Algorithm Foundations of Data Science and Engineering

Lecture 2: Hashing Algorithms

YANHAO WANG

DaSE @ ECNU
(for course related communications)
yhwang@dase.ecnu.edu.cn

Sep. 20, 2021

Outline

Introduction

Searching

To find an entry (field of information) in the table, you only have to use the contents of one of the fields (say name in the case of the telephone book). You don't have to know the contents of all the fields. The field you use to find the contents of the other fields is called the **key**.

Introduction

Searching

To find an entry (field of information) in the table, you only have to use the contents of one of the fields (say name in the case of the telephone book). You don't have to know the contents of all the fields. The field you use to find the contents of the other fields is called the **key**.

	key	entry
0	14	<data>
1	45	<data>
2	22	<data>
3	67	<data>
4	17	<data>
⋮	*and so on*	

Introduction

Searching

To find an entry (field of information) in the table, you only have to use the contents of one of the fields (say name in the case of the telephone book). You don't have to know the contents of all the fields. The field you use to find the contents of the other fields is called the **key**.

	key	entry
0	14	<data>
1	45	<data>
2	22	<data>
3	67	<data>
4	17	<data>
⋮	*and so on*	

A telephone book has fields name, address and phone number. When you want to find somebodys phone number, you search the book based on the name field.

Introduction

Searching

To find an entry (field of information) in the table, you only have to use the contents of one of the fields (say name in the case of the telephone book). You don't have to know the contents of all the fields. The field you use to find the contents of the other fields is called the **key**.

	key	entry
0	14	<data>
1	45	<data>
2	22	<data>
3	67	<data>
4	17	<data>
⋮	*and so on*	

A telephone book has fields name, address and phone number. When you want to find somebodys phone number, you search the book based on the name field.

Ideally, the key should uniquely identify the entry, i.e. if the key is the name then no two entries in the telephone book have the same name.

Hash function

Hash function h:

$$h : key \rightarrow \text{ value }, i.e., h(key) = value \in \mathcal{Z}^+,$$

it is a mathematical function.

Hash function

Hash function h:

$$h : key \rightarrow \text{ value }, i.e., h(key) = value \in \mathcal{Z}^+,$$

it is a mathematical function.It also proposes many other features:

Hash function

Hash function h:

$$h : key \rightarrow \text{ value }, i.e., h(key) = value \in \mathcal{Z}^{+},$$

it is a mathematical function.It also proposes many other features:

- **Compression:** the input key is generally greater than the value of output. In this case, it is considered that the hash function compress the data that is submitted to it.

Hash function

Hash function h:

$$h : key \rightarrow \text{ value }, i.e., h(key) = value \in \mathcal{Z}^+,$$

it is a mathematical function.It also proposes many other features:

- **Compression:** the input key is generally greater than the value of output. In this case, it is considered that the hash function compress the data that is submitted to it.
- **Resistant to collisions:** two keys of different inputs may not lead to a value of identical output. It should be noted that even a minimal variation between two keys of inputs can lead to two values of outputs completely different. The reverse is also true. The same value of entry may not lead to two different outputs.

Hash function

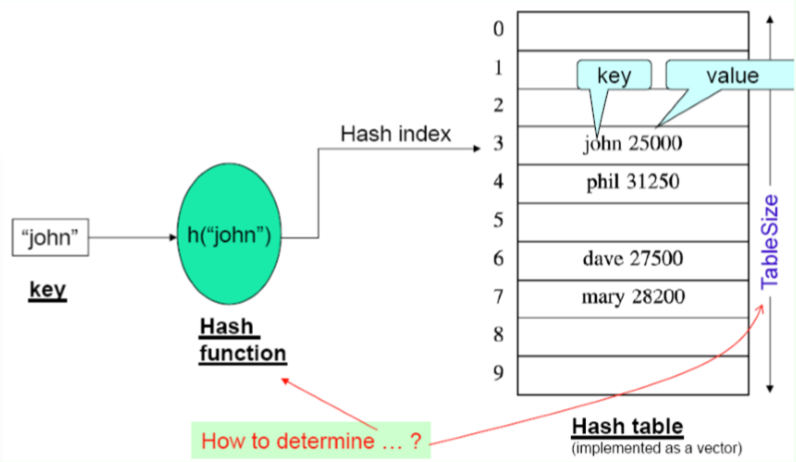
Hash function h:

$$h : key \rightarrow \text{ value }, i.e., h(key) = value \in \mathcal{Z}^{+},$$

it is a mathematical function.It also proposes many other features:

- **Compression:** the input key is generally greater than the value of output. In this case, it is considered that the hash function compress the data that is submitted to it.
- **Resistant to collisions:** two keys of different inputs may not lead to a value of identical output. It should be noted that even a minimal variation between two keys of inputs can lead to two values of outputs completely different. The reverse is also true. The same value of entry may not lead to two different outputs.
- **Pre-image resistance:** it is impossible to find an input key from the output. The only way to find the input value is to apply a method of "brute force".

Hash table



"john"
key
h("john")
Hash function
Hash index
0
1
2
3
4
5
6
7
8
9
key
value
john 25000
phil 31250
dave 27500
mary 28200
TableSize
Hash table
(implemented as a vector)
How to determine … ?

Complexity VS. Data structure

Operation	Unso. array	So. array	Linked list	Or. bin. Tree
Insert	$O(1)$	$O(n)$	$O(1)$ or $O(n)$	$O(\log n)$
Find	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$
Remove	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$

Table: Operation complexity of different data structures

Complexity VS. Data structure

Operation	Unso. array	So. array	Linked list	Or. bin. Tree
Insert	$O(1)$	$O(n)$	$O(1)$ or $O(n)$	$O(\log n)$
Find	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$
Remove	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$

Table: Operation complexity of different data structures

For hash table, insert, find and remove operations are usually $O(1)$, which certainly is not guaranteed.

Complexity VS. Data structure

Operation	Unso. array	So. array	Linked list	Or. bin. Tree
Insert	$O(1)$	$O(n)$	$O(1)$ or $O(n)$	$O(\log n)$
Find	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$
Remove	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$

Table: Operation complexity of different data structures

For hash table, insert, find and remove operations are usually $O(1)$, which certainly is not guaranteed.

There are three factors the influence the performance of hashing:

Complexity VS. Data structure

Operation	Unso. array	So. array	Linked list	Or. bin. Tree
Insert	$O(1)$	$O(n)$	$O(1)$ or $O(n)$	$O(\log n)$
Find	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$
Remove	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$

Table: Operation complexity of different data structures

For hash table, insert, find and remove operations are usually $O(1)$, which certainly is not guaranteed.

There are three factors the influence the performance of hashing:

- Hash function: should minimize collisions and distribute the keys and entries evenly throughout the entire table.

Complexity VS. Data structure

Operation	Unso. array	So. array	Linked list	Or. bin. Tree
Insert	$O(1)$	$O(n)$	$O(1)$ or $O(n)$	$O(\log n)$
Find	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$
Remove	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$

Table: Operation complexity of different data structures

For hash table, insert, find and remove operations are usually $O(1)$, which certainly is not guaranteed.
There are three factors the influence the performance of hashing:

- Hash function: should minimize collisions and distribute the keys and entries evenly throughout the entire table.
- Collision resolution strategy: store the key/entry in a different position, or chain several key/entries in the same position

Complexity VS. Data structure

Operation	Unso. array	So. array	Linked list	Or. bin. Tree
Insert	$O(1)$	$O(n)$	$O(1)$ or $O(n)$	$O(\log n)$
Find	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$
Remove	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$

Table: Operation complexity of different data structures

For hash table, insert, find and remove operations are usually $O(1)$, which certainly is not guaranteed.

There are three factors the influence the performance of hashing:

- Hash function: should minimize collisions and distribute the keys and entries evenly throughout the entire table.
- Collision resolution strategy: store the key/entry in a different position, or chain several key/entries in the same position
- Table size: too large a table will cause a wastage of memory; too small a table will cause increased collisions.

Bloom filter

A Bloom filter [1970] is a **space-efficient probabilistic data structure**, that is used to test whether an element is a member of a set.

Bloom filter

A Bloom filter [1970] is a **space-efficient probabilistic data structure**, that is used to test whether an element is a member of a set.

- False positive matches are possible, i.e., a query returns "possibly in set".

Bloom filter

A Bloom filter [1970] is a **space-efficient probabilistic data structure**, that is used to test whether an element is a member of a set.

- False positive matches are possible, i.e., a query returns “possibly in set”.
- False negatives are impossible, a query does not returns an item which is definitely not in set.

Bloom filter

A Bloom filter [1970] is a **space-efficient probabilistic data structure**, that is used to test whether an element is a member of a set.

- False positive matches are possible, i.e., a query returns "possibly in set".
- False negatives are impossible, a query does not returns an item which is definitely not in set.

There are many applications

Bloom filter

A Bloom filter [1970] is a **space-efficient probabilistic data structure**, that is used to test whether an element is a member of a set.

- False positive matches are possible, i.e., a query returns "possibly in set".
- False negatives are impossible, a query does not returns an item which is definitely not in set.

There are many applications

- **Spam address filter**: it can efficiently filter the spam address;

Bloom filter

A Bloom filter [1970] is a **space-efficient probabilistic data structure**, that is used to test whether an element is a member of a set.

- False positive matches are possible, i.e., a query returns "possibly in set".
- False negatives are impossible, a query does not returns an item which is definitely not in set.

There are many applications

- **Spam address filter**: it can efficiently filter the spam address;
- **Spell checker**: it enables you to correct the most cumbersome mistakes, with a high degree of accuracy and speed, and to improve your writing;

Bloom filter

A Bloom filter [1970] is a **space-efficient probabilistic data structure**, that is used to test whether an element is a member of a set.

- False positive matches are possible, i.e., a query returns "possibly in set".
- False negatives are impossible, a query does not returns an item which is definitely not in set.

There are many applications

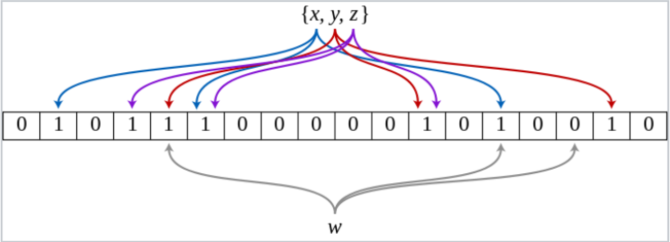
- **Spam address filter**: it can efficiently filter the spam address;
- **Spell checker**: it enables you to correct the most cumbersome mistakes, with a high degree of accuracy and speed, and to improve your writing;
- **Duplicate checker**: it help Web crawler to determine whether an URL has already been crawled.

Problem setting

Let S be a set of n elements. We are given a set of k hash functions, saying $h_1, h_2 \cdots, h_k$, with range $\{1, 2, \cdots, m\}$ or $(\{0, 1, \cdots, m-1\})$. Initially, $m-$long array of bits are set to 0.

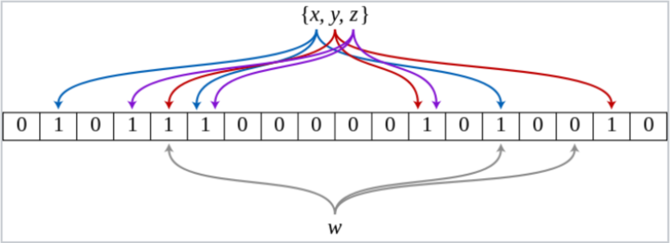
Problem setting

Let S be a set of n elements. We are given a set of k hash functions, saying $h_1, h_2 \cdots, h_k$, with range $\{1, 2, \cdots, m\}$ or $(\{0, 1, \cdots, m-1\})$. Initially, $m-$long array of bits are set to 0.



Problem setting

Let S be a set of n elements. We are given a set of k hash functions, saying $h_1, h_2 \cdots, h_k$, with range $\{1, 2, \cdots, m\}$ or $(\{0, 1, \cdots, m-1\})$. Initially, $m-$long array of bits are set to 0.



An example of a Bloom filter, representing the set $\{x, y, z\}$. The colored arrows show the positions in the bit array that each set element is mapped to. For this figure, $m = 18$ and $k = 3$.

Example

We insert and query on a Bloom filter of size $m = 10$ and number of hash functions $k = 3$. Let $H(x)$ denote the result of the three hash functions which we will write as a set of three values $\{h_1(x), h_2(x), h_3(x)\}$.

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	0	0	0	0	1

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	1	0	0	1	1

Example

We insert and query on a Bloom filter of size $m = 10$ and number of hash functions $k = 3$. Let $H(x)$ denote the result of the three hash functions which we will write as a set of three values $\{h_1(x), h_2(x), h_3(x)\}$.

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

We start with an empty 10-bit long array.

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	0	0	0	0	1

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	1	0	0	1	1

Example

We insert and query on a Bloom filter of size $m = 10$ and number of hash functions $k = 3$. Let $H(x)$ denote the result of the three hash functions which we will write as a set of three values $\{h_1(x), h_2(x), h_3(x)\}$.

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

We start with an empty 10-bit long array.

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	0	0	0	0	1

Insert x_0 with $H(x_0) = \{1, 4, 9\}$.

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	1	0	0	1	1

Example

We insert and query on a Bloom filter of size $m = 10$ and number of hash functions $k = 3$. Let $H(x)$ denote the result of the three hash functions which we will write as a set of three values $\{h_1(x), h_2(x), h_3(x)\}$.

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

We start with an empty 10-bit long array.

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	0	0	0	0	1

Insert x_0 with $H(x_0) = \{1, 4, 9\}$.

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	1	0	0	1	1

Insert x_1 with $H(x_1) = \{4, 5, 8\}$.

Query on Bloom filter

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	1	0	0	1	1

$H(x_0) = \{1, 4, 9\}$.
$H(x_1) = \{4, 5, 8\}$.

Query on Bloom filter

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	1	0	0	1	1

$H(x_0) = \{1, 4, 9\}$.
$H(x_1) = \{4, 5, 8\}$.

Check whether an item belongs to the set or not.

Query on Bloom filter

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	1	0	0	1	1

$H(x_0) = \{1, 4, 9\}$.
$H(x_1) = \{4, 5, 8\}$.

Check whether an item belongs to the set or not.

- Query y_0 with $H(y_0) = \{4, 5, 8\} \rightarrow$ Yes ($\surd$);

Query on Bloom filter

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	1	0	0	1	1

$H(x_0) = \{1, 4, 9\}$.
$H(x_1) = \{4, 5, 8\}$.

Check whether an item belongs to the set or not.

- Query y_0 with $H(y_0) = \{4, 5, 8\} \to$ Yes ($\surd$);
- Query y_1 with $H(y_1) = \{0, 4, 8\} \to$ NO ($\surd$);

Query on Bloom filter

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	1	0	0	1	1

$H(x_0) = \{1, 4, 9\}$.
$H(x_1) = \{4, 5, 8\}$.

Check whether an item belongs to the set or not.

- Query y_0 with $H(y_0) = \{4, 5, 8\} \rightarrow$ Yes ($\surd$);
- Query y_1 with $H(y_1) = \{0, 4, 8\} \rightarrow$ NO ($\surd$);
- Query y_2 with $H(y_2) = \{1, 5, 8\} \rightarrow$ Yes (False Positive);

False positive rate

When inserting an element into the filter, the probability that a certain bit is not set to one by a hash function is

$$1 - \frac{1}{m}.$$

False positive rate

When inserting an element into the filter, the probability that a certain bit is not set to one by a hash function is

$$1 - \frac{1}{m}.$$

Now, there are k hash functions, and the probability of any of them not having set a specific bit to one is given by

$$(1 - \frac{1}{m})^k.$$

False positive rate

When inserting an element into the filter, the probability that a certain bit is not set to one by a hash function is

$$1 - \frac{1}{m}.$$

Now, there are k hash functions, and the probability of any of them not having set a specific bit to one is given by

$$(1 - \frac{1}{m})^k.$$

After inserting n elements to the filter, the probability that a given bit is still zero is

$$(1 - \frac{1}{m})^{kn}.$$

False positive rate

When inserting an element into the filter, the probability that a certain bit is not set to one by a hash function is

$$1 - \frac{1}{m}.$$

Now, there are k hash functions, and the probability of any of them not having set a specific bit to one is given by

$$(1 - \frac{1}{m})^k.$$

After inserting n elements to the filter, the probability that a given bit is still zero is

$$(1 - \frac{1}{m})^{kn}.$$

And consequently the probability that the bit is one is

$$1 - (1 - \frac{1}{m})^{kn}.$$

False positive rate Cont'd

For an element membership test, if all of the k array positions in the filter computed by the hash functions are set to one, the Bloom filter claims that the element belongs to the set.

False positive rate Cont'd

For an element membership test, if all of the k array positions in the filter computed by the hash functions are set to one, the Bloom filter claims that the element belongs to the set.The probability of this happening when the element is not part of the set is given by

$$f = \left(1 - (1 - \frac{1}{m})^{kn}\right)^k \approx (1 - e^{-kn/m})^k.$$

x	$\left(1 - \frac{1}{x}\right)^{-x}$
4	3.160494
16	2.808404
64	2.739827
256	2.723610
1024	2.719610
4096	2.718614
16384	2.718365
65536	2.718303
262144	2.718287
1048576	2.718283
4194304	2.718282

False positive rate Cont'd

For an element membership test, if all of the k array positions in the filter computed by the hash functions are set to one, the Bloom filter claims that the element belongs to the set. The probability of this happening when the element is not part of the set is given by

$$f = \left(1 - (1 - \frac{1}{m})^{kn}\right)^k \approx (1 - e^{-kn/m})^k.$$

x	$\left(1-\frac{1}{x}\right)^{-x}$
4	3.160494
16	2.808404
64	2.739827
256	2.723610
1024	2.719610
4096	2.718614
16384	2.718365
65536	2.718303
262144	2.718287
1048576	2.718283
4194304	2.718282

- The false positive probability decreases as the size of the Bloom filter, m, increases.

False positive rate Cont'd

For an element membership test, if all of the k array positions in the filter computed by the hash functions are set to one, the Bloom filter claims that the element belongs to the set. The probability of this happening when the element is not part of the set is given by

$$f = \left(1 - (1 - \frac{1}{m})^{kn}\right)^k \approx (1 - e^{-kn/m})^k.$$

x	$\left(1 - \frac{1}{x}\right)^{-x}$
4	3.160494
16	2.808404
64	2.739827
256	2.723610
1024	2.719610
4096	2.718614
16384	2.718365
65536	2.718303
262144	2.718287
1048576	2.718283
4194304	2.718282

- The false positive probability decreases as the size of the Bloom filter, m, increases.
- The probability increases with n as more elements are added.

False positive rate Cont'd

For an element membership test, if all of the k array positions in the filter computed by the hash functions are set to one, the Bloom filter claims that the element belongs to the set.The probability of this happening when the element is not part of the set is given by

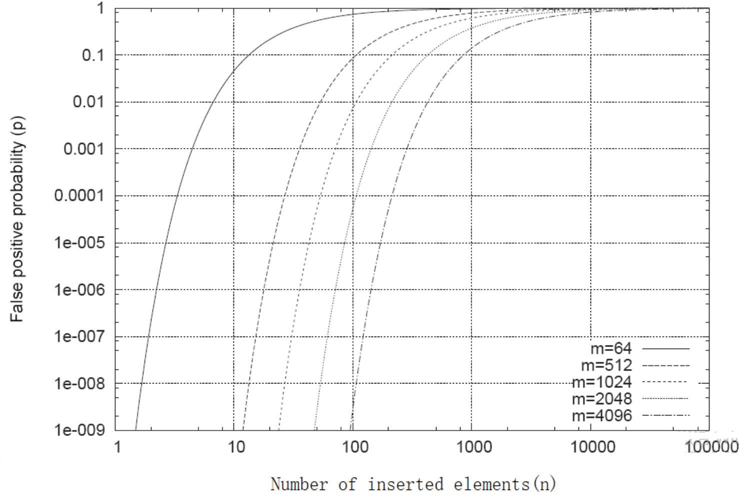
$$f = \left(1 - (1 - \frac{1}{m})^{kn}\right)^k \approx (1 - e^{-kn/m})^k.$$

x	$\left(1-\frac{1}{x}\right)^{-x}$
4	3.160494
16	2.808404
64	2.739827
256	2.723610
1024	2.719610
4096	2.718614
16384	2.718365
65536	2.718303
262144	2.718287
1048576	2.718283
4194304	2.718282

- The false positive probability decreases as the size of the Bloom filter, m, increases.
- The probability increases with n as more elements are added.

Now, we want to minimize the probability of false positives, by minimizing $(1 - e^{-kn/m})^k$ with respect to k.

False positive rate Cont'd



Choose k to minimize false positive rate

We are given m and n, so we choose a k to minimize the false positive rate.

Choose k to minimize false positive rate

We are given m and n, so we choose a k to minimize the false positive rate.
Let $p = e^{-kn/m}$. Thus we have

$$\begin{aligned} f &\approx (1 - e^{-kn/m})^k \\ &= (1-p)^k = e^{k \ln(1-p)}. \end{aligned}$$

Choose k to minimize false positive rate

We are given m and n, so we choose a k to minimize the false positive rate.
Let $p = e^{-kn/m}$. Thus we have

$$\begin{aligned} f &\approx (1 - e^{-kn/m})^k \\ &= (1-p)^k = e^{k \ln(1-p)}. \end{aligned}$$

So we wish to minimize $g = k \ln(1-p)$. Note that $\ln e^{-kn/m} = \frac{-kn}{m}$, we have

$$g = k \ln(1-p) = -\frac{m}{n} \ln(p) \ln(1-p)$$

Choose k to minimize false positive rate

We are given m and n, so we choose a k to minimize the false positive rate.

Let $p = e^{-kn/m}$. Thus we have

$$\begin{aligned} f &\approx (1 - e^{-kn/m})^k \\ &= (1-p)^k = e^{k \ln(1-p)}. \end{aligned}$$

So we wish to minimize $g = k \ln(1-p)$. Note that $\ln e^{-kn/m} = \frac{-kn}{m}$, we have

$$g = k \ln(1-p) = -\frac{m}{n} \ln(p) \ln(1-p)$$

Thus, we see that g is minimized when $p = \frac{1}{2}$.

Choose k to minimize false positives Cont'd

Since $p = e^{-kn/m}$, when $p = \frac{1}{2}$ we have

$$k = \ln 2 \cdot (\frac{m}{n}).$$

Choose k to minimize false positives Cont'd

Since $p = e^{-kn/m}$, when $p = \frac{1}{2}$ we have

$$k = \ln 2 \cdot (\frac{m}{n}).$$

Plugging back into $f = (1-p)^k$, we find the minimum false positive rate is

$$(\frac{1}{2})^k \approx (0.6185)^{\frac{m}{n}}.$$

Choose k to minimize false positives Cont'd

Since $p = e^{-kn/m}$, when $p = \frac{1}{2}$ we have

$$k = \ln 2 \cdot (\frac{m}{n}).$$

Plugging back into $f = (1-p)^k$, we find the minimum false positive rate is

$$(\frac{1}{2})^k \approx (0.6185)^{\frac{m}{n}}.$$

Thus, the optimal filter structure is $p = \frac{1}{2}$, which corresponds to a half-full Bloom filter array.

m, n, k Examples

False positive rates for choices of k given $\frac{m}{n}$

m/n	k	k=1	k=2	k=3	k=4	k=5
2	1.39	0.393	0.400			
3	2.08	0.283	0.237	0.253		
4	2.77	0.221	0.155	0.147	0.160	
5	3.46	0.181	0.109	0.092	0.092	0.101
6	4.16	0.154	0.0804	0.0609	0.0561	0.0578
7	4.85	0.133	0.0618	0.0423	0.0359	0.0347
8	5.55	0.118	0.0489	0.0306	0.024	0.0217

Distance measures

- Goal: Find near-neighbors in high-dimensional space. We formally define "near neighbors" as points that are a "small distance" apart

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

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

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

$$sim(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Distance measures

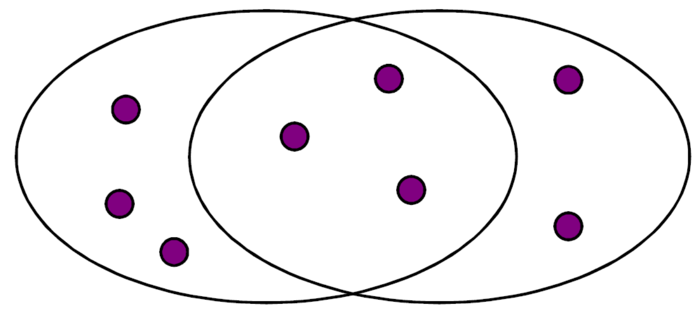
- Goal: Find near-neighbors in high-dimensional space. We formally define “near neighbors” as points that are a “small distance” apart
- For each application, we first need to define what “distance” means
- Today: Jaccard distance / similarity
  - The Jaccard similarity of two sets is the size of their intersection divided by the size of their union

$$sim(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

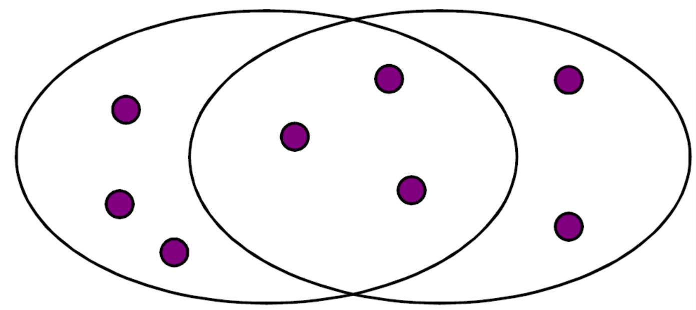
  - Jaccard distance (non-negative, symmetric, triangle inequality):

$$d(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}$$

Example

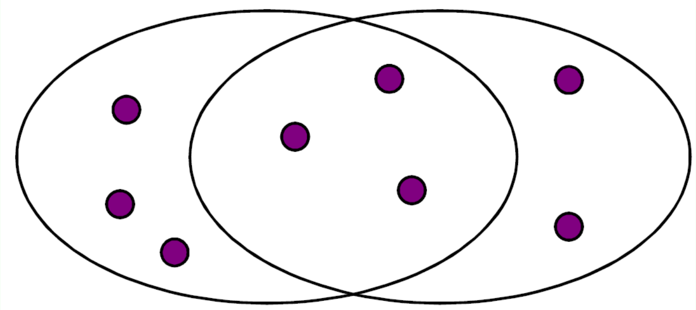


Example



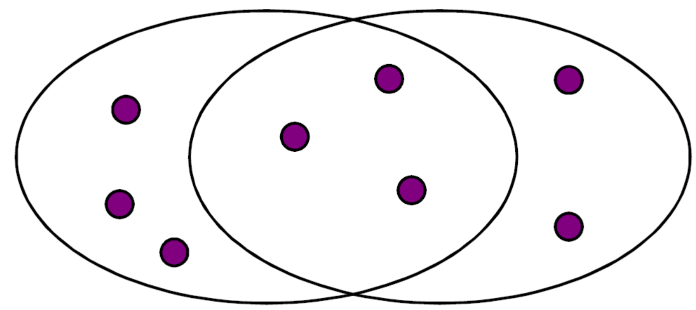
- 3 elements in intersection, 8 elements in union.

Example



- 3 elements in intersection, 8 elements in union.
- Jaccard similarity $= \frac{3}{8}$.

Example



- 3 elements in intersection, 8 elements in union.
- Jaccard similarity $= \frac{3}{8}$.
- Jaccard distance $= \frac{5}{8}$.

Task: Finding similar documents

- Goal: Given a large number (N in the millions or billions) of documents, find "near duplicate" pairs.

Task: Finding similar documents

- Goal: Given a large number (N in the millions or billions) of documents, find "near duplicate" pairs.
- Applications
  - Mirror websites, or approximate mirrors

Task: Finding similar documents

- Goal: Given a large number (N in the millions or billions) of documents, find "near duplicate" pairs.
- Applications
  - Mirror websites, or approximate mirrors
  - Similar new articles at many news sites

Task: Finding similar documents

- Goal: Given a large number (N in the millions or billions) of documents, find "near duplicate" pairs.
- Applications
  - Mirror websites, or approximate mirrors
  - Similar new articles at many news sites
  - Object blocking

Task: Finding similar documents

- Goal: Given a large number (N in the millions or billions) of documents, find “near duplicate” pairs.
- Applications
  - Mirror websites, or approximate mirrors
  - Similar new articles at many news sites
  - Object blocking
- Problems
  - Many small pieces of one document can appear out of order in another.

Task: Finding similar documents

- Goal: Given a large number (N in the millions or billions) of documents, find "near duplicate" pairs.
- Applications
  - Mirror websites, or approximate mirrors
  - Similar new articles at many news sites
  - Object blocking
- Problems
  - Many small pieces of one document can appear out of order in another.
  - Too many documents to compare all pairs.

Task: Finding similar documents

- Goal: Given a large number (N in the millions or billions) of documents, find "near duplicate" pairs.
- Applications
  - Mirror websites, or approximate mirrors
  - Similar new articles at many news sites
  - Object blocking
- 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.

Three essential steps for similar documents

- Shingling: Convert documents to sets

Three essential steps for similar documents

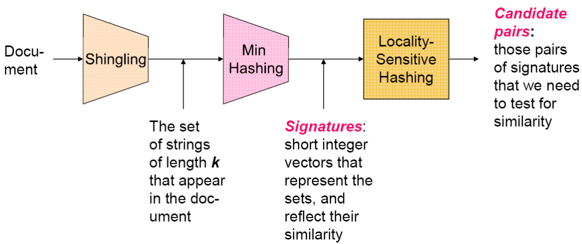
- Shingling: Convert documents to sets
- Min-Hashing: Convert large sets to short signatures, while preserving similarity

Three essential steps for similar documents

- Shingling: Convert documents to sets
- Min-Hashing: Convert large sets to short signatures, while preserving similarity
- Locality-Sensitive Hashing: Focus on pairs of signatures likely to be from similar documents

Three essential steps for similar documents

- Shingling: Convert documents to sets
- Min-Hashing: Convert large sets to short signatures, while preserving similarity
- Locality-Sensitive Hashing: Focus on pairs of signatures likely to be from similar documents



Shingling: Documents as high-dimensional data

Approach

Shingling: Convert documents to sets

Shingling: Documents as high-dimensional data

Approach

Shingling: Convert documents to sets

- Document = set of words appearing in document

Shingling: Documents as high-dimensional data

Approach

Shingling: Convert documents to sets

- Document = set of words appearing in document
- Document = set of “important” words

Shingling: Documents as high-dimensional data

Approach

Shingling: Convert documents to sets

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

Shingling: Documents as high-dimensional data

Approach

Shingling: Convert documents to sets

- 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!

Shingles

Definition

A k-shingle (or k-gram) for a document is a sequence of k tokens that appears in the document

- Tokens can be characters, words or something else, depending on the applications

Shingles

Definition

A k-shingle (or k-gram) for a document is a sequence of k tokens that appears in the document

- Tokens can be characters, words or something else, depending on the applications
- Assume tokens $=$ characters for examples

Shingles

Definition

A k-shingle (or k-gram) for a document is a sequence of k tokens that appears in the document

- Tokens can be characters, words or something else, depending on the applications
- Assume tokens = characters for examples

Let $k = 2$. String $D =$abcab can be converted into a set of 2-shlingles

$$S(D) = \{ab, bc, ca\}.$$

Shingles

Definition

A k-shingle (or k-gram) for a document is a sequence of k tokens that appears in the document

- Tokens can be characters, words or something else, depending on the applications
- Assume tokens = characters for examples

Let $k = 2$. String $D =$abcab can be converted into a set of 2-shlingles

$$S(D) = \{ab, bc, ca\}.$$

Option: Shingles as a bag (multiset), count ab twice

$$S(D) = \{ab, bc, ca, ab\}.$$

Compressing shingles

- To compress long shingles, we can hash them to (say) 4 bytes

Compressing shingles

- 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

Compressing shingles

- 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
- Idea: Two documents could (rarely) appear to have shingles in common, when in fact only the hash values were shared

Compressing shingles

- 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
- Idea: Two documents could (rarely) appear to have shingles in common, when in fact only the hash values were shared

Let $k = 2$. String $D =$abcab can be converted into a set of 2-shlingles

$$S(D) = \{ab, bc, ca\}.$$

Compressing shingles

- 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
- Idea: Two documents could (rarely) appear to have shingles in common, when in fact only the hash values were shared

Let $k = 2$. String $D =$abcab can be converted into a set of 2-shlingles

$$S(D) = \{ab, bc, ca\}.$$

Hash the singles: $h(D) = \{1, 5, 7\}$.

Similarity metric for Shingles

Document D is a set of its k-shingles $C = S(D)$.

- Equivalently, each document is a $0/1$ vector in the space of k-shingles

Similarity metric for Shingles

Document D is a set of its k-shingles $C = S(D)$.

- Equivalently, each document is a 0/1 vector in the space of k-shingles
  - Each unique shingle is a dimension

Similarity metric for Shingles

Document D is a set of its k-shingles $C = S(D)$.

- Equivalently, each document is a 0/1 vector in the space of k-shingles
  - Each unique shingle is a dimension
  - Vectors are very sparse

Similarity metric for Shingles

Document D is a set of its k-shingles $C = S(D)$.

- Equivalently, each document is a 0/1 vector in the space of k-shingles
  - Each unique shingle is a dimension
  - Vectors are very sparse
- A natural similarity measure is the Jaccard similarity:

$$sim(D_1, D_2) = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|}.$$

Similarity metric for Shingles

Document D is a set of its k-shingles $C = S(D)$.

- Equivalently, each document is a $0/1$ vector in the space of k-shingles
  - Each unique shingle is a dimension
  - Vectors are very sparse
- A natural similarity measure is the Jaccard similarity:

$$sim(D_1, D_2) = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|}.$$

Suppose we need to find near-duplicate documents among $n = 1$ million documents

- Naively, we would have to compute pairwise Jaccard similarities for every pair of docs

Similarity metric for Shingles

Document D is a set of its k-shingles $C = S(D)$.

- Equivalently, each document is a 0/1 vector in the space of k-shingles
  - Each unique shingle is a dimension
  - Vectors are very sparse
- A natural similarity measure is the Jaccard similarity:

$$sim(D_1, D_2) = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|}.$$

Suppose we need to find near-duplicate documents among $n = 1$ million documents

- Naively, we would have to compute pairwise Jaccard similarities for every pair of docs
  - $N(N-1)/2 \approx 5 \times 10^{11}$ comparisons

Similarity metric for Shingles

Document D is a set of its k-shingles $C = S(D)$.

- Equivalently, each document is a 0/1 vector in the space of k-shingles
  - Each unique shingle is a dimension
  - Vectors are very sparse
- A natural similarity measure is the Jaccard similarity:

$$sim(D_1, D_2) = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|}.$$

Suppose we need to find near-duplicate documents among $n = 1$ million documents

- Naively, we would have to compute pairwise Jaccard similarities for every pair of docs
  - $N(N-1)/2 \approx 5 \times 10^{11}$ comparisons
  - At 10^5 secs/day and 10^6 comparisons/sec, it would take 5 days.

Similarity metric for Shingles

Document D is a set of its k-shingles $C = S(D)$.

- Equivalently, each document is a 0/1 vector in the space of k-shingles
  - Each unique shingle is a dimension
  - Vectors are very sparse
- A natural similarity measure is the Jaccard similarity:

$$sim(D_1, D_2) = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|}.$$

Suppose we need to find near-duplicate documents among $n = 1$ million documents

- Naively, we would have to compute pairwise Jaccard similarities for every pair of docs
  - $N(N-1)/2 \approx 5 \times 10^{11}$ comparisons
  - At 10^5 secs/day and 10^6 comparisons/sec, it would take 5 days.
- For $N = 10$ million, it takes more than a year...

Encoding sets as bit vectors

Many similarity problems can be formalized as finding subsets that have significant intersection

Encoding sets as bit vectors

Many similarity problems can be formalized as finding subsets that have significant intersection

- Encode sets using 0/1 (bit, boolean) vectors, one dimension per element in the universal set

Encoding sets as bit vectors

Many similarity problems can be formalized as finding subsets that have significant intersection

- Encode sets using 0/1 (bit, boolean) vectors, one dimension per element in the universal set
- Interpret set intersection as bitwise AND, and set union as bitwise OR

Encoding sets as bit vectors

Many similarity problems can be formalized as finding subsets that have significant intersection

- Encode sets using 0/1 (bit, boolean) vectors, one dimension per element in the universal set
- Interpret set intersection as bitwise AND, and set union as bitwise OR

Example: $C_1 = 10111$, $C_2 = 10011$

Encoding sets as bit vectors

Many similarity problems can be formalized as finding subsets that have significant intersection

- Encode sets using 0/1 (bit, boolean) vectors, one dimension per element in the universal set
- Interpret set intersection as bitwise AND, and set union as bitwise OR

Example: $C_1 = 10111$, $C_2 = 10011$

- Size of intersection $= 3$; size of union $= 4$

Encoding sets as bit vectors

Many similarity problems can be formalized as finding subsets that have significant intersection

- Encode sets using 0/1 (bit, boolean) vectors, one dimension per element in the universal set
- Interpret set intersection as bitwise AND, and set union as bitwise OR

Example: $C_1 = 10111$, $C_2 = 10011$

- Size of intersection $= 3$; size of union $= 4$
- Jaccard similarity (not distance) $= 3/4$

Encoding sets as bit vectors

Many similarity problems can be formalized as finding subsets that have significant intersection

- Encode sets using 0/1 (bit, boolean) vectors, one dimension per element in the universal set
- Interpret set intersection as bitwise AND, and set union as bitwise OR

Example: $C_1 = 10111$, $C_2 = 10011$

- Size of intersection $= 3$; size of union $= 4$
- Jaccard similarity (not distance) $= 3/4$
- Distance: $d(C_1, C_2) = 1$ - (Jaccard similarity) $= 1/4$

Encoding sets as bit vectors

Documents

Shingles

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

Encoding sets as bit vectors

Documents

Shingles

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

Encoding sets as bit vectors

- Rows = elements (shingles), Columns = sets (documents)

Documents

Shingles

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

Encoding sets as bit vectors

Documents

Shingles

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

- Rows = elements (shingles), Columns = sets (documents)
- 1 in row e and column s if and only if e is a member of s

Encoding sets as bit vectors

Documents

Shingles

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

- Rows = elements (shingles), Columns = sets (documents)
- 1 in row e and column s if and only if e is a member of s
- Column similarity is the Jaccard similarity of the corresponding sets (rows with value 1)

Encoding sets as bit vectors

Documents

Shingles

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

- Rows = elements (shingles), Columns = sets (documents)
- 1 in row e and column s if and only if e is a member of s
- Column similarity is the Jaccard similarity of the corresponding sets (rows with value 1)
- Typical matrix is sparse!

Encoding sets as bit vectors

Documents

Shingles

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

- Rows = elements (shingles), Columns = sets (documents)
- 1 in row *e* and column *s* if and only if *e* is a member of *s*
- Column similarity is the Jaccard similarity of the corresponding sets (rows with value 1)
- Typical matrix is sparse!

Example: $sim(C_1, C_2) = ?$

Encoding sets as bit vectors

Documents

Shingles

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

- Rows = elements (shingles), Columns = sets (documents)
- 1 in row e and column s if and only if e is a member of s
- Column similarity is the Jaccard similarity of the corresponding sets (rows with value 1)
- Typical matrix is sparse!

Example: $sim(C_1, C_2) = ?$

- Size of interse. $= 3$; size of union $= 6$

Encoding sets as bit vectors

Documents

Shingles

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

- Rows = elements (shingles), Columns = sets (documents)
- 1 in row e and column s if and only if e is a member of s
- Column similarity is the Jaccard similarity of the corresponding sets (rows with value 1)
- Typical matrix is sparse!

Example: $sim(C_1, C_2) = ?$

- Size of interse. = 3; size of union = 6
- Jaccard similarity (not distance) = 3/6

Encoding sets as bit vectors

Documents

Shingles

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

- Rows = elements (shingles), Columns = sets (documents)
- 1 in row e and column s if and only if e is a member of s
- Column similarity is the Jaccard similarity of the corresponding sets (rows with value 1)
- Typical matrix is sparse!

Example: $sim(C_1, C_2) =?$

- Size of interse. $= 3$; size of union $= 6$
- Jaccard similarity (not distance) $= 3/6$
- Distance: $d(C_1, C_2) = 1$ - (Jaccard similarity) $= 3/6$

Finding similar columns

Next Goal: Find similar columns, Small signatures

Finding similar columns

Next Goal: Find similar columns, Small signatures

- Naive approach:

Finding similar columns

Next Goal: Find similar columns, Small signatures

- Naive approach:
  - Signatures of columns: small summaries of columns

Finding similar columns

Next Goal: Find similar columns, Small signatures

- Naive approach:
  - Signatures of columns: small summaries of columns
  - Examine pairs of signatures to find similar columns

Finding similar columns

Next Goal: Find similar columns, Small signatures

- Naive approach:
  - Signatures of columns: small summaries of columns
  - Examine pairs of signatures to find similar columns
  - Essential: Similarities of signatures and columns are related

Finding similar columns

Next Goal: Find similar columns, Small signatures

- Naive approach:
  - Signatures of columns: small summaries of columns
  - Examine pairs of signatures to find similar columns
  - Essential: Similarities of signatures and columns are related
  - Optional: Check that columns with similar signatures are really similar

Finding similar columns

Next Goal: Find similar columns, Small signatures

- Naive approach:
  - Signatures of columns: small summaries of columns
  - Examine pairs of signatures to find similar columns
  - Essential: Similarities of signatures and columns are related
  - Optional: Check that columns with similar signatures are really similar
- Warnings:

Finding similar columns

Next Goal: Find similar columns, Small signatures

- Naive approach:
  - Signatures of columns: small summaries of columns
  - Examine pairs of signatures to find similar columns
  - Essential: Similarities of signatures and columns are related
  - Optional: Check that columns with similar signatures are really similar
- Warnings:
  - Comparing all pairs may take too much time: Job for LSH

Finding similar columns

Next Goal: Find similar columns, Small signatures

- Naive approach:
  - Signatures of columns: small summaries of columns
  - Examine pairs of signatures to find similar columns
  - Essential: Similarities of signatures and columns are related
  - Optional: Check that columns with similar signatures are really similar
- Warnings:
  - Comparing all pairs may take too much time: Job for LSH
  - These methods can produce false negatives, and even false positives (if the optional check is not made)

Hashing columns (Signatures)

Key idea: "hash" each column C to a small signature $h(C)$, such that:

Hashing columns (Signatures)

Key idea: "hash" each column C to a small signature $h(C)$, such that:

- $h(C)$ is small enough that the signature fits in RAM

Hashing columns (Signatures)

Key idea: "hash" each column C to a small signature $h(C)$, such that:

- $h(C)$ is small enough that the signature fits in RAM
- $sim(C_1, C_2)$ is the same as the "similarity" of signatures $h(C_1)$ and $h(C_2)$

Hashing columns (Signatures)

Key idea: "hash" each column C to a small signature $h(C)$, such that:

- $h(C)$ is small enough that the signature fits in RAM
- $sim(C_1, C_2)$ is the same as the "similarity" of signatures $h(C_1)$ and $h(C_2)$

Goal: Find a hash function $h(\cdot)$ such that:

Hashing columns (Signatures)

Key idea: "hash" each column C to a small signature $h(C)$, such that:

- $h(C)$ is small enough that the signature fits in RAM
- $sim(C_1, C_2)$ is the same as the "similarity" of signatures $h(C_1)$ and $h(C_2)$

Goal: Find a hash function $h(\cdot)$ such that:

- If $sim(C_1, C_2)$ is high, then with high $P\big(h(C_1) = h(C_2)\big)$

Hashing columns (Signatures)

Key idea: "hash" each column C to a small signature $h(C)$, such that:

- $h(C)$ is small enough that the signature fits in RAM
- $sim(C_1, C_2)$ is the same as the "similarity" of signatures $h(C_1)$ and $h(C_2)$

Goal: Find a hash function $h(\cdot)$ such that:

- If $sim(C_1, C_2)$ is high, then with high $P\big(h(C_1) = h(C_2)\big)$
- If $sim(C_1, C_2)$ is low, then with high $P\big(h(C_1) \neq h(C_2)\big)$

Hashing columns (Signatures)

Key idea: "hash" each column C to a small signature $h(C)$, such that:

- $h(C)$ is small enough that the signature fits in RAM
- $sim(C_1, C_2)$ is the same as the "similarity" of signatures $h(C_1)$ and $h(C_2)$

Goal: Find a hash function $h(\cdot)$ such that:

- If $sim(C_1, C_2)$ is high, then with high $P\big(h(C_1) = h(C_2)\big)$
- If $sim(C_1, C_2)$ is low, then with high $P\big(h(C_1) \neq h(C_2)\big)$

Hash docs into buckets. Expect that "most" pairs of near duplicate docs hash into the same bucket!

Min-hashing

Goal: "most" pairs of near duplicate docs hash into the same bucket

Min-hashing

Goal: "most" pairs of near duplicate docs hash into the same bucket

- Clearly, the hash function depends on the similarity metric

Min-hashing

Goal: "most" pairs of near duplicate docs hash into the same bucket

- Clearly, the hash function depends on the similarity metric
- Not all similarity metrics have a suitable hash function

Min-hashing

Goal: "most" pairs of near duplicate docs hash into the same bucket

- Clearly, the hash function depends on the similarity metric
- Not all similarity metrics have a suitable hash function
- There is a suitable hash function for the Jaccard similarity: It is called **Min-Hashing**

Random permutation

Min-hashing

Goal: "most" pairs of near duplicate docs hash into the same bucket

- Clearly, the hash function depends on the similarity metric
- Not all similarity metrics have a suitable hash function
- There is a suitable hash function for the Jaccard similarity: It is called **Min-Hashing**

Random permutation

- A random permutation is a random ordering of a set of objects, that is, a permutation-valued random variable.

Min-hashing

Goal: "most" pairs of near duplicate docs hash into the same bucket

- Clearly, the hash function depends on the similarity metric
- Not all similarity metrics have a suitable hash function
- There is a suitable hash function for the Jaccard similarity: It is called **Min-Hashing**

Random permutation

- A random permutation is a random ordering of a set of objects, that is, a permutation-valued random variable.
- A good example of a random permutation is the shuffling of a deck of cards: this is ideally a random permutation of the 52 cards.

Min-hashing Cont'd

- Imagine the rows of the boolean matrix permuted under random permutation π

Min-hashing Cont'd

- Imagine the rows of the boolean matrix permuted under random permutation π
- Define a "hash" function $h_\pi(C)$ = the index of the first (in the permuted order π) row in which column C has value 1:

$$h_\pi(C) = \min_\pi \pi(C).$$

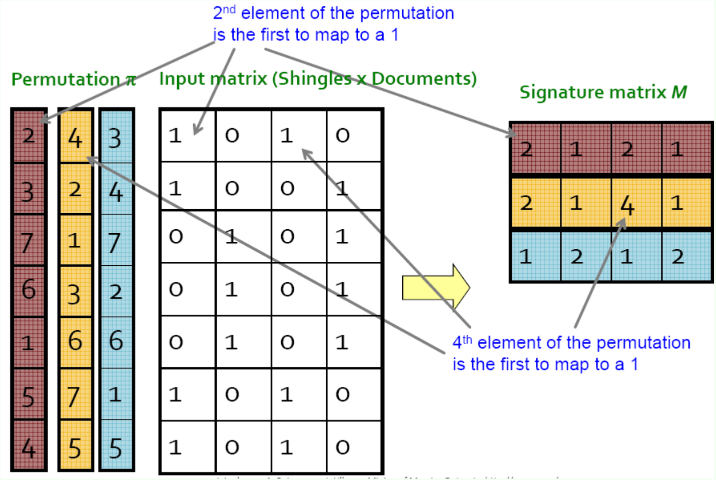
Min-hashing Cont'd

- Imagine the rows of the boolean matrix permuted under random permutation π
- Define a "hash" function $h_\pi(C)$ = the index of the first (in the permuted order π) row in which column C has value 1:

$$h_\pi(C) = \min_\pi \pi(C).$$

- Use several (e.g., 100) independent hash functions (that is, permutations) to create a signature of a column

Min-hashing example



The Min-hashing property

Given a random permutation π, we have

$$P\big(h_\pi(C_1) = h_\pi(C_2)\big) = sim(C_1, c_2)$$

The Min-hashing property

Given a random permutation π, we have

$$P\big(h_\pi(C_1) = h_\pi(C_2)\big) = sim(C_1, c_2)$$

- Let X be a doc (set of shingles), $y \in X$ is a shingle. Then

  $$P[\pi(y) = \min(\pi(X))] = 1/|X|,$$

  it is equally likely that any $y \in X$ is mapped to the *min* element.

The Min-hashing property

Given a random permutation π, we have

$$P\big(h_\pi(C_1) = h_\pi(C_2)\big) = sim(C_1, c_2)$$

- Let X be a doc (set of shingles), $y \in X$ is a shingle. Then

$$P[\pi(y) = \min(\pi(X))] = 1/|X|,$$

  it is equally likely that any $y \in X$ is mapped to the *min* element.

- Let y be s.t. $\pi(y) = \min(\pi(C_1 \cup C_2))$, then

$$\pi(y) = \min(\pi(C_1)) \text{ or } \pi(y) = \min(\pi(C_2))$$

The Min-hashing property

Given a random permutation π, we have

$$P\big(h_\pi(C_1) = h_\pi(C_2)\big) = sim(C_1, c_2)$$

- Let X be a doc (set of shingles), $y \in X$ is a shingle. Then

  $$P[\pi(y) = \min(\pi(X))] = 1/|X|,$$

  it is equally likely that any $y \in X$ is mapped to the *min* element.
- Let y be s.t. $\pi(y) = \min(\pi(C_1 \cup C_2))$, then

  $$\pi(y) = \min(\pi(C_1)) \text{ or } \pi(y) = \min(\pi(C_2))$$

- Thus, so the prob. that both are true is the prob. $y \in C_1 \cap C_2$. Final, we have

  $$P\big(\min(\pi(C_1)) = \min(\pi(C_2))\big) = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|} = sim(C_1, c_2).$$

Similarity for signatures

We know:

$$P\big(h_\pi(C_1) = h_\pi(C_2)\big) = sim(C_1, c_2).$$

Similarity for signatures

We know:

$$P\big(h_\pi(C_1) = h_\pi(C_2)\big) = sim(C_1, c_2).$$

- Now generalize to multiple hash functions

Similarity for signatures

We know:

$$P\big(h_\pi(C_1) = h_\pi(C_2)\big) = sim(C_1, c_2).$$

- Now generalize to multiple hash functions
- The similarity of two signatures is the fraction of the hash functions in which they agree

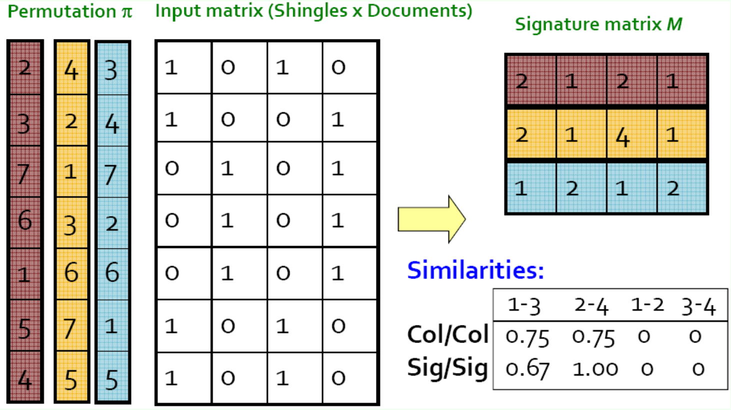
Similarity for signatures

We know:

$$P\big(h_\pi(C_1) = h_\pi(C_2)\big) = sim(C_1, c_2).$$

- Now generalize to multiple hash functions
- The similarity of two signatures is the fraction of the hash functions in which they agree
- Note: Because of the Min-Hash property, the similarity of columns is the same as the expected similarity of their signatures

Min-hashing-based similarity



Min-Hash signatures

- Pick $K = 100$ random permutations of the rows

Min-Hash signatures

- Pick $K = 100$ random permutations of the rows
- Think of $sig(C)$ as a column vector

Min-Hash signatures

- Pick $K = 100$ random permutations of the rows
- Think of $sig(C)$ as a column vector
- $sig(C)[i] =$ according to the i-th permutation, the index of the first row that has a 1 in column C

$$sig(C)[i] = \min(\pi_i(C)).$$

Min-Hash signatures

- Pick $K = 100$ random permutations of the rows
- Think of $sig(C)$ as a column vector
- $sig(C)[i] =$ according to the i-th permutation, the index of the first row that has a 1 in column C

$$sig(C)[i] = \min(\pi_i(C)).$$

- **Note:** The sketch (signature) of document C is small ~ 100 **bytes!**

Min-Hash signatures

- Pick $K = 100$ random permutations of the rows
- Think of $sig(C)$ as a column vector
- $sig(C)[i] =$ according to the i-th permutation, the index of the first row that has a 1 in column C

$$sig(C)[i] = \min\left(\pi_i(C)\right).$$

- **Note:** The sketch (signature) of document C is small ~ 100 **bytes!**
- We achieved our goal! We “compressed” long bit vectors into short signatures

LSH: First cut

- Goal: Find documents with Jaccard similarity $\geq s$ (e.g., $s = 0.8$)

LSH: First cut

- Goal: Find documents with Jaccard similarity $\geq s$ (e.g., $s = 0.8$)
- LSH - **General idea**: Use a function $f(x, y)$ that tells whether x and y is a candidate pair: a pair of elements whose similarity must be evaluated

LSH: First cut

- Goal: Find documents with Jaccard similarity $\geq s$ (e.g., $s = 0.8$)
- LSH - **General idea**: Use a function $f(x, y)$ that tells whether x and y is a candidate pair: a pair of elements whose similarity must be evaluated
- For Min-Hash matrices:

LSH: First cut

- Goal: Find documents with Jaccard similarity $\geq s$ (e.g., $s = 0.8$)
- LSH - **General idea**: Use a function $f(x, y)$ that tells whether x and y is a candidate pair: a pair of elements whose similarity must be evaluated
- For Min-Hash matrices:
  - Hash columns of signature matrix M to many buckets

LSH: First cut

- Goal: Find documents with Jaccard similarity $\geq s$ (e.g., $s = 0.8$)
- LSH - **General idea**: Use a function $f(x, y)$ that tells whether x and y is a candidate pair: a pair of elements whose similarity must be evaluated
- For Min-Hash matrices:
  - Hash columns of signature matrix M to many buckets
  - Each pair of documents that hashes into the same bucket is a candidate pair

Candidates from min-hash

LSH: First cut

- Goal: Find documents with Jaccard similarity $\geq s$ (e.g., $s = 0.8$)
- LSH - **General idea**: Use a function $f(x, y)$ that tells whether x and y is a candidate pair: a pair of elements whose similarity must be evaluated
- For Min-Hash matrices:
  - Hash columns of signature matrix M to many buckets
  - Each pair of documents that hashes into the same bucket is a candidate pair

Candidates from min-hash

- Pick a similarity threshold $s (0 < s < 1)$

LSH: First cut

- Goal: Find documents with Jaccard similarity $\geq s$ (e.g., $s = 0.8$)
- LSH - **General idea**: Use a function $f(x, y)$ that tells whether x and y is a candidate pair: a pair of elements whose similarity must be evaluated
- For Min-Hash matrices:
  - Hash columns of signature matrix M to many buckets
  - Each pair of documents that hashes into the same bucket is a candidate pair

Candidates from min-hash

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

LSH for min-hash

- Big idea: Hash columns of signature matrix M several times

LSH for min-hash

- Big idea: Hash columns of signature matrix M several times
- Arrange that (only) similar columns are likely to hash to the same bucket, with high probability

LSH for min-hash

- Big idea: Hash columns of signature matrix M several times
- Arrange that (only) similar columns are likely to hash to the same bucket, with high probability
- Candidate pairs are those that hash to the same bucket

Candidates from min-hash

LSH for min-hash

- Big idea: Hash columns of signature matrix M several times
- Arrange that (only) similar columns are likely to hash to the same bucket, with high probability
- Candidate pairs are those that hash to the same bucket

Candidates from min-hash

- Pick a similarity threshold $s(0 < s < 1)$

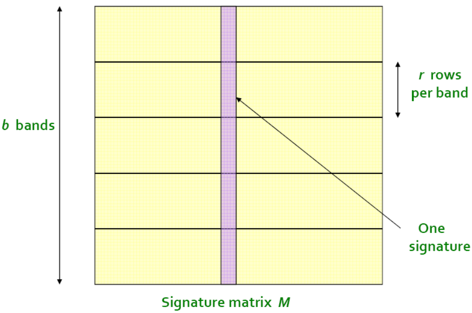
LSH for min-hash

- Big idea: Hash columns of signature matrix M several times
- Arrange that (only) similar columns are likely to hash to the same bucket, with high probability
- Candidate pairs are those that hash to the same bucket

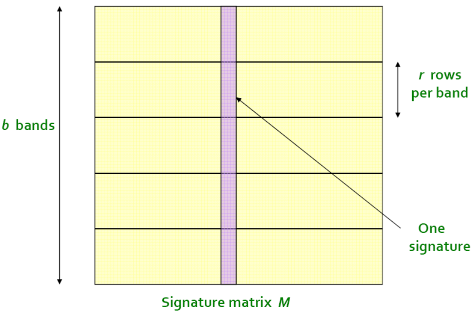
Candidates from min-hash

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

Grouping signatures

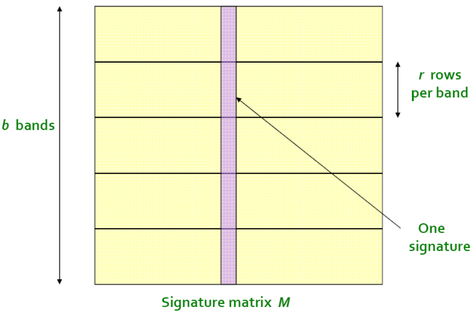


Grouping signatures



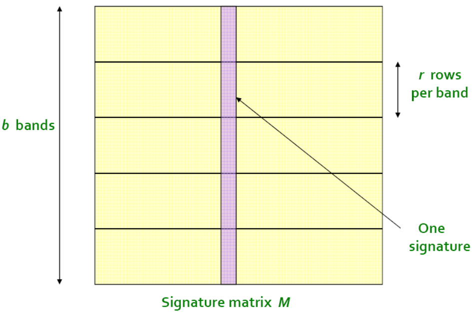
- Divide matrix M into b bands of r rows

Grouping signatures



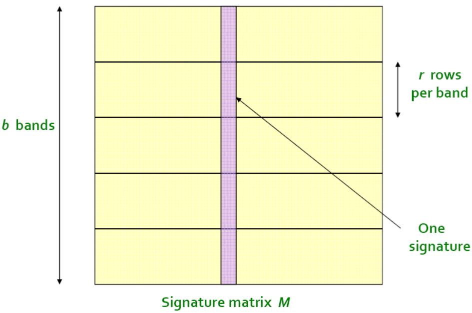
- Divide matrix M into b bands of r rows
- For each band, hash its portion of each column to a hash table with k buckets

Grouping signatures



- Divide matrix M into b bands of r rows
- For each band, hash its portion of each column to a hash table with k buckets
- Candidate column pairs are those that hash to the same bucket for ≥ 1 band

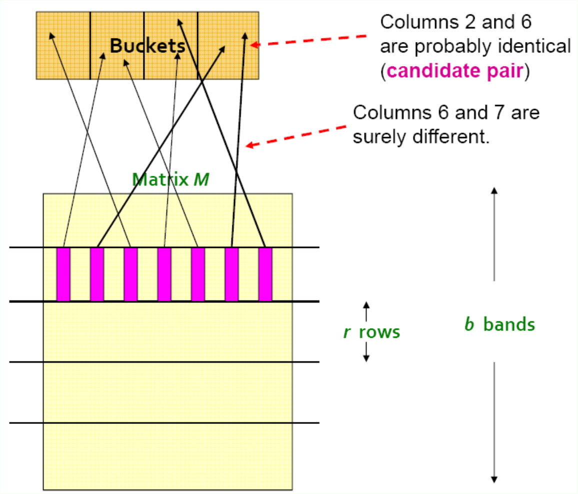
Grouping signatures



- Divide matrix M into b bands of r rows
- For each band, hash its portion of each column to a hash table with k buckets
- Candidate column pairs are those that hash to the same bucket for ≥ 1 band

Tune b and r to catch most similar pairs, but few non-similar pairs

Hashing bands



Assumptions

- There are enough buckets that columns are unlikely to hash to the same bucket unless they are identical in a particular band

Assumptions

- There are enough buckets that columns are unlikely to hash to the same bucket unless they are identical in a particular band
- Hereafter, we assume that "same bucket" means "identical in that band"

Assumptions

- There are enough buckets that columns are unlikely to hash to the same bucket unless they are identical in a particular band
- Hereafter, we assume that "same bucket" means "identical in that band"
- Assumption needed only to simplify analysis, not for correctness of algorithm

Example bands

Assume the following case:

Assumptions

- There are enough buckets that columns are unlikely to hash to the same bucket unless they are identical in a particular band
- Hereafter, we assume that "same bucket" means "identical in that band"
- Assumption needed only to simplify analysis, not for correctness of algorithm

Example bands

Assume the following case:

- Suppose 100,000 columns of M (100k docs)

Assumptions

- There are enough buckets that columns are unlikely to hash to the same bucket unless they are identical in a particular band
- Hereafter, we assume that "same bucket" means "identical in that band"
- Assumption needed only to simplify analysis, not for correctness of algorithm

Example bands

Assume the following case:

- Suppose 100,000 columns of M ($100k$ docs)
- Signatures of 100 integers (rows)

Assumptions

- There are enough buckets that columns are unlikely to hash to the same bucket unless they are identical in a particular band
- Hereafter, we assume that “same bucket” means “identical in that band”
- Assumption needed only to simplify analysis, not for correctness of algorithm

Example bands

Assume the following case:

- Suppose 100,000 columns of M (100k docs)
- Signatures of 100 integers (rows)
- Therefore, signatures take $40Mb$

Assumptions

- There are enough buckets that columns are unlikely to hash to the same bucket unless they are identical in a particular band
- Hereafter, we assume that "same bucket" means "identical in that band"
- Assumption needed only to simplify analysis, not for correctness of algorithm

Example bands

Assume the following case:

- Suppose 100,000 columns of M ($100k$ docs)
- Signatures of 100 integers (rows)
- Therefore, signatures take $40Mb$
- Choose $b = 20$ bands of $r = 5$ integers/band

Goal: Find pairs of documents that are at least $s = 0.8$

C_1, C_2 are 80% similar

Find pairs of $\geq s = 0.8$ similarity, set $b = 20, r = 5$

- Assume: $sim(C_1, C_2) = 0.8$. Since $sim(C_1, C_2) \geq s$, we want C_1, C_2 to be a candidate pair: We want them to hash to at least 1 common bucket (at least one band is identical)

C_1, C_2 are 80% similar

Find pairs of $\geq s = 0.8$ similarity, set $b = 20, r = 5$

- Assume: $sim(C_1, C_2) = 0.8$. Since $sim(C_1, C_2) \geq s$, we want C_1, C_2 to be a candidate pair: We want them to hash to at least 1 common bucket (at least one band is identical)
- Probability C_1, C_2 identical in one particular band: $(0.8)^5 = 0.328$

C_1, C_2 are 80% similar

Find pairs of $\geq s = 0.8$ similarity, set $b = 20, r = 5$

- Assume: $sim(C_1, C_2) = 0.8$. Since $sim(C_1, C_2) \geq s$, we want C_1, C_2 to be a candidate pair: We want them to hash to at least 1 common bucket (at least one band is identical)
- Probability C_1, C_2 identical in one particular band: $(0.8)^5 = 0.328$
  - Probability C_1, C_2 are not similar in all of the 20 bands:

$$(1 - 0.328)^{20} = 0.00035$$

C_1, C_2 are 80% similar

Find pairs of $\geq s = 0.8$ similarity, set $b = 20, r = 5$

- Assume: $sim(C_1, C_2) = 0.8$. Since $sim(C_1, C_2) \geq s$, we want C_1, C_2 to be a candidate pair: We want them to hash to at least 1 common bucket (at least one band is identical)
- Probability C_1, C_2 identical in one particular band: $(0.8)^5 = 0.328$
  - Probability C_1, C_2 are not similar in all of the 20 bands:

$$(1 - 0.328)^{20} = 0.00035$$

  - i.e., about 1/3000th of the 80%-similar column pairs are false negatives (we miss them)

C_1, C_2 are 80% similar

Find pairs of $\geq s = 0.8$ similarity, set $b = 20, r = 5$

- Assume: $sim(C_1, C_2) = 0.8$. Since $sim(C_1, C_2) \geq s$, we want C_1, C_2 to be a candidate pair: We want them to hash to at least 1 common bucket (at least one band is identical)
- Probability C_1, C_2 identical in one particular band: $(0.8)^5 = 0.328$
  - Probability C_1, C_2 are not similar in all of the 20 bands:

$$(1 - 0.328)^{20} = 0.00035$$

  - i.e., about 1/3000th of the 80%-similar column pairs are false negatives (we miss them)
  - We would find 99.965% pairs of truly similar documents

C_1, C_2 are 30% similar

Find pairs of $\geq s = 0.8$ similarity, set $b = 20, r = 5$

- Assume: $sim(C_1, C_2) = 0.3$. Since $sim(C_1, C_2) < s$, we want C_1, C_2 to hash to no common buckets (all bands should be different)

C_1, C_2 are 30% similar

Find pairs of $\geq s = 0.8$ similarity, set $b = 20, r = 5$

- Assume: $sim(C_1, C_2) = 0.3$. Since $sim(C_1, C_2) < s$, we want C_1, C_2 to hash to no common buckets (all bands should be different)
- Probability C_1, C_2 identical in one particular band: $(0.3)^5 = 0.00243$

C_1, C_2 are 30% similar

Find pairs of $\geq s = 0.8$ similarity, set $b = 20, r = 5$

- Assume: $sim(C_1, C_2) = 0.3$. Since $sim(C_1, C_2) < s$, we want C_1, C_2 to hash to no common buckets (all bands should be different)
- Probability C_1, C_2 identical in one particular band: $(0.3)^5 = 0.00243$
  - Probability C_1, C_2 are identical in at least 1 of 20 bands

$$1 - (1 - 0.0.00243)^{20} = 0.0474$$

C_1, C_2 are 30% similar

Find pairs of $\geq s = 0.8$ similarity, set $b = 20, r = 5$

- Assume: $sim(C_1, C_2) = 0.3$. Since $sim(C_1, C_2) < s$, we want C_1, C_2 to hash to no common buckets (all bands should be different)
- Probability C_1, C_2 identical in one particular band: $(0.3)^5 = 0.00243$
  - Probability C_1, C_2 are identical in at least 1 of 20 bands

$$1 - (1 - 0.0.00243)^{20} = 0.0474$$

  - In other words, approximately 4.74% pairs of docs with similarity 0.3% end up becoming candidate pairs

C_1, C_2 are 30% similar

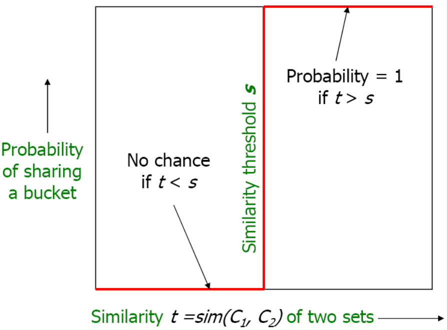
Find pairs of $\geq s = 0.8$ similarity, set $b = 20, r = 5$

- Assume: $sim(C_1, C_2) = 0.3$. Since $sim(C_1, C_2) < s$, we want C_1, C_2 to hash to no common buckets (all bands should be different)
- Probability C_1, C_2 identical in one particular band: $(0.3)^5 = 0.00243$
  - Probability C_1, C_2 are identical in at least 1 of 20 bands

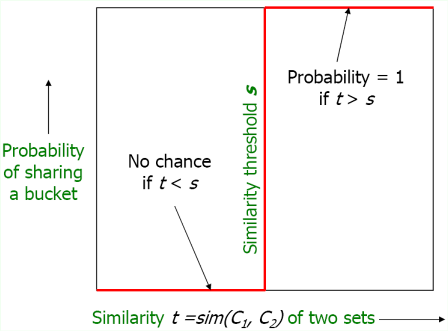
$$1 - (1 - 0.0.00243)^{20} = 0.0474$$

  - In other words, approximately 4.74% pairs of docs with similarity 0.3% end up becoming candidate pairs
  - They are false positives since we will have to examine them (they are candidate pairs) but then it will turn out their similarity is below threshold s

LSH involves a tradeoff

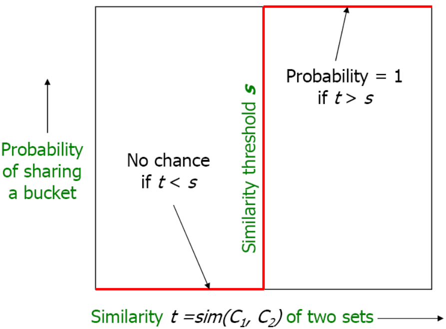


LSH involves a tradeoff



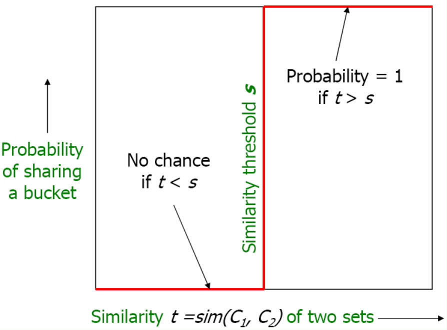
- Pick:

LSH involves a tradeoff



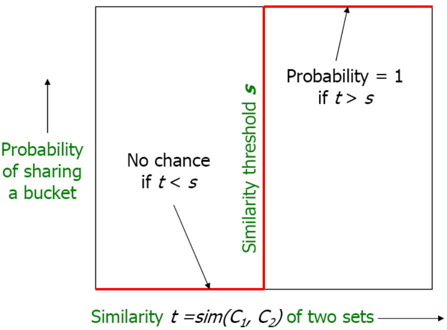
- Pick:
  - The number of Min-Hashes (rows of M)

LSH involves a tradeoff



- Pick:
  - The number of Min-Hashes (rows of M)
  - The number of bands b, and

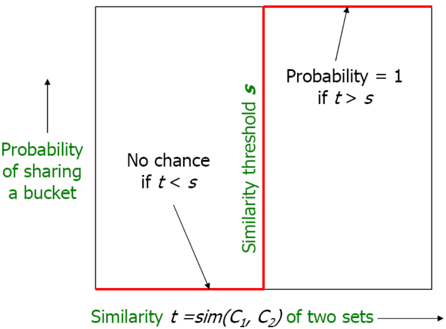
LSH involves a tradeoff



- Pick:
  - The number of Min-Hashes (rows of M)
  - The number of bands b, and
  - The number of rows r per band

  to balance false positives/negatives

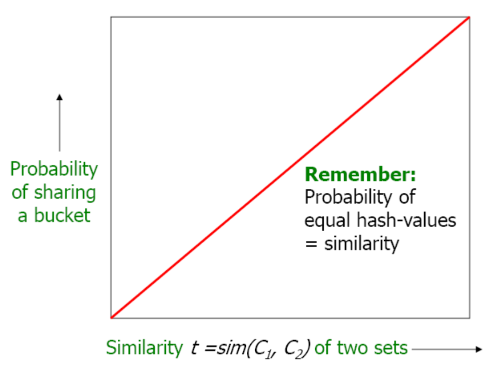
LSH involves a tradeoff



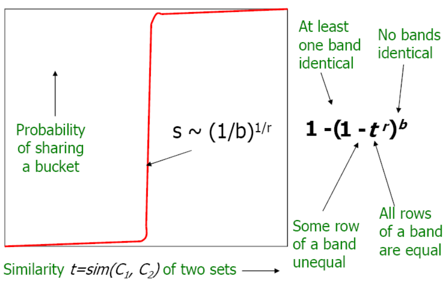
- Pick:
  - The number of Min-Hashes (rows of M)
  - The number of bands b, and
  - The number of rows r per band

  to balance false positives/negatives
- Example: If we had only 15 bands of 5 rows, the number of false positives would go down, but the number of false negatives would go up

What 1 band of 1 row gives you

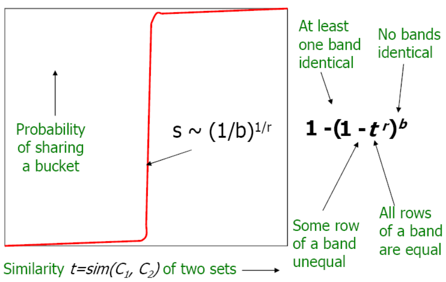


LSH involves a tradeoff

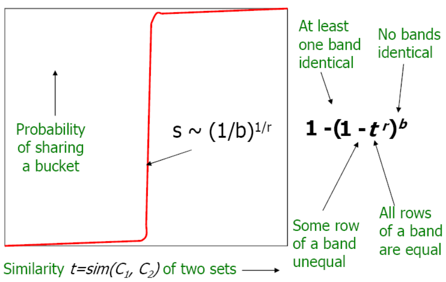


LSH involves a tradeoff

Columns C_1 and C_2 have similarity t



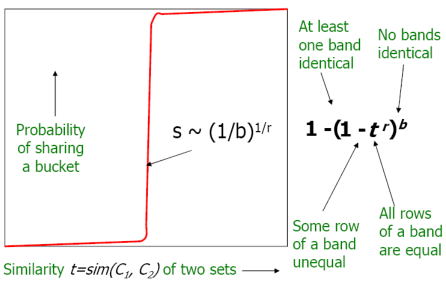
LSH involves a tradeoff



Columns C_1 and C_2 have similarity t

- Pick any band (r rows)

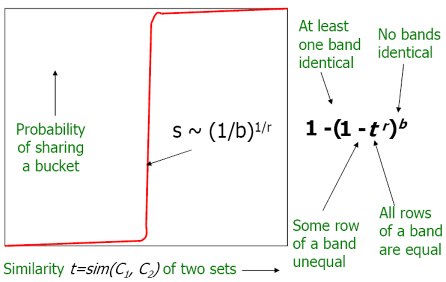
LSH involves a tradeoff



Columns C_1 and C_2 have similarity t

- Pick any band (r rows)
- Prob. that all rows in band equal $= t^r$

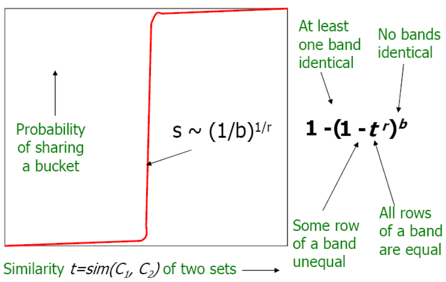
LSH involves a tradeoff



Columns C_1 and C_2 have similarity t

- Pick any band (r rows)
- Prob. that all rows in band equal $= t^r$
- Prob. that some row in band unequal $= 1 - t^r$

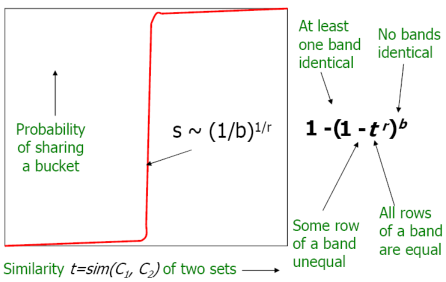
LSH involves a tradeoff



Columns C_1 and C_2 have similarity t

- Pick any band (r rows)
- Prob. that all rows in band equal $= t^r$
- Prob. that some row in band unequal $= 1 - t^r$
- Prob. that no band identical $= (1 - t^r)^b$

LSH involves a tradeoff



Columns C_1 and C_2 have similarity t

- Pick any band (r rows)
- Prob. that all rows in band equal $= t^r$
- Prob. that some row in band unequal $= 1 - t^r$
- Prob. that no band identical $= (1 - t^r)^b$
- Prob. that at least 1 band identical $= = 1 - (1 - t^r)^b$

Analysis of LSH

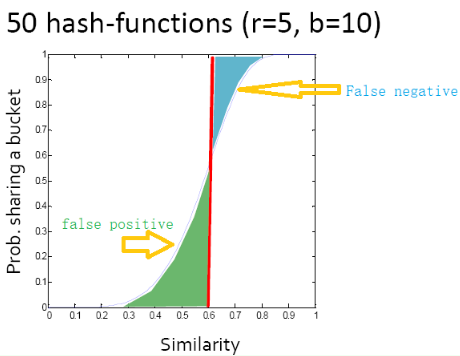
s	$1\text{-}(1\text{-}s^r)^b$
.2	.006
.3	.047
.4	.186
.5	.470
.6	.802
.7	.975
.8	.9996

Prob. that at least 1 band is identical.

Analysis of LSH

s	$1-(1-s^r)^b$
.2	.006
.3	.047
.4	.186
.5	.470
.6	.802
.7	.975
.8	.9996

Prob. that at least 1 band is identical.



Picking r and b to get the best S-curve

LSH summary

- Tune M, b, r to get almost all pairs with similar signatures, but eliminate most pairs that do not have similar signatures

LSH summary

- Tune M, b, r to get almost all pairs with similar signatures, but eliminate most pairs that do not have similar signatures
- Check in main memory that candidate pairs really do have similar signatures

LSH summary

- Tune M, b, r to get almost all pairs with similar signatures, but eliminate most pairs that do not have similar signatures
- Check in main memory that candidate pairs really do have similar signatures
- Optional: In another pass through data, check that the remaining candidate pairs really represent similar documents

Take-aways

- Hash functions
- Bloom filtering
- Locality sensitive hashing
  - Shingling
  - Min-hashing
  - Locality sensitive hashing