Introduction to

Information Retrieval

ΠΛΕ70: Ανάκτηση Πληροφορίας

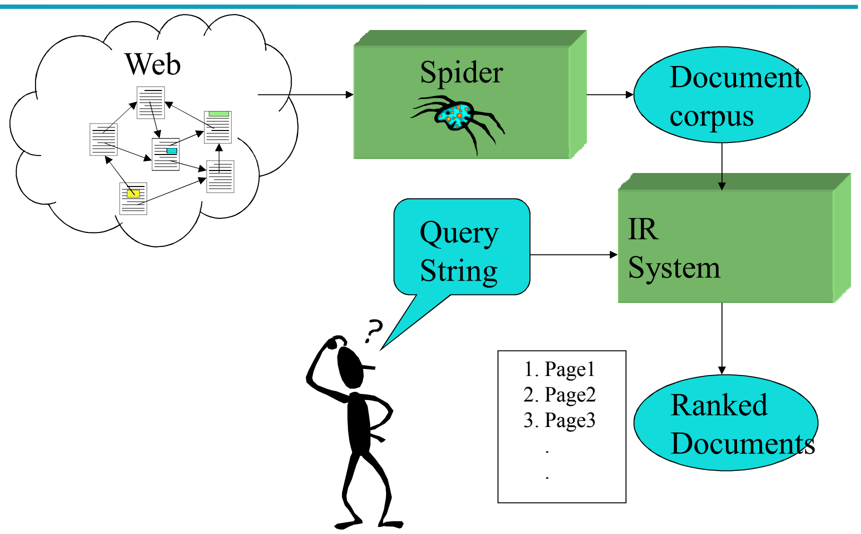
Διδάσκουσα: Ευαγγελία Πιτουρά

Διάλεξη 10: Σταχυολόγηση Ιστού και Ευρετήρια.

Τι θα δούμε σήμερα

1. Web crawlers or spiders (κεφ 20)
2. Personalization/Recommendations
3. Lucene

Spiders (σταχυολόγηση ιστού)



Web Crawling (σταχυολόγηση ιστού)

Web crawler or spider

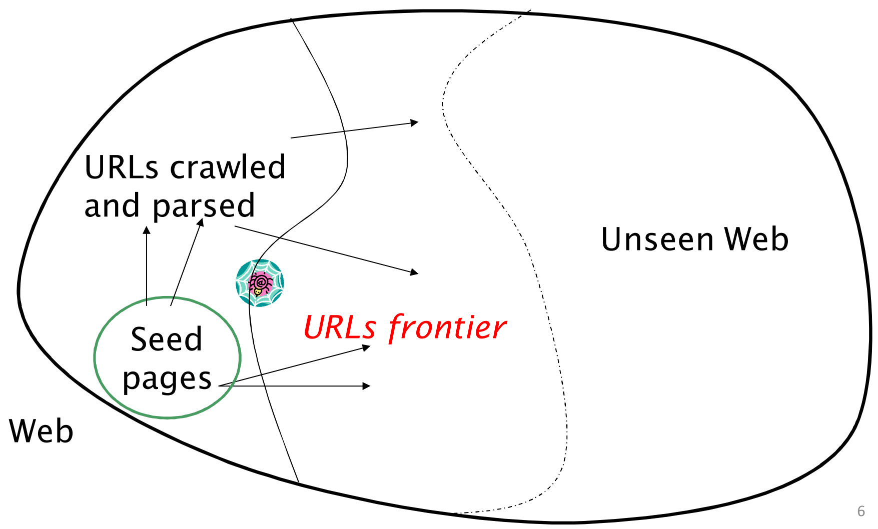
How hard and why?

- Getting the content of the documents is easier for many other IR systems.
  - E.g., indexing all files on your hard disk: just do a recursive descent on your file system
- For web IR, getting the content of the documents takes longer, because of latency.
  - But is that really a design/systems challenge?

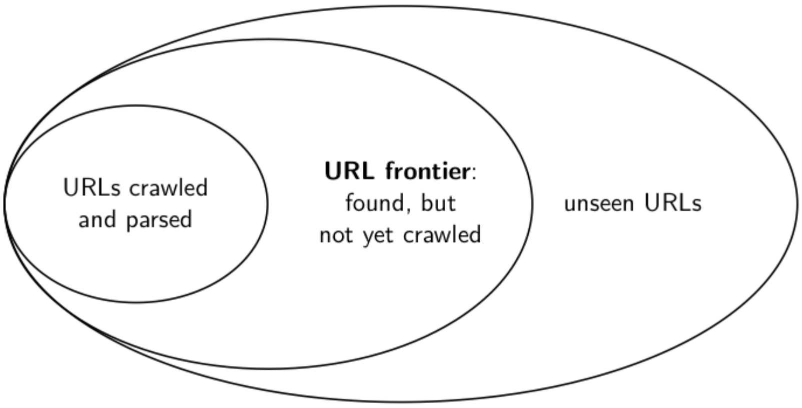
Βασική λειτουργία

- Begin with known “seed” URLs
- Fetch and parse them
  - Extract URLs they point to
  - Place the extracted URLs on a queue
- Fetch each URL on the queue and repeat

Crawling picture



URL frontier



Simple picture – complications

- Web crawling isn't feasible with one machine
  - All of the above steps distributed
- Malicious pages
  - Spam pages
  - Spider traps – incl dynamically generated
- Even non-malicious pages pose challenges
  - Latency/bandwidth to remote servers vary
  - Webmasters' stipulations
    - How "deep" should you crawl a site's URL hierarchy?
  - Site mirrors and duplicate pages
- Politeness – don't hit a server too often

Simple picture – complications

Magnitude of the problem

To fetch 20,000,000,000 pages in one month . . .
we need to fetch almost 8000 pages per second!

- Actually: many more since many of the pages we attempt to crawl will be duplicates, unfetchable, spam etc.

What any crawler *must* do

- Be Polite: Respect implicit and explicit politeness considerations
  - Only crawl allowed pages
  - Respect *robots.txt* (more on this shortly)
- Be Robust: Be immune to spider traps and other malicious behavior from web servers (very large pages, very large websites, dynamic pages etc)

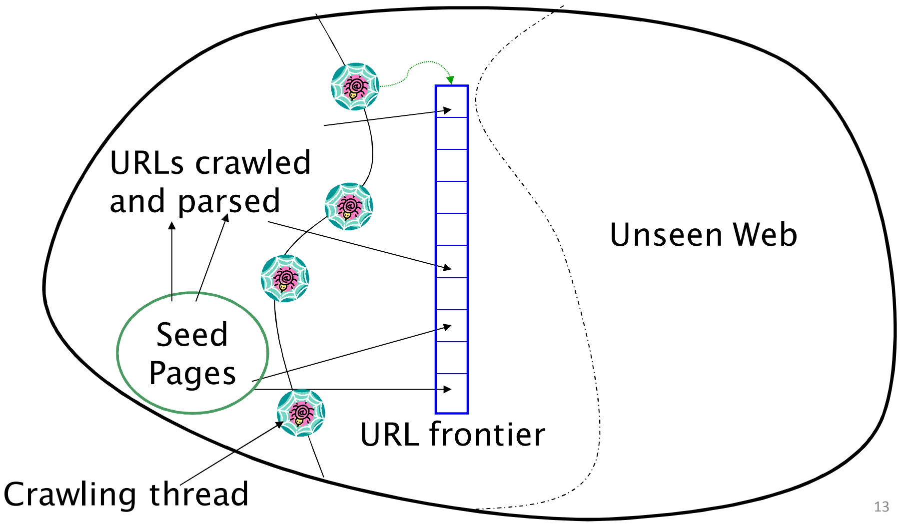
What any crawler *should* do

- Be capable of distributed operation: designed to run on multiple distributed machines
- Be scalable: designed to increase the crawl rate by adding more machines
- Performance/efficiency: permit full use of available processing and network resources

What any crawler *should* do

- Fetch pages of "higher quality" first
- Continuous operation: Continue fetching fresh copies of a previously fetched page
- Extensible: Adapt to new data formats, protocols

Updated crawling picture



URL frontier

- Can include multiple pages from the same host
- Must avoid trying to fetch them all at the same time
- Must try to keep all crawling threads busy

Processing steps in crawling

- Pick a URL from the frontier ← Which one?
- Fetch the document at the URL
- Parse the URL
  - Extract links from it to other docs (URLs)
- Check if URL has content already seen
  - If not, add to indexes
- For each extracted URL
  - Ensure it passes certain URL filter tests
  - Check if it is already in the frontier (duplicate URL elimination)

E.g., only crawl .edu, obey robots.txt, etc.

Explicit and implicit politeness

- Explicit politeness: specifications from webmasters on what portions of site can be crawled
  - robots.txt
- Implicit politeness: even with no specification, avoid hitting any site too often

Robots.txt

- Protocol for giving spiders ("robots") limited access to a website, originally from 1994
  - www.robotstxt.org/wc/norobots.html
- Website announces its request on what can(not) be crawled
  - For a server, create a file `/robots.txt`
  - This file specifies access restrictions

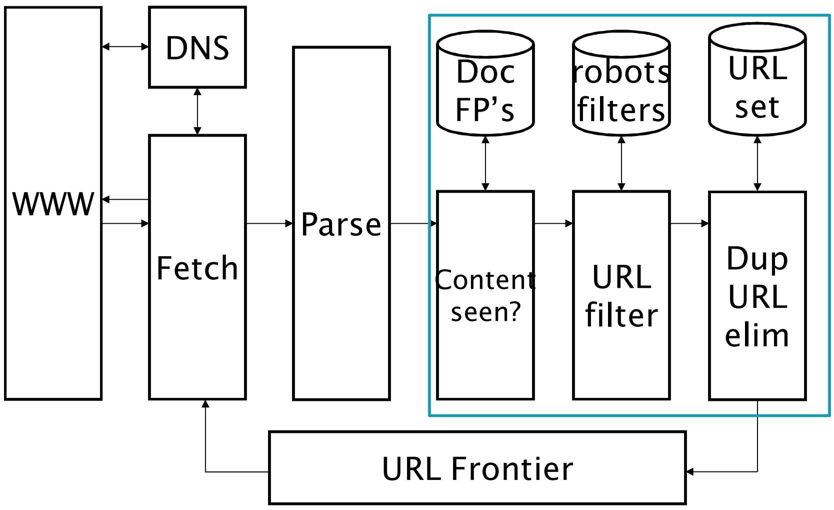
Robots.txt example

- No robot should visit any URL starting with "/yoursite/temp/", except the robot called "searchengine":

```
User-agent: *
Disallow: /yoursite/temp/

User-agent: searchengine
Disallow:
```

Βασική αρχιτεκτονική του σταχυολογητή



DNS (Domain Name Server)

- A lookup service on the internet
  - Given a URL, retrieve its IP address
  - Service provided by a distributed set of servers – thus, lookup latencies can be high (even seconds)
- Common OS implementations of DNS lookup are *blocking*: only one outstanding request at a time
- Solutions
  - DNS caching
  - Batch DNS resolver – collects requests and sends them out together

Parsing: URL normalization

- When a fetched document is parsed, some of the extracted links are *relative* URLs
- E.g., http://en.wikipedia.org/wiki/Main_Page has a relative link to /wiki/Wikipedia:General_disclaimer which is the same as the absolute URL http://en.wikipedia.org/wiki/Wikipedia:General_disclaimer
- During parsing, must normalize (expand) such relative URLs

Content seen?

- Duplication is widespread on the web
- If the page just fetched is already in the index, do not further process it
- This is verified using document fingerprints or shingles

Filters and robots.txt

- Filters – regular expressions for URL's to be crawled/not
- Once a robots.txt file is fetched from a site, need not fetch it repeatedly
  - Doing so burns bandwidth, hits web server
- Cache robots.txt files

Duplicate URL elimination

- For a non-continuous (one-shot) crawl, test to see if an extracted+filtered URL has already been passed to the frontier
- For a continuous crawl – see details of frontier implementation

Distributing the crawler

- Run multiple crawl threads, under different processes – potentially at different nodes
  - Geographically distributed nodes

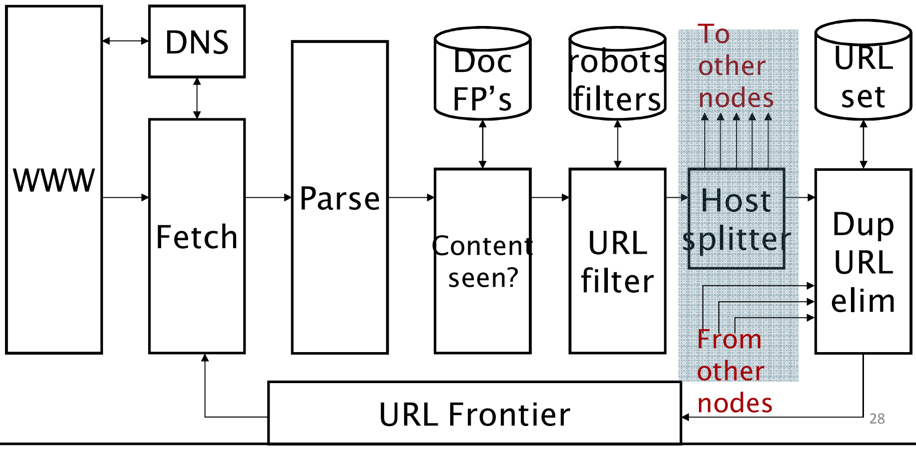
Distributing the crawler

Distributing the crawler

- Partition hosts being crawled into nodes
  - Hash used for partition
- How do these nodes communicate and share URLs?

Communication between nodes

- Output of the URL filter at each node is sent to the Dup URL Eliminator of the appropriate node



URL frontier: two main considerations

- Politeness: do not hit a web server too frequently
- Freshness: crawl some pages more often than others
  - E.g., pages (such as News sites) whose content changes often

These goals may conflict each other.

(E.g., simple priority queue fails – many links out of a page go to its own site, creating a burst of accesses to that site.)

Politeness – challenges

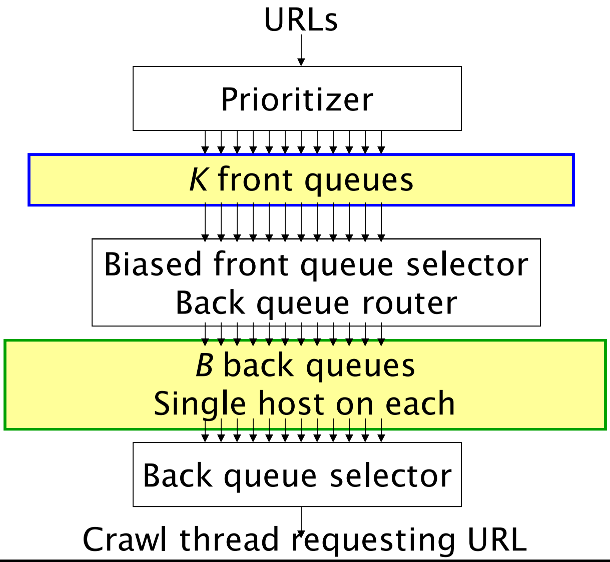
- Even if we restrict only one thread to fetch from a host, can hit it repeatedly
- Common heuristic: insert time gap between successive requests to a host that is >> time for most recent fetch from that host

Mercator URL frontier

Goals: ensure that

1. only *one connection* is open at a time *to any host;*
2. a *waiting time* of a few seconds occurs between successive requests
3. *high-priority pages* are crawled preferentially.

URL frontier: Mercator scheme

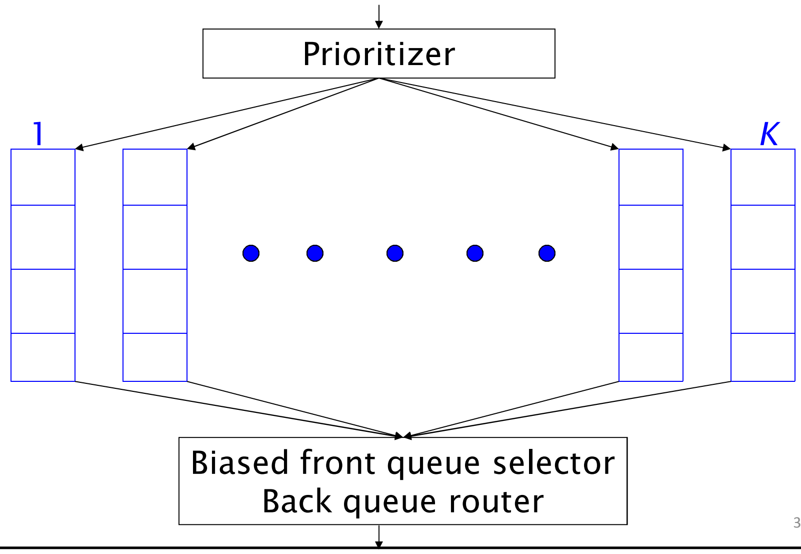


Mercator URL frontier

URLs flow in from the top into the frontier

- Front queues manage prioritization
- Back queues enforce politeness
- Each queue is FIFO

Front queues



Front queues

- Prioritizer assigns to URL an integer priority between 1 and *K*
  - Appends URL to corresponding queue
- Heuristics for assigning priority
  - Refresh rate sampled from previous crawls
  - Application-specific (e.g., “crawl news sites more often”)
  - Page-rank based

Biased front queue selector

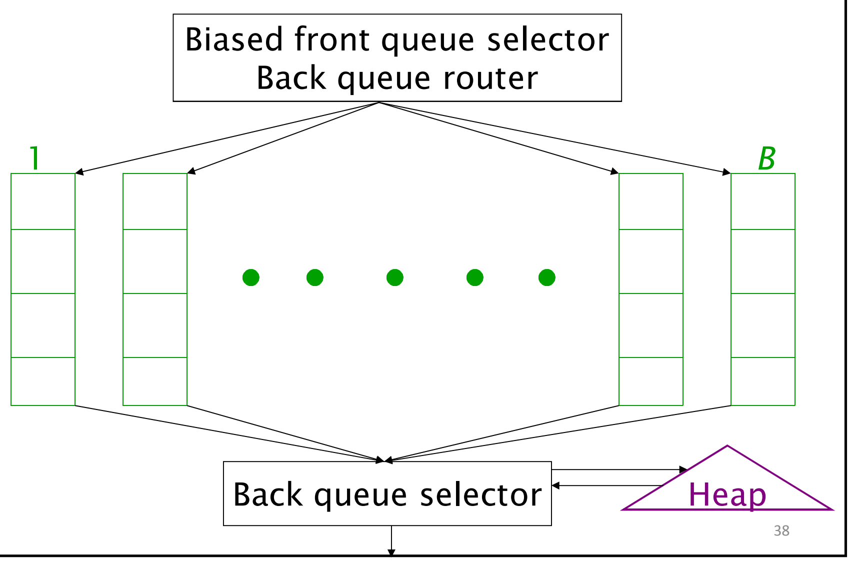
- When a back queue requests a URL (in a sequence to be described): picks a front queue from which to pull a URL
- This choice can be round-robin biased to queues of higher priority, or some more sophisticated variant
  - Can be randomized

Back queue invariants

- Each back queue is kept non-empty while the crawl is in progress
- Each back queue only contains URLs from a single host
  - Maintain a table from hosts to back queues

Host name	Back queue
…	3
	1
	B

Back queues



Back queue heap

- One entry for each back queue
- The entry is the earliest time t_e at which the host corresponding to the back queue can be hit again
- This earliest time is determined from
  - Last access to that host
  - Any time buffer heuristic we choose

URL processing

A crawler thread seeking a URL to crawl:

- Extracts the root of the heap
- If necessary waits until t_l
- Fetches URL at head of corresponding back queue *q* (look up from table)

URL processing

After fetching the URL

- Checks if (back)queue *q* is now empty – if so, pulls a URL *v* from front queues
  - If there's already a back queue for *v*'s host, append *v* to *q* and pull another URL from front queues, repeat
  - Else add *v* to *q*
- When *q* is non-empty, create heap entry for it

Number of back queues *B*

- Keep all threads busy while respecting politeness
- Mercator recommendation: three times as many back queues as crawler threads

DISTRIBUTED INDEXES

Κατανομή των Ευρετηρίων

How to distribute the term index across a large computer cluster that supports querying.

Two alternatives index implementations

- *partitioning by terms* or *global* index organization, and
- *partitioning by documents* or *local* index organization.

Κατανομή βάσει Όρων

- Index terms partitioned into subsets,
- Each subset resides at a node.
- Along with the terms at a node, we keep their postings

A query is routed to the nodes corresponding to its query terms.

In principle, this allows greater concurrency since a stream of queries with different query terms would hit different sets of machines.

Κατανομή βάσει Εγγράφων

- Documents partitioned into subsets
- Each subset resides in a node
- Each node contains the index for a subset of all documents.

A query is distributed to all nodes, with the results from various nodes being merged before presentation to the user.

Κατανομή βάσει Όρων: μειονεκτήματα

- In principle, index partition allows greater concurrency, since a stream of queries with different query terms would hit different sets of machines.
- In practice, partitioning indexes by vocabulary terms turns out to be non-trivial.

Κατανομή βάσει Όρων: μειονεκτήματα

- Multi-word queries require the *sending of long postings lists* between sets of nodes for merging, and the cost of this can outweigh the greater concurrency.
- *Load balancing* the partition is governed not by an a priori analysis of relative term frequencies, but rather by the distribution of query terms and their co-occurrences, which can drift with time or exhibit sudden bursts.
- More *difficult implementation*.

Κατανομή βάσει Εγγράφων

More common

- trades more local disk seeks for less inter-node communication.
- One difficulty: *global statistics* used in scoring - such as idf –
  - must be computed across the entire document collection even though the index at any single node only contains a subset of the documents.
  - Computed by distributed ``background'' processes that periodically refresh the node indexes with fresh global statistics.

Μέθοδος Κατανομής Εγγράφων

How to distributed documents to nodes?

❖ Hash of each URL to nodes

At query time,

the query is broadcast to each of the nodes, each node sends each top k results which are merged to find the top k documents for the query

Μέθοδος Κατανομής Εγγράφων

A common implementation heuristic:

Partition the document collection into

- indexes of documents that are more likely to score highly on most queries and
- low-scoring indexes with the remaining documents

Only search the low-scoring indexes when there are too few matches in the high-scoring indexes

CONNECTIVITY SERVERS

Connectivity Server

- Support for fast queries on the web graph
  - Which URLs point to a given URL?
  - Which URLs does a given URL point to?

Stores mappings in memory from

- URL to outlinks, URL to inlinks

- Applications
  - Crawl control
  - Web graph analysis
    - Connectivity, crawl optimization
  - Link analysis

Connectivity Server

- Assume that each web page is represented by a unique integer
- Maintain An *adjacency table*: a row for each web page, with the rows ordered by the corresponding integers.
- One for *pages link to and one for* pages linked to by
- Focus on the former

Boldi and Vigna 2004

- http://www2004.org/proceedings/docs/1p595.pdf
- Webgraph – set of algorithms and a java implementation
- Fundamental goal – maintain node adjacency lists in memory
  - For this, compressing the adjacency lists is the critical component

Adjacency lists

- The set of neighbors of a node
- Assume each URL represented by an integer
- E.g., for a 4 billion page web, need 32 bits per node
- Naively, this demands 64 bits to represent each hyperlink

Adjaceny list compression

- Properties exploited in compression:
  - Similarity (between lists)
    - Many rows have many entries in common. Thus, if we ***explicitly represent a prototype row for several similar rows***, the remainder can be succinctly expressed in terms of the prototypical row.
  - Locality (many links from a page go to “nearby” pages)
    - By encoding the destination of a link, we can often use small integers and thereby save space.
  - Use gap encodings in sorted lists
    - store the offset from the previous entry in the row

Storage

- Boldi/Vigna get down to an average of ~3 bits/link
  - (URL to URL edge)
- How?

Why is this remarkable?

Main ideas of Boldi/Vigna

- Consider lexicographically ordered list of all URLs, e.g.,
  - www.stanford.edu/alchemy
  - www.stanford.edu/biology
  - www.stanford.edu/biology/plant
  - www.stanford.edu/biology/plant/copyright
  - www.stanford.edu/biology/plant/people
  - www.stanford.edu/chemistry

Boldi/Vigna

- Each of these URLs has an adjacency list
- Main idea: due to templates, the adjacency list of a node is <u>similar</u> to one of the <u>7</u> preceding URLs in the lexicographic ordering
- Express adjacency list in terms of one of these
- E.g., consider these adjacency lists
  - 1, 2, 4, 8, 16, 32, 64
  - 1, 4, 9, 16, 25, 36, 49, 64
  - 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144
  - 1, 4, 8, 16, 25, 36, 49, 64

Why 7?

Encode as (-2), remove 9, add 8

ΤΕΛΟΣ α’ μέρους 11^{ου} Μαθήματος

Ερωτήσεις?

Χρησιμοποιήθηκε κάποιο υλικό των:
- ✓*Pandu Nayak and Prabhakar Raghavan, CS276:Information Retrieval and Web Search (Stanford)*
- ✓*Hinrich Schütze and Christina Lioma, Stuttgart IIR class*