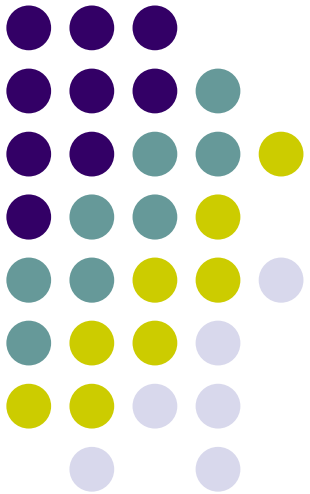
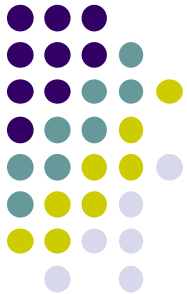
Nearest-Neighbor Search (Big Data)

J. Savoy
Université de Neuchâtel



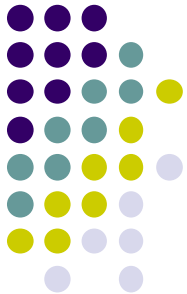
A. Rajaraman, J.D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2012.
H. Garcia-Molina, J.D. Ullman, J. Widom: *Database Systems The Complete Book*. Pearson, Upper Sade River, 2009.

Contents



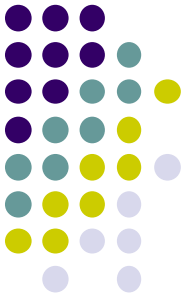
- **Applications**
- Method
- Minhashing
  - Data as Sparse Matrix
  - Jaccard Similarity Measure
  - Constructing Signature
  - Locality Sensitive Hashing (LSH)

Problem



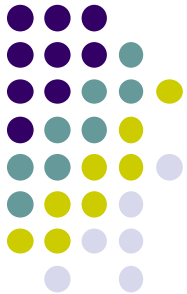
- The nearest neighbor search (NNS) problem is: given a set of *n* points $P = \{p_1, p_2, \ldots, p_n\}$ in a metric space X with distance function *d*, preprocess *P* so as to efficiently answer queries for finding the point in *P* closest to a query point $q \in X$.
- Interesting are *d*-dimensional Euclidian space where $X = R^d$
- Nearest Neighbor Search can be used in a variety of applications

Example: Face Recognition



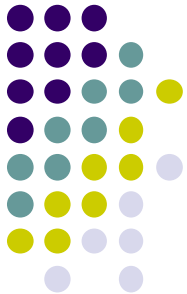
- We have a database of (say) 1 million face images.
- We are given a new image and want to find the most similar images in the database.
- Represent faces by (relatively) invariant values, e.g., ratio of nose width to eye width.
- Each image represented by a large number (say 1,000) of numerical features.
  (usually too large for *k*D-tree)
- Problem: given the features of a new face, find those in the DB that are close in at least ¾ (say) of the features.

Face Recognition

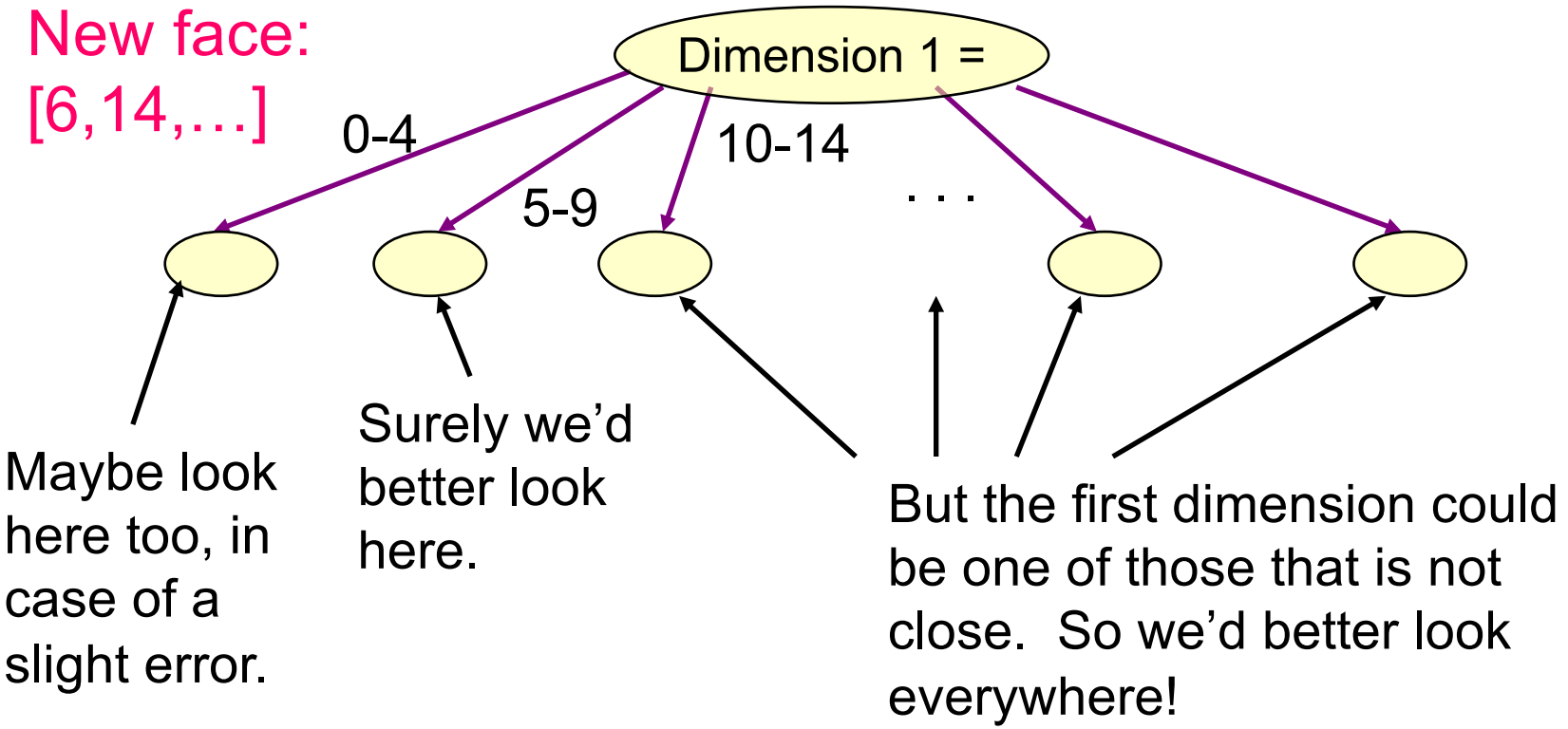


- *Many-one problem* : given a new face, verify if it is close to any of the 1 million previous faces.
- *Many-Many problem* : which pairs of the 1 million faces are similar.
- Represent each face by a vector of 1,000 values and score the comparisons.
- Out of the question for the many-many problem ($10^6 \cdot 10^6 \cdot 1{,}000$ numerical comparisons).
- We can do *better*!

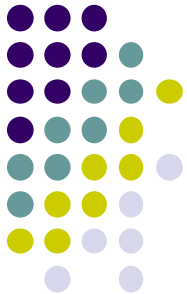
Multidimensional Indexes Don't Work



New face: [6,14,…]

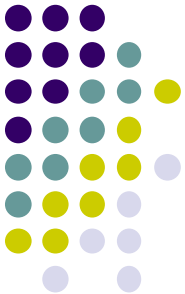


Problem: Entity Resolution



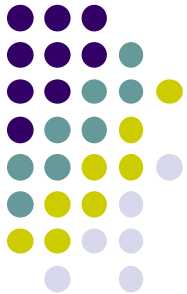
- Two sets of 1 million name-address-phone records.
- Some pairs, one from each set, represent the same person.
- Errors of many kinds:
  - Typos, missing middle initial, area-code changes, St./Street, Bob/Robert, etc.
- Choose a scoring system for how close names are.
  - Deduct so much for edit distance > 0;
    so much for missing middle initial, etc.
- Similarly score differences in addresses, phone numbers.
- Sufficiently high total score $\rightarrow$ records represent the same entity.

Simple Solution



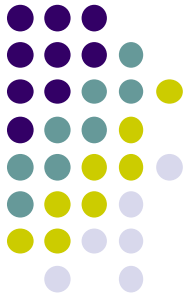
- Compare each pair of records, one from each set.
- Score the pair.
- Call them the same if the score is sufficiently high.
- We have an algorithm, but do we have a solution?
- But unfeasible for 1 million records.
- We need / can do better!

Another Problem: Finding Similar Documents



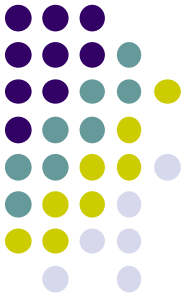
- Given a body of documents, e.g., the Web, find pairs of docs that have a lot of text in common.
- Find mirror sites, approximate mirrors, plagiarism, quotation of one document in another, “good” document with random spam, etc.
- The face problem had a way of representing a big image by a (relatively) small data-set.
- Entity records represent themselves.
- How do you represent a document so it is easy to compare with others?

Contents



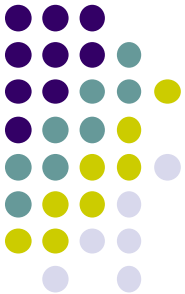
- Applications
- **Method**
- Minhashing
  - Data as Sparse Matrix
  - Jaccard Similarity Measure
  - Constructing Signature
  - Locality Sensitive Hashing (LSH)

Complexity



- Special cases are easy
  e.g., identical documents
  or one document contained verbatim in another.
- General case, where many small pieces of one document appear out of order in another, is harder (plagiarism detection)
- This is the real world of Big Data
  (NoSQL Not Only SQL)

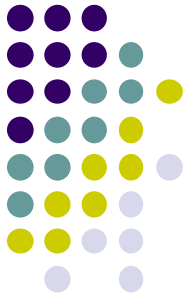
Representing Objects for Similarity Search



1. Represent object by its set of shingles (or *n*-grams for document).
2. Summarize shingle set by a *signature*
   = small data-set with the property:

- Similar documents are very likely to have "similar" signatures (but *no guarantee*).
- At that point, doc problem resembles the previous two problems.

Shingles / Features

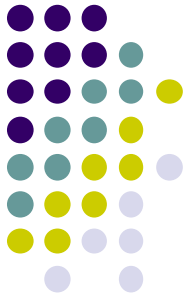


- A *k*-shingles (or *k*-gram) for a document is a sequence of *k* characters that appears in the document.

- Example:
  k = 2; document = "abcab bc".
  Set of 2-gram = {"ab", "bc", "ca", " b"}.
  - Option: regard features as a bag, and count "ab" twice.
  - In this example, we have used overlapping 2-grams

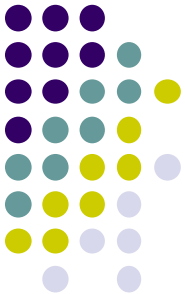
A.Z. Broder: On the resemblance and containment of documents. IEEE, 1998, pp. 21-29.

Shingles: Aside



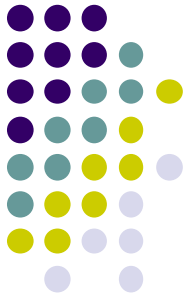
- Although we shall not discuss it, shingles are a *powerful* tool for characterizing the topic of documents.
  - k = 5 is the right number;
    (#characters)^5 >> #shingles in typical document.
- Example: “US pr” and “white” are most common in news articles.
- Effective indexing strategy for the Japanese, Chinese and Korean languages (e.g., 2-gram).
- But we can expect having more shingles than distinct words.

Shingles: Compression Option



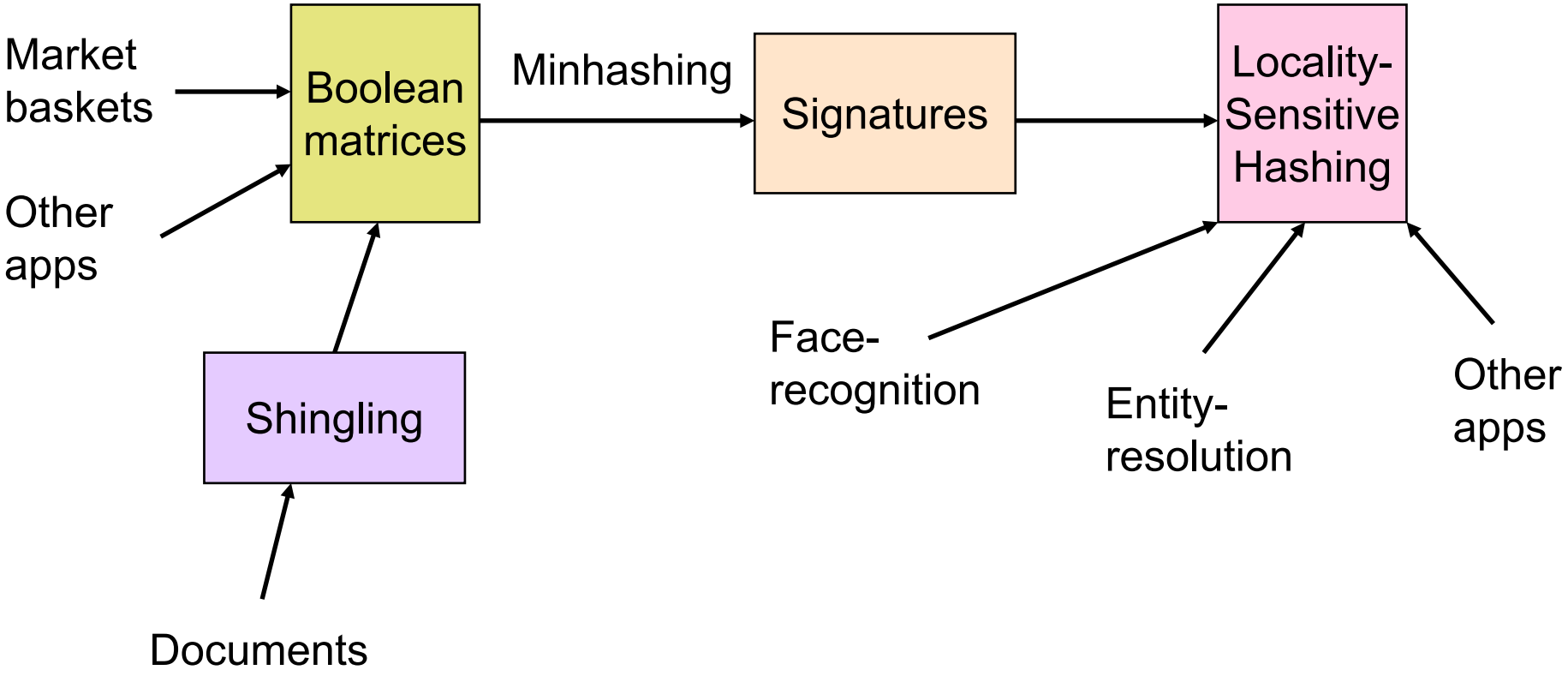
- To compress long shingles, we can hash them to (say) 4 bytes.
- Represent a document by the set of hash values of its *k*-shingles.
- Two documents could (rarely) appear to have shingles in common, when in fact only the hash-values were shared.

Contents

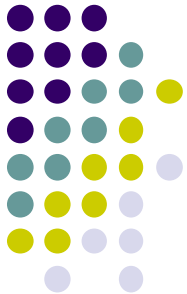


- Applications
- Method
- **Minhashing**
  - Data as Sparse Matrix
  - Jaccard Similarity Measure
  - Constructing Signature
  - Locality Sensitive Hashing (LSH)

Roadmap

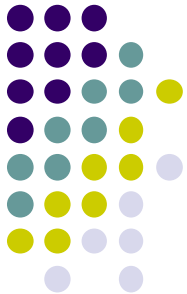

Market baskets
Other apps
Boolean matrices
Minhashing
Signatures
Locality-Sensitive Hashing
Shingling
Documents
Face-recognition
Entity-resolution
Other apps

Contents



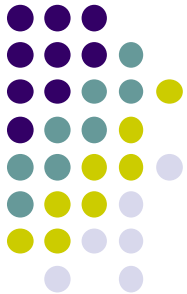
- Applications
- Method
- Minhashing
  - **Data as Sparse Matrix**
  - Jaccard Similarity Measure
  - Constructing Signature
  - Locality Sensitive Hashing (LSH)

Boolean Matrix Representation



- Data in the form of subsets of a universal set can be represented by a (typically sparse) matrix.
- Examples include:
    1. Documents represented by their set of shingles (or hashes of those shingles).
    2. Market baskets.

Matrix Representation of Item/Basket Data

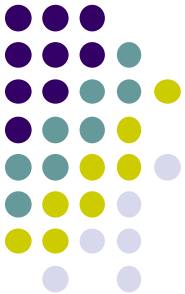


- Columns = items
- Rows = baskets
- Entry (*r* , *c*) = 1 if item *c* is in basket *r*
  = 0 if not.
- Typically matrix is almost all 0's.

In Matrix Form

	m	c	p	b	j
{m, c, b}	1	1	0	1	0
{m, p, b}	1	0	1	1	0
{m, b}	1	0	0	1	0
{c, j}	0	1	0	0	1
{m, p, j}	1	0	1	0	1
{m, c, b, j}	1	1	0	1	1
{c, b, j}	0	1	0	1	1
{c, b}	0	1	0	1	0

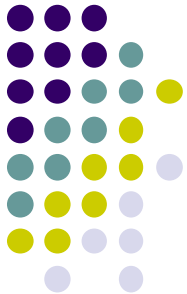
Documents in Matrix Form



- Columns = documents.
- Rows = shingles (or hashes of shingles).
- 1 in row *r*, column *c* iff document *c* has shingle *r*.
- Again expect the matrix to be sparse.

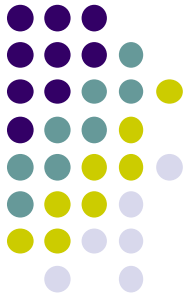
- We want to compare the columns (items / documents) to find near similar / correlated items / documents.
- How can we compute the distance / similarly between two columns?

Contents



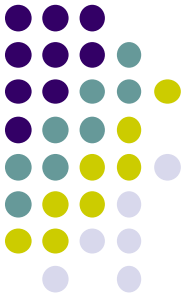
- Applications
- Method
- Minhashing
  - Data as Sparse Matrix
  - **Jaccard Similarity Measure**
  - Constructing Signature
  - Locality Sensitive Hashing (LSH)

Similarity of Columns



- Think of a column as the *set*
  but *consider only rows* in which 1 appears.
- The *similarity* of columns C_1 and C_2
  Sim(C_1,C_2) = is the ratio of the sizes of the intersection and union of C_1 and C_2.
  - *Jaccard measure*
    Sim(C_1,C_2) = $|C_1 \cap C_2| / |C_1 \cup C_2|$
  - Other measures are possible
    Dice: *Sim*(C_1,C_2) = $(2 \cdot |C_1 \cap C_2|) / (|C_1| + |C_2|)$
  - We can find correlated columns
    (similar items/documents)

Example



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

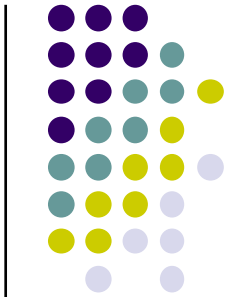
$Sim(C_1, C_2) =$

$2 / 5 = 0.4$

$Sim_{Dice}(C_1, C_2) =$

$(2 \cdot 2) / (3+4) = 4/7$

Example



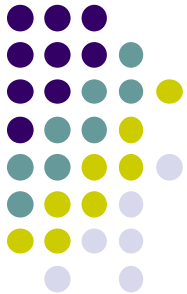
Input matrix

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

Similarities (Jaccard):

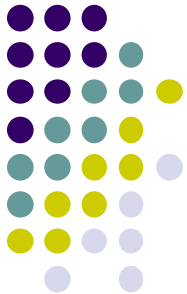
	1-3	2-4	1-2	3-4
Col/Col	0.75	0.75	0	0

Reduce the needed memory



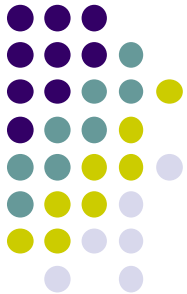
- We might not really represent the data by a Boolean matrix.
- Sparse matrices are usually better represented by the list of places where there is a non-zero value.
  - E.g., market baskets, shingle-sets.
  - Too large to fit in the main memory
    (#baskets x #items)
    (or difficult to access a given row / column)
- But the matrix picture is conceptually useful.
- We have the exact answer.

Contents



- Applications
- Method
- Minhashing
  - Data as Sparse Matrix
  - Jaccard Similarity Measure
  - **Constructing Signature**
  - Locality Sensitive Hashing (LSH)

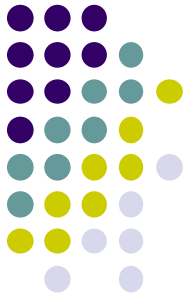
Assumptions



- To compare the columns (items / documents)

1. The huge number of items (columns) allows a small amount of main-memory / item.
   Thus we may have a constraint like
   main memory = number of items * 100
2. Too many items to store anything in main-memory for each *pair* of items.

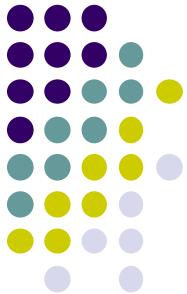
Outline of Algorithm



- We cannot store the Boolean matrix as it in the main memory. We need to reduce its size.
  How?

1. Compute signatures of columns = small summaries of columns.
   - Read from disk to main memory.
2. Examine signatures in main memory to find similar signatures.
   - Essential: similarities of signatures and columns are related.
3. Optional: check that columns with similar signatures are really similar.

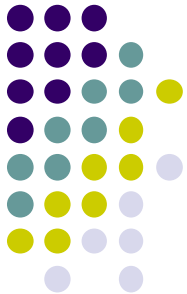
Warnings



1. Comparing all pairs of signatures may take too much time, even if not too much space
    - A job for Locality-Sensitive Hashing (LSH)

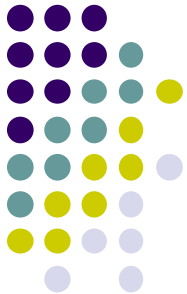
2. These methods can produce false positives (dissimilar items may produce similar signatures) if the optional check is not made

Signatures



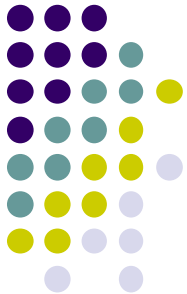
- Key idea: "hash" each column *C* to a small *signature* *Sig*(C), such that:
  1. *Sig*(C) is small enough that we can fit a signature in main memory for each column.
  2. *Sim*(C_1, C_2) is the same as the "similarity" or can be approximated by Sim_H[*Sig*(C_1), *Sig*(C_2)]

An Idea that Doesn't Work



- Use a random subset of the rows to define the signature. For example, pick 100 rows at random, and let the signature of column *C* be the 100 bits of *C* in those rows.
  (well-known in vote estimate)
- Because the matrix is sparse, many columns would have 00. . .0 as a signature, yet be very dissimilar because their 1's are in different rows and we would miss interesting part of the columns.

Four Types of Rows

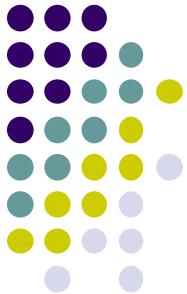


- Given columns C_i and C_j, rows may be classified as:

	C_i	C_j
a	1	1
b	1	0
c	0	1
d	0	0

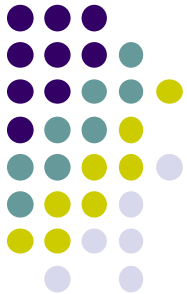
- We can find four types of rows
  Also, *a* = # rows of type *a* , etc.
- Important: note that $Sim(C_i, C_j) = a / (a+b+c)$

Minhashing



- Imagine that we want to permute the rows randomly (we do not want to physically permute the matrix)
- Example: With 5 = number of rows
  $P_1(x) = (x \bmod 5)+1$
  $P_2(x) = (2x + 1 \bmod 5)+1$
  $\rightarrow P_1(1) = 2, P_1(2) = 3, P_1(3) = 4, P_1(4) = 5, P_1(5) = 1$
  $\rightarrow P_2(1) = 4, P_2(2) = 1, P_2(3) = 3, P_2(4) = 5, P_2(5) = 2$
- We may consider index going from 0 to 4
- Define "hash" function $h(C)$ = the number of the first (in the permuted order) row in which column C has 1
- We can define more than one h() function …

Minhashing



Generate a random sequence of values

$x_{i+1} \equiv (a \cdot x_i + c) \bmod m$

If m = 10, x_0 = 7; c = 7; a = 7

we generate $\rightarrow$ 6, 9, 0, 7, 6, 9, 0, 7, …

Conditions:

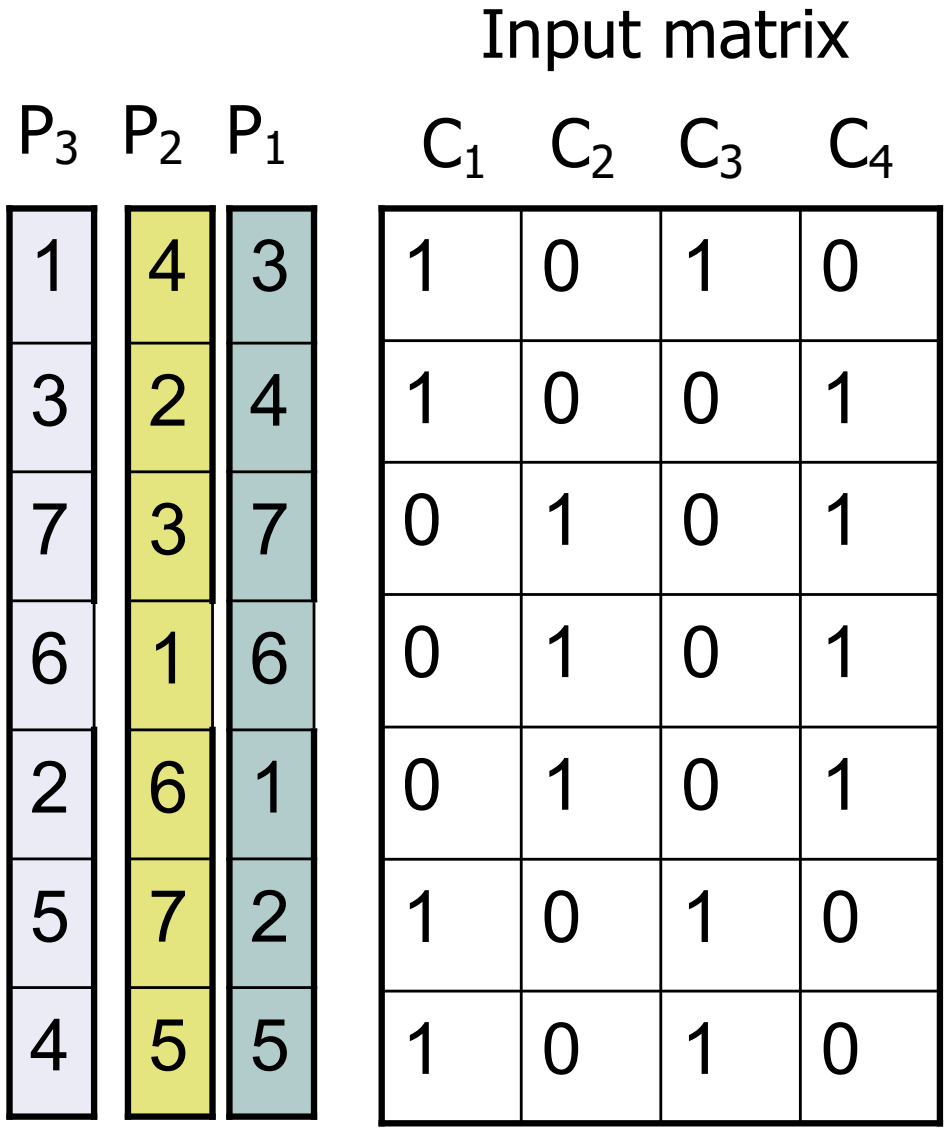
c and *m* are relatively prime (e.g., *c*=12, *m*=25)

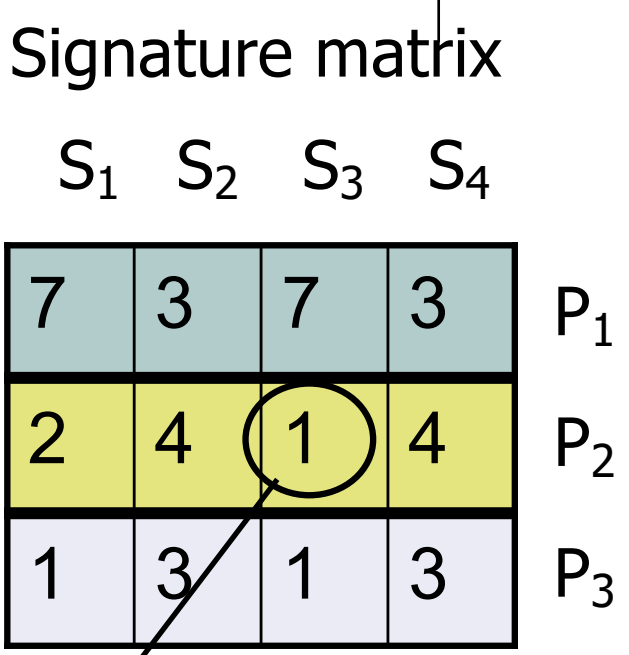
b=*a*-1 is multiple of *p*, for every prime *p* dividing *m*;

b is a multiple of 4 if *m* is a multiple of 4

Ex: m = 256, x_0 = 7; c = 71; a = 53

Example





The row with index = 1 in the order given by $P_2()$ is the first with value = 1

Example

Input matrix

P_3	P_2	P_1	C_1	C_2	C_3	C_4
1	4	3	1	0	1	0
3	2	4	1	0	0	1
7	3	7	0	1	0	1
6	1	6	0	1	0	1
2	6	1	0	1	0	1
5	7	2	1	0	1	0
4	5	5	1	0	1	0

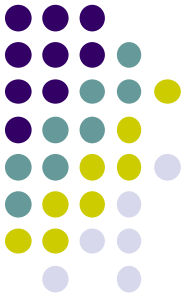
Signature matrix

S_1	S_2	S_3	S_4	
7	3	7	3	P_1
2	4	1	4	P_2
1	3	1	3	P_3

Objective...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

Another Example



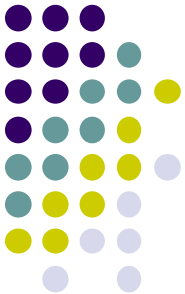
	C_1	C_2	C_3
1	1	0	1
2	0	1	1
3	1	0	0
4	1	0	1
5	0	1	0

Signatures

	S_1	S_2	S_3
Perm1 = (12345)	1	2	1
Perm2 = (54321)	4	5	4
Perm3 = (34512)	3	5	4

The row with index = 4 in the first with a "1" in the order given by *Perm2*().

Another Example



We *change the meaning* of the value stored in the Sig()!

	C_1	C_2	C_3
1	1	0	1
2	0	1	1
3	1	0	0
4	1	0	1
5	0	1	0

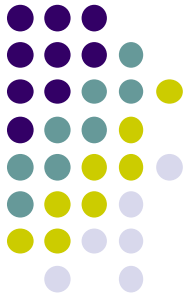
Signatures

	S_1	S_2	S_3
Perm1 = (12345)	1	2	1
Perm2 = (54321)	2	1	2
Perm3 = (34512)	1	3	2

The 2nd in the order given by *Perm2*() is the first "1"

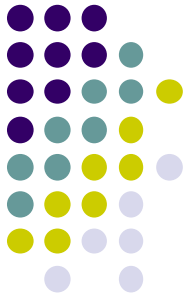
C_1 is also {1, 3, 4}, C_2 = {2, 5} and C_3 = {1, 2, 4}

Minhash Signatures



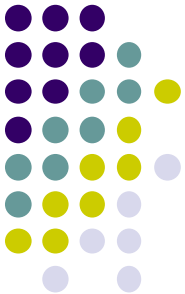
- Pick (say) 100 random permutations of the rows
- Think of *Sig*(C) as a column vector
  (with a reduced length, say *m*)
- Let *Sig*(C)[i] = according to the *i*th permutation over *m*, the number (in the order imposed) of the first row that has a "1" in column *C*.
- *Sig(C)* is now the signature of the column C (the storage needed has been reduced)

Surprising Property



- The probability (over all permutations of the rows) that $h(C_1) = h(C_2)$ is the same as $Sim(C_1, C_2)$.
- Both are $a / (a+b+c)$!
- Why?
  - Look down 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.
- Use several (100?) independent hash functions $h_1()$, $h_2()$, …, $h_k()$, …, $h_{100}()$ to create a signature.

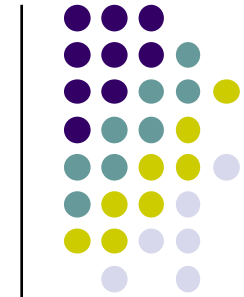
Similarity for Signatures



- The *similarity of signatures* is the fraction of the rows in which they agree.
- Similarity of signatures = fraction of permutations for which *minhash* values agree = (expected) similarity of columns
- See our examples
- We do not have a precise semantic attached to each signature (the value is related to the order of the values 1 and 0).

Example

The row with index = 7 in the order given by $P_1()$ is the first with value = 1



P_3	P_2	P_1	Input matrix			
1	4	3	1	0	1	0
3	2	4	1	0	0	1
7	3	7	0	1	0	1
6	1	6	0	1	0	1
2	6	1	0	1	0	1
5	7	2	1	0	1	0
4	5	5	1	0	1	0

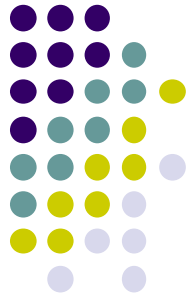
Signature matrix

7	3	7	3	P_1
2	4	1	4	P_2
1	3	1	3	P_3

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

Another Example



	C_1	C_2	C_3
1	1	0	1
2	0	1	1
3	1	0	0
4	1	0	1
5	0	1	0

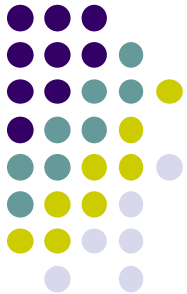
Signatures

	S_1	S_2	S_3
Perm1 = (12345)	1	2	1
Perm2 = (54321)	4	5	4
Perm3 = (34512)	3	5	4

Similarities

	1-2	1-3	2-3
Col-Col	0.00	0.50	0.25
Sig-Sig	0.00	0.67	0.00

Implementation



Idea: For each column C_i and each hash function $h_k()$, keep a "slot" $slot(C_i,h_k)$ for that *minhash* value.

- Pick (say) 100 hash functions
- For each column and each hash function, keep a "slot" for that min-hash value, init to ∞.
- For each row *r*
  for each column C_i with a 1 in the row *r*
    for each hash function $h_k(r)$ do:
      if hash function $h_k(r)$ < slot(h,c)
        replace slot$(h,c) \leftarrow h_k(r)$
- Minhash value : we select the min of a sequence of values.

Example

	Slots C1	C2
$h(1) = 1$	1	∞
$g(1) = 3$	3	∞

Row	C1	C2
1	1	0
2	0	1
3	1	1
4	1	0
5	0	1

We have a "1" in C_1, but not in C_2.
Thus ignore C_2 for the moment.
For C_1, store the two hash values

$h(2) = 2$	1	2
$g(2) = 5$	3	5

$h(x) = ((x\text{-}1) \bmod 5)+1$
$g(x) = (2x \bmod 5)+1$

We have a "1" in C_2, but not in C_1.
Thus ignore C_1 for this row.
For C_2, store the two hash values

Until now and for C_2, the min value returned by $h()$ is 2

Example

Row	C1	C2
1	1	0
2	0	1
3	1	1
4	1	0
5	0	1

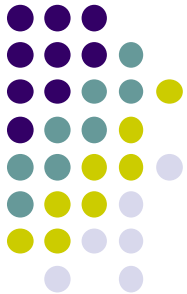
$h(x) = ((x\text{-}1) \bmod 5)+1$

$g(x) = (2x \bmod 5)+1$

	Slots C1	C2
$h(1) = 1$	**1***	∞
$g(1) = 3$	**3***	∞
$h(2) = 2$	1	**2***
$g(2) = 5$	3	**5***
$h(3) = 3$	**1***	**2***
$g(3) = 2$	**2***	**2***
$h(4) = 4$	**1***	2
$g(4) = 4$	**2***	2
$h(5) = 5$	1	**2***
$g(5) = 1$	2	**1***

The min value returned by $h()$ is 1 for C_1

Another Example



	C_1	C_2	C_3
1	1	0	1
2	0	1	1
3	1	0	0
4	1	0	1
5	0	1	0

	S_1	S_2	S_3	
Perm1 = (12345)	∞	∞	∞	Init
Perm2 = (54321)	∞	∞	∞	
Perm3 = (34512)	∞	∞	∞	
Perm1 = (12345)	1	∞	1	First row
Perm2 = (54321)	5	∞	5	
Perm3 = (34512)	3	∞	3	
Perm1 = (12345)	1	2	1	Second row
Perm2 = (54321)	5	4	4	
Perm3 = (34512)	3	4	3	

Another Example

	C_1	C_2	C_3
1	1	0	1
2	0	1	1
3	1	0	0
4	1	0	1
5	0	1	0

	S_1	S_2	S_3
Perm1 = (12345)	1	2	1
Perm2 = (54321)	5	4	4
Perm3 = (34512)	3	4	3
Perm1 = (12345)	1	2	1
Perm2 = (54321)	3	4	5
Perm3 = (34512)	3	4	3
Perm1 = (12345)	1	2	1
Perm2 = (54321)	2	4	2
Perm3 = (34512)	1	4	1

Second row

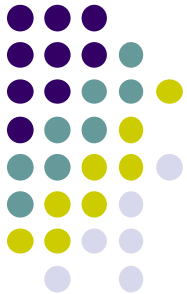
Third row

Fourth row

Perm1 = (12345)	1	2	1
Perm2 = (54321)	2	1	2
Perm3 = (34512)	1	2	1

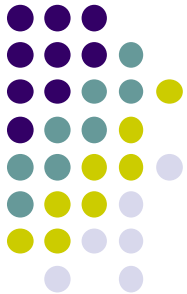
Fifth row

Checking Candidates



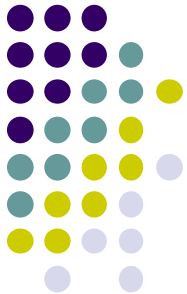
- Problem: Find the most similar pairs of items from a very large set of items
- While the signature of all columns may fit in main memory, comparing the signatures of all pairs of columns is quadratic in the number of columns
- Example: 10^6 columns implies $5 \cdot 10^{11}$ comparisons
  At 1 microsec./comparison: 6 days
  oups!
- *Locality Sensitive Hashing* (LSH) is a technique to limit the number of pairs of signatures we consider

Contents



- Applications
- Method
- Minhashing
  - Data as Sparse Matrix
  - Jaccard Similarity Measure
  - Constructing Signature
  - **Locality Sensitive Hashing (LSH)**

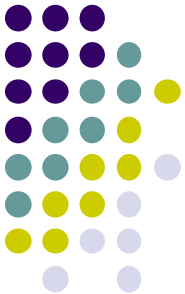
Partition into Bands



- Treat the *minhash* signatures as columns, with one row for each hash function
  But we still need to reduce this

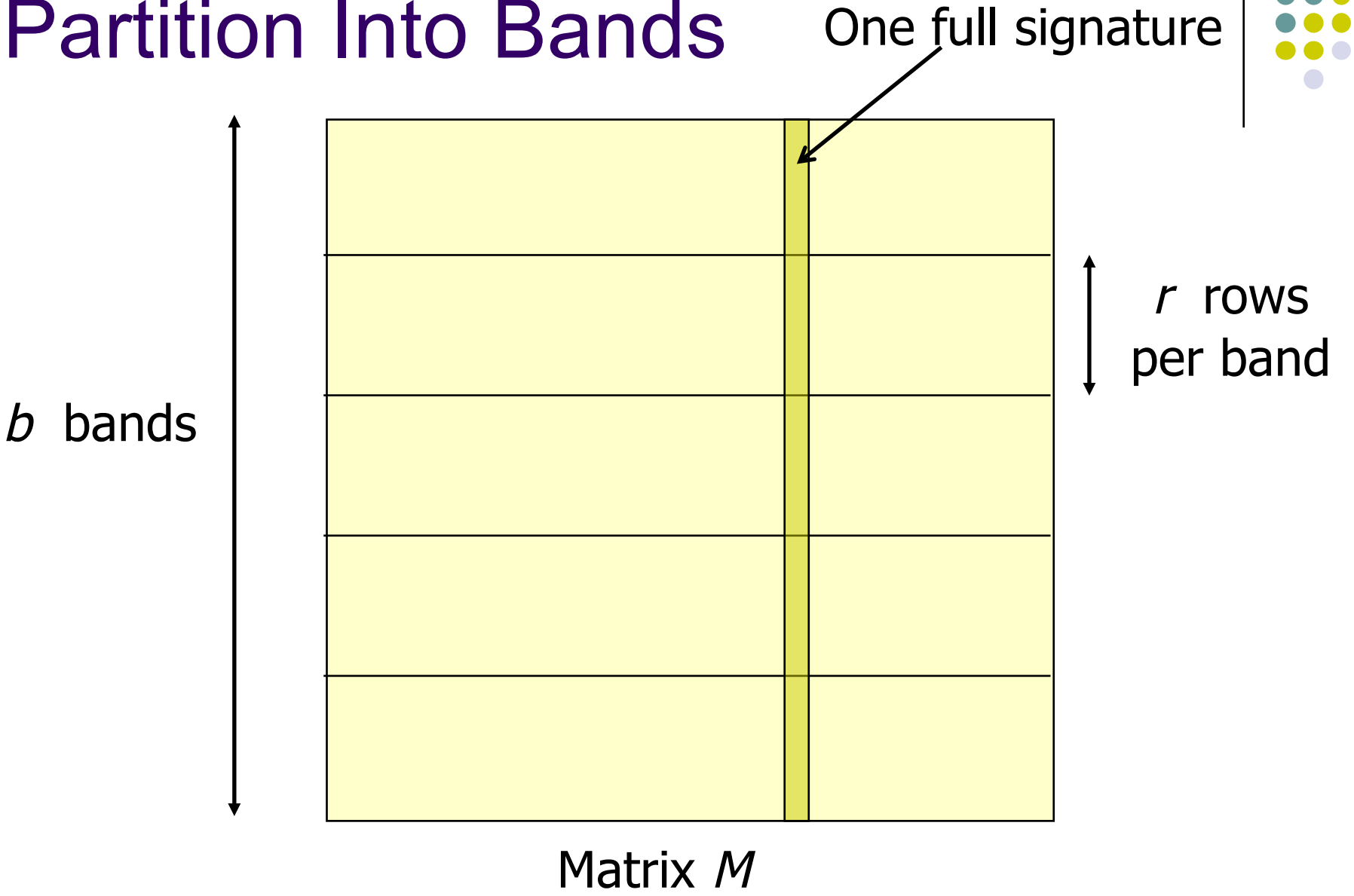
- Solution:
  Divide this matrix into *b* bands of *r* rows
  (free choice for *b* and *r*)
  in other words, each signature is divided into *b* bands and each band contains *r* values

Example

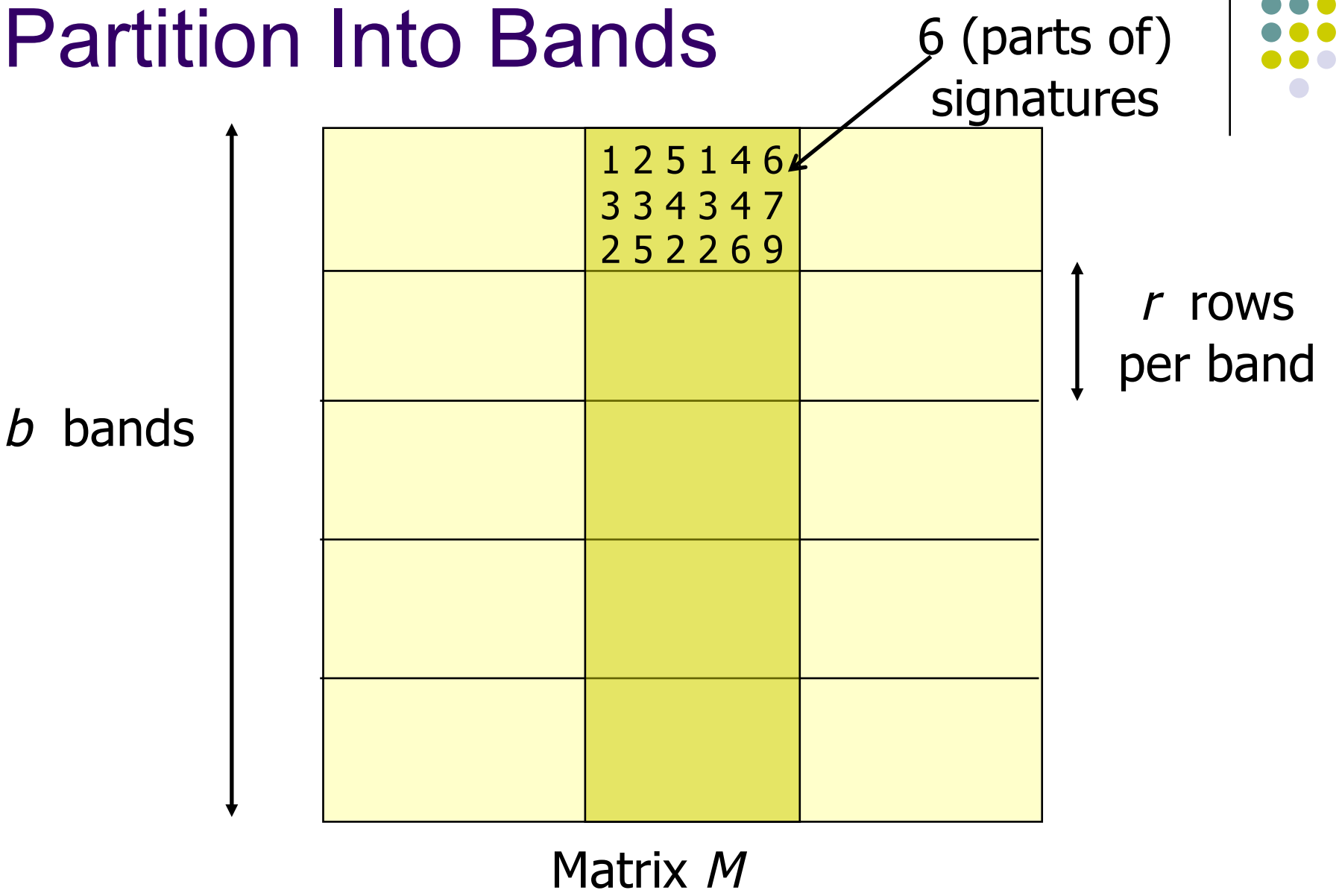


- Suppose 100,000 columns (10^5)
- Signatures of 100 integers (10^2)
- Therefore, memory needed: signatures take
  $4 \cdot 10^2 \cdot 10^5$ = 40MB
  we can store this into the main memory
- But $(10^5 \cdot 10^5)/2$ = 5,000,000,000 pairs of signatures
  can take a while to compare
- Choose b = 20 bands of r = 5 integers
  So $100 = 20 \cdot 5 = b \cdot r$

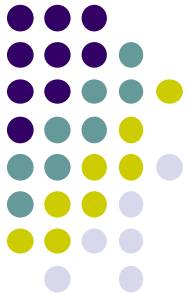
Partition Into Bands



Partition Into Bands



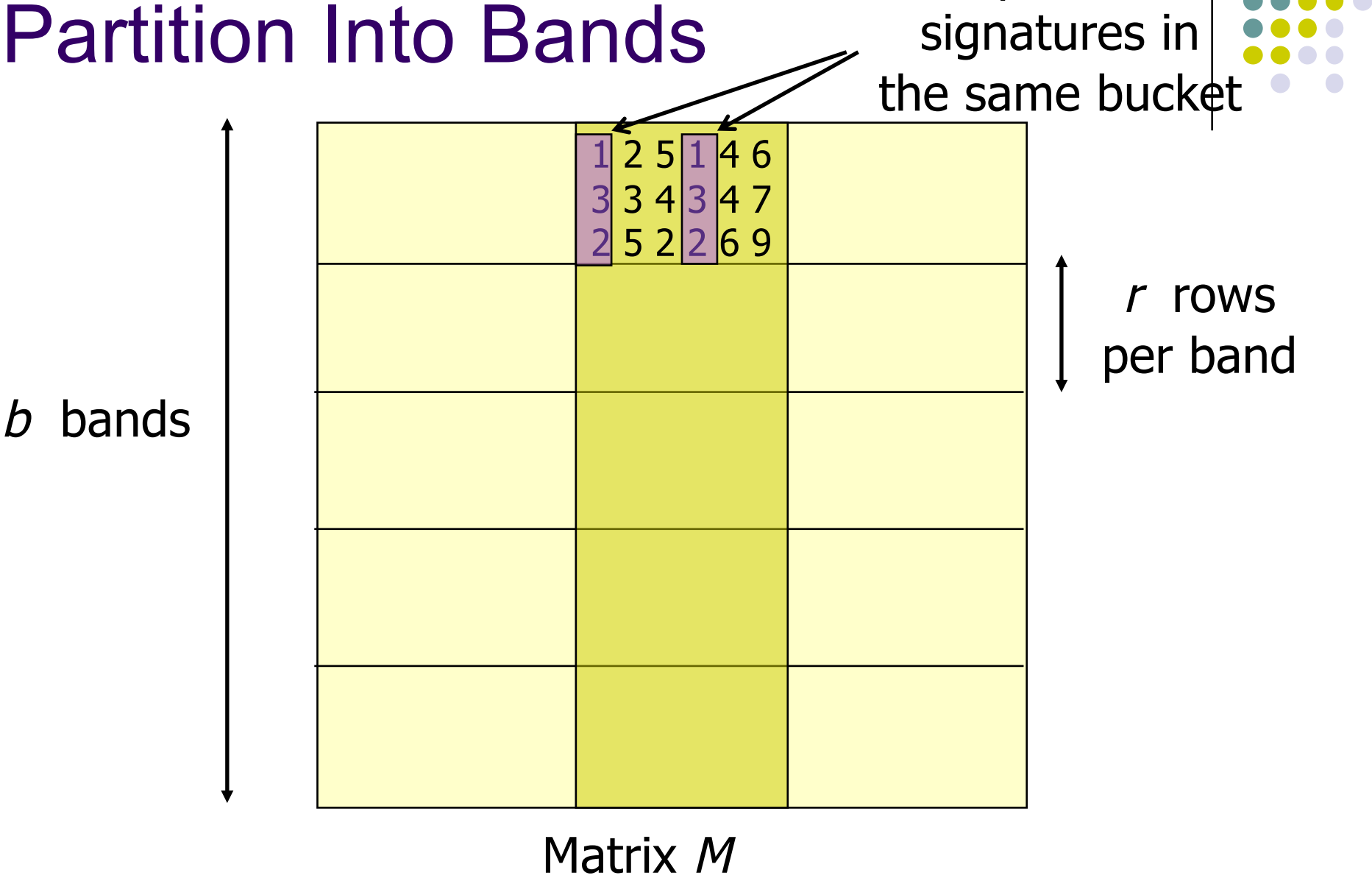
Partition into Bands



- Divide this signature matrix into *b* bands of *r* rows (free choice for *b* and *r*)
- For each band, hash its portion composed of *r* values of each column to *k* buckets (with *k* relatively large)

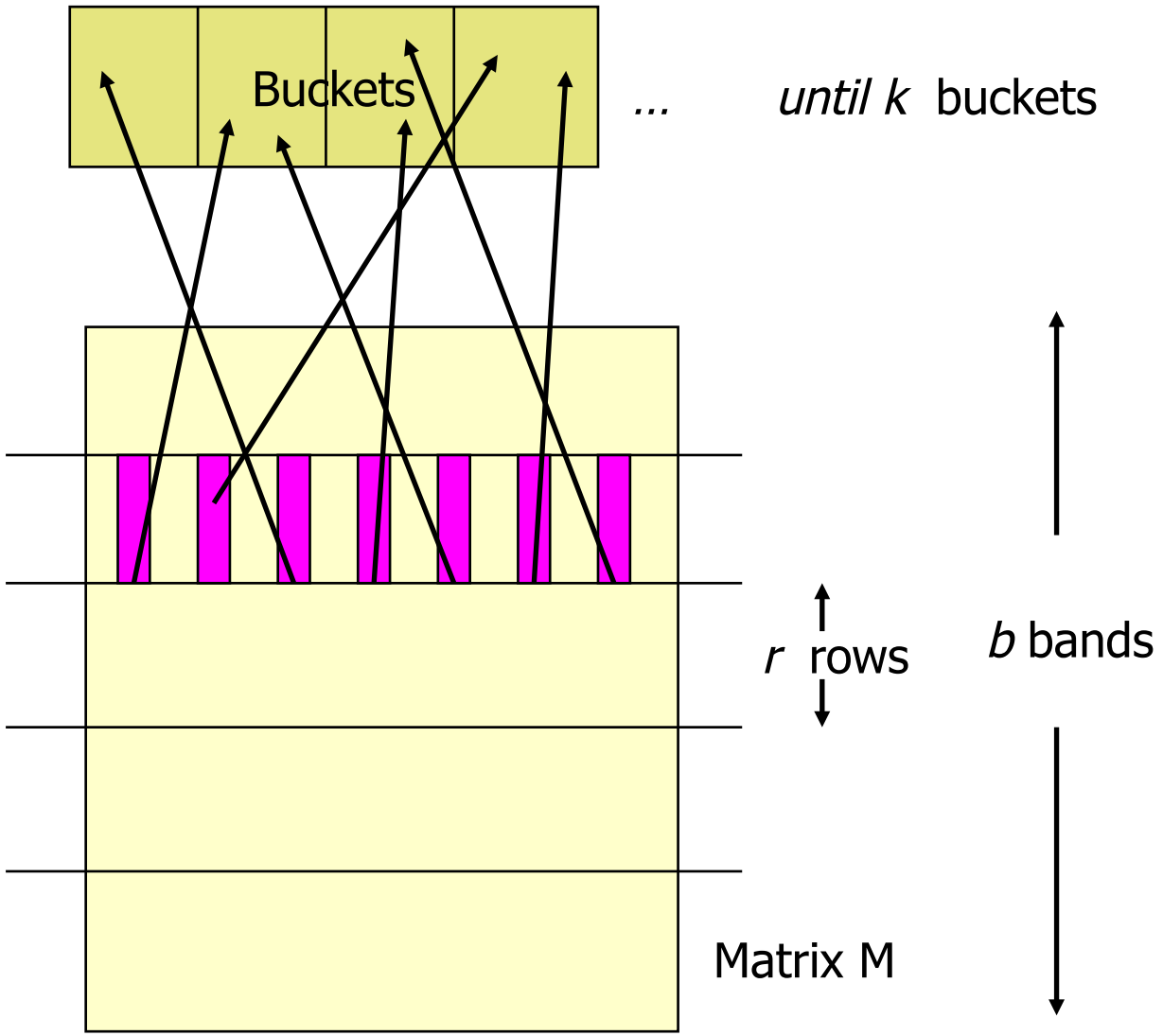
- Now…
- *Candidate* column pairs are those that hash to the same bucket for *one or more* of the *b* bands
- Can have false positive: dissimilar items appearing in the same bucket
- Tune *b*, *r*, *k* to catch most similar pairs, and minimize the nonsimilar pairs

Partition Into Bands

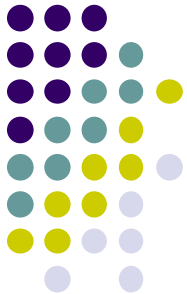


Partition Into Bands

- For each band, hash its portion of each column to a hash table with *k* buckets.
- Candidate column pairs are those that hash to the same bucket for *at least one* band.

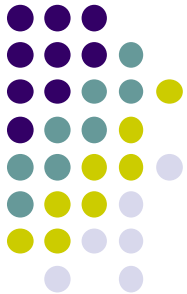


Partition into Bands



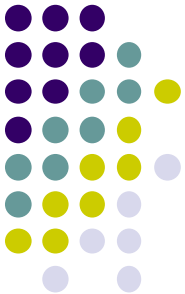
- Suppose C1, C2 are s = 80% similar
- Probability that C1, C2 identical in one particular band:
  $(0.8)^5 = 0.328$
  (because they must be equal for the all r = 5 values)
- Probability that C1, C2 are *not* similar in any of the b = 20 bands:
  $[1 - 0.328]^{20} = 0.00035$
  - i.e., we miss about 1/3000 of the 80% similar column pairs (this is our error due to compression!)
  - or the chance that we do find this pair of signature together in at least one bucket is 1-0.00035 = 0.99965

Partition into Bands



- Suppose C1, C2 are only *s* = 40% similar
  (not very interesting as pair of similar candidates)
- Probability that C1, C2 identical in one particular band:
  $(0.4)^5 = 0.01024$
- Probability that C1, C2 do not agree on any of the 20 bands:
  $[1 - 0.01024]^{20} = 0.814$
- Probability that C1, C2 are identical in 1 or more of the 20 b.
  $1 - [1 - 0.01024]^{20} = 1 - 0.814 = 0.186$
- If C1 and C2 are not identical in a band, there is a small probability that they hash to the same bucket
- False positives much lower for similarities *s* << 40%

Partition into Bands



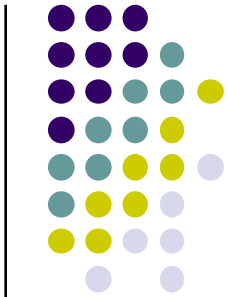
- In general, we have
- Probability that the signatures agree on one given row is

  s (Jaccard similarity)
- Probability that they agree on all r rows of a given band is

  s^r
- Probability that they do not agree on all the rows of a band is

  $1 - s^r$

Partition into Bands



- Probability that for none of the b bands do agree in all rows of that band is

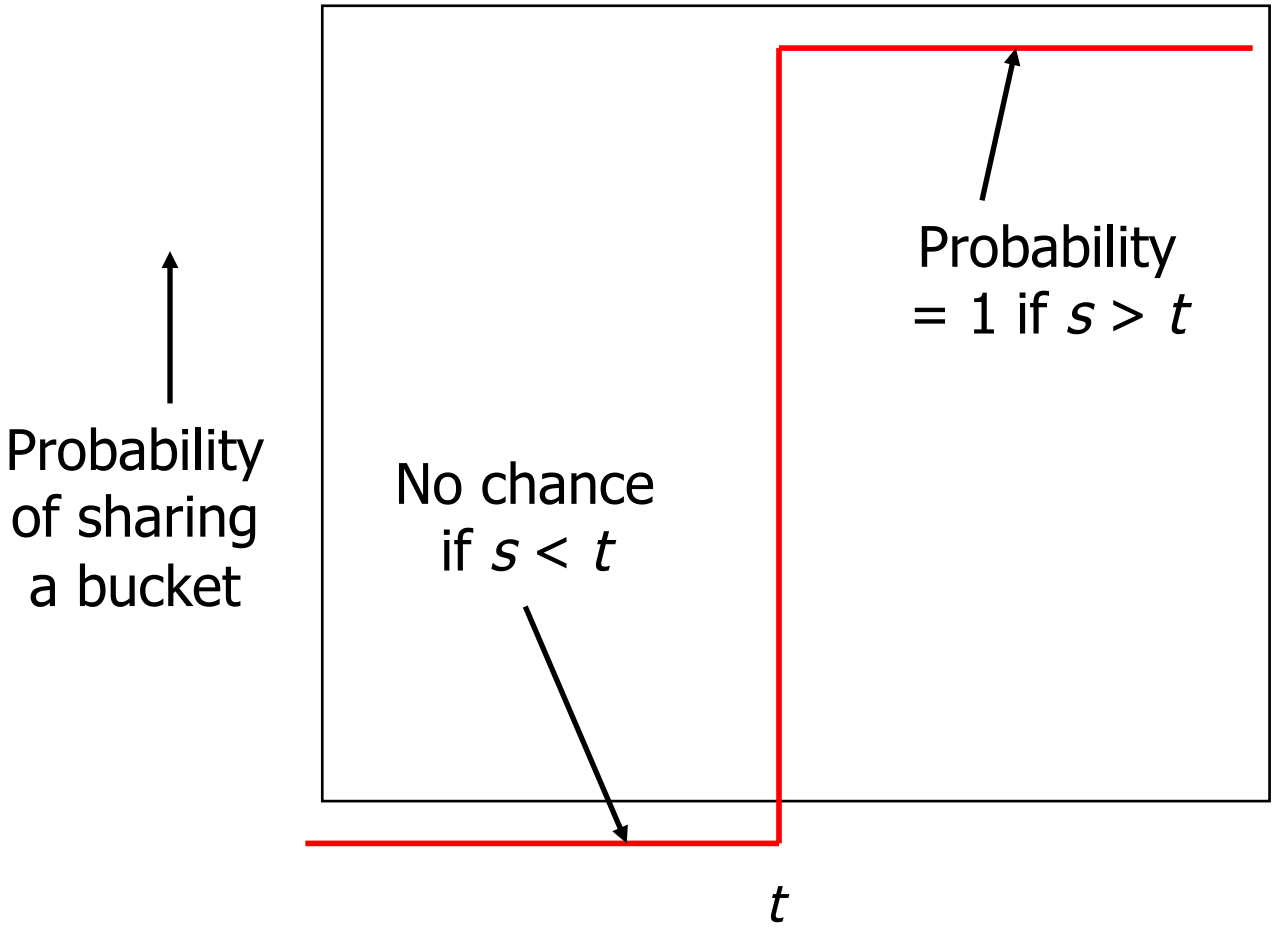
  $(1 - s^r)^b$

- Probability that the signatures will agree in all rows of at least one band is

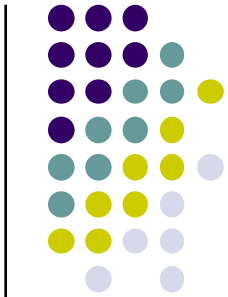
  $1 - (1 - s^r)^b$

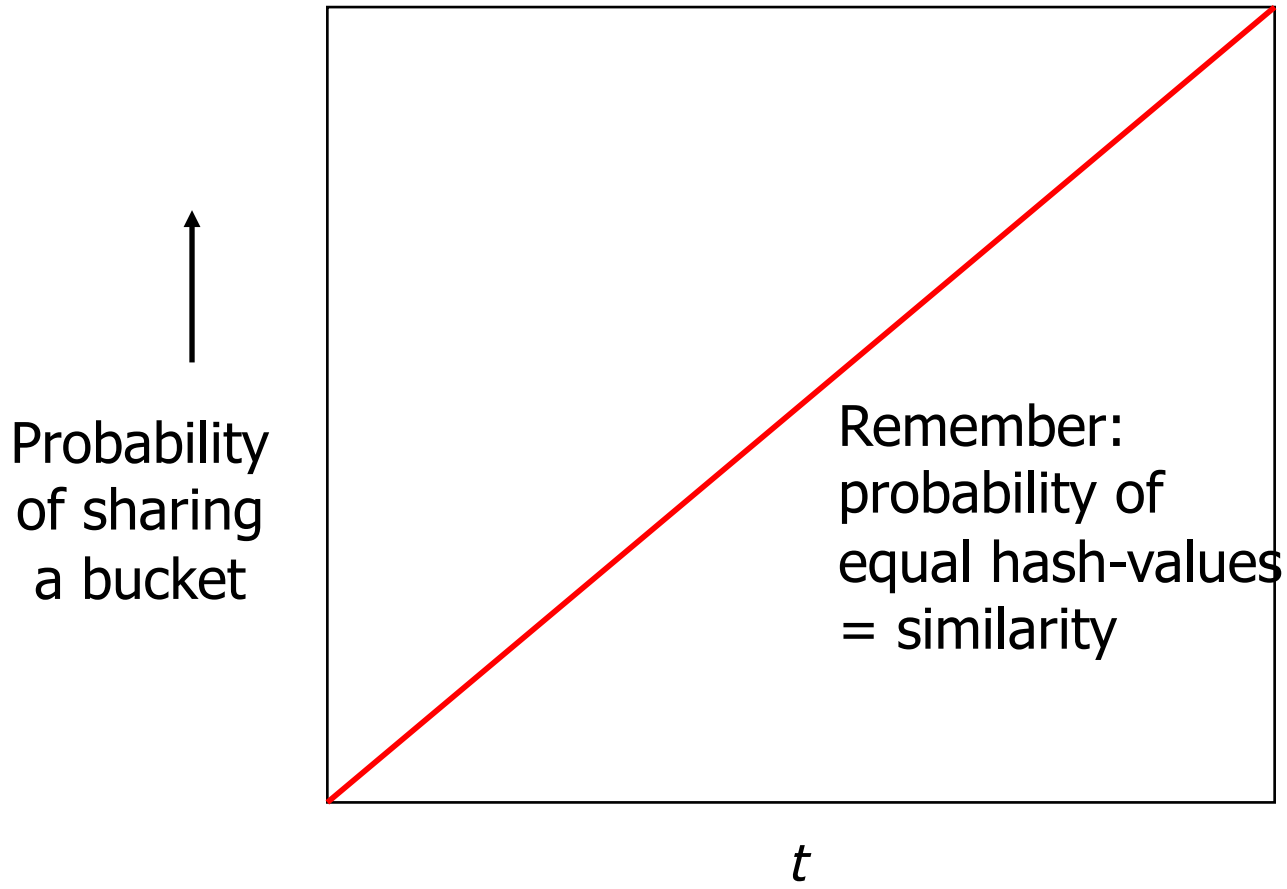
- This function is the probability that the signatures will be compared for similarity.

Analysis of LSH – What We Want

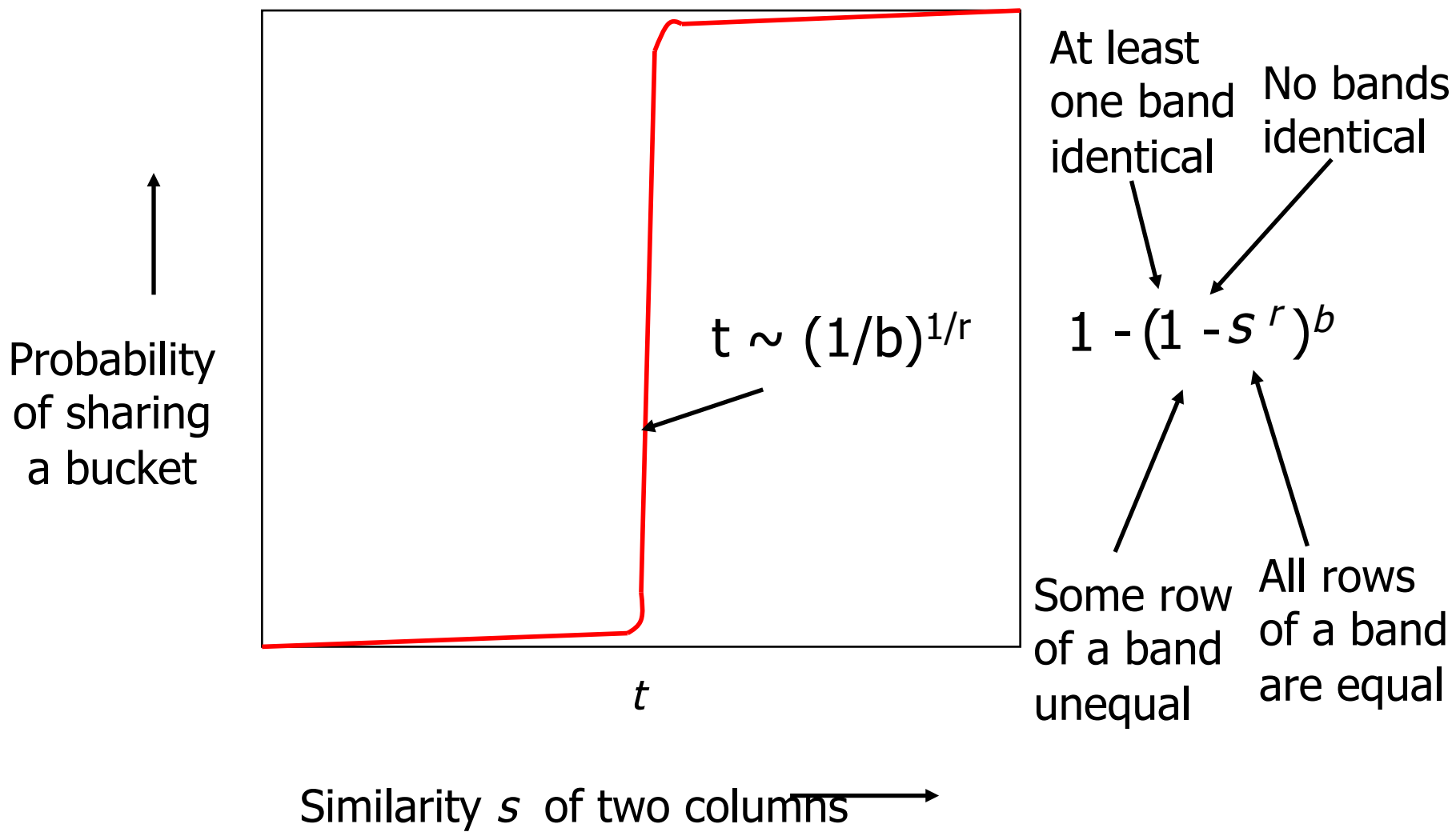


What One Row Gives You

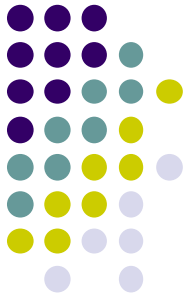




What *b* Bands of *r* Rows Gives You

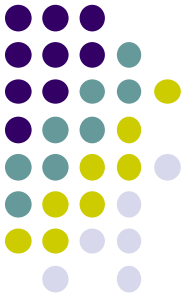


LSH Summary



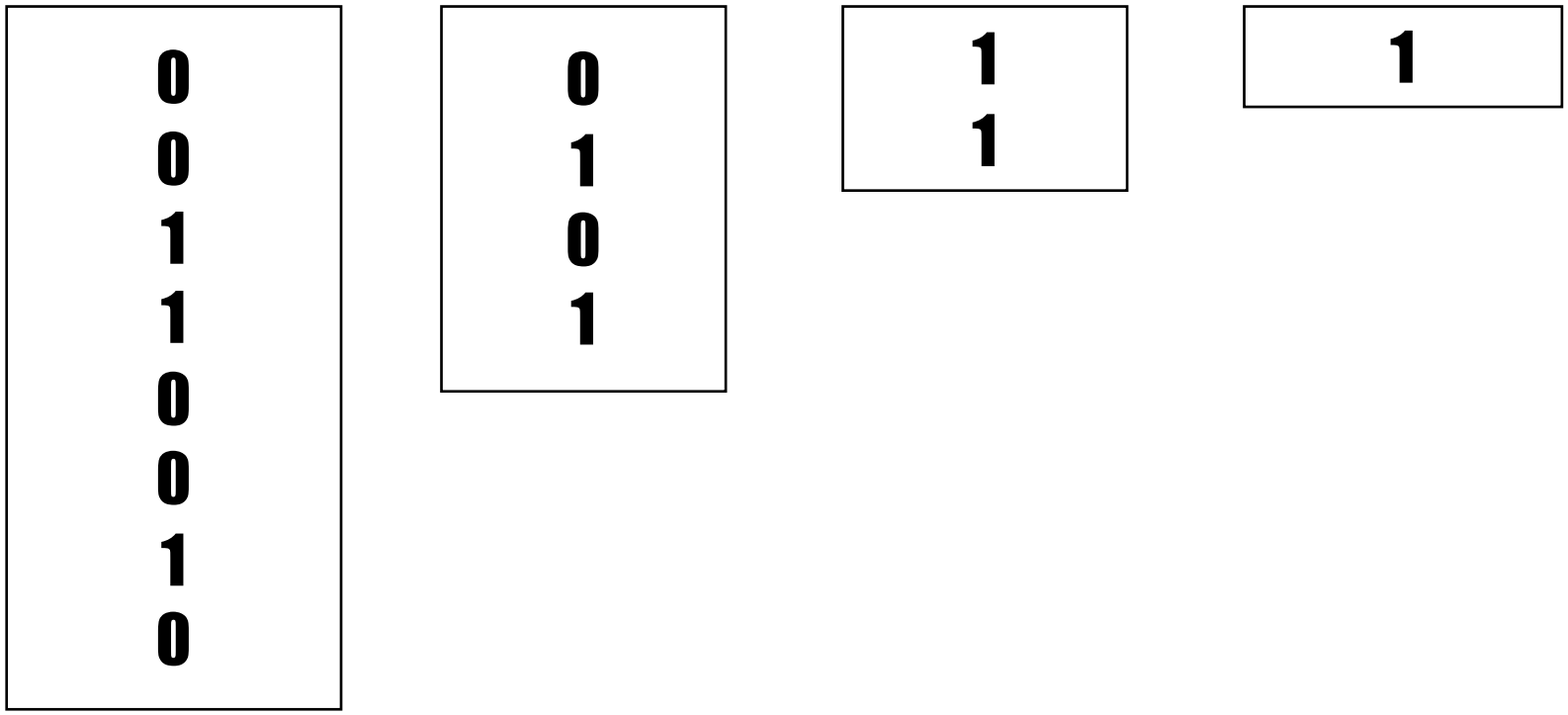
- Tune to get almost all pairs with similar signatures, but *eliminate* most pairs that do not have similar signatures
- Check in main memory that candidate pairs really do have similar signatures
- Then, in another pass through data, check that the remaining candidate pairs really are similar *columns*

Amplifications of 1's

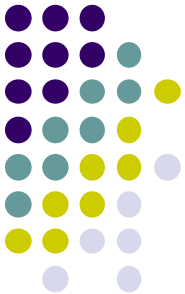


- If matrices are not sparse, then life is simpler: a random sample of (say) 100 rows serves as a good signature for columns
- *Hamming LSH* constructs a series of matrices, each with half as many rows, by OR-ing together pairs of rows
- Candidate pairs from each matrix have between 20% - 80% 1's and are similar in selected 100 rows

Example

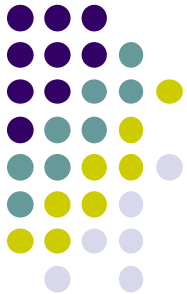


Using Hamming LSH



- Construct all matrices.
  - If there are *R* rows, then log *R* matrices.
  - Total work = twice that of reading the original matrix.
- Use standard LSH to identify similar columns in each matrix, but restricted to columns of "medium" density.

Conclusion



- Nearest Neighbor Search (NNS)
  - A problem with many applications
  - Working with Big Data means that a direct comparison between items is not possible (time constraint)
  - Shingles, MinHash functions and signatures can do the job
  - LSH when data size is huge