http://webdam.inria.fr

Web Data Management

Web Search

Serge Abiteboul
INRIA Saclay & ENS Cachan

Ioana Manolescu
INRIA Saclay & Paris-Sud University

Philippe Rigaux
CNAM Paris & INRIA Saclay

Marie-Christine Rousset
Grenoble University

Pierre Senellart
Télécom ParisTech

To be published by Cambridge University Press

`http://webdam.inria.fr/Jorge/`

Contents

With a constantly increasing size of dozens of billions of freely accessible documents, one of the major issues raised by the World Wide Web is that of searching in an effective and efficient way through these documents to find these that best suit a user's need. The purpose of this chapter is to describe the techniques that are at the core of today's search engines (such as Google[1], Yahoo![2], Bing[3], or Exalead[4]), that is, mostly keyword search in *very large* collections of text documents. We also briefly touch upon other techniques and research issues that may be of importance in next-generation search engines.

This chapter is organized as follows. In Section 1, we briefly recall the Web and the languages and protocols it relies upon. Most of these topics have already been covered earlier in the book, and their introduction here is mostly intended to make the present chapter self-contained. We then present in Section 2 the techniques that can be used to retrieve pages from the Web, that is, to *crawl* it, and to extract text tokens from them. First-generation search engines, exemplified by Altavista[5], mostly relied on the classical information retrieval (IR) techniques, applied to text documents, that are described in Section 3. The advent of the Web, and more generally the steady growth of documents collections managed by institutions of all kinds, has led to extensions of these techniques. We address scalability issues in Section 3.3, with focus on centralized indexing. Distributed approaches are investigated in Chapter **??**. The graph structure of the Web gives rises to *ranking* techniques that very

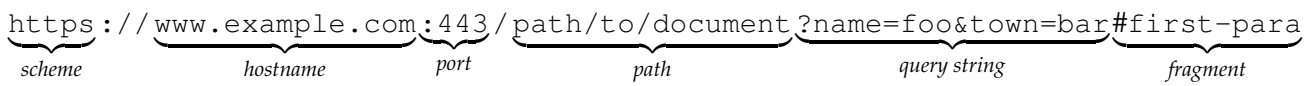
[1] `http://www.google.com/`
[2] `http://www.yahoo.com/`
[3] `http://www.bing.com/`
[4] `http://www.exalead.com/`
[5] `http://www.altavista.com/`

effectively complement information retrieval. We conclude with a brief discussion of currently active research topics about Web search in Section 5.

1 The World Wide Web

Whereas the Internet is a physical network of computers (or *hosts*) connected to each other from all around the world, the World Wide Web, WWW or Web in short, is a logical collection of hyperlinked documents shared by the hosts of this network. A hyperlinked document is just a document with references to other documents of the same collection. Note that documents of the Web may both refer to static documents stored on the hard drive of some host of the Internet, and to dynamic documents that are generated on the fly. This means that there is a virtually unlimited number of documents on the Web, since dynamic documents can change at each request. When one speaks of the Web, it is mostly about the *public* part of the Web, which is freely accessible, but there are also various *private* Webs that are restricted to some community or company, either on private *Intranets* or on the Internet, with password-protected pages.

Documents and, more generally, *resources* on the Web, are identified by a *URL* (*Uniform Resource Locator*) which is a character string that follows a fixed format illustrated on the imaginary URL below, with basic components, described next.



scheme: describes the way the resource can be accessed; on the Web, it is generally one of the Web *protocols* (`http`, `https`) that are described further.

hostname: this is the domain name of a host, as given by the domain name system (DNS). Frequently on the Web, the hostname of a website will start with `www.`, but this is only a common convention, not a rule.

port: TCP port where the server listens on the host; it defaults to 80 for the `http` scheme and 443 for the `https` scheme and is rarely present.

path: the logical path of the document; for simple cases, this corresponds to a path leading to the static document in the filesystem of the host.

query string: additional parameters identifying the resource, mostly used with dynamic documents.

fragment: identifies a specific part of the document.

Query strings and fragments are optional (and, most of the time, absent) and the path can be omitted, in which case the URL refers to the *root* of the Web host. URLs can also be relative (by opposition to the *absolute URL* above), when both the scheme and hostname portions are omitted. A relative URL is to be interpreted in a given URL *context* (for instance, the URL of the current document) and is resolved in a straightforward way: if the context is that of the URL above, the relative URLs `/images`[6] and `data` would be resolved, respectively,

[6]Note that here `/images` is considered as a relative URL, because it lacks the scheme and hostname part; the path `/images`, however, is an absolute path.

as `https://www.example.com:443/images` and `https://www.example.com:443/path/to/data` in a way similar to (Unix) relative paths resolution.

The usual format for documents (or, in this case, *pages*) on the Web is HTML (the HyperText Markup Language), though one can find many documents, including hyperlinked documents, in other formats. This is the case for PDF documents (another kind of hyperlinked structure), documents from word-processing softwares, and non-textual, *multimedia* documents such as images and audio files.

```
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      lang="en" xml:lang="en">
  <head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=utf-8" />
    <title>Example XHTML document</title>
  </head>
  <body>
    <p>This is a
      <a href="http://www.w3.org/">link to the
      <strong>W3C</strong>!</a></p>
  </body>
</html>
```

Figure 1: Example XHTML Document

HTML is originally a dialect of SGML, the ancestor of XML, but is hardly ever parsed as such. The most common versions found on today's Web are HTML 4.01 and XHTML 1.0, which is a direct XMLization of HTML 4.01, with minor differences. An example XHTML document is given in Figure 1. As it is an SGML or XML file, an (X)HTML document is made out of elements, attributes and text content. Elements carry, between other things, meta-information about the document (e.g., `<meta>`, `<title>`), structural information at the document level (e.g., `<table>`, `<ul>`, `<p>`), structural information at the character level (e.g., `<strong>`, `<em>`) or references to other media (e.g., `<img>`, `<object>`). An element of importance is `<a>`, which defines a hyperlink to another resource on the Web identified by the URL given as the `href` attribute. Both relative and absolute links are allowed here. The context of resolution is the URL of the current page, unless it contains a `<base>` element that indicates another context. HTML pages can also contain other *disguised* hyperlinks in the form of JavaScript code that loads other URLs, of redirection after a timeout with some specific use of the `<meta>` element, or of Flash or Java applets; all these links are less easy to identify and then less accessible to users and Web robots.

Although HTML pages are primarily seen thanks to a browser (and, most of the time, a graphical browser as Microsoft Internet Explorer or Mozilla Firefox), the HTML code is not supposed to describe the way the page will appear in a browser (it is the role of a styling language like CSS) but the core structure and content of the document in a way accessible to

Request	`GET /myResource HTTP/1.1` `Host: www.example.com`

Response	`HTTP/1.1 200 OK`
	`Content-Type: text/html; charset=ISO-8859-1`
	
	`<html>`
	`  <head><title>myResource</title></head>`
	`  <body><p>Hello world!</p></body>`
	`</html>`

Figure 2: HTTP request and response examples

all kind of browsers and a wide variety of *user agents* such as the crawlers that we describe in Section 2.1. For this reason, it is important that HTML documents be valid against the W3C specifications; tools like the W3C validator available at `http://validator.w3.org/` can be of help. Sadly, because of a history of browser wars, browser limitations, browser permissiveness, and author laziness, most of the (X)HTML pages on the Web are far from being valid, or even well-formed in the sense of XML well-formedness, accounting for what has been called *tag soup*.

Pages of the Web are accessed using the usual client-server architecture of the Internet: a *Web server* on a remote host accepts requests from a client for a given resource, and provides it to him. Two communication protocols are mainly used for this exchange: HTTP and HTTPS. The latter allows for encryption, authentication and advanced features such as session tracking; it is essential for e-commerce on the Web and all other sensitive applications, but rarely used for regular documents and will not be discussed further. HTTP, or *HyperText Transfer Protocol*, is a quite simple protocol built on top of the Internet protocols IP (*Internet Protocol*, for addressing) and TCP (*Transmission Control Protocol*, for transportation) that is widely used on the World Wide Web. Figure 2 shows an example request and response from, respectively, a Web client and a Web server. The request asks for the resource identified by the path `/myResource` on host `www.example.com` and the server answers that the document was found (code `200 OK`; other common codes are `404 NOT FOUND` or `500 SERVER ERROR`) and provides it. The reason why the hostname is given, whereas the server has already been contacted, is that a given server may have several different domain names, with different content (thus, `www.google.com` and `www.google.fr` point to the same machine). This *virtual hosting* is one of the novel feature of HTTP/1.1 with respect to previous versions. Other features of the HTTP protocol include login and password protection, content negotiation (the content served for a given URL is not fixed and depends on the preference indicated by the client, in terms of file formats or languages), cookies (persistent chunks of information that are stored on the client, e.g., for session management purpose), *keep-alive* requests (several requests can be made to the same server without closing the connection), and more.

2 Parsing the Web

The first task to build a search engine over the Web is to retrieve and index a significant portion of it. This is done by collecting Web documents through a navigation process called

Web crawling. These documents are then processed to extract relevant information. In case of text documents, sets of words or *tokens* are collected. Web crawling is introduced in Section 2.1. We describe in Section 2.2 general text preprocessing techniques that turn out to be useful for document retrieval.

2.1 Crawling the Web

Crawling is done by user agents that are called *crawlers*, *(Web) spiders* or *(Web) robots*. Their design raises a number of important engineering issues that will be discussed here.

Discovering URLs

Crawling the Web is basically just starting from a given URL or set of URLs, retrieving and indexing this document, discovering hyperlinks (mostly from the `<a>` elements of the HTML content) on the document and repeating the process on each link. There is no real termination condition here, as it is vain to try to retrieve the entire Web (which is actually virtually infinite, as already discussed), but the crawl can be terminated after some delay or after some number of URLs have been discovered or indexed. This is essentially a graph browsing problem, which can be tackled by either *breadth-first* (all pages pointed by a page are indexed before the links they contain are analyzed) or *depth-first* (a referred page is indexed, and its links are extracted in turn, as soon as a link to it is discovered) techniques. Obviously, because of the non-existence of termination conditions, and the possibility of being lost in *robot traps* (infinite paths in the graph), a breadth-first approach is more adapted here; actually, a mix of a breadth-first browsing with depth-first browsing of limited depth of each discovered site can be a good compromise.

There are other sources of URLs for Web crawlers than the ones found on Web pages. Web search engines have access to information about the Web page that a Web user comes from when she reaches the Web site of the search engine, thanks to the `Referrer` HTTP header. It can be used to discover pages that are not referenced anywhere. Finally, Web masters can provide search engines with *sitemaps*, a file, in XML format, that can be used to list all URLs in a given Web site, along with some meta-information (date of last modification, refresh rate). When this is available and properly used, for example automatically generated by a content management system, the situation is ideal: a Web crawler can get the list of all URLs of a Web site with a single HTTP request.

Deduplicating Web Pages

An important subtask of Web crawling is the identification of duplicate pages in the Web, in order to avoid browsing them multiple times. Trivial duplicates are documents that share the same URL, though it can be written in slightly different ways: this means that a *canonization* of URLs has to be performed, to detect for instance that `http://example.com:80/foo` and `http://example.com/foo/../bar` are actually the same resource. The identification of other kind of duplicates, that do not have the same URL, is more intricate but also crucial, since it would not be very interesting for a user to get a list of identical pages as a result to a search engine query. Identical duplicates are easy to identify by hashing, but there are often some little differences between two pages (for instance, a date that is automatically generated

at each request by the server, or random content such as *Tips of the day*) that present essentially the same content.

The first approach to detect such near-duplicates is simply to compute the *edit distance* between two documents, that is, the minimal number of basic modifications (additions or deletions of characters or words, etc.) to obtain a document from another one. This is a good notion of similarity between textual documents, and edit distance can be computed with dynamic programming in $O(m \cdot n)$ where m and n are the size of the documents. This does not scale to a large collection of documents, since it is definitely unreasonable to compute the edit distance between every pair of documents found on the Web. An alternative is the use of *shingles*: a shingle is a sequence of tokens (say, words) of a fixed length k (for instance, three consecutive words), as found in a document. Consider the following two simple documents:

$$d = \text{I like to watch the sun set with my friend.}$$
$$d' = \text{My friend and I like to watch the sun set.}$$

One can compute the set of shingles of length $k = 2$, disregarding punctuation and putting all tokens in lowercase:

$$S = \{\text{i like}, \text{like to}, \text{to watch}, \text{watch the}, \text{the sun}, \text{sun set}, \text{set with}, \text{with my}, \text{my friend}\}$$
$$S' = \{\text{my friend}, \text{friend and}, \text{and i}, \text{i like}, \text{like to}, \text{to watch}, \text{watch the}, \text{the sun}, \text{sun set}\}$$

The similarity between two documents can then be computed as the proportion of common shingles, using the Jaccard coefficient:

$$J(S, S') = \frac{|S \cap S'|}{|S \cup S'|} = \frac{7}{11} \approx 0.64$$

For a value of k between 2 to 10 depending on the applications, this gives a reasonable way to compare two textual documents (the markup of a HTML page can be either considered as part of the tokens, or disregarded altogether, depending on the granularity wished in the comparison). However, this is still costly to compute, since one has to compute the similarity between every two pair of documents. It is however possible to approximate $J(S, S')$ by storing a *summary* (or *sketch*) of the shingles contained in a document of a fixed size. Let N be a fixed integer, which is the size of the summary that will be kept for each document. The following algorithm can be shown to approximate in an unbiased way the Jaccard similarity between sets of shingles:

1. Choose N different and *independent* hash functions;
2. For each hash function h_i and set of shingles $S_k = \{s_{k1} \dots s_{kn}\}$, store $\phi_{ik} = \min_j h_i(s_{kj})$;
3. Approximate $J(S_k, S_l)$ as the proportion of ϕ_{ik}'s and ϕ_{il}'s that are equal:

$$J(S_k, S_l) \approx \frac{|\{i : \phi_{ik} = \phi_{il}\}|}{N}.$$

Then, in order to test if a document is a near-duplicate of one that has already been found, it is enough to compute its sketch of hashed shingles, and to make N accesses into a hash table. The larger N is, the better the approximation, but the costlier the computation. It is also possible to repeat the hashing on the set of hashed shingles to obtain a set of hashed *super-shingles*, when one is only interested in *very* similar documents, as is the case for finding near-duplicates.

```
User-agent: *
Allow: /searchhistory/
Disallow: /search
```

Figure 3: Example `robots.txt` robot exclusion file

Crawling Ethics

Going back to crawling *per se*, there are also some *crawling ethics* to abide to. A *standard for robot exclusion* has been proposed to allow webmasters to specify some pages not to be crawled by Web spiders (the reasons can be varied: for confidentiality purposes, in order not to put too heavy a load on a resource-intensive Web application, to help robots not to fall into robot traps, etc.). This standard is followed by all major search engines, and consists in a `/robots.txt` file, that can be put at the root of every Web server and contains restrictions on what part of the website spiders are allowed to crawl.

An example of such a file is given in Figure 3. It disallows the crawling of all URLs whose path starts with `/search`, with the exception of those starting with `/searchhistory/`, to any robots. Another way of expressing such limitations, at the level of a HTML page this time (which can be useful if a webmaster does not have control over the document root), is through a `<meta name="ROBOTS">` directive in the header of the document, such as

```
<meta name="ROBOTS" content="NOINDEX,NOFOLLOW">
```

which disallows robots to either index or follow links from the current Web page. Available keywords are `INDEX`, `FOLLOW`, `NOINDEX`, `NOFOLLOW`, with a default of `INDEX,FOLLOW`. Yet another way of influencing robot crawling and indexing is discussed in Section 4.3. A last rule that a spider programmer should respect is to avoid too many requests in a short time to a given host, since that could result in DOS (Denial Of Service) from the host. A good rule of thumb is to wait between 100ms and 1s between two successive requests to the same Web server.

Design Issues

Because of this last rule, and because network delays are typically much higher than the time needed to process a Web page, it is crucial to send in parallel a large number of requests to different hosts; this also means that a per-host queue of URLs to be processed has to be managed. This parallel processing is typically performed using a multi-threaded environment, although asynchronous input and outputs (with for instance the `select` POSIX C function) provide the same functionality without the overhead introduced by threads. The *keep-alive* feature of HTTP/1.1 can also be used to chain requests (after some delay) to the same host. In large-scale crawlers for general Web search, the crawling itself will be run in parallel on a number of machines, that have to be regularly synchronized, which raises further issued not discussed here.

Another aspect of crawling is the refreshing of URLs. Though we stated earlier that we did not want to crawl twice the same URL, it can be very important to do so in the context of perpetually changing Web content. Furthermore, it is also important to identify frequently changing Web pages in order to crawl them more often than others. Thus, the main page

d_1 The jaguar is a New World mammal of the Felidae family.
d_2 Jaguar has designed four new engines.
d_3 For Jaguar, Atari was keen to use a 68K family device.
d_4 The Jacksonville Jaguars are a professional US football team.
d_5 Mac OS X Jaguar is available at a price of US $199 for Apple's new "family pack".
d_6 One such ruling family to incorporate the jaguar into their name is Jaguar Paw.
d_7 It is a big cat.

Figure 4: Example set of documents

d_1 the_1 $jaguar_2$ is_3 a_4 new_5 $world_6$ $mammal_7$ of_8 the_9 $felidae_{10}$ $family_{11}$
d_2 $jaguar_1$ has_2 $designed_3$ $four_4$ new_5 $engines_6$
d_3 for_1 $jaguar_2$ $atari_3$ was_4 $keen_5$ to_6 use_7 a_8 $68k_9$ $family_{10}$ $device_{11}$
d_4 the_1 $jacksonville_2$ $jaguars_3$ are_4 a_5 $professional_6$ us_7 $football_8$ $team_9$
d_5 mac_1 os_2 x_3 $jaguar_4$ is_5 $available_6$ at_7 a_8 $price_9$ of_{10} us_{11} $\$199_{12}$ for_{13} $apple's_{14}$ new_{15} $family_{16}$ $pack_{17}$
d_6 one_1 $such_2$ $ruling_3$ $family_4$ to_5 $incorporate_6$ the_7 $jaguar_8$ $into_9$ $their_{10}$ $name_{11}$ is_{12} $jaguar_{13}$ paw_{14}
d_7 it_1 is_2 a_3 big_4 cat_5

Figure 5: Tokenization of document set of Figure 4

of an online newspaper should probably be crawled every day or even more often, while it may take up to a month to a large-scale crawler to crawl a significant portion of the Web. The HTTP protocol proposes a way to retrieve a document only if it has been modified since some given date (`If-Modified-Since` header) but this is often unreliable, even more so in the context of dynamic documents which are regenerated at each request. Changes in Web pages have then to be identified by the crawler, without taking into account minor changes, for instance using techniques described above for identifying near-duplicates. Crawling strategies have to be adapted accordingly.

2.2 Text Preprocessing

The techniques described here are general techniques for dealing with text corpora. Depending upon the application, some variant or other has to be applied. Furthermore, the original language or languages of the document have a major impact on the preprocessing made.

Tokenization

Let us consider the set of 7 (one-sentence) documents represented in Figure 4. We describe next and illustrate on this particular example how to process a collection of text documents in a suitable way to efficiently answer keyword queries. This problem and related ones are known as *information retrieval* or, simply, *search* problems.

The first step is to tokenize the initial documents into sequences or *tokens*, or simply words. This is illustrated on our example document set in Figure 5. At this stage, inter-word punctuation is generally removed and case is normalized (unless, obviously, the application requires differently, as may be the case in a search engine dedicated to linguists researching the usage of punctuations). This may seem like a very simple step where it is sufficient to replace whitespaces and punctuation by token separators, but the problem is actually trickier

than this.

- Whereas words are immediately visible in a language such as English, other languages (notably, Chinese or Japanese) do not use whitespace to mark word boundaries. Tokenization of such languages requires much more complex procedures that typically use both advanced linguistic routines and dictionaries. Specifically for Chinese, an alternative is to use individual ideograms as tokens, but this may lead to invalid matches in query answering.
- Some care has to be taken for a number of textual oddities, such as acronyms, elisions, numbers, units, URLs, e-mail addresses, etc. They should probably be preserved as single tokens in most applications, and in any case should be dealt with consistently.
- Even in English, tokenization is not always obvious, especially with respect to compound words. An immediate question is whether to consider intra-word hyphens as token separator, but the problem is broader. Consider for instance the term *hostname* that can be found in three variant forms: *hostname*, *host-name* and *host name*. If we want to be able to use any of these terms to query all three variants, some analysis has to be performed, probably with the help of a lexicon, either to consider *host name* as a single compound word, or to break *hostname* in two tokens. The latter solution will also allow searching for *host* and *name*, which may be appropriate, depending on the context. In languages where compounds are even more easily produced than in English, e.g., in German or Russian, such an analysis is indispensable.

Stemming

Once tokens are identified, an optional step is to perform some *stemming* in order to remove morphological markers from inflected words, or, more generally, to merge several lexically related tokens into a single *stem*. Such a step is often needed, for instance to be able to retrieve documents containing *geese* where *goose* is queried, but the degree of stemming varies widely depending upon the application (and upon the considered language, obviously: the notion of stemming does not make much sense in a language without any morphological variations like Chinese). Here is a scale of possible stemming schemes, from finest to coarsest:

Morphological stemming. This consists in the sole removal of bound morphemes (such as plural, gender, tense or mood inflections) from words. Note that this can be a very complex task in morphologically rich languages such as Turkish, or, in a lesser way, French, which require advanced linguistic processing for resolutions of *homographs* (different words that are written in the same way). Consider for instance the famous sentence in French: "Les poules du couvent couvent." (The hens of the monastery brood.) Here, the first *couvent* [monastery] is an uninflected noun, which should stay as is, whereas the second *couvent* [brood] is an inflected form of the verb *couver* [to brood], which should be stemmed accordingly. In English, the situation is simpler and plain procedures that remove final -s, -ed, -ing, etc., with a few adaptations for semi-regular (-y/-ies) or irregular (mouse/mice) inflections, can be enough. Some ambiguities remain, as illustrated with the words *rose* or *stocking*, which can be either uninflected nouns, or inflected forms of the verb *to rise* and *to stock*, respectively. Depending upon the application, one may choose either a cautious stemming (that does not remove all

d_1 the$_1$ jaguar$_2$ be$_3$ a$_4$ new$_5$ world$_6$ mammal$_7$ of$_8$ the$_9$ felidae$_{10}$ family$_{11}$
d_2 jaguar$_1$ have$_2$ design$_3$ four$_4$ new$_5$ engine$_6$
d_3 for$_1$ jaguar$_2$ atari$_3$ be$_4$ keen$_5$ to$_6$ use$_7$ a$_8$ 68k$_9$ family$_{10}$ device$_{11}$
d_4 the$_1$ jacksonville$_2$ jaguar$_3$ be$_4$ a$_5$ professional$_6$ us$_7$ football$_8$ team$_9$
d_5 mac$_1$ os$_2$ x$_3$ jaguar$_4$ be$_5$ available$_6$ at$_7$ a$_8$ price$_9$ of$_{10}$ us$_{11}$ \$199$_{12}$ for$_{13}$ apple$_{14}$ new$_{15}$ family$_{16}$ pack$_{17}$
d_6 one$_1$ such$_2$ rule$_3$ family$_4$ to$_5$ incorporate$_6$ the$_7$ jaguar$_8$ into$_9$ their$_{10}$ name$_{11}$ be$_{12}$ jaguar$_{13}$ paw$_{14}$
d_7 it$_1$ be$_2$ a$_3$ big$_4$ cat$_5$

Figure 6: Document set of Figure 4, after tokenization and stemming

morphological markers, and will then fail to retrieve some query matches) or a more aggressive one (that will retrieve invalid query matches). Figure 6 shows the result of a morphological stemming applied on our running example.

Lexical stemming. Stemming can be pushed further to merge lexically related words from different parts of speech, such as *policy*, *politics*, *political* or *politician*. An effective algorithm for such a stemming in English, Porter's stemming, has been widely used. Further ambiguities arise, with for instance *university* and *universal* both stemmed to *universe*. This kind of stemming can also be coupled to the use of lexicons in order to merge synonyms or near-synonyms.

Phonetic stemming. The purpose of phonetic stemming is to retrieve words despite spelling variations or errors. Soundex is a widely used loose phonetic stemming for English, especially known for its use in U.S. censuses, that stems for instance both *Robert* and *Rupert* to *R163*. As can be seen from this example, it is a very coarse form of stemming, and should probably not be used in contexts where the precision of matches is important.

In some circumstances, it can be useful to produce different indexes that use different forms of stemming, to support both exact and approximate queries.

Stop-word removal

The presence of some words in documents, such as determiners (*the*, *a*, *this*, etc.), function verbs (*be*, *have*, *make*, etc.), conjunctions (*that*, *and*, etc.) is very common and indexing them increases storage requirements. Furthermore, they are not informative: a keyword query on *be* and *have* is likely to retrieve almost all the (English) documents of the corpus. It is then common to ignore them in queries, and sometimes in the index itself (although it is hard to determine in advance whether an information can be useful or not). Figure 7 shows a further filtering of our running example that removes stop-words (to be compared with Figure 6).

3 Web Information Retrieval

Once all needed preprocessing has been performed, the (text) document set can be indexed in an *inverted file* (or inverted index) that supports efficient answering to keyword queries. Basically, such an index implements a binary association (i.e., a matrix) between the documents and the terms they contain. Documents are represented by their *ids*, a compact key that uniquely identifies a document. Note that the search system is not required to store the

d_1 jaguar$_2$ new$_5$ world$_6$ mammal$_7$ felidae$_{10}$ family$_{11}$
d_2 jaguar$_1$ design$_3$ four$_4$ new$_5$ engine$_6$
d_3 jaguar$_2$ atari$_3$ keen$_5$ 68k$_9$ family$_{10}$ device$_{11}$
d_4 jacksonville$_2$ jaguar$_3$ professional$_6$ us$_7$ football$_8$ team$_9$
d_5 mac$_1$ os$_2$ x$_3$ jaguar$_4$ available$_6$ price$_9$ us$_{11}$ \$199$_{12}$ apple$_{14}$ new$_{15}$ family$_{16}$ pack$_{17}$
d_6 one$_1$ such$_2$ rule$_3$ family$_4$ incorporate$_6$ jaguar$_8$ their$_{10}$ name$_{11}$ jaguar$_{13}$ paw$_{14}$
d_7 big$_4$ cat$_5$

Figure 7: Document set of Figure 4, after tokenization, stemming and stop-word removal

family	d_1, d_3, d_5, d_6
football	d_4
jaguar	$d_1, d_2, d_3, d_4, d_5, d_6$
new	d_1, d_2, d_5
rule	d_6
us	d_4, d_5
world	d_1
...	

Figure 8: Partial inverted index for document set of Figure 4

document itself, as long as it can be accessed from an external source using its id. Terms are, as discussed before, stemmed tokens that are not stop words. An inverted index supports very fast retrieval of the documents (ids) that contain the set of keywords in a query.

We describe next and illustrate on our running example how to index a collection of text documents in a suitable way to efficiently answer keyword queries. This problem and related ones are known as *information retrieval* or, simply, *search* problems. We present the *inverted index* model in Section 3.1. We then proceed to the problem of answering keyword queries using such an index. Large-scale indexing and retrieval is introduced in Section 3.3 and clustering in Section 3.4. We discuss briefly at the end of this section how we can go beyond traditional information retrieval techniques to search the Web.

3.1 Inverted Files

An inverted index is very similar to traditional indexes found at the end of printed books, where each term of interest is associated to a list of page numbers. In the IR context, an inverted index consists of a collection of *posting lists* $L(T)$, one for each term t, storing the ids (*postings*) of the documents where t occurs. A (partial) example of inverted index for our example is given in Figure 8.

For small-scale applications where the index fits on a single machine, lists of occurrences of documents are usually stored, packed, for each given term, in an *inverted file* that is then mapped to memory (using the POSIX system call `mmap`). A secondary structure (the *vocabulary*) gives the position of the list for each term in the index. In order to quickly find this position given a term t, the vocabulary can be indexed by, e.g., a B-tree.

Figure 9 shows the structure of an inverted index. Some details are noteworthy. First, the length of an inverted list is highly variable. This length depends on the number of referred documents in the list, which is high for common terms (t_1 in Figure 9), and small for rare ones

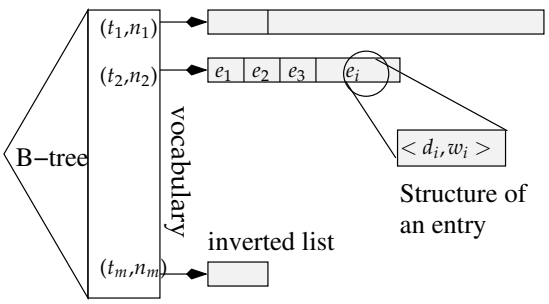


Figure 9: Structure of an inverted file

(t_m in Figure 9). In text processing, skewness is the norm and often the handle to strongly accelerate query processing by scanning the shortest lists first. The vocabulary features therefore, along with each term t_i, the number n_i of documents where t_i occurs.

Second, a list consists of a sequence of homogeneous *entries* which can often be quite efficiently compressed to limit the memory space required by the whole index. A key requirement for efficient compression is the ordering of entries on the document id (represented as an unsigned integer). The basic index illustrated in Figure 8 shows an ordering of each on the d_i component, assuming that $d_i < d_j$ when $i < j$.

Here is a first concrete example giving some measures for a basic representation of an inverted index.

Example 3.1 Consider an institution that wants to index its collection of emails. We assume that the average size of an email is 1,000 bytes and that each email contains an average of 100 words.

A collection of 1 million emails occupies 1 GB. It consists in 100×10^6 words. Suppose that, after parsing and tokenization, the number of distinct terms is 200,000. Then:

1. the index consists of 200,000 lists;
2. each document appears in 80 lists, if we make the (rough) assumption that 20% of the terms in a document appear twice;
3. each list consists, *on average*, of 400 entries;
4. if we represent document ids as 4-bytes unsigned integers, the *average* size of a list is 1,600 bytes;
5. the whole index contains $400 \times 200{,}000 = 80{,}000{,}000$ entries;
6. the index size is 320 MB (for inverted lists) plus 2,4 MB ($200{,}000 \times 12$) for the directory.

The index size is generally not negligible. It is more than 30% of the collection size on the above example, a ratio which can get even higher when additional information is added in inverted lists, as explained below.

family	$d_1/11, d_3/10, d_5/16, d_6/4$
football	$d_4/8$
jaguar	$d_1/2, d_2/1, d_3/2, d_4/3, d_5/4, d_6/8 + 13$
new	$d_1/5, d_2/5, d_5/15$
rule	$d_6/3$
us	$d_4/7, d_5/11$
world	$d_1/6$
...	

Figure 10: Partial inverted index for document set of Figure 4, with positions

Content of inverted lists

Storing the document id is sufficient for Boolean querying (e.g., to determine which document(s) contain a given term). However, applications sometimes require the position of each term in the original document. This is for instance the case when phrases can be searched (this is usually done in search engine interfaces by enclosing phrases between quotes), or when the search engine allows operators that need this position information (e.g., the NEAR operator of Altavista, that requires two terms to be close to each other). This information can easily be stored in the index, by simple addition of the positions (as integers) next to the document id in an entry. See Figure 10 for an illustration.

When a term appears several times in a document, the sorted list of its positions is stored after the document id. This additional information is likely to increase the size of the inverted file. Storing a list of integers in increasing order allows some effective compression techniques which are discussed in Section 3.3.

Note that this space overhead can be avoided at the price of a post-processing step at run-time. In that case, a phrase query is processed just as a traditional bag-of-words query, and proximity is checked once the document itself has been fetched. This method is however ineffective at large scale, due to the cost of fetching many useless documents.

Finally, for ranking purposes, a weight w_i is stored along with the document id d_i to represent the relevancy of the document with respect to the term. The value of w_i is discussed next.

Assessing document relevance

In traditional databases, the result of a query (say, a SQL query) is the set of tuples that match the query's criterion, without any specific order. This constitutes a major difference with IR queries. The set of documents (ids) that contain a term t is easily found by scanning the list associated with t. But, clearly, some of these documents are more *relevant* to the term than others. Assessing the relevance of documents during the matching process is essential in a system where query results may consist of hundreds of thousands of documents.

Relevance is measured by assigning some *weight* to the occurrence of a term in a document, depending on the relevance and informativeness of the term. A term that appears several times in a document is more relevant for indexing the document than single occurrences. If the term occurs rarely in the whole collection, this further strengthens its relevance. Conversely, a term that occurs frequently in many documents is less discriminative. Based on these principles, a common weighting scheme is *tf–idf*, or *term frequency–inverse document frequency*:

family	$d_1/11/.13$, $d_3/10/.13$, $d_5/16/.07$, $d_6/4/.08$
football	$d_4/8/.47$
jaguar	$d_1/2/.04$, $d_2/1/.04$, $d_3/2/.04$, $d_4/3/.04$, $d_5/4/.02$, $d_6/8+13/.04$
new	$d_1/5/.20$, $d_2/5/.24$, $d_5/15/.10$
rule	$d_6/3/.28$
us	$d_4/7/.30$, $d_5/11/.15$
world	$d_1/6/.47$
...	

Figure 11: Partial inverted index for document set of Figure 4, with positions and tf-idf weighting

this scheme assigns a weight to a term that is proportional to its number of occurrences in the document. It also raises the weight of terms that are present in few documents.

The *term frequency* is the number of occurrences of a term t in a document d, divided by the total number of terms in d. The division normalizes the term frequency to avoid the distorsion that would occur for large documents. In mathematical terms:

$$\mathrm{tf}(t,d) = \frac{n_{t,d}}{\sum_{t'} n_{t',d}}$$

where $n_{t',d}$ is the number of occurrences of t' in d.

The *inverse document frequency* qualifies the importance of a term t in a collection D of documents. A term which is rarely found is more characteristic of a document than another one which is very common. The idf is obtained from the division of the total number of documents by the number of documents where t occurs, as follows:

$$\mathrm{idf}(t) = \log \frac{|D|}{|\{d' \in D \,|\, n_{t,d'} > 0\}|}.$$

Finally, the mathematical definition for the weight $\mathrm{tfidf}(t,d)$ of term t in document d is the products of these two descriptors:

$$\mathrm{tfidf}(t,d) = \frac{n_{t,d}}{\sum_{t'} n_{t',d}} \cdot \log \frac{|D|}{|\{d' \in D \,|\, n_{t,d'} > 0\}|}$$

The first term raises the weight of frequently occurring terms in the given document, while the second term negatively depends of the global frequency of the term in the document set. This weighting scheme can then be added to each entry in the index, as shown on Figure 11.

Adding the weight or the position has an impact on the index size.

Example 3.2 Consider again the emails collection of Example 3.1. We add the term position and the weight to each entry, resulting in a storage overhead of 8 bytes, assuming a 4-byte representation for each component. The $80,000,000$ entries now occupy $80 \times 12 \times 10^6 =$ 960 MB, i.e., almost the size of the whole collection.

3.2 Answering Keyword Queries

Given an inverted index built as described in the previous sections, we can answer to keyword queries. This may involve some sophisticated operations if one wants to put the most significant answers on top of the result set. We begin with the simple case of Boolean queries that do not require to rank the result.

Boolean queries

If we want to retrieve all documents containing a given keyword, we just need to look up the (stemmed) keyword in the index and display the corresponding list of documents; associated weights give an indication of the relevance of each result to the keyword query. Consider now arbitrary multi-keyword Boolean queries (containing AND, OR, NOT operators), such as:

$$(\textit{jaguar} \text{ AND } \textit{new} \text{ AND NOT } \textit{family}) \text{ OR } \textit{cat}.$$

They can be answered in the same way, by retrieving the document lists from all keywords appearing in the query and applying the set operations corresponding to the Boolean operators (respectively, intersection, union, and difference for AND, OR, and AND NOT). Assigning a score to each document retrieved by the query is not completely straightforward, especially in the presence of NOT operators. For queries that only contain either the AND or the OR operator, some monotonous functions of the scores (or weights) of all matched terms can be used; a simple and effective way to compute the global score of the document is just to add all scores of matched terms. Another possibility is to use a similarity function, such as cosine (see Section 3.4) and compute the similarity between the query and the documents. Queries that give the location of terms relatively to each other (phrase queries or queries with a NEAR operator) can be answered in the same way, by retrieving from the index all matching documents with the associated positions, and checking whether the conditions imposed by the query (such as, position of keyword t should be that of keyword t' minus one for the phrase query $"t\ t'"$) are satisfied.

In most applications, it is often desirable to return only a subset of the documents that match a query, since a user cannot be expected to browse through thousands or even millions of documents. This is achieved by *ranking* the result.

Ranked queries: basic algorithm

We consider conjunctive keyword queries of the form:

$$t_1 \text{ AND } \ldots \text{ AND } t_n$$

and let k be a fixed number of documents to be retrieved (for instance, 10 or 50). We describe below two algorithms based on inverted files for top-k queries.

Recall that the inverted lists are sorted on the document id. Starting from the beginning of each list $L_{t_1} \ldots L_{t_n}$, a parallel scan is performed, looking for a tuple $[d_i^{(1)} \ldots d_i^{(n)}]$ (in other words, a document d_i matching all the terms). We denote by $s(t,d)$ the weight of t in d (e.g., tfidf) and $g(s_1,\ldots,s_n)$ the monotonous function that computes the global score of a document given the weight of each term in the document (e.g., addition). The global score $W_i = g(s(t_1,d_i),\cdots,s(t_n,d_i))$ of d_i is then computed and the pair $[d_i,W_i]$ inserted in an array.

family	$d_1/11/.13$, $d_3/10/.13$, $d_6/4/.08$, $d_5/16/.07$
new	$d_2/5/.24$, $d_1/5/.20$, $d_5/15/.10$
...	

Figure 12: Partial inverted index sorted on tf-idf weighting in descending order

When the parallel scans are finished, the array is sorted on the global score, and the k first documents constitute the output.

The algorithm is linear in the size of the inverted lists (we neglect here the cost of sorting the resulting array, which is in most cases much smaller than the inverted lists). In practice, the efficiency depends on the semantics of the query. If at least one occurrence of each term $t_1 \cdots t_n$ is required in each document of the result set, then the scan may stop as soon as one of the list is exhausted. However, in general (for a query involving the OR operator), a document can obtain a high global score even if one or several query terms are missing. A semantics that favors the global score and not the presence of each term requires a full scan of the lists.

Fagin's threshold algorithm

Fagin's threshold algorithm (TA in short) allows answering top-k queries without having to retrieve and compute the intersection of all documents where each term occurs. We now make the assumption that, in addition to an inverted index that stores lists in increasing documents identifier order, we have another inverted index sorted on the weights. The first index can be used to directly check the weight of a term in a document with binary search. The algorithm is as follows:

1. Let R, the result set, be the empty list.
2. For each $1 \leq i \leq n$:
   (a) Retrieve the document $d^{(i)}$ containing term t_i that has the next largest $s(t_i, d^{(i)})$.
   (b) Compute its global score $g_{d^{(i)}} = g(s(t_1, d^{(i)}), \cdots, s(t_n, d^{(i)}))$ by retrieving all $s(t_j, d^{(i)})$ with $j \neq i$. If the query is a conjunctive query, the score is set to 0 if some $s(t_j, d^{(i)})$ is 0.
   (c) If R contains less than k documents, add $d^{(i)}$ to R. Otherwise, if $g_{d^{(i)}}$ is greater than the minimum of the scores of documents in R, replace the document with minimum score in R with $d^{(i)}$.
3. Let $\tau = g(s(t_1, d^{(1)}), s(t_2, d^{(2)}), \ldots, s(t_n, d^{(n)}))$.
4. If R contains at least k documents, and the minimum of the score of the documents in R is greater than or equal to τ, return R.
5. Redo step 2.

We now illustrate this algorithm on our running example with the top-3 query "*new* OR *family*", using the sum of weights as our aggregation function. A first index sorts the inverted lists on the document ids (Figure 11), a second one on their weights (Figure 12 shows, respectively, the lists for *family* and *new*).

Initially, $R = \varnothing$ and $\tau = +\infty$. The query evaluation must retrieve the $k = 3$ top-ranked document in the result set. Here, $n = 2$ (the query consists of two keywords). We develop a step-by-step progress of TA.

Let $i = 1$. Document $d^{(1)}$ is the first entry in the list L_{family}, hence $d^{(1)} = d_1$. We now need $s(\text{new}, \text{d}_1)$, the weight of term *new* in d_1. Note that we cannot afford to scan the entries L_{new} since this would involve a linear cost at each step of the algorithm. This is where a binary search on another inverted list L'_{new} sorted on the document id is useful. One gets $s(\text{new}, \text{d}_1) = .20$. Finally, the global score for d_1 is $g(s(\text{family}, \text{d}_1), \text{s}(\text{new}, \text{d}_1)) = .13 + .20 = .33$.

Next, $i = 2$, and the highest scoring document for *new* is d_2. Applying the same process, one finds that the global score for d_2 is .24 (note that d_2 does not appear in L_{family}, so its weight for term *family* is 0). The algorithm quits the loop on i with $R = \langle [d_1, .33], [d_2, .24] \rangle$ and $\tau = .13 + .24 = .37$.

Since the termination condition is not fulfilled, we proceed with the loop again, taking d_3 and d_5 as, respectively, $d^{(1)}$ and $d^{(2)}$. The global score for d_3 is .13 and the global score for d_5 is $.10 + .07 = .17$. The element $[d_5, .17]$ is added to R (at the end) and the threshold τ is now $.10 + .13 = .23$. Although R now contains 3 documents, as required, the termination condition is not met because τ is larger than the minimal score in R. Therefore, it is still possible to find a document whose score is higher than .17. Assume for instance a document d at the next position in both lists, with weights .09 and .12. A last loop concludes that the next candidate is d_6, with a global score of .08 and $\tau = .08$. The algorithm halts with

$$R = \langle [d_1, .33], [d_2, .24], [d_5, .17] \rangle.$$

3.3 Large-scale Indexing with Inverted Files

Inverted files must be able to process keyword-based queries on large documents collections. As discussed above, the basic operation is a sequential scan of a list that retrieves the set of document containing a given term. This operation is linear in the size of the list, and therefore in the size (number of documents) of the collection. It is also linear in the number of terms in the query. This seems to keep inverted files from being scalable to very large collections. However, the following analysis shows that they actually constitute a quite effective structure.

Table 1 summarizes a few important properties of modern hardware. Values are given for a typical data server. The cost of a random disk access (about 5 ms) is several orders of magnitudes larger than an access to main memory (about 100 ns). Most of this cost accounts for positioning the disk head (seek time, or *disk latency*). A sequential read which avoids the seek time can retrieve as much as 100 MB/s from the disk. A typical data server holds tens of gigabytes of main memory, and stores several terabytes of data on disks. Two simple guidelines for the design of efficient large scale systems are: (i) keep data in main memory as much as possible and (ii) write and read, *sequentially* large chunks of contiguous data on disks. Note that the discussion assumes a single data server.

Performance of inverted files

Consider an inverted file structure which does not store the position of terms in the lists. An entry is a pair $[d_i, w_i]$. The document id d_i can be represented with a 4-byte unsigned integer, allowing 2^{32} = more than four billions of documents ids. The weight w_i only requires 2 bytes by setting its value to $n_{t,d}$. The tf–idf can then be computed on the fly.

Type	Size	Speed
Processor	Cache lines = a few MBs	3–4 GHz; typical processor clock rate $\approx 0{,}310^{-9}$s.
Memory	Tens of GBs	Access time $\approx 10^{-9}$s $- 10^{-8}$s (10–100 nanosec.)
Disk	Several Terabytes	Access time $\approx 5 \times 10^{-3}$s (5 millisec.); Max. disk transfer rate = 100 MB/s

Table 1: Hardware characteristics

A collection of 1 million documents can therefore be indexed with 1 million of entries that occupy 6 MBs (the size of secondary structures is considered negligible). For a collection of 1 billion documents, a 6 GBs index suffices. Clearly such an index fits in the main memory of any reasonable data server, and requires far less storage than the documents themselves. Assume for a moment that the collection contains 10,000 terms uniformly distributed in the documents. Each of the 10,000 lists contains 100,000 entries and occupies 600,000 bytes. Even if the list is on disk, it takes less than 1/100 s. to scan it and process its content. It practice, terms are *not* uniformly distributed, and this makes things even better, as explained below. These figures assume a contiguous storage of inverted lists, which is essential to ensure a high speed sequential disk-based retrieval.

If the index stores the positions of terms in the list entries, things become less pleasant. In a naive approach, a position is stored with 2 bytes (this limits the size of documents to $2^{16} = 65{,}536$ positions/terms) or 3 bytes (16,777,216 positions). This constitutes a storage overhead of at least 50 % with respect to the position-free index if each term occurs only once, and much more for terms with multiple occurrences in a same document.

What about Web-scale indexing? At the end of 2009, the size of the Web (a continuously changing value) is at least 20 billion pages (see `http://www.worldwidewebsize.com/` for an up-to-date estimate), and probably two of three times larger (note that a four-byte storage is no longer sufficient for documents ids). Several hundreds of gigabytes are necessary to index its content with the simplest possible inverted index, without storing the term positions and without replication. For Web-scale applications, such as search engines, the index is distributed over a cluster of machines. Distribution mechanisms are investigated in a dedicated chapter.

Building and updating an inverted file

Building inverted files is a complex task because of their large size, and because of the need to preserve the contiguous storage of inverted lists. We first discuss *static construction*, where the collection of documents is known in advance, then *dynamic maintenance* of the index as documents are added or removed.

The basic procedure consists in scanning the documents one by one, creating for each a sorted list of the tokens. One creates a matrix with documents ids in rows and terms in columns, which must then be inverted to obtain a row for each term. For large files, matrix inversion cannot be processed in main memory. Moreover, writing each entry on the disk as soon as it is found would result in a highly fragmented storage.

The index can be created in two passes. The first pass collects information of the frequency

of each term t. This determines the size of the inverted list for t, which can then be allocated on the disk. During the second pass, each entry can be written sequentially in the appropriate list. This results in a non-fragmented index.

The drawback of any two-pass approach is that documents must be processed twice. Another strategy relies on a preprocessing step, e.g., sorting. *Sort-based algorithms* first extract triplets $[d,t,f]$ from the collection, then sort the set of triplets on the term-docid pair $[t,d]$. Contiguous inverted lists can be created from the sorted entries.

The most commonly used algorithm for external sorting is an adaptation of the sort/merge main memory algorithm. In the first phase, sorted subfiles called "runs" are created from the data source. Assuming m blocks in main memory, each run occupies m pages and stores a sorted subset of the input. In order to create the runs, one repeats the following sequence of operations (Figure 13):

1. the m buffer pages in main memory are filled with triplets $[d,t,f]$ extracted from the documents;
2. the triplets in the m pages are sorted on $[t,d]$ with an internal-memory algorithm (usually quicksort);
3. the sorted blocks are written onto disk in a new run.

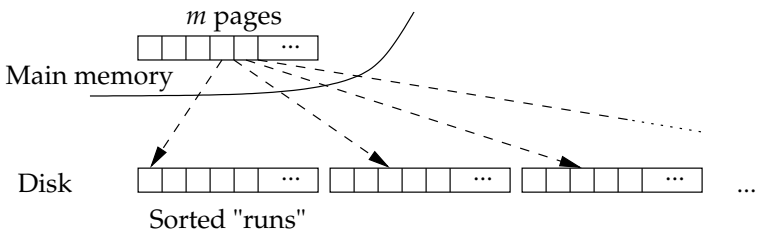


Figure 13: The creation of initial runs

Starting from the runs created in the first phase, the merge phase begins. One block in main memory is assigned to each run file: $m-1$ runs are merged in a single pass, and the last block in main memory is allocated to the output run.

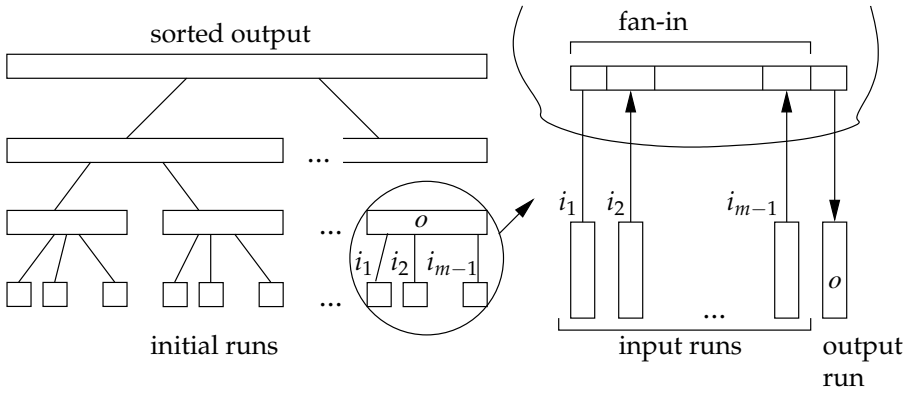


Figure 14: The merge phase

The process is illustrated in Figure 14. It can be represented by a tree, each node of the tree corresponding to a single merge operation described in the right part of the figure. One reads the first page of each input run $\{i_1, i_2, \ldots i_{m-1}\}$ in the main memory buffer (recall that the input runs are sorted). The merge then begins on the data in main memory. The record with the smallest t value is repeatedly picked in one of the $m-1$ blocks and stored in the output block (if several triplets with the same t exist, then the one with the smallest d is chosen). When all the triplets in an input block, say j, have been picked, one reads the following block in run i_j. Once the output block is full it is written in the output run. Eventually, an output run is full ($m \times (m-1)$ blocks have been written in the run), it is ready to be merged at a higher level with other runs obtained from the first-level merge operations, etc.

The merge of runs is done in linear time. Each run is read once and the size of the output is the sum of the sizes of the input runs. Therefore, at each level of the tree, one needs exactly $2n$ I/Os. If the fan-in is $m-1$, there are $O(\log_m n)$ levels in the tree and we obtain an $\Theta(n \log_m n)$ algorithm.

As a final improvement, *merge-based algorithms* avoid the sort phase by directly constructing sorted inverted lists in main memory. When the memory is full, sorted in-memory lists are flushed on the disk, just like sorted runs in the sort-merge algorithm. When all documents have been processed, the flushed lists that relate to a term t are merged and the result constitutes the final inverted list for t.

This is illustrated with Figure 15. For each document d in the collection, the *parsing* process extracts a set of terms, and a pair $[d, f]$ is inserted in the in-memory list L_i for each term t_i. Eventually the allocated memory gets full. A *flush* creates then a "run" on the disk. At the end of collection, several such runs may have been created. A final *merge* operation carries out a merge of the lists associated to each term. One obtains the final inverted index.

This algorithm enjoys several properties that make it widely used in practice. First, it avoids both a double parsing of the input documents and a costly external sort. Second, it is robust enough to behave correctly in the presence of large collections or moderate memory availability. Finally, this algorithm turns out to be useful for evolving collections, as discussed below.

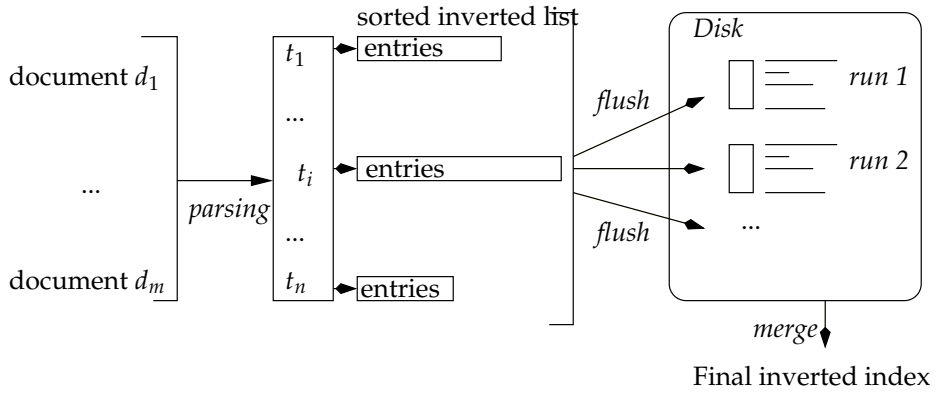


Figure 15: The merge-based algorithm

Indexing dynamic collections

When new documents are added or removed continuously (which is the standard situation is Web indexing), the inverted index must be updated to reflect the up-to-date knowledge acquired by the crawler which constantly runs in parallel. Updating the index is typically quite costly since it requires updating the document list of various terms, changing the structure of each list. Actually, applying the naive approach of directly accessing the list for each incoming document would result in awful performances.

The merge-based algorithm provides a solution to the problem. An in-memory index is maintained that holds the information related to the new documents. Searches are performed on the two indexes. When the index becomes full (that is, its size exceeds a given threshold), it can be seen as the last run of a merge-based approach, and a merge with the main index can be processed. During a merge, a copy of the old index must be kept to support the current searches operations. This doubles the space requirements. The above description disregards deletions, which can be processed thanks to a small variant (left as an exercise).

Compression of inverted lists

Compression of inverted lists is an important feature of text IR systems. It brings several advantages:

1. compressed files require less disk space;
2. the same amount of information can be read more quickly;
3. a larger part of the inverted index can be kept in main memory.

The price to pay is the need to uncompress the content of the lists when they are accessed. A basic criterion is that the total cost of reading (on disk) a compressed inverted list followed by its decompression should not exceed the cost of reading the uncompressed list. The compression would otherwise jeopardize the query evaluation process. Efficient decompression algorithms exist that take advantage of the very high speed of modern hardwares.

The standard storage size of an integer is 4 bytes (2 bytes for so-called "short" integers). A 4-byte unsigned integer can represent values in the range $[0; 2^{32} - 1] = 4,294,967,296$ (the maximal value is 65,535 for 2-bytes ints). The intuition behind the compression strategies of inverted list is that they can be seen as sequences of positive integers such that the gap between two consecutive entries in the list is typically small. Assume for instance that a term t is represented in the following documents, put in ascending order:

$$[87; 273; 365; 576; 810].$$

Note that we need at least one byte to store the first id, and 2 bytes for the other ones. This sequence of sorted documents id can be equivalently represented by the gaps between two consecutive ids:

$$[87; 186; 92; 211; 234].$$

An immediate advantage of this latter representation is that, since the gaps are much smaller that the absolutes ids, they can be represented (on this particular example) with 1-byte integers. More generally, the following facts help to greatly reduce the necessary storage:

1. relative gaps tend to be smaller than ids, and thus need *on average* a lot less space than the standard 4-byte storage;

2. for very common terms that appear in many documents, the gap often is the minimal value of 1, with high potential compression.

This representation, called *delta-coding*, is a very simple way to achieve a significant reduction of space requirements. Note, however, that although one may expect a lot of small gaps, we must be ready to face large gaps in the inverted lists of rare terms. The compression method must therefore adapt to these highly variable values, in order to choose the appropriate storage on a case-by-case basis. As mentioned above, this method must also support a very quick decompression mechanism.

We present below two efficient compression methods which respectively attempt at using the minimal number of bytes (bytewise compression) or the minimal number of bits (bitwise compression) to represent gaps. Both methods are *parameter-free* in the sense that they do not rely on any assumption on the distribution of gaps, and thus do not require the additional storage of parameters that would describe this distribution. Bitwise compression achieves a (slightly) better compression ratio than bytewise compression, at the price of a higher decompression cost.

Variable byte encoding

As the name suggests, *variable byte encoding* (VByte in short) encodes integers on a variable (but integral) number of bytes. The idea of VByte encoding is very simple. Given a positive integer value v, one tests whether d is strictly less than 128. If yes, d can be encoded on the last 7 bits of a byte, and the first bit is set to 1 to indicate that we are done. Otherwise:

1. take the reminder v' of $v/128$; encode v' as explained above in a byte b;

2. apply recursively the procedure to $v/128$, this time setting the first bit to 0; concatenate the result with b.

Let for example $v = 9$. It is encoded on one byte as 10001001 (note the first bit set to 1). Now consider a larger value, say $v = 137$.

1. the first byte encodes $v' = v \bmod 128 = 9$, thus $b = 10001001$ just as before;

2. next we encode $v/128 = 1$, in a byte $b' = 00000001$ (note the first bit set to 0).

The value 137 is therefore encoded on two bytes:

$$00000001\ 10001001.$$

Decoding is very simple: one reads the bytes $b_n, \cdots, b_2$ with a leading 0, until one finds a byte b_1 with a leading 1. The value is:

$$b_n \times 128^{n-1} + \cdots b_2 \times 128 + b_1.$$

The procedure is very efficient because it manipulates full bytes. It is also quite flexible since very large integers (there is no upper bound) can be encoded just as very small ones. The method also achieves a significant amount of compression, typically 1/4 to 1/2 of the fixed-length representation.

Variable bit encoding

We next illustrate bit-level encoding with γ-codes. Given an unsigned integer x, the starting point is the binary representation with $\lfloor \log_2 x \rfloor + 1$ bits. For instance, the binary representation of 13 is 1101, encoded with $\lfloor \log_2(13) \rfloor + 1 = 4$ bits. The binary representation is space-optimal, but since its size obviously depends on x, we need to represent this varying length as well to be able to decode the value.

Using γ-codes, the length $\lfloor \log_2 x \rfloor + 1$ is encoded in unary: a length l is represented with $l - 1$ '1' bits terminated by a '0' bit. The value 13 can therefore be represented by a pair $(1110, 1101)$, where 1110, the *length*, is in unary format, followed by the value (called the *offset*, see below) in binary.

γ-codes introduce an optimization based on the following observation: a non-null value x is of the form $2^{\lfloor \log_2(x) \rfloor} + d$. In terms of binary representation, the first term corresponds to a leading '1', followed by the binary representation of d (the *offset* of x).

Since the first term $2^{\lfloor \log_2(x) \rfloor}$ is determined by the length, which is known from the prefix of the code, we only need to store the value of d in the suffix. So, still taking 13 as an illustration, we put it in the form $2^3 + 5$. We encode the length in unary as before as 1110, and we encode the offset (5) in binary on 3 bits as 101. The γ-code for 13 is finally:

$$1110101.$$

Decoding first reads the number of leading '1' bits until a '0' is met. This gives us the length of the binary code that follows. On our example, we determine that the length is 3. The value is therefore $2^3 + decode(101) = 8 + 5 = 13$.

The length of a γ-code for a value x is $2 \times \lfloor \log_2(x) \rfloor + 1$, i.e., at most twice the minimal possible storage required for x (recall that using this minimum representation is not possible since we need to be able to find the boundaries of each value).

Better compression can only be achieved by using a *model* of the distribution of the values. It turns out that using such a model for inverted list is never done because (i) it leads to a compression/decompression cost which balances the gain of space reduction, and (ii) the maintenance of encoding becomes too complicated if the distribution changes.

Experiments show that bitwise compression achieves a better compression than bytewise (about 10% to 20% better, depending on the dataset), at the price of a more expensive pre- and post-processing of the inverted lists.

3.4 Clustering

If one wants to search on the Web some information about the jaguar animal, one is probably not interested in the other meanings of the word *jaguar*, such as the car make or the version of Mac OS X. *Clustering* can be used in such contexts to partition a set of documents (the result of the keyword query) into a set of homogeneous document collections. The result of a clustered search for *jaguar* on the Clusty[7] search engine is shown on Figure 16.

One way to achieve such a clustering is the following. Start from some document set that is to be clustered. We shall see this document set in a *document vector space* model, that is the dual of the inverted index model: documents are described by the terms that occur in them, with associated weighting, and each term is seen as a dimension of a vector space

[7] http://clusty.com/

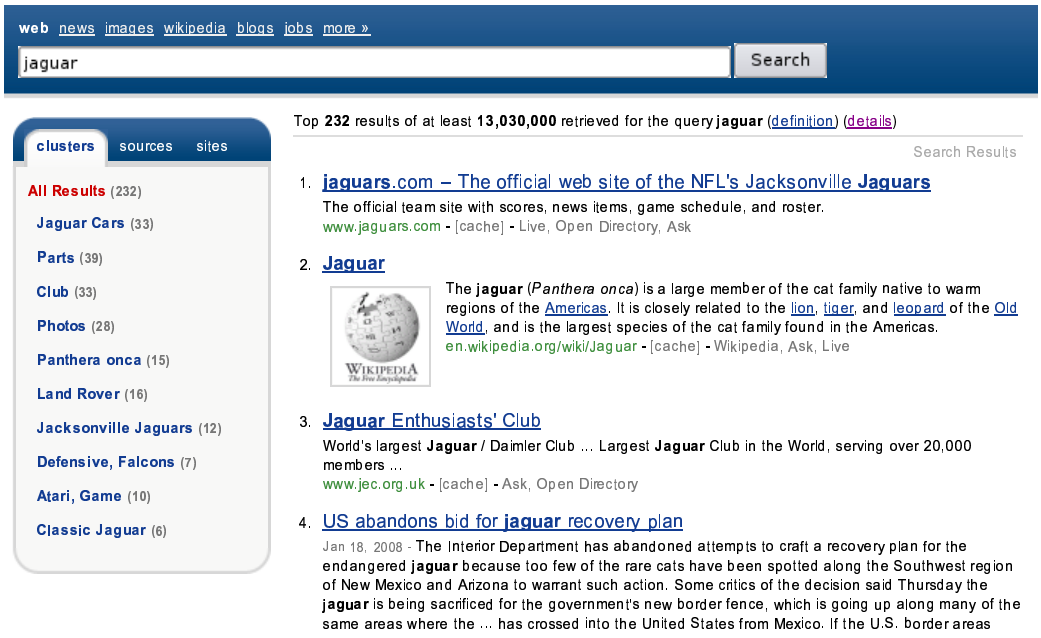


Figure 16: Example clustering from Clusty of the results of the query *jaguar*

documents live in. The coordinate of a document d in this vector space, along the dimension corresponding to t, will be the weight of t in d (say, tfidf(t,d)). We then consider the *cosine* similarity between two documents d and d', seen as vectors:

$$\cos(d,d') = \frac{d \cdot d'}{\|d\| \times \|d'\|}$$

where $d \cdot d'$ is the scalar product of d and d' and $\|d\|$ the norm of vector d. With this definition (which is a simple extension of the usual cosine function in the Euclidean plane), $\cos(d,d) = 1$ and $\cos(d,d') = 0$ if d and d' are orthogonal, that is, if they do not share any common term.

This is illustrated on Figure 17 which shows a 2-dimensional vector space built on the terms (jaguar, Mac OS). Documents are represented as normalized vectors with two coordinates representing respectively their scores for each term. The similarity is estimated by the angle between two vectors (and measured by the cosine of this angle). The figure shows that d_1 and d_3 share almost the same scores, resulting in a small angle θ and thus in a cosine close to 1.

This definition of similarity is all that we need to apply standard clustering algorithms, for instance the following simple agglomerative clustering:

1. Initially, each document forms its own cluster.
2. The similarity between two clusters is defined as the maximal similarity between elements of each cluster.
3. Find the two clusters whose mutual similarity is highest. If it is lower than a given threshold, end the clustering. Otherwise, regroup these clusters. Repeat.

Note that many other more refined algorithms for clustering exist.

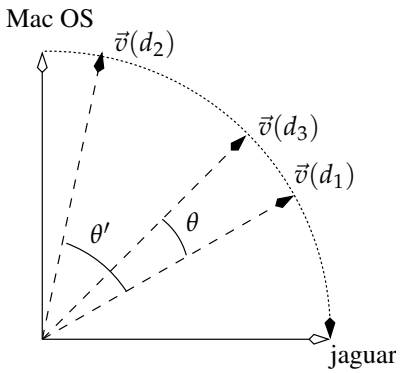


Figure 17: Illustration of similarity in the document vector space

3.5 Beyond Classical IR

HTML Web pages are not just text, but text enriched with meta-information and document-level and character-level structure. This enrichment can be used in different ways: a separate index for the title or other meta-information of a page can be built and independently queried, or the tokens of a document that are emphasized can be given a higher weight in the inverted index. For some applications, the tree structure of Web pages can be stored and queried with languages such as XPath or XQuery (cf. Chapter **??**); because most Web pages, even when they are well-formed and valid, do not really use HTML structural elements in a meaningful and consistent way, this approach is not very useful on the Web as a whole (see the part of the book devoted to the Semantic Web). Also present on the Web is multimedia content such as images or videos. They can be described and searched as text (file names, text present in the context of the content, etc.), or with more elaborate multimedia descriptors.

The material covered in this section is just a brief introduction to the field of Information Retrieval, taken as the art of efficiently and accurately searching for relevant documents in large collections. Note in particular that the techniques introduced above are by no way restricted to Web search, and apply to any collection of documents (e.g., a digital library). The next section is devoted to IR extensions that address the specificities of the Web, namely its graph structure. Modern search engines make also use of other kinds of information, especially *query logs*, the list of all queries made by users to the engine, and, in some cases, also consider their selection among the list of results returned. If a user never clicks on a link for a given query, it makes sense to decrease its relevance score.

4 Web Graph Mining

As all hyperlinked environments, the World Wide Web can be seen as a directed graph in the following way: Web pages are vertices of the graph, while hyperlinks between pages are edges. This viewpoint has led to major advances in Web search, notably with the PageRank and HITS algorithms presented in this section.

Extraction of knowledge from graphs, or *graph mining*, has been used on other graph structures than the Web, for instance on the graph of publications, where edges are the citation links between publications; *cocitation analysis* relies on the observation that two papers that are cited by about the same set of papers are similar. Other graphs susceptible to this kind of

analysis include graphs of dictionaries, or encyclopedias, or graphs of social networks.

4.1 PageRank

Though tf–idf weighting adds some relevance score to a document matching a keyword, it does not distinguish between reference documents that are highly trusted and obscure documents containing erroneous information. The idea of using the graph structure of the Web to assign some score to a document relies in the following idea or variants of it: if a document is linked by a large number of *important* documents, it is itself *important*.

PageRank, which was introduced with much success by the founders of the Google search engine, is a formalization of this idea. The *PageRank* of a page i can be defined informally as the probability $pr(i)$ that the random surfer has arrived on page i at some distant given point in the future. Consider for instance the basic graph on the left of Figure 18. A random surfer will reach node A at step i if it reaches node B, C or D at step $i-1$. Therefore:

$$pr(A) = pr(B) + pr(C) + pr(D)$$

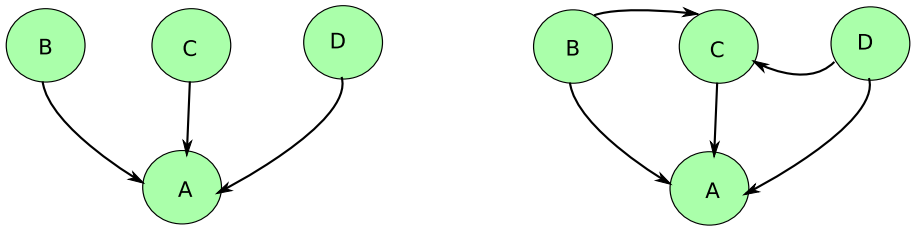


Figure 18: PageRank – Basic idea

In less simplistic cases, the surfer may have to choose among several outgoing edges. In that case one assumes that the probability is uniform. Looking at the right part of Figure 18, the probability for a surfer residing on node B (or D) to reach node C i 1/2. Hence, the probability to reach C at i given the position at $i-1$ is:

$$pr(C) = \frac{1}{2}pr(B) + \frac{1}{2}pr(D)$$

In general, let $G = (g_{ij})$ be the transition matrix of the Web graph (or a large part of it), that we assume to be normalized in the following way:

$$\begin{cases} g_{ij} = 0 & \text{if there is no link between page } i \text{ and } j; \\ g_{ij} = \frac{1}{n_i} & \text{otherwise, with } n_i \text{ the number of outgoing links of page } i. \end{cases}$$

This normalization ensures that the matrix is *stochastic* (all rows sum to 1), and that it describes a *random walk* on the pages of the Web: a *random surfer* goes from page to page, choosing with uniform probability any outgoing link.

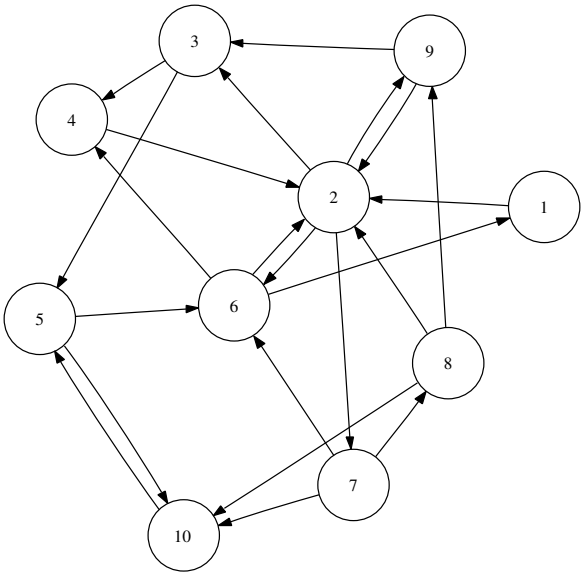


Figure 19: Illustration of PageRank: Example graph

Example 4.1 Consider the graph of Figure 19. Its normalized transition matrix is as follows:

$$G = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{4} & 0 & 0 & \frac{1}{4} & \frac{1}{4} & 0 & \frac{1}{4} & 0 \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} \\ \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{3} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{3} & 0 & \frac{1}{3} & 0 & \frac{1}{3} \\ 0 & \frac{1}{3} & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{3} & \frac{1}{3} \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Thus, the probability that a random surfer goes from page 2 to page 6 or, in other words, the probability of transition between nodes 2 and 6 in the random walk on the graph, is $g_{2,6} = \frac{1}{4}$.

Observe that if v denotes the initial position as a column vector (say, a uniform column vector would mean that the random surfer starts with uniform probability on each page), $(G^T)v$ is a column vector indicating the position after one step of the random walk. The PageRank can then be defined as the limit of this process, that is the PageRank of page i is the i-th component of the column vector:

$$\lim_{k \to +\infty} (G^T)^k v$$

if such a limit exists.

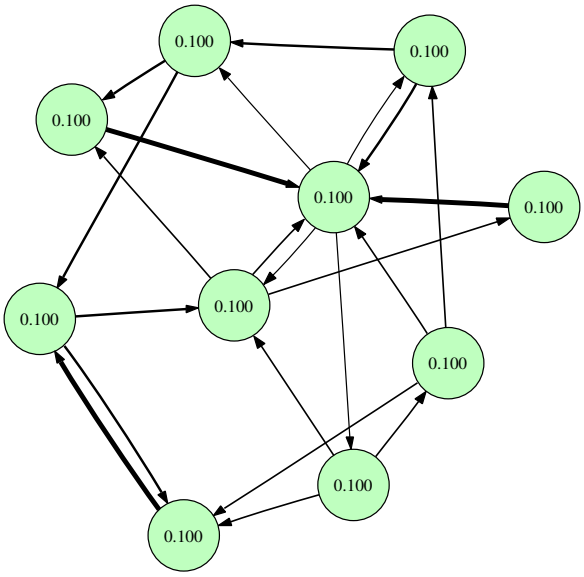


Figure 20: Illustration of PageRank: Initial uniform distribution

Example 4.2 Let us continue with the graph of Example 4.1. Let v be the uniform column vector of sum 1. This measure over the graph nodes can be displayed as in Figure 20. Consider one iteration of the PageRank computation. It amounts to multiplying the matrix of G^T by v, which gives:

$$G^T v = \left[\tfrac{1}{30} \quad \tfrac{19}{60} \quad \tfrac{3}{40} \quad \tfrac{5}{60} \quad \tfrac{3}{20} \quad \tfrac{13}{120} \quad \tfrac{1}{40} \quad \tfrac{1}{30} \quad \tfrac{7}{120} \quad \tfrac{7}{60}\right]^T$$
$$\approx \left[0.033 \quad 0.317 \quad 0.075 \quad 0.083 \quad 0.150 \quad 0.108 \quad 0.025 \quad 0.033 \quad 0.058 \quad 0.117\right]^T$$

This is the vector of probabilities of reaching a given node after one step, assuming a uniform probability for the initial node.

If we iterate this computation, we converge towards the measure displayed in Figure 21, the PageRank measure. Here, node 2 has the highest PageRank score because it is somehow more central in the graph: the probability of reaching node 2 after an arbitrarily long random walk in the graph is the greatest.

Some problems arise with this definition of the PageRank score. The convergence that is observed in the previous example is not guaranteed. The limit (if it exists) can be dependent on the initial position of the random surfer, which is kind of disappointing from a robustness point of view. Besides, some pages may have no outgoing links (they are called *sinks*) which means that the random surfer will eventually be blocked on these pages. Note that one can show that none of these problems occurs when the graph is *aperiodic* (the gcd of the length of all cycles is 1, a condition that is always verified in real-world examples) and *strongly connected* (i.e., there is a path in the graph from every node to every node). These assumptions were verified in Example 4.2, but the Web graph as a whole can definitely not be assumed to be strongly connected.

For this reason, we introduce some change in our random surfer model: at each step of the random walk, with some fixed probability d (typically around 15%; $1 - d$ is called the

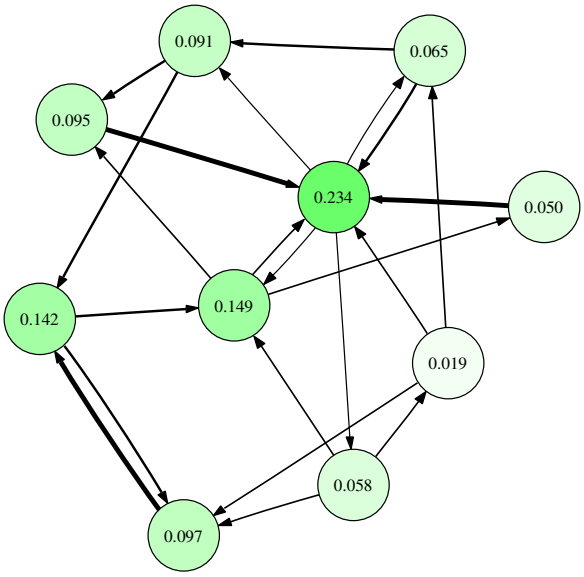


Figure 21: PageRank (damping factor of 1) for graph of Figure 19

damping factor), the surfer goes to an arbitrary uniformly chosen page of the Web; otherwise, it follows the outgoing links of the page with uniform probability as before (and if there are no outgoing links, the surfer goes in all cases to an arbitrary uniformly chosen page of the Web). With these modifications, the PageRank of page i is defined as the i-th component of the column vector:

$$\lim_{k \to +\infty} ((1-d)G^T + dU)^k v,$$

where G has been modified so that sink pages are replaced by pages with outgoing links to any page of the Web, and U is the matrix with all $\frac{1}{N}$ values where N is the number of vertices. One can show that this limit indeed exists, whenever $d > 0$ (Perron–Frobenius theorem) and is independent of the choice of the vector v, whenever $\|v\| = 1$. This formula can be used to compute the PageRank scores of all pages in an iterative way: starting from, say, the uniform column vector v, $((1-d)G^T + dU)v$ is computed by simple matrix multiplication, then $((1-d)G^T + dU)^2 v$ by another matrix multiplication, $((1-d)G^T + dU)^3 v$, etc., until convergence.

It is important to understand that PageRank assigns a *global* importance score to every page of the Web graph. This score is independent of any query. Then, PageRank can be used to improve scoring of query results in the following way: weights of documents in the inverted index are updated by a monotonous function of the previous weight and of the PageRank, say,

$$\text{weight}(t,d) = \text{tfidf}(t,d) \times \text{pr}(d),$$

thus raising the weight (and therefore their order in query results) of important documents.

Online Computation

The computation of PageRank by iterative multiplication of a vector by the dampened transition matrix requires the storage of, and efficient access to, the entire Web matrix. This

can be done on a cluster of PCs using an appropriate distributed storage technique (see the chapters devoted to distributed indexing and distributed computing). An alternative is to compute PageRank while crawling the Web, on the fly, by making use of the *random walk* interpretation of PageRank. This is what the following algorithm, known as OPIC for Online PageRank Importance Computation, does:

1. Store a global cashflow G, initially 0.
2. Store for each URL u its *cash* $C(u)$ and *history* $H(u)$.
3. Initially, each page has some initial cash c_0.
4. We crawl the Web, with some crawling strategy that accesses repeatedly a given URL.
5. When accessing URL u:
   - we set $H(u) := H(u) + C(u)$;
   - we set $C(u') := C(u)/n_u$ for all URL u' pointed to by u, with n_u is the number of outgoing links in u.
   - we set $G := G + C(u)$ and $C(u) := 0$.
6. At any given time, the PageRank of u can be approximated as $\frac{H(u)}{G}$.

It can be shown that this gives indeed an approximation of the PageRank value, whatever the crawling strategy is, as long as a URL is repeatedly accessed. A reasonable strategy is for instance a *greedy* one, to crawl the URL with the largest amount of cash at each step. Finally, similarly to the use of the damping factor in the iterative computation of PageRank, convergence can be ensured by adding a virtual node u that is pointed to by each URL and that points to all URLs. In addition to making the storage of the Web matrix unnecessary, such an online algorithm also is more adapted to the case of a changing Web, when pages are added continuously. However, since computation power tends to be cheap and the storage of the content of Web pages already necessitates appropriate storage, search engines currently stick with the classical iterative computation of PageRank.

4.2 HITS

The *HITS* algorithm (Hypertext Induced Topic Selection) is another approach proposed by Kleinberg. The main idea is to distinguish two kinds of Web pages: *hubs* and *authorities*. Hubs are pages that point to good authorities, whereas authorities are pages that are pointed to by good hubs. As with PageRank, we use again a mutually recursive definition that will lead to an iterative fixpoint computation. For example, in the domain of Web pages about automobiles, good hubs will probably be portals linking to the main Web page of car makers, that will be good authorities.

More formally, let G' be the transition matrix (this time, not normalized, i.e., with Boolean 0 and 1 entries) of a graph (say, a subgraph of the Web graph). We consider the following iterative process, where a and h are column vectors, initially of norm 1:

$$\begin{cases} a := \frac{1}{\|G'^T h\|} G'^T h \\ h := \frac{1}{\|G' a\|} G' a \end{cases}$$

If some basic technical conditions on G' hold, we can show that this iterative process converges to column vectors a and h which represent respectively the authority and hub scores of vertices of the graph. Kleinberg proposes then the following way of using authority scores to order query results from the Web:

1. Retrieve the set D of Web pages matching a keyword query.
2. Retrieve the set D^* of Web pages obtained from D by adding all linked pages, as well as all pages linking to pages of D.
3. Build from D^* the corresponding subgraph G' of the Web graph.
4. Compute iteratively hubs and authority scores.
5. Sort documents from D by authority scores.

The process is here very different from PageRank, as authority scores are computed for each request (on a subgraph kind of centered on the original query). For this reason, and although HITS give interesting results, it is not as efficient as PageRank, for which all scores can be precomputed and top-k optimization is possible.

4.3 Spamdexing

The term *spamdexing* describes all fraudulent techniques that are used by unscrupulous webmasters to artificially raise the visibility of their website to users of search engines. As with virus and antivirus, or spam and spam fighting, spamdexing and the fight against it is an unceasing series of techniques implemented by spamdexers, closely followed by countertechniques deployed by search engines. The motivation of spamdexers is to bring users to their webpages so as to generate revenue from pay-per-view or pay-per-use content (especially in the industries of online gambling and online pornography), or from advertising.

A first set of techniques consists in lying about the document by adding to a page keywords that are unrelated to its content; this may be done either as text present in the page but invisible to users through the use of CSS, JavaScript or HTML presentational elements, or in the meta-information about the page that can be provided in the `<meta name="description">` or `<meta name="keywords">` tags in the header. As a result, current search engines tend not to give a strong importance to this kind of meta-information, or even to ignore them altogether. Furthermore, they implement automatic methods to find text hidden to a regular user and ignore it. In some cases, this is even used as a reason to lower the importance of the Web page.

PageRank and similar techniques are subject to *link farm* attacks, where a huge number of hosts on the Internet are used for the sole purpose of referencing each other, without any content in themselves, to raise the importance of a given website or set of websites. Countermeasures by search engines include detection of websites with empty or duplicate content, and the use of heuristics to discover subgraphs that look like link farms. A collection of algorithms have also been proposed to assign importance scores to Web pages in a way that is more robust to these kind of attacks. TrustRank, for instance, is defined using the same kind of iterative computation as PageRank, except that random jumps toward uniformly selected Web pages are replaced by random jumps to a small subset of "safe" seed pages, to prevent

being trapped in link farms; this has the downside of artificially raising the importance of the set of seed pages, and thus of biasing the computation.

An assumption made by the graph mining techniques described earlier is that the addition of a link to a Web page is a form of approval of the content of the linked page, thus raising its importance. While this is mostly true when Web pages are written by a single individual or entity, this does not hold with user-editable content, such as wikis, guestbooks, blogs with comment systems, and so on. Spamdexers have an incentive to use these platforms to add links to their website. They can also exploit security faults in Web applications to achieve the same effect. While most webmasters take care to control whatever is added to their websites and to remove spam content, this cannot be assumed on a global level. A partial solution to this is the possibility of adding a `rel="nofollow"` attribute to all `<a>` links that have not been validated or are not approved by the webmaster (some content management systems and blog platforms automatically add this attribute to any link inside content provided by users). Most current-day Web spiders recognize this attribute and ignore this link.

4.4 Discovering Communities on the Web

The graph structure of the Web can be used beyond the computation of PageRank-like importance scores. Another important graph mining technique that can be used on the Web is graph clustering: using the structure of the graph to delimitate homogeneous sets of Web pages, e.g., communities of Web sites about the same topic. The assumption is that closely connected set of pages in a Web page will share some common semantic characteristic.

Various algorithms for graph clustering have been studied, in this context and others (to isolate local networks of interest on the Internet, to find communities of people in social networks, etc.). Let us just present briefly one of them, which has the advantage of being simple to understand. Given a graph, the purpose is to separate it, hierarchically, into smaller and smaller, and more and more homogeneous, communities of nodes. We introduce the notion of *betweenness* of an edge, as the number of shortest paths between any two nodes of the graph that use this edge. The main idea of the algorithm is that edges with high betweenness tend to be connecting distinct communities. The algorithm proceeds by computing the betweenness of all edges (i.e., computing all shortest paths between pairs of nodes) removing the edge with highest betweenness, and then iterating the whole procedure, recomputing all betweenness values at each step. The algorithm ends when enough components of the graph have been separated, or when the highest betweenness is less than a given threshold. This is not a particularly efficient technique (the number of required operations is cubic in the number of nodes). Cubic-time algorithms are not appropriate for the whole graph of the Web, but such an algorithm might be used to cluster subgraphs.

Other graph clustering methods, usually more efficient, rely on another principle, namely that of minimum cut in a transport network. This is a classical algorithmic problem: given a directed graph with weights (positive numbers) on the edges, one wants to find the set of edges of minimum weight to remove from the graph to separate two given nodes. This can be used to cluster the Web: in order to find the community to which a given Web page belongs, just compute the minimum cut (with some appropriate weighting of the edges) that separate this page from the rest of the Web, represented as a virtual node all pages point to. Such an approach differs from the previous one in that it is more local: we are not looking for a global partition into clusters, but for the cluster of a given Web page.

5 Hot Topics in Web Search

We conclude this chapter with some research topics related to the search of information on the Web that are particularly active at the moment of writing.

Web 2.0

Web 2.0 is a *buzzword* that has appeared recently to refer to recent changes in the Web, notably:

- Web applications with rich dynamic interfaces, especially with the help of AJAX technologies (AJAX stands for *Asynchronous JavaScript And XML* and is a way for a browser to exchange data with a Web server without requiring a reload of a Web page); it is exemplified by GMail[8] or Google Suggest[9];
- user-editable content, collaborative work and social networks, e.g., in blogs, wikis such as Wikipedia[10], and social network websites like MySpace[11] and Facebook[12];
- aggregation of content from multiple sources (e.g., from RSS feeds) and personalization, that is proposed for instance by Netvibes[13] or YAHOO! PIPES (see Chapter **??**).

Though Web 2.0 is more used in marketing contexts than in the research community, some interesting research problems are related to these technologies, especially in the application of graph mining techniques similar to those employed on the graph of the Web to the graph of social network websites, and in the works about *mashups* for aggregating content from multiple sources on the Web.

Deep Web

The *deep Web* (also known as *hidden Web* or *invisible Web*) is the part of Web content that lies in online databases, typically queried through HTML forms, and not usually accessible by following hyperlinks. As classical crawlers only follow these hyperlinks, they do not index the content that is behind forms. There are hundreds of thousands of such deep Web services, some of which with very high-quality information: all *Yellow pages* directories, information from the US *Census bureau*, weather or geolocation services, etc.

There are two approaches to the indexing of the deep Web. A first possibility is an *extensional* approach, where content from the deep Web is generated by submitting data into forms, and the resulting Web pages are stored in an index, as with classical Web content. A more ambitious *intensional* approach is to try to understand the structure and semantics of a service of the deep Web, and to store this semantic description in an index. A semantic query from a user would then be dispatched to all relevant services, and the information retrieved from them. Whatever the method, searching the deep Web requires first discovering all relevant forms, and some analysis to understand what data to submit to a form. In the *intensional* approach, deep Web search is also needed to extract information from the pages resulting from the submission of a form, which is the topic of the next section.

[8] `http://mail.google.com/`
[9] `http://www.google.com/webhp?complete=1`
[10] `http://www.wikipedia.org/`
[11] `http://www.myspace.com`
[12] `http://www.facebook.com/`
[13] `http://www.netvibes.com/`

Showing results 1 through 25 (of 94 total) for all:xml

1. cs.LO/0601085 [abs, ps, pdf, other] :
Title: **A Formal Foundation for ODRL**
Authors: Riccardo Pucella, Vicky Weissman
Comments: 30 pgs, preliminary version presented at WITS-04 (Workshop on Issues in the Theory of Security), 2004
Subj-class: Logic in Computer Science; Cryptography and Security
ACM-class: H.2.7; K.4.4

2. astro-ph/0512493 [abs, pdf] :
Title: **VOFilter, Bridging Virtual Observatory and Industrial Office Applications**
Authors: Chen-zhou Cui (1), Markus Dolensky (2), Peter Quinn (2), Yong-heng Zhao (1), Francoise Genova (3) ((1)NAO China, (2) ESO, (3) CDS)
Comments: Accepted for publication in ChJAA (9 pages, 2 figures, 185KB)

3. cs.DS/0512061 [abs, ps, pdf, other] :
Title: **Matching Subsequences in Trees**
Authors: Philip Bille, Inge Li Goertz
Subj-class: Data Structures and Algorithms

4. cs.IR/0510025 [abs, ps, pdf, other] :
Title: **Practical Semantic Analysis of Web Sites and Documents**
Authors: Thierry Despeyroux (INRIA Rocquencourt / INRIA Sophia Antipolis)
Subj-class: Information Retrieval

5. cs.CR/0510013 [abs, pdf] :
Title: **Safe Data Sharing and Data Dissemination on Smart Devices**
Authors: Luc Bouganim (INRIA Rocquencourt), Cosmin Cremarenco (INRIA Rocquencourt), François Dang Ngoc (INRIA Rocquencourt, PRISM - UVSQ), Nicolas Dieu (INRIA Rocquencourt), Philippe Pucheral (INRIA Rocquencourt, PRISM - UVSQ)
Subj-class: Cryptography and Security; Databases

Figure 22: Example pages resulting from the submission of a HTML form

Information Extraction

Classical search engines do not try to extract information from the content of Web pages, they only store and index them as they are. This means that the only possible kind of queries that can be asked is keyword queries, and provided results are complete Web pages. The purpose of Web *information extraction* is to provide means to extract structured data and information from Web pages, so as to be able to answer more complex queries. For instance, an information extractor could extract phone numbers from Web pages, as well as the name of their owner, and provide an automatically built directory service. Information extraction is facilitated by very structured Web pages, such as those that are dynamically generated on response to the submission of an HTML form (e.g., Figure 22); a *wrapper* for this kind of dynamic site can be generated, in order to abstract away its interface.

Most research works in information extraction are in a supervised or semi-supervised context, where humans pre-annotate Web pages whose content is to be extracted, or where human give some feedback on automatic wrapper construction. Unsupervised approaches rely either on the detection of linguistic or sentence-level patterns that express some concept or relation between concepts (e.g., addresses usually follow some kind of fixed format that can be discovered in corpus; textual patterns like *was born in* `year` can be found to automatically extract birth dates of individuals), or the detection of structural patterns in the Web page (repetitive structures such as tables or lists, for instance).

6 Further Reading

We provide references on the material found in this chapter. More information, as well as in-depth coverage of some other parts of this chapter, can be found in [Cha03].

Web Standards

HTML 4.01 [W3C99] is described by a recommendation of the World Wide Web Consortium (or W3C), an organism that regroups academics and industrials for the development of

standards about the World Wide Web, as is XHTML 1.0 [W3C02]. The W3C is working at the time of writing on the successor to both languages, HTML5 [W3C10]. The DNS and HTTP protocols, which are Internet protocols, are published by the Internet Engineering Task Force (IETF) and can be found, respectively, in [IET99a] and [IET99b].

The standard for robot exclusion and sitemaps have both unofficial specifications, not supported by any normalization organization. The former is described in [Kos94]. Sitemaps are an initiative of Google, that has been embraced by other search engines. The specification of sitemaps is available in [sit08].

Web Parsing and Indexing

Computation of the edit distance between two text documents is a classical problem, and a dynamic algorithm for solving it can be found in textbooks on algorithmics, such as [CLR90]. The Jaccard similarity coefficient has been introduced by the botanist Paul Jaccard for comparing floral populations across areas [Jac01]. Hashing shingles of a document to build a compact sketch that can be used to efficiently detect near-duplicates has been proposed in [BGMZ97].

The stemming technique described in the text is from Porter [Por80]. Soundex [US 07] is a widely used loose phonetic stemming for English.

[ZM06] is a recent and accurate survey on inverted files. From the same authors, the book [WMB99] provides a larger (but less up-to-date) coverage of the field, including a detailed presentation of the most useful text and index compression techniques. The recent book [MRS08] covers information retrieval techniques, and supplies on-line material at `http://nlp.stanford.edu/IR-book/information-retrieval-book.html`. The byte-level compression technique has been developed and experimented in [SWYZ02, AM05, AM06]. Experimental results show that byte-level compression is twice as fast as bit-level compression. The compression loss with respect to the latter approach is reported to be approximately 30%. Integer compression methods have been studied for a long time in computer science. The γ code presented here is from [Eli75].

Efficient external memory algorithms to construct index structures that cannot fit in memory is one of the core issues in databases and information retrieval. See for instance [Vit01] for an in-depth survey. The external sort/merge is a standard algorithm implemented in all DBMS (and used, for instance, during non-indexed joins or grouping operations). [HZ03] cover in detail the one-pass construction algorithm outlined in the present chapter.

Fagin's threshold algorithm (TA) that computes the top-k result of a ranked query is from [FLN03]. It improves an earlier algorithm proposed by Fagin in [Fag99].

Graph mining

PageRank was introduced in [BP98] by the founders of the Google search engine, Sergey Brin and Lawrence Page. The OPIC algorithm is from [APC03]. HITS has been proposed by Kleinberg in [Kle99]. TrustRank has been presented by researchers from Stanford University and Yahoo! in [GGMP04]. Interestingly, Google registered *TrustRank* as a trademark in 2005, suggesting they might adopt the technology, but the trademark was abandoned in 2008.

The graph clustering algorithm relying on betweenness is the work of two physicists, published in [NG04]. The idea of using minimum cuts on the Web graph has been proposed in [FLG00, FLGC02]. A large number of graph clustering techniques exist, some of them are reviewed in [Sch07]. One particular technique of interest, particularly interesting because

of its efficiency and the availability of an optimized implementation, is MCL, the Markov CLustering algorithm [vD00].

The Deep Web and Information Extraction

The first study about the amount of content is [Bri00]. Other works [CHL+04] have confirmed the fact that an impressive amount of content is hidden to current-day search engines. Google believes in an extensional approach to crawling the deep Web, see [MHC+06]. Research works that go towards intensional indexing include [CHZ05, SMM+08].

A survey of existing information extraction techniques on the Web can be found in [CKGS06]. Unsupervised techniques, which are probably the only relevant at the scale of the whole world, include RoadRunner [CMM01], ExAlg [AGM03], and the various works derived from the MDR system [LGZ04, ZL05].

References

[AGM03] Arvind Arasu and Hector Garcia-Molina. Extracting structured data from Web pages. In *Proc. ACM Intl. Conf. on the Management of Data (SIGMOD)*, pages 337–348, San Diego, USA, June 2003.

[AM05] Vo Ngoc Anh and Alistair Moffat. Inverted Index Compression Using Word-Aligned Binary Codes. *Inf. Retrieval*, 8(1):151–166, 2005.

[AM06] Vo Ngoc Anh and Alistair Moffat. Improved Word-Aligned Binary Compression for Text Indexing. *IEEE Transactions on Knowledge and Data Engineering*, 18(6):857–861, 2006.

[APC03] Serge Abiteboul, Mihai Preda, and Grégory Cobena. Adaptive on-line page importance computation. In *Proc. Intl. World Wide Web Conference (WWW)*, 2003.

[BGMZ97] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the Web. *Computer Networks*, 29(8-13):1157–1166, 1997.

[BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks*, 30(1–7):107–117, April 1998.

[Bri00] BrightPlanet. The Deep Web: Surfacing Hidden Value. White Paper, July 2000.

[Cha03] Soumen Chakrabarti. *Mining the Web: Discovering Knowledge from Hypertext Data*. Morgan Kaufmann, San Fransisco, USA, 2003.

[CHL+04] Kevin Chen-Chuan Chang, Bin He, Chengkai Li, Mitesh Patel, and Zhen Zhang. Structured Databases on the Web: Observations and Implications. *SIGMOD Record*, 33(3):61–70, 2004.

[CHZ05] Kevin Chen-Chuan Chang, Bin He, and Zhen Zhang. Toward Large Scale Integration: Building a MetaQuerier over Databases on the Web. In *Proc. Intl. Conference on Innovative Data Systems Research (CIDR)*, Asilomar, USA, January 2005.

[CKGS06] Chia-Hui Chang, Mohammed Kayed, Mohem Ramzy Girgis, and Khaled F. Shaalan. A survey of Web information extraction systems. *IEEE Transactions on Knowledge and Data Engineering*, 18(10):1411–1428, October 2006.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[CMM01] Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. RoadRunner: To-

wards Automatic Data Extraction from Large Web Sites. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, 2001.

[Eli75] Peter Elias. Universal code word sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[Fag99] Ronald Fagin. Combining fuzzy information from multiple systems. *Journal of Computer and System Sciences*, 58:83–99, 1999. Abstract published in PODS'96.

[FLG00] Gary Flake, Steve Lawrence, and C. Lee Giles. Efficient Identification of Web Communities. In *Proc. ACM Intl. Conf. on Knowledge and Data Discovery (SIGKDD)*, pages 150–160, 2000.

[FLGC02] Gary William Flake, Steve Lawrence, C. Lee Giles, and Frans Coetzee. Self-Organization of the Web and Identification of Communities. *IEEE Computer*, 35(3):66–71, 2002.

[FLN03] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66:614–656, 2003. Abstract published in PODS'2001.

[GGMP04] Zoltán Gyöngyi, Hector Garcia-Molina, and Jan O. Pedersen. Combating Web Spam with TrustRank. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, 2004.

[HZ03] Steffen Heinz and Justin Zobel. Efficient single-pass index construction for text databases. *Journal of the American Society for Information Science and Technology (JASIST)*, 54(8):713–729, 2003.

[IET99a] IETF. Request For Comments 1034. Domain names—concepts and facilities. `http://www.ietf.org/rfc/rfc1034.txt`, June 1999.

[IET99b] IETF. Request For Comments 2616. Hypertext transfer protocol—HTTP/1.1. `http://www.ietf.org/rfc/rfc2616.txt`, June 1999.

[Jac01] Paul Jaccard. Étude comparative de la distribution florale dans une portion des Alpes et du Jura. *Bulletin de la Société Vaudoise des Sciences Naturelles*, 37, 1901.

[Kle99] Jon M. Kleinberg. Authoritative Sources in a Hyperlinked Environment. *Journal of the ACM*, 46(5):604–632, 1999.

[Kos94] Martijn Koster. A standard for robot exclusion. `http://www.robotstxt.org/orig.html`, June 1994.

[LGZ04] Bing Liu, Robert L. Grossman, and Yanhong Zhai. Mining Web Pages for Data Records. *IEEE Intelligent Systems*, 19(6):49–55, 2004.

[MHC+06] Jayant Madhavan, Alon Y. Halevy, Shirley Cohen, Xin Dong, Shawn R. Jeffery, David Ko, and Cong Yu. Structured Data Meets the Web: A Few Observations. *IEEE Data Engineering Bulletin*, 29(4):19–26, December 2006.

[MRS08] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008. Online version at *http://informationretrieval.org/*.

[NG04] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(2), 2004.

[Por80] Martin F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, July 1980.

[Sch07] Satu Elisa Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27–64, 2007.

[sit08] sitemaps.org. Sitemaps XML format. `http://www.sitemaps.org/protocol.php`, February 2008.

[SMM+08] Pierre Senellart, Avin Mittal, Daniel Muschick, Rémi Gilleron, and Marc Tommasi. Automatic Wrapper Induction from Hidden-Web Sources with Domain Knowl-

edge. In *Proc. Intl. Workshop on Web Information and Data Management (WIDM)*, pages 9–16, Napa, USA, October 2008.

[SWYZ02] Falk Scholer, Hugh E. Williams, John Yiannis, and Justin Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. ACM Symp. on Information Retrieval*, pages 222–229, 2002.

[US 07] US National Archives and Records Administration. The Soundex indexing system. `http://www.archives.gov/genealogy/census/soundex.html`, May 2007.

[vD00] Stijn Marinus van Dongen. *Graph Clustering by Flow Simulation*. PhD thesis, University of Utrecht, May 2000.

[Vit01] Jeffrey Scott Vitter. External memory algorithms and data structures. *ACM Computing Surveys*, 33(2):209–271, 2001.

[W3C99] W3C. HTML 4.01 specification, September 1999. `http://www.w3.org/TR/REC-html40/`.

[W3C02] W3C. XHTML 1.0: The extensible hypertext markup language (second edition). `http://www.w3.org/TR/xhtml1/`, August 2002.

[W3C10] W3C. HTML5, 2010. Working draft available at `http://dev.w3.org/html5/spec/Overview.html`.

[WMB99] I. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan-Kaufmann, 1999.

[ZL05] Yanhong Zhai and Bing Liu. Web data extraction based on partial tree alignment. In *Proc. Intl. World Wide Web Conference (WWW)*, 2005.

[ZM06] J. Zobel and A. Moffat. Inverted Files for Text Search Engines. *ACM Computing Surveys*, 38(2), 2006.