Web Search Engine Architecture

Overview

Chapter 1,2,3.1-3.4,4.2.2,4.3,4.4,5.6.3,8.4

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 2016 ; distribution outside this group of students is NOT allowed.*

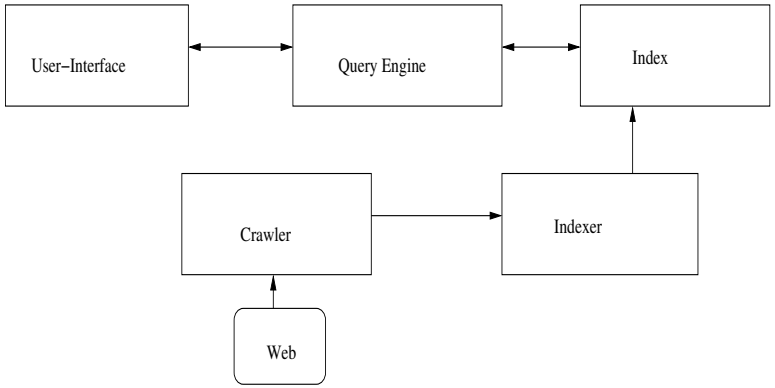


Figure 1: A centralized architecture of a Web search engine.

We introduce in this subject **the architecture of a web 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.

In a centralized architecture such as that found in the Altavista search engine shown in Figure 1, 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.

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 1. This is Homework 6.

Later on, in Subject 3, we 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.

(c) Copyright A. Gerbessiotis. All rights reserved.

The first web search engines such as Excite (1994), InfoSeek (1994), Altavista(1995) employed primarily Information Retrieval (IR) principles and techniques on web available documents. Such search engines were evaluating the similarity of a given query q to a given web document d_j of a corpus of web-documents retrieved from the Web. That process was repeated for all documents d_j, $\forall j$.

The query q itself was being treated as a "small document" consisting of the query terms input by a customer/user and thus the **similarity** $s(q, d_j)$ was established based on IR principles/measures between two documents, one very small, the user query and the other a document d_j of the collection. For a specific d_j this determined a **"rank"** for d_j relative to query q, the similarity measure $s(q, d_j)$ that establishes how relevant is d_j to query q. Repeating this process for all documents of the corpus and then ordering them for decreasing $s(q, d_j)$ provided the output to user query.

However Yahoo! (1995), Google (1998), Teoma (2000), Bing (2006) differed from those early approaches in the sense that link information (e.g. the Web structure and link information of Web-based documents) was used to determine the **linkrank 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 became a weighed sum of $s(q, d_j)$ and $r(d_j)$ plus other additional (eg advertising) 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.

```
s(q , dj)     : Similarity (i.e. information retrieval based ) measure between query q and document dj
r(dj)         : static linkrank of document dj highlighting importance of dj in corpus
R(q,  dj)     : Rank of dj relative to query q
  R(q,dj)  =   weight1 *  s(q,dj)  + weight2 * r(dj)   , 0 <=  s(q,dj) , r(dj) <= 1
                                                       , weight1>0, weight2>0,  weight1 + weight2 = 1.0
```

Criteria. Any search engine architecture must satisfy two major criteria listed below. 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.

- **A. Effectiveness (Quality)** that will satisfy the relevance criterion.
- **B. 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.

(c) Copyright A. Gerbessiotis. All rights reserved.

High Level Phases and Steps.

Four high-level phases can be used to describe the operations of a Web Search Engine Architecture: **Search, Sort, Select** and **Query**. **Search** means collecting information from the Web, information that is indexable and in the form of a Web document. **Sort** means processing this information and building the data structure known as the Index. **Query** is the interface provided to external users who type queries to the web search engine in order to get (retrieve) those documents (most) relevant to the query. **Select** decides the most relevant queries to the documents i.e. establishes $s(q, d_j), r(d_j)$ and $R(q, d_j)$ and then selects the higher "ranked" among them.

S1. Modeling: Step 1. The first such step is `Modeling`, a preprocessing step. This step tries to effectively utilize all compute and storage infrastructure effectively. There is no point fetching pages that the architecture does not know how to index or have no resources to store. With reference to Figure 1 the modeling deals with the interaction between the component known as the `Crawler` and the component known as the `Web` i.e. the two bottom left boxes of the figure. The `Crawler` is the program that will crawl the web and find and fetch web-pages. The modeling established what will be fetched and from where and where this information will be subsquently stored and in what format and by what resources. How this information (eg web-page) fetched, will be fed to the **Indexer** that will process it and subsequently build an **Index** out of them and how and where this will be stored, and whether enough resources are available for this processing and storage. Note the difference between Indexer, a program or set of programs, and the Index, a data-structure or a collection of data-structures. The wed search engine user is oblivious to the Crawler or Indexer since the user interacts through the User-Interface (aka the search box or text box of an engine such as Google or Bing) with the Query Engine. The Query Engine interacts with the Index. When we perform a "web search" we never search the web. The query engine searches a static representation of the Web, previously fetched by the Crawler, and built by the Indexer and this representation is known as the Index. The Index allows for an efficient search of the Web without accessing the Web itself! How the resources available will be split among the Query Engine, the Index, and the Crawler needs to be established here as well!

`Modeling` involves the definition of the **the Corpus** and the **text model** that will be used. The `document model`, the collection of Web documents that will be crawled and processed (i.e. indexed) based on their extensions (.html vs .txt vs .pptx vs .pdf) is also decided, along with languages supported and whether compression and what protocols are to be used or not. All these would depend on the capabilities of the web search engine (can it index .pdf documents?) and resources available (compute and storage power).

(c) Copyright A. Gerbessiotis. All rights reserved.

S1.A. Modeling: Text-model (language and character encoding, compression) and protocols. This is where the corpus is defined, i.e. the collection of (Web) documents that will form the corpus. First the **text-model** is decided and the languages that will be supported established (eg English-only and what variants of it, or also foreign languages). In American English the word `billion` means one thousand million but in (British) English it means one million million. What language encoding will then be supported and also internally used consistently (eg ASCII vs Unicode vs UTF-8 or not) is decided then. At this time it is also decided how compression will be dealt with (e.g. .zip, .tgz, .gz) and which other protocols are to be supported or not (php, paywalls).

S1.B. Modeling: Document-model (file extensions, multimedia handling). 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 i.e. **multimedia handling** (video, images, etc): just retrieved, file-names or tags indexed, or contents searched and processed. 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.C. Modeling: Repository and its organization and other storage requirements. 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. Metadata information collected related to these documents will be processes, 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. All data structures (eg Index) that will be built or intermediate structures built and used need to be stored as well. This generates additional storage requirements. One cannot afford to run out of space (hard disk drives) or resources during the cycles or phases of building a web search engine. One needs to resolve these issues first. The availability of resources determines what will be crawled, indexed, and thus queried!

(c) Copyright A. Gerbessiotis. All rights reserved.

Step 2: Crawler. Crawlers are software agents (programs) 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 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.A. Selection Policy. Step S1.A and S1.B in fact determine the selection policy i.e. what fraction of web available documents are to be crawled and then processed (indexed) and subsequently made available for query/"searching". Thus the implementation of the preprocessing modeling is realized through the selection policy of the crawler. Moreover, the crawler needs to determine the location of the information that will be retrieved. Every IP address does not correspond to a web server and a web server might serve multiple domains. For example, in 2009, out of a universe of 4 billion IP addresses (IPv4) only a sixth or so were registered and by now this number is a least a quarter or more. 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).

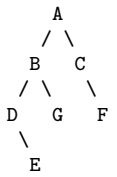
S2.A.1 Volume of Information. In 1994 it was estimated that there were approximately 100,000 web sites. By 1997 this number grew to roughly 100,000,000. By the end of 2015 there might be 1 billion web-sites, one trillion potential URLs but not substantially more than 30-50 billion indexed web-pages. 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.

(c) Copyright A. Gerbessiotis. All rights reserved.

S2.B. 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?)

We have already mentioned that **URL** means **Uniform Resource Locator** and describes the location of a document on the Web and information on how to access it. The crawler operates on URLs. The simplest way to understand its operation is the following. Start with one URL, access the resource it points to, i.e. a web-page, and then parse that web-page and extract its hyperlinks and convert them into URLs. Those extracted URLs are then crawled similarly. Which one of multiple URLs is to be crawled and for how long determines whether dfs or bfs is to be used. If one of multiple URLs of a page is being pursued until it reaches a dead-end (a "sink") then dfs has been used. If otherwise a level-by-level crawling of URLs is pursued, a bfs has been used.

Consider the example below. Web-page A contains two hyperlinks to web-pages B and C. B contains a hyperlink to D and C to F, and D to E. Starting with a node say A (see example), BFS first visits the pages at distance one from A, i.e. B and C. Then it visits those at distance one from B and C i.e. distance two from A and these are D and F respectively (order between the two is arbitrary). Then the only page at distance three from A which is E. In DFS, one of the links of A say B is pursued to the end. This means the link to D is then followed and then its link to E. Page E is a sink and given D has no other links a backtracking to B occurs to pursue G. The further backtracking goes to A and the thread starting from C is explored.




```
BFS:   A B C D G F E
DFS:   A B D E G C F
```

S2.B.1 URL server. Sometimes a crawler that has been in use for sometime is utilizing a URL server that can supply it with a preexisting set of URLs as they were established from previous crawls. The representation of the graph of the Web is more or less complete and few extra (but thousands or millions) links pop up from one crawl to the next. Someone can do the dfs or bfs ordering offline and present the resulting solution in batches through the URL server (think of the URL server sending to the crawler first `A B C D` and then `G F E`). Crawler utilization is higher and cpu usage lower this way as the crawler does not extract links any more.

(c) Copyright A. Gerbessiotis. All rights reserved.

S2.B.2 Refresh rate and other scheduling issues. The **refresh rate** determines how often a given page will be re-crawled and this depends on how often contents of it changes through time. Other issues involved in crawling is what time to schedule a crawl: night hours vs day hours for example. But consider a French audience page hosted on a domain name or host that is located in the USA or China. What is night and what is day? Sometimes such decision are affected by the observance policy (to be discussed) of a web-site. Metadata collected in previous crawls (eg time of last modification of a file) can affect such decisions as well. Around 1998, crawlers used to index several million pages a day; this number has grown to several billion by 2015. A crawl of the web used to take a week or more and indexing a month or more. These numbers are better nowadays by at least an order of magnitude.

S2.B.3 Information Collected. The crawler collect and maintain `metainformation` about the crawled URL. This metainformation or **metadata** includes beyond the obvious (URL, name of file), the language and 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 cryptographic hash/signature of the document, etc.

S2.B.4 URLs and docIDs. A web-page first crawled might be assigned a unique ID usually called `docID` for **document id**. The docID relates not to its name or its contents but to its location. Subsequent crawls that retrieve the same or updated versions of the document may use the same docID of that URL. A docID may be a "cryptographic hash" (unique fingerprint) of the document's URL. More often however the `docID` comes from a predetermined range however. A docID used to be a 32-bit word (eg. Google up to around 2004) but now uses more than four bytes. It is easier to reference a document by its 4- or 8-byte docID than say its variable length URL than can exceed 50 characters (50 or 100 bytes if it includes non English characters). Thus less space is used for storing docIDs than URLs. The docID value 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) and one does not need an extra field (space) for the rank.

S2.B.5 Duplicates. The crawler sometimes needs to determine whether a page is a duplicate of an existing page of the same or other domain. 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 and under different URLs; there is no reason to process all of them separately since processing one web-page suffices.

(c) Copyright A. Gerbessiotis.

S2.C. 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 2.

```
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 2: A `robots.txt` file

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. 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.

(c) Copyright A. Gerbessiotis. All rights reserved.

S2.D. Parallelization or coordination/synchronization policy of the crawler. Because of the vastness of the Web, there is no such thing as a single crawler (program). There are multiple crawlers i.e. multiple processes running multiple threads of crawling each directed to a different set of URLs or a different set of domains or different geographic regions. In such cases their activity needs to be properly synchronized. A URL server might serve such purpose.

S2.1. Examples 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.

S2.2. Deep Web interaction. In several cases the Deep Web may interact with a crawler by feeding it information that has just been created and is behind a paywall. 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. One can also search a page behind a paywall but in order to access it one needs to still be a subscriber to it.

(c) Copyright A. Gerbessiotis. All rights reserved.

S3. The indexer and the indexing process. The **indexer** is the collection of programs that generates the data structures known collectively as the index in Step 3 of the Web Search Engine Architecture design process. This process that involves the "indexer" relates to the execution of multiple phases that have different names (such as parsing, linguistic analysis, source file based transformations, indexing) and is also known collectively as the **indexing process**. The input to the `indexer` is the `Corpus`, the collection of web pages fetched from the Web by the crawler and then stored locally possibly in a `Repository`: the indexer usually receives them through the Repository rather than directly from the Web. The output of the indexer is the **(inverted) index**. There is a reason for using a `Repository` rather than a direct line of communication between the Crawler and the Indexer that feeds pages from the Crawler directly to the Indexer: efficiency and the fact that the Crawler might be working on one copy (the newer) of the Web and the Indexer might be working on another (the older) copy of the Web at the same time. This abstraction (two copies worked out simultaneously) was used by Goodle through around 2008 or so. Efficiency results from the fact that the Indexer does not wait for a page to be fetched from the Web by the crawler: the web-server hosting it might not be available.

S3.1 Phases in building the index and intermediate data structures. The `index` (also known as `inverted index`, `inverted file`) is *an efficient data structure that represents the documents of a corpus and allows for fast searching of the corpus documents using that indexed information, and not the documents themselves.*

The building of the the index involves document processing in a series of text operations or transformations. These include Phase A when parsing takes place in the form also known as tokenization or lexical analysis. Then a linguisitic analysis is pursued in Phase B, and Phase B′, if available, involves source file-based tranformations used for auxiliary purposes to extract link information. Thus far the indexer first creates an intermediate data structure that is known as the **forward index**. Subsequently in Phase C Indexing, the forward index is **inverted** by the indexer using a sorting operation. The output of this inversion is then used to build the **index**.

- **Phase A. Parsing (or Tokenization or Lexical Analysis)**,
- **Phase B. Linguistic Analysis**,
- **Phase B′. Source file-based tranformations**,
- **Phase C. Indexing** .

(c) Copyright A. Gerbessiotis.

Figure 3 outlines the various phases (A,B,C) involved in Step 3, the Indexing Process.

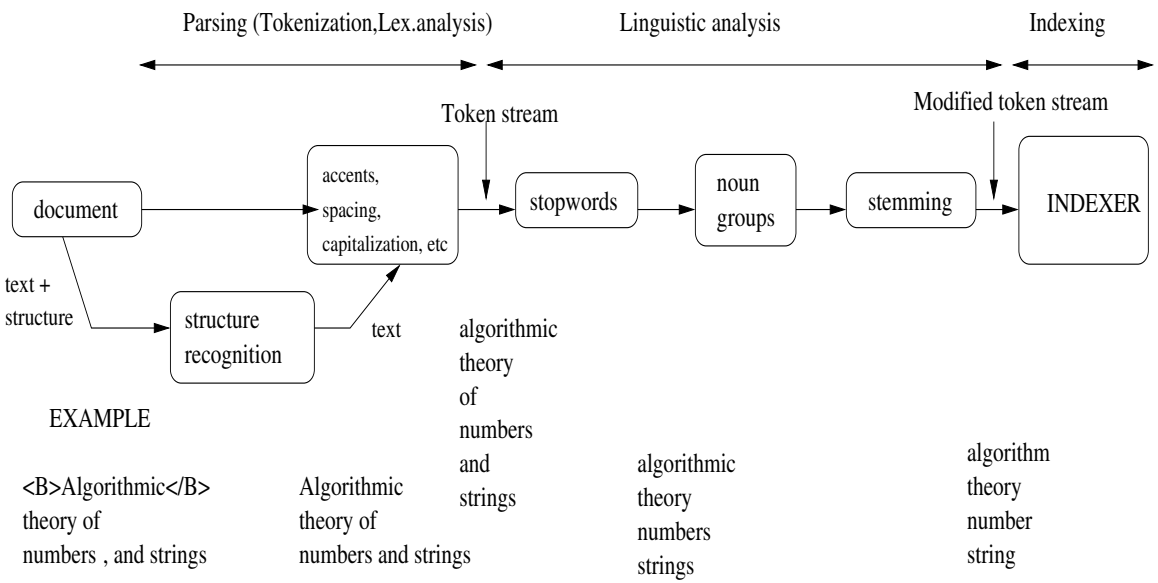


Figure 3: Indexing Process

S3.A Parsing. The first phase known as **parsing** is also known with alternative names such as **lexical analysis** or **tokenization** and it involves the tokenization of the contents of a (Web) document made available by the crawler to the indexer (through the Repository). This phase converts the character stream of a document into **tokens** (a `word`, a `number`, a `date` forms a token). The objective is to determine which ones of the tokens can serve as keywords and finally to generate **index-terms** out of them. It is the **index-terms** that will be stored in the index, not (necessarily) ordinary words or more other tokens. The notion of a `keyword` is more *abstract*: it identifies the subset of the words/tokens that are considered important in a document. But not all such words might directly become index-terms. Parsing also includes a number of processing steps that resolve other issues such as treatment of numbers, dates, case (upper vs lower), accents and punctuation, fonts.

As a result of this phase the tokenizer returns a stream in the form of one tuple per token retrieved. Moreoever metadata information is also returned as well.

Furthermore before this first phase starts some preliminary work needs to be performed to determine what constitute a document, what encoding is being used (ASCII, Unicode, UTF-8), what documents are to be parsed, what natural language (English vs other Latin languages vs Germanic languages vs Greek vs Eastern European vs Middle-Eastern with right-to-left parsing or Asian languages) they use and whether the Indexer is capable of indexing. As Web documents are primarily HTML document is an HTML extension file treated differently and in more detailed fashion? An HTML parser can isolate words in say title information recognizing tags such as `<TITLE>` and offsets of words inside them. Font-Type Info and Size is then noted (e.g. text is `between <B> and </B>`) or what the font-size is relative to the default one of the page, header text info is recorded (`H1,H2,H3 H4,H5,H6`), list info recognized `UL, OL, DL, LI` and URL are properly attributed to Anchor text and linked page, rather than the page that contains it. How one can parse data from metadata if they are both available in the same document, how compressed documents will be treated?

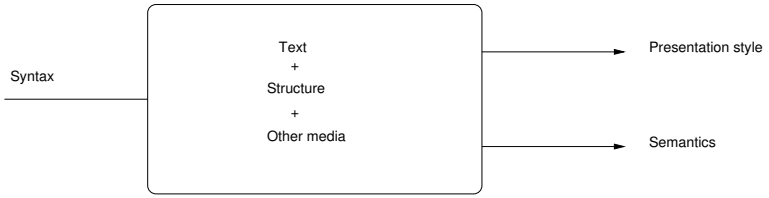


Figure 4: Indexing Process

S3A.0.1 Document Syntax, Structure, Presentation Style and Semantics. Every document and this includes web-based documents has some

- **structure** that can be visible (body of the document that includes its text) and invisible (DTD i.e. Document Type Declaration).
- **syntax** that expresses its structure and affects its semantics. (For example = or == or := might be used to denote an assignment or equation testing, as require.) Syntax can be explicit (HTML or XML) or implicit (English language).
- **semantics** that expresses the meaning of a specific syntax structure. (The presence of an = sign in Pascal indicates equation testing; but in C or C++ is an assignment.)
- **presentation style** of a document indicates how it looks like if displayed. For a web-document its presentation is described in a Cascading Styled Sheet (CSS) but the document contents and its structure are written in a markup language such as HTML.

S3A.0.2 Formats of 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.

(c) Copyright A. Gerbessiotis. All rights reserved.

S3A.0.3 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. SGML (Standard Generalized Markup Language, ISO 8879) developed by Goldfarb is such an example and instances of it include HTML, XML, Latex/Tex documents/files. HTML files can be easily recognized by their file names which have an `.html` or `.htm` suffix, XML have `.xml`, RTF and SGML files use `.rtf` and `.sgml`. The markup declarations of such a language can be defined in a DTD (Document Type Declaration) schema or not. 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. TeX/LaTeX are used for typesetting scientific text. The marks are called tags (similarly to HTML) and initial `\begin{document}` and ending `\end{document}` tags surround the marked text. TeX/LaTeX documents share a `.tex` suffix in their filenames. One might have to parse the document itself to determine whether it is a LaTeX or a TeX document. All this information needs to be taken into consideration when the Web Search Engine Architecture's components are designed and implemented.

- HTML (Hyper Text Markup Language, as of August 2016 is in version 5.0) is an instance of SGML, Some limitations of HTML include:
  - no validation of data/text, nuser-specified tags,
  - no support for nested structures to represent DataBase schemas.
- CSS (Cascading Style Sheets) added to HTML in 1997, to describe the presentation style of an HTML document,
- XML (Extensible Markup Language) is a simplified subset of SGML,
  - there is an XSL, the counterpart of CSS,
  - 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,
  - metadata info is in RDF format (Resource Description Format).

S3A.0.4 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" >
```

S3A.0.5 HTML vs XML. 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 5: HTML vs XML: An example

S3A. Parsing. After the type of the document has been resolved, its language and encoding, and other issues already highlighted in the preliminaries of this Step (and its Phases), the parsing (Phase A) of a document is to commence. In fact before Phase A starts one needs to decide the treatment of numbers, dates, case (upper vs lower), accents and punctuation, fonts.

- **S3A.1.1 Token.** Is a token a sequence of alphanumeric characters separated by white space (space characters, end-of-line symbols, tabs, etc)? Is order of parsing as recognized earlier, important (eg. left-to-right or right-to-left).
- **S3A.1.2 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 ?
- **S3A.1.3 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?
- **S3A.1.4 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?
- **S3A.1.5 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`?
- **S3A.1.6 Acronyms.** How does one handle XP, NT, WWII, CAT (the feline) or CAT (for Caterpillar)?
- **S3A.1.7 Phrases.** Is it `on-line` or `online` ? Is United States two words or one? But how can one distinguish between **The coach of Manchester United states ...** and a sentence **United States**). Does the **United states** the same as the latter's `United States`? Spelling error vs capitalization?
- **S3A.1.8 Accents and punctuation marks.** Is resume a verb or does it mean 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` ?

(c) Copyright A. Gerbessiotis.

S3B. Linguistic Analysis. This phase is applied to the output of parsing Language-based text transformations are applied to generate the index-terms. The set of text transformations involve the application of a list of `stopwords`, `stemming`, that would reduce the number of tokens that will eventually become index-terms. Sometimes an index-term is a token (word) that does not appear per se in the text or it appears there in a completely different form (with a suffix). For example with reference to Figure 3, `algorithm` is an index-term, although the word `algorithm` never appears in the text: the input document contains only the word `algorithmic`.

- **S3B.1.1 Stopwords.** Frequent not very useful tokens or words are poor index-terms and bloat (increase the size of) indexes. We call them **stopwords**. These include articles and connectives. If space requirements are not an issue then one can just ignore stopwords. A list of stopwords is shown below

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

  Note that or is not a stopword. OR is an operator and more over an acronym for the state of Oregon, Operations Research, Operating Room, Outdoor Research, etc. What do you feel about `us`?
- **S3B.1.2 Short Words.** Are there short words that are important ? For example `xp, nt` might become important in the context of `Windows XP` or `Windows NT`. Or `US` or `WWII`.
- **S3B.1.3 English language variations: American English vs (British) English vs** .... Or how many millions in a billion ? In English a billion is a million millions (our trillion or 10^{12}) but in (American) English it is a thousand million (10^9).
- **S3B.1.4 Stemming.** A stem is the substring of the word left after the removal of its affixes (prefix and suffix). A stemming operation attempts to reduce distinct words to their common grammatical root. The operation known as stemming reduces the number of distinct words to their common grammatical root and thus `fish` and `fishes` and the occasional `fishing` (sic) will generate one index-term if stemming is used. There have always been arguments in favor or not (e.g. Harman(1991), Frakes (1992)). Algorithms for stemming include Porter's and Harman's with the latter being described on the following page. A third widely used algorithm is an one by Krovetz. Porter observed that his stemmer can reduce the number of tokens by 30% or so. The textbook on Page 92 has an incomplete description of Porter's method.

(c) Copyright A. Gerbessiotis.

```
 HarmanStemmingAlgorithm(word)
1. If word ends in -ies but not -eies or -aies
    then -ies --> -y;
2. If word ends in -es  but not -aes, -ees or -oes
    then -es  --> -e;
3. If word ends in -s but not -us or -ss
    then -s   --> -;
```

```
Example Question
****************
How does it stem  tries, retrieves, foxes, dies?
```

- **S3B.1.5 Noun groups** How many index-terms in `kitchen table`). How about `down under` ? **Hint: Australia (for the latter).**
- **S3B.1.6 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? How about `metro` vs `underground` vs `subway`?
- **S3B.1.7 Acronyms.** Assuming or is not a stopword is OR the state of Oregon, Operations Research, Outdoor Research, or an Operating Room or all of the above? Or what else can it be?
- **S3B.1.8 Lemmatization.** It refers to the grouping of different inflected forms of a word (e.g. `car, cars, car's, cars'`, and also `good, better, best`).
- **S3B.1.9 Spelling.** British vs American. Is `center` a `centre` ? Or the more challenging `towards` vs `toward`. Is one of them a typo? Do we apply lemmatization?
- **S3B.1.10 Typing Errors.** Do you correct them ? How does one handle them? Usually at query time one can correct spelling by checking each word against the lexicon for distance editing (definition to follow) of at most 2-3.
- **S3B.1.11 Thesaurus use.** Consider the case of a `thesaurus` for different subjects that determines the list of index-terms as a subset of the tokens identified so far. Most search engines assume that the set of all tokens of all documents (after stemming and stopword elimination) is the thesaurus. Identify synonyms, words with similar meaning and search for all of them if one is in the query list.
- **S3B.1.12 Temporal (i.e. time) factors.** How would you have treated a 9/11 on September 1, 2001?

(c) Copyright A. Gerbessiotis. All rights reserved.

Note 1. The parser should also be capable of accommodating errors including syntax errors and thus be able to deal with free-form text in which no specific syntax is implied. Thus if there is a syntax and semantics implied in the text the parser should be able to take advantage of this additional information. In the absence of such information it should also be able to work properly and miss as few words as possible, since a word missed might be an index-term missed after all.

Note 2. Hamming distance of two strings. Two documents or two words can be compared to determine their similarity. One way to measure similarity is by using a **distance function**. The Hamming distance of say two strings of the same length is the number of positions in which they are different. Distances satisfy the triangular inequality $d(a, c) \leq d(a, b) + d(b, c)$.

Note 3. Levenshtein distance. The **edit** or **Levenshtein** distance is the minimum number of character insertions, deletions, and substitutions we need to perform in any of the strings to make them equal. The edit distance of `center` and `centre` is two and of `color` and `colour` is one.

Note 4. Longest Common sequence. Another measure is the longest common sequence (deletion of characters is only allowed). For example `color` and `colour` have `color` whereas `survey` and `subway` have `suy`.

Note 5. N-grams. An n-gram is a sequence of n words that have a special meaning if they occur together. According to the textbook (page 101) the following data related to n-grams can be obtained from Google's text collection.

```
          Google Text  Collection
Number of words   n   :  1   trillion
Number of 1-grams     : 13   million
Number of 2-grams     : 314  million
Number of 3-grams     : 977  million
Number of 4-grams     : 1.3  billion
Number of 5-grams     : 1.17 billion
```

Figure 6: Google-reported n-gram information

Most frequent ENGLISH phrase (or 3-gram) is: `All rights reserved`.

Most frequent CHINESE phrase (or 3-gram) is: `Limited Liability Corporation` (LLC).

Step 3 Phase B′ : Source-file based transformations

Just after Phase B has been completed, a **semantics-based classification** of a Web-page might also take place that characterizes it for example as `sports`, `politics`, `academic`, etc. Certain source file text transformations also take place.

S3B′. Source file-based transformations. Links and anchor text from web-pages can also be identified and extracted. The example 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 through its anchor text. 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`. The link from `index.html` to `paper1.pdf` and the anchor text related to the latter can be exploited by certain web search engines' 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 rather than `index.html`. This information must be stored or temporarily saved somewhere and used when the indexing of `paper1.pdf , paper2.pdf` actually takes place!

Moreover for both `paper1.pdf` and `paper2.pdf` their complete URL information needs to be reconstructed from the incomplete link information of `index.html` and temporarily saved somewhere. Such URLs might never have been crawled already and thus the crawler needs to be told so by the Indexer!

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

```
<!-- 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>
```

(c) Copyright A. Gerbessiotis. All rights reserved.

As soon as Phases A and B are completed the next Phase C involves the actual build-up of the index. As already discussed in `S3.` and `S3.1` the output of Phase B is a `forward index` that is being used as input to Phase C to generated the `(inverted) index`.

S3C.1 Forward index. For every document in the collection, the indexer generates by the completion of Phase B a sequence of tuples that describe that document. Each such tuple contains information about the document (docID), the index-terms encountered (wordID), an offset indicating relative position of the index-term in the document encountered (mostly, word offset), and some context information about the index-term (eg. 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). A representation of data in the forward index is in the form of tuples sorted by `docID` and within a given docID by `wordoffset`. The tuples look as follows:
`( docID, wordID, wordoffset, context )`

S3C.2 (Inverted) Index. A sorting operation is then applied to the forward index to prepare for the generation of the `(inverted) index` also known as `index`. This inversion (sorting process) would sort the forward index tuples by wordID first, then (for same wordID tuples) by docID, and then (for same wordID and docID tuples) by offset/context. This ways we can group tuples based on `wordID`. This way the definition of **index** can be refined to mean *that for a given index-term it is the list of documents (docIDs/URLs) that contain that index-term.*

S3C.3 Why (Inverted)? The index 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.)

(c) Copyright A. Gerbessiotis.

S3C.4 Auxiliary structures. Both the forward and the inverted index deal with `wordID`s and also `docIDs`. There must be a mechanism that maps `wordID` into `index-term`s and also `docIDs` into URLs or the other way around.

S3C.4.1 Mapping a `URL` to a `docID` or the other way around. Moreover another auxiliary data-structure should allow us to retrieve URLs or `docID`s from the other. 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 thousand times or more in the index. To facilitate efficient look-up we replace it with a shorter numeric `docID`. In addition we might maintain the "cryptographic hash" of the URL of the document: the "URLsum" 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 maintained such as its type, size, whether it is compressed or not, encoding (eg. 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 "URLsum" or `docID` that includes also a pointer to the location storing the URL itself plus the metadata information, and (b) a sorted table by "URLsum" that contains fixed-length records containing the "URLsum" and `docID`. That latter table might be used once to determine whether a given URL (through its "URLsum") exists or not and what its `docID` is! Metadata of such a data-structure can also be accessed by the Crawler: the last fetch of the web-page with a given URL might get recorded as well as the date of the next fetch or cryptographic fingerprints of the last two or three fetches to determine whether the URL has changed in content or not.

S3C.4.2 Mapping a `wordID` to an index-term. This forms an auxiliary data-structure that is part of the index and used both by the forward index and the inverted index; in fact this data structure is created at forward index creation time. Note that the indexing process before Phase C or after Phase B does not deal any more with tokens but with `index-terms`. The set of all the index-terms that are 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`. 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 `''Wordsum''`. The "Wordsum" 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.

(c) Copyright A. Gerbessiotis. All rights reserved.

S3C.4.3 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; **t** is the number of distinct index-terms, and **T** the total number of index-terms.
- n_i is the number of distinct documents containing index-term k_i, $i = 1, \ldots, t$.
- **Term frequencies** tf_i **,** tf_{ij}, f_j 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, (c) the number of index-terms of 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. Here $\lg x$ is the logarithm of x base two, i.e. $2^{\lg x} = x$. For example $\lg 8 = 3$ since $2^3 = 8$.

S3C.5 Index Distribution. The data structures used in the indexing process (including the forward and inverted index) are potentially larger in size than the input, the corpus. For web searching 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.

S3C.6 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.

(c) Copyright A. Gerbessiotis. All rights reserved.

S3C.7 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).

```
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    qudruplet  (docID, wordID, wordoffset, context ) or a
                        triplet  (docID, wordID, wordoffset )          : We use this one for the example

Forward Index                                                          Phases A and B combined.
docID=1 Algorithm-> algorithm, offset = 1  ->      (1,1,1)
        data                 , offset = 2  ->      (1,2,2)
        structure            , offset = 3  ->      (1,3,3)
        alex                 , offset = 4  ->      (1,4,4)  forward index
docID=2 algorithms->algorithm, offset = 1  ->      (2,1,1)
        Data      ->data     , offset = 2  ->      (2,2,2)
        alex                 , offset = 3  ->      (2,4,3)
        structure            , offset = 4  ->      (2,3,4)
docID=3 structures->structure, offset = 1  ->      (3,3,1)
        Alex      ->alex     , offset = 2  ->      (3,4,2)
        data                 , offset = 3  ->      (3,2,3)
        alex                 , offset = 4  ->      (3,4,4)
```

(c) Copyright A. Gerbessiotis.

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                                 Inverted Index                Phase C
(1,1,1)                                         (1,1,1)
(1,2,2)                                         (2,1,1)
(1,3,3)                                         (1,2,2)
(1,4,4)                                         (2,2,1)                     Stage 1: Sorting
(2,1,1)    ---- Group by or Sort by -------->    (3,2,3)
(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)
                                                (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)                     Stage 2:  Grouping by wordID
                                                (1,4,4)
                                                (2,4,3)
                                                (3,4,2)
 DocList :  1                                   (3,4,4)
**********  2
            3
 Vocabulary (sorted by #word)                 Lexicon  (''Hashed'';no ordering)
 #word      #wordID  #hash? #other            #word    #wordID
 alex       4                                 data      2
 algorithm  1                                 algorithm 1
 data       2                                 structure 3
 structure  3                                 alex      4
```

(c) Copyright A. Gerbessiotis.

S3C.8 MAP-REDUCE on the forward index. In the past Google was using a distributed and parallel computing system known as MAP-REDUCE to performa a variety of operations during the indexing process and afterwards. We show with an example how one can utilize MAP-REDUCE to collect information statistics such as those shown in S3C.4.3 out of the forward index or the inverted index. The stream of the forward or inverted index that is generated by billions of documents include hundreds of billions of tuples. Generating them by a single computer is inefficient. Hundreds of thousands of computers need to coordinate their activities to produce the desired effect. The program that coordinates the activities of those computers, provide file-system access methods to deal with large swaths of data, and a programming abstraction that make it easier to work is MAP-REDUCE. During the Indexing Process and while the forward index is being formed, several of section S3C.4.3 statistics can be collected. These include the following:

```
n : corpus size    ; T : # index-terms    ;  t  : # (distinct) index-terms of corpus
dj  : j-th document of corpus is dj   ;  ki : i-th index-term
ni  : number of distinct documents containing the i-th index-term ki
tfi : number of occurrences of ki in all documents (duplicates make tfi >= ni)
tfij: number of occurrences of ti in dj only!
 fj : number of terms of d_j i.e. sum over i of tfij
idfi: lg(n/ni)
```

S3C.8.1 Problem 1: Number of documents n of Corpus. In order to compute the number of documents n of the Corpus, we need to find, for each server containing the forward index, the number of documents processed by that server of MAP-REDUCE. Then a sum overall the servers gives the answer.

S3C.8.2 Problem 2: How many index-terms f_j are in d_j? In order to compute the number of index-terms $f_j = \sum_i$ of document d_j, we just perform a local computation in a server for document d_j by counting the tuples of document d_j.

S3C.8.3 Problem 3: How many index-terms T (duplicates ok) in Corpus? In order to compute the number of index-terms T of the Corpus, we need to find the number of tuples per server, and then a sum overall servers gives the answer.

An example of "MAP-REDUCE" operations

Having the forward index in tuple form the three Problems can be solved using linear scan operations as shown.

```
Doc1  : docID=51:   algorithm data structure alex     Triplet of forward index  (docID, wordID, wordoffset )
Doc2  : docID=13:   algoritms data alex structure
Doc3  : docID=22:   structures alex data alex

            STAGE 0: Build forward index (in this simple example, a triplet).
Forward Index
(51,1,1)                           Problem 1     :  Number n of docID. (Per Server and Grand Total n.)
(51,2,2)                             Answer    :  (a)  Scan top-to-bottom record changes of docID.
(51,3,3)                                          (b)  Output total number of changes    (Per Server)
(51,4,4)                                          (c)  Global sum of total over all servers  is n.
(13,1,1)
(13,2,2)                           Problem 2     : How many index-terms fj in dj ?
(13,4,3)                             Answer    :   Count tuples for given dj
(13,3,4)
(22,3,1)                           Problem 3     : How many index-terms T? (Per server and Grand Total T)
(22,4,2)                             Answer    :  (a) Total is number of tuples minus 1 (sentinel discounted)
(22,2,3)                                          (b) Global sum of total over all servers is T.
(22,4,4)
(99,99,99) 'sentinel record all 9 record'
```

(c) Copyright A. Gerbessiotis. All rights reserved.

S3C.9 MAP-REDUCE on the inverted index.

S3C.9.1 Problem 4: How many distinct index-terms t? In order to compute the number of index-terms t of the Corpus, we need to have the forward index sorted by wordID. A change of wordID in the tuple order indicates a new index-term; the total number of changes is t. Then a sum overal servers gives the answer. (Attention: all tuples of a given index-term must reside on a single server.)

S3C.9.2 Problem 5: How many (distinct) documents n_i contain index-term k_i? Assuming the forward index has been sorted by way of Problem 4 (sorted by wordID), in order to compute the number of documents n_i in which a given index-term k_i appears we just perform a local scan per server and count the number of tuples with non-duplicate docIDs, i.e. for a given wordID we count changes of a docID.

S3C.9.3 Problem 6: How many instances of index-term k_i i.e. $\sum_j \mathbf{tf}_{ij}$? In order to compute $\sum_j \mathbf{tf}_{ij}$, we just count the number of tuples for given wordID!

```
Forward Index                 Inverted Index
(51,1,1)                     (51,1,1)      MAP REDUCE SCALING: Global sorting
(51,2,2)                     (13,1,1)                          across all servers
(51,3,3)                     (51,2,2)
(51,4,4)                     (13,2,2)     Problem  4: How many distinct index-term t?
(13,1,1)  - Sort by ---->    (22,2,3)       Answer: (a) Scan per server, record changes in wordID
(13,2,2)     wordID          (51,3,3)               (b) Global sum overall servers gives t
(13,4,3)     docID           (13,3,4)
(13,3,4)     offset          (22,3,1)     Problem  5: How many docs ni contain index-term ki?
(22,3,1)                     (51,4,4)       Answer: For wordID of ki, count non duplicates docIDs
(22,4,2)                     (13,4,3)               (i.e. changes of a docID)
(22,2,3)                     (22,4,2)
(22,4,4)                     (22,4,4)     Problem  6: How many instance of index-term ki? (sum tfij over j)
(99,99,99)                   (99,99,99)     Answer: For a given wordID count tuples inside a server!
```

An example of "MAP-REDUCE" operations

```
            STAGE 2: The (inverted) index
                                    Inverted Index
                   (51,1,1)       1 :  (51,-,1) (22,-,1)
                   (22,1,1)       2 :  (51,-,2) (22,-,2) (13,-,3)
                   (51,2,2)       3 :  (51,-,3) (22,-,4) (13,-,1)
                   (22,2,2)       4 :  (51,-,4) (22,-,3) (13,-,2) (13,-,4)
                   (13,2,3) --->
                   (51,3,3)
                   (22,3,4)
                   (13,3,1)
                   (51,4,4)
                   (22,4,3)
                   (13,4,2)
                   (13,4,4)
 DocList :  51 is Doc1 , 13 is Doc2 ,  22 is Doc3
 ***********************************************
 Vocabulary (Sorted by index-term)              Lexicon  (Hashed for easy look up)
****************************************       ****************************************
 #word      #wordID  #hash? #other              #word    #wordID pointer-to-vocabulary
0 alex       4                                  data      2      2
1 algorithm  1                                  algorithm 1      1
2 data       2                                  structure 3      3
3 structure  3                                  alex      4      0
```

(c) Copyright A. Gerbessiotis.

S4. Query Process and the Query Engine. 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 (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 0.5 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]

S4.1 Query Engine: Query Language. 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 also 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. In web search engines a query involves the typing of `search-terms` into the text box of the search engine.

Step 4: Query Process

When a **query** q is being typed i.e. it is presented to the search engine, it is "parsed" by it and expected to conform to the syntax of the **query language** supported by the search engine. This processing (including results output) is done during the **query process** that involves a program or sets of programs known as the **query engine**. The query language into which queries are to be expressed should be simple and intuitive enough for the average user to use it correctly and effectively. The simplest way to use this query language is by typing words separated by space in the text box of a search engine. Implied in such an input is the assumption that the query language involved interprets this as a logical conjunction with the space indicating the `AND` conjunctive operator and the user words become the terms (arguments) of the conjunction. 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`.

(c) Copyright A. Gerbessiotis.

S4.2 Query engine: Text transformations. The words described in the query are user defined words. One can call them a *query request*. In order for the user to search the index, the user must provide index-terms as only index-terms can be used to search the index-based representation of the Web maintained by the web search engine. Thus a query request must then be translated by the **query engine** into a *query* that is compatible with the search-engine view of the Web. It would not make sense to look for `algorithmic` just because a user typed it into a query box, if all instances of `algorithmic` in the corpus are stemmed into `algorithm` and thus are treated as synonyms of the latter. Most users do not know the difference between a word they type and an index-term. Thus the search engine must apply to the user words the same text operations/transformations that were applied to the documents of the corpus in Phases A and B of Step 3 to convert tokens (or not) into index-terms. Thus the query provided by a user is transformed the same way a document of the corpus is textually transformed. Thus *one can treat a query as a small document*! Moreover, users make mistakes during typing. The words a 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 into index-terms that can be recognized by the indexer (and stored in the index vocabulary). When all these issues have been resolved, 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.

(c) Copyright A. Gerbessiotis. All rights reserved.

S4.3 Query processing: Information retrieval methods for computing $s(q, d_j)$. For a given query q, during the query process for every d_j of the corpus the query engine establishes the similarity $s(q, d_j)$ or closeness of query q and document d_j 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 is a purely information retrieval approach to measuring similarity. It might be based on factor 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.

S4.3.1 Information Retrieval and keyword stuffing: Use of link information. One can fool such a statistical (i.e. 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 that relied on Information Retrieval Approaches in calculating $s(q, d_j)$. 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.

S4.3.2 Ranking based "evaluation" of a query using link information. A rank $r(d_j)$ for document d_j or $r(q, d_j)$ can be computed that is more reliable than $s(q, d_j)$.

S4.3.3 Link farming to manipulate link values. 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).

(c) Copyright A. Gerbessiotis. All rights reserved.

S4.4 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.)

(c) Copyright A. Gerbessiotis. All rights reserved.

Estimating the Corpus size (number of documents) 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.

(c) Copyright A. Gerbessiotis. All rights reserved.

Search Engine Volume, Market Share, and query length

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

	SEARCHES PER DAY (in millions)			Share of		Query Length
	02/2003	02/2009		the Market ***		Share ****
Engine	Searches/day*	Searches/day**			%	
Google	112	320	Google	Google	81%	1 word : 20%
AOL Search	93	3	AOL	AOL	1%	2 words: 23%
Yahoo!	42	74	Yahoo!	Yahoo!	9%	3 words: 22%
MSN Search	32	28	MSN/Live	Bing	5%	4 words: 15%
Ask Jeeves	14	10	Ask.com	Other	4%	5 words: 9%
InfoSpace	7	-	-			6 words: 4%
						7 words: 3%
TOTALS	319	430				8 or > : 4%

* 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/
*** Source (June 2009) :
http://marketshare.hitslink.com/search-engine-market-share.aspx?qprid=4
**** Source (July 2009) :
http://www.hitwise.com/press-center/hitwiseHS2004/google-searches-apr-09.php

Figure 7: Search Engines Statistics: 2003 vs 2009; Worldwide Market Share; Query length

IP addresses and domain names

The most difficult task of a search engine relates to locating a web-site. A web-crawler that fetches pages faces a number of daunting tasks: enumerating all possible locations in which information (in the form of a web-page) appears.

- There are 4 billion IPv4 addresses; In 2011, 1 out of 7 were registered, close to 600 million. Yet there were approximately 183,000,000 domain name registrations According to Figure 8, one expects only 60% of the domain name registrations to have useful indexable information. Fewer than half of them would correspond to a web-server.
- Domain name registrations are symbolic names that a user of a web-page remembers. There a Many to One mapping of domains to IP addresses or One to Many utilized for load balancing purposes. For example at some time 74.125.67.100, 74.125.127.100, 74.125.45.100 were IP addresses resolving `google.com`.

```
         Domain Name  Registrations       Out of a sample of 92,000,000 tested domain names
2003       32,000,000                            89% resolve to a web-site of which
2004       64,500,000                               24% to one-page web-sites
2009      183,000,000                               64% to multipl-page web-sites
2013      250,000,000-300,000,000                   12% to no web-site.
                                          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]
```

Figure 8: Web-Host Count Estimate

```
GoogleIndexed Volume (unique pages)
2000   1.0bn
2001   1.3bn
2002   2.0bn
2003   3.0bn
2004   3.3bn
2005   8.0bn  ~ 20% of crawled pages!
2006  25.0bn
 ...
2016  25.0bn  ~2.5% of crawled pages!
```

Figure 9: Search Engine Volume: Indexed vs Crawled 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 10: 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 11: Search Engine Statistics: Google

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.*