Introduction to Information Retrieval
http://informationretrieval.org

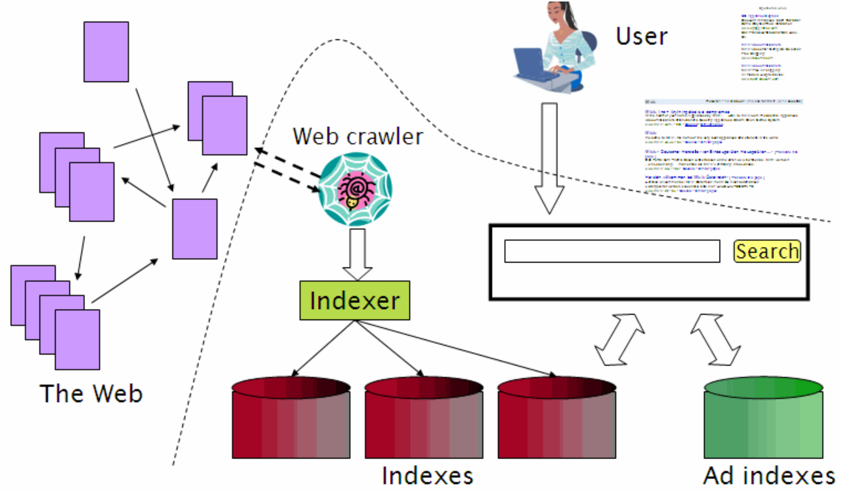
IIR 20: Crawling

Mihai Surdeanu
(Based on slides by Hinrich Schütze at informationretrieval.org)

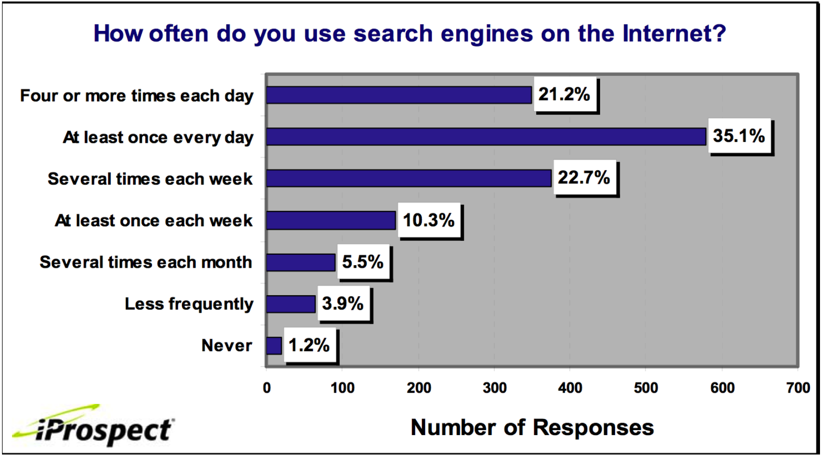
Spring 2017

Outline

Web search overview



Search is a top activity on the web



Without search engines, the web wouldn't work

- Without search, content is hard to find.
- → Without search, there is no incentive to create content.
  - Why publish something if nobody will read it?
  - Why publish something if I don't get ad revenue from it?
- Somebody needs to pay for the web.
  - Servers, web infrastructure, content creation
  - A large part today is paid by search ads.
  - Search pays for the web.

Outline

How hard can crawling be?

- Web search engines must crawl their documents.
- 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
- Ok: for web IR, getting the content of the documents takes longer . . .

How hard can crawling be?

- Web search engines must crawl their documents.
- 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
- Ok: for web IR, getting the content of the documents takes longer . . .
- . . . because of latency.
- But is that really a design/systems challenge?

Basic crawler operation

- Initialize queue with URLs of known seed pages
- Repeat
  - Take URL from queue
  - Fetch and parse page
  - Extract URLs from page
  - Add URLs to queue
- Fundamental assumption: The web is well linked.

Exercise: What's wrong with this crawler?

```
urlqueue := (some carefully selected set of seed urls)
while urlqueue is not empty:
myurl := urlqueue.getlastanddelete()
mypage := myurl.fetch()
fetchedurls.add(myurl)
newurls := mypage.extracturls()
for myurl in newurls:
if myurl not in fetchedurls and not in urlqueue:
urlqueue.add(myurl)
addtoinvertedindex(mypage)
```

What's wrong with the simple crawler

- Scale: we need to distribute.
- Duplicates: need to integrate duplicate detection (next section)
- Spam and spider traps: need to integrate spam detection
- Politeness: we need to be "nice" and space out all requests for a site over a longer period (hours, days)
- Freshness: we need to recrawl periodically.
  - Because of the size of the web, we can do frequent recrawls only for a small subset.
  - Again, subselection problem or prioritization

Magnitude of the crawling 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 a crawler must do

Be polite

- Don't hit a site too often
- Only crawl pages you are allowed to crawl: robots.txt

Be robust

- Be immune to spider traps, duplicates, very large pages, very large websites, dynamic pages etc

Robots.txt

- Protocol for giving crawlers ("robots") limited access to a website, originally from 1994
- Examples:
  - User-agent: *
    Disallow: /yoursite/temp/
  - User-agent: searchengine
    Disallow: /
- Important: cache the robots.txt file of each site we are crawling

Example of a robots.txt (nih.gov)

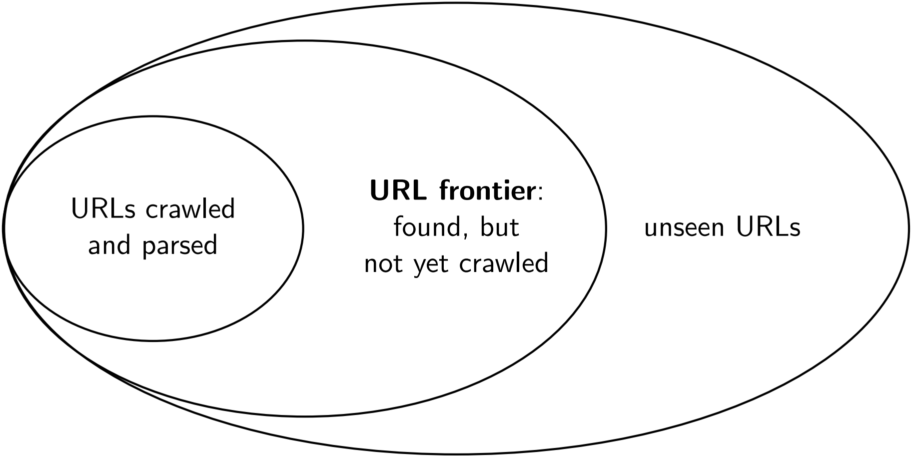
```
User-agent: PicoSearch/1.0
Disallow: /news/information/knight/
Disallow: /nidcd/
...
Disallow: /news/research_matters/secure/
Disallow: /od/ocpl/wag/
User-agent: *
Disallow: /news/information/knight/
Disallow: /nidcd/
...
Disallow: /news/research_matters/secure/
Disallow: /od/ocpl/wag/
Disallow: /ddir/
Disallow: /sdminutes/
```

What any crawler should do

- Be capable of distributed operation
- Be scalable: need to be able to increase crawl rate by adding more machines
- Fetch pages of higher quality first
- Continuous operation: get fresh version of already crawled pages

Outline

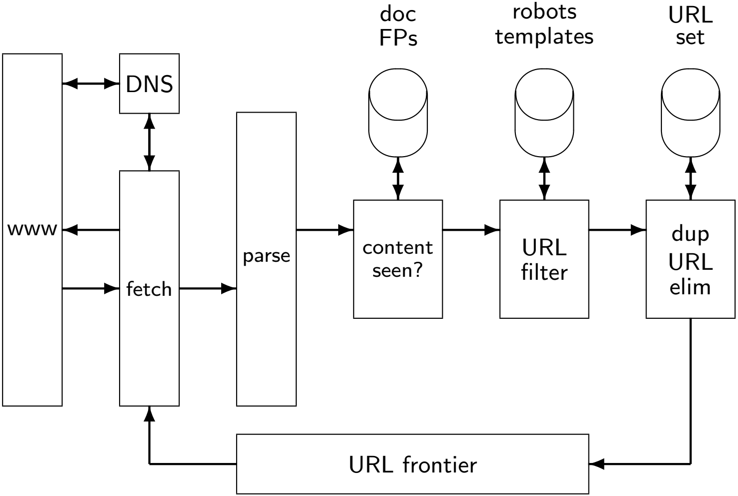
URL frontier



URL frontier

- The URL frontier is the data structure that holds and manages URLs we've seen, but that have not been crawled yet.
- Can include multiple pages from the same host
- Must avoid trying to fetch them all at the same time
- Must keep all crawling threads busy

Basic crawl architecture



URL normalization

- Some URLs extracted from a document are relative URLs.
- E.g., at http://mit.edu, we may have aboutsite.html
  - This is the same as: http://mit.edu/aboutsite.html
- During parsing, we must normalize (expand) all relative URLs.

Content seen

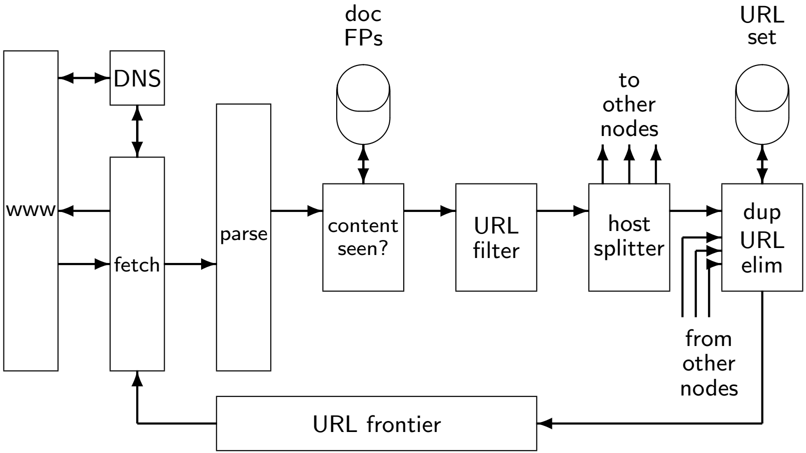
- For each page fetched: check if the content is already in the index
- Check this using document fingerprints or shingles (next section)
- Skip documents whose content has already been indexed

Distributing the crawler

- Run multiple crawl threads, potentially at different nodes
  - Usually geographically distributed nodes
- Partition hosts being crawled into nodes

Google data centers (wayfaring.com)

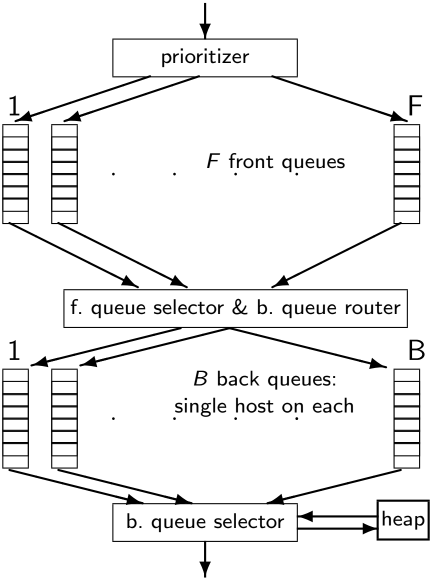
Distributed crawler



URL frontier: Two main considerations

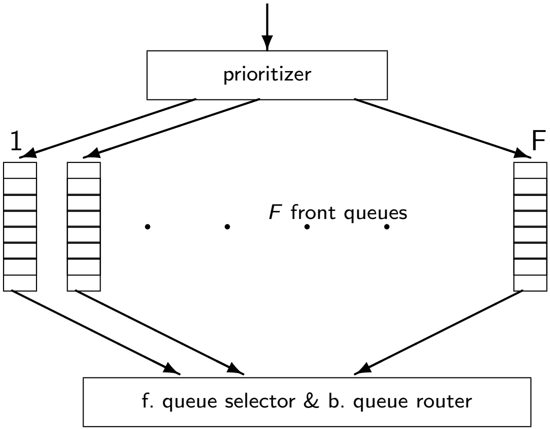
- Politeness: Don't hit a web server too frequently
  - E.g., insert a time gap between successive requests to the same server
- Freshness: Crawl some pages (e.g., news sites) more often than others
- Not an easy problem: simple priority queue fails.

Mercator URL frontier



- URLs flow in from the top into the frontier.
- Front queues manage prioritization.
- Back queues enforce politeness.
- Each queue is FIFO.

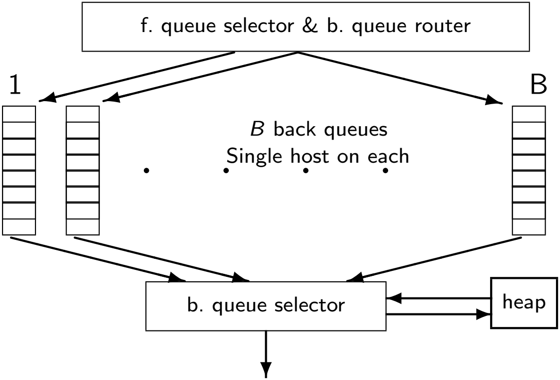
Mercator URL frontier: Front queues



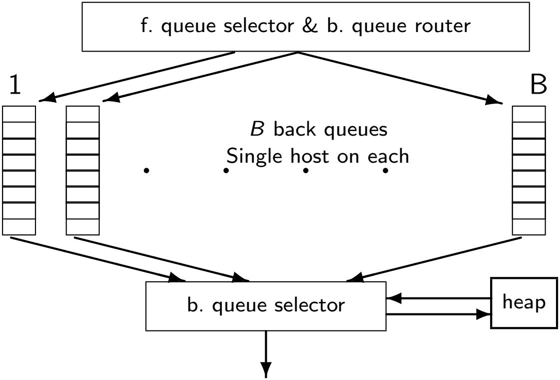
- Prioritizer assigns to URL an integer priority between 1 and F.
- Then appends URL to corresponding queue
- Heuristics for assigning priority: refresh rate, PageRank etc

- Selection from front queues is initiated by back queues
- Pick a front queue from which to select next URL: Round robin, randomly, or more sophisticated variant
- But with a bias in favor of high-priority front queues

Mercator URL frontier: Back queues

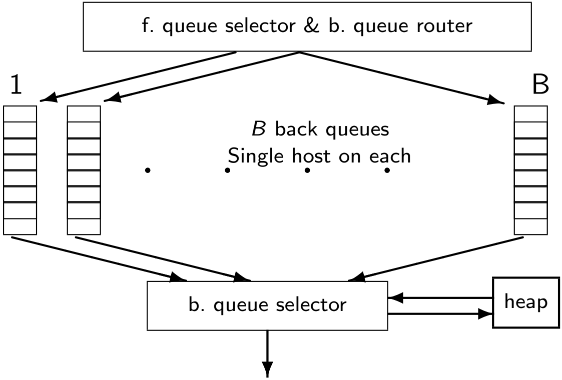


Mercator URL frontier: Back queues



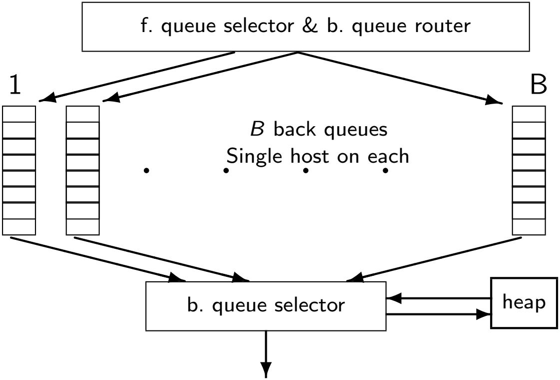
- Invariant 1. Each back queue is kept non-empty while the crawl is in progress.
- Invariant 2. Each back queue only contains URLs from a single host.
- Maintain a table from hosts to back queues.

Mercator URL frontier: Back queues



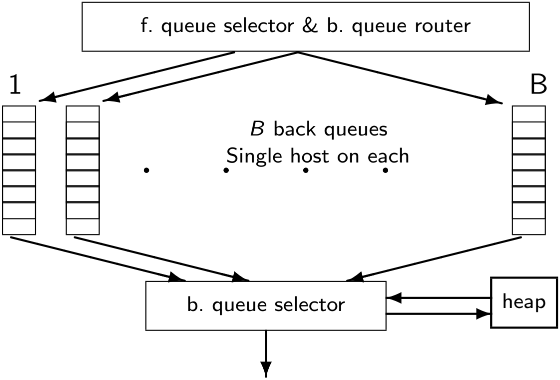
- In the heap:
- One entry for each back queue
- The entry is the earliest time t_e at which the host corresponding to the back queue can be hit again.
- The earliest time t_e is determined by (i) last access to that host (ii) time gap heuristic

Mercator URL frontier: Back queues



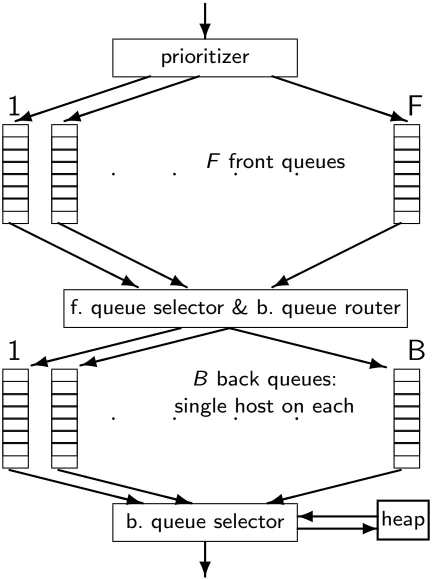
- How fetcher interacts with back queue:
- Repeat (i) extract current root q of the heap (q is a back queue)
- and (ii) fetch URL u at head of q

Mercator URL frontier: Back queues



- When we have emptied a back queue q:
- Repeat (i) pull URLs u from front queues and (ii) add u to its corresponding back queue ...
- ... until we get a u whose host does not have a back queue.
- Then put u in q and create heap entry for it.

Mercator URL frontier



- URLs flow in from the top into the frontier.
- Front queues manage prioritization.
- Back queues enforce politeness.
- Each queue is FIFO.

Spider trap

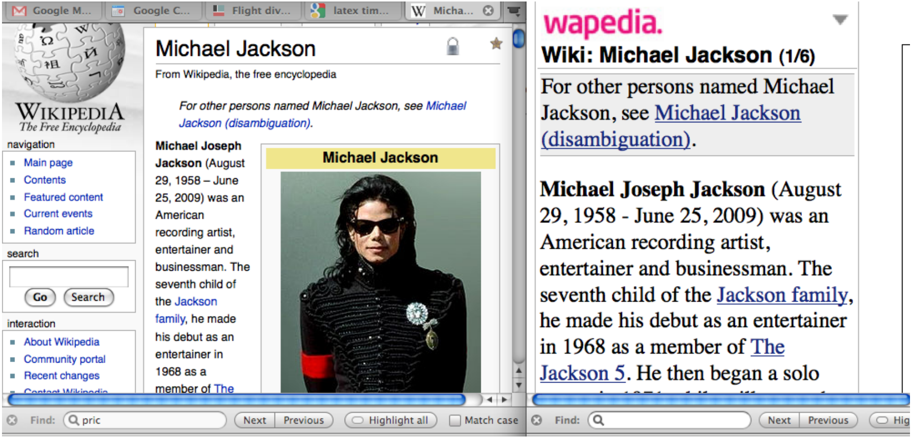
- Malicious server that generates an infinite sequence of linked pages, to force the crawler to index many pages from that site.
- Sophisticated spider traps generate pages that are not easily identified as dynamic.
- Why?

Outline

Duplicate detection

- The web is full of duplicated content.
- More so than many other collections
- Exact duplicates
  - Easy to eliminate
  - E.g., use hash/fingerprint
- Near-duplicates
  - Abundant on the web
  - Difficult to eliminate
- For the user, it's annoying to get a search result with near-identical documents.
- Marginal relevance is zero: even a highly relevant document becomes nonrelevant if it appears below a (near-)duplicate.
- We need to eliminate near-duplicates. □

Near-duplicates: Example



Exercise

How would you eliminate near-duplicates on the web?

Detecting near-duplicates

- Compute similarity with an edit-distance measure
- We want "syntactic" (as opposed to semantic) similarity.
  - True semantic similarity (similarity in content) is too difficult to compute.
- We do not consider documents near-duplicates if they have the same content, but express it with different words.
- Use similarity threshold θ to make the call "is/isn't a near-duplicate".
- E.g., two documents are near-duplicates if similarity $> \theta = 80\%$. □

Represent each document as set of **shingles**

- A shingle is simply a word n-gram.
- Shingles are used as features to measure syntactic similarity of documents.
- For example, for $n = 3$, "a rose is a rose is a rose" would be represented as this set of shingles:
  - { a-rose-is, rose-is-a, is-a-rose }
- We can map shingles to $1..2^m$ (e.g., $m = 64$) by fingerprinting.
- From now on: s_k refers to the shingle's fingerprint in $1..2^m$.
- We define the similarity of two documents as the Jaccard coefficient of their shingle sets. □

Recall: Jaccard coefficient

- A commonly used measure of overlap of two sets
- Let A and B be two sets
- Jaccard coefficient:

$$\text{JACCARD}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

  $(A \neq \emptyset$ or $B \neq \emptyset)$
- $\text{JACCARD}(A, A) = 1$
- $\text{JACCARD}(A, B) = 0$ if $A \cap B = 0$
- A and B don't have to be the same size.
- Always assigns a number between 0 and 1. □

Jaccard coefficient: Example

- Three documents:
  d_1: "Jack London traveled to Oakland"
  d_2: "Jack London traveled to the city of Oakland"
  d_3: "Jack traveled from Oakland to London"
- Based on shingles of size 2 (2-grams or bigrams), what are the Jaccard coefficients $J(d_1, d_2)$ and $J(d_1, d_3)$?
- $J(d_1, d_2) = 3/8 = 0.375$
- $J(d_1, d_3) = 0$
- Note: very sensitive to dissimilarity □

Represent each document as a **sketch**

- The number of shingles per document is large.
- To increase efficiency, we will use a sketch, a cleverly chosen subset of the shingles of a document.
- The size of a sketch is, say, $n = 200$
- (See the book for the sketch-building algorithm – not required for final)

Resources

- Nutch web crawler: `http://nutch.apache.org`