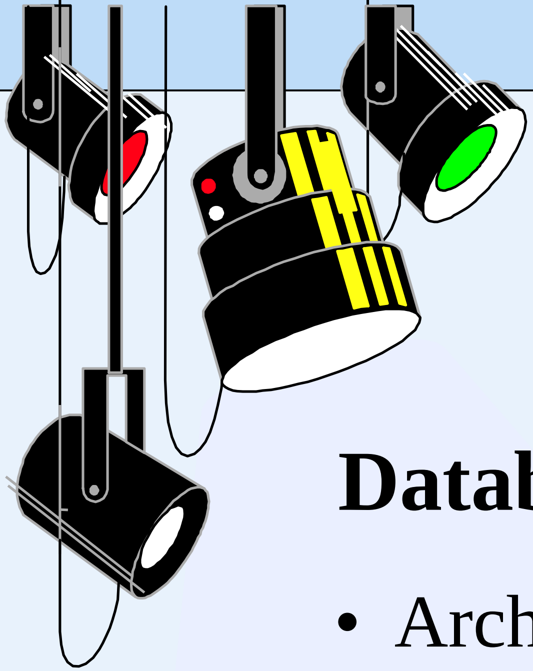
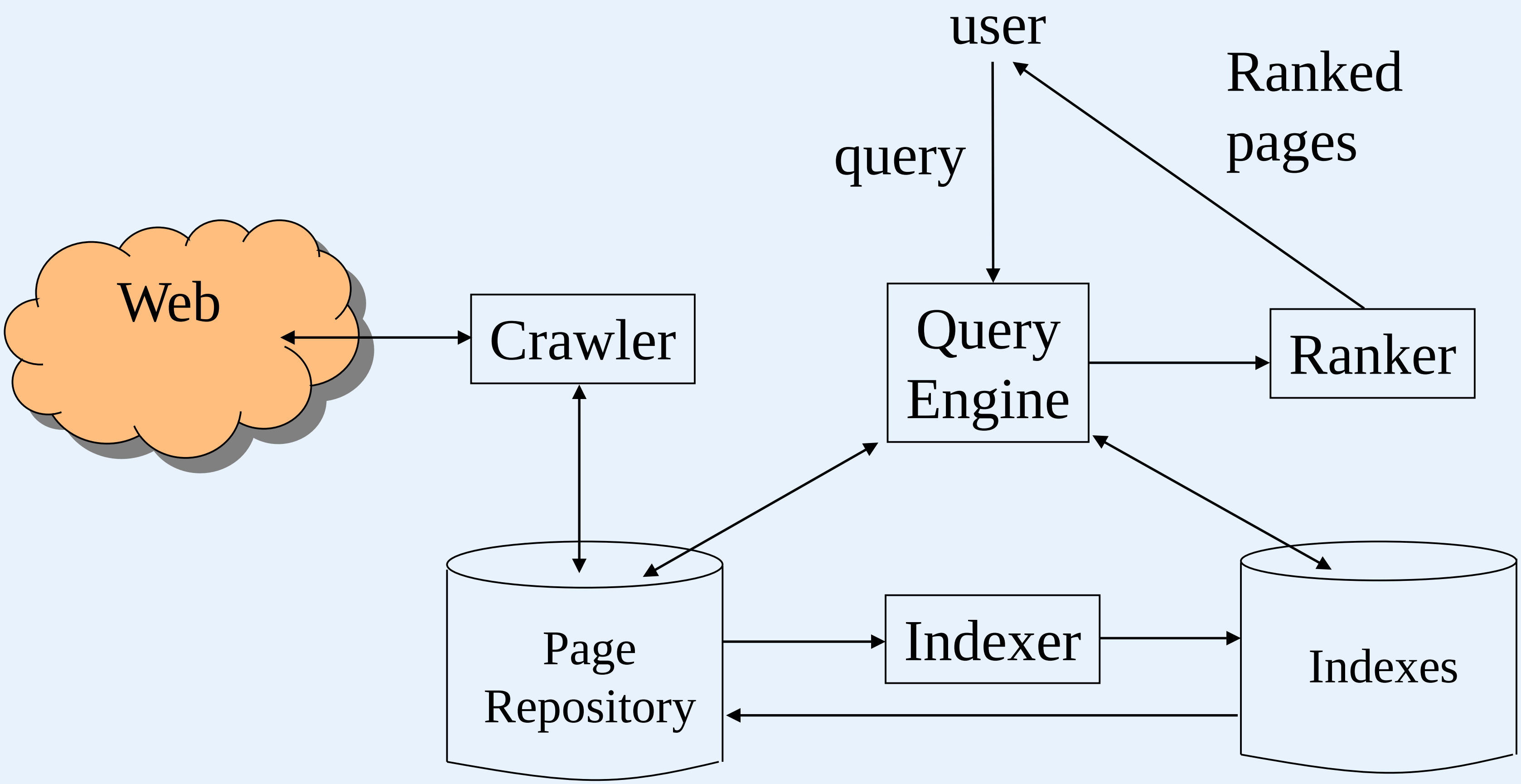


Database Systems and Internet

- Architecture of a search engine
  - Web crawler
  - Query engine
  - PageRank for identifying important pages
- Data stream management
  - What is a steam?
  - Stream compression
  - Stream mining

The Architecture of a Search Engine



The Architecture of a Search Engine

There are two main functions that a search engine must perform.

1. The Web must be crawled. That is, copies of many of the pages on the Web must be brought to the search engine and processed.
2. Queries must be answered, based on the material gathered from the Web. Usually, a query is in the form of a word or words that the desired Web pages should contain, and the answer to a query is a ranked list of the pages that contain all those words, or at least some of them.

The Architecture of a Search Engine

Crawler – interact with the Web and find pages, which will be stored in Page Repository.

Query engine – takes one or more words and interacts with indexes, to determine which pages satisfy the query.

Indexer – inverted file: for each word, there is a list of the pages that contain the word. Additional information in the index for the word may include its locations within the page or its role, e.g., whether the word is in the header.

Ranker – order the pages according to some criteria.

Web Crawler

A crawler can be a single machine that is started with a set ***S***, containing the URL's of one or more Web pages to crawl. There is a repository ***R*** of pages, with the URL's that have already been crawled; initially *R* is empty.

Algorithm: A simple Web Crawler
Input: an initial set of URL's *S*.
Output: a repository *R* of Web pages

Web Crawler

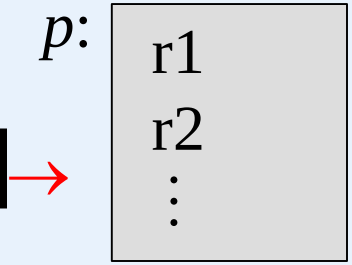
Method: Repeatedly, the crawler does the following steps.

1. If *S* is empty, end.
2. Select a URL *r* from the set *S* to “crawl” and delete *r* from *S*.
3. Obtain a page *p*, using its URL *r*. If *p* is already in repository *R*, return to step (1) to select another URL from *S*.
4. If *p* is not already in *R*:

   (a) Add *p* to *R*.

   (b) Examine *p* for links to other pages. Insert into *S* the URL of each page *q* that *p* links to, but that is not already in *R* or *S*.
5. Go to step (1).

r: https://www.youtube.com/watch?v =EctlAlYVWwU



p:

r1

r2

⋮

Web Crawler

The algorithm raises several questions.

a) How to terminate the search if we do not want to search the entire Web?
b) How to check efficiently whether a page is already in repository *R?*
c) How to select a URL *r* from *S* to search next?
d) How to speed up the search, e.g., by exploiting parallelism?

Terminating Search

The search could go on forever due to dynamically constructed pages.

Set limitation:

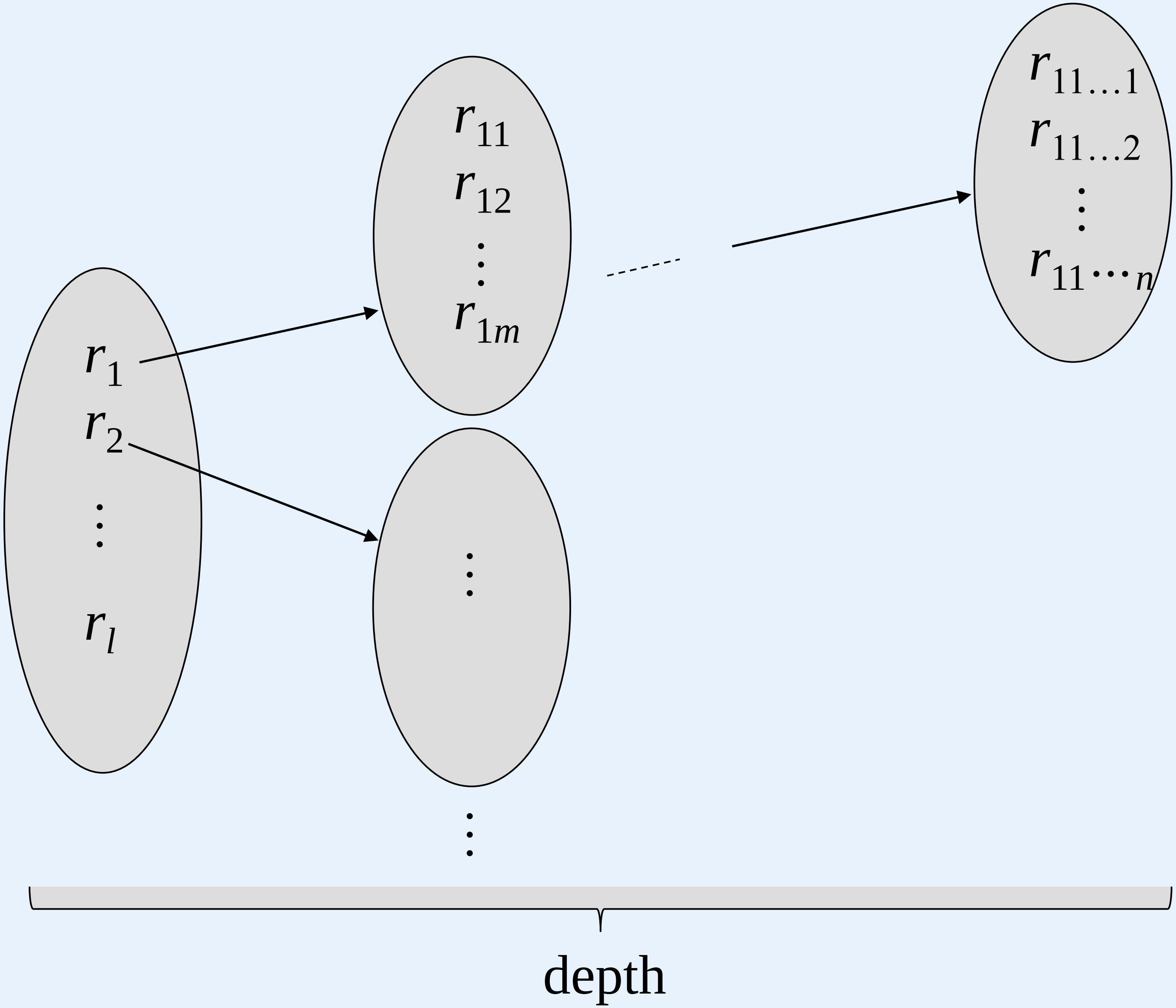
- Set a limit on the number of pages to crawl.

  The limit could be either on each site or on the total number of pages.

- Set a limit on the depth of the crawl.

  Initially, the pages in set *S* have depth 1. If the page *p* selected for crawling at step (2) of the algorithm has depth *i*, then any page *q* we add to *S* at step 4-(b) is given depth *i* + 1. Moreover, if *p* has depth equal to the limit, then do not examine links out of *p* at all. Rather we simply add *p* to *R* if it is not already there.

DBS and the Internet



Managing the Repository

- When we add a new URL for a page *p* to the set *S*, we should check that it is not already there.

- When we decide to add a new page *p* to *R* at step 4-(a) of the algorithm, we should be sure the page is not already there.

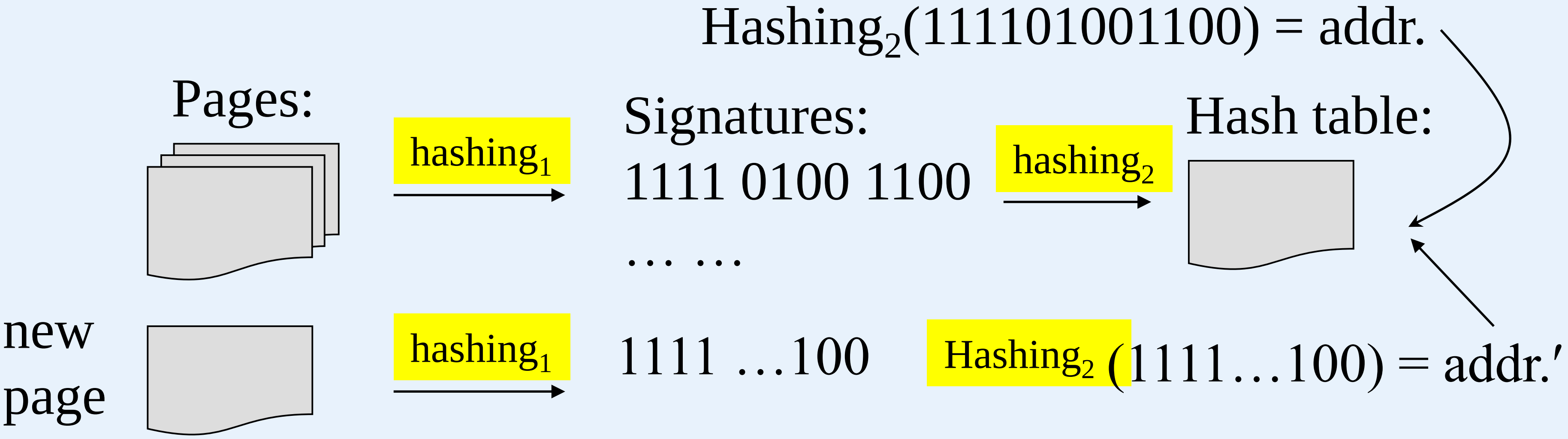
Page signatures:

- Hash each Web page to a signature of, say, 64 bits.
- The signatures themselves are stored in a hash table *T*, i.e., they are further hashed into a smaller number of buckets, say one million buckets.

DBS and the Internet

Page signatures:

- Hash each Web page to a signature of, say, 64 bits.
- The signatures themselves are stored in a hash table *T*, i.e., they are further hashed into a smaller number of buckets, say one million buckets.
- When inserting *p* into *R*, compute the 64-bit signature *h*(*p*), and see whether *h*(*p*) is already in the hash table *T*. If so, do not store *p*; otherwise, store *p* in *T*.



Hashing$_2$(111101001100) = addr.

Pages:

hashing$_1$

Signatures:

1111 0100 1100

… …

hashing$_2$

Hash table:

new page

hashing$_1$

1111 …100

Hashing$_2$ (1111…100) = addr.′

DBS and the Internet

- **Signature file**
  - A signature file is a set of bit strings, which are called *signatures*.
  - In a signature file, each signature is constructed for a record in a table, a block of text, a page or an image.
  - When a query arrives, a query signature will be constructed according to the key words involved in the query. Then, the signature file will be searched against the query signature to discard non-qualifying signatures, as well as the objects represented by those signatures.

- **Signature generation**

  - Generate a signature for an attribute value or a key word
    Before we generate the signature for an attribute value, or a key word, three parameters have to be determined

    F: number of 1s in bit string
    m: length of bit string
    D: number of attribute values in a record (or average number of the key words in a page)

    Optimal choice of the parameters:

$$m \times \ln 2 = F \times D$$

- **Signature generation**

  - Decompose an attribute value (or a key word) into a series of triplets
  - Using a hash function to map a triplet to an integer *p*, indicating that the *p*th bit in the signature will be set to 1.

Example: Consider the word “professor”. We will decompose it into 6 triplets:

“pro”, “rof”, “ofe”, “fes”, “ess”, “sor”.

Assume that hash(pro) = 2, hash(rof) = 4, hash(ofe) =8, and hash(fes) = 9.

Signature: 010 100 011 000

- **Signature file**
  - Generate a signature for a record (or a page)

page: ... SGML ... databases ... information ...

word signature:

SGML		010 000 100 110
database		100 010 010 100
information	$\vee$	010 100 011 000
page signature (OS)		110 110 111 110

superimposing

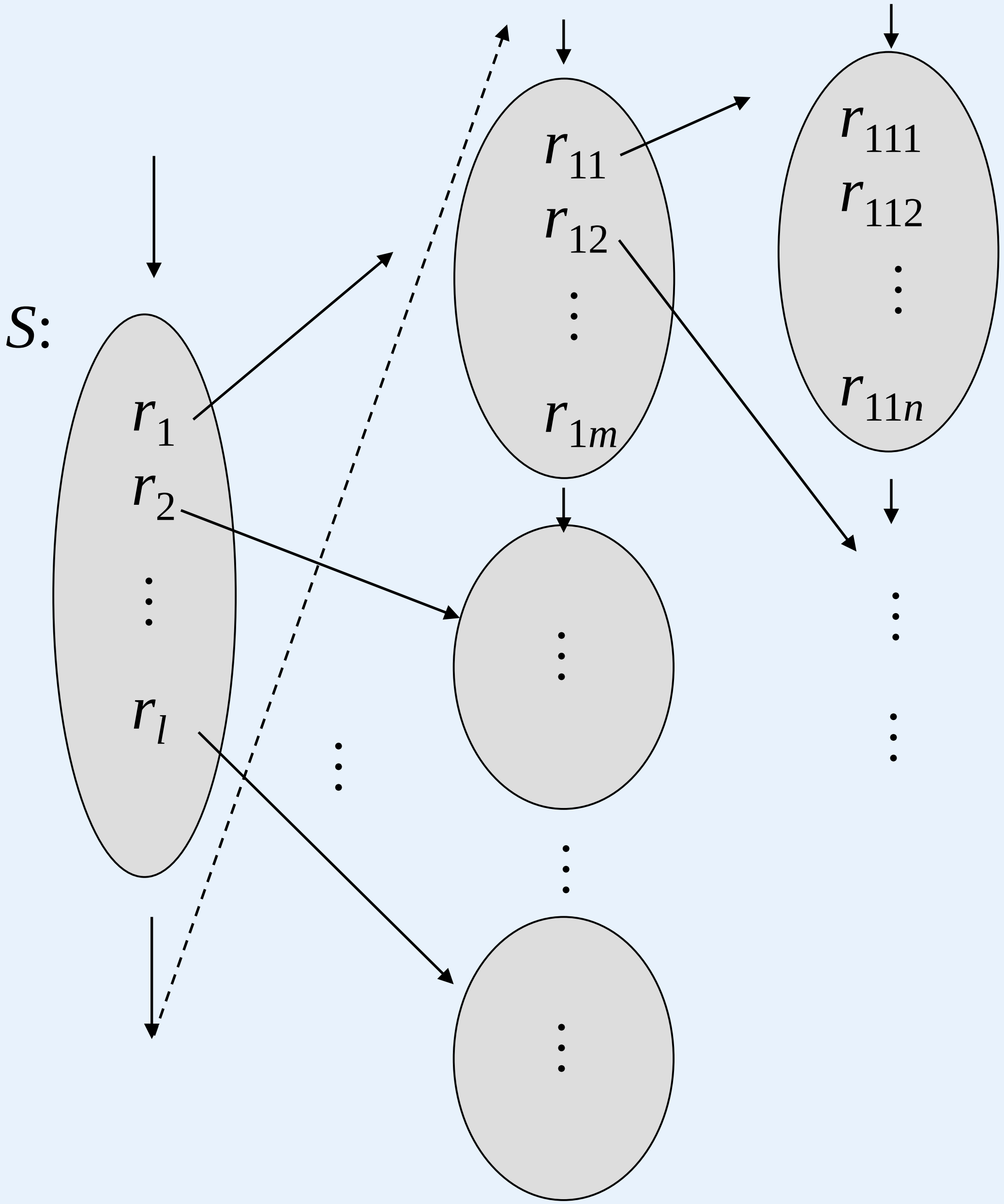
Selecting the next URL from *S*

- Completely random choice of next page.

- Maintain *S* as a queue. Thus, do a breadth-first search of the Web from the starting point or points with which we initialized *S*. Since we presumably start the search from places in the Web that have "important" pages, we are assured of visiting preferentially those portions of the Web.

- Estimate the importance of page links in *S*, and to favor those pages we estimate to be the most important.
  - PageRank
  - Priority queue

DBS and the Internet



Speeding up the Crawl

- More than one crawling machine
- More crawling processes in a machine
- Concurrent access to *S*

Query Processing in Search Engine

- Search engine queries are word-oriented: a boolean combination of words
- Answer: all pages that contain such words
- Method:

  - The first step is to use the inverted index to determine those pages that contain the words in the query.
  - The second step is to evaluate the boolean expression:

The AND of bit vectors (a bit vector represents an inverted list) gives the pages containing both words.
The OR of bit vectors gives the pages containing one or both.

$(\text{word1} \wedge \text{word2}) \vee (\text{word3} \wedge \text{word4})$

DBS and the Internet

word1 appears in document *i*

word1: 10 … 001 … 00 ← Inverted list

$\wedge$ word2: 10 … 101 … 10

---

10 … 001 … 00 ← Show all the documents which contain word1 and word2

word3: 10 … 001 … 01

$\wedge$ Word4: 10 … 101 … 11

---

10 … 001 … 01

(word1 $\wedge$ word2) $\vee$ (word3 $\wedge$ word4):

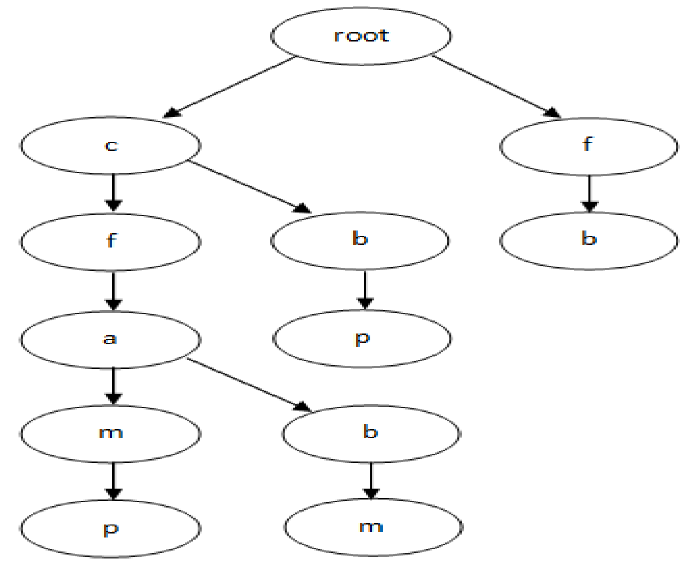
10 … 001 … 00

$\vee$ 10 … 001 … 01

Trie-based Method for Query Processing

- A trie is a multiway tree, in which each path corresponds to a string, and common prefixes in strings to common prefix paths.
- Leaf nodes include either the documents themselves, or links to the documents containing the string that corresponds to the path.

Example:



A trie constructed for The following strings:

s1: cfamp
s2: cbp
s3: cfabm
s4: fb

Trie-based Method for Query Processing

- Item sequences sorted (decreasingly) by appearance frequency (*af*) in documents.

DocID		Items		Sorted item sequence
1		*f, a, c, m, p*		*c, f, a, m, p*
2		*a, b, c, f*		*c, f, a, b*
3		*b, f*		*f, b*
4		*b, c, p*		*c, b, p*
5		*a, f, c, m, p, e*		*c, f, a, m, p, e*

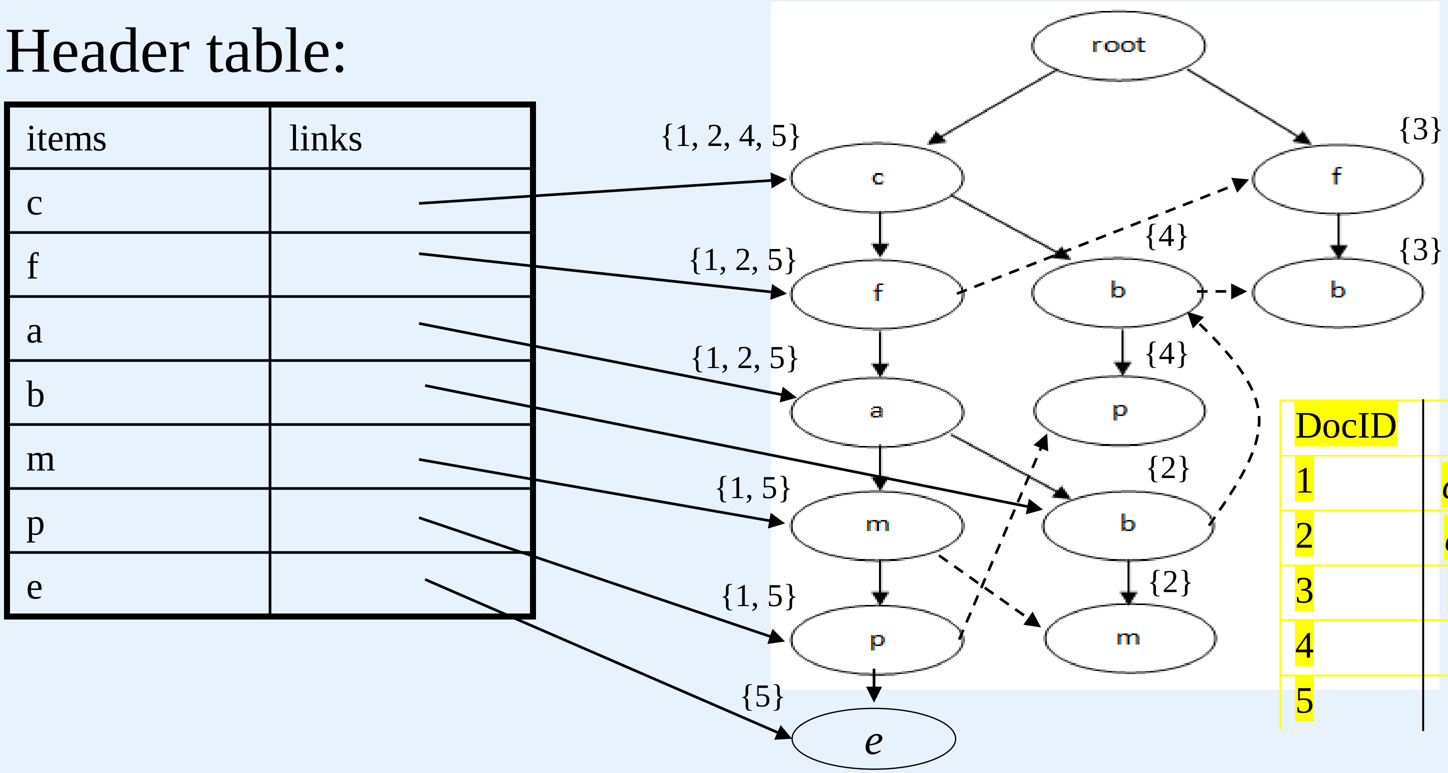
$$af(w) = \frac{\text{No. of doc. Containing } w}{\text{No. of doc.}}$$

- View each sorted item sequence as a string
- Construct a trie over them, in which each node is associated with a set of document IDs each containing the substring represented by the corresponding prefix.

Trie-based Method for Query Processing

- View each sorted item sequence as a string and construct a trie over them.

Header table:



items	links
c	
f	
a	
b	
m	
p	
e	

DocID	Sorted item sequence
1	*c, f, a, m, p*
2	*c, f, a, b, m*
3	*f, b*
4	*c, b, p*
5	*c, f, a, m, p, e*

Trie-based Method for Query Processing

- Evaluation of queries

  - Let Q = word$_1$ $\wedge$ word$_2$ … $\wedge$ word$_k$ be a query
  - Sort increasingly the words in Q according to the appearance frequency:

$$\text{word}_{i_1} \wedge \text{word}_{i_2} \wedge \ldots \wedge \text{word}_{i_k}$$

  - Find a node in the trie, which is labeled with word_{i_1}
  - If the path from the root to word_{i_1} contains all word$_i$ (i = 1, …, k), return the document identifiers associated with word_{i_1}
  - The check can be done by searching the path bottom-up, starting from word_{i_1}. In this process, we will first try to find word_{i_2}, and then word_{i_3}, and so on.

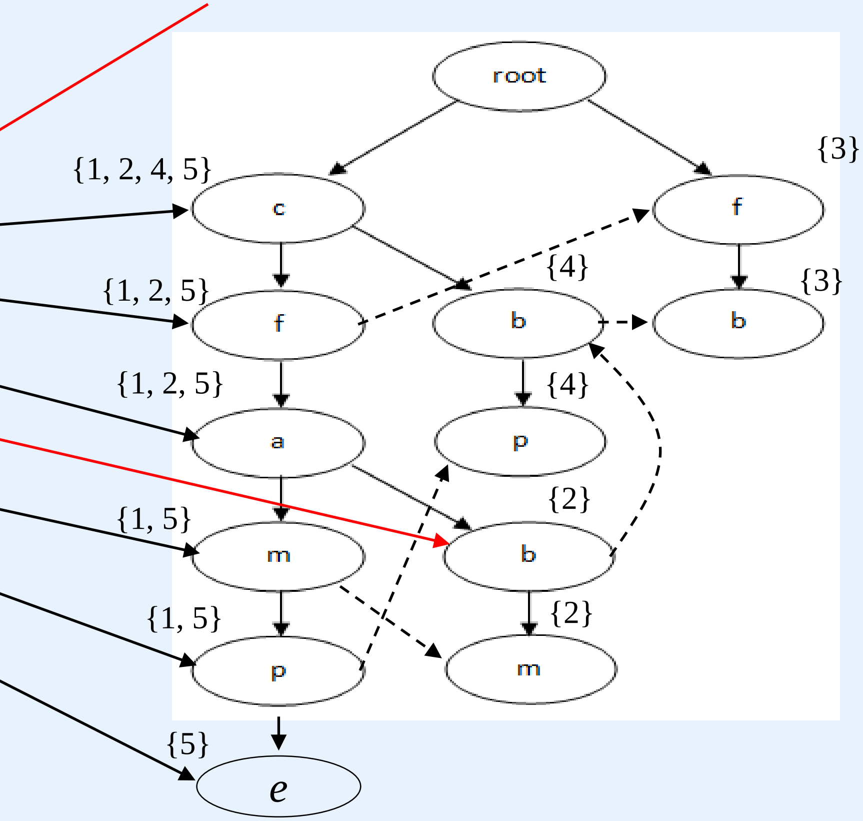
Trie-based Method for Query Processing

- Example

query: $c \wedge b \wedge f \xrightarrow{\text{sorting}} b \wedge f \wedge c$

Header table:

items	links
c	
f	
a	
b	
m	
P	
e	



Ranker: ranking pages

Once the set of pages that match the query is determined, these pages are ranked, and only the highest-ranked pages are shown to the user.

Measuring PageRank:

- The presence of all the query words
- The presence of query words in important positions in the page
- Presence of several query words near each other would be a more favorable indication than if the words appeared in the page, but widely separated.
- Presence of the query words in or near the anchor text in links leading to the page in question.

PageRank for Identifying Important Pages

One of the key technological advances in search is the PageRank algorithm for identifying the “importance” of Web pages.

The Intuition behind PageRank

When you create a page, you tend to link that page to others that you think are important or valuable

A Web page is important if many important pages link to it.

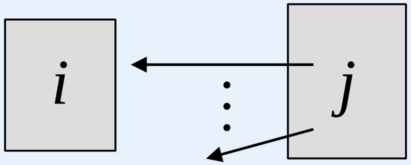
Recursive Formulation of PageRank

The Web navigation can be modeled as random walker move. So we will maintain a *transition matrix* to represent links.

- Number the pages 1, 2, …, *n*.
- The transition matrix **M** has entries m_{ij} in row *i* and column *j*, where:

  1. $m_{ij} = 1/r$ if page *j* has a link to page *i*, and there are a total $r \geq 1$ pages that *j* links to.

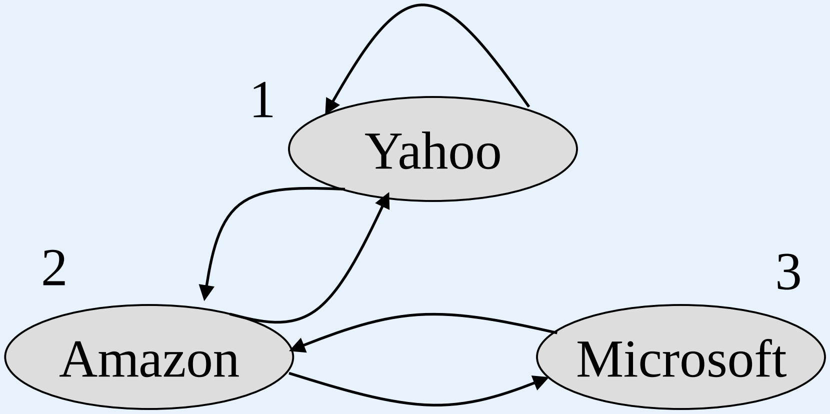
  2. $m_{ij} = 0$ otherwise.



- If every page has at least one link out, then **M** is *stochastic* – elements are nonnegative, and its columns each sum to exactly 1.
- If there are pages with no links out, then the column for that page will be all 0's. **M** is said to be *substochastic* if there are columns sum to less than 1.

DBS and the Internet

$$\begin{array}{cccc} & p1 & p2 & p3 \\ \mathbf{M} = & \frac{1}{2} & \frac{1}{2} & 0 \\ & \frac{1}{2} & 0 & 1 \\ & 0 & \frac{1}{2} & 0 \end{array}$$



Let *y*, *a*, *m* represent the fractions of the time the random walker spends at the three pages, respectively. We have

$$\begin{pmatrix} y \\ a \\ m \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & 1 \\ 0 & \frac{1}{2} & 0 \end{pmatrix} \begin{pmatrix} y \\ a \\ m \end{pmatrix}$$

It is because after a large number of moves, the walker’s distribution of possible locations is the same at each step.

The time that the random walker spends at a page is used as the measurement of “importance”.

DBS and the Internet

$$\begin{bmatrix} y \\ a \\ m \end{bmatrix} = \begin{pmatrix} ½ & ½ & 0 \\ ½ & 0 & 1 \\ 0 & ½ & 0 \end{pmatrix} \begin{bmatrix} y \\ a \\ m \end{bmatrix}$$

$$y = ½ \cdot y + ½ \cdot a + 0 \cdot m$$

$$a = ½ \cdot y + 0 \cdot a + 1 \cdot m$$

$$m = 0 \cdot y + ½ \cdot a + 0 \cdot m$$

DBS and the Internet

$y = \frac{1}{2} \cdot y + \frac{1}{2} \cdot a + 0 \cdot m$ $P(y) = \frac{1}{2} \cdot P(y) + \frac{1}{2} \cdot P(a) + 0 \cdot P(m)$

$a = \frac{1}{2} \cdot y + 0 \cdot a + 1 \cdot m$ $P(a) = \frac{1}{2} \cdot P(y) + 0 \cdot a\, P(a) + 1 \cdot P(y)$

$m = 0 \cdot y + \frac{1}{2} \cdot a + 0 \cdot m$ $P(m) = 0 \cdot P(y) + \frac{1}{2} \cdot P(a) + 0 \cdot P(m)$

$$P(y) = P(y \mid y) \cdot P(y) + P(y \mid a) \cdot P(a) + P(y \mid m) \cdot P(m)$$

$$P(a) = P(a \mid y) \cdot P(y) + P(a \mid a) \cdot P(a) + P(a \mid m) \cdot P(m)$$

$$P(m) = P(m \mid y) \cdot P(y) + P(m \mid a) \cdot P(a) + P(m \mid m) \cdot P(m)$$

Conditional probability

DBS and the Internet

Solutions to the equation:

$$\begin{pmatrix} y \\ a \\ m \end{pmatrix} = \begin{pmatrix} ½ & ½ & 0 \\ ½ & 0 & 1 \\ 0 & ½ & 0 \end{pmatrix} \begin{pmatrix} y \\ a \\ m \end{pmatrix}$$

- If (y_0, a_0, m_0) is a solution to the equation, then (cy_0, ca_0, cm_0) is also a solution for any constant c.

- $y_0 + a_0 + m_0 = 1$.

Gaussian elimination method – $O(n^3)$. If n is large, the method cannot be used. (Consider billions pages!)

Approximation by the method of *relaxation*:

- Start with some estimate of the solution and repeatedly multiply the estimate by **M.**
- As long as the columns of **M** each add up to 1, then the sum of the values of the variables will not change, and eventually they converge to the distribution of the walker's location.
- In practice, 50 to 100 iterations of this process suffice to get very close to the exact solution.

Suppose we start with (*y*, *a*, *m*) = (1/3, 1/3, 1/3). We have

$$\begin{pmatrix} 2/6 \\ 3/6 \\ 1/6 \end{pmatrix} = \begin{pmatrix} ½ & ½ & 0 \\ ½ & 0 & 1 \\ 0 & ½ & 0 \end{pmatrix} \begin{pmatrix} 1/3 \\ 1/3 \\ 1/3 \end{pmatrix}$$

DBS and the Internet

At the next iteration, we multiply the new estimate (2/6, 3/6, 1/6) by **M**, as:

$$\begin{pmatrix} 5/12 \\ 4/12 \\ 3/12 \end{pmatrix} = \begin{pmatrix} ½ & ½ & 0 \\ ½ & 0 & 1 \\ 0 & ½ & 0 \end{pmatrix} \begin{pmatrix} 2/6 \\ 3/6 \\ 1/6 \end{pmatrix}$$

If we repeat this process, we get the following sequence of vectors:

$$\begin{pmatrix} 9/24 \\ 11/24 \\ 4/24 \end{pmatrix}, \begin{pmatrix} 20/48 \\ 17/48 \\ 11/48 \end{pmatrix}, \ldots., \begin{pmatrix} 2/5 \\ 2/5 \\ 1/5 \end{pmatrix}$$

Spider Traps and Dead Ends

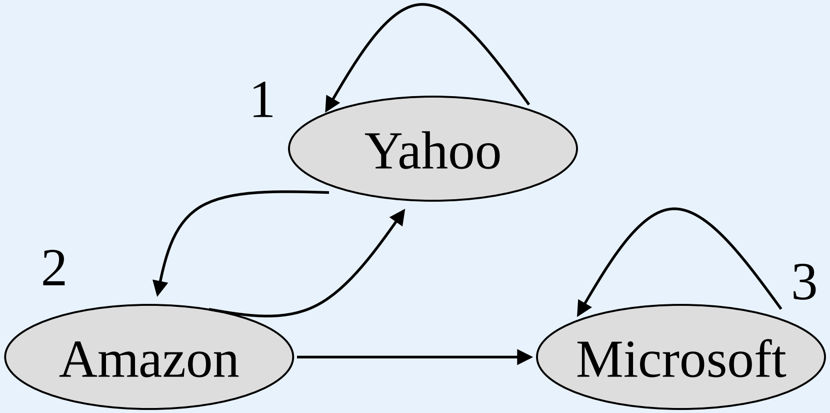
- Spider traps. There are sets of Web pages with the property that if you enter that set of pages, you can never leave because there are no links from any page in the set to any page outside the set.
- Dead ends. Some Web pages have no out-links. If the random walker arrives at such a page, there is no place to go next, and the walk ends.
  - Any dead end is, by itself, a spider trap. Any page that links only to itself is a spider trap.
  - If a spider trap can be reached from outside, then the random walker may wind up there eventually and never leave.

Spider Traps and Dead Ends

Problem:

Applying relaxation to the matrix of the Web with spider traps can result in a limiting distribution where all probabilities outside a spider trap are 0.

Example.



$$\mathbf{M} = \begin{array}{c} \begin{array}{ccc} p1 & p2 & p3 \end{array} \\ \begin{pmatrix} ½ & ½ & 0 \\ ½ & 0 & 0 \\ 0 & ½ & 1 \end{pmatrix} \end{array}$$

DBS and the Internet

Solutions to the equation:

$$\begin{pmatrix} y \\ a \\ m \end{pmatrix} = \begin{pmatrix} ½ & ½ & 0 \\ ½ & 0 & 1 \\ 0 & ½ & 0 \end{pmatrix} \begin{pmatrix} y \\ a \\ m \end{pmatrix}$$

Initially, $\begin{pmatrix} y \\ a \\ m \end{pmatrix} = \begin{pmatrix} 1/3 \\ 1/3 \\ 1/3 \end{pmatrix}$

$$\begin{pmatrix} 1/3 \\ 1/3 \\ 1/3 \end{pmatrix} \quad \begin{pmatrix} 2/6 \\ 1/6 \\ 3/6 \end{pmatrix} \quad \begin{pmatrix} 3/12 \\ 2/12 \\ 7/12 \end{pmatrix} \quad \begin{pmatrix} 5/24 \\ 3/24 \\ 16/24 \end{pmatrix} \quad \begin{pmatrix} 8/48 \\ 5/48 \\ 35/48 \end{pmatrix} \quad \dots \; \dots \quad \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

This shows that with probability 1, the walker will eventually wind up at the Microsoft page (page 3) and stay there.

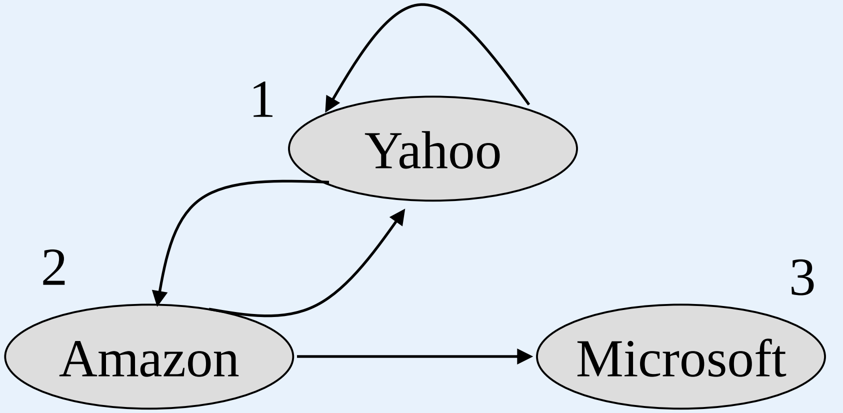
Problem Caused by Spider Traps

- If we interpret these PageRank probabilities as “importance” of pages, then the Microsoft page has gathered all importance to itself simply by choosing not to link outside.
- The situation intuitively violates the principle that other pages, not you yourself, should determine the importance of your page.

Problem Caused by Dead Ends

- The dead end also cause the PageRank not to reflect importance of pages.

Example.



$$\mathbf{M} = \begin{array}{c} \\ \\ \\ \\ \end{array}\begin{array}{ccc} p1 & p2 & p3 \\ \vdots & \vdots & \vdots \end{array}$$

$$\mathbf{M} = \begin{pmatrix} ½ & ½ & 0 \\ ½ & 0 & 0 \\ 0 & ½ & 0 \end{pmatrix}$$

$$\begin{pmatrix} 1/3 \\ 1/3 \\ 1/3 \end{pmatrix} \quad \begin{pmatrix} 2/6 \\ 1/6 \\ 1/6 \end{pmatrix} \quad \begin{pmatrix} 3/12 \\ 2/12 \\ 1/12 \end{pmatrix} \quad \begin{pmatrix} 5/24 \\ 3/24 \\ 2/24 \end{pmatrix} \quad \begin{pmatrix} 8/48 \\ 5/48 \\ 3/48 \end{pmatrix} \quad \cdots \; \cdots \quad \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

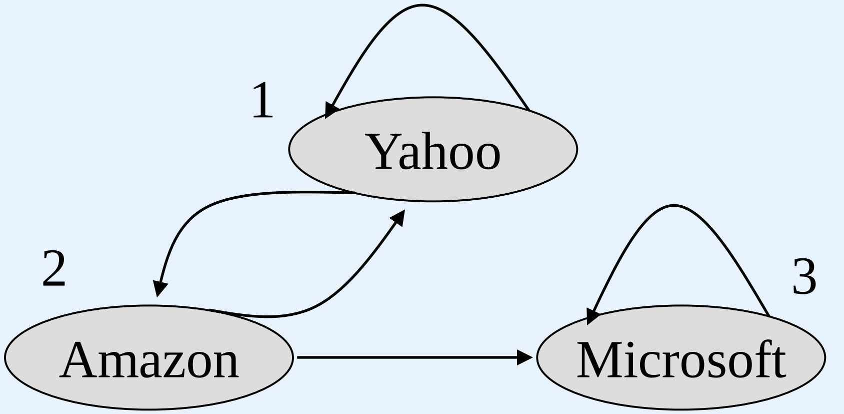
PageRank Accounting for Spider Traps and Dead Ends

We simulate the web navigation by a random walk. Each time a walker goes to a page, we let the walker follow a random out-link, if there is one, with probability β (normally, $0.8 \leq \beta \leq 0.9$). With probability $1 - \beta$ (called the taxation rate), we remove that walker and deposit a new walker at a randomly chosen Web page.

- If the walker gets stuck in a spider trap, it doesn’t matter because after a few time steps, that walker will disappear and be replaced by a new walker.
- If the walker reaches a dead end and disappears, a new walker will take over shortly.

DBS and the Internet

Example.



$$\mathbf{M} = \begin{array}{c} \\ \\ \\ \end{array}\begin{array}{ccc} p1 & p2 & p3 \\ \vdots & \vdots & \vdots \end{array}$$

$$\mathbf{M} = \begin{pmatrix} ½ & ½ & 0 \\ ½ & 0 & 0 \\ 0 & ½ & 1 \end{pmatrix}$$

Let $\mathbf{P}_{new}$ and $\mathbf{P}_{old}$ be the new and old distributions of the location of the walker after one iteration, the relationship between these two can be expressed as:

$$\mathbf{P}_{new} = \underset{\beta}{0.8} \begin{pmatrix} ½ & ½ & 0 \\ ½ & 0 & 0 \\ 0 & ½ & 1 \end{pmatrix} \mathbf{P}_{old} + \underset{1 - \beta}{0.2} \begin{pmatrix} 1/3 \\ 1/3 \\ 1/3 \end{pmatrix}$$

DBS and the Internet

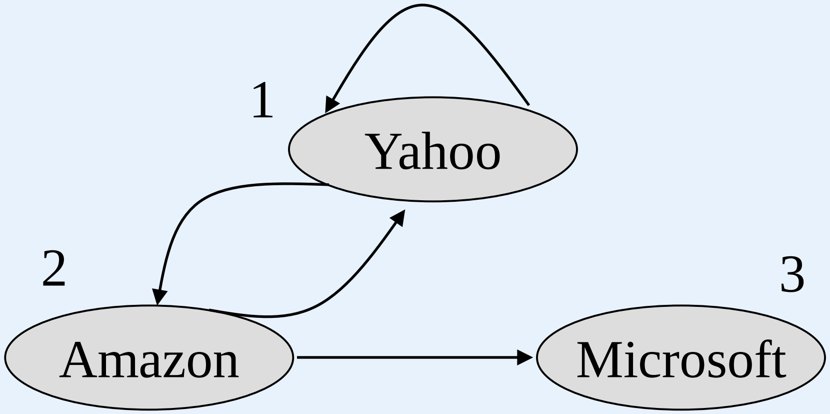
The meaning of the above equation is:
With probability 0.8, we multiply $\mathbf{P}_{old}$ by the matrix of the Web to get the new location of the walker, and with probability 0.2 we start with a new walker at a random place.

If we start with $\mathbf{P}_{old}$ = (1/3, 1/3, 1/3) and repeatedly compute $\mathbf{P}_{new}$ and then replace $\mathbf{P}_{old}$ by $\mathbf{P}_{new}$, we get the following sequence of approximation to the asymptotic distribution of the walker:

$$\begin{pmatrix} .333 \\ .333 \\ .333 \end{pmatrix} \quad \begin{pmatrix} .333 \\ .200 \\ .467 \end{pmatrix} \quad \begin{pmatrix} .280 \\ .300 \\ .520 \end{pmatrix} \quad \begin{pmatrix} .259 \\ .179 \\ .563 \end{pmatrix} \quad \ldots\ \ldots \quad \begin{pmatrix} 7/33 \\ 5/33 \\ 21/33 \end{pmatrix}$$

DBS and the Internet

Example.



$$\mathbf{M} = \begin{array}{c} \\ \\ \end{array}\begin{array}{ccc} p1 & p2 & p3 \\ \vdots & \vdots & \vdots \end{array}$$

$$\mathbf{M} = \begin{pmatrix} ½ & ½ & 0 \\ ½ & 0 & 0 \\ 0 & ½ & 0 \end{pmatrix}$$

$$\mathbf{P}_{new} = \underset{\beta}{0.8} \begin{pmatrix} ½ & ½ & 0 \\ ½ & 0 & 0 \\ 0 & ½ & 0 \end{pmatrix} \mathbf{P}_{old} + \underset{1\ -\ \beta}{0.2} \begin{pmatrix} 1/3 \\ 1/3 \\ 1/3 \end{pmatrix}$$

DBS and the Internet

If we start with $\mathbf{P}_{old}$ = (1/3, 1/3, 1/3) and repeatedly compute $\mathbf{P}_{new}$ and then replace $\mathbf{P}_{old}$ by $\mathbf{P}_{new}$, we get the following sequence of approximation to the asymptotic distribution of the walker:

$$\begin{pmatrix} .333 \\ .333 \\ .333 \end{pmatrix} \begin{pmatrix} .333 \\ .200 \\ .200 \end{pmatrix} \begin{pmatrix} .280 \\ .200 \\ .147 \end{pmatrix} \begin{pmatrix} .259 \\ .179 \\ .147 \end{pmatrix} , \ldots, \begin{pmatrix} 35/165 \\ 25/165 \\ 21/165 \end{pmatrix}$$

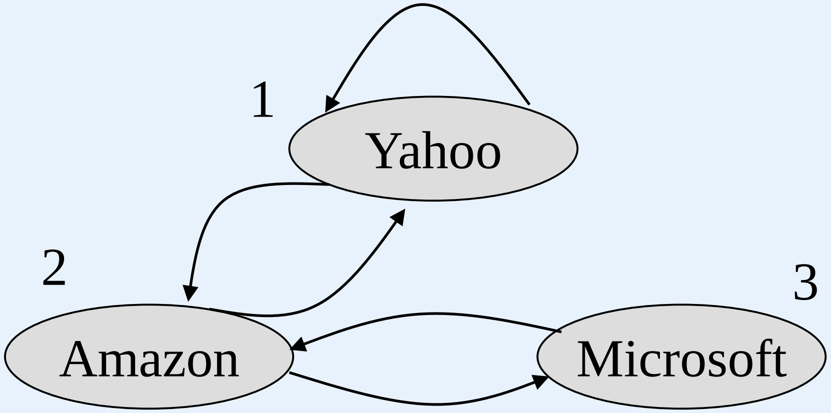
Notice that these probabilities do not sum to one, and there is slightly more than 50% probability that the walker is “lost” at any given time. However, the ratio of the importance of Yahoo!, and Amazon are the same as in the above example. That makes sense because in both the cases there are no links from the Microsoft page to influence the importance of Yahoo! or Amazon.

Topic-Specific PageRank

The calculation o PageRank should be biased to favor certain pages.

Teleport Sets

Choose a set of pages about a certain topic (e.g., sport) as a teleport set.



Assume that we are interested only in retail sales, so we choose a teleport set that consists of Amazon alone.

DBS and the Internet

$$\begin{pmatrix} y \\ a \\ m \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & 1 \\ 0 & \frac{1}{2} & 0 \end{pmatrix} \begin{pmatrix} y \\ a \\ m \end{pmatrix}$$

$$\Downarrow$$

$$\begin{pmatrix} y \\ a \\ m \end{pmatrix} = 0.8 \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & 1 \\ 0 & \frac{1}{2} & 0 \end{pmatrix} \begin{pmatrix} y \\ a \\ m \end{pmatrix} + 0.2 \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

The entry for Amazon is set to 1.

Topic-Specific PageRank

The *general rule* for setting up the equations in a topic-specific PageRank problem is as follows.

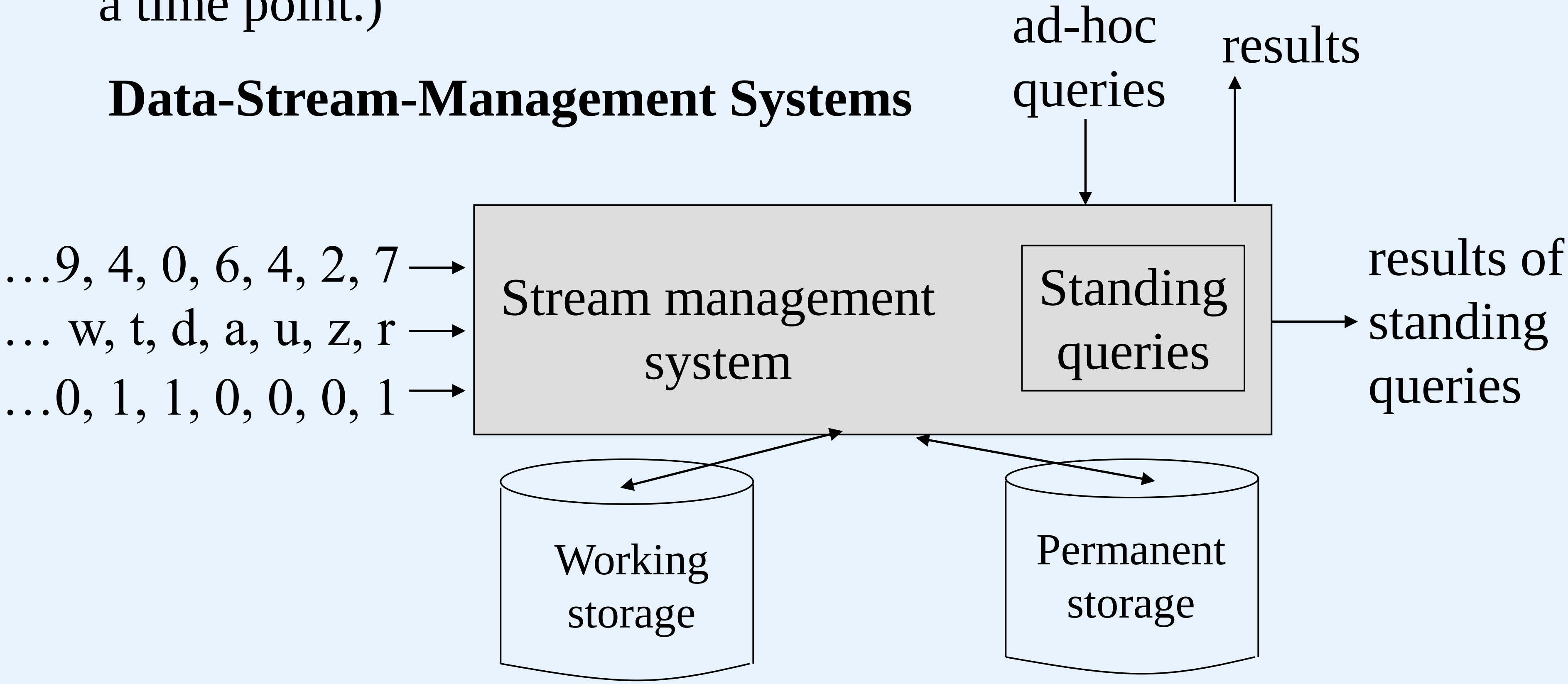
Suppose there are *k* pages in the teleport set. Let **T** be a column-vector that has 1/*k* in the positions corresponding to members of the teleport set and 0 elsewhere. Let **M** be the transition matrix of the Web. Then, we must solve by relaxation the following iterative rule:

$$\mathbf{P}_{new} = \beta\mathbf{M}\mathbf{P}_{old} + (1 - \beta)\mathbf{T} \qquad \mathbf{T} = \begin{pmatrix} 0 \\ 1/k \\ 0 \\ \vdots \\ 1/k \\ \vdots \end{pmatrix}$$

Data Streams

A data steam is a sequence of tuples, which may be unbounded. (Note that a relation is a set of tuples. The set is always bounded at a time point.)

Data-Stream-Management Systems



DBS and the Internet

Using the PostgreSQL DDL, we can not only define tables, but also data streams:

```
CREATE STREAM streamname (colname datatype, …)
        type ARCHIVED | UNARCHIVED
```

- ARCHIVED - streams will take data that is received from a wrapper and insert it into a relation named ‘streamname’.
- UNARCHIVED - An unarchived stream is never backed by disk storage, and is implemented in terms of shared-memory vectors.
- Wrapper – a component to receive data stream from a source (e.g., a sensor, a seismometer, … .)

A simple wrapper:

```
CREATE WRAPPER mycsvwrapper (
    init=csv_init,
    next=csv_next,
    done=csv_done);
```

By default, the wrapper listens for connections from a data sources on port 5533, by which a simple *perl* script called source.pl is provided to receive data and send them to the wrapper.

CSV - Comma-separated values

DBS and the Internet

Associate a wrapper with a stream

```
ALTER STREAM streamname ADD
WRAPPER mycsvwrapper;
```

Data Streams

The system accepts data streams as input, and also accepts queries. Two kinds of queries:

1. Conventional ad-hoc queries.
2. Standing queries that are stored by the system and run on the input streams at all times.

Example.

Suppose we are receiving streams of radiation levels from sensors around a nuclear electricity power plant.

1. DSMS stores a *sliding window* of each input stream in the "working storage". All readings from all sensors for the past 24 hours.
2. Data from further back in time could be dropped, summarized, or copied in its entirety to the permanent store (archive)

Stream Applications

1. Click streams. A Web site might wish to analyze the clicks it receives. (An increase in clicks on a link may indicate that the link is broken, or that it has become of much more interest recently.)
2. Packet streams. We may wish to analyze the sources and destinations of IP packets that pass through a switch. An unusual increase in packets for a destination may warn of a denial-of-service attack.
3. Sensor data. There are many kinds of sensors whose outputs need to be read and considered collectively, e.g., tsunami warning sensors that record ocean levels at subsecond frequencies or the signals that come from seismometers around the world.

Stream Applications

4. Satellite data. Satellites send back to the earth incredible streams of data, often petabytes per day.
5. Financial data. Trades of stocks, commodities, and other financial instruments are reported as a stream of tuples, each representing one financial transaction. These streams are analyzed by software that looks for events or patterns that trigger actions by traders.

A Data-Stream Data Model

- Each stream consists of a sequence of tuples. The tuples have a fixed relation schema (list of attributes), just as the tuples of a relation do. However, unlike relations, the sequence of tuples in a stream may be unbounded.
- Each tuple has an associated arrival time, at which time it becomes available to DSMS for processing. The DSMS has the option of placing it in the working storage or in the permanent storage, or of dropping the tuple from memory altogether. The tuple may also be processed in simple ways before it is stored.

A Data-Stream Data Model

For any stream, we can define a sliding window, which is a set consisting of the most recent tuples to arrive.

- Time-based. It consists of the tuples whose arrival time is between the current time t and t - τ, where τ is a constant.
- Tuple-based. It consists of the most recent n tuples to arrive for some fixed n.

For a certain stream S, we use the notation $S[W]$ to represent a window, where W is:

1. Range τ, meaning all tuples that arrived within the previous amount of time τ.
2. Row n, meaning the most recent n tuples of the stream; or

DBS and the Internet

Example.

Let Sensors(sensID, temp, time) be a stream, each of whose tuples represent a temperature reading of temp at a certain time by the sensor named sensID.

Sensors[Row 1000]

describes a window on the Sensor stream consisting of the most recent 1000 tuples.

Sensors[Range 10 seconds]

describes a window on the Sensor stream consisting of all tuples that arrived in the past 10 seconds.

Handling Streams as Relations

Each stream window can be handled as a relation, whose content changes rapidly.

Suppose we would like to know, for each sensor, the highest recorded temperature to arrive at the DSMS in the past hour.

```
SELECT sensID, MAX(temp)
FROM Sensors[Range 1 hour]
GROUP BY sensID;
```

Handling Streams as Relations

Suppose that besides the stream Sensors, we also maintain an ordinary relation:

Calibrate(sensID, mult, add),

which gives a multiplicative factor and additive term that are used to correct the reading from each sensor.

```
SELECT MAX(mult*temp + add)
FROM Sensors[Range 1 hour], Calibrate
WHERE Sensors.sensID = Calibrate.sensID
```

The query finds the highest, properly calibrated temperature reported by any sensor in the past hour.

Handling Streams as Relations

Suppose we wanted to give, for each sensor, its maximum temperature over the past hour, but we also wanted the resulting tuples to give the most recent time at which that maximum temperature was recorded.

```
SELECT s.sensID, s.temp, s.time
FROM Sensors[Range 1 Hour] s
WHERE NOT EXISTS (
        SELECT * FROM Sensors[Range 1 Hour]
        WHERE sensID = s.sensID AND (
                temp > s.temp OR
                (temp = s.temp AND time > s.time)
));
```

Stream compression and stream mining

Streams tend to be very large. So they should be compressed to save space.
However, querying a compressed stream can be very difficult.

Consider two problems:

I. Let *S* be a binary stream (a stream of 0's and 1's). We will ask the number of 1's in any time range contained within the window.

II. Let *S* be a stream. We will count the distinct elements in a window on *S*.

I. Let *S* be a binary stream (a stream of 0’s and 1’s). We will ask the number of 1’s in any time range contained within the window.

Assumption:

i) The length of the sliding window is *N*.

ii) The stream began at some time in the past. We associate a *time* with each arriving bit, which is its position; i.e., the first to arrive is at time 1, the next at time 2, and so on.

Our query, which may be asked at any time, is of the form “how many 1’s are there in the most recent *k* bits?” ($1 \leq k \leq N$)

DBS and the Internet

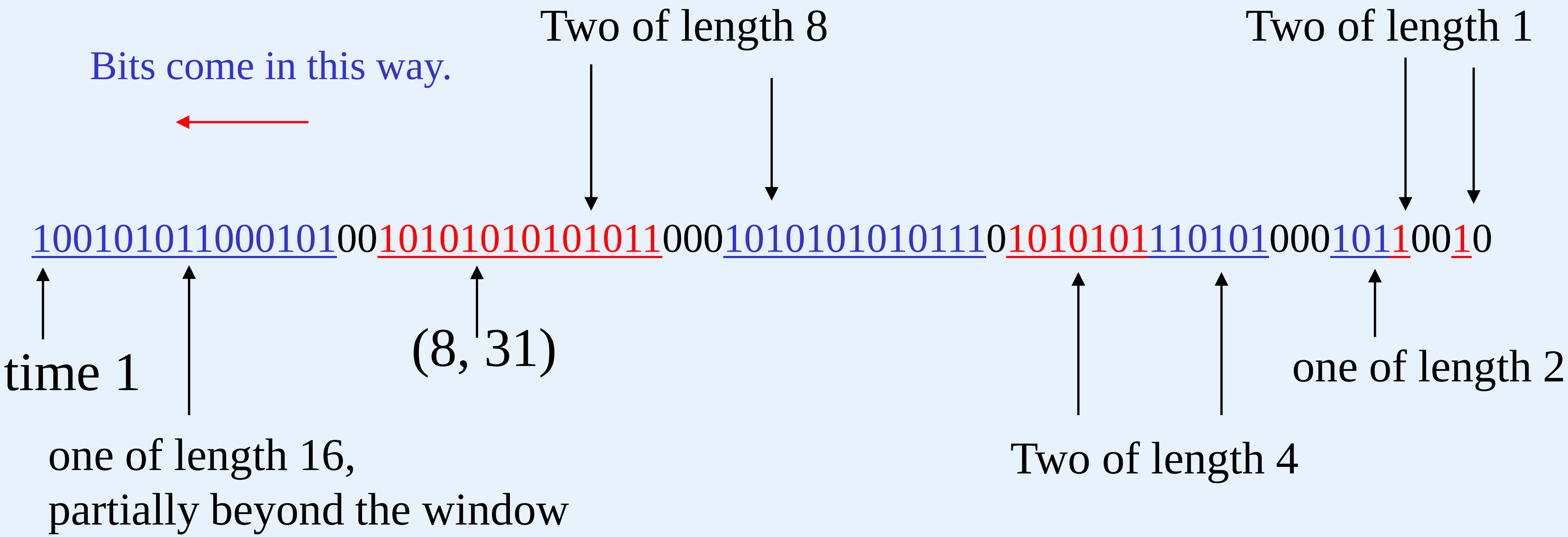
Bucket of size *m* – a section of the window that contains exactly *m* 1's.

So the window will be partitioned completely into such buckets, except possibly for some 0's that are not part of any bucket.

- A bucket is denoted as (*m*, *t*), where *t* is the time of the most recent 1 belonging to the bucket.
- Rules for determining the buckets:
  1. The size of every bucket is a power of 2 (2^i for some *i*).
  2. As we look back in time, the sizes of the buckets never decrease.
  3. For *m* = 1, 2, 4, 8, … up to some largest-size bucket, there are one or two buckets of each size, never zero and never more than two.
  4. Each bucket begins somewhere within the current window, although (part of the largest) bucket may be outside of the window.

DBS and the Internet

- Rules for determining the buckets:



Sequence of bucket sizes: 16, 8, 8, 4, 4, 2, 1, 1

DBS and the Internet

- How to compress buckets, and then compress bit strings?

- How to answer the queries by using compressed buckets?

- How to dynamically construct buckets?

Representing Buckets

A bucket size can be represented by O(log N) bits. Furthermore, there are at most O(log N) buckets in a window that must be represented. Thus, a window of length N an be represented in space O($\log^2 N$), rather than O(N) bits.

- A bucket (m, t) can be represented in O(log N) bits. First, m, the size of a bucket, can never get above N. Moreover, m is always a power of 2, so we don't have to represent m itself, rather we can represent $\log_2 m$. That requires O(log N) bits. To represent t, the time of the most recent 1 in the bucket, we need another O(log N) bits. In principle, t can be an arbitrarily large integer, but it suffices to represent t modulo N since t is in the window of size N.

1001010110001010010101010101011000101010101011101010101110101000101100100

$(16, t_1)(8, t_2)(8, t_3)(4, t_4)(4, t_5)(2, t_6)(1, t_7)(1, t_8)$

- There can be only O(log N) buckets. The sum of the sizes of the buckets is at most N, and there can be at most two of any size. If there are more than $2 + 2\log_2 N$ buckets, then the largest one is of size at least

$$2 \times 2^l \ (l = \log_2 N),$$

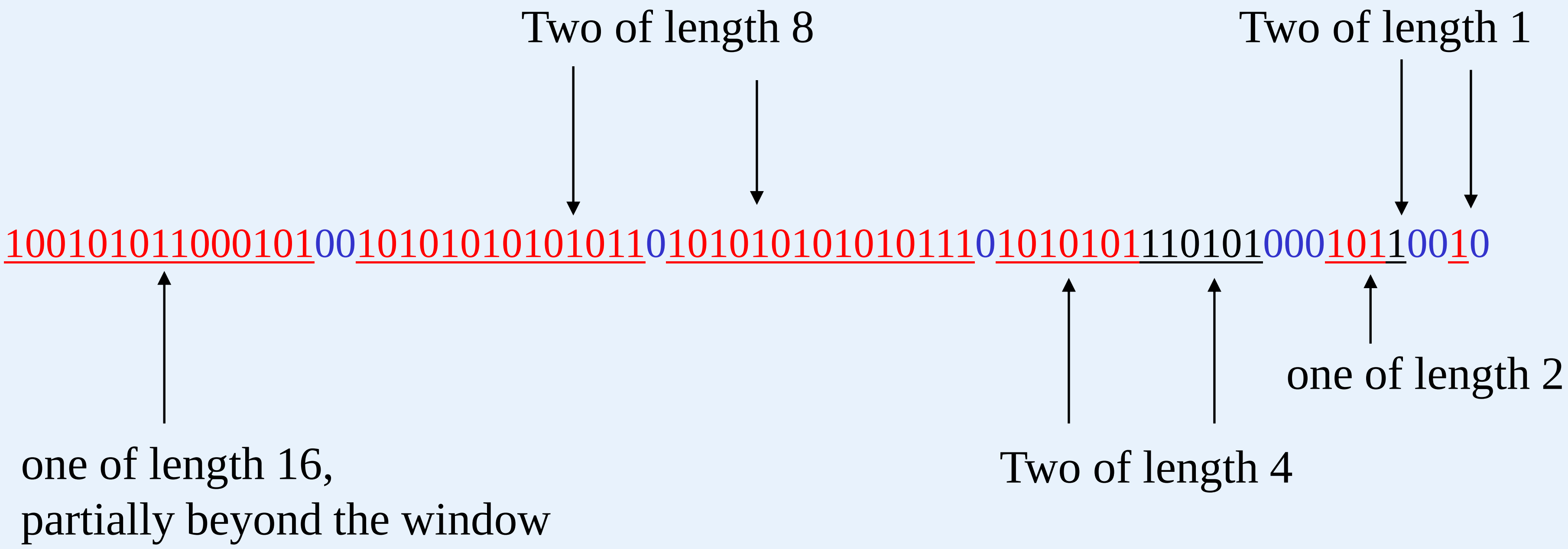
which is $2N$. Therefore, there are at most O(log N) buckets.

Answering queries approximately, using buckets

How many 1's are there in the most recent k bits?

- Find the least recent bucket ***B*** whose most recent bit arrives within the last k time units.
- All later buckets are entirely within the range of k time units.
- How many 1's in each of these buckets is known. It is their sizes.
- The bucket ***B*** may be partially in the query's range, and partially outside it. So we choose half its size as the best guess.

DBS and the Internet


Two of length 8

Two of length 1

10010101100010100101010101010110101010101010111010101011101010001011001 0

one of length 2

one of length 16,
partially beyond the window

Two of length 4

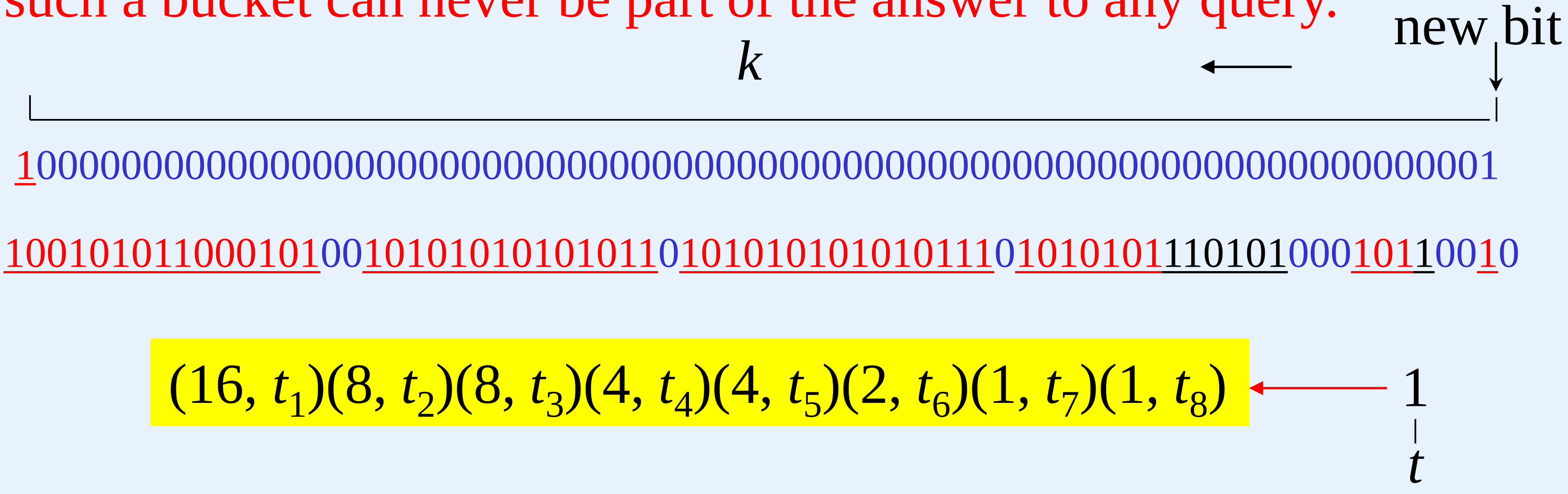
Suppose $k = N$. We see two buckets of size 1 and one of size 2, which implies four 1's. Then, there are two buckets of size 4, giving another eight 1's, and two buckets of size 8, implying another sixteen 1's. Finally, the last bucket, of size 16, is partially in the window, so we add another 8 to the estimate.

$$2 \times 1 + 1 \times 2 + 2 \times 4 + 2 \times 8 + 8 = 36.$$

Maintaining Buckets

We consider two cases.

Case 1: If a new bit arrives, and the last bucket now has a most recent bit that is more than *k* time units before the time of the arriving bit. In this case, we can drop that bucket from the representation since such a bucket can never be part of the answer to any query.



100000000000000000000000000000000000000000000000000000000000000000000001

100101011000101001010101010101101010101010101110101010111010100010110010

$(16, t_1)(8, t_2)(8, t_3)(4, t_4)(4, t_5)(2, t_6)(1, t_7)(1, t_8)$

If $t - t_1 > k$, remove $(16, t_1)$ from the above bucket sequence.

DBS and the Internet

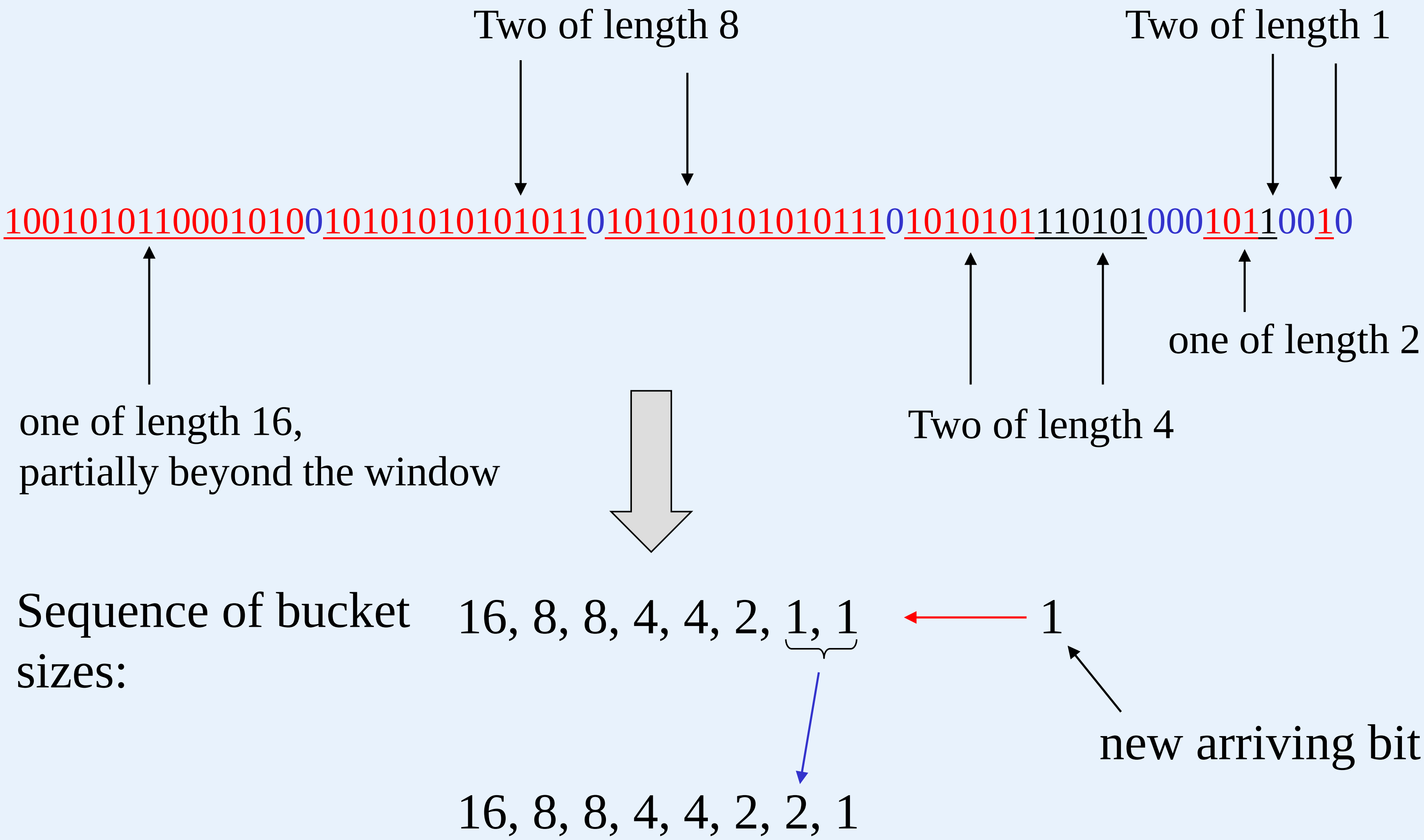
Case 2: The time of the arriving bit and the most recent bit in the last bucket are within the *k* time units.

If the new bit is 0, nothing will be done.

Otherwise, a new bucket of size 1 (representing just that bit) is created, which may cause a recursive combining-buckets process.

- Suppose we have three consecutive buckets of size m, say (m, t_1), (m, t_2) and (m, t_3), where $t_1 < t_2 < t_3$. We combine the two least recent of the buckets, (m, t_1), (m, t_2) , into one bucket of size $2m$: $(2m, t_2)$. (Note that (m, t_1) disappears.)
- This combination may cause three consecutive buckets of size $2m$ if there were two of that size previously. Thus, we apply the combination algorithm recursively, with the size now $2m$. (It can take O(log$N$) time to do all the necessary combinations.)

DBS and the Internet



Sequence of bucket sizes: 16, 8, 8, 4, 4, 2, 1, 1

16, 8, 8, 4, 4, 2, 2, 1

II. Let *S* be a stream. We will count the distinct elements in a window on *S*.

Applications:

1. The popularity of a Web site is often measured by unique visitors per month or similar statistics. Think of the logins at a site like Yahoo! as a stream. Using a window of size one month, we want to know how many different logins there are.
2. Suppose a crawler is examining sites. We can think of the words encountered on the pages as forming a stream. If a site is legitimate, the number of distinct words will fall in a range that is neither too high (few repetitions of words) nor too low (expressive repetitions of words). Falling outside that range suggests that the site could be artificial, e.g., a spam site.

DBS and the Internet

N – a number, at least as large as the number of distinct values in a stream.

R – a variable to record the number of different values. Initially $R = 0$.

h – a hash function that maps values to $\log_2 N$ bits.

As each stream value v arrives, do the following:

1. Compute $h(v)$.
2. Let i be the number of *trailing* 0's in $h(v)$.
3. If $i > R$, set R to be i.

Then, the estimate of the number of distinct values seen so far is 2^R.

If the data stream contains m different values, then R is about $\log_2 m$.

Argument

a) The probability that $h(v)$ ends in at least i *trailing* 0's is 2^{-i}. (The probability that a bit is 0 in a bit string is ½.)

b) If there are m distinct values in the stream so far, the probability that each of the m words has less than i trailing 0's is $(1 - 2^{-i})^m$ and then the probability that $R < i$ is $(1 - 2^{-i})^m$.

c) If i is much less than $\log_2 m$, then this probability is close to 0 (i.e. the probability that $R < i$ is close 0. So R is not much less than $\log_2 m$). If i is much larger than $\log_2 m$, then this probability is close to 1 (thus R is definitely smaller than i and close to $\log_2 m$.)

d) Thus, R will be frequently close to $\log_2 m$, and 2^R (our estimate) will be frequently near m.

1. Compute $h(v)$.
2. Let i be the number of *trailing* 0's in $h(v)$.
3. If $i > R$, set R to be i.

$(1 - 2^{-i})$ – the probability that for a value v $h(v)$ ends at less than i trailing 0's.

DBS and the Internet

If the data stream contains m different values, then R is about $\log_2 m$.

If the data stream contains m different values, and each v of such m values is with i trailing 0’s in h (v) with i much smaller than $\log_2 m$, then the probability of $R < i$

$$(1 - 2^{-i})^m.$$

is close to 0. (That is, R is not likely $< i < \log_2 m$.) So, R should be $\geq \log_2 m$.

But for any v of such m values, if the number i of trailing 0’s in h(v) is much larger than $\log_2 m$, the probability of $R < i$

$$(1 - 2^{-i})^m$$

is close to 1. So, we have R close $\log_2 m$.

DBS and the Internet

Argument

a) The probability that $h(v)$ ends in at least i *trailing* 0’s is 2^{-i}. (The probability that a bit is 0 in a bit string is ½.)

b) If there are m distinct values in the stream so far, the probability that each of the m words has less than i trailing 0’s is $(1 – 2^{-i})^m$ and then the probability that $R < i$ is $(1 – 2^{-i})^m$.

c) If i is much less than $\log_2 m$, then this probability is close to 0 (i.e. the probability that $R < i$ is close 0. So R is not much less than $\log_2 m$). If i is much larger than $\log_2 m$, then this probability is close to 1 (thus R is definitely smaller than i and close to $\log_2 m$.)

c) shows that the probability that R is much < $\log_2 m$ is close to 0, and the probability of $R < i$ is close to 1 if i is much larger than $\log_2 m$.

d) Thus, R will be frequently close to $\log_2 m$, and 2^R (our estimate) will be frequently near m.

$(1 – 2^{-i})$ – the probability that for a value u $h(u)$ ends at less than i trailing 0’s.

Discussion

While the above argument is comforting, it is actually inaccurate. Especially, the expected value of 2^R is infinite, or at least it is as large as possible given that N is finite. The intuitive reason is that, for large R, when R increases by 1, the probability of R being that large halves, but the value of R doubles, so each possible value of R contributes the same to the expected value.

It is therefore necessary to get around the fact that there will occasionally be value of R that is so large it biases the estimate of m upwards. But we can avoid this bias by

a) Take many estimates of R, using different hash functions.
b) Group these estimates into small groups and take the median of each group. Doing so estimates the effect of occasional large R's.
c) Take the average of medians of the groups.