Distributed Data Management

Summer Semester 2015

TU Kaiserslautern

Prof. Dr.-Ing. Sebastian Michel

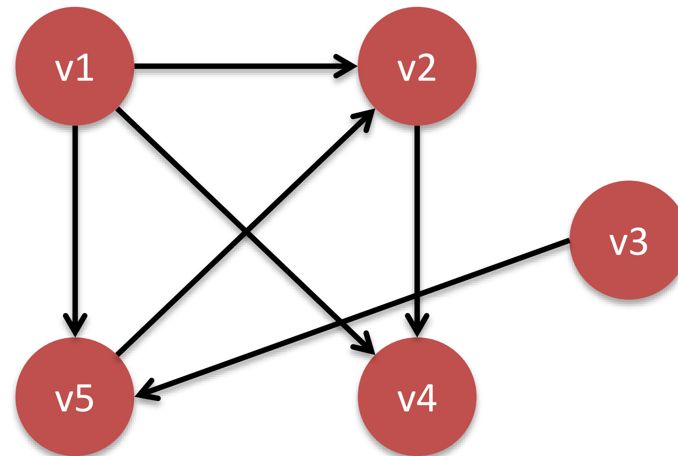
Databases and Information Systems Group (AG DBIS)

http://dbis.informatik.uni-kl.de/

GRAPH PROCESSING IN MAPREDUCE

Graph Processing in MapReduce

- General: Graph Representation
  - usually: Adjacency list
    - v1 -> v2, v4, v5
    - v2 -> v4
    - v3 -> v5
    - …



See Chapter 5 in https://lintool.github.io/MapReduceAlgorithms/MapReduce-book-final.pdf

Refresher: Breadth First Search (BFS)

Q = FIFO queue

enqueue start node

while not found:

  n := Q.dequeue

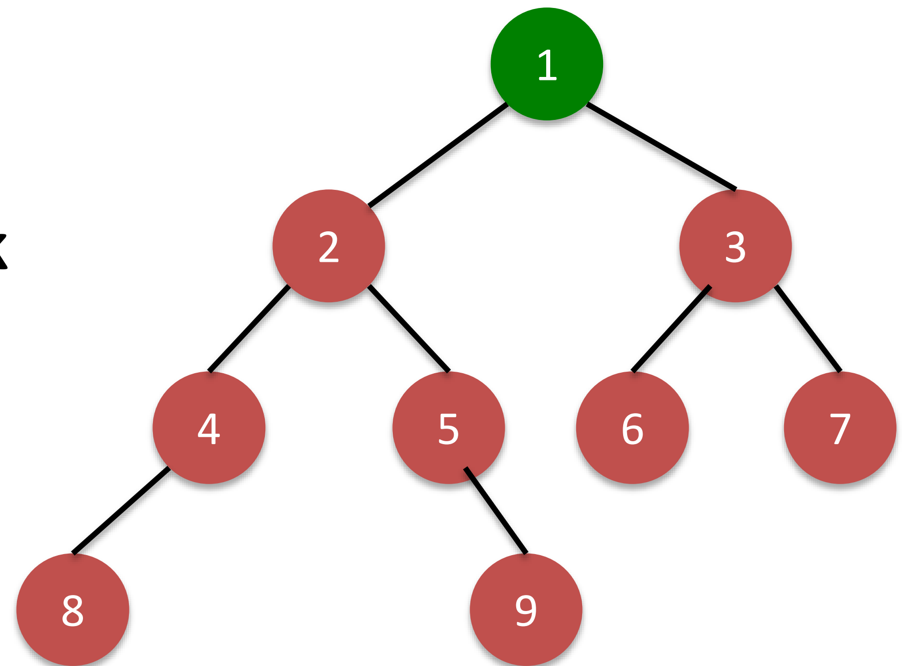
  if n== target **then break**

  foreach c in n.childlist

    Q.enqueue(c)

Keep also list of visited nodes!

- Example graph and visiting order:



Graph Processing in MapReduce

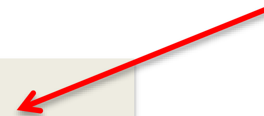
- **No global state** in MapReduce
- Need to **pass on results AND graph structure**

```
map(id, node) {
  emit(id, node)
  partial_result = local_compute()
  for each neighbor in node.adjacencyList {
     emit(neighbor.id, partial_result)
  }
}
```

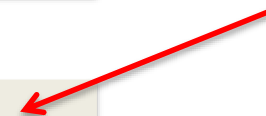
Graph Processing in MapReduce (2)

```
reduce(id, list) {
  foreach msg in list{
    if instanceof(msg) == Node
      node = msg
    else
      result = aggregate(result, msg)
    end
  }
  node.value = result
  emit(id, node)
}
```

re-construct outgoing edges for next round



make use of incoming results



BFS in MapReduce

- How to implement Breadth First Search in MapReduce?

- *Hint:* Need to pass on structure (as seen) before. Augment nodes with additional information: visited, distance.

Application: Computing PageRank

- Link analysis model proposed by Brin&Page
- Compute **authority scores**
- In terms of:
  – incoming links (weights) from other pages
- "**Random surfer model**"

Source: Wikipedia

S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. WWW Conf., 1998.

PageRank: Formal Definition

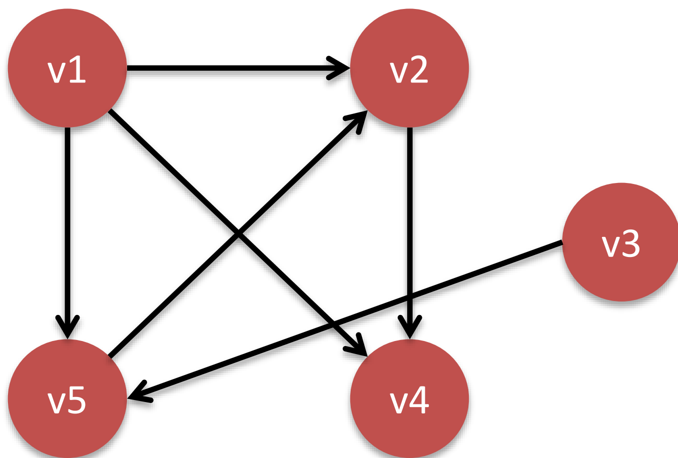
- **PageRank of a page q:**

$$PR(q) = \varepsilon \times \sum_{p|p \rightarrow q} \frac{PR(p)}{out(p)} + (1 - \varepsilon) \times \frac{1}{N}$$

  - *N* = Total number of pages;
  - *PR(p)* = PageRank of page p;
  - *out(p)* = **Outdegree** of p
  - *ε* = **Random jump probability**
- **Iterative computation until convergence**
- *Dangling nodes:* “Sinks”. Solution: Add random jump (uniform) to any other nodes.

Formal Model of Web Graph

- **Matrix representation** of graphs
- Given a graph G, its **adjacency matrix A** is n x n and
  - $a_{ij} = 1$, it there is a link from node i to node j
  - $a_{ij} = 0$, otherwise



	v1	v2	v3	v4	v5
v1	0	1	0	1	1
v2	0	0	0	1	0
v3	0	0	0	0	1
v4	0	0	0	0	0
v5	0	1	0	0	0

PageRank: Matrix Notation

- A → **Matrix containing the transition probabilities**

$$A = \varepsilon P^T + (1-\varepsilon)E$$

- where $P_{ij} = 1/out(i)$, if there is a link from *i* to *j*, 0 otherwise; E is the random jumps matrix
- Probability distribution vector at time k

$$\vec{x}^{(k)} = A^k \vec{x}^{(0)}$$

- $\vec{x}^{(0)}$ is the starting vector
- PageRank → **Stationary distribution** of the **Markov Chain described by A**, i.e., principal eigenvector or A

$$\text{PageRank} = lim_{k\rightarrow\infty}\vec{x}^{(k)}$$

More Details about PageRank upcoming in IRDM lecture

PageRank in MapReduce

- Reconsider: $$PR(q) = \varepsilon \times \sum_{p|p \to q} \frac{PR(p)}{out(p)} + (1 - \varepsilon) \times \frac{1}{N}$$

➔ to compute *PR(q)* we need only information about PR scores and out degree of nodes that **link to** q

Have info: (page q, PR) linking to page p1, p2, …

➔ Need to invert that pattern

See Chapter 5.3 in https://lintool.github.io/MapReduceAlgorithms/MapReduce-book-final.pdf

PR in MR: Map Phase

node has pageRank attribute and list of outgoing edges

- **map**(nid m, node M)

  p = M.pageRank / |M.adjacencyList|

  emit(nid m, M) ← send info about outgoing edges

  for all nid x in M.adjacencyList do

    emit(nid x, p) ← "send" score mass to nodes M links to

PR in MR: After Map Phase

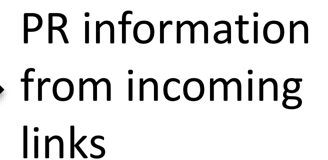
- We have now:

for page K (group by id of K):

[pageIN1, PR(IN1)/INn1],

[pageIN2, PR(IN2)/INn2],

....



[pageO1, pageO2, page03, …]

PR in MR: Reduce Phase

- **reduce**(nid m, [p1,p2,…])

```
s=0; M = node
for all p in [p1,p2,…] do
    if p instanceof node then
        M = p
    else
        s += p
M.pageRank = (1-ε)/N + ε*s
emit(nid m, node M)
```

COMPUTING SIMILARITY BETWEEN DOCUMENTS (OR ITEMS)

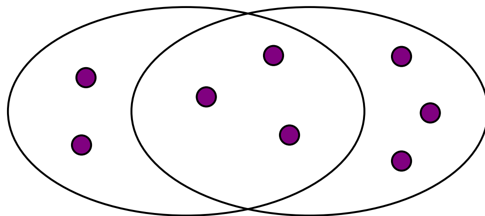
This part is to a large extent based on slides obtained from http://www.mmds.org

Distance Measures

- For finding similar documents,

  we consider the Jaccard distance/similarity

  - The **Jaccard similarity** of two **sets** is the size of their intersection divided by the size of their union: $\textit{sim}(C_1, C_2) = |C_1 \cap C_2| / |C_1 \cup C_2|$
  - **Jaccard distance:** $d(C_1, C_2) = 1 - |C_1 \cap C_2| / |C_1 \cup C_2|$



3 in intersection
8 in union
Jaccard similarity= 3/8
Jaccard distance = 5/8

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
    - Don't want to show both in search results
  - Similar news articles at many news sites
    - Cluster articles by "same story"
- **Problems:**
  - Many small pieces of one document can appear out of order in another
  - Too many documents to compare all pairs
  - Documents are so large or so many that they cannot fit in main memory

3 Essential Steps for Similar Docs

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

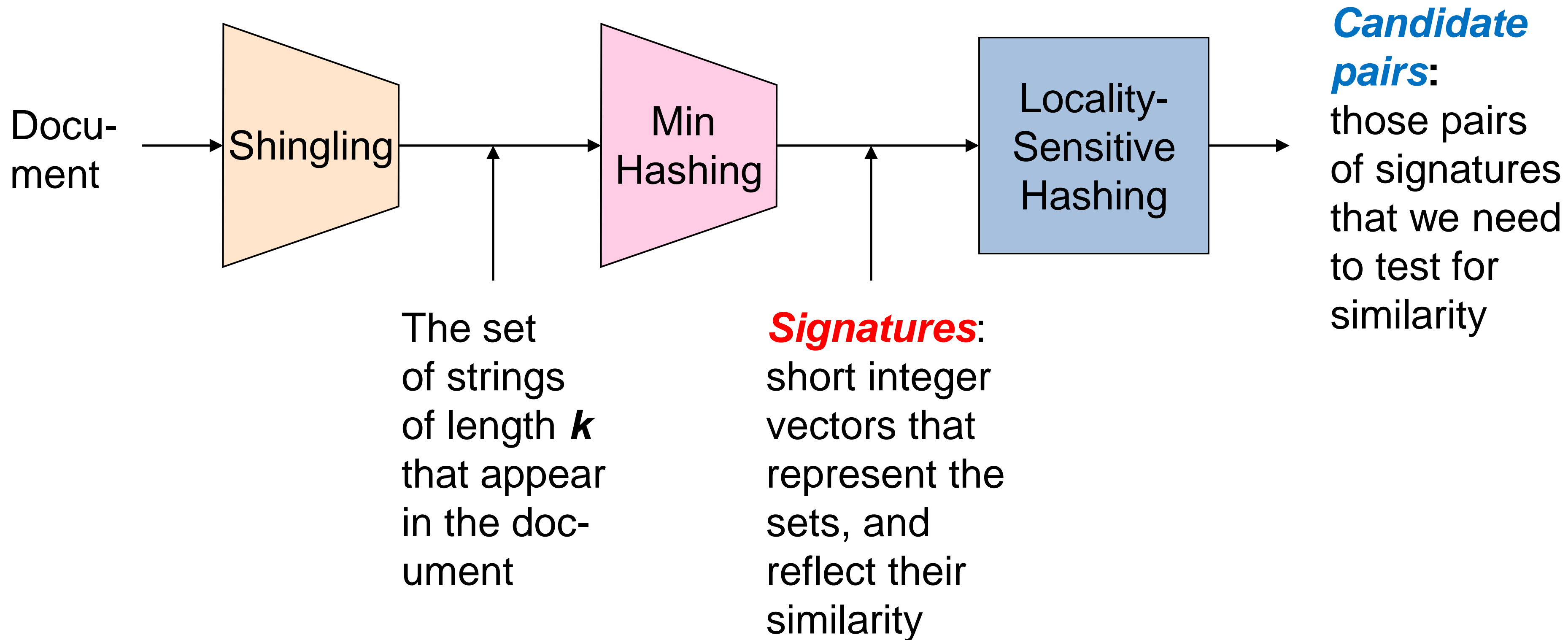
2. ***Min-Hashing:*** Convert large sets to short signatures, while preserving similarity

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

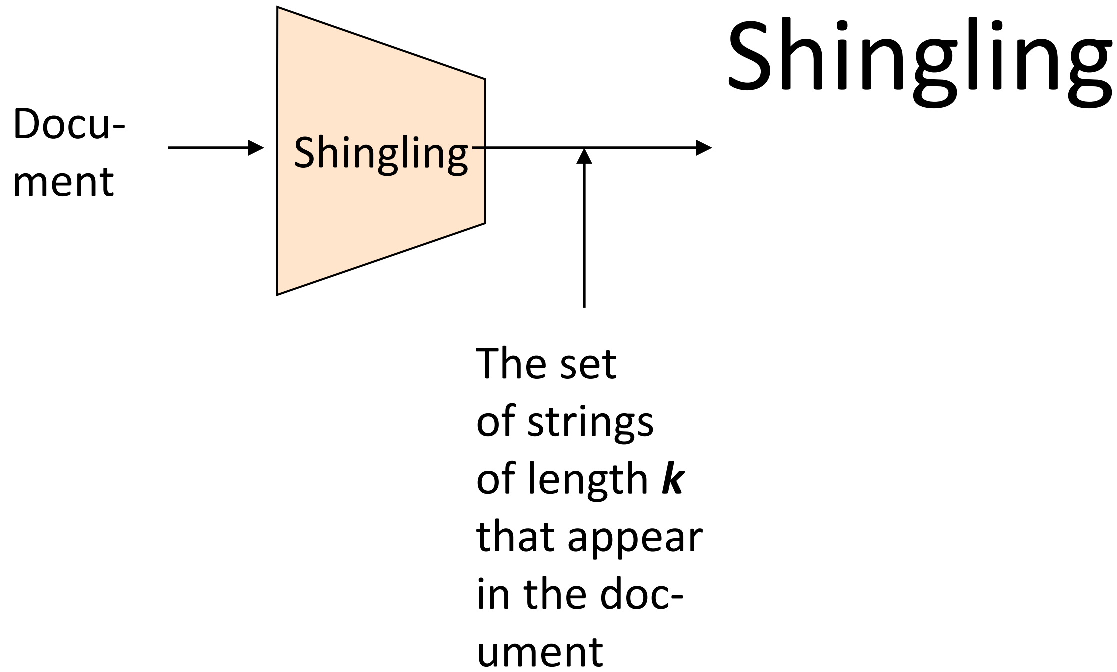
   -> Candidate pairs!

- Andrei Z. Broder. On the resemblance and containment of documents. SEQUENCES, 1997.
- Broder, Charikar, Frieze, Mitzenmacher: Min-Wise Independent Permutations. J. Comput. Syst. Sci. 60(3), 2000.

The Big Picture



Shingling



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

Documents as High-Dim Data

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

- **Simple approach:**
  - Document = set of words appearing in document

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

Define: Shingles

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

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

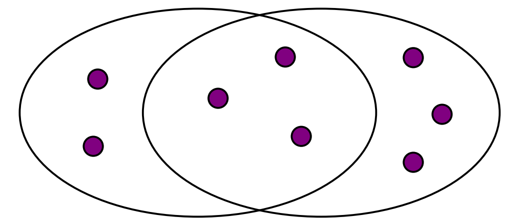
Compressing Shingles

- 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
- **Example:** **k=2**; document $\mathbf{D_1}$= abcab
  Set of 2-shingles: $\mathbf{S(D_1)}$ = {ab, bc, ca}
  Hash the singles: $\mathbf{h(D_1)}$ = {1, 5, 7}

Similarity Metric for Shingles

- **Document D_1 is a set of its k-shingles $C_1=S(D_1)$**
- 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:**

$$sim(D_1, D_2) = |C_1 \cap C_2| / |C_1 \cup C_2|$$



Working Assumption

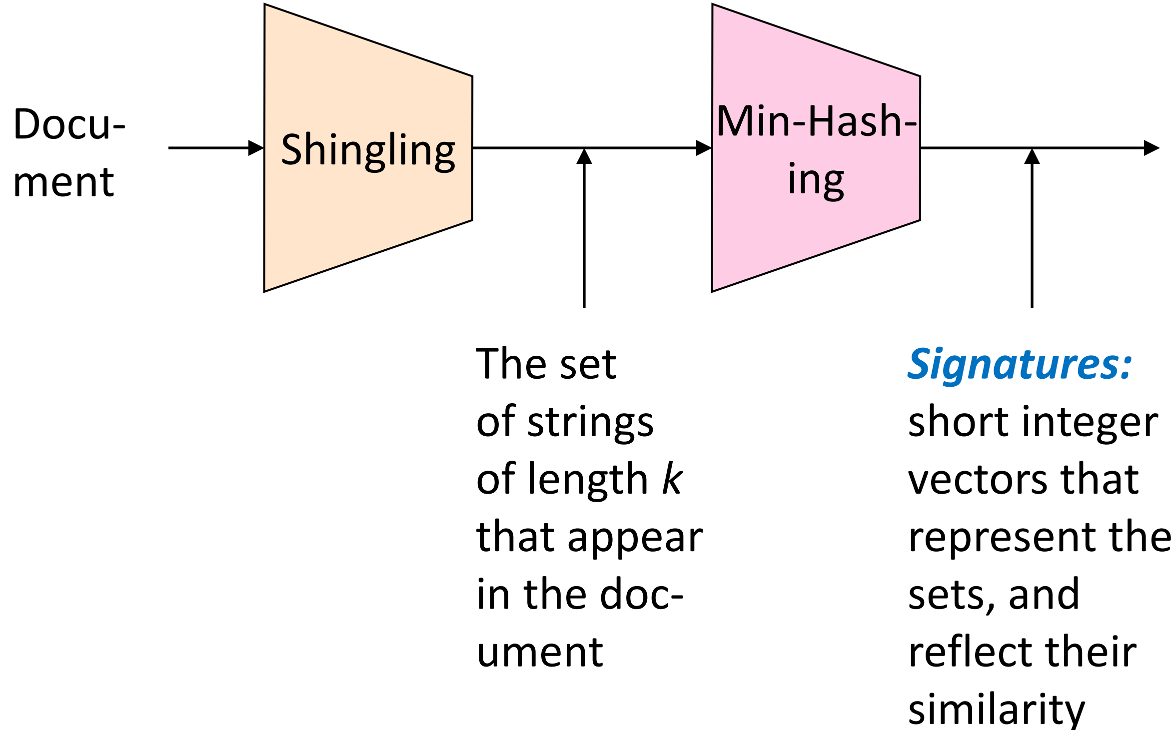
- **Documents that have lots of shingles in common have similar text, even if the text appears in different order**

- **Caveat:** You must pick ***k*** large enough, or most documents will have most shingles
  - ***k*** = 5 is OK for short documents
  - ***k*** = 10 is better for long documents

Motivation for Minhash/LSH

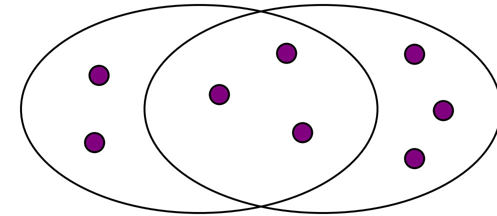
- **Suppose we need to find near-duplicate documents among $N = 1$ million documents**
- Naïvely, we would have to compute **pairwise Jaccard similarities** for **every pair of docs**
  - $N(N-1)/2$ ≈ **5*10^{11}** comparisons
  - At 10^5 secs/day and 10^6 comparisons/sec, it would take **5 days**
- For $N = 10$ million, it takes more than a year…

MinHashing



- **Step 2:** ***Minhashing:*** Convert **large sets** to **short signatures**, while **<u>preserving similarity</u>**

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:**
  - **Example: sim(C_1 ,C_2) = ?**
    - Size of intersection = 3; size of union = 6, Jaccard similarity (not distance) = 3/6
    - **d(C_1,C_2) = 1 – (Jaccard similarity) = 3/6**

Documents (columns); Shingles (rows)

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

- **So far:**
  - Documents → Sets of shingles
  - Represent sets as Boolean vectors in a matrix
- **Next goal: Find similar columns while computing small signatures**
  - **Similarity of columns == similarity of signatures**

Outline: Finding Similar Columns

- **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. $h(C_1) \neq h(C_2)$
- **Hash docs into buckets. Expect that "most" pairs of near duplicate docs hash into the same bucket!**

Min-Hashing

- **Goal: Find a hash function $h(\cdot)$ 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. $h(C_1) \neq h(C_2)$
- **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**

Min-Hashing

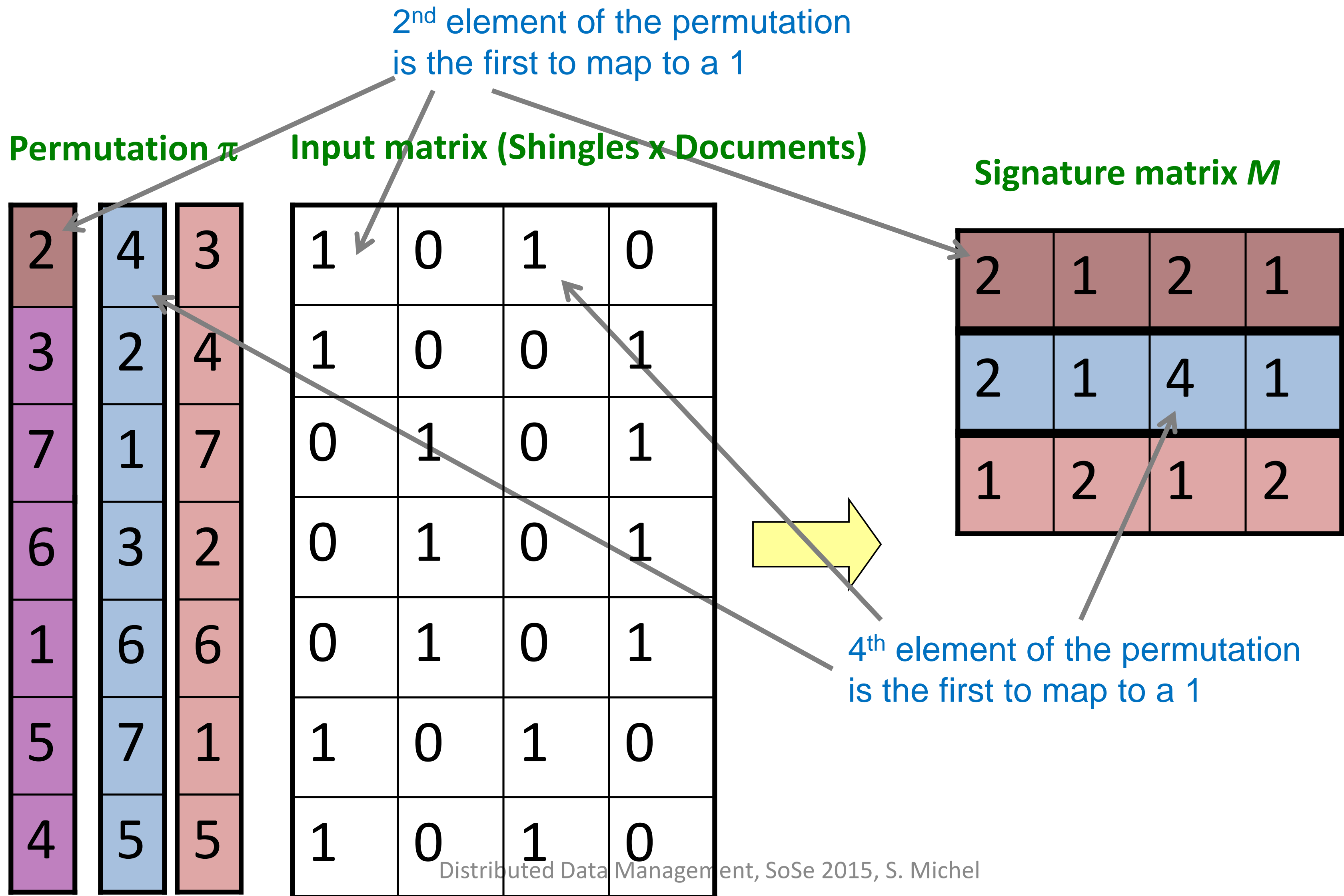
- Imagine the rows of the boolean matrix permuted under **random permutation** π

- 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-Hash Property

0	0
0	0
1	**1**
0	0
0	1
1	0

- **Choose a random permutation π**
- **Claim:** **Pr[$h_\pi(C_1) = h_\pi(C_2)$] = *sim*(C_1, C_2)**
- **Why?**
  - Let **X** be a doc (set of shingles), ***y*** $\in$ ***X*** is a shingle
  - **Then: Pr[$\pi(y) = \min(\pi(X))$] = 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 either:** $\pi(y) = \min(\pi(C_1))$ if $y \in C_1$, **or**
    $\pi(y) = \min(\pi(C_2))$ if $y \in C_2$
  - So the prob. that **both** are true is the prob. ***y*** $\in C_1 \cap C_2$
  - **Pr[$\min(\pi(C_1))=\min(\pi(C_2))$]=$|C_1 \cap C_2|/|C_1 \cup C_2|$= *sim*($C_1$, C_2)**

One of the two cols had to have 1 at position ***y***

Similarity for Signatures

- We know: **$\Pr[h_\pi(C_1) = h_\pi(C_2)] = sim(C_1, C_2)$**
- 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

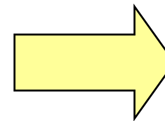
Min-Hashing Example

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
- **s*ig*(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!**

- **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)

How to make use of this?

- **What do we have so far?**
  - **Signatures** of documents
  - We can take them to compute the similarity of two documents
- **Need to avoid brute-force pairwise investigation if two docs are similar**
- **Instead:** Want to find candidate pairs for which we eventually evaluate the similarity.

Min-Hashing and MapReduce

- Given signatures of documents

Signature matrix *M*

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

- How to find documents that are similar, above a specific similarity threshold?
- **How do map and reduce look like?**

In MapReduce

- Need to make sure that **potentially similar documents** end up at the same reducer
- **Creation of signatures and shingling is done at mapper.**

- Given document d with signature [2,5,3,7,9]
- What do we emit in the mapper?
  - **emit**(,)

Case 1: Full Signature as Key

- Case 1:
  - Use full signature [2,5,3,7,9] as key
  - emit([2,5,3,7,9], d_1)

- Does this work?

Case 2: Each Hashf. used as Key

- **Case 2:**
  - Use individual parts of [2,5,3,7,9] as keys
  - emit([2], d_1)
  - emit([5], d_1)
  - emit([3], d_1)
  - emit([7], d_1)
  - emit([9], d_1)

+
Keys need also to be annotated where "number" came from

- Does this work?

Case 3: Chunks

- **Case 3:**
  - Use larger parts of [2,5,3,7,9] as keys
  - emit([2,5], d_1)
  - emit([3,7], d_1)
  - emit([7,9], d_1)
  - ..... (some or all sub-parts of certain size).....

+
Keys need also to be annotated where "number" came from

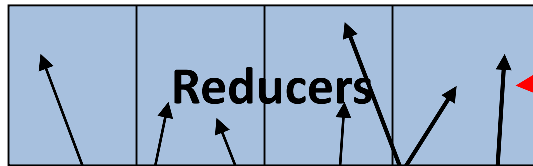
- Does this work?

Discussion

- LSH (Locality Sensitive Hashing) principle:
  - Map “similar” documents to the same hash bucket
  - Each bucket as “label” generated by s hash functions
  - There are l hash tables

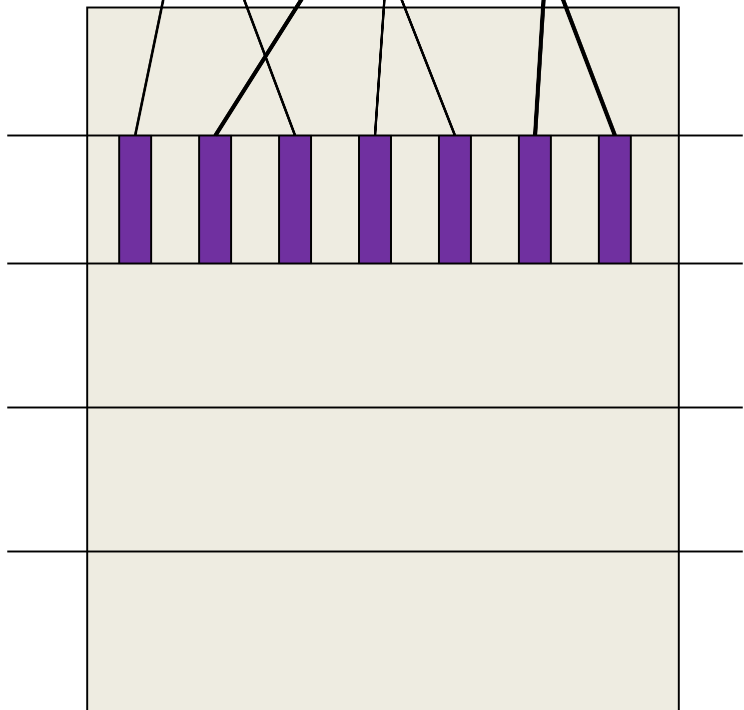
- Large bucket labels => Only really similar stuff in the same bucket.
- But low recall expected.
- Multiple hash tables counter this issue.

Illustration of Principle



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

Columns 6 and 7 are surely different.



How parameters can be tuned to query similarity threshold s is beyond the scope now.

There is plenty of theory around LSH (for various distance functions).

http://infolab.stanford.edu/~ullman/mmds/ch3.pdf

Computation of True Results

- The above MapReduce job leads to identified potentially similar documents at the Reducers.
- But so far the actual similarity values are not computed!
- How can this finally be done?
  - Can **ship to reducer full signature** and trust this is good enough to compute reasonably good approximation of similarity. Pros and Cons?
  - Or have a second (subsequent) MapReduce job. Pros and Cons?