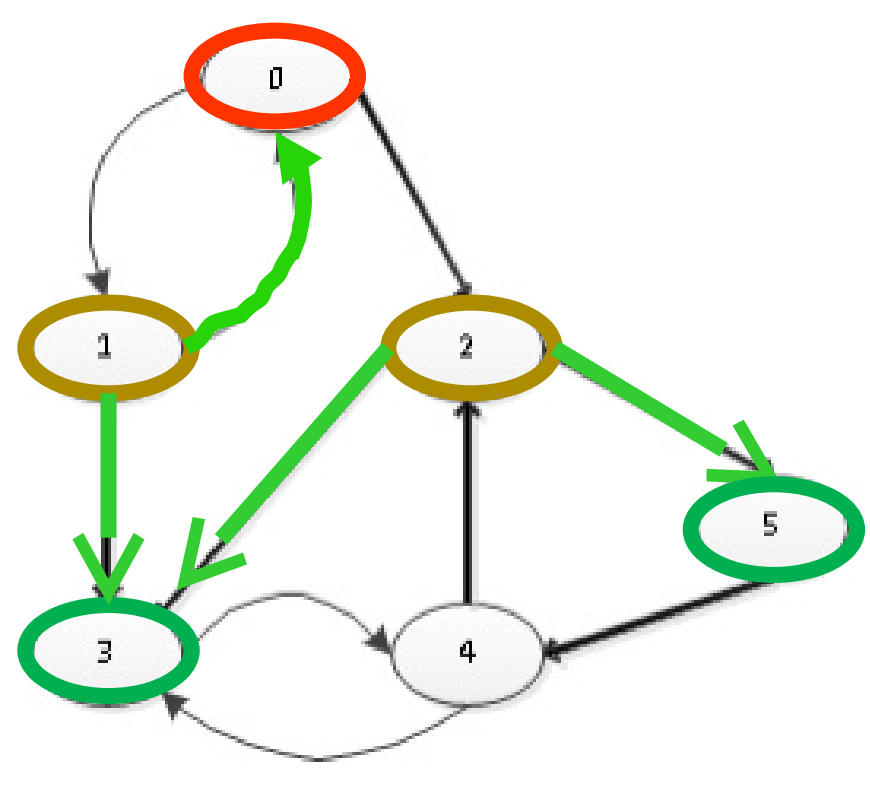
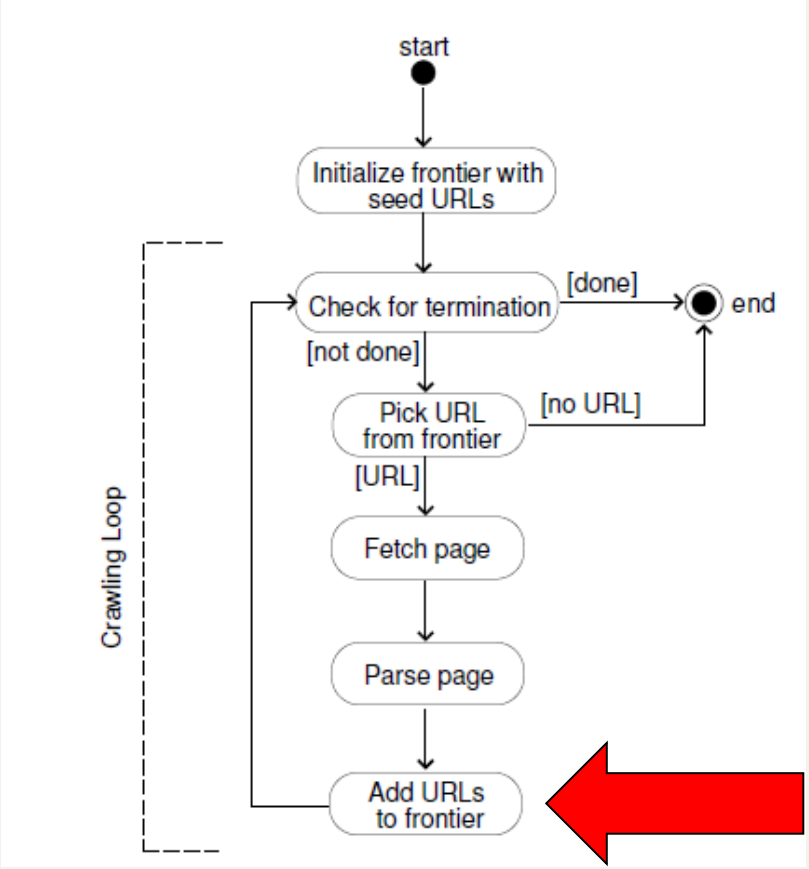
Crawling

A small example



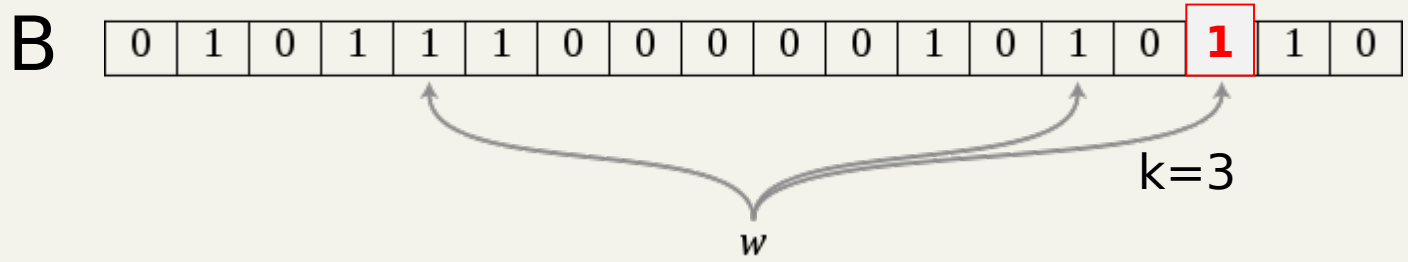


This page is a new one ?

- Check if the page has been parsed/downloaded before
  - **URL match**
  - Duplicate **document** match
  - Near-duplicate **document** match
- Some solutions:
  - Hashing on URLs
    - after 50 bln pages, we have “seen” over 500 bln URLs
    - each URL is at least 1000 bytes on average
    - Overall we have about 500.000 Tb (=500 Pb) for just the URLS
  - Disk access with caching (e.g. Altavista)
    - > 5 ms per URL check
    - > 5 ms * 5 * 10^{11} URL-checks => 80 years/1PC => 30gg/1000 PCs
  - Bloom Filter (Archive)
    - For 500 bln URLs → about 500 Tbit = 50Tb *[cfr. 1/1000 hashing]*

Is the page new? [Bloom Filter, 1970]

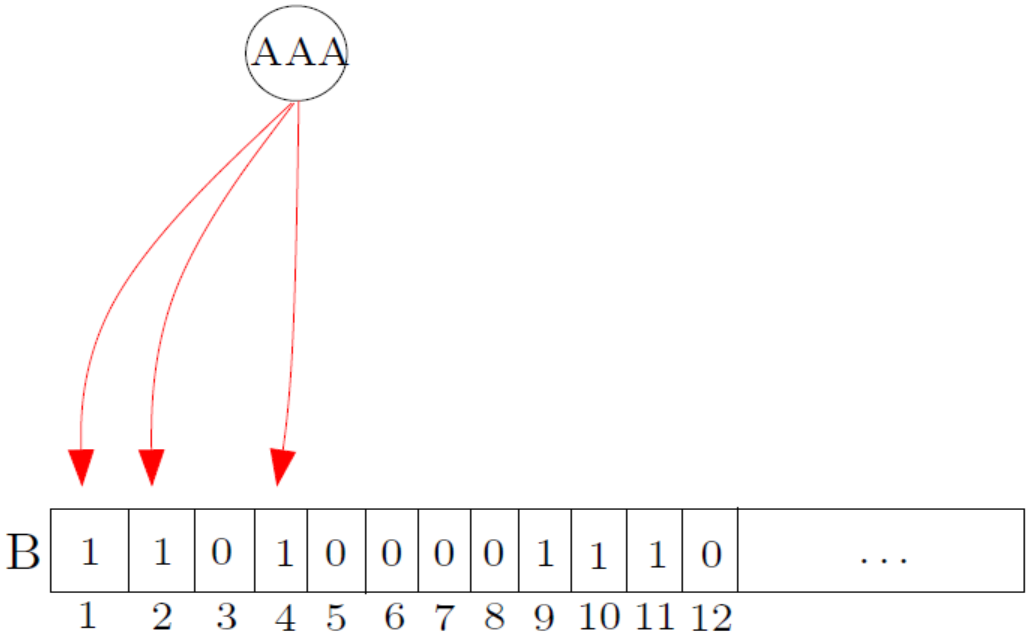
- Create a binary array B[1,m]
- Consider a family of k hash functions that map a key (URL) to a position (integer) in B



Pro: No need to store keys, less complex
≈ 50*X bln bytes *versus* 50.000 bln chars/bytes

Cons: false positives

Example: searching B



	h_1	h_2	h_3
ACG	3	6	4
ATA	2	11	10
CGA	11	9	6
TTA	1	10	9
TTT	6	2	1
CGC	9	3	3
AAA	4	1	2
TCT	10	4	11
...	...	...	...

$$AAA \stackrel{?}{\in} S \rightarrow YES$$

false positive

$$S = \{TTA, TCT, ATA\}$$

Probability of a *false positive*

m/n = 30 bits

$\varepsilon = 4 * 10^{-11}$

- assumption that hash are perfectly random
- after *build*

$$\mathcal{P}(b_i = 0) = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m} = p$$

- probability of a *false positive* is

$= 0.62^{m/n}$

Minimize prob. error for k = (m/n) ln 2

Advantageous when (m/n) « (key-length in bits + log n)

Pattern Matching

A set of objects whose keys are complex and time-costly to be compared (e.g. URLs, matrices, MP3,...).

- Use BF to reduce the number of explicit comparisons.
- Effective in hierarchical memories.
- Example on Dictionary matching [Bloom '70].

Set Intersection

We have two machines M_A and M_B each storing a set of items A and B, respectively. We wish to compute $A \cap B$ exchanging a *small* number of bits.
Typical applications: *data replication check*, *distributed search engines*.

Solution ?

(Approximate) Set Difference

We have two machines M_A and M_B each storing a set of items A and B, respectively.

We wish to *approximate* $B - A$ by exchanging few bits ($|A| \log \log |B|$), time depending on $|B - A|$, and just 1 communication round.

The previous algorithm solves it (i.e. $B - Q$) in $\Theta(|B|)$ time and *false negatives* at M_B (since $Q \supseteq A \cap B$), or it can solve it *exactly* at M_A as $A - (A \cap B)$ in $\Theta(|A|)$ time.

Scenario:

- Bandwidth between M_A and M_B is small (IrDA, BT, ...);
- CPU of M_B may be slow (PDAs, Phones, ...).

Patricia Tree over $|U| = 64$

Detect $B - A$ without comparing all of B's items. PT splits the space $[0, 63]$ in half at every level (drops *unary* nodes).

An attempt

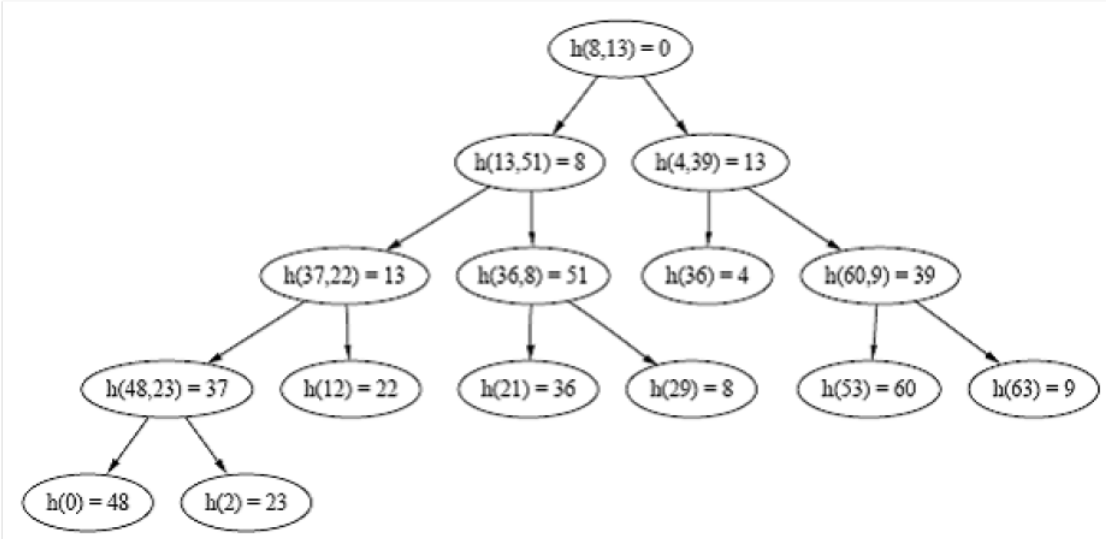
Given PT_A and PT_B at machine M_B, we proceed as follows:

- Visit PT_B top-down and compare a node of PT_B against the corresponding node in PT_A.
- If match (B's subset exists in A), the visit backtracks; otherwise proceeds to all children (A's subset $\not\subset B$).
- If we reach a leaf, then the corresponding element of B is declared to be in $B - A$.

Main issue: How to encode the subsets at the tree nodes? HASH!!!

Merkle Tree over $|U| = 64$

Merkle Tree = Patricia Tree plus Hashing.



We can *shuffle* data by hashing them onto $(\max\{|A|, |B|\})^2$.
The resulting PT or MT are *balanced*!

An approximate Algorithm: $|U| = h = 64 > |A|^2 = 49$

Use $BF(MT_A)$ to send MT_A in less bits and no bookkeeping for its structure.
But this introduces false-positive errors.

The algorithm

Given $BF(MT_A)$ and MT_B, the machine M_B proceeds as follows:

- Visits MT_B top-down and, for each node, check its *hash* in $BF(MT_A)$.
- If match (B's subset exists in A), the visit backtracks; otherwise proceeds to the children (A's subset $\not\subset B$).
- If we reach a leaf, then the corresponding element of B is declared to be in $B - A$.

Time and communication costs

Let $m_A = \Theta(|A| \log\log|B|)$ and the optimal $k_A = \Theta(\log\log|B|)$.

- We send m_A bits for the $BF(A)$.
- $\epsilon_A = (1/2)^{k_A} = O(1/\log|B|)$ is the error of $BF(A)$.
- Depth of (shuffled) MT_B is $d = O(\log|B|)$.
- Probability of success for a leaf is $(1-\epsilon_A)^d = \Theta(1)$. [Use boosting to increase it...]
- For each correct leaf, we visited its downward path of length $\Theta(\log|B|)$ computing $\Theta(\log\log|B|)$ hash functions per node.

This needs 1 round, $O(|A| \log\log|B|)$ bits, and $O(|B-A| \log|B| \log\log|B|)$ reconciliation time.

Spectral Bloom Filters (SBF)

Definition

$M = \langle S, f_x \rangle$ is a multiset were

- S is a set
- f_x is a function returning the #occurrences of x in M

Notice that a *stream* might be looked at as a *multiset*.

ex Given $\{A, A, B, C, C\}$

We have $S = \{A, B, C\}$ and $f_A = f_C = 2$, $f_B = 1$

Main features

- space usage is slightly larger, performance are better
- insertions/deletions are possible with some tricks
- can be built incrementally for *streaming data*

Applications:

- Iceberg query: Given x, check if $f_x > T$ *dynamic threshold*
- Aggregate query: `SELECT count(a1) FROM R WHERE a1=v`

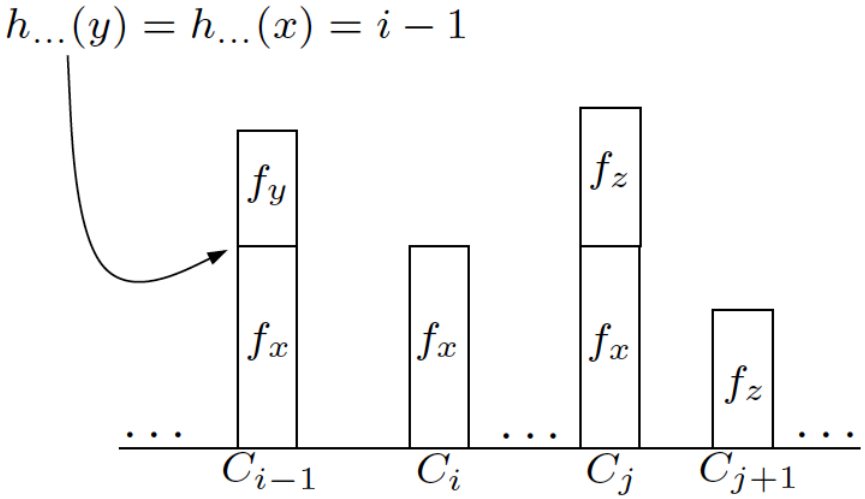
SBF

- B vector is replaced by a vector of counters $C_1, C_2, \ldots, C_m$
  - C_i is the sum of f_x values for elements $x \in S$ mapping to i
- Approximations of f_x are stored into

$$C_{h_1(x)}, C_{h_2(x)}, \ldots, C_{h_k(x)}$$

- Due to conflicts, the C_i provide approximations...

The *Minimum Selection*



- C_{i-1} is not a good approximation of f_x (neither of f_y)
- C_i is an exact approximation of f_x
- C_{j+1} is an exact approximation of f_z

Insertion and Deletion

- insertion is simple
  - increase each counter by 1

```
...
  for each h in H do
    C[h(x)] = C[h(x)] + 1;
  done
...
```

- deletion is simple
  - decrease each counter by 1

- search for an element x
  - return the *Minimum Selection* (MS) value $m_x = \min\{C_{h_1(x)}, C_{h_2(x)}, \ldots, C_{h_k(x)}\}$

On the error of SBF

- The error is the same as for Bloom Filters

Theorem

For all x, it is $f_x \leqslant m_x$. Furthermore $f_x \neq m_x$ with probability

$$E_{SBF} = \varepsilon \approx (1 - p)^k$$

Proof.

The case $m_x < f_x$ cannot happen.

The event $m_x > f_x$ is *"all counters $C_{h_i(x)}$ have a collision"*, that corresponds to a *"false positive"* event of classical BF. □

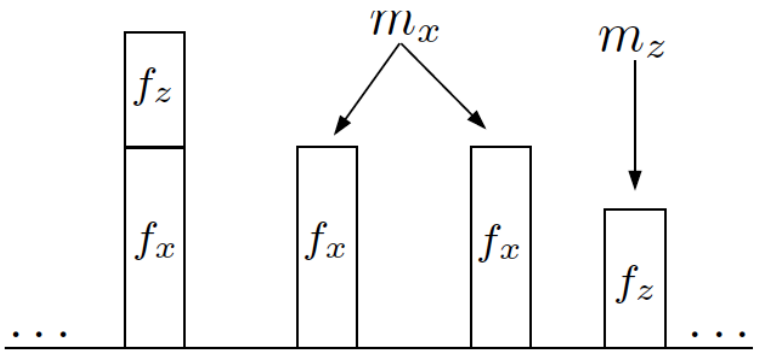
Implementing a SBF: challenges

Mainly two challenges

1. *allow insertion/deletion keeping low* E_{SBF}
2. *dynamic array of variable-length counters*

Solving Problem 1 with *Recurring Minimum(RM)*

- Strengthen the use of minimum, and support for ins/del!



  - x has a *Recurring Minimum* (RM)
  - z has a *Single Minimum* (SM)

"An element has a RM iff more than one of its counters has value equal to the minimum".

Solving Problem 1 with *Recurring Minimum(RM)*

An item which is subject to a Bloom Error is typically less likely to have recurring minimum among its counters.

Basic idea: We operate as in MS but over *two SBF*

1. For item x with RM we use m_x as estimator, which is highly probable to be correct: hence, $E_{SBF_1} < \varepsilon$.
2. For items with a SM, we use a secondary SBF which is $|SBF_2| \ll |SBF_1|$ and thus can guarantee $E_{SBF_2} \ll \varepsilon$.

We use more space, which could be used for enlarging the single BF, but experiments show that improvements may be remarkable!!!

Recurring Minimum: insertion and deletion

- insertion handles potential future errors
  1. increase ALL counters of x in SBF_1
  2. if x has a RM in SBF_1, stop
  3. otherwise, look for x in SBF_2
     1. if $x \in SBF_2$, increase ALL counters of x in SBF_2
     2. else set x in SBF_2 as its min value in SBF_1
- deletion is the inverse of insertion
  1. decrease ALL counters of x in SBF_1
  2. if x has SM in SBF_1, decrease (if any) ALL counters of x in SBF_2

Recurring Minimum: lookup

- lookup in both SBF, if needed.
  1. if x has a RM in SBF_1, return it
  2. else, say m_x^2 is value of x in SBF_2
     1. if $m_x^2 > 0$, return it
     2. else return min value of x in SBF_1

Deletion can't create *false negatives*: 0 can be returned only from SBF_1, and we always add/delete *over all* counters of both SBFs.

Parallel Crawlers

Web is too big to be crawled by a single crawler, work should be divided **avoiding duplication**

- Dynamic assignment
  - Central coordinator dynamically assigns URLs to crawlers
  - It needs communication bwt coordinator/crawl threads
- Static assignment
  - Web is statically partitioned and assigned to crawlers
  - Crawler only crawls its part of the web, no need of coordinator and thus communication

Two problems with static assignment

- Load balancing the #URLs assigned to cr
  - *Static schemes based on hosts may fail*
    - *www.geocities.com/….*
    - *www.di.unipi.it/*
  - *Dynamic "relocation" schemes may be co*
- Managing the fault-tolerance:
  - *What about the death of downloaders ? D→D-1,* ***new hash !!!***
  - *What about new downloaders ? D→D+1,* ***new hash !!!***

Let D be the number of crawlers.

hash(URL) maps an URL to {0,…,D-1}.

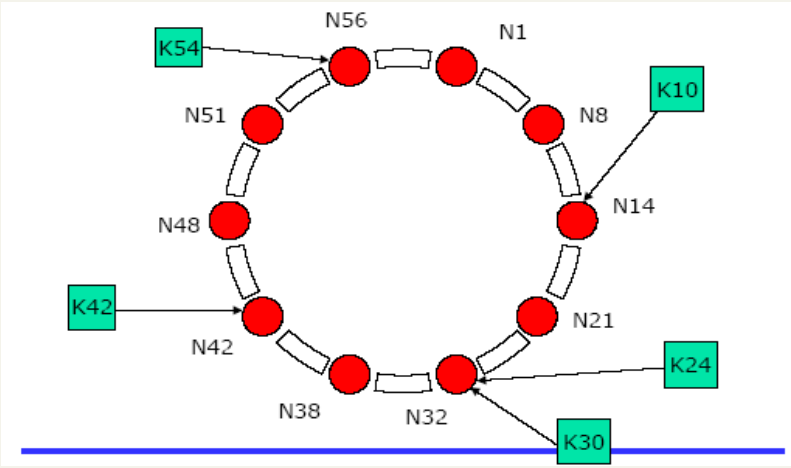
Crawler x manages the URLs U s.t. *hash*(U) = x

Which hash would you use?

A nice technique: Consistent Hashing

- A tool for:
  - *Spidering*
  - *Web Cache*
  - *P2P*
  - *Routers Load Balance*
  - *Distributed FS*

- Item *and* servers mapped to unit circle via hash function ID()
- Item K assigned to first server N such that ID(N) ≥ ID(K)
- What if a crawler goes down?
- What if a new crawler appears?



Each server gets replicated log S times

[monotone] adding a new server moves points between an old server to the new one, only.

[balance] Prob item goes to a server is ≤ O(1)/S

[load] any server gets ≤ (I/S) log S items w.h.p

[scale] you can copy each server more times...

Open Source

- Nutch (+ hadoop), also used by WikiSearch
  - http://nutch.apache.org/

Compressed storage of the Web-graph

Definition

Directed graph G = (V,E)

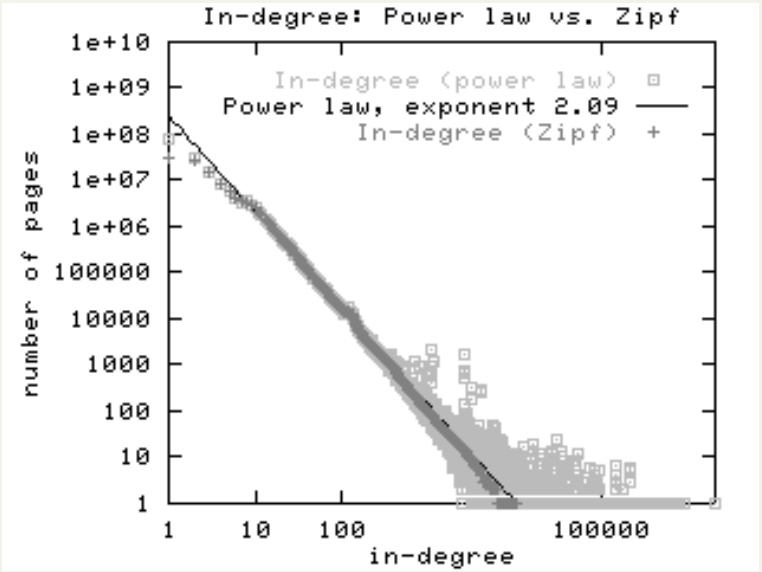
- V = URLs, E = (u,v) if u has an hyperlink to *v*

Isolated URLs are ignored (no IN & no OUT)

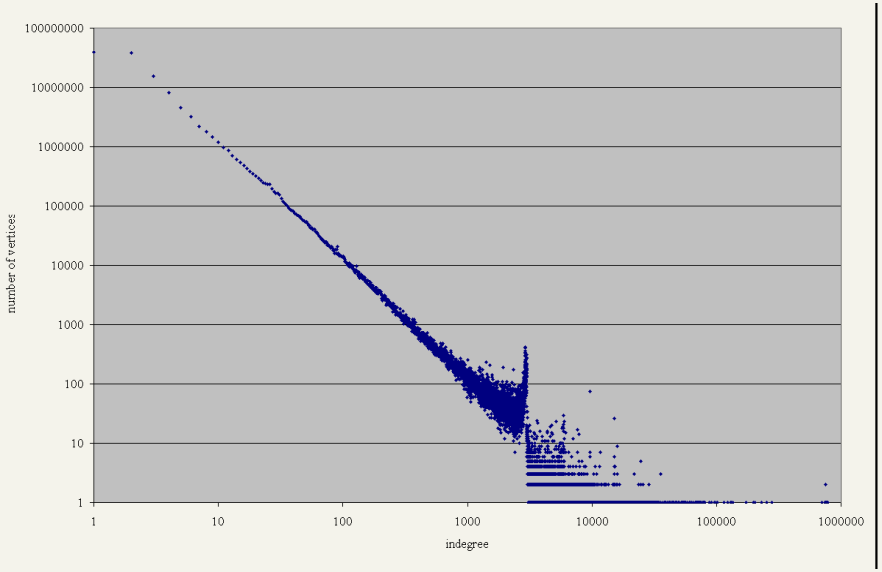
Three key properties:

- Skewed distribution: Pb that a node has x links is $1/x^{\alpha}$, $\alpha \approx 2.1$

The In-degree distribution



Altavista crawl, 1999



WebBase Crawl 2001

Indegree follows power law distribution

$$\Pr[\text{in-degree}(u) = k] \propto \frac{1}{k^{\alpha}}$$

$$\alpha = 2.1$$

This is true also for: out-degree, size of CC and SCC,...

Definition

Directed graph G = (V,E)

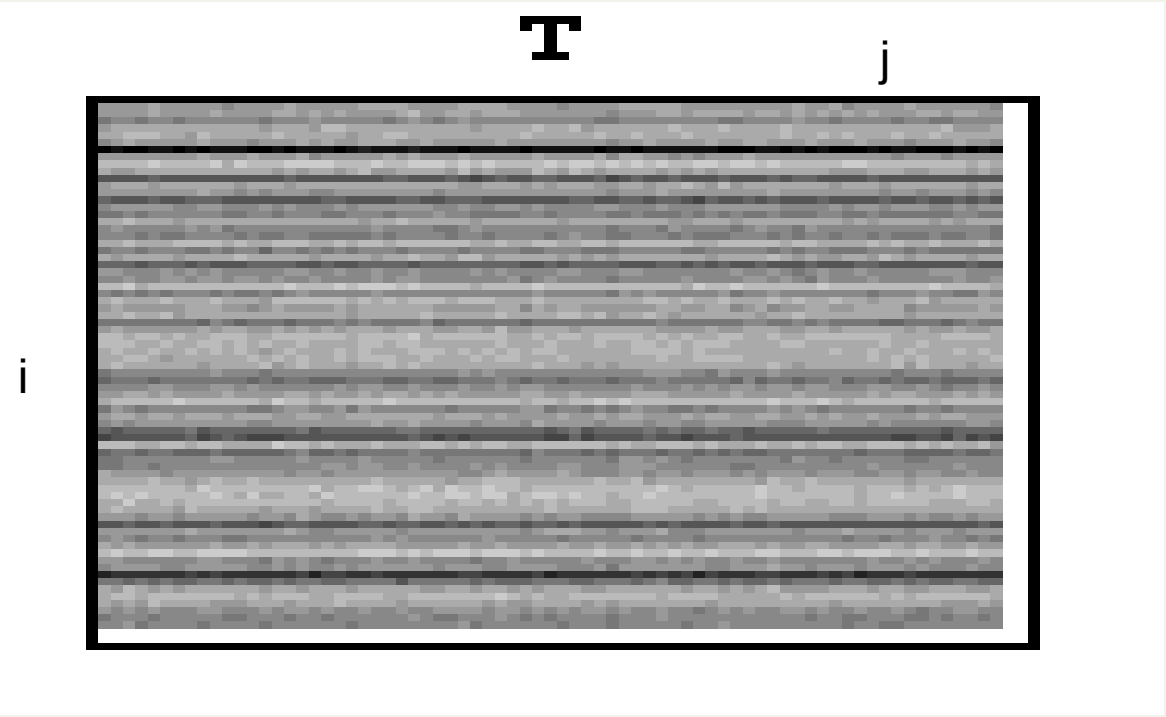
- V = URLs, E = (u,v) if u has an hyperlink to *v*

Isolated URLs are ignored (no IN, no OUT)

Three key properties:

- Skewed distribution: Pb that a node has x links is $1/x^{\alpha}$, $\alpha \approx 2.1$
- Locality: usually, most of the hyperlinks from URL **u** point to other URLs that are in the same host of **u** (about 80%).
- Similarity: if URLs **u** and **v** are close in lexicographic order, then they tend to share many hyperlinks

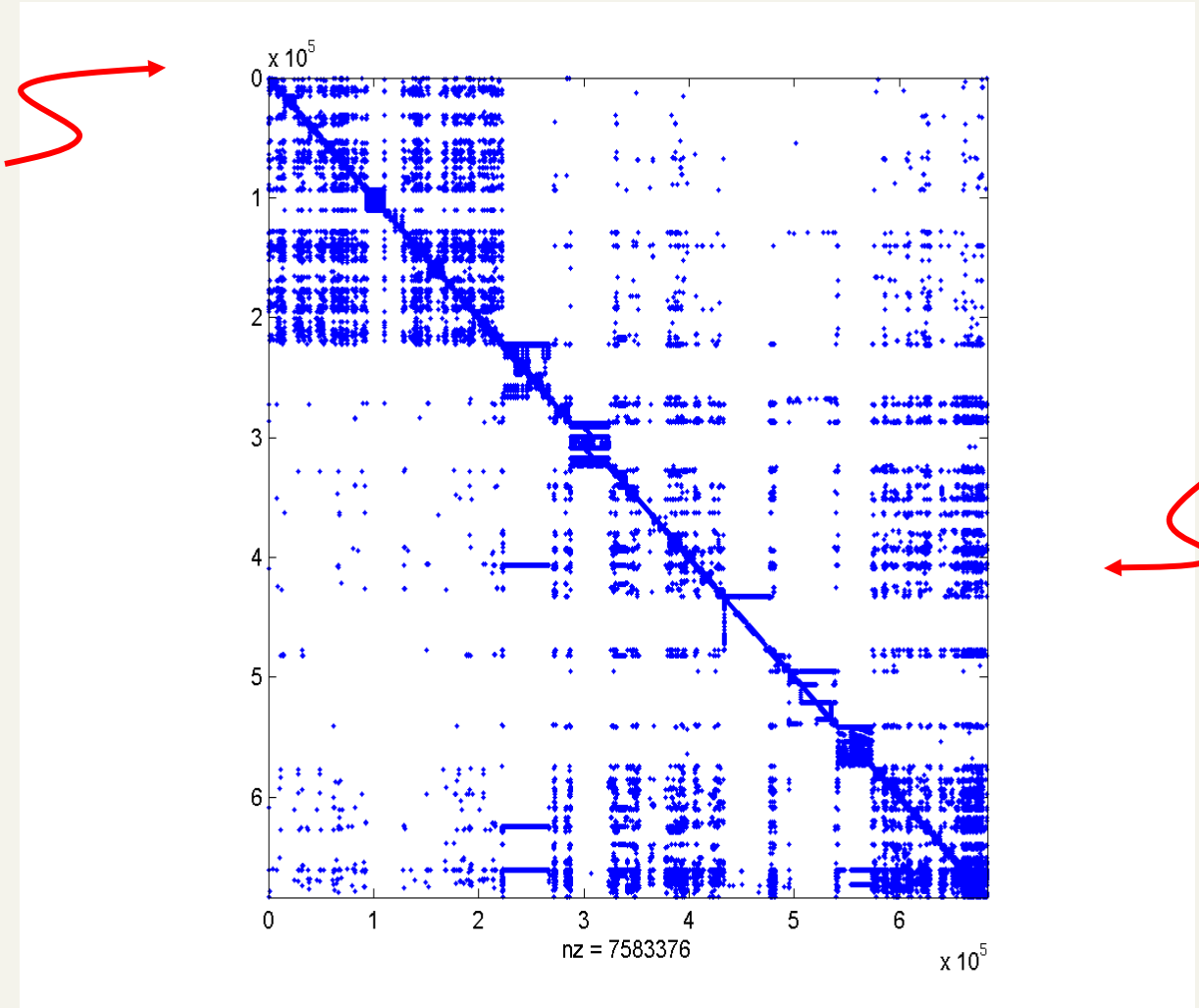
A Picture of the Web Graph



21 millions of pages, 150millions of links

URL-sorting

Berkeley



Stanford

Copy-lists: Locality

Uncompressed adjacency list

Node	Outdegree	Successors
...	...	...
15	11	13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	15, 16, 17, 22, 23, 24, 315, 316, 317, 3041
17	0	
18	5	13, 15, 16, 17, 50
...	...	...

Locality: most of the hyperlinks from URL **u** point to other URLs that are in the same host of **u** (about 80%).

Hosts in the same domain → are close to each other in the lexicographically sorted order, and thus they get **close** docIDs

→ Compress them via gap-encoding and variable-length representations

Reference copy-back are small (e.g. ≤ 8)

Copy-lists: Locality & Similarity

Uncompressed adjacency list

Node	Outdegree	Successors
...	...	...
15	11	13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	15, 16, 17, 22, 23, 24, 315, 316, 317, 3041
17	0	
18	5	13, 15, 16, 17, 50
...	...	...

(Similarity: if **u** and **v** are close in the lexicographic order,

then they tend to share many hyperlinks)

Node	Outd.	Ref.	Copy list	Extra nodes
...	...	...	...	...
15	11	0		13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	1	01110011010	22, 316, 317, 3041
17	0			
18	5	3	11110000000	50
...	...	...	...	...

Each bit of the copy-list informs whether the corresponding successor of *y* is also a successor of the reference *x*;

The reference index is the one in [0,W] that gives the best compression.

This is a Java and C++ lib
Called WebGraph ≈3 bits/edge

Copy-blocks = RLE(Copy-list)

Adjacency list with copy lists.

Node	Outd.	Ref.	Copy list	Extra nodes
...	...	...	...	...
15	11	0		13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	1	01110011010	22, 316, 317, 3041
17	0			
18	5	3	11110000000	50
...	...	...	...	...

Adjacency list with copy blocks

(RLE on bit sequences)

Node	Outd.	Ref.	# blocks	Copy blocks	Extra nodes
...	...	...	...	...	...
15	11	0			13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	1	7	0, 0, 2, 1, 1, 0, 0	22, 316, 317, 3041
17	0				
18	5	3	2	1,3	50
...	...	...	...	...	...

The first bit specifies the first copy block

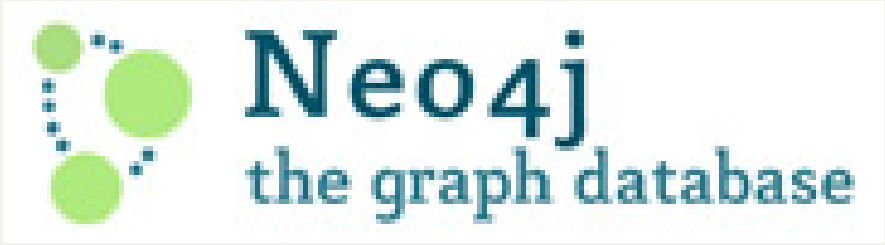
Each RLE-length is decremented by one for all blocks

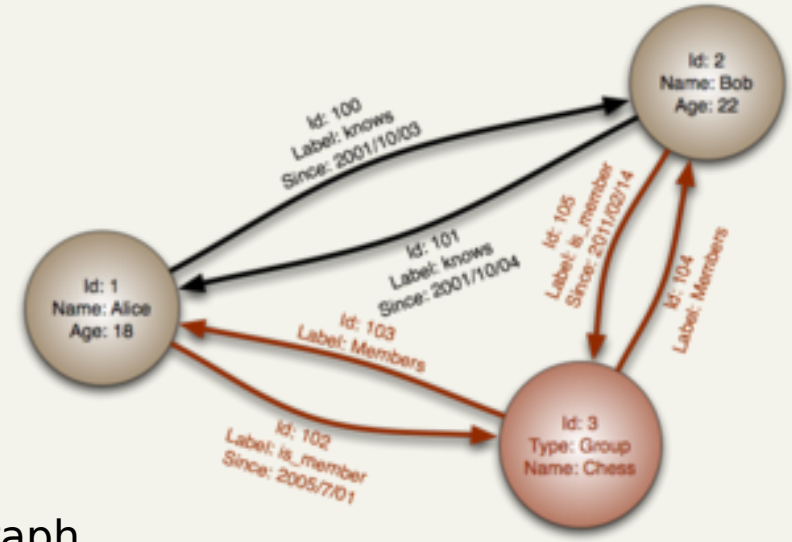
The last block is omitted (we know the length from *Outd*);

More on extra nodes, but not here !!

What about other graphs?

A directed graph G = (V,E) not necessarily satisfy Web properties above





Apache Giraph is an iterative graph processing system built for high scalability. It is currently used at Facebook. Giraph originated as the open-source counterpart to *Pregel*, the graph processing architecture developed at Google (2010). Both systems are inspired by the Bulk Synchronous Parallel model of distributed computation.