Crawling

Module Introduction

Motivating Problem

Internet crawling is discovering web content and downloading it to add to your index.

This is a technically complex, yet often overlooked aspect of search engines. “Breadth-first search from facebook.com” doesn’t begin to describe it.

http://xkcd.com/802/

Coverage

The first goal of an Internet crawler is to provide adequate coverage. *Coverage* is the fraction of available content you've crawled.

Challenges here include:

- Discovering new pages and web sites as they appear online.
- Duplicate site detection, so you don't waste time re-crawling content you already have.
- Avoiding *spider traps* – configurations of links that would cause a naive crawler to make an infinite series of requests.

Freshness

Coverage is often at odds with freshness. *Freshness* is the recency of the content in your index. If a page you've already crawled changes, you'd like to re-index it.

Freshness challenges include:

- Making sure your search engine provides good results for breaking news.
- Identifying the pages or sites which tend to be updated often.
- Balancing your limited crawling resources between new sites (coverage) and updated sites (freshness).

Politeness

Crawling the web consumes resources on the servers we're visiting. *Politeness* is a set of policies a well-behaved crawler should obey in order to be respectful of those resources.

- Requests to the same domain should be made with a reasonable delay.
- The total bandwidth consumed from a single site should be limited.
- Site owners' preferences, expressed by files such as robots.txt, should be respected.

And more…

Aside from these concerns, a good crawler should:

- Focus on crawling high-quality web sites.
- Be distributed and scalable, and make efficient use of server resources.
- Crawl web sites from a geographically-close data center (when possible).
- Be extensible, so it can handle different protocols and web content types appropriately.

Let’s get started!

HTTP Crawling

Crawling, session 2

Northeastern University
College of Computer and Information Science

Slides by: Jesse Anderton

A Basic Crawler

A crawler maintains a *frontier* – a collection of pages to be crawled – and iteratively selects and crawls pages from it.

- The frontier is initialized with a list of *seed pages*.
- The next page is selected carefully, for politeness and performance reasons.
- New URLs are processed and filtered before being added to the frontier.

We will cover these details in subsequent sessions.

```
def crawl(seeds):

    # The frontier is initially the seed set
    frontier.add_pages(seeds)

    # Iteratively crawl the next item in the frontier
    while not frontier.is_empty():

        # Crawl the next URL and extract anchor tags from it
        url = frontier.choose_next()
        page = crawl_url(url)
        urls = parse_page(page)

        # Update the frontier and send the page to the indexer
        frontier.add_pages(urls)
        send_to_indexer(page)
```

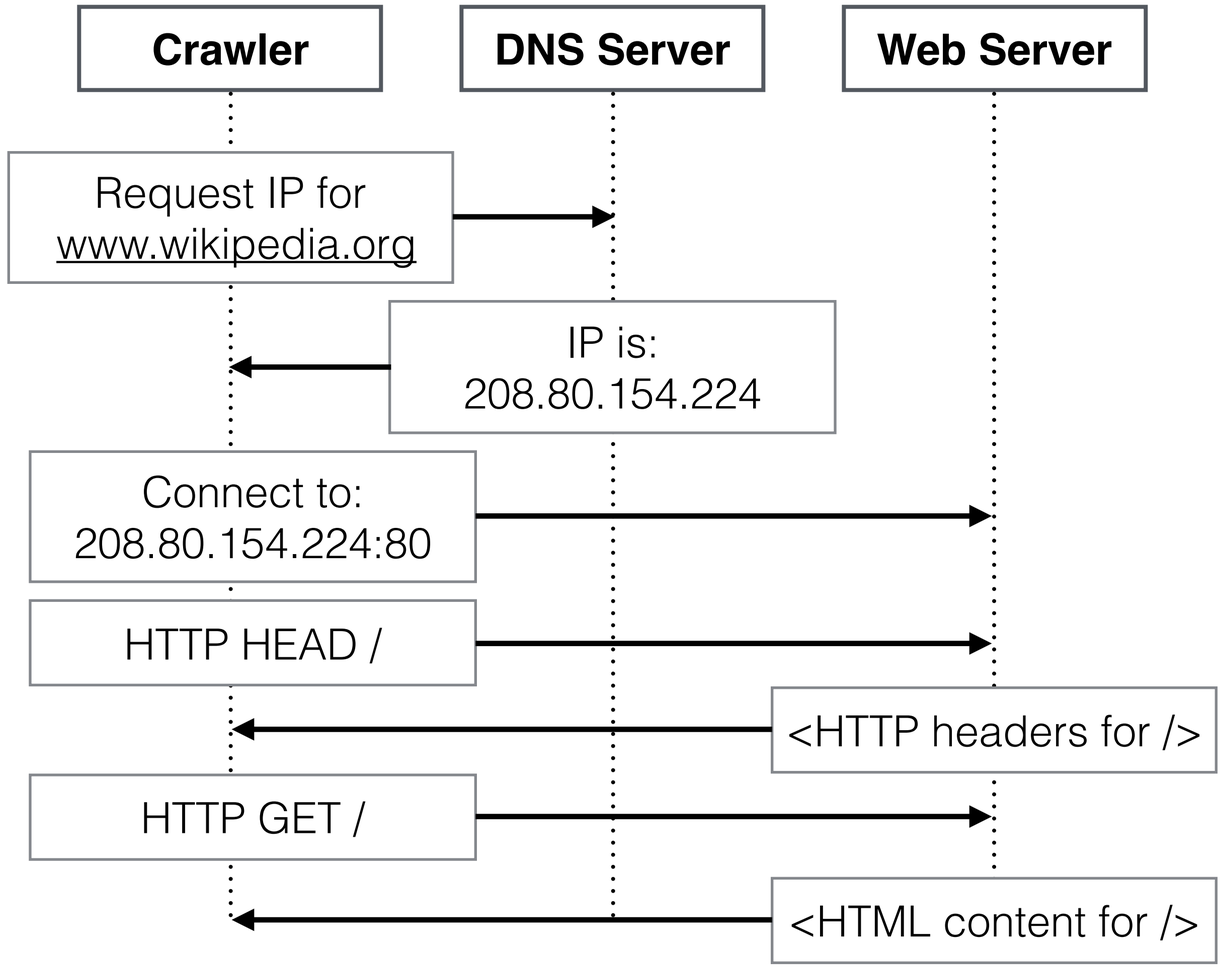
HTTP Fetching

Requesting and downloading a URL involves several steps.

1. A DNS server is asked to translate the domain into an IP address.

2. (optional) An HTTP HEAD request is made at the IP address to determine the page type, and whether the page contents have changed since the last crawl.

3. An HTTP GET request is made to retrieve the new page contents.



Request process for http://www.wikipedia.org/

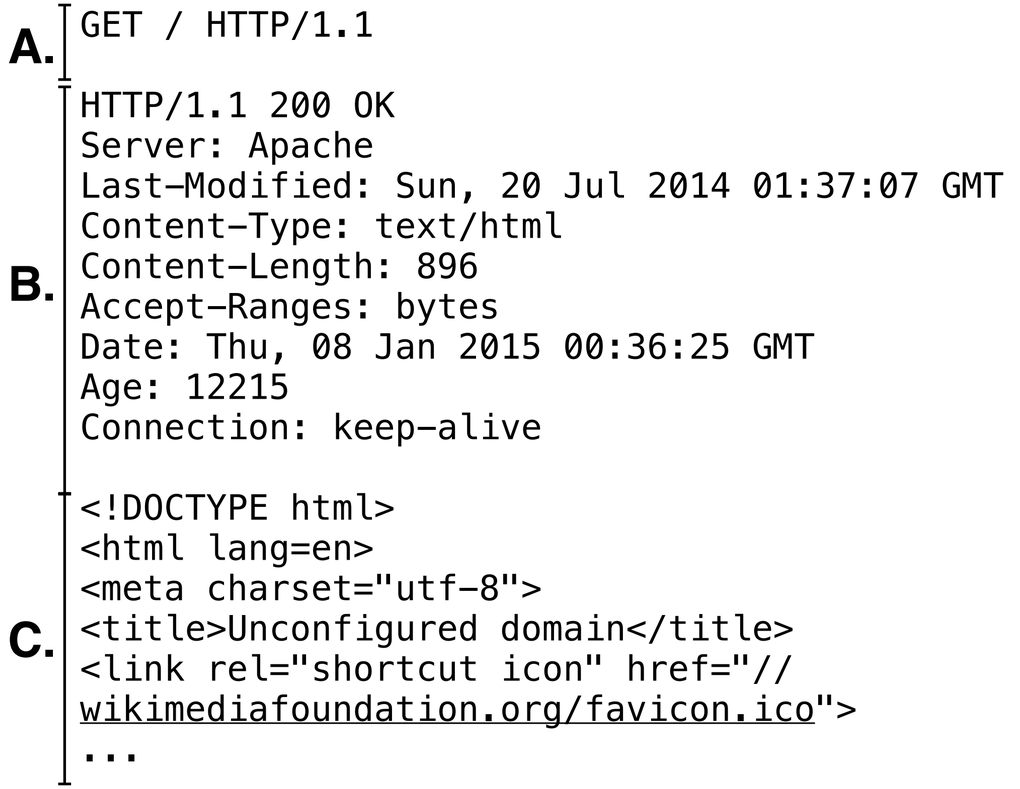
HTTP Requests

The HTTP request and response take the following form:

A. HTTP Request:
<method> <url> <HTTP version>
[<optional headers>]

B. Response Status and Headers:
<HTTP version> <code> <status>
[<headers>]

C. Response Body



HTTP Request and Response

URL Extraction

Downloaded files must be parsed according to their content type (usually available in the Content-Type header), and URLs extracted for adding to the frontier.

HTML documents in the wild often have formatting errors which the parser must address. Other document formats have their own issues. URLs may be embedded in PDFs, Word documents, etc.

Many URLs are missed, especially due to dynamic URL schemes and web pages generated by JavaScript and AJAX calls. This is part of the so-called “dark web.”

URL Canonicalization

Many possible URLs can refer to the same resource. It's important for the crawler (and index!) to use a canonical, or normalized, version of the URLs to avoid repeated requests.

Many rules have been used; some are guaranteed to only rewrite URLs to refer to the same resource, and others can make mistakes.

It can also be worthwhile to create specific normalization rules for important web domains, e.g. by encoding which URL parameters result in different web content.

http://example.com/some/../folder?id=1#anchor

http://example.com/some/../folder

http://www.example.com/folder

Conversion to Canonical URL

Rules for Canonicalization

Here are a few possible URL canonicalization rules.

Rule	Safe?	Example
Remove default port	Always	http://example.com:80 → http://example.com
Decoding octets for unreserved characters	Always	http://example.com/%7Ehome → http://example.com/~home
Remove . and ..	Usually	http://example.com/a/./b/../c → http://example.com/a/c
Force trailing slash for directories	Usually	http://example.com/a/b → http://example.com/a/b/
Remove default index pages	Sometimes	http://example.com/index.html → http://example.com
Removing the fragment	Sometimes	http://example.com/a#b/c → http://example.com/a

Wrapping Up

Web crawling requires attending to many details. DNS responses should be cached, HTTP HEAD requests should generally be sent before GET requests, and so on.

Extracting and normalizing URLs is important, because it dramatically affects your coverage and the time wasted on crawling, indexing, and ultimately retrieving duplicate content.

Next, we'll see how to detect duplicate pages hosted from different URLs.

Duplicate Detection

Crawling, session 3

Northeastern University
College of Computer and Information Science

Slides by: Jesse Anderton

Page De-duplication

There are many duplicate or near-duplicate pages on the Internet: in a large crawl, roughly 30% of pages duplicate content found in the other 70%.

This happens for many reasons: copies, mirrors, versioning, plagiarism, spam, etc.

Exact duplicate detection is straightforward, relying on *checksums* for rapid detection.

A simple checksum: sum of bytes

T	r	o	p	i	c	a	l		f	i	s	h	*Sum*
54	72	6F	70	69	63	61	6C	20	66	69	73	68	508

Selected Checksum Types

Type	Example	Goal
Checksum	MD5	Duplicate detection
Error-correcting Checksum	Cyclic redundancy checks	Data verification and correction
Cryptographic Checksum	SHA-512	Non-reversible data identifiers
Hash Function	Jenkins hash functions	Hash table keys

Detecting Near-Duplicates

Detecting near-duplicates is much harder, especially with performance constraints.

A pairwise content comparison requires $O(n)$ comparisons to find matches for a single document, or $O(n^2)$ to find matches for all pairs.

We will explore two approaches here:

- Fingerprint methods generate a smaller document description which can be used for faster approximate comparison based on ngram overlap.
- Similarity hashing efficiently calculates approximate cosine similarity between documents.

Fingerprint Calculation

Fingerprints based on n-grams can be calculated and used as follows.

1. A document is converted into a sequence of words; all punctuation and formatting is removed.

2. Words are grouped into (possibly overlapping) n-grams, for some n.

3. A subset of the n-grams are selected to represent the document.

4. The selected n-grams are hashed, and the resulting hashes stored in an inverted index.

5. Documents are compared based on the number of overlapping n-grams.

Fingerprint Example

1. Original text

Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

2. Overlapping trigrams

tropical fish include, fish include fish, include fish found, fish found in, found in tropical, in tropical environments, tropical environments around, environments around the, around the world, the world including, world including both, including both freshwater, both freshwater and, freshwater and salt, and salt water, salt water species

3. Hashed trigrams

938 664 463 822 492 798 78 969 143 236 913 908 694 553 870 779

4. Selected trigrams (using *0 mod 4*)

664 492 236 908

Similarity Hashing

Word-based comparisons are more effective at finding near-duplicates than comparing n-grams, but efficiency is a problem.

The simhash algorithm performs word-based comparisons with a hashing approach similar to n-gram fingerprints. It can be run over any weighted document features.

The cosine similarity between two documents is proportional to the number of bits in common between their simhash fingerprints.

```
def simhash(doc, hash_size):

    # Create a zero vector of length `hash_size`
    sums = [0.0] * hash_size

    # Calculate the sums vector
    for word, weight in doc:

        # Create a unique hash for the word
        word_hash_bits = calculate_hash(word, hash_size)

        # Add/subtract the weight to sums based on its hash
        for i = 0 to (hash_size-1):
            if word_hash_bits[i] == 1:
                sums[i] += weight
            else:
                sums[i] -= weight

    # Calculate the final hash
    doc_hash_bits = [0] * hash_size
    for i = 0 to (hash_size-1):
        if sums[i] > 0:
            doc_hash_bits[i] = 1
        else:
            doc_hash_bits[i] = 0
    return doc_hash_bits
```

Similarity Hashing Example

1. Original text

Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

2. Weighted features (word TF scores)

tropical: 2, fish: 2, include: 1, found: 1, environments: 1, around: 1, world: 1, including: 1, both: 1, freshwater: 1, salt: 1, water: 1, species: 1

3. 8-bit Word Hashes

tropical: 01100001, fish: 10101011, include: 11100110, found: 00011110,
environments: 00101101, around: 10001011, world: 00101010, including: 11000000,
both: 10101110, freshwater: 00111111, salt: 10110101, water: 00100101, species: 11101110

4. Summed feature weights

sums = [1, -5, 9, -9, 3, 1, 3, 3]

5. Document Hash

10101111

Wrapping Up

Detecting near-duplicate page content is important for conserving limited crawling and indexing resources, and for making sure that search result lists contain meaningfully-distinct results.

Most approaches are based on hashing algorithms for fast but approximate similarity comparisons.

These techniques can also be readily adapted to detect plagiarism and copyright violations in plain text and in source code. In the latter case, document features often include syntactic parse trees rather than literal words.

Next, we'll learn about the rules of politeness that a crawler should follow.

Politeness

Crawling, session 4

The need for politeness

Web crawlers are generally distributed and multithreaded, and are capable of taxing the capabilities of most web servers. This is particularly true of the crawlers for major businesses, such as Bing and Google.

In addition, some content should not be crawled at all.

- Some web content is considered private or under copyright, and its owners prefer that it not be crawled and indexed.

- Other URLs implement API calls and don't lead to indexable content.

This can easily create conflict between search providers and the web sites they're linking to. Politeness policies have been created as a way to mediate these issues.

Robots Exclusion Protocol

Web site administrators can express their crawling preferences by hosting a page at /robots.txt. Every crawler should honor these preferences.

These files indicate which files are permitted and disallowed for particular crawlers, identified by their user agents. They can also specify a preferred site mirror to crawl.

More recently, sites have also begun requesting crawl interval delays in robots.txt.

```
#
# robots.txt for http://www.wikipedia.org/ and friends
#
# Please note: There are a lot of pages on this site, and there are
# some misbehaved spiders out there that go _way_ too fast. If you're
# irresponsible, your access to the site may be blocked.
#

# advertising-related bots:
User-agent: Mediapartners-Google*
Disallow: /

[...]

#
# Friendly, low-speed bots are welcome viewing article pages, but not
# dynamically-generated pages please.
#
# There is a special exception for API mobileview to allow dynamic
# mobile web & app views to load section content.
# These views aren't HTTP-cached but use parser cache aggressively
# and don't expose special: pages etc.
#
User-agent: *
Allow: /w/api.php?action=mobileview&
Disallow: /w/
Disallow: /trap/
Disallow: /wiki/Special:Collection
Disallow: /wiki/Special:Random
Disallow: /wiki/Special:Search

[...]
```

Portion of http://www.wikipedia.org/robots.txt

Request Intervals

It is very important to limit the rate of requests your crawler makes to the same domain. Too-frequent requests are the main way a crawler can harm a web site.

Typical crawler delays are in the range of 10-15 seconds per request. You should never crawl more than one page per second from the same domain. For large sites, it can take days or weeks to crawl the entire domain – this is preferable to overloading their site (and possibly getting your IP address blocked).

If the site's robots.txt file has a Crawl-delay directive, it should be honored.

In a distributed crawler, all requests for the same domain are typically sent to the same crawler instance to easily throttle the rate of requests.

Sitemaps

In addition to robots.txt, which asks crawlers *not* to index certain content, a site can request that certain content be indexed by hosting a file at /sitemap.xml.

Sitemaps can direct your crawler toward the most important content on the site, indicate when it has changed, etc.

```
<?xml version="1.0" encoding="utf-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.sitemaps.org/schemas/
sitemap/0.9 http://www.sitemaps.org/schemas/sitemap/0.9/
sitemap.xsd">
   <url>
      <loc>http://example.com/</loc>
      <lastmod>2006-11-18</lastmod>
      <changefreq>daily</changefreq>
      <priority>0.8</priority>
   </url>
</urlset>
```

Example courtesy Wikipedia

Wrapping Up

It’s important to remember that we are providing web site owners with a service, and to be mindful of the resources we consume by crawling their sites.

Following robots.txt and using sitemaps.xml can also help the crawler avoid pitfalls particular to the web site.

Next, we’ll see strategies for improving the coverage of a web crawl.

Coverage

Crawling, session 5

Northeastern University
College of Computer and Information Science

Slides by: Jesse Anderton

Coverage Goals

The Internet is too large and changes too rapidly for any crawler to be able to crawl and index it all. Instead, a crawler should focus on strategic crawling to balance coverage and freshness.

A crawler should prioritize crawling high-quality content to better answer user queries. The Internet contains a lot of spam, redundant information, and pages which aren't likely to be relevant to users' information needs.

```
def crawl(seeds):

    # The frontier is initially the seed set
    frontier.add_pages(seeds)

    # Iteratively crawl the next item in the frontier
    while not frontier.is_empty():

        # Crawl the next URL and extract anchor tags from it
        url = frontier.choose_next()
        page = crawl_url(url)
        urls = parse_page(page)

        # Update the frontier and send the page to the indexer
        frontier.add_pages(urls)
        send_to_indexer(page)
```

Basic Crawler Algorithm

Selection Policies

A **selection policy** is an algorithm used to select the next page to crawl. Standard approaches include:

- **Breadth-first search**: This distributes requests across domains relatively well and tends to download high-PageRank pages early.
- **Backlink count**: Prioritize pages with more in-links from already-crawled pages.
- **Larger sites first**: Prioritize pages on domains with many pages in the frontier.
- **Partial PageRank**: Approximate PageRank scores are calculated based on already-crawled pages.

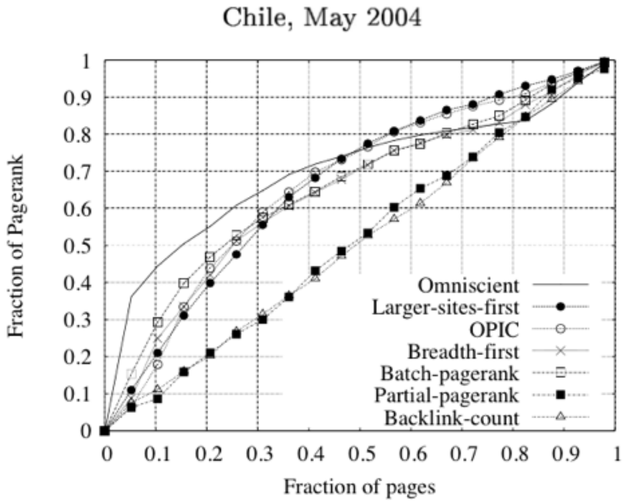
There are also approaches which estimate page quality based on a prior crawl.

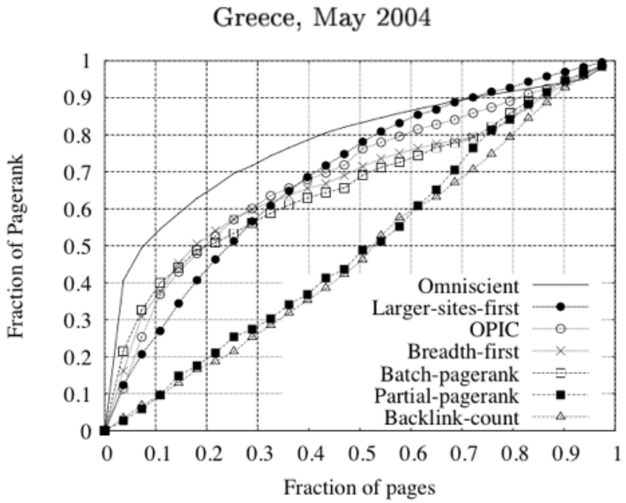
Comparing Approaches

Baeza-Yates et al compare these approaches to find out which fraction of high quality pages in a collection is crawled by each strategy at various points in a crawl.

Breadth-first search does relatively poorly. Larger sites first is among the best approaches, along with “historical” approaches which take PageRank scores from a prior crawl into account.

OPIC, a fast approximation to PageRank which can be calculated on the fly, is another good choice. The “omniscient” baseline always fetches the highest PR page in the frontier.





Ricardo Baeza-Yates, Carlos Castillo, Mauricio Marin, and Andrea Rodriguez. 2005. Crawling a country: better strategies than breadth-first for web page ordering.

Obtaining Seed URLs

It's important to choose the right sites to initialize your frontier. A simple baseline approach is to start with the sites in an Internet directory, such as http://www.dmoz.org.

In general, good hubs tend to lead to many high-quality web pages. These hubs can be identified with a careful analysis of a prior crawl.

http://www.dmoz.org

The Deep Web

Despite these techniques, a substantial fraction of web pages remains uncrawled and unindexed by search engines. These pages are known as “the deep web.”

These pages are missed for many reasons.

- Dynamically-generated pages, such as pages that make heavy use of AJAX, rely on web browser behavior and are missed by a straightforward crawl.
- Many pages reside on private web sites and are protected by passwords.
- Some pages are intentionally hidden, using robots.txt or more sophisticated approaches such as “darknet” software.

Special crawling and indexing techniques are used to attempt to index this content, such as rendering pages in a browser during the crawl.

Wrapping Up

Good coverage is obtained by carefully selecting seed URLs and using a good page selection policy to decide what to crawl next.

Breadth-first search is adequate when you have simple needs, but many techniques outperform it. It particularly helps to have an existing index from a previous crawl.

Next, we'll see how to adjust page selection to favor document freshness.

Freshness

Crawling, session 6

Page Freshness

The web is constantly changing as content is added, deleted, and modified. In order for a crawler to reflect the web as users will encounter it, it needs to recrawl content soon after it changes.

This need for freshness is key to providing a good search engine experience. For instance, when breaking news develops, users will rely on your search engine to stay updated.

It’s also important to refresh less time-sensitive documents so the results list doesn’t contain spurious links to deleted or modified data.

HTTP HEAD Requests

A crawler can determine whether a page has changed by making an HTTP HEAD request.

The response provides the HTTP status code and headers, but not the document body. The headers include information about when the content was last updated.

However, it's not feasible to constantly send HEAD requests, so this isn't an adequate strategy for freshness.

Request

```
HEAD /csinfo/people.html HTTP/1.1
Host: www.cs.umass.edu
```

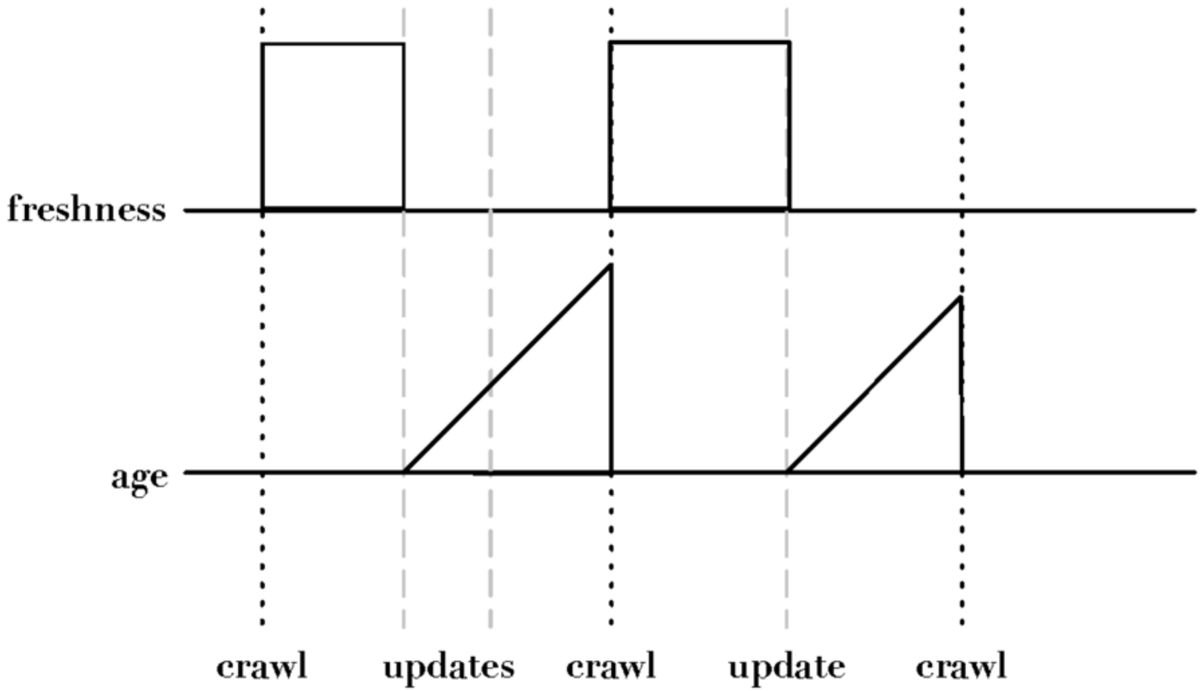
Response

```
HTTP/1.1 200 OK
Date: Thu, 03 Apr 2008 05:17:54 GMT
Server: Apache/2.0.52 (CentOS)
Last-Modified: Fri, 04 Jan 2008 15:28:39 GMT
ETag: "239c33-2576-2a2837c0"
Accept-Ranges: bytes
Content-Length: 9590
Connection: close
Content-Type: text/html; charset=ISO-8859-1
```

Freshness vs. Age

It turns out that optimizing to minimize freshness is a poor strategy: it can lead the crawler to ignore important sites.

Instead, it's better to re-crawl pages when the age of the last crawled version exceeds some limit. The *age* of a page is the elapsed time since the first update after the most recent crawl.



Freshness is binary, age is continuous.

Expected Page Age

The expected age of a page t days after it was crawled depends on its update probability:

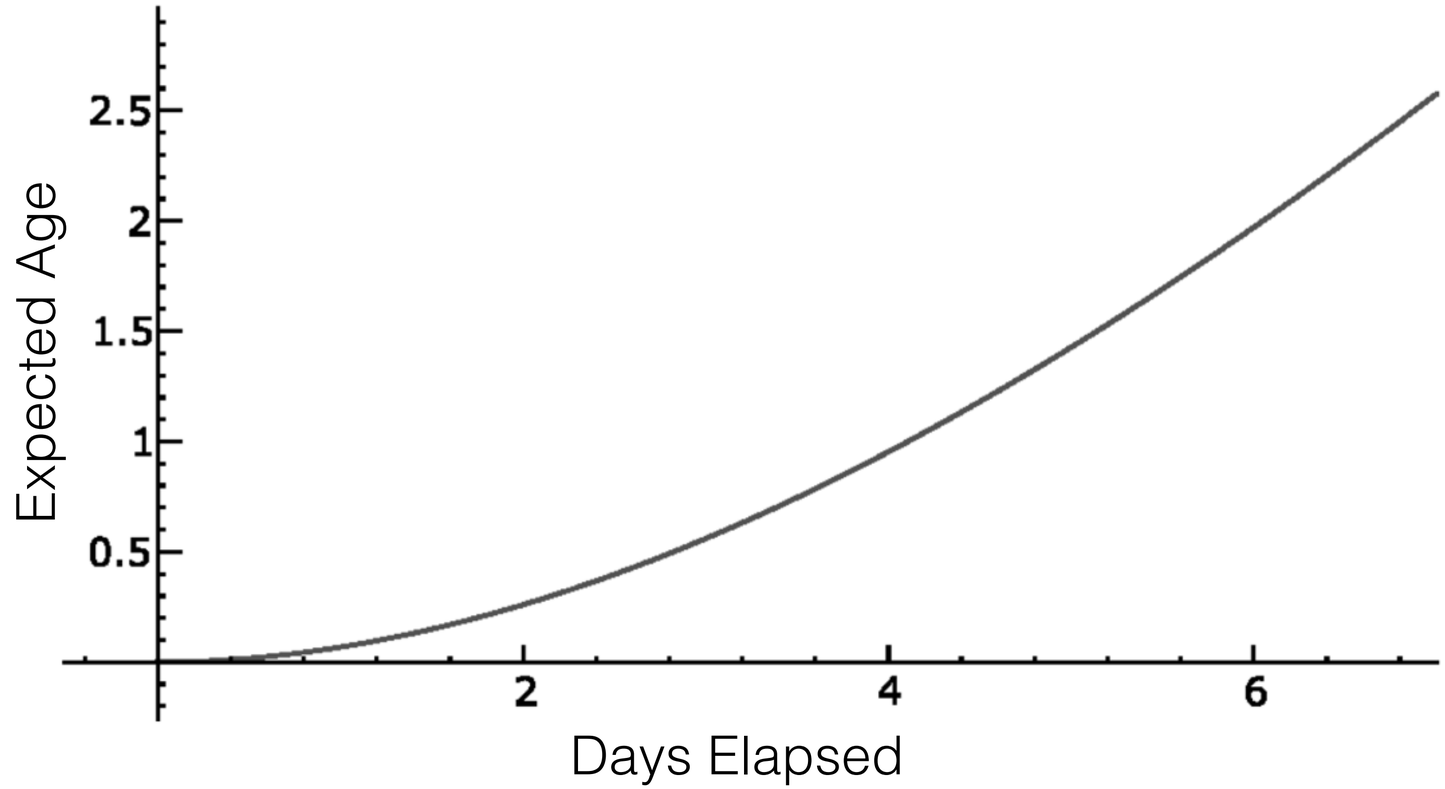
$$age(\lambda, t) = \int_0^t P(\textbf{page changed at time } x)(t - x)dx$$

On average, page updates follow a Poisson distribution – the time until the next update is governed by an exponential distribution. This makes the expected age:

$$age(\lambda, t) = \int_0^t \lambda e^{-\lambda x}(t - x)dx$$

Cost of Not Re-crawling

The cost of not re-crawling a page grows exponentially in the time since the last crawl. For instance, with page update frequency $\lambda = 1/7$ days:



Freshness vs. Coverage

The opposing needs of Freshness and Coverage need to be balanced in the scoring function used to select the next page to crawl.

Finding an optimal balance is still an open question. Fairly recent studies have shown that even large name-brand search engines only do a modest job at finding the most recent content.

However, a reasonable approach is to include a term in the page priority function for the expected age of the page content. For important domains, you can track the site-wide update frequency λ.

Wrapping Up

The web is constantly changing, and re-crawling the latest changes quickly can be challenging.

It turns out that aggressively re-crawling as soon as a page changes is sometimes the wrong approach: it's better to use a cost function associated with the expected age of the content, and tolerate a small delay between re-crawls.

Next, we'll take a look at what can go wrong with crawling.

Pitfalls of Crawling

Crawling, session 7

Northeastern University
College of Computer and Information Science

Slides by: Jesse Anderton

Crawling at Scale

A commercial crawler should support thousands of HTTP requests per second. If the crawler is distributed, that applies for each node. Achieving this requires careful engineering of each component.

- DNS resolution can quickly become a bottleneck, particularly because sites often have URLs with many subdomains at a single IP address.

- The frontier can grow extremely rapidly – hundreds of thousands of URLs per second are not uncommon. Managing the filtering and prioritization of URLs is a challenge.

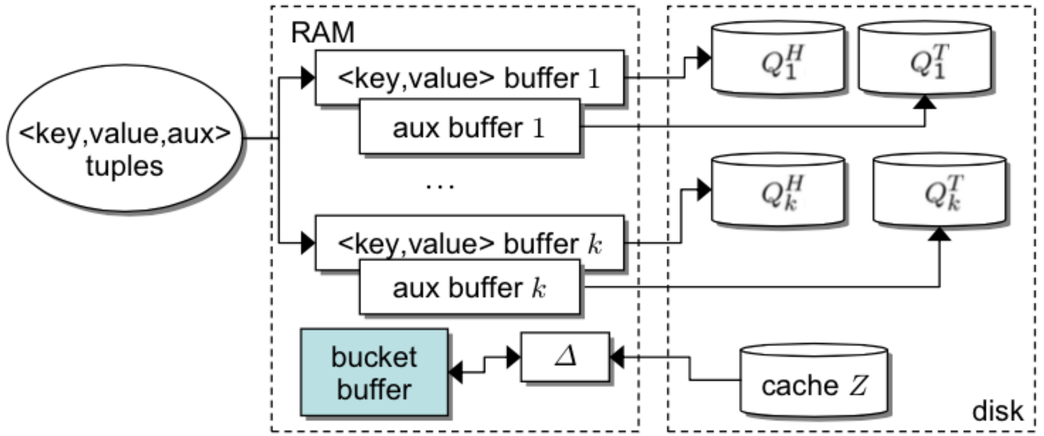
- Spam and malicious web sites must be addressed, lest they overwhelm the frontier and waste your crawling resources. For instance, some sites respond to crawlers by intentionally adding seconds of latency to each HTTP response. Other sites respond with data crafted to confuse, crash, or mislead a crawler.

Duplicate URL Detection at Scale

Lee et al's DRUM algorithm gives a sense of the requirements of large scale de-duplication.

It manages a collection of tuples of keys (hashed URLs), values (arbitrary data, such as quality scores), and aux data (URLs). It supports the following operations:

- `check` – Does a key exist? If so, fetch its value.
- `update` – Merge new tuples into the repository.
- `check+update` – Check and update in a single pass.

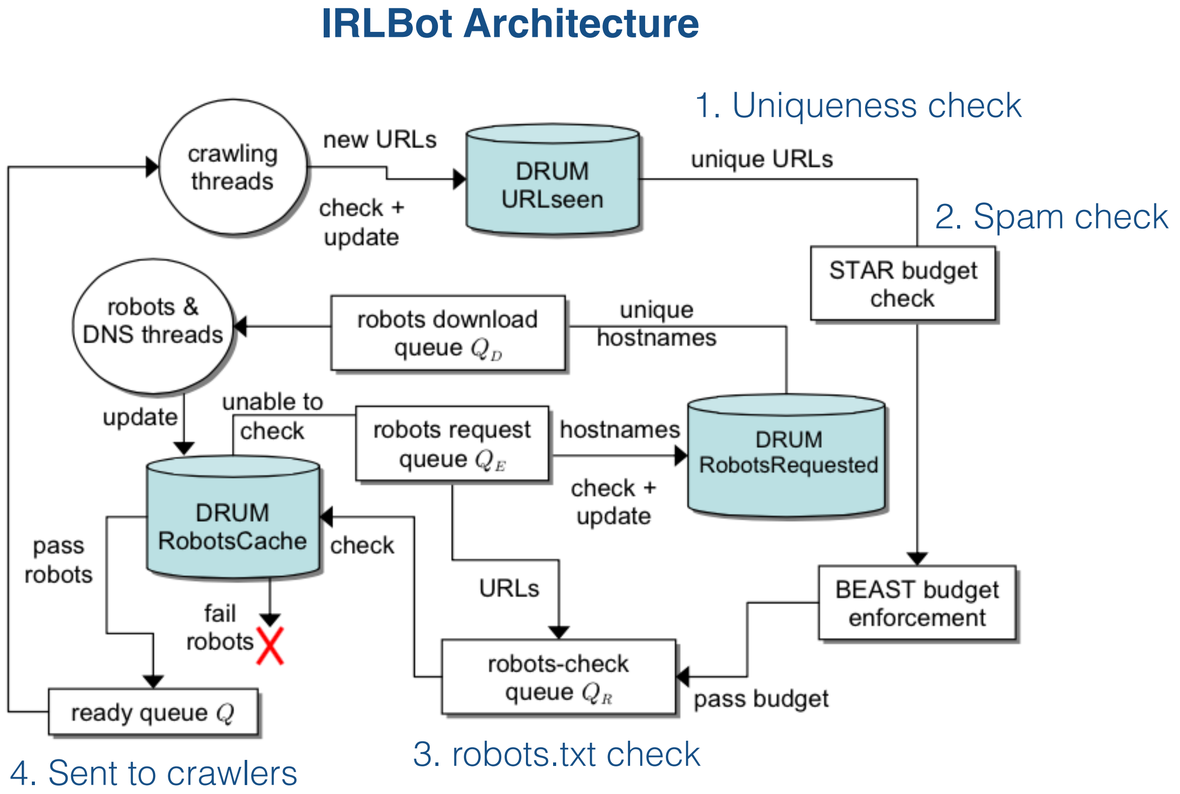


Data flow for DRUM: A tiered system of buffers in RAM and on disk is used to support large-scale operations.

IRLBot Operation

DRUM is used a storage for the IRLBot crawler. A new URL passes through the following steps.

1. The URLSeen DRUM checks whether the URL has already been fetched.
2. If not, two budget checks filter out spam links (discussed next).
3. Next, we check whether the URL passes its robots.txt. If necessary, we fetch robots.txt from the server.
4. Finally, the URL is passed to the queue to be crawled by the next available thread.



Link Spam

The web is full of link farms and other forms of link spam, generally posted by people trying to manipulate page quality measures such as PageRank.

These links waste a crawler's resources, and detecting and avoiding them is important for correct page quality calculations.

One way to mitigate this, implemented in IRLBot, is based on the observation that spam servers tend to have very large numbers of pages linking to each other.

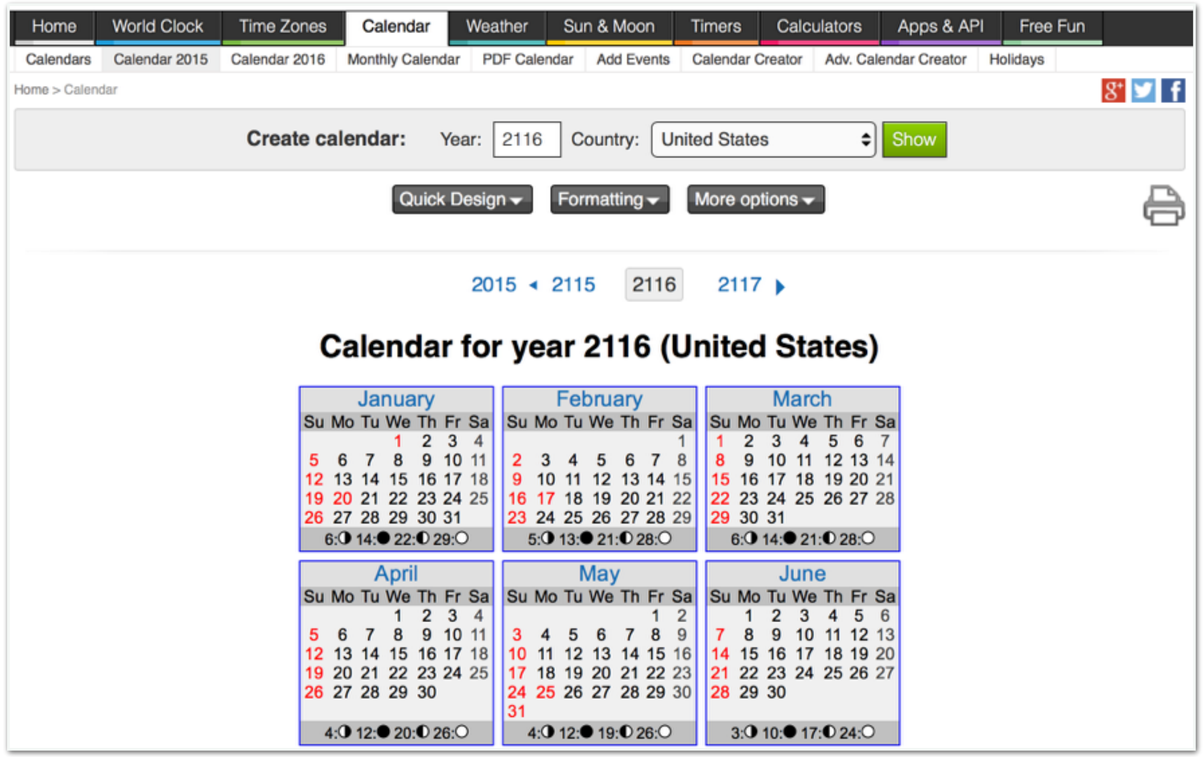
They assign a budget to each domain based on the number of in-links from other domains. The crawler de-prioritizes links from domains which have exceeded their budget, so link-filled spam domains are largely ignored.

Spider Traps

A *spider trap* is a collection of web pages which, intentionally or not, provide an infinite space of URLs to crawl.

Some site administrators place spider traps on their sites in order to trap or crash spambots, or defend against malicious bandwidth-consuming scripts.

A common example of a benign spider trap is a calendar which links continually to the next year.



A benign spider trap on http://www.timeanddate.com

Avoiding Spider Traps

The first defense against spider traps is to have a good politeness policy, and always follow it.

- By avoiding frequent requests to the same domain, you reduce the possible damage a trap can do.
- Most sites with spider traps provide instructions for avoiding them in robots.txt.

```
[...]
User-agent: *
Disallow: /createshort.html
Disallow: /scripts/savecustom.php
Disallow: /scripts/wquery.php
Disallow: /scripts/tzq.php
Disallow: /scripts/savepersonal.php
Disallow: /information/mk/
Disallow: /information/feedback-save.php
Disallow: /information/feedback.html?
Disallow: /gfx/stock/
Disallow: /bm/
Disallow: /eclipse/in/*?iso
Disallow: /custom/save.php
Disallow: /calendar//index.html
Disallow: /calendar//monthly.html
Disallow: /calendar//custom.html
Disallow: /counters//newyeara.html
Disallow: /counters//worldfirst.html
[...]
```

From http://www.timeanddate.com/robots.txt

Wrapping Up

A breadth-first search implementation of crawling is not sufficient for coverage, freshness, spam avoidance, or other needs of a real crawler.

Scaling the crawler up takes careful engineering, and often detailed systems knowledge of the hardware architecture you're developing for.

Next, we'll look at how to efficiently store the content we've crawled.

Storing Crawled Content

Crawling, session 8

Northeastern University
College of Computer and Information Science

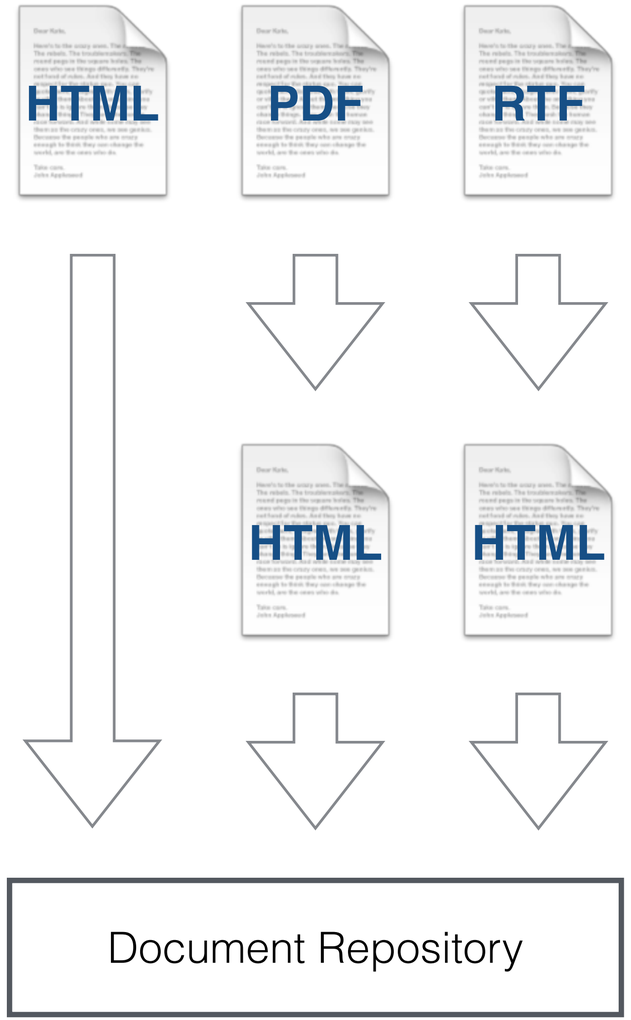
Slides by: Jesse Anderton

Content Conversion

Downloaded page content generally needs to be converted into a stream of tokens before it can be indexed.

Content arrives in hundreds of incompatible formats: Word documents, PowerPoint, RTF, OTF, PDF, etc. Conversion tools are generally used to transform them into HTML or XML.

Depending on your needs, the crawler may store the raw document content and/or normalized content output from a converter.



Character Encodings

Crawled content will be represented with many different character encodings, which can easily confuse text processors.

A *character encoding* is a map from bits in a file to glyphs on a screen. In English, the basic encoding is ASCII.

ASCII uses 8 bits: 7 bits to represent 128 letters, numbers, punctuation, and control characters and an extra bit for padding.

USASCII code chart

Bits $b_7 b_6 b_5$ →					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4	b_3	b_2	b_1	Row ↓ Column →	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

Image courtesy Wikipedia

Unicode

The various Unicode encodings were invented to support a broader range of characters. Unicode is a single mapping from numbers to glyphs, with various encoding schemes of different sizes.

- UTF-8 uses one byte for ASCII characters, and more bytes for extended characters. It's often preferred for file storage.
- UTF-32 uses four bytes for every character, and is more convenient for use in memory.

	ASCII	UTF-8	UTF-32
A	0x41	0x41	0x00000041
&	0x26	0x26	0x00000026
π	N/A	0xCF 0x80	0x000003C0
👍🏼	N/A	0xF0 0x9F 0x91 0x8D	0x0001F44D

UTF-8

UTF-8 uses a variable-length encoding scheme.

If the most significant (leftmost) bit of a given byte is set, the character takes another byte.

The first 128 numbers are the same as ASCII, so any ASCII document could be said to (retroactively) use UTF-8.

UTF-8 is designed to minimize disk space for documents in many languages, but UTF-32 is faster to decode and easier to use in memory.

UTF-8 Encoding Scheme

Decimal	Hexadecimal	Encoding			
0–127	0–7F	0xxxxxxx			
128–2047	80–7FF	110xxxxx	10xxxxxx		
2048–55295	800–D7FF	1110xxxx	10xxxxxx	10xxxxxx	
55296–57343	D800–DFFF	Undefined			
57344–65535	E000–FFFF	1110xxxx	10xxxxxx	10xxxxxx	
65536–1114111	10000–10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Document Repositories

What do we need from our document repository?

- **Fast random access** – need to store and obtain documents by their URLs (or a hash of the URL)
- **Fast document updates** – need to associate and update metadata with documents, and replace (or append to) records when documents are re-crawled
- **Compressed storage** – greatly reduces storage needs, and minimizes disk reads for access
- **Large file storage** – multiple documents are stored in a single large file to reduce filesystem overhead

Most companies use custom storage systems, or distributed systems like Big Table.

Large File Storage

Placing millions or billions of web pages in individual files results in substantial filesystem overhead for opening, writing, and finding files.

It's important to store many files into larger files, generally with an indexing scheme to give fast random access.

A simple index might store a B-tree mapping document URL hash values to the byte offset to the document contents in the file.

TREC Web Format

```
<DOC>
<DOCNO>WTX001-B01-10</DOCNO>
<DOCHDR>
http://www.example.com/test.html 204.244.59.33 19970101013145 text/html 440
HTTP/1.0 200 OK
Date: Wed, 01 Jan 1997 01:21:13 GMT
Server: Apache/1.0.3
Content-type: text/html
Content-length: 270
Last-modified: Mon, 25 Nov 1996 05:31:24 GMT
</DOCHDR>
<HTML>
<TITLE>Tropical Fish Store</TITLE>
Coming soon!
</HTML>
</DOC>
<DOC>
<DOCNO>WTX001-B01-109</DOCNO>
<DOCHDR>
http://www.example.com/fish.html 204.244.59.33 19970101013149 text/html 440
HTTP/1.0 200 OK
Date: Wed, 01 Jan 1997 01:21:19 GMT
Server: Apache/1.0.3
Content-type: text/html
Content-length: 270
Last-modified: Mon, 25 Nov 1996 05:31:24 GMT
</DOCHDR>
<HTML>
<TITLE>Fish Information</TITLE>
This page will soon contain interesting
information about tropical fish.
</HTML>
</DOC>
```

Wrapping Up

We need to normalize and store the contents of web documents so they can be indexed, so snippets can be generated, and so on.

Online documents have many formats and encoding schemes. There are hundreds of character encoding systems we haven't mentioned here.

A good document storage system should support efficient random access for lookups, updates, and content retrieval. Often, a distributed storage system like Big Table is used.

Next, we'll look at how to tune a crawler for a vertical search engine.

Vertical Search

Crawling, session 9

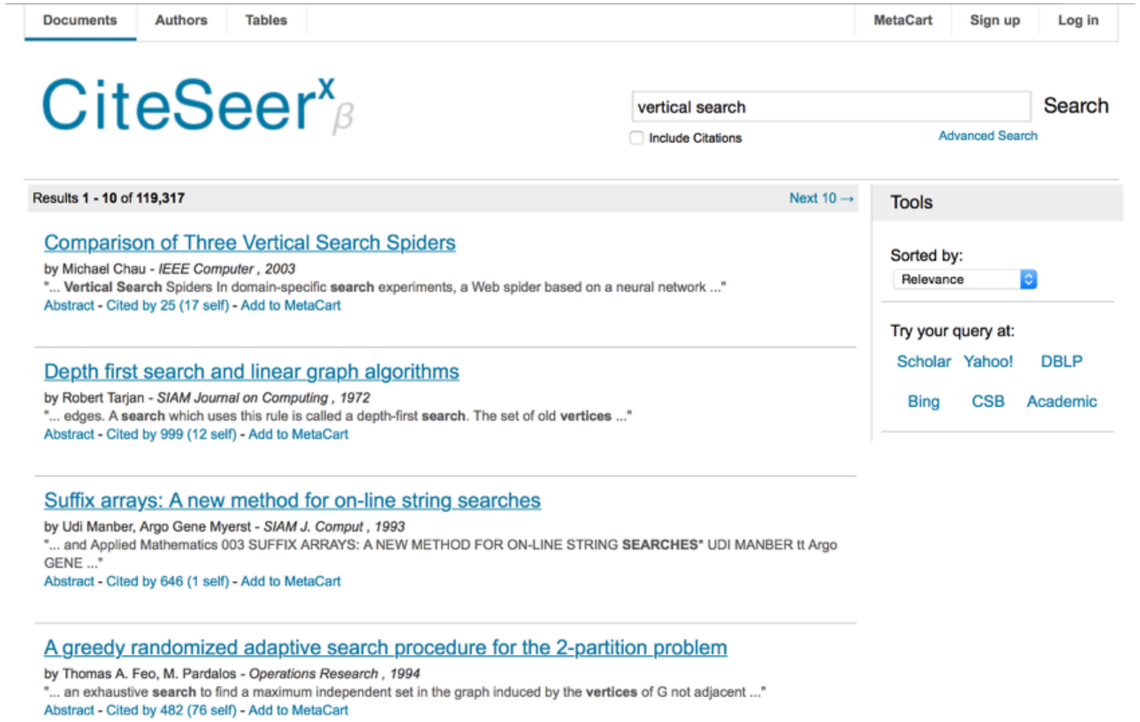
Northeastern University
College of Computer and Information Science

Slides by: Jesse Anderton

Vertical Search

Vertical Search engines focus on a particular domain of information.

The primary difference between vertical and general search engines is the set of documents they crawl. Vertical Search engines typically use what are known as *topical crawlers*.



CiteSeer, Vertical Search for Research

Topical Crawlers

Topical Crawlers focus on documents related to a particular topic of interest.

These crawlers are useful for improving the collection quality of general search engines, too. Many search engines use a variety of topical crawlers to supplement their primary crawler.

A basic approach uses a topical set of seed URLs and text classifiers to decide whether links appear to be on topic.

```
def crawl(seeds):

    # High quality topical hubs are used as seeds
    frontier.add_pages(seeds)

    # Iteratively crawl the next item in the frontier
    while not frontier.is_empty():

        # Crawl the next URL and extract anchor tags from it
        url = frontier.choose_next()
        page = crawl_url(url)
        urls = parse_page(page)

        # The URLs are filtered to stay on topic
        urls = filter_by_topic(urls)

        # Update the frontier and send the page to the indexer
        frontier.add_pages(urls)
        send_to_indexer(page)
```

Basic Topical Crawler

Text Classifiers

Text classification is a Machine Learning task that we'll see later in the course.

The idea is to use properties of the URL, anchor text, and document to predict whether the URL links to a page on the topic of interest.

For example, we could use a unigram language model trained on anchor text for topical links.

Classification with Language Models

1. Collect anchor text for links to topical and non-topical pages.

2. Train a unigram language model by producing smoothed probability estimates of topicality for each term.

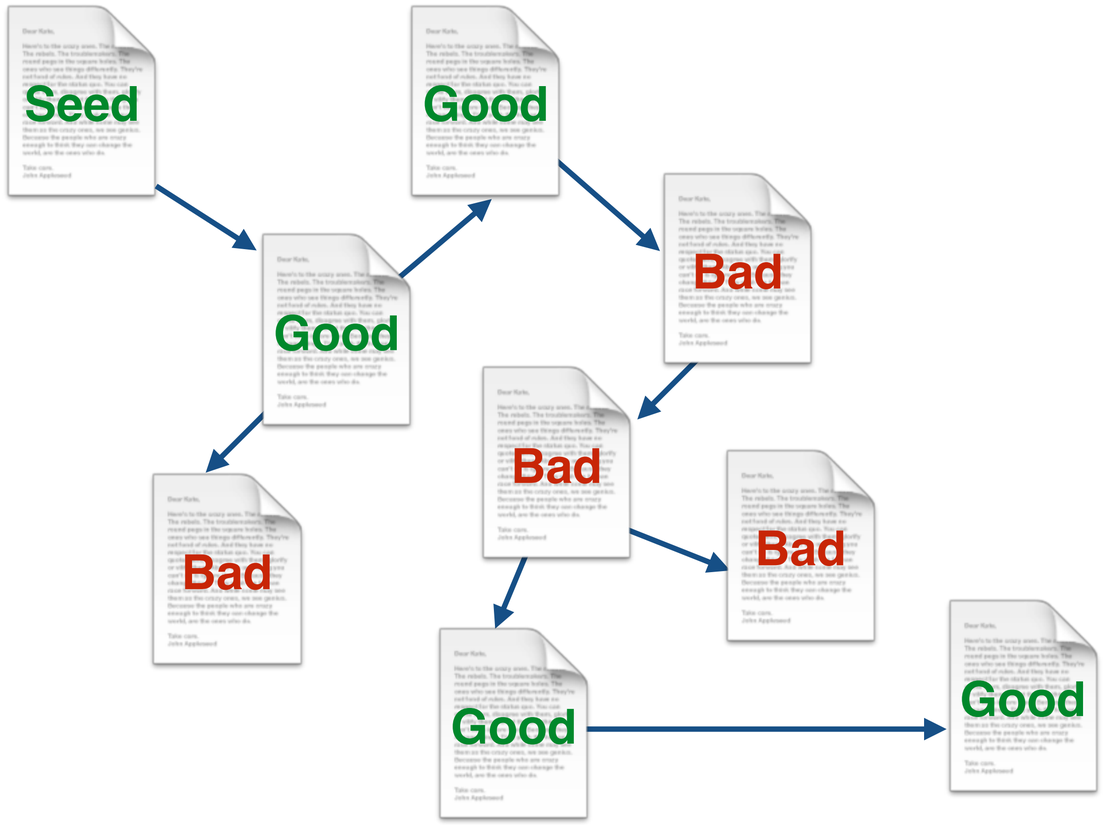
3. Classify new links using the odds ratio from training data for some threshold λ:

$$\prod_{w \in text} \frac{Pr(w|topic = 1)}{Pr(w|topic = 0)} \stackrel{?}{>} \lambda$$

Explore vs. Exploit Tradeoff

More sophisticated topical crawlers use machine learning techniques to balance the tradeoff between *exploring* new territory and *exploiting* links which are probably high-quality.

- Exploit-only strategies may miss high quality pages which aren't tightly linked to the seed set.
- Explore-only strategies will ignore high-quality pages we can easily find.



Sometimes bad links must be explored to find good links

Careful Exploration

There are many ways to balance exploration and exploitation, and this topic is actively researched for many applications. Here are some simple ways for this task.

- Adjust the classification threshold to manage your risk threshold.
- Flip a biased coin to decide whether to visit a page which doesn't seem promising.
- If using a document quality score such as PageRank, explore for a while without updating quality scores. Links on crawled pages won't be taken into account, so scores will be somewhat inaccurate and you will explore more.

There are more sophisticated approaches if maximizing performance is important.

Wrapping Up

Vertical Search depends on crawling a collection on the topic of interest.

General search engines also use topical crawlers to improve their coverage for key topics.

The main trick to topical crawling is finding topical pages which are only reachable by exploring off-topic pages through careful risk-taking.

Next, we'll look at crawling based on feeds.

Crawling Structured Data

Crawling, session 10

Northeastern University
College of Computer and Information Science

CS6200: Information Retrieval

Slides by: Jesse Anderton

Structured Web Data

In addition to unstructured document contents, a great deal of structured data exists on the web. We'll focus here on two types:

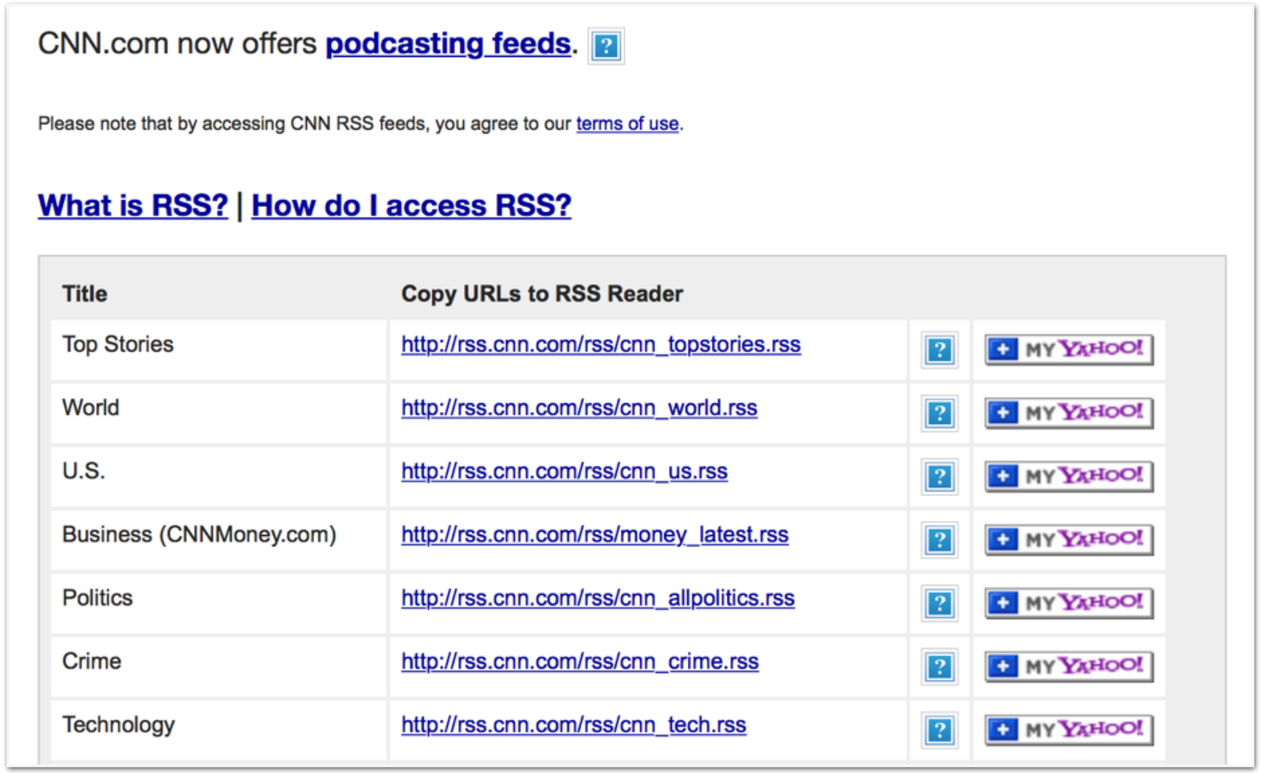
- Document feeds, which sites use to announce their new content
- Content metadata, used by web authors to publish structured properties of objects on their site

Document Feeds

Sites which post articles, such as blogs or news sites, typically offer a listing of their new content in the form of a document feed.

Several common feed formats exist. One of the most popular is RSS, which stands for (take your pick):

- Rich Site Summary
- Really Simple Syndication
- RDF Site Summary
- …?



CNN.com now offers **podcasting feeds**.

Please note that by accessing CNN RSS feeds, you agree to our terms of use.

What is RSS? | **How do I access RSS?**

Title	Copy URLs to RSS Reader
Top Stories	http://rss.cnn.com/rss/cnn_topstories.rss
World	http://rss.cnn.com/rss/cnn_world.rss
U.S.	http://rss.cnn.com/rss/cnn_us.rss
Business (CNNMoney.com)	http://rss.cnn.com/rss/money_latest.rss
Politics	http://rss.cnn.com/rss/cnn_allpolitics.rss
Crime	http://rss.cnn.com/rss/cnn_crime.rss
Technology	http://rss.cnn.com/rss/cnn_tech.rss

http://www.cnn.com/services/rss/

RSS Format

RSS is an XML format for document listings.

RSS files are obtained just like web pages, with HTTP GET requests.

The `ttl` field provides an amount of time (in minutes) that the contents should be cached.

RSS feeds are very useful for efficiently managing freshness of news and blog content.

```
<?xml version="1.0"?>
<rss version="2.0">
  <channel>
    <title>Search Engine News</title>
    <link>http://www.search-engine-news.org/</link>
    <description>News about search engines.</description>
    <language>en-us</language>
    <pubDate>Tue, 19 Jun 2008 05:17:00 GMT</pubDate>
    <ttl>60</ttl>

    <item>
      <title>Upcoming SIGIR Conference</title>
      <link>http://www.sigir.org/conference</link>
      <description>The annual SIGIR conference is coming!
        Mark your calendars and check for cheap
        flights.</description>
      <pubDate>Tue, 05 Jun 2008 09:50:11 GMT</pubDate>
      <guid>http://search-engine-news.org#500</guid>
    </item>
    <item>
       <title>New Search Engine Textbook</title>
       <link>http://www.cs.umass.edu/search-book</link>
       <description>A new textbook about search engines
         will be published soon.</description>
       <pubDate>Tue, 05 Jun 2008 09:33:01 GMT</pubDate>
       <guid>http://search-engine-news.org#499</guid>
     </item>
   </channel>
 </rss>
```

RSS Example

Structured Data

Many web pages are generated from structured data in databases, which can be useful for search engines and other crawled document collections.

Several schemas exist for web authors to publish their structured data for these tools.

The WHATWG web specification working group has produced several standard formats for this data, such as *microdata* embedded in HTML.

```
<section itemscope itemtype="http://schema.org/Person">
    Hello, my name is
    <span itemprop="name">John Doe</span>,
    I am a
    <span itemprop="jobTitle">graduate research assistant</span>
    at the
    <span itemprop="affiliation">University of Dreams</span>.
    My friends call me
    <span itemprop="additionalName">Johnny</span>.
    You can visit my homepage at
    <a href="http://www.JohnnyD.com" itemprop="url">www.JohnnyD.com</a>.
    <section itemprop="address" itemscope itemtype="http://schema.org/PostalAddress">
        I live at
        <span itemprop="streetAddress">1234 Peach Drive</span>,
        <span itemprop="addressLocality">Warner Robins</span>,
        <span itemprop="addressRegion">Georgia</span>.
    </section>
</section>
```

Source: http://en.wikipedia.org/wiki/Microdata_(HTML)

Web Ontologies

The main web ontology is published at schema.org. These schemas are used to annotate web pages for automated information extraction tools.

As the published information is not necessarily authoritative, the data needs to be carefully validated for quality and spam removal.

Popular schema.org entities

- Creative works: CreativeWork, Book, Movie, MusicRecording, Recipe, TVSeries ...
- Embedded non-text objects: AudioObject, ImageObject, VideoObject
- Event
- Health and medical types: notes on the health and medical types under MedicalEntity.
- Organization
- Person
- Place, LocalBusiness, Restaurant ...
- Product, Offer, AggregateOffer
- Review, AggregateRating
- Action

Wrapping Up

In addition to the obvious content for human readers, the web contains a great deal of structured content for use in automated systems.

- Document feeds are an important way to manage freshness at some of the most frequently-updated web sites.
- Much of the structured data owned by various web entities is published in a structured format. This can provide signals for relevance, and can also aid in reconstructing structured databases.