# COMP 9313 REVIEW

**Nov 9th 2018: 12:00~4:30**

- ❑ Topic 1: MapReduce (Chapters 2-4)
- ❑ Topic 2: Spark Core and GraphX (Chapters 6 and 7)

**Nov 11th 2018: 5:00~9:30**

- ❑ Topic 3: Finding Similar Items (Chapter 9)
- ❑ Topic 4: Mining Data Streams (Chapter 7)
- ❑ Topic 5: Recommender Systems (Chapter 11)
- ❑ Summary + Outlook

## Exam Questions

- Question 1 MapReduce
  - Part A: MapReduce concepts
  - Part B: MapReduce algorithm design
- Question 2 Spark
  - Part A: Spark concepts
  - Part B: Show output of the given code
  - Part C: Spark algorithm design
    - Spark Core
    - Spark GraphX
- Question 3 Finding Similar Items
  - Shingling, Min Hashing, LSH
- Question 4 Mining Data Streams
  - Sampling, DGIM, Bloom filter
- Question 5 Recommender Systems

# COMP 9313 REVIEW

**20181111**

## Outline

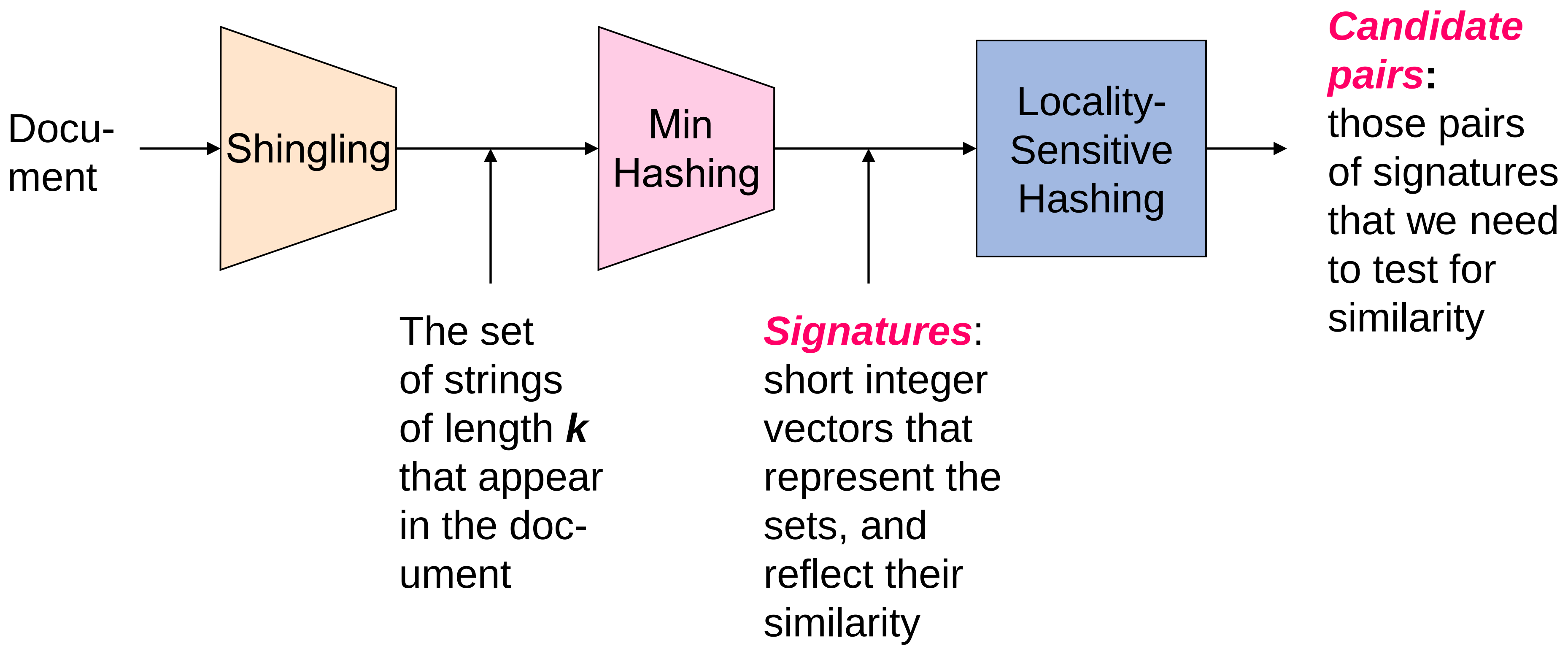
- ❑ Topic 3: Finding Similar Items (Chapter 9)
    - Part A: Shingling
    - Part B: Min Hashing
    - Part C: LSH
- ❑ Topic 4: Mining Data Streams (Chapter 7)
    - Part A: Sampling
    - Part B: DGIM
    - Part C: Bloom filter
- ❑ Topic 5: Recommender Systems (Chapter 11)

# Topic 3: Finding Similar Items

- ***Part A: Shingling***
- Part B: Min Hashing
- Part C: LSH

# Topic 3: Finding Similar Items (Chapter 9)

- The Big Picture



## Shingling

- A *k*-shingle (or *k*-gram) for a document is a sequence of *k* tokens that appears in the doc
  - Tokens can be characters, words or something else, depending on the application
  - Assume tokens = characters for examples

- **Example:** **k=2**; document $\mathbf{D_1}$ = abcab
  Set of 2-shingles: $\mathbf{S(D_1)}$ = {ab, bc, ca}

- Documents that are intuitively similar will have many shingles in common.
  - Example: k=3, “The dog which chased the cat” versus “The dog that chased the cat”.
    - Only 3-shingles replaced are g_w, _wh, whi, hic, ich, ch_, and h_c.

## Question 3 Finding Similar Items

- k-Shingles:

Consider two documents A and B. Each document's number of token is O(n). What is the runtime complexity of computing A and B's k-shingle resemblance (using Jaccard similarity)? Assume that comparison of two k-shingles to assess their equivalence is O(k). Express your answer in terms of n and k.

Answer:

Assuming n >> k,

Time to create shingles = O(n)

Time to find intersection (using brute force algorithm) = $O(kn^2)$

Time to find union = O(n)

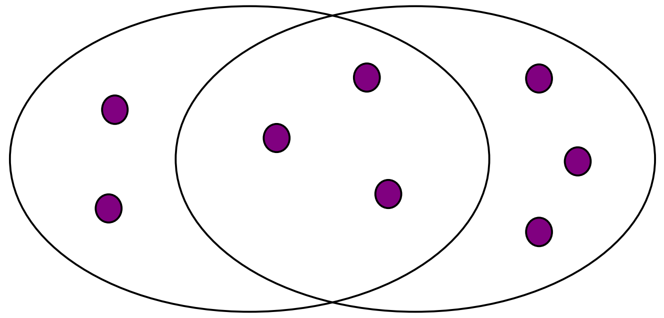
Total time = $(kn^2)$

# Topic 3: Finding Similar Items

- ❑Part A: Shingling
- ❑***Part B: Min Hashing***
- ❑Part C: LSH

# 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:** $\mathbf{C_1}$ = 10111; $\mathbf{C_2}$ = 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**

# From Sets to Boolean Matrices

- **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!**

- **Each document is a column:**

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 |

# Outline: Finding Similar Columns

- **Next Goal: Find similar columns, Small signatures**

- **Naïve approach:**
  - **1) Signatures of columns:** small summaries of columns
  - **2) Examine pairs of signatures** to find similar columns
    - **Essential:** Similarities of signatures and columns are related
  - **3) 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:
  - **(1)** ***h(C)*** is small enough that the signature fits in RAM
  - **(2)** ***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(·)* such that:**
  - If ***sim($C_1$,$C_2$)*** is high, then with high prob. ***h($C_1$) = h($C_2$)***
  - If ***sim($C_1$,$C_2$)*** is low, then with high prob. ***h($C_1$) ≠ h($C_2$)***

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

## Min-Hashing

- Imagine the rows of the boolean matrix permuted under **random permutation** $\pi$

- Define a **"hash" function** $h_\pi(C)$ = the index of the **first** (in the permuted order $\pi$) 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

$$\begin{array}{c|cccc}
1 & 0 & \textcircled{1} & \textcircled{1} & 0 \\
2 & 0 & 0 & 1 & \textcircled{1} \\
3 & \textcircled{1} & 0 & 0 & 0 \\
4 & 0 & 1 & 0 & 1 \\
5 & 0 & 0 & 0 & 1 \\
6 & 1 & 1 & 0 & 0 \\
7 & 0 & 0 & 1 & 0
\end{array}$$

Input Matrix

$$\begin{array}{|c|c|c|c|}
3 & 1 & 1 & 2
\end{array}$$

Signature Matrix

# Four Types of Rows

- **Given cols $C_1$ and $C_2$, rows may be classified as:**

|   | $C_1$ | $C_2$ |
|---|---|---|
| A | 1 | 1 |
| B | 1 | 0 |
| C | 0 | 1 |
| D | 0 | 0 |

  - **a** = # rows of type A, etc.
- **Note: $sim(C_1, C_2) = a/(a+b+c)$**
- **Then: Pr**$[h(C_1) = h(C_2)] = Sim(C_1, C_2)$
  - Look down the cols $C_1$ 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

# Similarity for Signatures

**Permutation π**

| | | |
|---|---|---|
| 2 | 4 | 3 |
| 3 | 2 | 4 |
| 7 | 1 | 7 |
| 6 | 3 | 2 |
| 1 | 6 | 6 |
| 5 | 7 | 1 |
| 4 | 5 | 5 |

**Input matrix (Shingles x Documents)**

| | | | |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 |

**Signature matrix *M***

| | | | |
|---|---|---|---|
| 2 | 1 | 2 | 1 |
| 2 | 1 | 4 | 1 |
| 1 | 2 | 1 | 2 |

**Similarities:**

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

## 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 **$\sim$100 bytes!**
- **We achieved our goal! We "compressed" long bit vectors into short signatures**

# Implementation Trick

- **Permuting rows even once is prohibitive**
- **Row hashing!**
  - Pick **K = 100** hash functions $k_i$
  - Ordering under $k_i$ gives a random row permutation!
- **One-pass implementation**
  - For each column $C$ and hash-func. $k_i$ keep a "slot" for the min-hash value
  - Initialize all $sig(C)[i] = \infty$
  - **Scan rows looking for 1s**
    - Suppose row $j$ has 1 in column $C$
    - Then for each $k_i$ :
      - If $k_i(j) < sig(C)[i]$, then $sig(C)[i] \leftarrow k_i(j)$

**How to pick a random hash function h(x)?**
**Universal hashing:**
$h_{a,b}(x)=((a \cdot x+b) \bmod p) \bmod N$
where:
a,b … random integers
p … prime number (p > N)

# Implementation Example

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

0. Initialize all ***sig(C)[i] = ∞***

|  | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| $h_1$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| $h_2$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ |

- Row 0: we see that the values of $h_1(0)$ and $h_2(0)$ are both 1, thus $sig(S_1)[0] = 1$, $sig(S_1)[1] = 1$, $sig(S_4)[0] = 1$, $sig(S_4)[1] = 1$,

|  | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| $h_1$ | 1 | $\infty$ | $\infty$ | 1 |
| $h_2$ | 1 | $\infty$ | $\infty$ | 1 |

- Row 1, we see $h_1(1) = 2$ and $h_2(1) = 4$, thus $sig(S_3)[0] = 2$, $sig(S_3)[1] = 4$

|  | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| $h_1$ | 1 | $\infty$ | 2 | 1 |
| $h_2$ | 1 | $\infty$ | 4 | 1 |

# Implementation Example

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

- Row 2: $h_1(2) = 3$ and $h_2(2) = 2$, thus sig($S_2$)[0] = 3, sig($S_2$)[1] = 2, no update

| | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| $h_1$ | 1 | 3 | 2 | 1 |
| $h_2$ | 1 | 2 | 4 | 1 |

- Row 3: $h_1(3) = 4$ and $h_2(3) = 0$, update sig($S_1$)[1] = 0, sig($S_3$)[1] = 0, sig($S_4$)[1]

| | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| $h_1$ | 1 | 3 | 2 | 1 |
| $h_2$ | 0 | 2 | 0 | 0 |

- Row 4: $h_1(4) = 0$ and $h_2(4) = 3$, update sig($S_3$)[0] = 0,

| | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| $h_1$ | 1 | 3 | 0 | 1 |
| $h_2$ | 0 | 2 | 0 | 0 |

# Implementation Example

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

|  | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| $h_1$ | 1 | 3 | 0 | 1 |
| $h_2$ | 0 | 2 | 0 | 0 |

- We can estimate the Jaccard similarities of the underlying sets from this signature matrix.
  - Signature matrix: SIM($S_1$, $S_4$) = 1.0
  - Jaccard Similarity: SIM($S_1$, $S_4$) = 2/3

# Implementation Practice

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

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

|  | Sig1 | Sig2 |
|---|---|---|
| $h(1) = 1$ | $\infty$ | $\infty$ |
| $g(1) = 3$ | $\infty$ | $\infty$ |
| $h(1) = 1$ | 1 | $\infty$ |
| $g(1) = 3$ | 3 | $\infty$ |
| $h(2) = 2$ | 1 | 2 |
| $g(2) = 0$ | 3 | 0 |
| $h(3) = 3$ | 1 | 2 |
| $g(3) = 2$ | 2 | 0 |
| $h(4) = 4$ | 1 | 2 |
| $g(4) = 4$ | 2 | 0 |
| $h(5) = 0$ | 1 | 0 |
| $g(5) = 1$ | 2 | 0 |

# Question 3 Finding Similar Items

- MinHash:

We want to compute min-hash signature for two columns, $C_1$ and $C_2$ using two pseudo-random permutations of columns using the following function:

$h_1(n) = 3n + 2 \bmod 7$

$h_2(n) = 2n - 1 \bmod 7$

| Row | $C_1$ | $C_2$ |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |
| 2 | 0 | 1 |
| 3 | 0 | 0 |
| 4 | 1 | 1 |
| 5 | 1 | 1 |
| 6 | 1 | 0 |

Here, n is the row number in original ordering. Instead of explicitly reordering the columns for each hash function, we use the implementation discussed in class, in which we read each data in a column once in a sequential order, and update the min hash signatures as we pass through them.

Complete the steps of the algorithm and give the resulting signatures for $C_1$ and $C_2$.

# Solution

| | Sig1 | Sig2 |
|---|---|---|
| $h1(1) = 1$ | $\infty$ | $\infty$ |
| $h2(1) = 3$ | $\infty$ | $\infty$ |
| $h1(0) = 1$ | $\infty$ | 2 |
| $h2(0) = 3$ | $\infty$ | 6 |
| $h1(1) = 5$ | 5 | 2 |
| $h2(1) = 1$ | 1 | 6 |
| $h1(2) = 1$ | 5 | 1 |
| $h2(2) = 3$ | 1 | 3 |
| $h1(4) = 0$ | 0 | 0 |
| $h2(4) = 0$ | 0 | 0 |

# Topic 3: Finding Similar Items

- ❑Part A: Shingling
- ❑Part B: Min Hashing
- ❑***Part C: LSH***

## LSH: First Cut

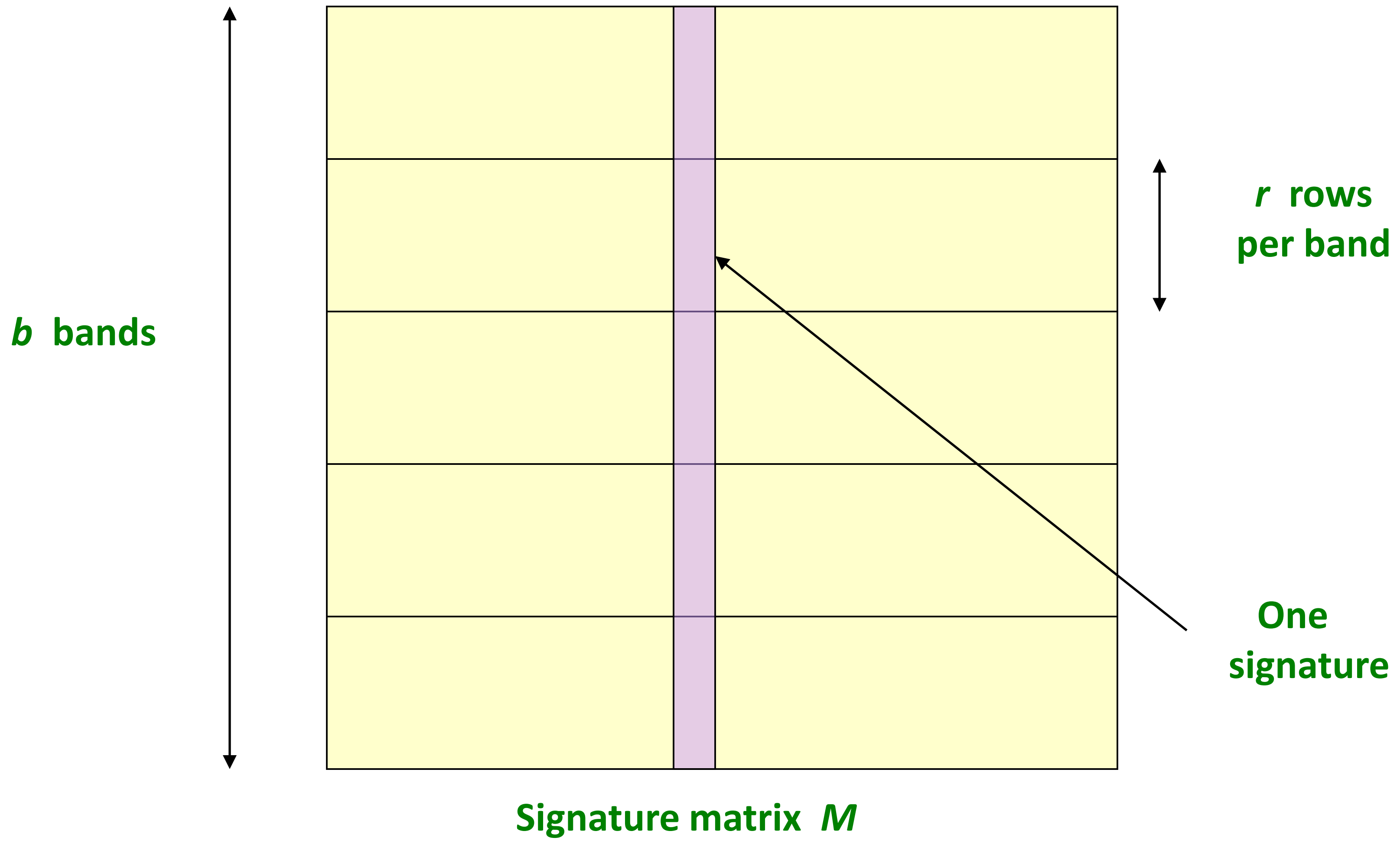
| | | | |
|---|---|---|---|
| 2 | 1 | 4 | 1 |
| 1 | 2 | 1 | 2 |
| 2 | 1 | 2 | 1 |

- **Goal:** Find documents with Jaccard similarity at least ***s*** (for some similarity threshold, 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**

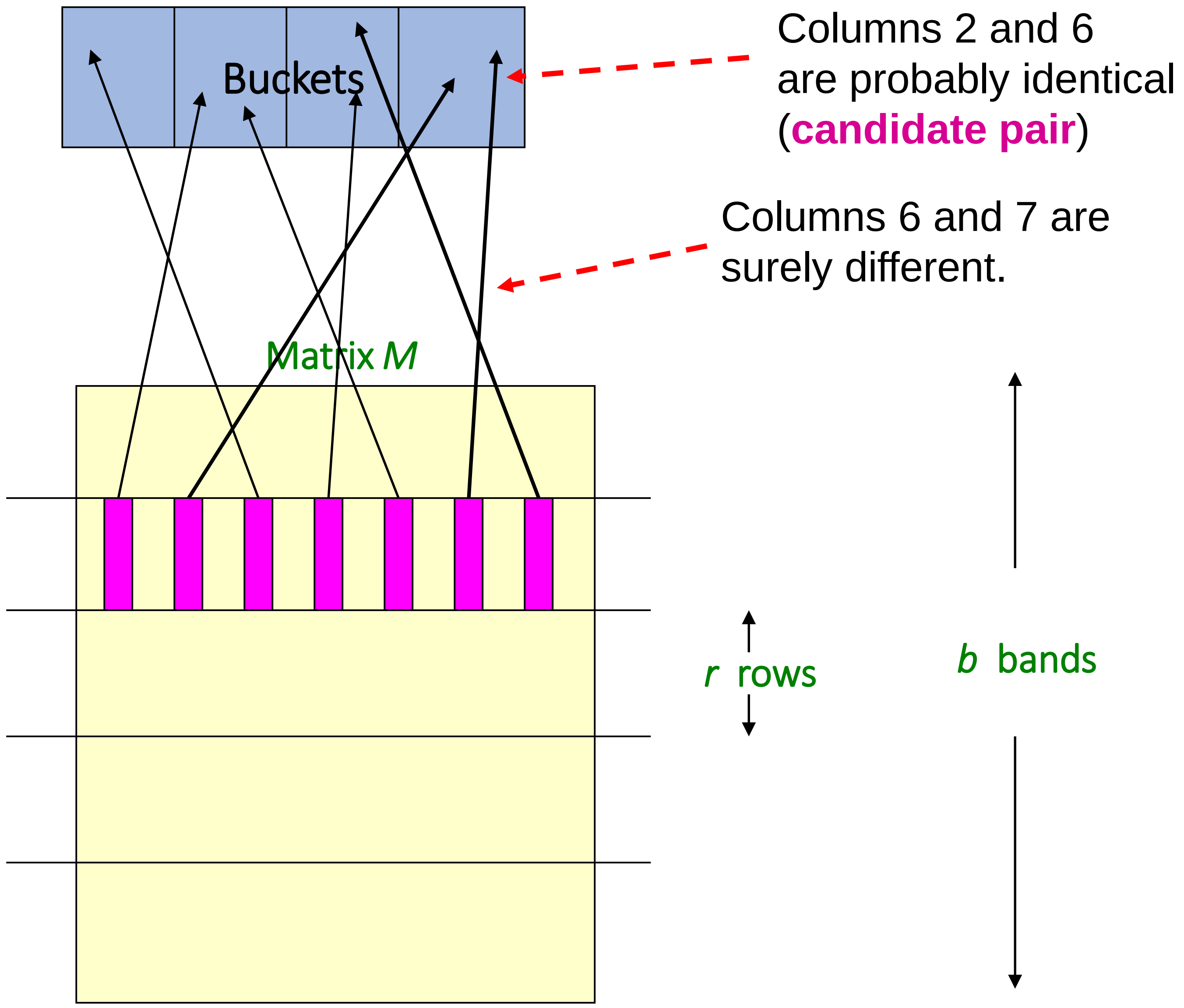
# Partition *M* into *b* Bands



## Partition M into Bands

- 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
  - Make ***k*** as large as possible
- ***Candidate*** column pairs are those that hash to the same bucket for $\geq$ **1** band
- Tune ***b*** and ***r*** to catch most similar pairs, but few non-similar pairs

# Hashing Bands



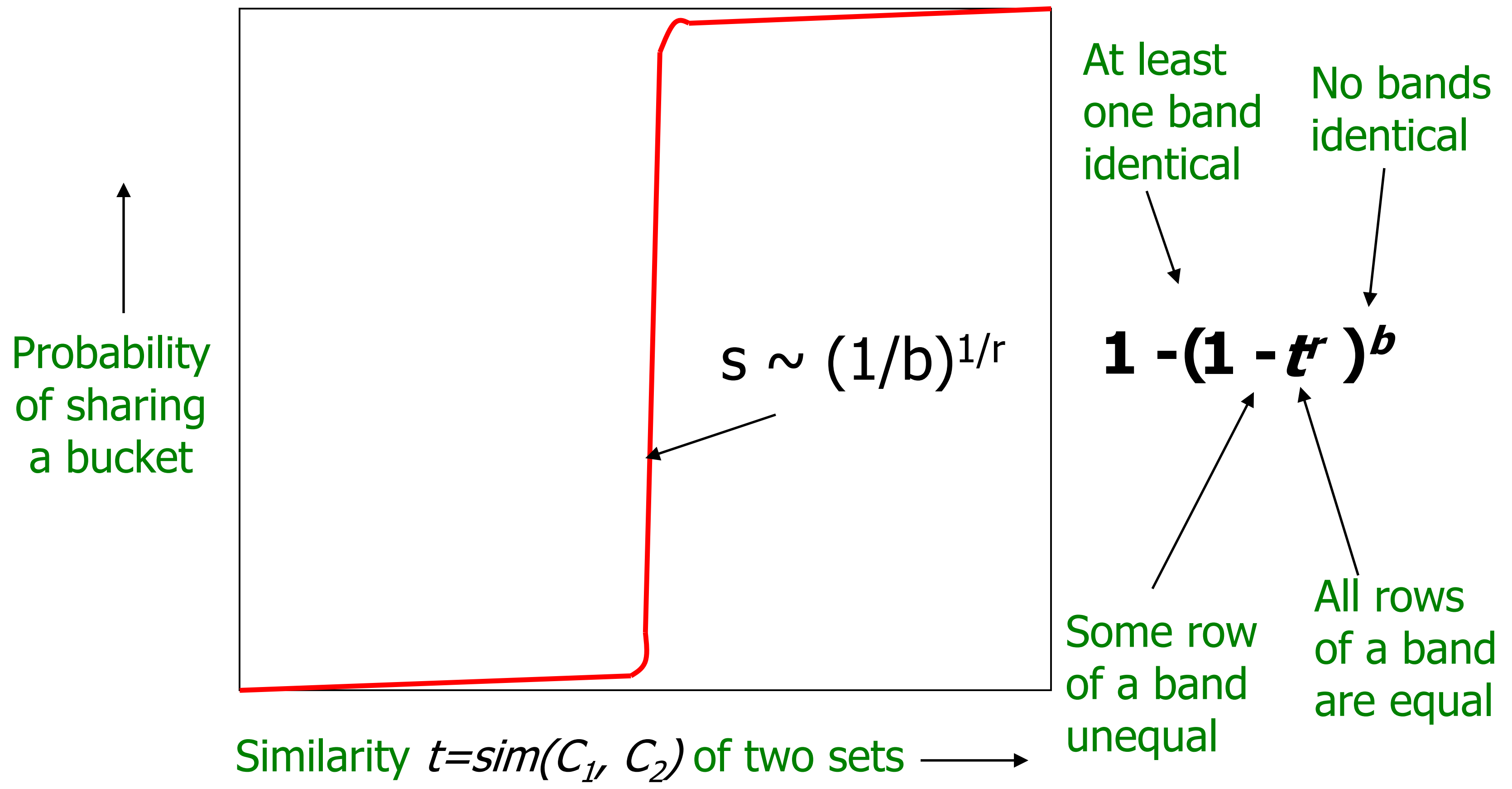
Columns 2 and 6 are probably identical (**candidate pair**)

Columns 6 and 7 are surely different.

## *b* bands, *r* rows/band

- The probability that the min-hash signatures for the documents agree in any one particular row of the signature matrix is $t$ ($sim(C_1, C_2)$ )
- 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 = $1 - (1 - t^r)^b$

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



$$\mathbf{1 - (1 - t^r)^b}$$

At least one band identical

No bands identical

Some row of a band unequal

All rows of a band are equal

# Question 3 Finding Similar Items

- LSH:

Suppose we wish to find similar sets, and we do so by minhashing the sets 10 times and then applying locality-sensitive hashing using 5 bands of 2 rows (minhash values) each. If two sets had Jaccard similarity 0.6, what is the probability that they will be identified in the locality-sensitive hashing as candidates (i.e. they hash at least once to the same bucket)? You may assume that there are no coincidences, where two unequal values hash to the same bucket. A correct expression is sufficient: you need not give the actual number.

Answer: $1-(1-0.6^2)^5=0.893$

# Topic 4: Mining Data Streams

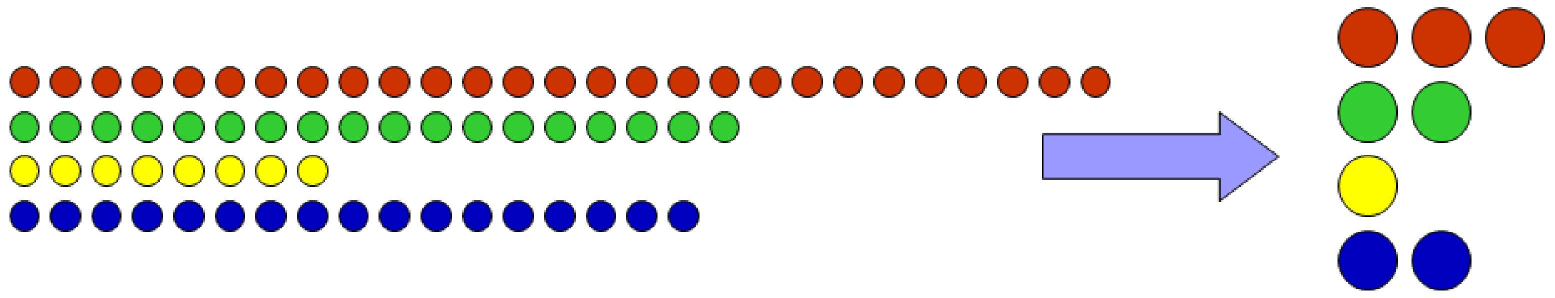
- ❑***Part A: Sampling***
- ❑Part B: DGIM
- ❑Part C: Bloom filter

# Topic 4: Mining Data Streams (Chapter 7)

- Types of queries one wants on answer on a data stream: (we'll learn these today)
  - Sampling data from a stream
    - Construct a random sample
  - Queries over sliding windows
    - Number of items of type x in the last *k* elements of the stream
  - Filtering a data stream
    - Select elements with property x from the stream

# Sampling Data Streams

- Since we can not store the entire stream, one obvious approach is to store a **sample**



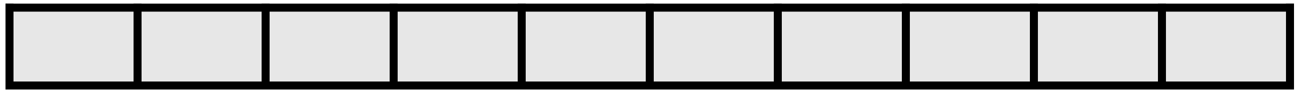
- Two different problems:
  - **(1)** Sample a **fixed proportion** of elements in the stream (say 1 in 10)
    - As the stream grows the sample also gets bigger
  - **(2)** Maintain a **random sample of fixed size** over a potentially infinite stream
    - As the stream grows, the sample is of fixed size
    - At any "time" ***t*** we would like a random sample of ***s*** elements
      - **What is the property of the sample we want to maintain?** For all time steps ***t***, each of ***t*** elements seen so far has equal probability of being sampled

## Sampling a Fixed Proportion

- Problem 1: Sampling fixed proportion
- Scenario: Search engine query stream
  - **Stream of tuples:** (user, query, time)
  - **Answer questions such as: How often did a user run the same query in a single days**
  - Have space to store **1/10th** of query stream
- **Naïve solution:**
  - Generate a random integer in **[0..9]** for each query
  - Store the query if the integer is **0**, otherwise discard

## Generalized Problem and Solution

- Problem: Give a data stream, take a sample of fraction a/b.
- Stream of tuples with keys:
  - Key is some subset of each tuple's components
    - e.g., tuple is (user, search, time); key is **user**
  - Choice of key depends on application
- To get a sample of a/b fraction of the stream:
  - Hash each tuple's key uniformly into ***b*** buckets
  - Pick the tuple if its hash value is at most ***a***



**How to generate a 30% sample?**

Hash into b=10 buckets, take the tuple if it hashes to one of the first 3 buckets

# Maintaining a Fixed-size Sample

- Problem 2: Fixed-size sample
- Suppose we need to maintain a random sample S of size exactly s tuples
  - E.g., main memory size constraint
- **Why?** Don't know length of stream in advance
- Suppose at time *n* we have seen *n* items
  - Each item is in the sample *S* with equal prob. *s/n*

Note that the same item is treated as different tuples at different timestamps

**How to think about the problem: say s = 2**

**Stream:** a x c y z k c d e g…

At **n= 5,** each of the first 5 tuples is included in the sample **S** with equal prob.

At **n= 7,** each of the first 7 tuples is included in the sample **S** with equal prob.

**Impractical solution would be to store all the *n* tuples seen so far and out of them pick *s* at random**

# Solution: Fixed Size Sample

- **Algorithm (a.k.a. Reservoir Sampling)**
  - Store all the first ***s*** elements of the stream to ***S***
  - Suppose we have seen ***n-1*** elements, and now the ***n***$^{th}$ element arrives (***n > s***)
    - With probability ***s/n***, keep the ***n***$^{th}$ element, else discard it
    - If we picked the ***n***$^{th}$ element, then it replaces one of the ***s*** elements in the sample ***S***, picked uniformly at random

- **Claim:** This algorithm maintains a sample ***S*** with the desired property:
  - After ***n*** elements, the sample contains each element seen so far with probability ***s/n***

# Topic 4: Mining Data Streams

- ❑Part A: Sampling
- ❑***Part B: DGIM***
- ❑Part C: Bloom filter

## The Datar-Gionis-Indyk-Motwani (DGIM) Algorithm

- Maintaining Stream Statistics over Sliding Windows (SODA'02)

- DGIM solution that does not assume uniformity

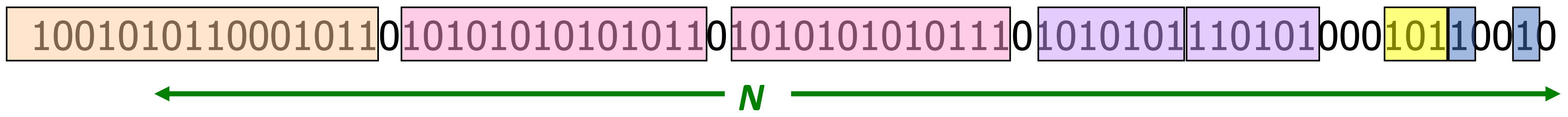
- We store $\boldsymbol{O}(\log^{\mathbf{2}}\boldsymbol{N})$ bits per stream

- Solution gives approximate answer, never off by more than 50%
    - Error factor can be reduced to any fraction > 0, with more complicated algorithm and proportionally more stored bits

# Fixup: DGIM Algorithm

- **Idea:** Instead of summarizing fixed-length blocks, summarize blocks with specific number of **1s**:
  - Let the block ***sizes*** (number of **1s**) increase exponentially

- When there are few 1s in the window, block sizes stay small, so errors are small

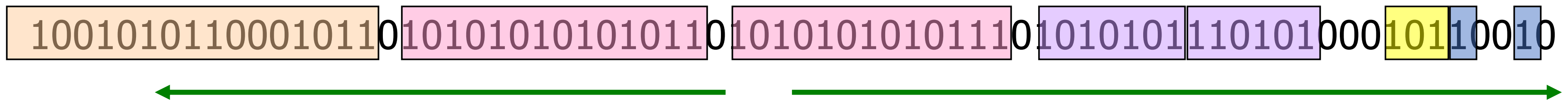


- Timestamps:
  - Each bit in the stream has a timestamp, starting from **1**, **2,** …
  - Record timestamps modulo ***N*** (**the window size**), so we can represent any **relevant** timestamp in $\boldsymbol{O}(\log_2 \boldsymbol{N})$ bits
    - E.g., given the windows size 40 (***N***), timestamp 123 will be recorded as 3, and thus the encoding is on 3 rather than 123

# DGIM: Buckets

- A bucket in the DGIM method is a record consisting of:
  - (A) The timestamp of its end [$\boldsymbol{O}(\log \boldsymbol{N})$ bits]
  - (B) The number of 1s between its beginning and end [$\boldsymbol{O}(\log\log \boldsymbol{N})$ bits]

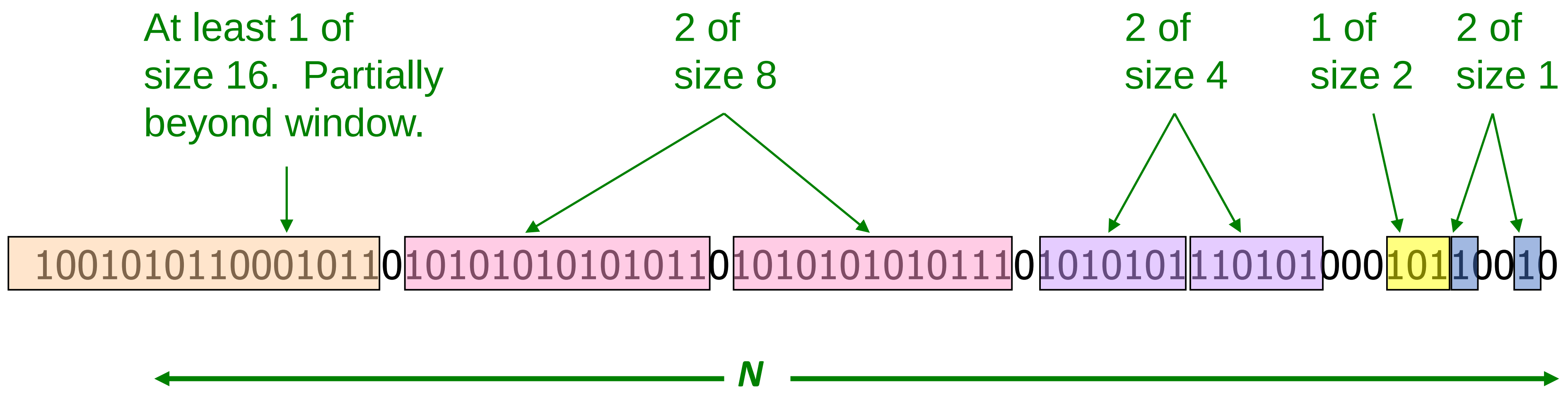
- Constraint on buckets:
  - Number of 1s must be a power of 2
  - That explains the $\boldsymbol{O}(\log\log \boldsymbol{N})$ in (B) above



## Representing a Stream by Buckets

- The right end of a bucket is always a position with a 1
- Every position with a 1 is in some bucket
- Either **one** or **two** buckets with the same **power-of-2 number** of **1s**
- Buckets do not overlap in timestamps
- **Buckets are sorted by size**
  - Earlier buckets are not smaller than later buckets
- Buckets disappear when their end-time is **>** ***N*** time units in the past

# Example: Bucketized Stream



Three properties of buckets that are maintained:

Either **one** or **two** buckets with the same power-of-2 number of 1s

Buckets do not overlap in timestamps

Buckets are sorted by size

## Updating Buckets

- When a new bit comes in, drop the last (oldest) bucket if its end-time is prior to ***N*** time units before the current time

- 2 cases: Current bit is **0** or **1**

- If the current bit is 0: no other changes are needed
- If the current bit is 1:
  - (1) Create a new bucket of size 1, for just this bit
    - End timestamp = current time
  - (2) If there are now three buckets of size 1, combine the oldest two into a bucket of size 2
  - (3) If there are now three buckets of size 2, combine the oldest two into a bucket of size 4
  - (4) And so on …

# Example: Updating Buckets

**Current state of the stream:**

10010101100010110101010101010101101010101010101111010101011101010001011001 0

**Bit of value 1 arrives**

001010110001011010101010101010110101010101010111101010101110101000101100101

**Two white buckets get merged into a yellow bucket**

00101011000101101010101010101011010101010101011110101010111010100010110010 1

**Next bit 1 arrives, new orange white is created, then 0 comes, then 1:**

0101100010110101010101010101101010101010101111010101011101010001011001011 01

**Buckets get merged…**

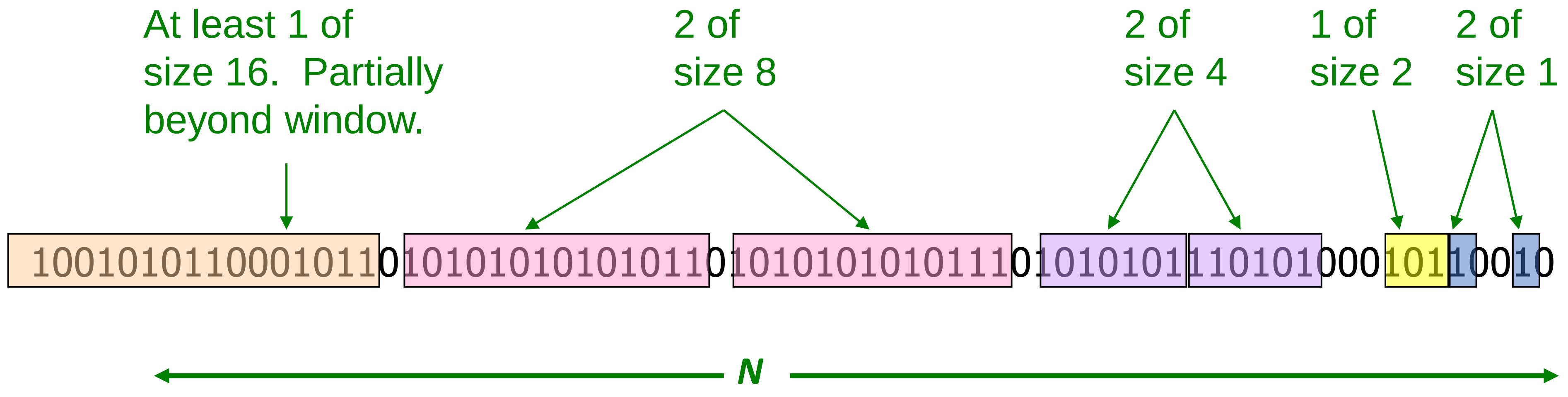
0101100010110101010101010101101010101010101111010101011101010001011001011 01

**State of the buckets after merging**

010110001011010101010101010110101010101010111101010101110101000101100101101

# How to Query?

- To estimate the number of 1s in the most recent N bits:
  - Sum the sizes of all buckets but the last
    - (note “size” means the number of 1s in the bucket)
  - Add half the size of the last bucket
- Remember: We do not know how many 1s of the last bucket are still within the wanted window
- Example:



# Question 4 Mining Data Streams

- DGIM

Suppose we are maintaining a count of 1s using the DGIM method. We represent a bucket by (i, t), where i is the number of 1s in the bucket and t is the bucket timestamp (time of the most recent 1).

Consider that the current time is 200, window size is 60, and the current list of buckets is: (16, 148) (8, 162) (8, 177) (4, 183) (2, 192) (1, 197) (1, 200). At the next ten clocks, 201 through 210, the stream has 0101010101. What will the sequence of buckets be at the end of these ten inputs?

## Solution

There are 5 1s in the stream. Each one will update to windows to be: [each step 2 marks]

- (1) (16, 148)(8, 162)(8, 177)(4, 183)(2, 192)(1, 197)(1, 200), (1, 202)

=> (16, 148)(8, 162)(8, 177)(4, 183)(2, 192)(2, 200), (1, 202)

- (2) (16, 148)(8, 162)(8, 177)(4, 183)(2, 192)(2, 200), (1, 202), (1, 204)
- (3) (16, 148)(8, 162)(8, 177)(4, 183)(2, 192)(2, 200), *(1, 202) (1, 204),* (1, 206)

=> (16, 148)(8, 162)(8, 177)(4, 183), *(2, 192)(2, 200),* (2, 204), (1, 206)

=> (16, 148)(8, 162)(8, 177)(4, 183)(4, 200), (2, 204), (1, 206)

- (4) Windows Size is 60, so (16,148) should be dropped due to 208-60=148.

(16, 148)(8, 162)(8, 177)(4, 183)(4, 200), (2, 204), (1, 206), (1, 208)

=> (8, 162)(8, 177)(4, 183)(4, 200), (2, 204), (1, 206), (1, 208)

- (5) (8, 162)(8, 177)(4, 183)(4, 200), (2, 204), (1, 206), (1, 208), (1, 210)

=> (8, 162)(8, 177)(4, 183)(4, 200), (2, 204), (2, 208), (1, 210)

## Topic 4: Mining Data Streams

- ❑Part A: Sampling
- ❑Part B: DGIM
- ❑***Part C: Bloom filter***

# Bloom Filter

- Consider: **$|S| = m$, $|B| = n$**
- Use $k$ independent hash functions $h_1, \ldots, h_k$
- **Initialization:**
  - Set **B** to all **0s**
  - Hash each element $s \in S$ using each hash function $h_i$, set $B[h_i(s)] = 1$ (for each $i = 1, .., k$)
- **Run-time:**
  - When a stream element with key $x$ arrives
    - If $B[h_i(x)] = 1$ for all $i = 1, \ldots, k$ then declare that $x$ is in $S$
      - That is, $x$ hashes to a bucket set to **1** for every hash function $h_i(x)$
    - Otherwise discard the element $x$

# Bloom Filter Example

- Consider 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.
- The 10-bit array is initialized as below

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

- 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 | 0 | 0 | 0 | 0 | 1 |

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

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

- Query $y_0$ with $H(y_0)$ = {0, 4, 8} => ???
- Query $y_1$ with $H(y_1)$ = {1, 5, 8} => ???

**False positive!**

- Another Example: https://llimllib.github.io/bloomfilter-tutorial/

# Bloom Filter – Analysis

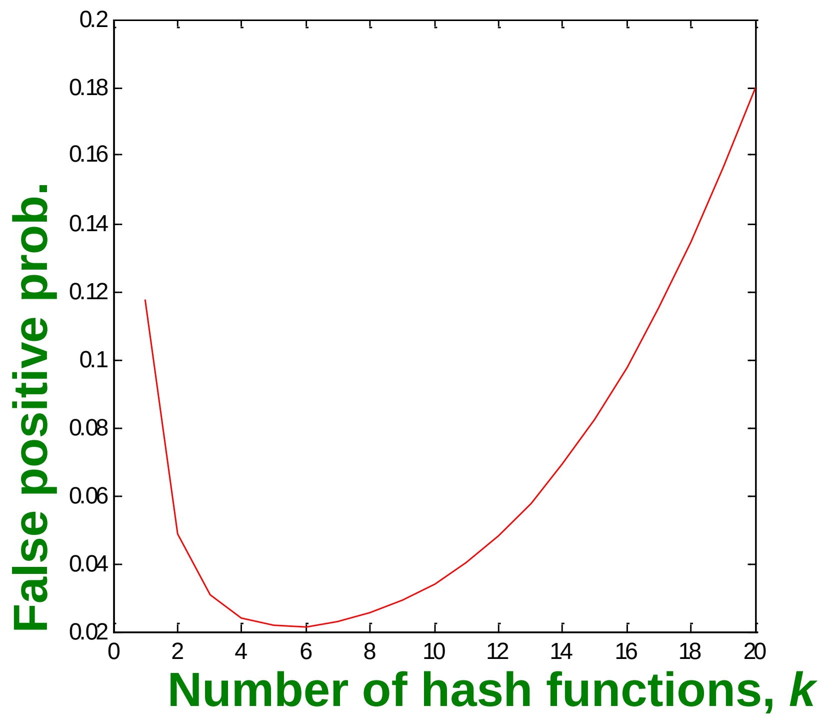
- **What fraction of the bit vector B are 1s?**
  - Throwing $k \cdot m$ darts at $n$ targets
  - So fraction of **1**s is $(1 - e^{-km/n})$

- But we have $k$ independent hash functions and we only let the element $x$ through **if all** $k$ hash element $x$ to a bucket of value **1**

- So, false **positive probability =** $(1 - e^{-km/n})^k$

## Bloom Filter – Analysis (2)

- ***m* = 1 billion, *n* = 8 billion**
  - **k = 1**: $(1 - e^{-1/8})$ = **0.1175**
  - **k = 2**: $(1 - e^{-1/4})^2$ = **0.0493**



- **What happens as we keep increasing *k*?**

- "Optimal" value of ***k***: ***n/m* ln(2)**
  - **In our case:** Optimal **k = 8 ln(2) = 5.54 ≈ 6**
    - **Error at k = 6**: $(1 - e^{-1/6})^2$ = **0.0235**

# Bloom Filter: Wrap-up

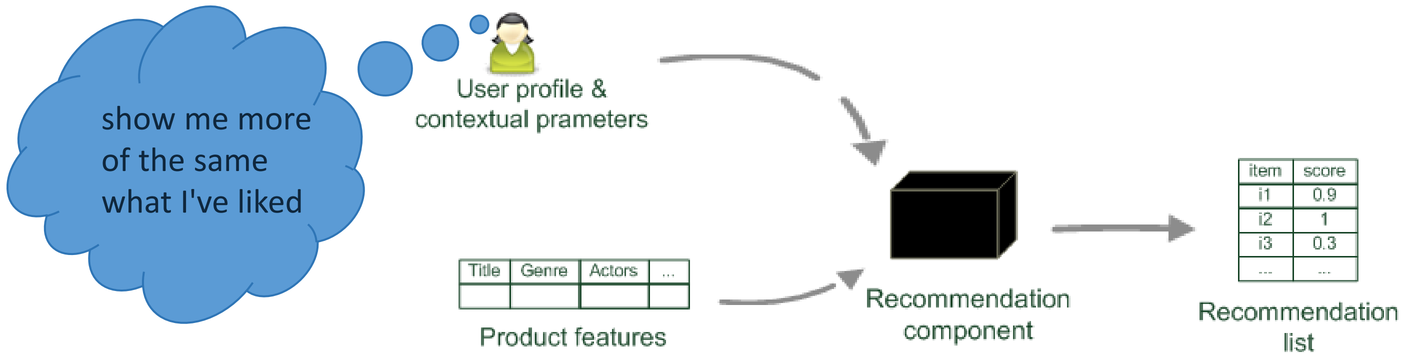
- Bloom filters guarantee no false negatives, and use limited memory
  - Great for pre-processing before more expensive checks

- Suitable for hardware implementation
  - Hash function computations can be parallelized

- Is it better to have **1** big **B** or ***k*** small **B**s?
  - **It is the same:** $(1 - e^{-km/n})^k$ vs. $(1 - e^{-m/(n/k)})^k$
  - But keeping **1 big B** is simpler

# Topic 5: Recommender Systems

# Content-based Recommendation


show me more of the same what I've liked
User profile & contextual prameters
Title | Genre | Actors | ...
Product features
Recommendation component
item | score
i1 | 0.9
i2 | 1
i3 | 0.3
... | ...
Recommendation list

## Content-based Recommendations

- **Main idea:** Recommend items to customer ***x*** similar to previous items rated highly by ***x***

- What do we need:
  - Some information about the available items such as the genre ("content")
  - Some sort of *user profile* describing what the user likes (the preferences)

- ***Example:***
  - Movie recommendations:
    - Recommend movies with same actor(s), director, genre, …
  - Websites, blogs, news:
    - Recommend other sites with “similar” content

## Pros: Content-based Approach

- **+: No need for data on other users**
- **+: Able to recommend to users with unique tastes**
- **+: Able to recommend new & unpopular items**
  - No first-rater problem
- **+: Able to provide explanations**
  - Can provide explanations of recommended items by listing content-features that caused an item to be recommended

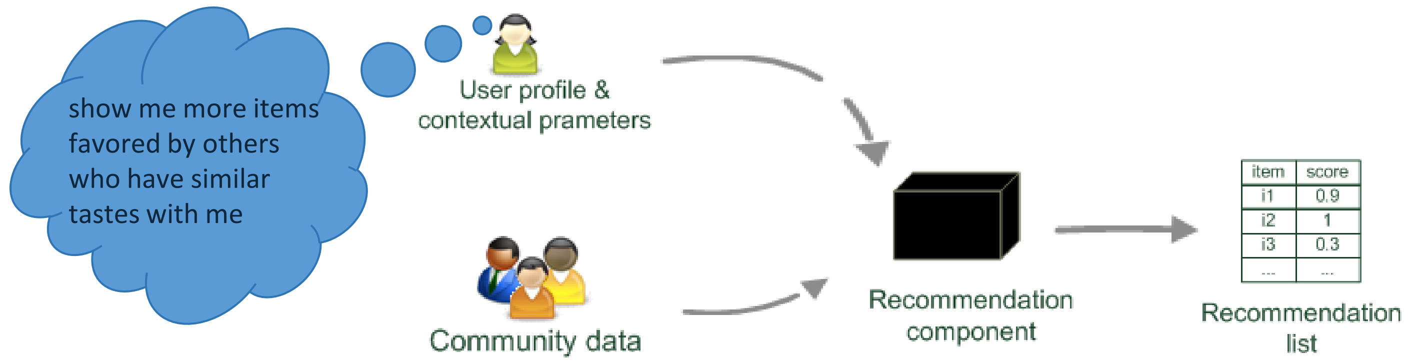
## Cons: Content-based Approach

- **–: Finding the appropriate features is hard**
  - E.g., images, movies, music

- **–: Recommendations for new users**
  - **How to build a user profile?**

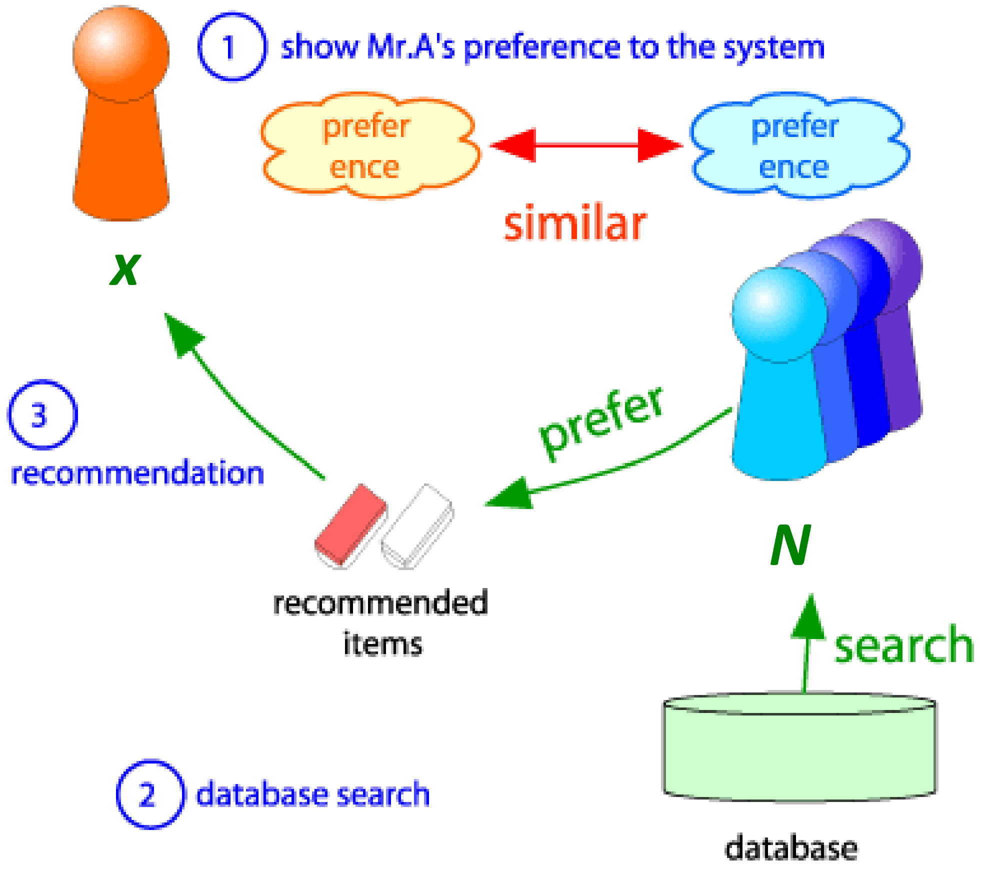
- **–: Overspecialization**
  - Never recommends items outside user’s content profile
  - People might have multiple interests
  - **Unable to exploit quality judgments of other users**

# Collaborative Filtering


show me more items favored by others who have similar tastes with me
User profile & contextual prameters
Community data
Recommendation component
item
score
i1
0.9
i2
1
i3
0.3
...
...
Recommendation list

# Collaborative Filtering

- Consider user ***x***
- Find set ***N*** of other users whose ratings are "**similar**" to ***x***'s ratings
- Estimate ***x***'s ratings based on ratings of users in ***N***



# User-based Nearest-Neighbor Collaborative Filtering

- The basic technique
  - Given an "active user" (Alice) and an item $i$ not yet seen by Alice
    - find a set of users (peers/nearest neighbors) who liked the same items as Alice in the past **and** who have rated item $i$
    - use, e.g. the average of their ratings to predict, if Alice will like item $i$
    - do this for all items Alice has not seen and recommend the best-rated

- Basic assumption and idea
  - If users had similar tastes in the past they will have similar tastes in the future
  - User preferences remain stable and consistent over time

# Finding "Similar" Users

- Let $\boldsymbol{r_x}$ be the vector of user $\boldsymbol{x}$'s ratings
- **Jaccard similarity measure** $\frac{||r_x \cap r_y||}{||r_x \cup r_y||}$
  - **Problem:** Ignores the value of the rating
- **Cosine similarity measure**
  - sim(***x***, ***y***) = cos($\boldsymbol{r_x}$, $\boldsymbol{r_y}$) = $\frac{r_x \cdot r_y}{||r_x|| \cdot ||r_y||}$
  - **Problem:** Treats missing ratings as "negative"
- **Pearson correlation coefficient**
  - $S_{xy}$ = items rated by both users ***x*** and ***y***

$$sim(x,y) = \frac{\sum_{s \in S_{xy}} (r_{xs} - \overline{r_x})(r_{ys} - \overline{r_y})}{\sqrt{\sum_{s \in S_{xy}} (r_{xs} - \overline{r_x})^2} \sqrt{\sum_{s \in S_{xy}} (r_{ys} - \overline{r_y})^2}}$$

$\boldsymbol{r_x}$ = [*, _, _, *, ***]
$\boldsymbol{r_y}$ = [*, _, **, **, _]

***$r_x$, $r_y$ as sets:***
$r_x$ = {1, 4, 5}
$r_y$ = {1, 3, 4}

***$r_x$, $r_y$ as points:***
$r_x$ = {1, 0, 0, 1, 3}
$r_y$ = {1, 0, 2, 2, 0}

$\overline{\mathbf{r_x}}$, $\overline{\mathbf{r_y}}$ … avg. rating of **x**, **y**

# Rating Predictions

**From similarity metric to recommendations:**

- Let $\boldsymbol{r_x}$ be the vector of user ***x***'s ratings
- Let ***N*** be the set of ***k*** users most similar to ***x*** who have rated item ***i***
- **Prediction for item *s* of *user x*:**
  - $r_{xi} = \frac{1}{k} \sum_{y \in N} r_{yi}$
  - $r_{xi} = \frac{\sum_{y \in N} s_{xy} \cdot r_{yi}}{\sum_{y \in N} s_{xy}}$
  - Other options?
- **Many other tricks possible…**

**Shorthand:**
$\boldsymbol{s_{xy} = sim(x, y)}$

# Item-Item Collaborative Filtering

- **So far: User-user collaborative filtering**
- **Another view: Item-item**
  - Basic idea:
    - Use the similarity between items (and not users) to make predictions
  - For item ***i***, find other similar items
  - Estimate rating for item ***i*** based on ratings for similar items
  - Can use same similarity metrics and prediction functions as in user-user model

$$\boxed{r_{xi} = \frac{\sum_{j \in N(i;x)} s_{ij} \cdot r_{xj}}{\sum_{j \in N(i;x)} s_{ij}}}$$

$s_{ij}$… similarity of items ***i*** and ***j***
$r_{xj}$…rating of user ***u*** on item ***j***
***N(i;x)***… set items rated by ***x*** similar to ***i***

# Item-Item Collaborative Filtering

- Example:
  - Look for items that are similar to Item5
  - Take Alice's ratings for these items to predict the rating for Item5

| | Item1 | Item2 | Item3 | Item4 | Item5 |
|---|---|---|---|---|---|
| Alice | 5 | 3 | 4 | 4 | ? |
| User1 | 3 | 1 | 2 | 3 | 3 |
| User2 | 4 | 3 | 4 | 3 | 5 |
| User3 | 3 | 3 | 1 | 5 | 4 |
| User4 | 1 | 5 | 5 | 2 | 1 |

# Item-Item CF (|N|=2)

users

movies

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | | 3 | | | 5 | | | 5 | | 4 | |
| 2 | | | 5 | 4 | | | 4 | | | 2 | 1 | 3 |
| 3 | 2 | 4 | | 1 | 2 | | 3 | | 4 | 3 | 5 | |
| 4 | | 2 | 4 | | 5 | | | 4 | | | 2 | |
| 5 | | | 4 | 3 | 4 | 2 | | | | | 2 | 5 |
| 6 | 1 | | 3 | | 3 | | | 2 | | | 4 | |

⬜ - unknown rating    🟨 - rating between 1 to 5

# Item-Item CF (|N|=2)

users

| movies | 1 | 2 | 3 | 4 | **5** | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | | 3 | | ? | 5 | | | 5 | | 4 | |
| 2 | | | 5 | 4 | | | 4 | | | 2 | 1 | 3 |
| 3 | 2 | 4 | | 1 | 2 | | 3 | | 4 | 3 | 5 | |
| 4 | | 2 | 4 | | 5 | | | 4 | | | 2 | |
| 5 | | | 4 | 3 | 4 | 2 | | | | | 2 | 5 |
| 6 | 1 | | 3 | | 3 | | | 2 | | | 4 | |

🟥 - estimate rating of movie **1** by user **5**

# Item-Item CF (|N|=2)

users

| movies | 1 | 2 | 3 | 4 | **5** | 6 | 7 | 8 | 9 | 10 | 11 | 12 | sim(1,m) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 |  | 3 |  | ? | 5 |  |  | 5 |  | 4 |  | 1.00 |
| 2 |  |  | 5 | 4 |  |  | 4 |  |  | 2 | 1 | 3 | -0.18 |
| 3 | 2 | 4 |  | 1 | 2 |  | 3 |  | 4 | 3 | 5 |  | 0.41 |
| 4 |  | 2 | 4 |  | 5 |  |  | 4 |  |  | 2 |  | -0.10 |
| 5 |  |  | 4 | 3 | 4 | 2 |  |  |  |  | 2 | 5 | -0.31 |
| 6 | 1 |  | 3 |  | 3 |  |  | 2 |  |  | 4 |  | 0.59 |

**Neighbor selection:**
Identify movies similar to movie **1**, **rated by user 5**

**Here we use Pearson correlation as similarity:**
**1)** Subtract mean rating $m_i$ from each movie $i$
$m_1 = (1+3+5+5+4)/5 = 3.6$
***row 1:*** *[-2.6, 0, -0.6, 0, 0, 1.4, 0, 0, 1.4, 0, 0.4, 0]*
**2)** Compute cosine similarities between rows

# Item-Item CF (|N|=2)

users

| movies | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | sim(1,m) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 |  | 3 |  | ? | 5 |  |  | 5 |  | 4 |  | **1.00** |
| 2 |  |  | 5 | 4 |  |  | 4 |  |  | 2 | 1 | 3 | **-0.18** |
| **3** | 2 | 4 |  | 1 | 2 |  | 3 |  | 4 | 3 | 5 |  | **0.41** |
| 4 |  | 2 | 4 |  | 5 |  |  | 4 |  |  | 2 |  | **-0.10** |
| 5 |  |  | 4 | 3 | 4 | 2 |  |  |  |  | 2 | 5 | **-0.31** |
| **6** | 1 |  | 3 |  | 3 |  |  | 2 |  |  | 4 |  | **0.59** |

**Compute similarity weights:**

$s_{1,3}=0.41$, $s_{1,6}=0.59$

# Item-Item CF (|N|=2)

users

| movies | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | | 3 | | **2.6** | 5 | | | 5 | | 4 | |
| 2 | | | 5 | 4 | | | 4 | | | 2 | 1 | 3 |
| **3** | 2 | 4 | | 1 | 2 | | 3 | | 4 | 3 | 5 | |
| 4 | | 2 | 4 | | 5 | | | 4 | | | 2 | |
| 5 | | | 4 | 3 | 4 | 2 | | | | | 2 | 5 |
| **6** | 1 | | 3 | | 3 | | | 2 | | | 4 | |

**Predict by taking weighted average:**

$r_{1.5}$ = **(0.41*2 + 0.59*3) / (0.41+0.59) = 2.6**

$$r_{ix} = \frac{\sum_{j \in N(i;x)} s_{ij} \cdot r_{jx}}{\sum s_{ij}}$$

# Item-Item vs. User-User

| | Avatar | LOTR | Matrix | Pirates |
|---|---|---|---|---|
| Alice | 1 | | 0.8 | |
| Bob | | 0.5 | | 0.3 |
| Carol | 0.9 | | 1 | 0.8 |
| David | | | 1 | 0.4 |

In practice, it has been observed that **item-item often works better than user-user**

Why? Items are simpler, users have multiple tastes

# Evaluating Predictions

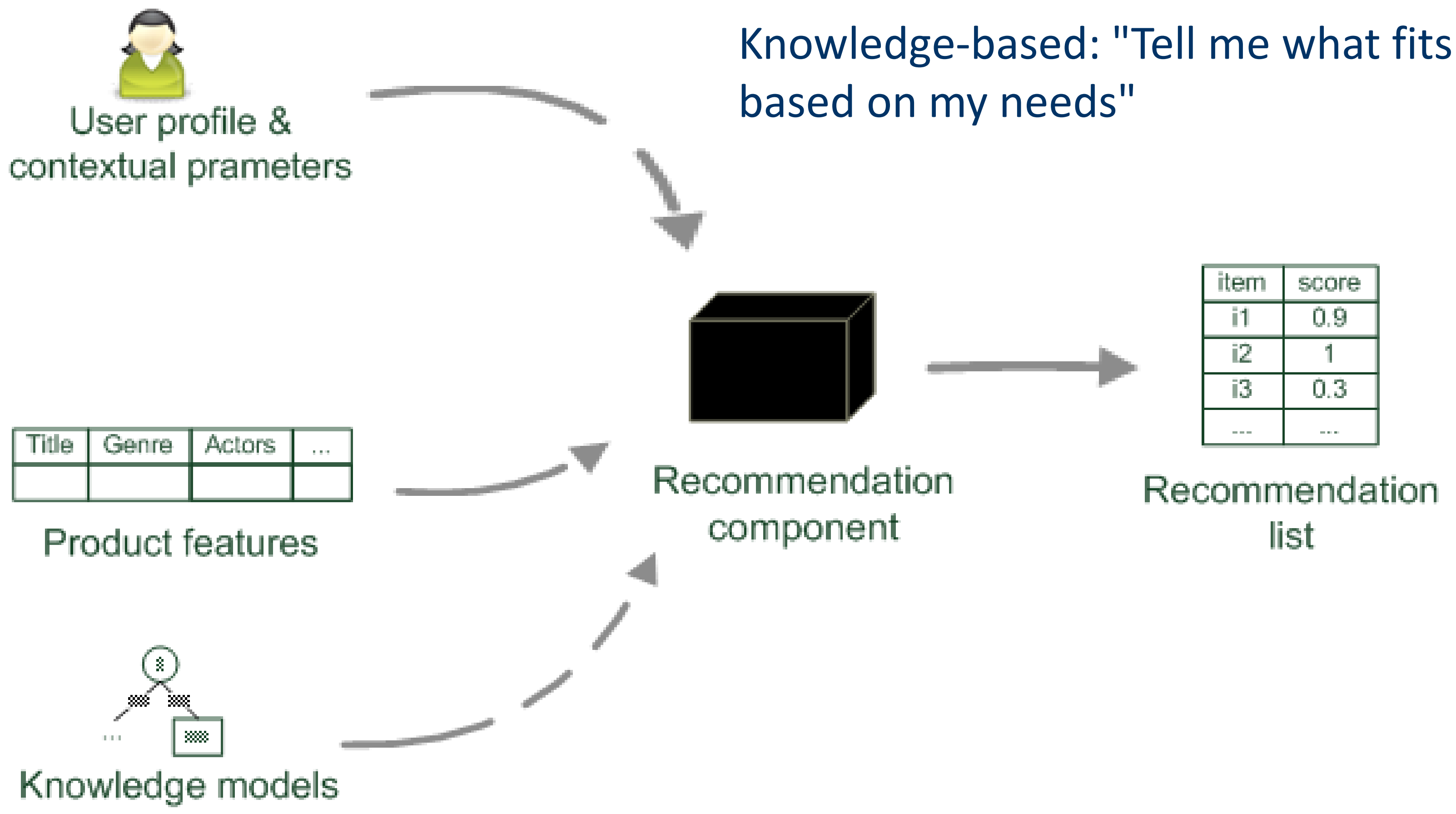
- **Compare predictions with known ratings**
  - **Root-mean-square error** (RMSE)
    - $\sqrt{\sum_{xi}\left(r_{xi} - r_{xi}^{*}\right)^{2}}$ where $\boldsymbol{r_{xi}}$ is predicted, $\boldsymbol{r_{xi}^{*}}$ is the true rating of $\boldsymbol{x}$ on $\boldsymbol{i}$
  - **Precision at top 10**:
    - % of those in top 10
  - **Rank Correlation**:
    - Spearman's *correlation* between system's and user's complete rankings
- **Another approach: 0/1 model**
  - **Coverage:**
    - Number of items/users for which system can make predictions
  - **Precision:**
    - Accuracy of predictions
  - **Receiver operating characteristic** (ROC)
    - Tradeoff curve between false positives and false negatives

# Pros/Cons of Collaborative Filtering

- **+ Works for any kind of item**
  - No feature selection needed
- **- Cold Start:**
  - Need enough users in the system to find a match
- **- Sparsity:**
  - The user/ratings matrix is sparse
  - Hard to find users that have rated the same items
- **- First rater:**
  - Cannot recommend an item that has not been previously rated
  - New items, Esoteric items
- **- Popularity bias:**
  - Cannot recommend items to someone with unique taste
  - Tends to recommend popular items

# Knowledge-Based Recommendation

Knowledge-based: "Tell me what fits based on my needs"



# Topic 5: Recommender Systems (Chapter 11)

- Recommender systems
  - Content-based recommendation
  - Collaborative recommendation
    - User-user collaborative filtering
    - Item-item collaborative filtering
  - BellKor Recommender System (the idea)
    - ~~Matrix Factorization~~

| | Avatar | LOTR | Matrix | Pirates |
|---|---|---|---|---|
| Alice | 1 | | 0.2 | |
| Bob | | 0.5 | | 0.3 |
| Carol | 0.2 | | 1 | |
| David | | | | 0.4 |

# Question 5 Recommender Systems

- Consider three users $u_1$, $u_2$, and $u_3$, and four movies $m_1$, $m_2$, $m_3$, and $m_4$. The users rated the movies using a 4-point scale: -1: bad, 1: fair, 2: good, and 3: great. A rating of 0 means that the user did not rate the movie. The three users' ratings for the four movies are: $u_1$ = (3, 0, 0, -1), $u_2$ = (2, -1, 0, 3), $u_3$ = (3, 0, 3, 1)
  - Which user has more similar taste to $u_1$ based on cosine similarity, $u_2$ or $u_3$? Show detailed calculation process.
  - Answer: sim(u1, u2) = (3*2 -1*3)/(sqrt(10)*sqrt(14)) ≈ 0.2535, sim(u1, u3) = (3*3-1*1)/(sqrt(10)*sqrt(19)) ≈ 0.5804. Thus u3 is more similar to u1.
  - User $u_1$ has not yet watched movies $m_2$ and $m_3$. Which movie(s) are you going to recommend to user $u_1$, based on the user-based collaborative filtering approach? Justify your answer.
  - Answer: You can use either cosine similarity or Pearson correlation coefficient to compute the similarities between users. However, the conclusion should be that only m3 is recommended to u1.