Introduction to

Information Retrieval

CS276
Information Retrieval and Web Search
Chris Manning and Pandu Nayak
Crawling and Duplicates

Today's lecture

- Web Crawling
- (Near) duplicate detection

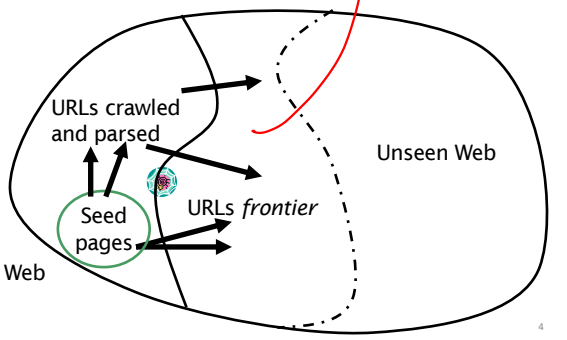
The initial URL frontier will have only seed pages i.e. the starting pages

Basic crawler operation

- 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

URLs waiting in the Q



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 → My website my rules
    - How "deep" should you crawl a site's URL hierarchy?
  - Site mirrors and duplicate pages
- Politeness – don't hit a server too often This is imp

Crawlers can be caught in inf. loops. Pages are generated on the fly

What any crawler *must* do

- Be Robust: Be immune to spider traps and other malicious behavior from web servers
- Be Polite: Respect implicit and explicit politeness considerations

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

File containing rules which must be followed by crawlers

Robots.txt

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

Many crawlers are present in a machine. And many machines are present in different geolocations.

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:
```


What any crawler *should* do

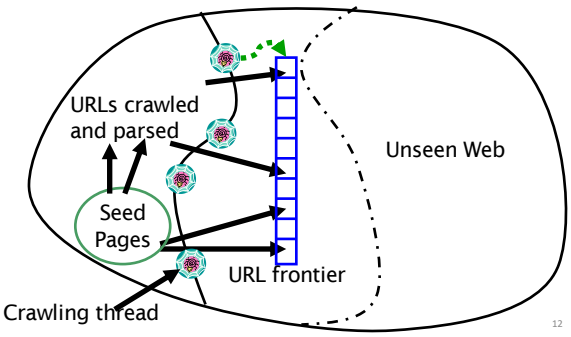
- Be capable of distributed operation: designed to run on multiple distributed machines
- Be scalable: designed to increase the crawl rate by adding more machines
- Performance/efficiency: permit full use of available processing and network resources

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

Like google should keep on crawling and indexing pages continuously

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

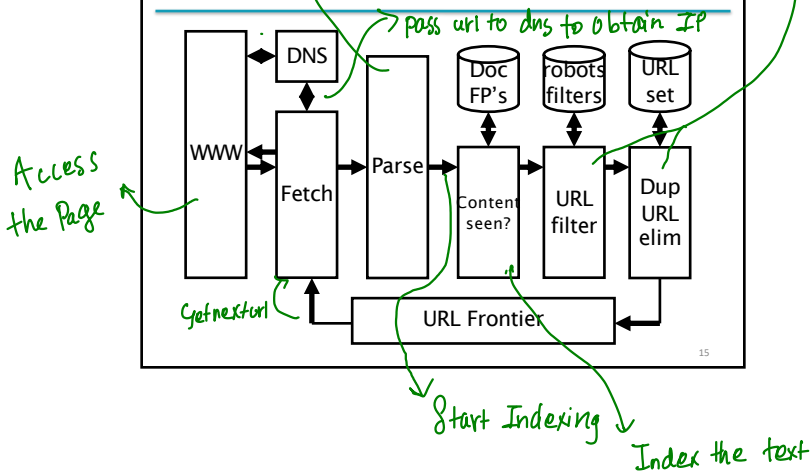
Processing steps in crawling

- 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.

Parse the documents. All the links in the page are processed so that only valid links are passed to the queue.

Basic crawl architecture



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

Every website (pages maybe different), has an IP address. For fetching a page that is needed. But getIp() is very slow. It takes seconds to get it. Hence, the DNS caching is done so that if other pages are crawled in same website, no need to look it up again (for near future only)

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

Sometimes different url on expansion point to the same website. So use this so that duplicates don't occur

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
  - Second part of this lecture

Filters and robots.txt

- Filters – regular expressions for URLs 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

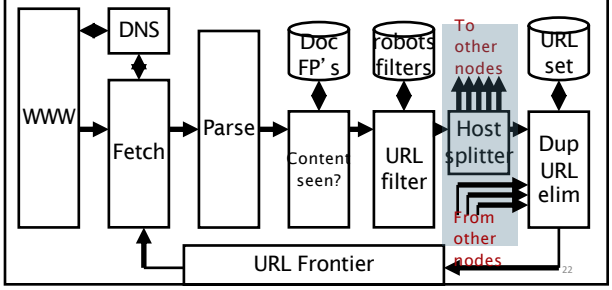
- Run multiple crawl threads, under different processes – potentially at different nodes (machine)
  - Geographically distributed nodes
- Partition hosts being crawled into nodes
  - Hash used for partition
- How do these nodes communicate and share URLs?

Each machine or node only process those Authority Links that it is assigned to. (e.g. FB by M1, gmail by M2 etc.)

After url filter, all the links that are generated must be sent to the url frontiers of the correct machine they are assigned to. This is done using a simple hash function to getMachineId(url)

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 Prioritization of crawling basically
  - E.g., pages (such as News sites) whose content changes often

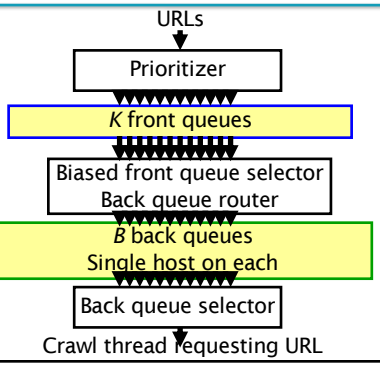
These goals may conflict with each other.

(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

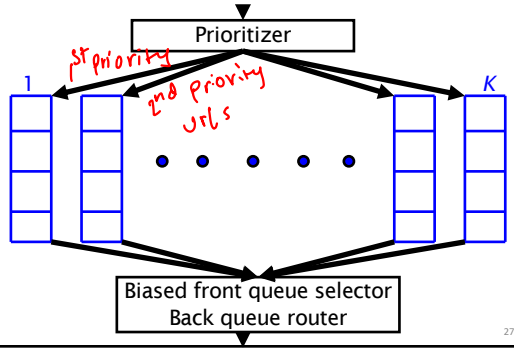
URL frontier: Mercator scheme



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 -Decides the Priority



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")

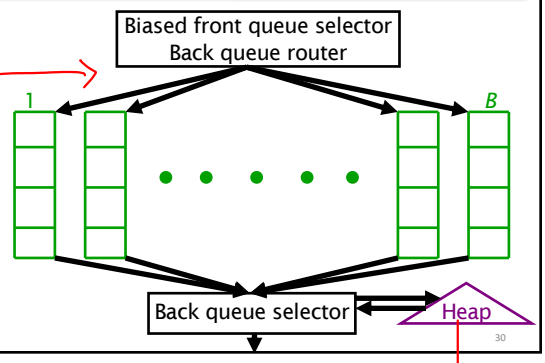
Fast updating websites should be given higher priority

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

Front Q can be randomly selected by assigning more weights to the queues with higher priority

Back queues



- A min heap that stores <time , qno. >
- The time = last accessed time + politeness time (te)
- generally politenes time = 10*latency time experienced from the website

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
…codeforces	3
cisco.com	1
github.com	*B*

Q no. for that authority

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

Back queue processing

- A crawler thread seeking a URL to crawl:
- Extracts the root of the heap
- Fetches URL at head of corresponding back queue *q* (look up from table)
- Checks if 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 it 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

If suppose Back Q 3 contained CF. After having processed all CF, the Q can now be used for some other web url as well.

Introduction to

Information Retrieval

Near duplicate document detection

Duplicate documents

- The web is full of duplicated content
- Strict duplicate detection = exact match
  - Not as common
- But many, many cases of near duplicates
  - E.g., Last modified date the only difference between two copies of a page

Distributed Indexing: Didn't understand

Each Machine provides posting lists for certain "User Defined" regex or terms.

1. Partitioning by Terms:

In practice, partitioning indexes by vocabulary terms turns out to be non-trivial. 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. Achieving good partitions is a function of the co-occurrences of query terms and entails the clustering of terms to optimize objectives that are not easy to quantify. Finally, this strategy makes implementation of dynamic indexing more difficult.

This is tough.

2. Partitioning by Docs:

Each Machine only provide index for a limited number of documents.

A more common implementation is to partition by documents: each node contains the index for a subset of all documents. Each query is distributed to all nodes, with the results from various nodes being merged before presentation to the user. This strategy trades more local disk seeks for less inter-node communication. One difficulty in this approach is that 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. These are computed by distributed "background" processes that periodically refresh the node indexes with fresh global statistics.