Web Search Engine Architecture

Overview

Chapter 1,2,3.1-3.4

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Fall 2015 ; distribution outside this group of students is NOT allowed.*

We introduce in this subject **the architecture of a search engine**. It consists of its software components, the interfaces provided by them, and the relationships between any two of them. (An extra level of detail could include the data structures supported.) In this subject, we use the example of an early centralized architecture as reflected by the Altavista search engine of the mid 90s to provide a high-level description of the major components of such a system.
We then (Subject 3) give an example of the Google search engine architecture as it was originally developed and used back in 1997 and 1998. There are more components involved in the Google architecture, but a high-level abstraction of that architecture (minus the ranking engine perhaps) is not much different from Altavista's.

The first search engines such as Excite (1994), InfoSeek (1994), Altavista(1995) employed primarily Information Retrieval principles and techniques and were search engines that were evaluating the similarity of a query q relative to the web document d_j of a corpus of web-documents retrieved from the Web. The query was being treated as a "small document" consisting of the index-terms input by a customer/user and the **similarity** $s(q, d_j)$ was established based on IR principles/measures. This determined a **"rank"** of d_j for query q. However Yahoo! (1995), Google (1998), Teoma (2000), Bing (2006) differ from those early approaches in the sense that link information (e.g. the Web structure and link information of Web-based documents) is used to determine the **rank of a web document** $r(d_j)$ in addition to other monetary criteria (say, for the case of Google, advertising revenue by paying customers). Thus the $R(q, d_j)$ the **overall rank** of a document d_j relative to a query q becomes a weighed sum of $s(q, d_j)$ and $r(d_j)$ plus other additional factors. Somewhere in between these two groups of search engines we have Lycos (1994) a search engine that only used content and structure information for ranking results.

1.1 Criteria. Any search engine architecture must satisfy two major criteria.

- **1.1.1 Effectiveness (Quality)** that will satisfy the relevance criterion.
- **1.1.2 Efficiency (Speed)** that will satisfy response times and throughput requirements i.e. process as many queries as quickly as possible. Related to it is the notion of scalability.

Other criteria or goals can also be satisfied that relate to critical features of a search engine such as those described in the previous subject.

2.1 (Web) document and Corpus. (Web) document is a single unit of information in digital form. (From now on we drop the web prefix altogether.) All the documents of a collections are know collectively as the **Corpus**.

2.2 A document: tokens and metadata. A document is a collection (in fact, a sequence) of **tokens** and also of **metadata**. (The term metadata refers to data that describe "the" data.) By tokens we mean primarily **words** but also **numbers**, **dates**, **phrases**, **acronyms** etc.

2.3 Keywords: tokens or words of importance. Not all tokens in a document are of key importance. The key tokens, which are usually words, are denoted as **keywords**. These are sometimes important enough to summarize the contents of a document.

2.4 Index-terms from keywords. Through text transformations that include case-folding, stemming, stopword elimination, synonym and noun-group identification, tokens and in general keywords can be reduced into a smaller number of **index terms**. These are the terms that are maintained by a web-search engine. When a user types an arbitrary word in a query box of a (web) search engine they are appropriately converted into corresponding index-terms.

Keywords or index-terms can be automatically extracted from the document or be user-specified (e.g. an expert decides them). Sometimes we use the term `index-term` differently from the term `keyword` but some other time interchangeably. For example `algorithmic` and `algorithms` might be words (aka tokens) of a text and both become keywords of the text. If however we perform an operation called stemming on both of these words and eliminate the plural form of the latter and eliminate certain suffixes from the former, the common index-term extracted from both, indexed and subsequently used will be `algorithm`, a word that might not even appear in the document at all. Thus multiple tokens, words, or keywords might give rise to the same index-term that is neither a token, word let alone a keyword of the original text.

2.5 Logical view, full-text logical view, index-term logical view of a document. The keywords form the **logical view** of a given document. A **full-text** logical view of a document is one in which all the words (or tokens) of a document become keywords. Because of indexing engine space limitations not all keywords can be accommodated as individual index-terms and thus an **index-term** logical view is more preferable and might be used instead.

Note: Sequences vs Sets. In a sequence order matters, in a set order does not matter. Thus the two sequences are different as $\langle 1, 2, 3\rangle \neq \langle 2, 1, 3\rangle$, but the two sets $\{1, 2, 3\} = \{2, 1, 3\}$ are the same as they contain the same elements.

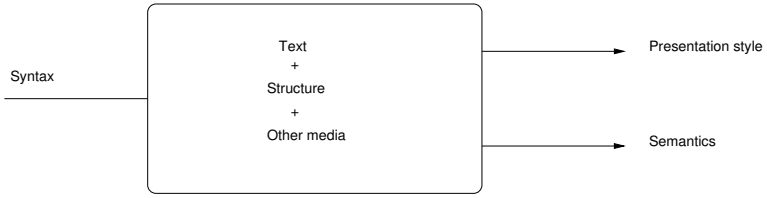


Figure 1: Document Processing

2.6 Document: Syntax, Structure, Presentation Style and Semantics. Every document and this includes web-based documents has some **syntax**, **structure**, **presentation style** and **semantics**. A syntax and structure might have been provided by the application or person who created it, the semantics by the author of the document who might also provide a style for its presentation.

More often **the syntax** of a document expresses its structure, presentation style and also semantics. Such a syntax can be explicit, as it is in HTML/XML documents or implicit (e.g. English language). A document might also have a **presentation style** implied by its syntax and structure. It might also contain information about itself in the form of say, metadata information. A document has or might also have **semantics** which is the meaning assigned to a specific syntax structure. For example in some languages = is an equality testing (i.e. relational) operator, whereas other languages use == for this and use = as an assignment operator; in some languages the assignment operator is :=. The semantics of a document are also associated with its usage. For example Postscript document commands are designed for drawing. The semantics of human language are not fully understandable by computers; simplified programming languages such as SGML, HTML, or XML are used instead in documents.

Documents come in a variety of formats that might be classified as follows: (a) text-based documents (e.g. plain text ASCII or UNICODE, HTML, XML, TeX,LaTeX, RTF, etc), (b) encoded documents (e.g. MIME-encoded which is an acronym for Multipurpose Internet Mail Exchange), (c) proprietary word-processing formatted documents such as Microsoft Word, Framemaker, etc, (d) documents intended for displaying or printing such as Adobe Acrobat PDF, and Adobe Postscript, (e) documents intended for other purposes that also store text (e.g. Microsoft Excel and Powerpoint), and (f) compressed document formats.

2.7 Markup languages. A **markup** language is a language for annotating a text in a syntactically distinguishable way; both the text and the annotations reside in the document. These markup declarations for example can be defined in a DTD (Document Type Declaration) schema for SGML-family markup languages such as SGML,HTML, XML. A DTD is similar to an XML schema. Thus, a parser can use these declarations to separate the annotations from the text in the document.

2.8 Markup languages and filenames (and their extensions). HTML files can be easily recognized by their file names which have an `.html` or `.htm` suffix. The same applies to XML `.xml` files and potentially to RTF or SGML files that might use `.rtf` and `.sgml` suffixed file-names. Things can get a little complicated for LaTeX/TeX documents that can share a `.tex` filename. One might have to parse the document itself to determine whether it is a LaTeX or a TeX document. Other text files in which English, or another language is used (and it is in none of the previous formats) might become more difficult to detect. Files with `.txt` suffixes are sometimes but not always used. Programming-language files might have a variety of formats and suffixes such as `.c , .cc, .cpp, .java , .p`. All this information needs to be taken into consideration when the Search Engine Architecture's components are designed and implemented.

2.9 Multimedia files. These same components also need to determine how to handle multimedia files (images, video, voice files) or binary-format files. Non-text multimedia data are usually ignored by search engines in the context of web-searching.

2.10 Filename and file contents: Encoding. Even if a file has been identified to be in a known format (say .txt), other problems might arise. For example, a text file might be encoded in a variety of characters sets such as ASCII or UNICODE. In the former a character is encoded in a single byte; in the latter it uses two or more bytes. Some component of the architecture needs to determine what it is. If for example the file is in ASCII, it might describe an English text or a text written in some other foreign language. ASCII characters are 7-bit, yet they are stored in an 8-bit byte. The upper positions of the character set (128 to 255) can be used to encode another language. A variety of **code pages** can result: all of them are ASCII, all of them can use English characters which are traditionally stored in the lower positions of the character set (0 to 127) but also allow support for a foreign language. An ASCII character or special characters that can not be typed directly using the keys of a keyboard can be inserted into a variety of ways in a file that uses a special syntax such as HTML. For example `&#32;` is the way to print a SPACE in HTML. In this case, 32 is the ASCII code (also the UNICODE) for the space character. Different file formats allow us to insert characters in direct (keyboard punching) or indirect ways (e.g. the alternative shown for space).

2.11 Examples of Markup languages.

- TeX/LaTeX are considered markup language(s) for typesetting scientific text (i.e. papers). The marks are called tags and initial `\begin{document}` and ending `\end{document}` tags surround the marked text,
- SGML (Standard Generalized Markup Language, ISO 8879) developed by Goldfarb,
- HTML (Hyper Text Markup Language, current version between 4.01 and 5) is an instance of SGML,
- CSS (Cascade Style Sheets) were added to HTML in 1997, since the latter does not describe the presentation style of a document. Some limitations of HTML include:
  - No validation of data/text
  - No support for nested structures to represent DataBase schemas.
  - No user-specified tags.
- XML (Extensible Markup Language) is a simplified subset of SGML,
  - XML allows users to do things that now require Javascript,
  - XML is restrictive (ending tags MUST exist),
  - `BR, IMG` have `/>` at the end,
  - distinguishes between upper and lower case,
  - XSL is the counterpart of CSS,
  - metadata info in RDF format (Resource Description Format),

2.12 HTML tags. Here are some useful HTML tags.

```
<html>                </html>                          : HTML document
<head>    <title>     </title>   </head>               : HTML head/title
<h1>      </h1>   to  <h6>       </h6>                 : HTML headings
<p>                   </p>                             : HTML paragraphs
<a href="URL"  > Anchored Text </a>                    : HTML Links (URL) with anchored text
<font size=+2>        </font>                          : HTML font size change
<hr>                  <hr>                             : HTML horizontal ruler
<option value="URL">  Text </option>                   : HTML Option update
<meta name="keywords" content="web-search, course">    : HTML Meta tag(s)
<area href="URL" >                                     : HTML other tags
<frame src="URL" >
<img src="URL" >
```

The differences between HTML and XML can be highlighted by the example below.

```
     H T M L                                  X M L
     ********                                 *****
<h1> Bibliography </h1>                       <bibliography>
<p> <i> Search Engines Information Retrieval  <book> <title>
        in Practice </i>                      Search Engines Information Retrieval
                                              in Practice </title>
    W. Bruce Croft, D. Metzler, T. Strohman   <author> W. Bruce Croft </author>
        <br> Addison Wesley 2010              <author> D. Metzler     </author>
                                              <author> T. Strohman    </author>
<p> <i> Introduction to Algorithms </i>       <publisher> Addison Wesley</publisher>
        Cormen, Leiserson, Rivest, and Stein  <year> 2010 </year>
        <br> McGraw-Hill 2009                 </book>
                                              ....
                                              </bibliography>
```

Figure 2: HTML vs XML: An example

3.1 Query: Parsing. When a **query** q is typed into the search box of a web search engine, i.e. it is presented to the search engine, such a query is "parsed" by the search engine first i.e. it is treated by the search engine as if it was a small `document`. Any query must adhere to the syntax (query language) supported by the search engine (and this language should be simple enough for the average to use it correctly).
3.2 Query: Text transformations. The words described in the query are then manipulated by the same text operations/-transformations that were applied to the documents of the corpus to thus derive the index-terms corresponding to the query. It would not make sense to look for `algorithmic` just because a user typed it in a query box, if all instances of `algorithmic` in the corpus are treated as synonyms of `algorithm` and only this latter word becomes an index-term. If the query contains not only words but some operators supported by the query language of the engine this processing becomes more evolved.
3.3 Query processing. Then for every document d_j of the corpus, the query engine establishes the similarity $s(q, d_j)$ or closeness of query q and document d_j (i.e. of two "documents") by comparing the index-term logical view of the query to the index-term logical view of every d_j of the corpus. The more rare index-terms two documents share the more similar they are. The similarity $s(q, d_j)$ is established by using statistical properties of the two "texts"; no attempt is (usually) being made to understand the contents of q or d_j. This approach is a pure information retrieval one to measuring similarity.

3.4 Keyword stuffing and purely IR approaches. One can fool such a statistical (or purely information retrieval) approach by creating a document that includes several (i.e. all identifiable) rare words or possible index-terms, a technique known as **keyword stuffing**. This was a problem with early generation search engines. Afterwards, search engines started using not only structure (aka keyword) information but also linked-ness or age information of the page or the site hosting the page to determine its similarity or "closeness" to query q. A rank $r(d_j)$ for document d_j or $r(q, d_j)$ can be computed that is more reliable than $s(q, d_j)$.
3.5 Link farming. Even such schemes that use link information could also be defeated. **Link farming** has been used to defeat a primitive ranking algorithm that relies solely on links. Create artificial links between **phoney/phony** web-pages and populate them with keywords (keyword-stuffing).

This Subject. In the remainder we describe the Architecture of an early generation search engine that was **Altavista**. In the following subject we present in more detail the **Google Architecture** as it was implemented between 1998 and 2002 or so. Later on in the course we build on this knowledge to explore it further and also examine what it looks like (or we think it looks like) nowadays.

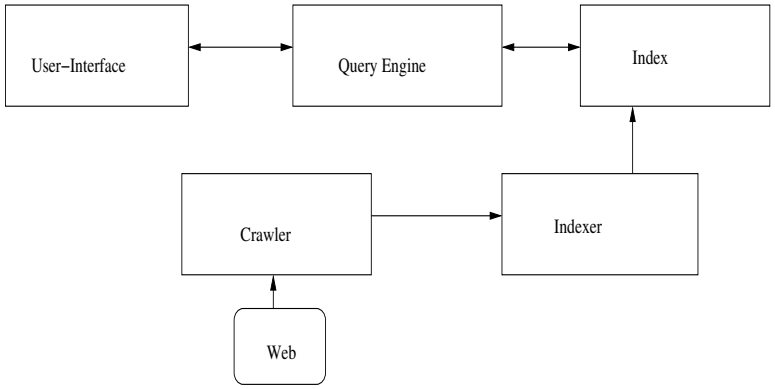


Figure 3: A centralized architecture of a Web search engine.

4.1 Web Search Engine Architecture: Altavista. In a centralized architecture such as that found in the Altavista search engine (see Figure 3), the architecture consists of two major components: (A) the query process, (B) the indexing process. The indexing process can also be further split into two major sub components consisting of: (B1) the crawler, and (B2) the indexer.

4.2 Web Search Engine vs a Desktop Search Engine Architecture. With that view of a Web search engine architecture, a Desktop search engine architecture can be obtained by eliminating or replacing the crawler with a program that traverses the directory structure of a locally available hard disk drive. Nothing else needs to be modified in Figure 3.

Four high-level steps can be used to describe the operations of this Web Search Engine Architecture.

S1.1. Preprocessing: Modeling. With reference to Figure 3 the preprocessing is related to the interaction between the component known as Crawler and the Web (2 bottom left boxes of the figure). The crawler is the program that will crawl the web, and find and fetch web-pages. The preprocessing is also related to what is going to happen afterwards, when those web-page fetched by the crawler are then fed into the Indexer that will process them and subsequently build an Index out of them. The wed search engine user is oblivious to that activity as it intercacts through the User-Interface (aka search box) with the Query Engine that performs operations similar to the ones described in 3.1-3.3 earlier. Item 3.3 in particular requires interaction with the Index itself and nothing else. Note that during the user interaction the Web is never accessed by the web search engine to facilitate the user's query. **Only the Index is involved**.

S1.1.1 Modeling: Text-model, file extensions, Multimedia. This is where the Corpus is defined, i.e. the collection of (Web) documents that will form the Corpus. The type of documents that will be crawled and retrieved and then parsed is decided at this early phase. Such documents can support a variety of "file extensions" such as .html, .txt, .pdf, .ps, .tex, .doc, docx, .ppt, .pptx, .xls, .xlsx. At this time it is decided what to do with non text sources (video, images, etc). The determination of what is going to be fetched, stored, and then processed will depend on the capabilities of the Indexer. The Indexer guides the modeling and the modeling depends on the Indexer's capabilities.

S1.1.2 Modeling: Languages and character encoding. One or multiple languages (e.g. English-only or more languages) and language encodings (eg. ASCII or Unicode) may also be supported or not.

S1.1.3 Modeling: Compression and other Protocols. At this time it is also decided how compression will be dealt with and which protocols are to be supported or not (e.g. .zip, .tgz, .gz). Whether .php related requests can or will be handled, and whether paywalls might be traversed.

S1.2 Preprocessing: Repository and its organization. All the documents that will be fetched from the web will be locally stored in some format. The organization of these documents and their storage determines or defines the **Repository: a local copy of (all or a subset of) the Web,** as detemined by the Crawler and Indexer. A repository of actual documents in possibly compressed format will be built. Metadata information collected related to these documents will be collected, stored and used for a variety of tasks including the determination of future Crawler schedules for the retrieval of updated copies of these and other documents.

S2.1 Crawler. Crawlers are programs (e.g. software agents) that traverse the Web in a methodical and automated manner sending new or updated pages to a repository for post processing. Crawlers are also referred to as **robots**, **bots**, **spiders** or **harvesters**.

S2.2 Crawler Policies. A Web crawler traverses the web according to a set of **policies** that include
(a) a selection policy,
(b) a visit policy,
(c) an observance policy, and
(d) a parallelization/coordination policy.

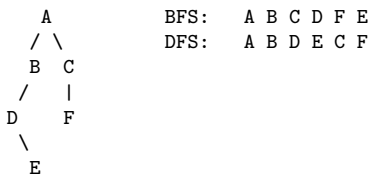
S2.2a. Selection Policy. A selection policy is in fact defined in Step 1 when based on the capabilities of the Indexer it is determined that only a fraction of the web-pages on the Web will be accessed, retrieved and subsequently processed/parsed and indexed. That fraction would depend on file extensions, multimedia handling, languages and encoding, compression and other protocols supported. Thus the implementation of the preprocessing modeling is realized through the selection policy of the crawler. Moreover, the location of potential Web documents needs to be determined.

For example, in 2009, out of a universe of 4 billion IP addresses (IPv4) only a sixth or so were registered. That number was expected to grow to one fourth by 2013 or so. The number of web-site names registered in 2009 was estimated to roughly 200 million, with that number growing to 950 million or more by 2015. These web-sites correspond to a number of approximately 300 million domain names and over 1 billion host names. Roughly 75% of the web-sites are inactive (what we call "parked web-site/domain-names"). The number of unique IP addresses corresponding to the web servers supporting the non-parked web-sites is even smaller. As a reminder, for reasons of load-balancing a given host name might get resolved to a different IP address at different times (or at different locations).

Volume of Information. In 1994 it was estimated that there were approximately 100,000 web sites. By 1997 this number grew to roughly 100,000,000. By the end of 2015 there might be 1 billion web-sites, one trillion potential URLs but not substantially more than 30-50 billion indexed web-pages. Google in 1997/1998 was crawling 50 million web-pages in roughly 4 to 6 weeks. More recently it can crawl the same number of pages in a minute or two.

S2.2b. Visit policy. Techniques for crawling the Web or visit policies are variants of the two fundamental graph search operations: Breadth-First Search (BFS) and Depth-First Search (DFS): the former uses a queue to maintain a list of to be visited sites, and the latter a stack. Between the two the latter (DFS) is more often used in crawling. (Why?)

Starting with a node say A (see example), BFS first visits the nodes at distance one from A, then those at distance two, and so on. In other words, for document A it finds its links to B and C and visits those pages one after the other. Then it extracts from those pages links to other pages and goes on as needed. In DFS on the other hand, you go as deep as you can go on any given link. Thus after A you follow the first link to B and then you realize that B points to D and that link is followed that include a linke to E. Because E has no links DFS backtracks once to explore other links from D that do not exist. It then backtracks once again and as B has no other links to follow DFS backtracks all the way to A to realize that A has one more link other than B to web-page C (and so on then to F).



S2.2b.1 Hyperlinked Web-pages. As already mentioned, the term **URL** refers to **Uniform Resource Locator** that describes the position/location of a document on the Web. The **crawler** collects web-pages by following hyperlinks of existing web-pages, in a depth-first-search or breadth-first-search fashion and possibly utilizing a URL server that can supply it with preexisting (from previous crawls) URLs. The supplied URLs can be used to initiate a new crawling or restart one when the current DFS or BFS search leads to a "sink" (dead-end) that points to no other pages that can be followed. Moreoever the new hyperlink information extracted from the currently retrieved URLs/documents can also be stored into that URL server instead of being followed immediately. As part of the visit policy, it is also decided the rate of crawling, often called **refresh rate** (i.e. how often pages are expected to change and thus re-crawled), the time schedule of a crawl (day or night), how to deal with anchors/links in documents, etc. Such decisions or actions might be affected by the availability of collected metadata: size of files, date of creation or last modification, hash of file contents, hash of URL.

Around 1998, crawlers used to undex several million pages a day; this number has grown to several billion by 2015.

S2.2b.2 URL server or not. A simple way to crawl the web is to (periodically) give the crawler program a list of URLs to visit. This information is provided by a URL server. The crawler can then expand on additional URLs reached from that initial search. This is the approach Google followed back in 1998. Another alternative is to stick to the list of URLs provided by the URL server; if some of the supplied URLs correspond to pages with links in them, these newly encountered links are not crawled but only sent back to the URL server: the latter decides whether it is a new link or not, and also determines the time to visit the link. Another alternative is to start with the most popular URLs; one more is to initiate a visit based on exhaustive search of URLs or more formally of IP addresses. However the considerations of the previous pages and in particular Step 1, make such approaches too ineffective unless. they are only used the first time or times a crawl is undertaken from scratch!

S2.2b.3 Information collected. Additional work may be performed on every page crawled. The crawler might collect and maintain additional `metainformation` about the crawled URL. This metainformation or **metadata** might include the size of the document, date/time information related to the crawling, date/time information of modification (write) of this document by its user creator, a hash (signature) of the document, etc.

S2.2b.4 URLs and docIDs. A web-page first crawled might be assigned a unique ID usually called `docID` for **document id**. Subsequent crawls that retrieve the same or newer versions of the document may not change the docID of that URL. A docID may be a "hash" (aka unique fingerprint) of the document's URL. More often however the hash is not the `docID` but is being used to assign to the document's URL a docID from a predetermined range. A docID can be a 32-bit word (used to be so in Google up to around 2004) or a 64-bit word. It is easier to reference a document by its 4- or 8-byte docID than say its variable length (50-byte or more) URL: less space is thus used for storing docIDs than actual URLs. Other information stored with a URL is a priority that determines how often the URL will be visited. The number of bytes of a docID is sometimes being used to optimize access to documents. More popular (or highly ranked) documents have "small" docIDs. Thus when one ranks documents, the docID serves as the rank itself (smaller values, higher ranks). This helps in cases where other information is stored in increasing docID (which implies that popular documents' information comes first aka at the top).

S2.2c. Observance policy. Other issues that can affect crawling include the visited web-server load. This is taken into consideration so that the crawler does not slow down the server's work or overload it. Some guidelines provided by the server will also help determine the robot's/crawler's behavior. Such guidelines are expressed through a `robots.txt` file and are called `observance policies`.

An example of a `robots.txt` file is depicted in the following Figure 4. Equivalently, one could include in the body of a web-page a `<META>` tag to request that the web-page not be indexed. This for example can be achieved as follows: `<META NAME="ROBOTS" CONTENT="NOINDEX, NOFOLLOW">` . A web-page that includes this tag is not only not indexed, but its links are not followed by a web-crawler that respects these tags.

```
User-agent: *
Disallow: private-data-directory
Disallow: /home/my-personal-files
Disallow: /cgi-bin/
Disallow: /images/
Disallow: /tmp/
Disallow: /private/
Crawl-delay: 10
Allow: /dir/mydir.html
Sitemap: http://www.njit.edu/sitemap_index.xml
Request-rate: 1/5           # Extended standard: one page every 5 seconds
Visit-time: 0600-0700       # UTC/GMT time


Reference:   http://www.searchengineworld.com/robots/robots_tutorial.htm
```

Figure 4: A `robots.txt` file

Note. As of 2014, Google ignores non top-level `robots.txt` directives, and uses only META tag information inside a web-page for observance policy processing.

S2.2d. Parallelization or coordination/synchronization policy of the crawler. Because of the vastness of the Web, there is not such thing as a single crawler (program). There are multiple crawlers running multiple threads of crawling each directed to a different set of URLs supplied by a URL server. In such cases their activity needs to be properly synchronized. The URL server might thus send URLs affecting a given domain or country to the same or different crawler.

S2.3 Example of Crawlers. First instances of crawlers include RBSE (Eichmann, 1994), WebCrawler (Pinkerton, 1994) , WebSPHINX (Miller and Bharat, 1998) written in Java (more on this in an assignment), the Google Crawler (Brin and Page, 1998), CobWeb (da Silva et al, 1999) written in Perl, Mercator (Heydon and Najork, 1999) written in Java, Webfountain (Edwards et al., 2001) written in C++, PolyBOT (Shlapenyuk and Suel, 2002), WebRACE (Zeinalipour-Yazti and Dikaiakos, 2002), Ubicrawler (Boldi et al., 2004), FAST Crawler (Risvik and Michelsen, 2002), WIRE (Baeza-Yates and Castillo, 2002) written in C++, Webbot: `http://www.w3.org/Robot/`. Spider Digimark searches only images (with watermarks). How large (as a software program) or how small a Web-crawler can be it depends on its capabilities. At the end of this subject one can find an example Web-crawler written in MATLAB.

S2.4 Duplicates. A web crawler in its simplest depiction of Figure 3 retrieves pages from the Web and sends them to another component (or actor) called the `indexer` (either directly or through the Repository). One task that this other program must perform is to decide whether the retrieved web-page is a duplicate of another web-page located at "another" URL. For example it might be the case that two host names in the same or different domain e.g. `web.njit.edu` and `www.cs.njit.edu` are redirects of one another. One would not want to deal with both since this would duplicate (unnecessarily) the relevant processing work. In some other cases a web-page has not been modified since the previous crawl; there is no reason to be processed (indexed) again. Also, multiple copies of the same web-page might co-exist in the same directory or even in different URLs; there is no reason to process all of them separately since processing one web-page suffices.

S2.5 Deep Web vs Surface Web. Implied in Figure 3 is that the term "Web" refers to the Surface Web. In several cases the Deep Web may interact with a crawler by feeding it information that has just been created. For example a news organization might provide in the form of a news feed a continuous stream of news items to a crawler. Tagged video or audio stream might also be fed this way; in such cases only the text-based tags will be indexed afterwards.

S3.1: The indexer. The input to the **indexer** is the collection of web pages fetched from the Web by the crawler and then delivered to the indexer directly or indirectly through the Repository. The output of the indexer is the **(inverted) index**. The index (aka inverted index) is **an efficient data structure that represents the documents of a Corpus and allows fast searching of the Corpus documents using that indexed information**.

S3.2 Indexing Process: Forward index and (inverted) index. The indexer first creates an intermediate data structure that is known as the **forward index**. Subsequently the forward index is **inverted** by the indexer using a sorting operation. The output of this inversion is known as the **inverted index** or plainly as **index** or sometimes as **inverted file**. The whole set of operations that generates the forward index, the (inverted) index and other auxiliary data structures that support both is known collectively as the **indexing process**.

S3.2.1 Forward index. A representation of data in it is in the form of tuples sorted by `docID` and within a docID by `wordoffset`: `(docID, wordID, wordoffset, context)`

S3.2.2 (Inverted) index. The same tuples of the forward index are now odered by `wordID`, then `docID` and then `wordoffset`.

S3.2.3 Index: Distribution. An index is several times larger than the Corpus it indexes. Several (tens or hundreds) of machines (aka "servers") might be required to just store one or more copies of the index. The **parallelization** or not of all or certain portions of the indexing process is important in getting good performance out of a search engine. `Parallelization` in this context means that multiple computers will cooperatively work in parsing and tokenizing the Corpus, computing the forward index, sorting (i.e. inverting) it and then generating the (inverted) index, storing and (thus distributing) the (inverted) index among a multiplicity of servers, that will then facilitate the efficient use of the index by the query engine!

For example as multiple servers will need to store the original Corpus and the index derived from that Corpus, it is of paramount importance that high availability of servers is achieved. In order to achieve redundancy, file or index **replication** might also be involved (e.g. Google File System) in a way that is transparent to the user of the system. Additional information will then need to be maintained to have a distributed index.

S3.3 Web Search is Searching the Index of the Web not the Web itself! When we perform an ad-hoc search of the Web, we do not search the Web directly but the representation of the documents of the Web as abstracted by the constructed index. The index usually stores a representation of a (sometimes incomplete) copy of the Web.

S3.4 Crawler and Indexer. Although Figure 3 implies a direct connection or communication between the crawler and the indexer, more often than not this might not be the case. If there is a communication it is indirect through a server (or servers) known as the **Repository**. The Repository received information from the crawler and supplies this information to the Indexer: the information is the set of web-pages (Corpus) crawled from the Web along with metainformation that was collected by the Crawler and stored into the Repository. In several cases there is no direct communication between the crawler and the indexer. The crawler retrieves the URLs supplied to it (by say, the URL server) and converts them into local copies of the Web-available documents along with a variety of metainformation. Then the indexer, independently of the crawler starts the indexing process. With such a scheme the indexer can work/index one copy of the Web, the crawler work on a more recent one, and the query process/engine access an earlier, older version of the representation of the Web, older than the ones used by the Crawler or the Indexer. Periodically, say every month, the three copies are cycled over and one becomes obsolete (the current copy operated on by the query engine). This abstraction and form was the one used by Goodle through around 2008 to 2010. Since then, things have changed significantly as it will be explained later.

S3.5 Index-terms and index vocabulary. Search terms. As explained earlier after tokens are extracted from a Web document, certain text transformation are performed and those tokens become first keywords and then determine the set of **index-terms** of the document. Those index-terms can be words, phrases, names, dates, and even links to a web-page. It is also possible that those same index-terms do not appear in a (the) document themselves. Those index-terms are then used in the building of the forward and inverted indexes (or indices). The set of all the index-terms that will be indexed is sometimes called the **index vocabulary**. The index vocabulary is known not only by the indexer but also by the query engine. The same transformations that the indexer performs to convert tokens into index-terms must also be performed by the query engine to convert user-supplied **search terms** into indexable `index-terms`.

S3.6 Forward index. For every document in the collection, the indexer generates a sequence of tuples that describe that document. Each such tuple contains information about the document (docID), the token/keyword/index-term encountered (wordID), an offset of the token etc in that document from the start of the document (word or character offset), and some context information (does the word appear in the title of an HTML or other document, or in the anchor field of a link, in modified font such as in bold or emphasized font or in elevated font size). Compression of the forward index is quite possible: all tuples of the same document have the same docID, the one of the document itself. Consecutive tuples have offsets that are close to each other, i.e. two consecutive offsets differ by a very small number (such as one or two for word offsets).

S3.7 Index or Inverted index. A sorting operation is applied to the forward index to derive the index (inverted-index). This inversion (sorting process) would sort the forward index tuples by wordID first, then (for the same wordID tuples) by docID, then (for the same wordID, docID) by offset/context.

This way the **index is defined** *for a given index-term to be the list of documents (docIDs/URLs) that contain that index-term.*

Implicit to it is also an ordering by docID, word (or otherwise) offset. The index needs to be efficient and easily updated when new web-pages or updated versions of a current web-page are to be reencountered.

S3.7.1 Why Inverted? It is called an inverted index because its information is the inverse of that found in a file: a file contains a list of index-terms (i.e. the words or tokens in the file) whereas an inverted index contains for a given index-term the documents that contains that term. (And inversion is the sorting operation that converts one form into the other.)

S3.7.2 An example. On the example of the following page, Stage 1 generates the inverted index from the forward index after the sorting operation by `wordID, docID, offset` as stated above. Consecutive (top to bottom tuples) of the inverted index are thus ordered by wordID (middle i.e. second field) and those tuples that have the same wordID value are also ordered by offset (i.e. third field). Thus by a linear scan we can group those tuples by wordID. This is stage 2. There are two tuples with wordID equal to 1. Thus for each wordID we build a list of tuples i.e. the instances of appearance of the corresponding word in the Corpus. Thus the word with wordID equal to 4 (i.e. alex) appears 4 times in the Corpus. Auxiliary structures include a doclist aka a mapping of docID into "document descriptors aka URLs", a "vocabulary" that maintains in sorted order the word with associated wordIDs plus other interesting information, and a "hash table"-based Lexicon that allow for fast look-up by a word value to retrieve the word's corresponding wordID. Hashing is faster than say binary search (on the average).

Step 3: An example

```
Doc1  : docID=1 :   algorithm data structure alex
Doc2  : docID=2 :   algoritms data alex structure
Doc3  : docID=3 :   structures alex data alex

Forward index is a triplet  (docID, wordID, wordoffset ) or a
                qudruplet  (docID, wordID, wordoffset, context )

Forward Index                                  Inverted Index
(1,1,1)                                          (1,1,1)
(1,2,2)                                          (2,1,1)
(1,3,3)                                          (1,2,2)
(1,4,4)                                          (2,2,1)
(2,1,1)    ---- Group by or Sort by -------->     (3,2,3)               Stage 1
(2,2,1)             wordID                       (1,3,3)
(2,4,3)             docID                        (2,3,4)
(2,3,4)             offset                       (3,3,1)
(3,3,1)                                          (1,4,4)
(3,4,2)                                          (2,4,3)
(3,2,3)                                          (3,4,2)
(3,4,4)                                          (3,4,4)


                                    Inverted Index
                   (1,1,1)       1 :  (1,-,1) (2,-,1)
                   (2,1,1)       2 :  (1,-,2) (2,-,1) (3,-,3)          Stage 2
                   (1,2,2)       3 :  (1,-,3) (2,-,4) (3,-,1)
                   (2,2,1)       4 :  (1,-,4) (2,-,3) (3,-,2) (3,-,4)
                   (3,2,3) --->
                   (1,3,3)
                   (2,3,4)
                   (3,3,1)
                   (1,4,4)
                   (2,4,3)
                   (3,4,2)
 DocList :  1
**********  2
            3
 Vocabulary                                    Lexicon
 #word      #wordID  #hash? #other              #word    #wordID
 alex       4                                  data      2
 algorithm  1                                  algorithm 1
 data       2                                  structure 3
 structure  3                                  alex      4
```


The indexing process in more detail

S3.8 Preparing for tokenization in Step 3-1: File encoding, language. The Corpus that was fetched by the crawler in Step 2 and potentially stored into a Repository can be viewed as a stream of files of multiple formats (.txt, .html, .pdf, .doc, .xml, etc) compressed or not that will be subsequently converted into text form; this will also generate some additional metadata about the Corpus files. Such processing might be more involved for .pdf or .doc or .docx formats than say .txt or .html or .xml files for example. At that time the encoding scheme of the text content will be determined as well as the language used in a document. Metadata related to the document will be accessed (date of creation, modification, etc). During this process data structures related to the index might get updated. It is for this reason that in Figure 3 the indexer interacts with the index by reading and writing (updating) information relevant to the fetched web-pages. The end result of this phase of Step 3 is that multiple formats have been converted into some common text-based form along with related metadata.

S3.9. Step 3-2: Text transformations involving parsing and tokenization. Operations that will be performed on the designated text(s) need to be defined and the text model used needs to be specified (i.e. the text structure, what elements of the text can be retrieved), dealing with issues such as whether numbers (e.g. 1776) or dates (eg. July 4, or 9/11) are to be further processed or not, how hyphenization will be dealt with, if common words will become keywords, and so on. The indexer then goes into a second phase of transforming raw unstructured text into `index-terms`. One of the first operations in this second phase of Step 3 is the **parsing** phase that will extract through `tokenization` what will eventually become its `index-terms`. Parsing and tokenization first generate tokens which after further transformations will lead to the formation of "keywords" or "index-terms". These first transformations have a simple linguistic and context-based element. They deal with issues such as what will become a token, and whether its position within a document is important or not. Data compression (or decompression) might also be re-used if as a result of the first phase of Step 3 the raw text was compressed. What constitutes a word, which of the words are treated as keywords, and which, if any, of the keywords become index-terms depends on many factors one of which has to do with the amount of available space that will be utilized by the IR system (or search engine). Web-based text might be unstructured or very poorly structured!

Step 3-1 and Step 3-2 (Prior to computing the forward index)

Relevant questions at this point to identify what constitutes a token are as follows.

- **3-2.1 Token.** Is a token (word) a sequence of alphanumeric characters separated by white space (space characters, end-of-line symbols, tabs, etc)? Is order of parsing important (eg. left-to-right or right-to-left).
- **3-2.2 Case-folding.** Is capitalization important? Does it suffice to convert capital case to lower case for every character/word of the text? Can capitalization be of importance (other than the obvious one)? Is `apple` different from `Apple` ? Is `windows` and `Windows` the same ? Is OR (i.e. Oregon) different from or ?
- **3-2.3 Numbers and Expressions.** Will numbers become keywords or index terms? How do you treat 1776? Is 9/11 a division (integer or otherwise) or a date?
- **3-2.4 Dates.** Are dates indexable? How about July 4? Is 9/11 a date or a division (calculation)? How about 7/4, 4 July , or 4/7 , July 4, 1776, or 17760704, or 7/4/1776 or 4/7/1776. What is 7/4/76? Is it 7/4/76 indeed or the Bicentennial 7/4/1976?
- **3-2.5 Hyphenization.** Resolve hyphens (e.g. state-of-the-art or state of the art). Is it `Web search` or `Web-search`? `e-bay` or `ebay`, `wal-mart` or `walmart`?
- **3-2.6 Phrases.** Is it `on-line` or `online` ? In a sentence of the form **The coach of Manchester United states ...** or a sentence `United States`) what needs to be done to associate `United` with `Manchester` in the former and `States` in the latter? Take into consideration capitalization (spelling errors?) or syntactic context?
- **3-2.7 Accents and punctuation marks.** For example resume vs résumé. Is the apostrophe in `O'Brien` the same as in `George's` or `owner's` ? Is `I.B.M.` different from `IBM`? Or is the feline `CAT` different from a `C.A.T`, the medical device? Are punctuation marks important and significant and variable? Is it 480B.C. or 480 BC. or 480 BC or 480 BCE, how about USA or U.S.A. or US or US? How about `a.out` ?
- **3-2.8 Document structure.** Does the document have structure that will help us in dealing with it? Is it an HTML document? For example headers in HTML signify an important label as in `<h2>Search</h2>` and thus we might wish to treat that `Search` word as a more important indicator of the contents of the document. Thus the tokenizer might return not just the token or combination of tokens identified (say `Search`) but also metadata information about it other than its value (name). For example for the HTML-based `Search`, its appearance in the header of an HTML document becomes important and it will be transmitted as metadata information by the tokenizer.

Step 3-3: Language based transformations

S3-3. Language-based transformations. In the next (third) phase of text transformations (during parsing), a token derived through parsing might become an index-term or further processing or refining might be necessary before it becomes so. Additional language-related transformations may apply. These include the following.

- **S3-3.1 Stopword.** Frequent not very useful short words or tokens such as articles and connectives might not be important. Will they become index-terms eventually or not? One reason NOT to index stopwords is storage or space issues. The inverted index might grow too large beyond the index's storage capabilities. If space is not an issue one can decide not to treat some or all of these words as stopwords (and thus index them).

  ```
  I a about an are as at be by com en for from how in is it  of on or that
  the this to was what when where who will with www
  ```

- **S3-3.2 Short Words.** Are there short words that are important ? For example `xp, nt` might be important in the context of `Windows XP` or `Windows NT`. Or `US` or `WWII`.
- **S3-3.3 English language: American English vs (British) English vs ....** Or how many millions in a billion ? In English a billion is a million millions (aka trillion or 10^{12}) but in (American) English it is a thousand million (i.e. 10^9).
- **S3-3.4 Stemming.** The operation known as stemming reduces the number of distinct words to their common grammatical root and thus `fish` and `fishes` and the occasional `fishess` (sic) will generate one index-term if stemming is used.
- **S3-3.5 Noun groups.** How many index-terms will `kitchen table` generate? How about `down under` ? **Hint: Australia (for the latter).**
- **S3-3.6 Synonyms.** The identification of **similar (in meaning) words** called **synonyms** might reduce the number of index-terms. Is a `car`, an `automobile`, a `sedan`, a `coupe`, `vehicle`, a `convertible` the same or not? Will they generate one index term?
- **S3-3.7 Acronyms.** (And treatment of acronyms.) Is OR the state of Oregon, Operations Research, Outdoor Research, or an Operating Room or all of the above?
- **S3-3.8 Temporal factors.** How would you have treated a 9/11 on September 1, 2001?

Step 3-4: Source file-based text transformations

Just after Step 3-3, a **semantics-based classification** of a Web-page might also take place that characterizes it for example as `sports`, `politics`, `academic`, etc. Around this time certain source file text transformations take place.

S3-4. Source file-based transformations. Links and anchor text from web-pages can also be identified and extracted. For example, the file below "named" `index.html` points to a pdf document `paper1.pdf` and it also tells us that this `paper1.pdf` is a paper related to Web search. Thus the keywords `Web, Search, Paper` are important not for the Web-document `index.html` in which they are located but for the document to which the link points i.e. `paper1.pdf`. This information indicates that `index.html` points towards `paper1.pdf` and this link information can be exploited by certain ranking algorithms. Google's PageRank algorithm (Brin and Page, 1998) is such an example. Likewise, `new-dir/paper2.pdf` is also pointed by `index.html` and keywords associated with `paper2.pdf` include `Data, Structures, Paper`. Somehow during the parsing or indexing of `index.html` information that relates to `paper1.pdf` and `paper2.pdf` is collected that would prove useful for the indexing of these two latter files. This information must be stored or temporarily saved somewhere and used when the indexing of `paper1.pdf , paper2.pdf` takes place!

Moreover for both `paper1.pdf` and `paper2.pdf` their complete URL information needs to be reconstructed as well. Moreover this complete information will be sent to a URLserver (or indexer as needed) that would remind them that those two files are to be indexed as well (and also crawled prior to indexing). If the URLserver does not know about this files, it will have to create records for them and schedule their crawling.

Thus the activity that takes place during parsing is not strictly related to the file being parsed!

```
<!-- This is file index.html -->
<!DOCTYPE html> <HTML> <HEAD><TITLE>Some file with paper links</TITLE></HEAD>
<BODY>
<UL>
<LI>
<A HREF="paper1.pdf"><EM> Web Search Paper</EM></A>
</LI>
<LI>
<A HREF="new-dir/paper2.pdf"><EM> Data Structures Paper</EM></A>
</LI>
</UL>
</BODY></HTML>
```


Step 3-5: Auxiliary Information in Index

The index will maintain a variety of "data structures" and information that will facilitate web-search through index-term searching.

S3-5.1 Document-related info: `docID` and `URL`. Every document has a URL that is indicative of its location. For example the course URL is
`http://www.cs.njit.edu/~ alexg/courses/cs345/index.html`
We do not want however to store this long character string several times in the index. To facilitate efficient look-up we replace it with a shorter `docID` as explained earlier. In addition we might maintain the "hash" of the URL of the document: the "hashURL" is a unique fingerprint of the URL, shorter than the string of characters it contains. Other metadata information about the URL or its `docID` might be stored such as its type, size, whether it is compressed or not, encoding (ASCII vs Unicode), language (English or not), and other. All this information might get organized in a variety of ways: (a) A hash table indexed by the "hashURL" or `docID` that includes also a pointer to the location storing the URL itself plus the metadata information, and (b) a sorted table by "hashURL" that contains fixed-length records containing the "hashURL" and `docID`. That latter table might be used once to determine whether a given URL (through its "hashURL") exists or not and what its `docID` is!

S3-5.2 Word-related. Similarly to the use of a `docID` instead of a `URL`, we might maintain a shorter `wordID` to identify a word that is an index-term and also a hash of the word let us call it `''hashWord''`. The "hashWord" might be used to access a hash table, that stores the word itself, its `wordID` and other statistical information related to the word. The same `wordID` might identify in fact all synonyms. Such table or tables might contain statistical information about the word such as the number of distinct documents that contain it (at least one instance of it) and also the total number of occurrences of the word in the Corpus.

S3-5.3 Statistical information. Index-term weights will indicate the relative importance of words/token/index-terms in documents. This information can be used for measuring the similarity of a query to a given document of the Corpus. Although some calculations will be performed at query time, some other ones can be performed at index creation time.

- **N** is the number of documents in the corpus.
- n_i is the number of distinct documents containing index-term i.
- **Term frequencies** tf_i **and** tf_{ij} is (a) the number of occurrences of index-term i in all documents of the corpus, and (b) the number of occurrences of index-term i in document d_j respectively.
- **Inverted document frequency** idf_i is the inverted document frequency for term i, i.e. how important a rare term is and is defined as $\lg(N/n_i)$.

Note. In this context $\lg x$ is the logarithm of x base two, i.e. that quantity such that $2^{\lg x} = x$. For example $\lg 8 = 3$ since $2^3 = 8$.

Step 4: Query Process (User Interface and Query Language)

S4.1: Query Process. A web-search engine communicates with the outside world with a **query user interface** supplied by the **query engine**. The user communicates through that interface with the query engine to provide the query, and then the query engine accesses the index to satisfy the query. In fact this "standard approach" is being utilized roughly 15-20% of the time; more often than not the query has been asked before and thus **the answer has already been precomputed** and is available in a cache maintained by the web search engine. As soon as the answer is computed an HTML output is generated through a web-server that contains the results of the query. In order to form this output the web search engine will also access the local copy of the Web maintained through the crawling process/Repository (Step 2); this information is used to provide the title page information, as well as context information for each result generated. The ranking of the results in the output might require access of other data structures as well. In addition the query engine communicates with ad server(s) whose objective is to provide and insert sponsored link information into the HTML output. Statistics (IP address or user information), usage, user habits, and of course the query itself are collected and archived through the whole process. A query that requires access to the index is answered more often than not in under 0.2 seconds using fewer than 1000 or so servers.

Year	Total queries/day	Google's share (queries/day)	
1994	1,500	-	
1997	20,000,000	-	
1998	22,000,000	10,000	
2003	320,000,000	112,000,000	
2006/US	220,000,000	90,000,000	[US share only for both measures!]
2013	4,500,000,000	3,000,000,000	[Worldwide]
2013/US	600,000,000	400,000,000	[US]
2015	5,200,000,000	3,500,000,000	[Worldwide]

S4.2 Query Process: User interface (query text box) and Query Engine. The query process consists of the **user-interface** and the **query engine.** In its simplest form the user-interface consists of a **query text box** that accepts user queries. The user-interface is provided in the main area of a browser or is installed as an addon in the toolbar area of the browser. One or the other is the user's entry point to the Web Search Engine.

S4.3. Query Process: Query Language. The user is writing a query using (sometimes implicitly) the search engine's **query language**. The user-interface parses user queries, and converts search terms in a form that is acceptable for input to the query engine i.e. into index terms that appear in the index vocabulary. User queries should be written in accordance to the **query language** used by the search engine. As the average user is not programming language proficient the query language needs to be quite simple and obvious. Such an input language may allow for conjunctive or disjunctive connectors (i.e. operators). An operator is a command that treats certain text in a special way.

For example several search engines treat specially operators such as `AND` (conjunctive operator) and `OR` (disjunctive operator). Some search engines (e.g. Bing) also recognize `NOT` (negation operation). (Note that for Google − is the negation operator whereas Bing accepts both − and `NOT`.) Thus `data AND structures` is equivalent to `data structures` in most search engines, in the sense that the query is parsed to deliver all documents that contain both the term `data` AND the term `structures` (the two terms do not need to be next to each other in any result document). In that sense an empty space between the two keywords is equivalent to writing a capital-case `AND` and treat it as a conjunction. A disjunction operator `OR` needs to be explicitly typed in upper case. It is treated as an operator depending on the context of the query: if there are no left and right operands around it the query interface/query engine might interpret it as an acronym, among others, for the State of Oregon, or (no pun intended) for Operations Research, Operating Room, etc. Even if the query language may treat `AND` specially this MAY not apply to `and` which is treated potentially as an ordinary word/keyword/index-term. Thus `data AND structures` and (no pun intended) `data and structures` might not be equivalent and might provide different results. In addition to these operators, the query language might also allow for quotation operators to indicate an exact match. Thus `"data and structures"` will find and return to the user all documents in which the three keywords appear in this exact order, one after the other. One way to denote negation is by prepending a minus in from of the keyword. Thus `-data` finds all documents that DO NOT contain the word `data`.

In addition rudimentary typo correction may also be undertaken. With the example of Subject 1 in mind, a user who types `extrem` might get alerted by the `user-interface` into getting the spelling corrected or offered a list of alternatives or matches including say the correct `extreme` instead. This is because the `query engine` realizes that there is an `extreme` but (possibly) not an `extrem` in the index vocabulary maintained by the indexer and accessible by the query engine, and this information is communicated by the query engine to the user through the `user-interface`.

S4.4 Query engine: Accepting and translating user query requests. The query the user types in the query text box is just a *query request*. It is then translated by the **query engine** into a *query* that is compatible with the search-engine view of the Web. For example the words the user types are checked for typos and alternatives or corrections are suggested or initiated, and then converted, after parsing and applying the same text transformations used in Step 3, into index-terms that can be recognized by the indexer (and stored in the index vocabulary).

S4.5. Query engine: Query Operation. Then and only then will the **query engine** realize a *query operation* in which the index is accessed. For each index-term in the query the inverted-list for that term is generated (sequence of documents containing that term). If the query is a conjunction or disjunction a sequence of operations is performed on those doc-lists (i.e. lists of docIDs) to resolve the query. Thus a list of *potentially relevant documents* as determined by the search engine is generated. This initial list might contain docIDs that are then converted into URLs. The results (i.e. documents that are relevant to the query) are then ordered by using the engine's **ranking algorithm** or **ranking function** and are presented in a given format to the user (e.g. by breaking them into pages of say 10 document links per browser page, by highlighting and printing the text area of each document that contains the query etc). In addition advertising-related or other commercial activity might also take place in this step. For example certain areas of the displayed output might include advertising links related to the keywords input. In addition, if foreign language documents are found, a translation (say into English) component might also get activated.

As we have already mentioned, most search engines determine document relevancy to a query based on statistical properties of the text of the document by establishing a similarity measure $s(q, d_j)$. That measure can be a 0 or 1 or a real number in general depending on factors such as (a) whether d_j satisfies or not q, (b) whether q is satisfied multiple times by d_j, (c) the proximity of the keywords of q in d_j, (d) the context in which the keywords appear in the document (is it in the title or header of the document, do they appear in higher font size, italicized or in bold-face), (e) the importance of a keyword and a weight assigned to that keyword (e.g. common frequent words have less weight that uncommon infrequent words whose appearance both in the query and in the document might signify a more important event), (f) link information, (g) the reputation and age of the location in which the document was found.

S4.6 Post-mortem query analysis. Logging. Query logs of the users' interactions with the search engine are obtained and are of paramount importance. They can improve the search experience, speed up results, store results of common queries, and identify source of new revenue. In addition the query engine maintains logs of user activity on the output. Pages that are clicked or ignored might be logged to improve the overall quality of the search engine but also detect patterns in user activity (i.e. data-mining) that will help in subsequent commercialization efforts. Information on ad-clicks is also collected to facilitate accounting/revenue gathering.

Query logs can be used for a variety of other reasons that include

- keeping track of a history of user queries,
- generation of spell checking logs (instead of running the spellchecker every time)
- recording of time spent on the query or a particular document,
- query caching of important but time dependent queries (e.g. election result-related queries might be cached and be readily available in the first two weeks of November, or US Open tennis results in the first few weeks of September).

Monitoring of the user experience might also take place. If response times are higher than normal, a message to the user might be generated. If successive queries generate a large and similar output then it is possible that queries will not be reexecuted by the query engine.

Challenges

Here we outline some the challenges related to web-searching that can be more clearly understood in the context of designing a search engine architecture, and given the (brief) knowledge obtained in this Subject.

- Modeling.
- Querying. User Interfaces.
- Distributed Data/Architectures. Data spans over many computers and platforms. Web addresses of web servers not informative. Network reliability and topology unpredictable and varying.
- High percentage of volatile data. Internet is a dynamic entity; things get updated often, links become dangling, data get destroyed.
- Quality of data. (False pages, typos, out-of-date information.) Error rates range from 1 in 200 to 1 in 3.
- Large Volume of data!
- Ranking. (and Cheating.)
- Dynamic Pages. (Deep Web, Hidden Web.)
- Indexing. What should be indexed?
- Unstructured and Redundant/Duplicate Data. The Web is NOT a distributed hypertext; *hypertext assumes the existence of a conceptual model behind it, which organizes the collection and adds consistency to the data and hyperlinks*. Approximately 30% of Web pages are duplicates.
- Multimedia : enough said!
- Heterogeneous data (not only in multiple document types but also languages).
- Browsing. (Unify searching with browsing.)

Corpus size of a search engine

1. Corpus size N. Let the corpus size of a collection be N. If we do not its value, we can get an estimate of it by performing three queries involving two index-terms w_1 and w_2 that are highly uncorrelated to each other. Two such terms can be $w_1 =$ **Tropical**, $w_2 =$ **Lincoln**, as noted in the textbook. However a better choice might be $w_1 =$ **delicious**, $w_2 =$ **CS345**.

2. Why does (might) this experiment work ? For a query involving term w_1 we get a hit count of f_1 (number of documents containing w_1). This frequency f_1 can become an a-priori probability p_1 of a randomly (uniformly at random) selected document containing term w_1. p_1 is then equal to

$$p_1 = f_1/N$$

Similarly for w_2 we observe its frequency f_2 and we derive a probability

$$p_2 = f_2/N$$

Then we perform a query that contains the conjunction of w_1 and w_2 and observe a frequency count of f_{12}. This implies that

$$p_{12} = f_{12}/N$$

If w_1, w_2 are unrelated and independent of each other then $p_{12} = p_1 \cdot p_2$. The latter implies $(f_1/N) \cdot (f_2/N) = f_{12}/N$. Solving for N we obtain.

$$N = \frac{f_1 f_2}{f_{12}}$$

For the first pair of keywords, Google gives frequencies of 1.92bn, 342million, 12.8 million implying an $N = 51$billion. Bing on the other hand gives, 86.7million, 117million, 1.98million implying an $N = 5.1$ billion.

For the other delicious pair, Google gives 604million, 87800, and 2100 implying an $N = 25.2$ billion (**a good estimate**). Bing was giving 90.6million, 18400, but was unstable ranging between 508 to 30800 for w_{12}. This gives an $N = 3$billion, which is a bit off. However Yahoo! seems to be using Bing technology: the first query gave 90.3million, the second 18400, and the third one a 31,100 that is in fact a 78. This latter 78 was used to estimate an $N = 21.3$billion (**a good estimate**). The max i ever got for everything out of the three search engines was 25.27billion for Google, and 29.6billion for Yahoo! and 29.5billion for Bing.

	SEARCHES PER DAY (in millions)		
	02/2003	02/2009	
Engine	Searches/day	Searches/day	
Google	112	320	Google
AOL Search	93	3	AOL
Yahoo!	42	74	Yahoo!
MSN Search	32	28	MSN/Live
Ask Jeeves	14	10	Ask.com
InfoSpace	7	-	-
TOTALS	319	430	

	02/2003 [in millions]		
Engine	hrs/month	mins/day	search/day
Google	18.7	37	112
AOL Search	15.5	31	93
Yahoo!	7.1	14	42
MSN Search	5.4	11	32
Ask Jeeves	2.3	5	14
InfoSpace	1.1	2	7
TOTALS	53.2	106	319

Source (Feb. 25, 2003)
SearchEngineWatch.com,
Source (April 2009)
http://blog.compete.com/2009/04/13/search-market-share-march-google-yahoo-msn-live-ask-aol-2/

Figure 5: Search Engines Statistics: 2003 vs 2009

Search Engine Market Share

Figure 5 shows some raw statistics for several popular search engines for 2003 and 2009. Over a 6 year period the search volume increased by 35% . Some search engines (e.g. Google) grew in popularity, and some others faltered. The total volume of searches back in 2003 was approximately 319 million realized within 106 million minutes. Each search took approximately 20 seconds of processing time. If a cluster of 1000 machines was used to realize such a search, it would have required on the average 20 milliseconds per machine. The 319 million figure, means that on the average there were 4,000 searches per second over all the popular search engines.

The global marketshare of Google for example was more than 80% in 2009 (but nowadays in 2015 it is a lower 67%), as it can be seen in Figure 6 as well.

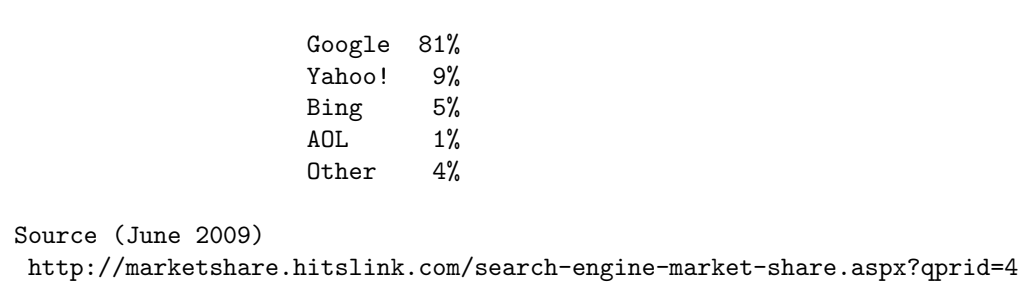


Figure 6: Search engine market share (worldwide)

Search Engine Query Length

Most users type in a search engine no more than 4 keywords. Figure 7 shows the distribution of keyword length in Google queries.

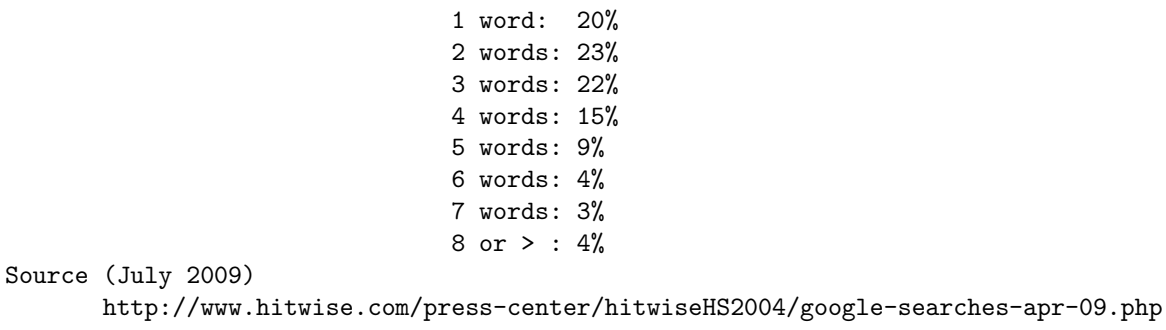


Figure 7: Search engine query keyword length

IP addresses and domain names

The most difficult task of a search engine is not to index a web-page; it is to find it in the first place. Indexing is much faster and more efficient than finding. A web-crawler that grabs pages faces a number of daunting tasks: enumerating all possible locations in which information (in the form of a web-page) appears.

- Out of approximately 4 billion IP addresses a crawler needs to decide which ones are useful, i.e. find the ones that they correspond to a web-server that responds to HTTP requests. IP addresses are quadruplets of numbers in the range 0-255. For example 74.125.67.100 is an IP address. Roughly 1 out of 7 IP addresses are registered, or close to 600 million. Yet there are approximately 183,000,000 domain name registrations (2011).
- Domain name registrations are symbolic names that a user of a web-page remembers. Several times multiple domain names get mapped to the same IP address. But several other times the same domain or host name can get mapped to multiple IP addresses for load-balancing or other reasons. For example, 74.125.67.100, 74.125.127.100, 74.125.45.100, are IP addresses for `google.com` whereas `64.233.169.99, 64.233.169.103, 64.233.169.104, 64.233.169.147` are addresses for `www.l.google.com` that resolves a lookup for the IP address of the `www.google.com` host name.

Searching by IP address would be time consuming and highly inefficient. 6 out of 7 possible IP addresses would not be active. And fewer of the remaining one would correspond to a web-server. Out of a universe of 180 million domain name registrations, according to Figure 8, one expects only 60% or less to have useful indexable information. This is corroborated by a 2004 study, shown in Figure 9 that predicts that there were 32 million of host name registrations responding to an HTTP request and mapped to fewer than 9 million IP addresses out of the universe of 4 billion of them (or close to 1 out 400).

figs345 8 and 9 present a variety of web-host count estimates.


```
         Domain Name                              Average DNS traffic volume:   38,000,000,000
        Registrations                             [DNS=Domain Name Server that translates a
2003       32,000,000                                  domain name into an IP address]
2004       64,500,000
2009      183,000,000
2013      250,000,000-300,000,000

                                                   Second-Level DN           Third-Level DN
Host Count:     625,226,456  [Jan 09]  ------->  33026822  ne.jp
               ~541,000,000  [Jan 08]            23451389  rr.com
               ~433,000,000  [Jan 07]            21023670  myvzw.com
               ~394,000,000  [Jan 06]            14598646  telecomitalia.it
               ~317,000,000  [Jan 05]            13015858  verizon.net ----> 6312040 east.verizon.net
               ~233,000,000  [Jan 04]                                       3434661 fios.verizon.net
               ~171,000,000  [Jan 03]                                       2775048 dsl-w.verizon.net
               ~147,000,000  [Jan 02]
               ~109,000,000  [Jan 01]
               ~ 72,000,000  [Jan 00]
               ~ 43,000,000  [Jan 99]          SOURCE
               ~ 29,000,000  [Jan 98]          Verisign Domain Name Industry Brief  / Domain Report
               ~ 16,000,000  [Jan 97]             (Uses a 2009/2010 brief except as noted)
               ~  9,000,000  [Jan 96]
               ~  5,000,000  [Jan 95]
               ~  2,000,000  [Jan 94]
               ~  1,000,000  [Jan 93]

Of 92,000,000 tested Domain Names,        89% resolve to a Web-site.
                                          Of them
                                           24%   one-page web sites
                                           64%   multiple-page web sites
                                           12%   no web-site.
```

Figure 8: Web-Host Count Estimate

```
          2004 Data
Size of Web    ~32,000,000 Based on domain name registration
               ~42,800,000 Web servers responding to HTTP requests
               ~ 9,000,000 IP addresses responding to HTTP request (each IP
                           can maintain many virtual domain names).
```

Figure 9: Another Web-Host Count Estimate (2004)

Figure 10 shows the popularity of variable web-browsers and operating systems linked to search engines or web-pages.

```
          IE6  IE7 IE8 Firefox  Chrome Safari Opera
JUNE 09   15   18   7  47       6      3       2.1
MAY  09   14   21   5  47       5      3       2.2
JUNE 08   27   27   -  41       -      2.6     1.7

         Platform Statistics
         WinXP Win2K  Win7  Vista  W2003  Linux Mac
JUNE 09  67    1      1.6   18.3   1.7    4.2   5.9
JUNE 08  74    2.6    -     10.0   1.9    3.7   4.8


Source http://www.w3schools.com
```

Figure 10: Web browser statistics

Web pages

The structure of a web-page is analyzed in Figure 11. Data from 1995 and more recently from 2008 are shown. The average page size has increased almost ten-fold within a 10-year period. Web-pages used to be text-based; most of the content right now is graphics. The text content (HTML) has not increased dramatically; if one considers scripting a form of text, this doubles the text content of an average web-page.

The useful indexable content of a web-page that is the the number of links per page has gone up significantly from 8 to more than 40 (10 external, and 30+ internal links).

When Google was introduced back in 1998, it was to index close to 26,000,000 web-pages. By 2003 this number has gone up to almost 20,000,000,000, a thousand-fold increase. In 2008, Google's claim to be retrieving close to a trillion of URLs. Not all of them are unique, nor all of them are indexed, however.

In 1998, the 26 million Google pages had an overall size of 150GB uncompressed or roughly 7KB each. By 2003, the 19 billion indexed pages have a volume of close to 167TB, or around 8KB each on the average. This is what is known as the Surface Web. What is hidden from a Web-crawler is information whose size is many times greater. The Deep Web is estimated to be 500x larger, the Email volume almost 3000x larger, and the instant-messaging generated volume is more than that of Surface Web. Thus a great volume of information can and has been indexed but an even greater volume of web-based information is almost inaccessible for a variety reasons to indexing.

Figure 11 gives information about the contents of the average web-page. Figure 12 provides a distribution of web-pages based on file format.

	1995	2008
Average page size	18.7Kbytes	130K
HTML		25K
GIF		3K
JPEG		12K
PNG		15K
SWF		32K
Script size		11K
Style size		17K
Number of links per page	> 8	10 external+31 internal links
		474 words
		281 HTML tags

	2003	2008
Surface Web	167Tbytes	
Deep Web	91850Tbytes	
Email	440606Tbytes	
Instant messaging	274Tbytes	
Web size	19bn docs	1000bn (Google) *

* Unique URLs not necessarily all indexed

Figure 11: Web-page content statistics, Web-size (information)

IMAGES	23%
HTML	18%
PHP	13%
PDF	9%
Movies	4%
Compressed	4%
Executables	1.4%
Powerpoint	0.8%
Word	0.4%
Text	0.1%
Java	0.1%

Source: http://www.sims.berkeley.edu/research/projects/how-much-info-2003/internet.htm

Figure 12: Web-page file format distribution

Figure 13 summarizes Figure 11, Figure 8 and Figure 9 findings. For example in 2003 the two major search engines Google and Yahoo! crawled roughly 20 billion pages each. Several of these pages were duplicates, other were not searchable (e.g. executable files). The total number of available web-sites was estimated to 70 million. Thus on the average < 300 pages were indexed from each web-site. Only 20% of this volume was further indexed.

Web sites	70mln	Webpages/site	270

	Crawled Volume (pages)		
	1998	2003	2008
Google	25mi	20bn	1000bn
Yahoo!		19bn	
BBC news	320mln		

	Indexed Volume (unique pages)	
Google	2000	1.0bn
	2001	1.3bn
	2002	2.0bn
	2003	3.0bn
	2004	3.3bn
	2005	8.0bn
	2006	25.bn

Figure 13: Search Engine Volume: Crawled vs Indexed pages


```
  ??  Dewey System
1950  IR
1963  IR/ G. Salton Work / Vector Space model
1971  SMART system
198x  Lexis Nexis other similar systems
1990  Archie
1991  Gopher
1992  Veronica
1993  Wonderer (first crawler)
1994  Lycos , Excite, Infoseek
1994  Yahoo! directory
1995  Meta Crawler (mete search engine), Altavista
1998  [Sep 7]  Google (25 million pages indexed)
1998  [Oct 7]  10,000 queries answered that day by Google
2000  Teoma  ; Google indexes 1 billion pages
2001  [Jan] Google answers 100,000,000 queries
2004  Google IPO ; Google indexes approx. 4 billion pages.
2005  Google indexes 1 billion images
2006  "to google" becomes a verb in the Oxford English Dictionary
2008+ Google vs Microsoft vs Yahoo
```

Figure 14: A timeline of Information Retrieval and Search engine systems


```
A.  Google's Crawler @ Stanford
                            [circa 1998]                  [2006]
   Active crawlers       :  3-4 typically
   Open Connections      : 300
   Web-page read         : 100 concurrently
   Data                  : 600Kbytes/second
B.  Google@Stanford   size statistics (in millions)
   # of Web pages        : 25 (9 days per crawl)           25000
     (Or just 63 hrs to retrieve 11mil of preexisting pages)
   #  of URLs            : 76
   #  of Email addresses : 1.7
   #  of 404 messages    : 1.6
C. Google Indexer Performance
   Indexer Speed         : 54 pages/second
   Sorter  Speed         : 24 hours on 4 PCs
D.  Google Query Speed   : 1-10 seconds/query
E. Data Collected from Web-pages
   Link Data             : fromURL, toURL, anchor-text
                           [25million URLs, 260mil anchors]
   Short Barrel (4.1GB)  : [title and anchor text only]
   Long  Barrel (37GB)   : [all]
F. Google Index Data
   Fetched Information   : 147Gbytes                       5000Gbytes
   Compressed            :  53Gbytes (3 to 1)
   Inverted Index Full   :  41Gbytes
       Short Barrel      :    4.1Gbytes
       Long  Barrel      :   37.0Gbytes
   Anchors               : 6.6Gbytes [260mil]
   Links                 : 3.9Gbytes
   Lexicon               :  .3Gbytes [14mil words]
   DocIndex              : 9.7Gbytes
G. Source:  Brin & Page [ http://www-db.stanford.edu/~backrub/google.html ]
```

Figure 15: Search Engine Statistics: Google

Practice makes perfect

Exercise 1 *Download to AFS or your own PC/laptop one Web crawler such as WebSphinx (there is a version of it available in Java as a .jar file). Google to find out where it is. Read the online available manual and run the program. If you have never run a java program that's fine; under afs do a* `java -help | more` *for information.*

When you manage to have it running, go and grab first all the text files at
`http://www.cs.njit.edu/~alexg/courses/cs345/handouts.html`

If you feel more confident, try
`http://www.cs.njit.edu/~alexg/courses/cs345/index.html`

Be considerate, careful, and restrain yourself. Do such testing late in the afternoon or early in the morning.

Exercise 2 *Read the manuals/paper introduction/documentation of 3 web-crawlers and do a comparison of their capabilities. Be brief, tabulate your results and restrict yourselves to an one page summary.*

Exercise 3 *Explore the capabilities of Webcrawler and wget beyond Homework 2.*

