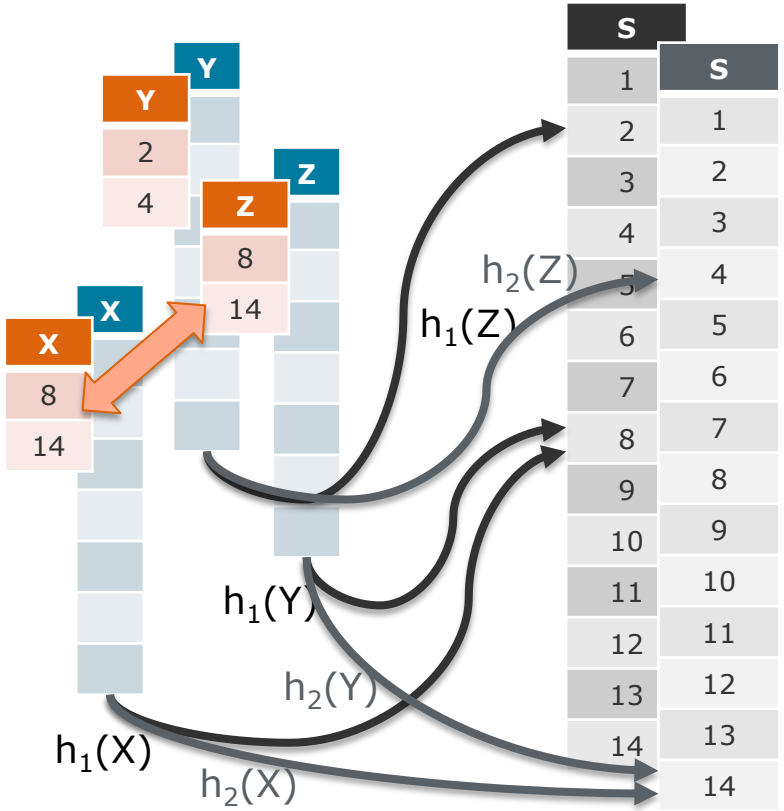
Locality-Sensitive Hashing (LSH)

LSH family

- The buckets of a value set X generate signatures.

- Signature use:
  a) Blocking:
  Compare all X and Y that share at least one (or a certain amount of) buckets with one of the expensive measures.
  b) Hashing:
  Compare X and Y by their bucket signature.

now



Locality-Sensitive Hashing (LSH)

LSH family and token-based similarity (Jaccard)

- Task: Calculate $sim_{Jaccard}(X,Y)$

$$sim_{jaccard}(x,y) = \frac{|tok(x) \cap tok(y)|}{|tok(x) \cup tok(y)|}$$

1. Calculate the signatures with k MinHash LSH functions:
   - $H(X) = [h_1(X), \ldots, h_k(X)]$
   - $H(Y) = [h_1(Y), \ldots, h_k(Y)]$
2. Calculate the number y of matches from the MinHash signatures:
   - $y = \sum_{i=1}^{k} (hi(X) == h$
3. Calculate an approximation of the Jaccard similarity:
   - $sim_{Jaccard}(X,Y) = Pr_{h \in H}(h(X) = h(Y)) \approx y/k$

To calculate this efficiently, do not sort columns multiple time but simply hash them to new positions.

Locality-Sensitive Hashing (LSH)

LSH family and hybrid-based similarity (Monge-Elkan)

- Task: Calculate $sim_{MongeElkan}(X,Y)$

$$sim_{MongeElkan}(x,y) = \frac{1}{|x|}\sum_{i=1}^{|x|} \max_{j=1,|y|} sim'(x[i],y[j])$$

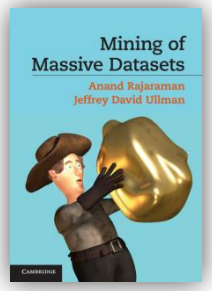
1. Calculate the signatures with k MinHash LSH functions:
   - $H(X) = [h_1(X), \ldots, h_k(X)]$
   - $H(Y) = [h_1(Y), \ldots, h_k(Y)]$
2. Calculate an approximation of the Monge-Elkan similarity:
   - $sim_{MongeElkan}(X,Y) \approx sim_{MongeElkan}(H(X), H(Y))$

Now quadratic in the number of hash functions and not quadratic in the number of values!

Locality-Sensitive Hashing (LSH)

Discussion

- Many further LSH approaches exist!
  - With alternative LSH functions
  - With alternative signature comparisons
  - With alternative compositions
    - See e.g.:



- LSH is, in general, very popular for matching value sets, documents, relational attributes, and complex entities/records