# A Survey On Web Information Retrieval Technologies

Lan Huang

Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
lanhuang@cs.sunysb.edu

## Abstract

Retrieving information from the Web is becoming a common practice for internet users. However, the size and heterogeneity of the Web challenge the effectiveness of classical information retrieval techniques. In this paper, we report the latest technology in search engines and hierarchical directories. We discuss the algorithmic and implementation issues of these software tools. The focus of this report is to survey the Web information retrieval research from a system researcher's point of view. We also propose new mechanisms to enable online update to large data corpus stored as inverted index lists. The proposed mechanism can save the storage cost and provide near real-time update to users.

## 1 Introduction

Web information retrieval is highly popular and at the same time presents a technical challenge due to the heterogeneity and size of the Web, which is over 350 million pages as of July, 1998. Many software tools are available for web information retrieval, such as search engines (Google, AltaVista et al.), hierarchical directories (Yahoo!), many other software agents and collaborative filtering systems. In this report, we explore the engineering details and algorithmic issues behind Web information retrieval systems.

First, we compare Web information retrieval and classical information retrieval and show where the challenges are. In the third section, we review representative search engines and their architectural features. After describing the general architecture of a search engine, we present the engineering issues in building a robust search engine and the existing algorithms developed to provide high-quality and high-relevance results. In this section, we also describe our Codir system which is designed to solve the online update problem. In the fourth section, we discuss the algorithms, architecture and performance of an automatic classification system (TAPER) for the Web built by IBM Santa Teresa Research Lab as well as another type of classification system (OpenGrid/ODP). In the fifth section, the problem of web statistics collection is discussed together with interesting results from an analysis of AltaVista query log.

# 2 Web Information Retrieval

With the fast growth of the Internet, more and more information is available on the Web and as a result, Web information retrieval has become a fact of life for most Internet users. However, compared with classic information retrieval, web information retrieval systems are faced totally different data sets. The uniqueness of Web information retrieval is listed as the following:

- **Bulk** The bulk size of the Internet is 350 million documents as measured on July, 1998, which is growing at the speed of 20M per month.
- **Dynamic Internet** The Internet is changing everyday while most classic IR systems are designed for mostly static text databases.
- **Heterogeneity** The Internet contains a wide variety of document types: pictures, audio files, text and scripts etc.
- **Variety of Languages** The types of languages used in the Internet is more than 100.
- **Duplication** Copying is another important characteristic of the Web, as claimed that nearly 30% of the Web pages are duplicates.
- **High Linkage** Each document averagely has more than 8 links to other pages.
- **Ill-formed queries** Web information retrieval systems are required to service short and not particularly well represented queries from the Internet users.
- **Wide Variance in Users** Each web user varies widely in their needs, expectations and knowledge.
- **Specific Behavior** It is estimated that nearly 85% users only look at the first screen of the returned results from search engines. 78% users never modify their very first query.

So the big challenge to the Web information retrieval is to meet the users information needs given the heterogeneity of the Web and the ill-formed queries.

# 3 General-purpose Search Engines

Most widely-used general-purpose search engines are Northern Light, AltaVista, Google, Infoseek(GoTo.com). This section, we first describe a performance overview of the existing search engines based on a survey from [24]. Next, details for search engines are presented along two dimensions: engineering issues for building a robust search engine, and algorithmic issues for providing a high-quality IR service.

## 3.1 The Goal

In classic information retrieval, the performance of an IR system is evaluated along three lines: recall, precision and precision at top 10 result pages. Precision means the percentage of pages retrieved that are relevant. Recall means the percentage of relevant pages that are returned. In Web IR, the quality of pages varies widely and thus just being relevant is not enough. The goal is to return both high-relevance and high-quality (in other word, valuable) pages.

## 3.2 Current Status of Search Engines

[24] did a survey on performance evaluation for various existing search engines based on librarians' personal experience. Table 1 shows a comparison among five most popular search engines. Google incorporates an innovative ranking algorithm in result page ranking. It provides relatively more relevant, high-quality result pages than others because of this ranking mechanism. AltaVista has the largest data collection among all the existing search engines. Northern Light is better serving queries on academia and business topics. Infoseek distinguishes itself with a *searching within results* feature, which is very powerful and as good as an excellent ranking mechanism. FastSearch has the second largest data collection as well as the fastest search engine.

## 3.3 Architecture of A Search Engine

In this section, we look at the architecture of a search engine and its general data structures.

### 3.3.1 Architecture

A search engine contains three components: an indexer, a crawler, a query server. The crawler collects pages from the Web. The indexer processes the retrieved documents and represents them in an efficient search data structure. The query server accepts the query from the user and returns the result pages by consulting with the search data structures. Figure 1 shows the system architecture of Google [7, 16] .

Most of Google is implemented in C or C++ for efficiency and can run on either Solaris or Linux.

In Google, the web crawling (downloading of web pages) is done by several distributed crawlers. There is a URLserver that sends lists of URLs to be fetched to the crawlers. The web pages that are fetched are then sent to the storeserver. The storeserver then compresses and stores the web pages into a repository. Every web page has an associated ID number called a docID which is assigned whenever a new URL is parsed out of a web page. The indexing function is performed by the indexer and the sorter. The indexer performs a number of functions. It reads the repository, uncompresses the documents, and parses them. Each document is converted into a set of word occurrences called hits. The hits record the word, its position in document, an approximation of font size and capitalization. The indexer distributes these hits into a set of "barrels", creating a partially sorted forward index. The indexer performs another important function. It parses out all the links in every web page and stores important information about them in an anchors file. This file contains enough information to determine where each link points from and to, and the text of the link.

The URLresolver reads the anchors file and converts relative URLs into absolute URLs and in turn into docIDs. It puts the anchor text into the forward index, associated with the docID that the anchor points to. It also generates a database of links which are pairs of docIDs. The links database is used to compute PageRanks for all the documents.

The sorter takes the barrels, which are sorted by docID and resorts them by wordID to generate the inverted index. This is done in place so that little temporary space is needed for this operation. The sorter also produces a list of wordIDs and offsets into the inverted index. A program called DumpLexicon takes this list together with the lexicon produced by the indexer and generates a new lexicon to be used by the searcher. The searcher is run by a web server and uses the lexicon built by DumpLexicon together with the inverted index and the PageRanks to answer queries.

| Software | Google | Alta Vista | Northern Light | InfoSeek | FastSearch |
|---|---|---|---|---|---|
| Size, Type. Size varies frequently and widely. | 90-100 million. General Web database. Use the Advanced Search with Boolean operators and results ranking options. | Possibly 275 million General Web database. Use the Advanced Search with Boolean operators and results ranking options. | Possibly 250 million. Integrates huge Web database and full text of recent articles from 5,400+ journals, and business/news sources available for $1 to $4. *Industry Search* allows searching within industry-based subject categories. *Publication Search* allows keyword searching within specific journal titles or keywords in journal titles. *Power Search* combines all the features above, plus date limiting and more. | About 75-90 million. *Search within results* features precise and easy to use. Excellent content makes and consolidation of pages from the same site make up for Infoseek's smaller size. | Possibly 250 million General Web database. Second Largest and the fastest search engine to date. Excellent ranking. Few options for refining search results, however. |
| Phrase Searching | Yes | Yes | Yes | Yes | Yes |
| Boolean Logic | AND always assumed (all keywords appear in all results). No OR (submit query twice). | AND (default), OR, AND NOT, NEAR (within 10 words). | AND, OR, NOT with nesting using ( ). Available from any search box. | No | No |
| Results Ranking | Based on page popularity measured in links to it from other pages. All terms (except stop words without preceding +) appear in all documents. Matching and ranking based on *cached* version of pages that may not be the most recent version. | By terms you specify in smaller SORT box on Advanced screen. *Sort* box searches on body of documents retrieved with Boolean logic in larger *Boolean expression* box on Advanced screen. Very powerful! | Automatic Fuzzy AND. Weights results by search term sequence, left to right. Can request results sort by date in Power Search. Fuzzy AND means: In ranking of results, documents with any terms (Boolean OR) are retrieved, but those with all terms (Boolean AND) are ranked first. | Automatic Fuzzy AND. Can sort results by date at top of results. | Automatic Fuzzy AND. |
| Multi-Language | No. | Yes, extensive list includes major non-Romanized Asian language | Yes, as a folder category. In Power Search, limit by major Romanized languages. | No. | No. |
| Sub-Searching | No. Revise or add terms. Can exclude (-) to search to refine. | Not explicitly, but Advanced Search offers (see box above) a sub-search. | No. Use *Folders* to select aspects of results | Yes. In results, specify *Search within these results*. Very powerful! | No. Revise or add terms or exclude (-) to search to refine. |

Table 1: Most Popular General-Purpose Search Engines(November, 1999)

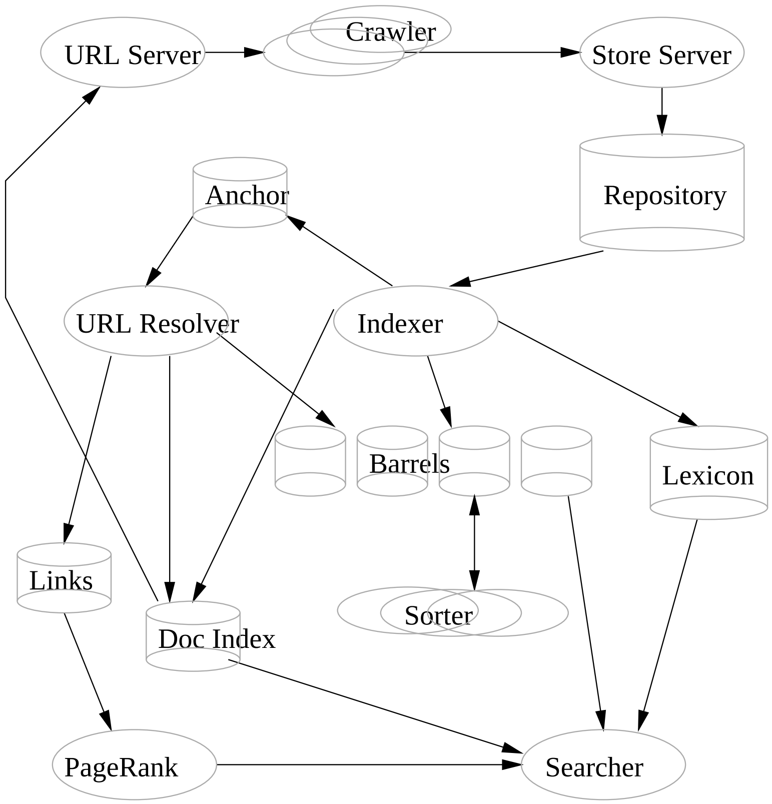


Figure 1: High Level Google Architecture

### 3.3.2 Data structure

1. Repository: The repository contains the full HTML texts of every web page. Each page is compressed using zlib (see RFC1950). In the repository, the documents are stored one after the other and are prefixed by docID, length, and URL. The repository requires no other data structures to be used in order to access it.

2. Document Index: The information stored in each entry includes the current document status, a pointer into the repository, a document checksum, and various statistics. If the document has been crawled, it also contains a pointer into a variable width file called docinfo which contains its URL and title. Otherwise the pointer points into the URLlist which contains just the URL. Additionally, there is a file which is used to convert URLs into docIDs. It is a list of URL checksums with their corresponding docIDs and is sorted by checksum. In order to find the docID of a particular URL, the URL's checksum is computed and a binary search is performed on the checksums file to find its docID. URLs may be converted into docIDs in batch by doing a merge with this file. This is the technique the URLresolver uses to turn URLs into docIDs.

3. Lexion: In the current implementation Google can keep the lexicon in memory on a machine with 256 MB of main memory. The current lexicon contains 14 million words.

4. Hit Lists: A hit list corresponds to a list of occurrences of a particular word in a particular document including position, font, and capitalization information. Google's writers considered several alternatives for encoding position, font, and capitalization – simple encoding (a triple of integers), a compact encoding (a hand optimized allocation of bits), and Huffman coding. In the end they chose a hand optimized compact encoding since it required far less space than the simple encoding and far less bit manipulation than Huffman coding. The details of the hit list is shown in Figure 2.

   Google's compact encoding uses two bytes for every hit. There are two types of hits: fancy hits and plain hits. Fancy hits include hits occurring in a URL, title, anchor text, or meta tag. Plain hits include everything else. A plain hit consists of a capitalization bit, font size, and 12 bits of word position in a document (all positions higher than 4095 are labeled 4096). Font size is represented relative to the rest of the document using three bits (only 7 values are actually used because 111 is the flag that signals a fancy hit). A fancy hit consists of a capitalization bit, the font size set to 7 to indicate it is a fancy hit, 4 bits to encode the type of fancy hit, and 8 bits of position. For anchor hits, the 8 bits of position are split into 4 bits for position in anchor and 4 bits for a hash of the docID the anchor occurs in. This gives some limited phrase searching as long as there are not that many anchors for a particular word. They expect to update the way that anchor hits are stored to allow for greater resolution in the position and docID hash fields. Google uses font size relative to the rest of the document because when searching, you do not want to rank otherwise identical documents differently just because one of the documents is in a larger font.

5. Forward Index: The forward index is actually already partially sorted. It is stored in a number of barrels (we used 64). Each barrel holds a range of wordID's. If a document contains words that fall into a particular barrel, the docID is recorded into the barrel, followed by a list of wordID's with hit lists which correspond to those words. This scheme requires slightly more

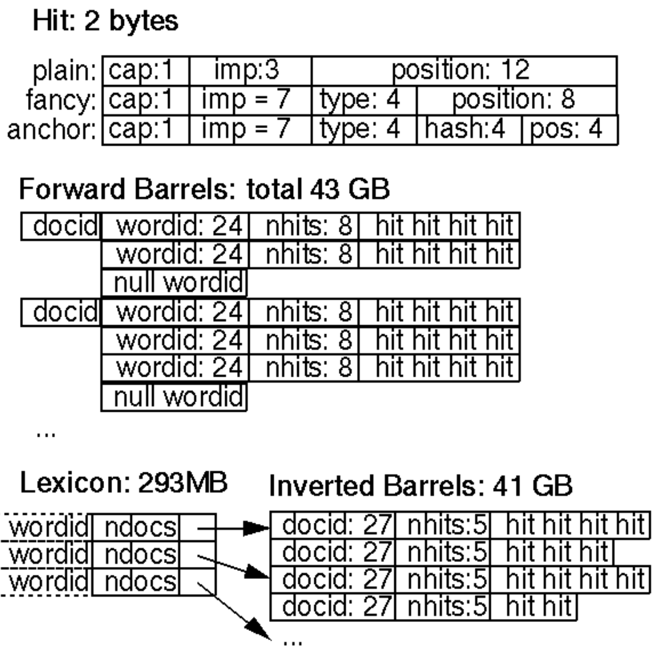

Hit: 2 bytes
plain: cap:1 imp:3 position: 12
fancy: cap:1 imp = 7 type: 4 position: 8
anchor: cap:1 imp = 7 type: 4 hash:4 pos: 4
Forward Barrels: total 43 GB
docid wordid: 24 nhits: 8 hit hit hit hit
wordid: 24 nhits: 8 hit hit hit hit
null wordid
docid wordid: 24 nhits: 8 hit hit hit hit
wordid: 24 nhits: 8 hit hit hit hit
wordid: 24 nhits: 8 hit hit hit hit
null wordid
...
Lexicon: 293MB
Inverted Barrels: 41 GB
wordid ndocs
wordid ndocs
wordid ndocs
docid: 27 nhits:5 hit hit hit hit
docid: 27 nhits:5 hit hit hit
docid: 27 nhits:5 hit hit hit hit
docid: 27 nhits:5 hit hit
...

Figure 2: Forward and Reverse Indexes and Lexion

storage because of duplicated docIDs but the difference is very small for a reasonable number of buckets and saves considerable time and coding complexity in the final indexing phase done by the sorter. Furthermore, instead of storing actual wordID's, each wordID is stored as a relative difference from the minimum wordID that falls into the barrel the wordID is in. This way, just 24 bits are used for the wordID's in the unsorted barrels, leaving 8 bits for the hit list length.

6. Inverted Index: The inverted index consists of the same barrels as the forward index, except that they have been processed by the sorter. For every valid wordID, the lexicon contains a pointer into the barrel that wordID falls into. It points to a doclist of docID's together with their corresponding hit lists. This doclist represents all the occurrences of that word in all documents. An important issue is in what order the docID's should appear in the doclist. One simple solution is to store them sorted by docID. This allows for quick merging of different doclists for multiple word queries. Another option is to store them sorted by a ranking of the occurrence of the word in each document. This makes answering one word queries trivial and makes it likely that the answers to multiple word queries are near the start. However, merging is much more difficult. (Note that we are now dealing with millions of files) Also, this makes development much more difficult in that a change to the ranking function requires a rebuild of the index. Google chose a compromise between these options, keeping two sets of inverted barrels – one set for hit lists which include title or anchor hits and another set for all hit lists. Hit lists which include title or anchor hits are considered as of higher ranking. This way, we check the first set of barrels first and if there are not enough matches within those barrels we check the larger ones.

## 3.4 Engineering Issues

### 3.4.1 Crawling the Web

Systems which access large parts of the Internet need to be very robust and carefully tested. Because of the immense variation in web pages and servers, it is virtually impossible to test a crawler without running it on a large part of the Internet. Invariably, there are hundreds of obscure problems which may only occur on one page out of the whole web and cause the crawler to crash, or worse, cause unpredictable or incorrect behavior. Externally, the crawler must avoid overloading Web sites or network links as it goes about its business. Internally, the crawler must efficiently deal with huge volumes of data. It must decides: in what order to scan the URLs in the queue, in what frequency to revisit pages to keep it up to date. Rule of thumb is important pages first. The ranking algorithm will be discussed in Algorithmic Issues section.

In order to scale to hundreds of millions of web pages, Google has a fast distributed crawling system. A single URLserver serves lists of URLs to a number of crawlers . Both the URLserver and the crawlers are implemented in Python. Each crawler keeps roughly 300 connections open at once. This is necessary to retrieve web pages at a fast enough pace. At peak speeds, the system can crawl over 100 web pages per second using four crawlers. This amounts to roughly 600K per second of data. A major performance stress is DNS lookup. Each crawler maintains its own DNS cache so it does not need to do a DNS lookup before crawling each document. Each of the hundreds of connections can be in a number of different states: looking up DNS, connecting to host, sending request, and receiving response. These factors make the crawler a complex component of the system.

To deal with huge volumes of data, Google uses asynchronous I/O to manage events, and a number of queues to move page fetches from state to state.

Cho etc. [19] uses the following technique to spread the work load among all crawlers in their crawling system: First, their crawler splits all URLS which are going to be crawled into 500 queues based on a hash of their server name. This causes all URLs from a given server to go into the same queue. The crawlers then read one URL from each queue at a time, moving to a new queue for each URL. This makes sure a given server is hit only once for every 500 URLs that are crawled. Also, for servers that are slow at returning documents, only one connection is allowed from the crawler to a particular server at a time.

### 3.4.2 Caching Query Results

Caching documents to reduce access delay is extensively used on the web. Most web browsers cache documents in the client's main memory or in some local disk. Although this is widely used, it is the least effective, since it rarely results in large hit rates. To improve cache hit rates, caching proxies are used. Proxies employ large caches which they use to serve a stream of requests coming from a large number of users. Cache replacement policies for web proxy have been intensively researched recently [9].

[26] proposed a query result caching policy for search engine site query cache. Their experiments on traces from Excite [14] shows that there exists temporal locality (20%-30%) in the queries submitted. The two-stage LRU (LRU-2S) cache replacement policy is as follows:

1. The blocks in the cache are ordered in completely the same fashion as a cache using LRU replacement.

2. The cache is divided into two parts, a *primary* section and an *secondary* one.
3. On a miss, the last block of the secondary section is replaced, and the new block is put on the head of the secondary section.
4. On a hit, the block is moved to the head of the primary section - if the block was in the secondary section, then the last block of the primary becomes the first of the secondary section.

From the performance evaluation, [26] drew the following conclusions:

- Medium-sized caches (a few hundred Mbytes large) can easily exploit the locality found and serve a significant percentage of the submitted queries from the cache. The experiments suggest that medium sized-cache can result in hit rates ranging from 25% to 30% (for warm caches).
- Effective Cache Replacement Policies should take into account both recency and frequency of access in their replacement decisions. Experimental results suggest that LBR-2S always perform better than simple LRU which does not take frequency of access into account.
- The proposed LRU-2S replacement policy has performance comparable to (and sometimes better than) algorithms in [30] while maintaining a low execution cost.

Other similar replacement policy can be referred from [30, 18].

### 3.4.3 Incremental Updates to Inverted Index

The inverted index lists are built for *static* data collection instead of dynamic data collection which allows add/deletion/update. To the best of our knowledge, no such search engine exists which can reflect the latest update to search engine users in nearly real time. Most search engines update their database monthly by re-crawling the Web then rebuilding the whole inverted index lists. The inverted lists for a multi-gigabyte document collection will range in size from a few bytes to millions of bytes, and they are typically laid out contiguously in a flat inverted file with now gaps between the lists. Adding to inverted lists stored in such a fashion requires expensive relocation of growing lists and careful management of free-space in the inverted file. In this section, we first discuss previous efforts in information retrieval to speed up incremental update, and next we discuss our Codir system which supports online update.

[13] presented an information retrieval system – INQUERY, which supports incremental update using the Mnenme persistent object store to manage its inverted file index. Inverted lists are allocated in fixed-size objects with a finite range of sizes, limiting the number of relocations a growing list will experience. When an inverted list has exceeded the largest object size, additional large objects are allocated and chained together in a linked list. Experimental results show that this system is able to provide superior incremental indexing performance in terms of both time and space, with only a small impact on query processing. The distribution of inverted list sizes for the TIPSTER volume 1 document collection used in their performance evaluation fits Zipf [38] distribution well. Over 90% of the inverted lists are less than 1000 bytes(in their compressed form), and account for less than 10% of the total inverted file size. Furthermore, nearly half of all lists are less than 16 bytes. This means that many inverted lists will never grow after their initial creation. These lists should be allocated in a space efficient manner. Most of the inverted file size is accounted

for by a very small number of large inverted lists. These inverted lists will experience continuous, possibly vigorous growth. They must be allocated in a manner that allows efficient access.

The main allocation scheme in [13] is as following: Instead of allocating each inverted lists in a single object of the exact size, lists are allocated using a range of fixed size objects, where the sizes range from 16 to 8192 bytes by powers of 2. When a new list is created, an object of the smallest size large enough to contain the list is allocated. A list can then grow to fill the object. When it exceeds the object, a new object of the next larger object size is allocated, the content of the old object is copies to this object, and the old object is freed. When a list exceeds the largest object size, a linked list of 8192 byte objects is started.

Best performance is achieved when documents are added in the largest batches possible, both in terms of incremental indexing time and resultant query processing speed. The cost of adding inverted list is proportional to the size of the update when batching is used.

[1] proposed a new data structure to support incremental indexing, and presented a detailed simulation to support incremental indexing over a variety of disk allocation schemes. Their data structure manages small inverted lists in buckets(similar to the disk blocks in [21] ) and dynamically selects large inverted lists to be managed separately. The simulation results indicate that the best long list allocation scheme for update performance is to write the new portion of a long list in a new chunk at the end of the file. This is quite similar to the scheme used in [13].

Another scheme handles large lists distinctly from small list is proposed by Faloustsos and Jagadisk [15]. The general case is "HYBRID" scheme. The HYBRID scheme essentially chains together chunks of the inverted list and provides a number of parameters to control the size of the chunks and the length of the chains. At one extreme, limiting the length of a chain to one and allowing the chunks to grow results in contiguous inverted lists, where relocation of the inverted list into a larger chunk is required when the current chunk is filled. At the other extreme, fixed size chunks and unlimited chain lengths give a standard linked list.

Cutting and Pederson [11] investigated optimizations for dynamic update of inverted lists managed with a B-tree. For a speed optimization, they proposed accumulating postings in a main memory postings buffer. They also proposed storing the smallest inverted lists directly in the B-tree index.

To provide non-stopping search engine services, the above solutions have to keep a second copy of the inverted lists. The update operation is done on the second copy. In this way, all queries come during the updating period can be uninterupttedly serviced on the first copy of the inverted lists. When the update process is completed, the old copy is discarded and the second copy is now in use for servicing search queries. Thus changes are made visible to web users. The incremental index building is much faster than rebuilding the whole inverted lists, however, updates are still not seen by the users immediately. Our system Codir [10] is implemented in such a way that update and searching are serviced simultaneously on the same set of inverted lists. Thus changes are reflected to the users in an almost real-time fashion, and the extra storage cost is avoided.

The data structures are shown in Figure 3. The Codir system interface supports both query and update. Each is serviced by a thread/process. They share data through shared memory. Update operation can be deletion or insertion of a document. The dictionary component of the inverted index, which is a hash table, is always memory-resident. Each hash table entry points to a resolution list, which in turn points to a per-word list, which contains locations of occurrence of the associated word. Each per-word list entry corresponds to a document in the database, and contains a document ID and a block bitmap, which identifies the set of blocks within the document

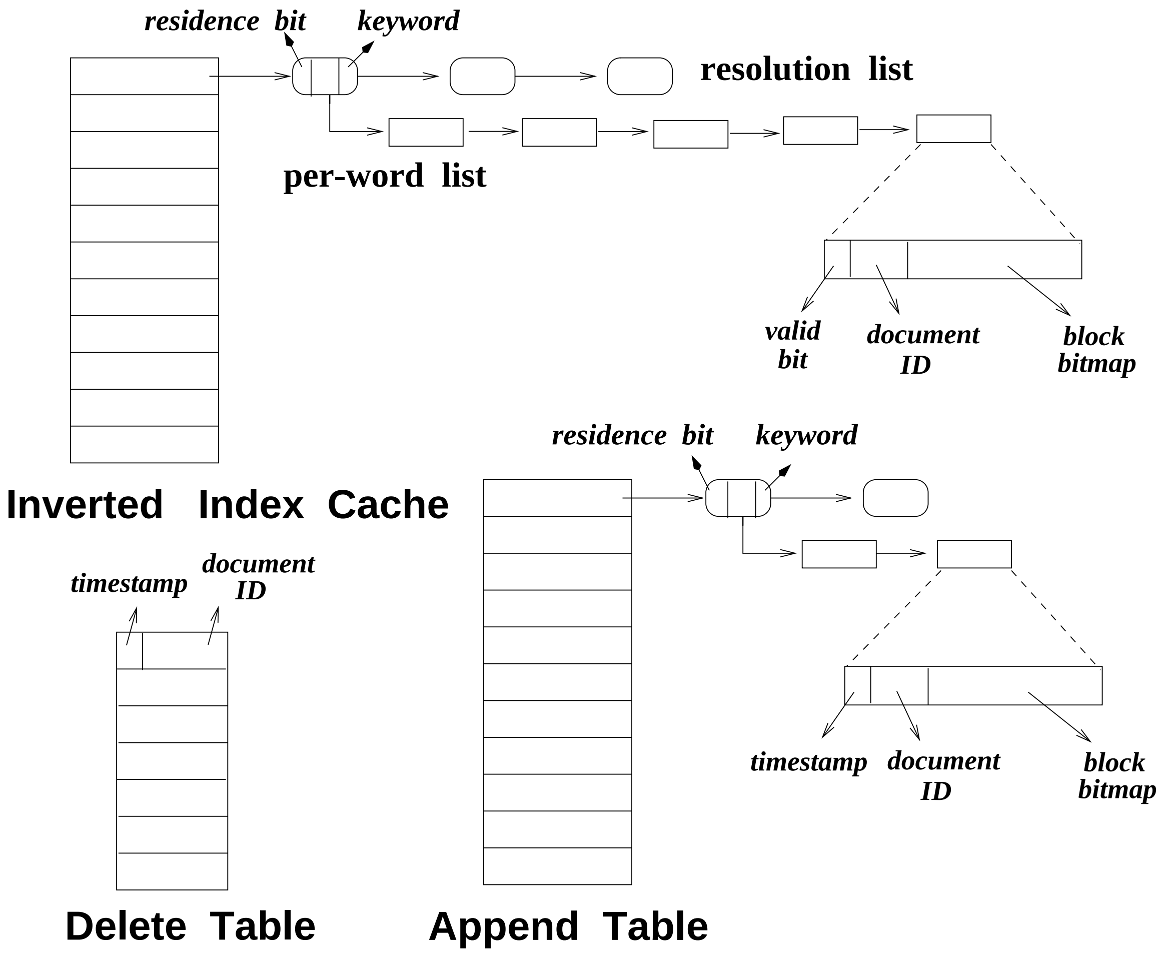


Figure 3: Data Structure Used in Codir

that contain the associated word. Each per-word list entry also includes a valid bit to indicate if the corresponding document has been deleted. At any point in time, only a subset of the inverted index is memory resident. The basic unit of inverted index caching is a per-word list, which could vary in length. The residence bit of those resolution list entries that point to memory-resident per-word lists is turned on. The Append Table has the same structure as the inverted index cache. The only difference is that the per-word list entries in the Append Table contain a timestamp field, rather than a valid bit, for concurrency control. Each Delete Table entry includes the ID of the deleted document, and a timestamp(CTS) for concurrency control.

- Upon a deletion request, the URL/filename's associated docID is inserted into Deletion Table along with the current system timestamp as its CTS.

- On an insertion request, the document is parsed and inverted list is built for this file into Append Table. Each posting also contains a CTS value for concurrency control used in query service. When the Append Table is too huge to be held in main memory, it will be integrated into the permanent inverted list copy on the disk.

- On a query request, the query engine will search the inverted list cache, if it is a cache miss, the corresponding inverted list will be brought into the cache from the disk. Then the query engine will combine this inverted list with the corresponding list in the Append Table. Before the result set is returned to the client, the Delete Table entries are scanned against the result list to mark those deleted docID. This scan and marking procedure is fast since all docID are sorted beforehand. Other ranking algorithms can be applied onto the result list. To maintain the semantic correctness of the result set, CTS(current timestamp) is used. The query engine decides the maximum common CTS among all those words in the query as the current working timestamp. The reason of choosing this CWTS (current working timestamp) is that updating thread may change some of the lists used by the current query. Also we implemented necessary locking mechanism to guarantee that the inverted list in the cache is not replaced out if some thread already is working on it.

- The Append Table and Delete Table are reflected into the permanent storage periodically.

We haven't got chance to test Codir system against a large Web data collection. Current experiments are against some plain text database with size about a few hundred MBytes. We believe that Codir idea can be scaled to larger data collection and improve the performance of existing search engine.

## 3.5 Algorithmic Issues

### 3.5.1 Ranking

1. PageRank Algorithm

   L. Page and S. Brin proposed PageRank algorithm in [23, 19, 7]. PageRank makes use of the citation(link) graph of the web. The PageRank algorithm is defined as follows:

   We assume page A has pages T1...Tn which point to it (i.e., are citations). The parameter $d$ is a damping factor which can be set between 0 and 1. Google usually sets $d$ to 0.85. Also $C(A)$ is defined as the number of links going out of page A. The PageRank of a page A is given as follows:

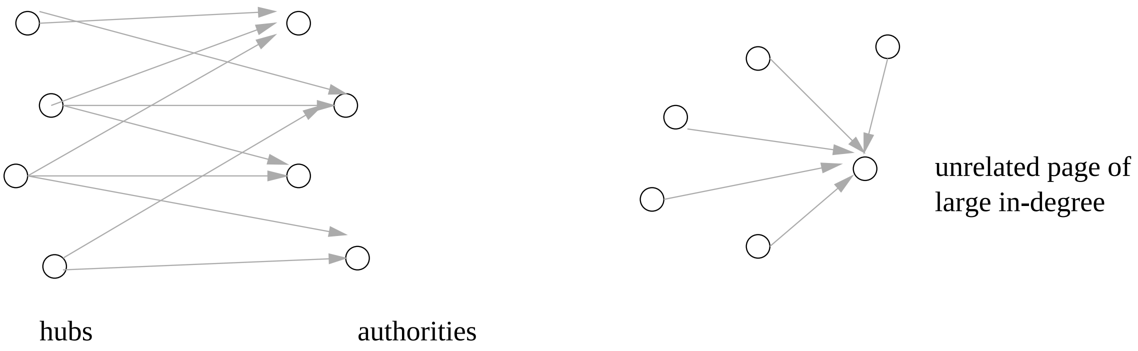


Figure 4: A densely linked set of hubs and authorities

$$PR(A) = (1 - d) + d(PR(T1)/C(T1) + ... + PR(Tn)/C(Tn))$$

Note that the PageRanks form a probability distribution over web pages, so the sum of all web pages' PageRanks will be one.

PageRank or PR(A) can be calculated using a simple iterative algorithm, and corresponds to the principal eigenvector of the normalized link matrix of the web. Also, a PageRank for 26 million web pages can be computed in a few hours on a medium size workstation.

PageRank can be thought of as a model of user behavior. Assume there is a "random surfer" who is given a web page at random and keeps clicking on links, never hitting "back" but eventually gets bored and starts on another random page. The probability that the random surfer visits a page is its PageRank. And, the d damping factor is the probability at each page the "random surfer" will get bored and request another random page. One important variation is to apply the damping factor $d$ only to a single page, or a group of pages. This allows for personalization and can make it nearly impossible to deliberately mislead the system in order to get a higher ranking.

Another intuitive justification is that a page can have a high PageRank if there are many pages pointing to it, or if there are some pages that point to it and have a high PageRank. Intuitively, pages that are well cited from many places around the web are worth looking at. Also, pages that have perhaps only one citation from something like the Yahoo! homepage are also generally worth looking at. If a page was not high quality, or was a broken link, it is quite likely that Yahoo's homepage would not link to it. PageRank handles both these cases and everything in between by recursively propagating weights through the link structure of the Web.

2. HITS Algorithm

   HITS is proposed by J. Kleinberger when he visited IBM Almaden Research Lab. It was implemented in the Clever search engine from IBM. Given a query, HITS [22] will find good sources of content (defined as authorities) and good sources of links (defined as hubs). Authorities have large in-degree. Hub pages are pages that "pull together" authorities on a given topic, and allow us to throw out unrelated pages of large in-degree(Those pages are simply universally popular like Yahoo!). Hubs and Authorities exhibit a *mutually reinforcing relationship*: A good hub is a page that points to many good authorities; a good authority is a page that is pointed to by many good hubs.

Before we go on describing the algorithm calculating the hubs and authorities, we have to decide the set of pages we will work on. Ideally we would like to focus on a collection $S_\sigma$ of pages with the following properties.

(i) $S_\sigma$ is relatively small.

(ii) $S_\sigma$ is rich in relevant pages.

(iii) $S_\sigma$ contains most (or many) of the significant authorities.

By keeping it small, one is able to afford the computational cost of applying non-trivial algorithms; by ensuring it is rich in relevant pages it is made easier to find good authorities, as these are likely to be heavily referenced within $S_\sigma$.

Kleinberger suggests the following solution to find such a collection of pages. For a parameter $t$ (typically set to about 200), HITS first collects the $t$ highest-ranked pages for the query $\sigma$ from a text-based search engine such as AltaVista [3] or Hotbot [17]. Those t pages are referred as *root set* $R_\sigma$. HITS then increases the number of strong authorities in our subgraph by expanding $R_\sigma$ along the links that enter and leave it. In concrete terms we define the following procedure.

```
Subgraph($\sigma, \varepsilon, t, d$)
    $\sigma$: a query string
    $\varepsilon$: a text-based search engine
    $t, d$: natural numbers
    Let $R_\sigma$ denote the top t results of $\varepsilon$ on $\sigma$
    Set $S_\sigma := R_\sigma$
    For each page $p \in R_\sigma$
        Let $\Gamma^+(p)$ denote the set of all pages p points to.
        Let $\Gamma^-(p)$ denote the set of all pages pointing to p.
        Add all pages in $\Gamma^+(p)$ to $S_\sigma$.
        if $|\Gamma^-(p)| \leq d$ then
          Add all pags in $|\Gamma^-(p)|$ to $S_\sigma$.
        Else
          Add an arbitary set of d pages from $|\Gamma^-(p)|$ to $S_\sigma$.
    End
    Return $S_\sigma$
```

A link is considered as *transverse* if it is between pages with different domain names, and *intrinsic* if it is between pages with the same domain name. All intrinsic links are deleted from the graph $G[S_\sigma]$, keeping only the edges corresponding to *transverse* links; this results in a graph $G_\sigma$. As follows, the algorithm to compute the hubs and authorities given the graph $G_\sigma$ is:

**An Iterative Algorithm.** HITS associates a non-negative *authority weight* $x^{<p>}$ and a non-negative *hub weight* $y^{<p>}$. The weights of each type are normalized so that their square sum to 1 (The following *Iterate* algorithm normalizes the result weight values of each interation using this criteria.):

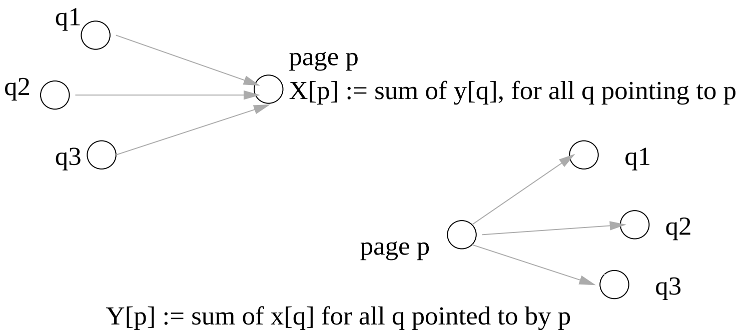

q1
q2
q3
page p
X[p] := sum of y[q], for all q pointing to p
q1
q2
q3
page p
Y[p] := sum of x[q] for all q pointed to by p

Figure 5: The basic operations

Numerically the mutually reinforncing relationship can be expressed as follows: if $p$ points to many pages with large x-values, then it should receive a large y-value; and if $p$ is pointed to by many pages with large y-values, then it should receive a large x-value. Given weights $x^{<p>}$ ,$y^{<p>}$ , the $\mathcal{I}$ operation updates the x-weights as follows.

$$x^{<p>} \longleftarrow \sum_{q:(q,p)\in E} y^{<q>}$$

The $\mathcal{O}$ operation updates the y-value as follows.

$$y^{<p>} \longleftarrow \sum_{q:(p,q)\in E} x^{<q>}$$

```
Iterate(G,k)
    G: a collection of n linked pages
    k: a natural number
    Let z denote the vector(1,1,1,...,1) ∈ $R^n$.
    Set $x_0$ := z.
    Set $y_0$ := z.
    For i = 1, 2, ... k
        Apply the $\mathcal{I}$ operation to ($x_{i-1}$,$y_{i-1}$), obtaining new x-weights $x_i'$.
        Apply the $\mathcal{O}$ operation to ($x_i$',$y_{i-1}$), obtaining new y-weights $y_i'$.
        Normalize $x_i'$, obtaining $x_i$.
        Normalize $y_i'$, obtaining $y_i$.
    End
    Return ($x_k$,$y_k$).
```

The top c authorities and top c hubs are those pages that have the highest authority and hub weights at the end of this algorithm.

However, Bharat and Henzinger [6] point out that HITS did not work well in all cases due to the following three reasons:

- *Mutually Reinforcing Relationships Between Hosts:* Sometimes a set of documents on one host point to a single document on a second host. This drives up the hub scores of the documents on the first host and the authority score of the document on the second host. The reverse case, where there is one document on a first host pointing to multiple documents on a second host, creates the same problem. Since our assumption is that the set of documents on each host was authored by a single author or organization, these situations give undue weight to the opinion of one "author".
- *Automatically Generated Links.* Web documents generated by tools (e.g. Web authoring tools, database conversion tools) often have links that were inserted by the tool.
- *Non-relevant Nodes:* [6] shows that the neighborhood graph often contains documents not relevant to the query topic. If these nodes are well connected, *topic drift* problem arises: the most-highly ranked authorities and hubs tend not to be about the orginal topic. For example, when running the algorithm on the query "mango fruit" the computation drifted to the general topic "fruit".

[6] presents two basic approaches to tackle topic drift. Assuming one can determine the relevance of a node to the query topic: (i) eliminating non-relevant nodes from the graph. (ii) regulating the influence of a node based on its relevance. If W[n] is the relevance weight of a node n and A[n] the authority score of the node we use $W[n] * A[n]$ instead of A[n] in computing the hub scores of nodes that point to it. Similarly we use $W[n] * H[n]$ instead of H[n] in computing the authority score of nodes it point to. To avoid the undue weight given to the opinion of a single person, fractional weights to edge is used in such cases: If there are k edges from documents on a first host to a single document on a second host we give each edge an *authority weight* of $1/k$. The weight is used when computing the authority score of the document on the second host. If there are l edges from a single document on a first host to a set of documents on a second host, we give each edge a hub weight of $1/l$. Additionally isolated nodes are discarded from the graph. Now A[n], H[n] are computed as following:

$$A[n] := \sum_{(n',n)\in N} H[n'] \times auth\_wt(n',n)$$

$$H[n] := \sum_{(n',n)\in N} A[n'] \times hub\_wt(n',n)$$

3. Others – Anchor Text, Headings etc.

The text of links is treated in a special way in Google. Most search engines associate the text of a link with the page that the link is on. In addition, Google associates it with the page the link points to. This has several advantages. First, anchors often provide more accurate descriptions of web pages than the pages themselves. Second, anchor texts may exist for documents which cannot be indexed by a text-based search engine, such as images, programs, and databases. This makes it possible to return web pages which have not actually been crawled. Note that pages that have not been crawled can cause problems, since they are never checked for validity before being returned to the user. In this case, the search engine can even return a page that never actually existed, but had hyperlinks pointing to it. However, it is possible to sort the results [7], hoping that this type of pages always ranked as of low value, so that this particular problem rarely happens.

This idea of propagating anchor text to the page it refers to was implemented in the World Wide Web Worm [27] especially because it helps search non-text information, and expands the search coverage with fewer downloaded documents. Anchor propagation can help provide better quality results. Using anchor text efficiently is technically difficult because of the large amounts of data which must be processed according to [7]. In current Google's crawl of 24 million pages, it indexed over 259 million anchors.

[28] assigns different weight to headings as well as anchor text. They group HTML tags into six classes: Plain Text, Title, H1-H2, H3-H6, Strong and Anchor. Strong class includes those tags such as STRONG, B, EM, I, U, DL, OL, UL. Their conclusion is that anchor texts and STRONG class should carry more weight.

### 3.5.2 Duplicate Elimination

Approximately 30% of the pages on the Web are (near) duplicates. To automatically identify mirrored collections on the web can improve the performance of crawling, ranking, archiving and caching. For example Google search engine presents the results to user after filtering out the duplicates. Also avoiding crawling those duplicate pages can save storage space of tens of Giga size.

However there are important challenges in (1) defining the notion of a replicated collection precisely, (2) developing efficient algorithms that can identify such collections, and (3) effectively exploiting this knowledge of replication. One of the major difficulties in detecting replicated collections is that many replicas may not be strictly identical to each other. The reasons include:

1. Update Frequency: The primary copy of a collection may be updated constantly while mirror copies are updated not so frequently.

2. Mirror Partial Coverage: In some cases, only a subset of a collection may be mirrored.

3. Different Formats: The document in a collection may not themselves appear as exact replicas in another collection. For example, in some cases, it appears as Postscript file, in some other collection, it may be Adobe PDF or Microsoft Word format.

4. Partial Crawls: In some cases, the crawler does not have the complete set of data. For example, the crawler may only fetch part of the directory tree for an online book manuscript with many chapters.

One can determine the similarity of two pages in a variety of ways. One can use the information retrieval notion of textual similarity [35]. One could also use data mining techniques to cluster pages into groups that share meaningful terms [29], and then define pairs of pages within a cluster to be similar. A third option is to compute textual overlap by counting the number of common *chunks* of text(e.g., sentences or paragraphs) that pages share [2, 36, 37, 5]. This option is used in [20]. In the experiments, each page is first converted from its native format (e.g., HTML, PostScript) into simple textual information using standard converters. The resulting page is chunked into smaller textual units (e.g. lines or sentences). Each textual chunk is hashed down to a 32-bit fingerprint. If two pages share more than some threshold T of chunks with identical fingerprints, then the pages are said to be similar.

The formal definition for similar collection follows: Equisized collections $C_1$ and $C_2$ are similar (i.e, $C_1 \cong C_2$) if there is a one-to-one mapping M (and vice-versa) that maps all $C_1$ pages to all $C_2$ pages such that:

- Similar pages: Each page $p \in C_1$ has a matching page M(p) $\in C_2$, such that $p \approx M(p)$.
- Similar links: For each page p in $C_1$, let $P_1(p)$ be the set of pages in $C_1$ that have a link to page p. Similarly define $P_2(M(p))$ for pages in $C_2$. Then we have pages $p_1 \in P_1(p)$ and $p_2 \in P_2(M(p))$ such that $p_1 \approx p_2$(unless both $P_1(p)$ and $P_2(M(p))$ are empty). That is, each of the two corresponding pages should have at least one parent (in their corresponding collections) that are also similar pages.

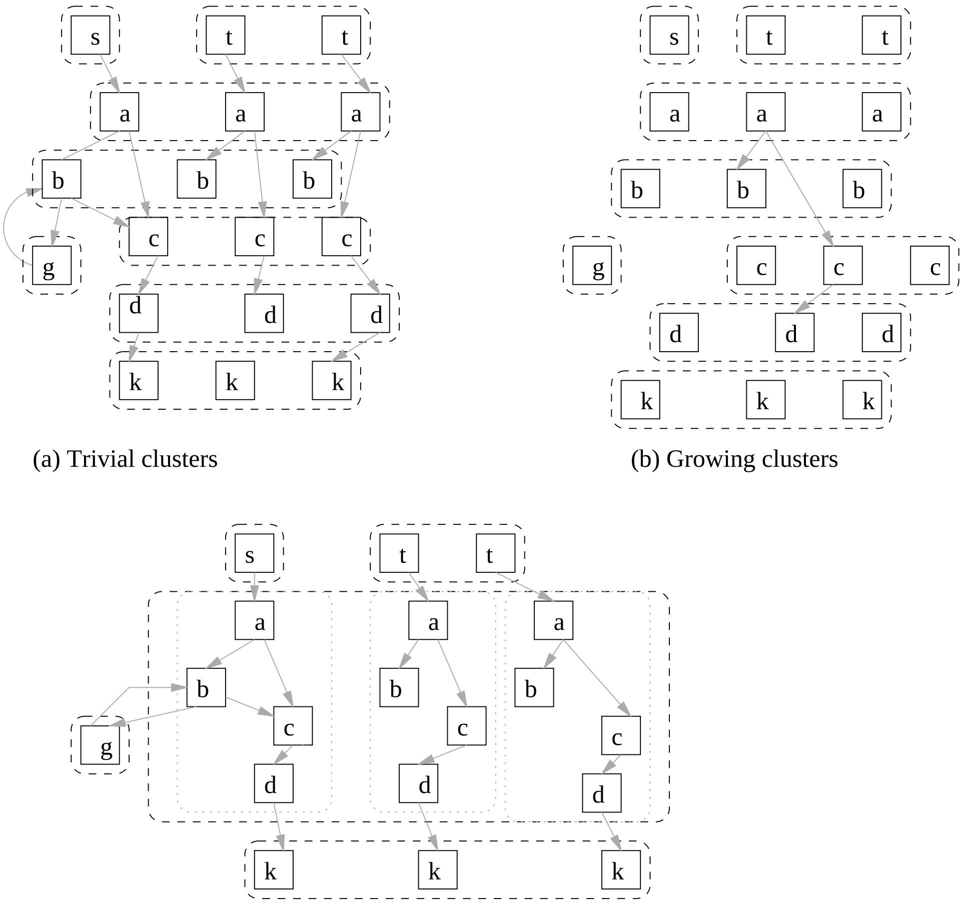
Before we describe the Growth Strategy which can identify similar clusters within a given web graph, we need to clarify the following definitions:

- **Cluster** We define a set of equi-sized collections to be a cluster. The number of collections in the cluster is the *cluster cardinality*. If $s$ is the collection size of each collection in the cluster, the cluster size is $s$ as well.
- **Identical Cluster** Cluster $R = C_1, C_2, ..., C_n$ is an *identical cluster* if for $\forall i, j, C_i \equiv C_j$. That is, all its collections are identical.
- **Similar Cluster** A cluster $R = C_1, C_2, ..., C_n$ is similar if all of its collections are pairwise similar, i.e., if for $\forall i, j, C_i \cong C_j$.

The goal is to identify similar clusters within a given web graph. Examples show that different types of similar clusters can be found in a web graph, and that there are complex interactions between the number of clusters found, their cardinalities, and the sizes of their collections. Following is *Growth Stategy* algorithm which *grows* clusters from smaller sized ones. Figure 6 shows an example of growing clusters. The algorithm first computes *trivial clusters* on the given graph. These clusters have collections of size one and are found by grouping similar pages. For example, the two pages labeled $t$ are determined to be similar, and form one of the trivial clusters. (This cluster has two collections, each collection with a single page.) Figure 6(a) shows the trivial clusters found in this example web graph. Next, the algorithm merges trivial clusters that can lead to similar clusters with larger collections. Figure 6(b) detail the process. The outcome is shown in Figure 6(c), where trivial clusters a, b, c and d have been merged into a larger-size cluster. (Clusters are shown with dashed lines, collections with dotted lines.) The remaining trivial clusters are not merged. For example, the $t$ cluster is not merged with the larger cluster because it would lead the collections to be of different sizes and hence not similar by the above definition.

We now discuss how to merge clusters. Consider two trivial similar clusters $R_i = p_1, p_2, ..., p_n$ and $R_j = q_1, q_2, ..., q_m$, where $p_k$ and $q_k$ ($1 \leq k \leq n$) are pages in $R_i$ and $R_j$. [20] define:

1. $s_{i,j}$ to be the number of pages in $R_i$ with links to at least one page in $R_j$,
2. $d_{i,j}$ to be the number of pages in $R_j$ with links from at least one page in $R_j$,
3. $|R_i|$ to be the number of pages in $R_i$, and
4. $|R_j|$ to be the number of pages in $R_j$.



(a) Trivial clusters

(b) Growing clusters

(c) Maximal cardinality clusters

Figure 6: Growing similar clusters

Two trivial clusters $R_i, R_j$ can be joined if they satisfy the *merge condition* $|R_i| = s_{i,j} = d_{i,j} = |R_j|$.

The Growth Strategy to find similar clusters:

1. Find all trivial clusters.
2. Form a trivial cluster graph (TCG) G=(V,E), where each vertex $v_i$ in V corresponds to one of the trivial clusters, and we have an edge from $v_i$ to $v_j$ if $v_i, v_j$ satisfy the merge condition.
3. Merge each connected component of the TCG into a similar cluster. The merge process for a connected component can start any node from which all the others in the component can be reached.

Now duplicate filtering is practiced in most search engines.

## 4 Hierarchical Directories and Automatic Categorization

Hierarchical directories are the ontologies of the Web. They are portals to the Web for Internet users, i.e. good starting points for a user to surf the Web. Also they provide a more focused

| name | Librarians' Index | Infomine | Britannica Web's Best | Yahoo! | Galaxy |
|---|---|---|---|---|---|
| Size,type | About 5,000. Compiled by public librarians in information supply business. Highest quality sites only. Great annotations. | About 16,000. Compiled by academic librarians. | About 150,000. Hand-picked, annotated, and ranked by Britannica editors. | About 1 million. Scarce descriptions and annotations. Biggest and most famous directory around. Many sub-Yahoo's by region, country, topic. | About 300,000. Generally good annotations. |
| Phrase searching | No. | Yes. Use ” ” | Yes. More than word searched as phrase. | Yes. Use ” ” | No. |
| Boolean logic | AND implied between words. Also accepts OR and NOT | AND implied between words. Also accepts OR. | Accepts AND, OR, NOT | No. | OR implied between words. Also accepts AND, OR, NOT |
| Sub-Searching | No. | No. | In results, specify SORT by subject in result. | Yes. In results, select search within category or all of Yahoo. | No. |

Table 2: Most Popular Directories(Nov,1999)

way to document searching. Instead of searching the entire Web, query can be restricted to a particular relevant category. This way the quality and relevance of the result set can be improved. In this section, we first list the most popular existing hierarchical directories. Then we address the issue of categorizing the Web from two directions: first we describe the research efforts devoted to automatic categorization, and then we describe an interesting proposal for Web categorization.

## 4.1 Current Status of Hierarchical Directories

Yahoo! is a good example of Web directories. In the sea of information provided by the Web, Web portals provide Internet users a good starting point. Table 2 [24] lists the most popular hierarchical directories existing on the Web today.

Another widely used open directory source is the Open Directory Project [31]. This is an open source and maintained by volunteers. Each volunteer can maintain a sub-category he/she is familiar with. The open directory has been used in AltaVista, Netscape, HotBot, Lycos etc. However it is still a manual directory and can hardly scale with the fast growth of the Web.

## 4.2 Automatic Categorization I – Taper

To keep up with the fast growth of the Web, IBM Santa Teresa Research Lab has been researching automatic Web categorization. Taper [34, 32] is the system they built for this purpose. Taper represents: A taxonomy-and-path-enhanced-retrieval system.

### 4.2.1 Capabilities and features

Most queries posted to search engines are very short. Such queries routinely suffer from the *abundance problem:* there are many aspects to, and even different interpretations of the keywords typed. In a database that associated not only keywords but also topics with documents, the query could elicit not a list of documents, but a list of topic paths. This will solve abundance problem to a large extent.

Taper group claims that the common notion of a document abstract or signature as a function of the document alone is of limited utility. In the case of a taxonomy, they argue that a useful signature is a function of both the document and the topic path(context). These feature words are called context-sensitive features.

### 4.2.2 Models and Algorithms

**Statistics collection** The goal of this module is to collect term statistics from a document and dispenses with it as fast as possible. A term is a 32-bit ID, which could represent a word, a phrase, words from a linked document, etc.

**Feature selection** After statistics collection, next step is feature selection. Fix an internal node $c_0$ and its children $c, c_1, c_2$, etc. in a taxonomy tree, and consider the classifier at $c_0$, whose goal is to route documents into those subtrees that best match a test document. Some terms are better than others for discriminating one document from another because they occur more frequently in a few classes compared to the others. These terms are good discriminators; the others are noise and should be discarded before the classifier forms document models for the classes. Note that these good discriminators can be diverse for different internal nodes. Finding the best discriminators depends on the statistical model one assumes the data is drawn from. Many proposed models for text, related to Poisson and multinomial processes, exist in the literature. For most such models, finding exactly the optimal subset of terms out of a lexicon of 50,000-100,000 terms appears impractical. In TAPER, rather than search for arbitrary subsets, it first orders the terms by decreasing ability to separate the classes; one notion of such ability that derived from Pattern Recognition literature [12] and found effective is the following score, mathematical explanations for this formula are omitted here:

$$score(t) = \frac{\sum_{c_1,c_2}(\mu(c_1,t) - \mu(c_2,t))^2}{\sum_c \frac{1}{|c|}\sum_{d \in c}(f(t,d,c) - \mu(c,t))^2}, (1)$$

where c, $c_1, c_2$ are children of internal node $c_0$, f(t,d,c) is the number of times term t occurs in document d in the training set of class c, with document length normalized to 1, and $\mu(c,t) = \frac{1}{|c|}\sum_{d \in c} f(t,d,c)$. TAPER computes the score of all terms, order the terms by decreasing score, and pick a set F terms which are of top above scores that gives the best classification accuracy over a (random) validation set (Figure 7).

**Evaluation** After feature selection, a classifier is associated with each $c_0$. Suppose $c_0$ has children $c_1, ...c_l$, given a class model, the classifier at $c_0$ estimates model parameters for each child. When a new document is input, the classifier evaluates, using the class models and Bayes's law, the posteriori probability of the document being generated from each child $c \in \{c_1, ..., c_l\}$ in turn, given it was generated from $c_0$. TAPER uses very simple document models for large scale computational efficiency. In the *Bernoulli* model, every class c has an associated coin with as many sides as there are terms in the lexicon. Each face of the coin corresponds to some term t and has some success

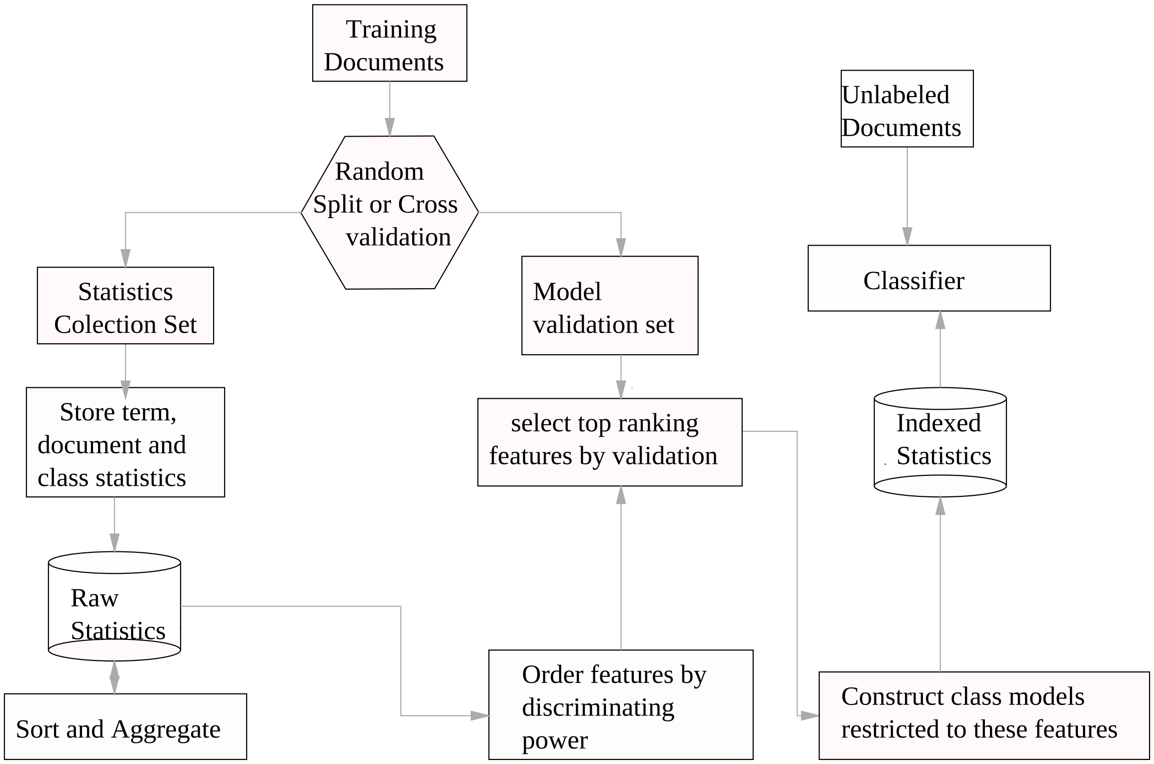

Training Documents
Random Split or Cross validation
Statistics Colection Set
Model validation set
Unlabeled Documents
Classifier
Store term, document and class statistics
select top ranking features by validation
Indexed Statistics
Raw Statistics
Sort and Aggregate
Order features by discriminating power
Construct class models restricted to these features

Figure 7: A sketch of the TAPER hierarchical feature selection and classification engine

probability $\theta(c,t)$, estimated using f(t,d,c) and additional statistics collected during the document scanning step. A document in the class is generated by repeatedly flipping the coin and writing down the term corresponding to the face that turns up. Assuming this model, we have:

$$Pr[d \in c|c_0, F] = \frac{\pi(c)\prod_{t\in d\cap F}\theta(c,t)^n(d,t)}{\sum_{c'}\pi(c')\prod_{t\in d\cap F}\theta(c',t)^n(d,t)}$$

where $\theta(c,t)$ is the probability that "face" t turns up when "coin" c is tossed. $\pi(c)$ is the prior probability of class c, typically, the fraction of documents in the training or testing set that from class c. n(d,t) is the number of times term t occurred in document d. The details of the estimation of $\theta$ have been described in [33]. An alternative binary model truncate n(d,t), the number of times term t occurs in document d, to a {0, 1} value. Classification over the entire taxonomy can then be posed as a shortest path problem on the taxonomy.

A sketch of the TAPER hierarchical feature selection and classification engine is shown in Figure 7.

### 4.2.3 Enhanced Categorization Using Hyperlinks

Links in hypertexts clearly contain high-quality semantic clues that are lost upon a purely term-based classifier, but exploiting link information is non-trivial because it is noisy. Authors of TAPER improved the classification accuracy of TAPER by exploiting hyperlink information. They found out that: Simply adding terms from neighbor texts will make error rate even higher. This can be seen as a signal-to-noise issue. There may be so many irrelevant terms that fitting-the-noise effects could become inevitable with realistic training corpus sizes.

**Radius-one specialization:** During testing, the algorithms will have the following general form. First, the classifier will grow a neighborhood graph around the test document. It will use features directly, or derived from the neighborhood to classify the test document. In the case where all immediate neighbors of the test documents are exposed, this completes the algorithm. So radius-one means the direct first level in links or out links. Since on average one page typically has more than 8-10 citations or out-links, a radius-one neighborhood already has the potential to slow down a classifier by a significant factor.

Experiments with text from neighbors show that neighbor text, or tagging(classifier can distinguish local, non-local terms) don't help. Reason is that the heuristic feature selection and approximate class models got overwhelmed by the signal-to-noise ratio.

Rather than using raw data from records of the parents, one can pre-process those raw attribute values into a synthesized feature, for example, the class of parent/neighbor. This is called *feature engineering.* In hypertext classification, if the classes for all documents neighboring an particular document were known, replacing each hyperlink in this document with the class ID of the corresponding document, could provide a highly distilled and adequate (for classification) representation of this document's neighborhood.

If some or all of the neighbor classes are unknown, we need a bootstrap mechanism: we first classify the unclassified documents from the neighborhood(for example by using a terms-only classifier) and then use this knowledge to classify the given document in the same way as above. This procedure can be performed iteratively, to increase the quality of the classification. Let $\triangle = \delta_i, i = 1, 2, ..., n$ be a set of documents and these are linked by directed links $i \rightarrow j$. Let $G(\triangle)$ be the graph defined by these documents and links. Let $\tau_i$ represent the text of document i and let $T$ represent the entire collection of text corresponding to $\triangle$. Let $c_i$ be the class assignment of document i and let $C$ the set of class assignments for the entire collection $\triangle$. Assuming that there is probability distribution for such collections, we want to choose C such that $Pr(C|G,T)$ is maximum. The general form for the classifier is shown in following:

$$Pr[C|G,T] = \frac{Pr[C,G,T]}{Pr[G,T]} = \frac{Pr[G,T|C]Pr[C]}{Pr[G,T]}$$

where Pr[G,T] = $\sum_{C'} Pr[G,T|C']Pr[C']$. To incorporate neighborhood class information into classifier, we chose $c_i$ to maximize $Pr(c_i|N_i)$, where, in this context, $N_i$ represents the collection of all the known class labels of all the neighbor documents. This collection can be further decomposed into in-neighbors and out-neighbors: $N_i = \{I_i, O_i\}$. Now we are faced with choosing $c_i$ to maximize:

$$Pr(N_i|C_i)Pr(C_i)$$

For $Pr(N_i|c_i)$ we will assume independence of all the neighbor classes. That is:

$$Pr(N_i|c_i) = \prod_{\delta_j \in I_i} Pr(c_j|c_i, j \rightarrow i) \prod_{\delta_k \in O_i} Pr(c_k|c_i, i \rightarrow j)$$

A pseudocode sketch follows:

```
Given test node A
Construct a radius-r subgraph G around A
```

```
Assign initial classes to all documents in G using local text
Iterate until consistent:
    Recompute the class for each document based on
        local text and class of neighbors
```

Experiments show that text-based and text-and-link-based classifiers use of the order of 50,000 features. In contrast, the link-based classifier use only 15. Thus it is very fast. However text-and-link-based classifier always perform the best.

**Radius-two specialization:** Co-citation is well-studied in linked corpora such as academic papers. Documents that cite or are cited by many common documents may be regarded as similar. Citation-based similarity or a combination of citation-based and term-based similarity can then be used to perform unsupervised clustering of documents. Bridges are common documents hinting that two or more pages belong to the same class, without committing what that class could be. Two documents $\delta_1$ and $\delta_2$ are said to be bridge-connected if there exists a document $\beta$ pointing to both $\delta_1$ and $\delta_2$; $\beta$ is the bridge. To go from $\delta_1$ to $\delta_2$ one traverse edge $(\beta, \delta_1)$ against its direction and then $(\beta, \delta_2)$. We call this an IO-path because we follow an Inlink to $\beta$ followed by an Outlink. We call $\beta$ an IO-bridge for $\delta_1$ and $\delta_2$. *TAPER with IO-bridges* is tested as follows:

1. Consider a sample of documents from Yahoo! such that each is IO-bridged to at least one other page in Yahoo!. Randomly divide the set into 70% for training and the rest for testing.

2. For training, following features from each document $\delta$ is used: all the documents in the training or testing set that are IO-bridged to the document under scrutiny. These have known class paths in topic taxonomy. All prefixes of these paths are used to construct an engineered document $\delta_{\prime}$.

3. For testing, the class paths for all documents in the training set that are IO-bridged, but not the test set are used. These features are input to TAPER as usual.

One further optimization is to use *Bridges with locality:* check if the classes of the closest classified pages before or after the link to page A equal; if so, we include that classes as a feature in the engineered page $A'$ . However the dramatic boost in accuracy is at some cost of coverage.

### 4.3 Automatic Categorization II – OpenGrid and ODP

The best result from automatic categorization is reported in [32], which still has error rate as high as 21% even with all kinds of optimization to improve the results. Manual categorization faces the scalability problem. The librarians' work can hardly keep up with the fast growth of the Web. No company could ever hire enough experts to categorize the Web. ODP(Open Directory Project) allows thousands of volunteers who are familiar with some specific topics to classify sub-directories. It is a centralized system. As for ranking, they simply rank homepages as cool pages and not-so-cool ones. Currently its directory database is open source and used by many big names such as Netscape, AltaVista. M. Lifantsev [25] proposed a solution(OpenGrid system) which intends to utilize the thousands of surfers' opinion and comments on the pages to rank the documents in the Web. Different from ODP, OpenGrid is a distributed system utilizing all potential web surfers' opinions and not restricted to limited number of registered volunteers as ODP.

The proposed solution is based on a small but crucial extension of HTML standard together with corresponding modifications to a search engine. The modification of HTML standard is that we add two optional fields to a hyperlink:

- Classifying field, named *cat*, whose values can be for example formed in a similar way current Web directories are structured.
- A field indicating evaluation of the page on some fixed scale (including positiveness and negativity), named *rank*.

In this case a link from a site www.newsexperts.foo of the following form:

```
<A href="http://www.somenews.foo/" cat="/News/Computers" rank="80%">
Good computer news</A>
```

would mean that the author of www.newsexperts.foo considered the information on www.somenews.foo to be *good computer news* with a quality rank of 80% out of 100%.

A link of the form:

```
<A href="http://www.somenews.foo/" cat="News/World" rank="10%">
Some world news</A>
```

means that the author of www.newsexperts.foo considered international news on www.somenews.foo to have rank of 10%.

And a link of the form:

```
<A href="http://www.somenews.foo/" cat="/News/Business" rank="-30%">
Misinterpreted business news</A>
```

means that www.newsexperts.foo considered business news on www.somenews.foo to have rank of -30%. That is, they think that trusting or using those news might harm to some extent; that is www.newsexperts.foo do not recommend one to rely on business news from www.somenews.foo with strength of negative recommendation being 30%.

Links without categorization values and rank values would mean that there is no statement about the contents of the referred page made by the referring site.

These all are very subjective opinions of www.newsexperts.foo about the contents of www.somenews.foo. But when one computes "computer news" rank of www.somenews.foo based on *all* such opinion-loaded links from all sites worldwide and the "computer expert", "computer news expert", etc. ranks of all these sites, one will get the very precise opinion about the quality of computer news on www.somenews.foo as expressed by everybody on the Web who has such voting links and whose vote either directly or indirectly has influenced the computation of this particular www.somenews.foo rank.

This rank computation scheme again has a recursive(cyclic) nature very similar to the one used in Google. But one difference is that for each page there might be many ranks reflecting its value in different categories.

Lifantsev said that he did not claim that some particular rank computation scheme is *the* best. The point is that once we have all this voting data, one can implement many different strategies to compute ranks reflecting many different things. The OpenGrid project will give everyone the ability to make those searches and get those results easily.

However, OpenGrid still remains as a proposal, no system is running yet. The suggested extension hasn't been used in today's HTML files and this will be an obstacle of the OpenGrid project. This project needs Web page authors to express their opinions explicitly.

# 5 Measuring the Web

There are many questions related to the statistics of Web itself: how big is the web? How fast does the Web grow? What is the proportion of pages in French, XML, etc? How do various search engines compare? Bharat and Broder described a technique in [4] to compare the coverage of different search engines. They measure search engine coverage and overlap through random queries.

The idea behind their approach is simple: Consider sets A and B of size 4 and 12 units respectively. Let the size of their intersection $A \cap B$ be 2. Suppose that we do not know any of these sizes but can sample uniformly from any set and check membership in any set. If we sample uniformly from $A \cap B$, we will find that about $\frac{1}{2}$ of the samples are in A as well. Hence the size of A is approximately 2 times the size of the intersection $A \cap B$. Similarly we will find that the size of B is approximately 6 times the size of $A \cap B$, then B is about 6/2=3 times the size of A. More formally, let Pr(A) represent the probability that an element belongs to the set A and let $Pr(A \cap B|A)$ be the conditional probability that an element belongs to both sets given that it belongs to A. Then $Pr(A \cap B|A) = size(A \cap B)/size(A)$ and similarly, $Pr(A \cap B|B) = size(A \cap B)/size(B)$, and therefore size(A)/size(B) = $Pr(A \cap B|B)/Pr(A \cap B|A)$.

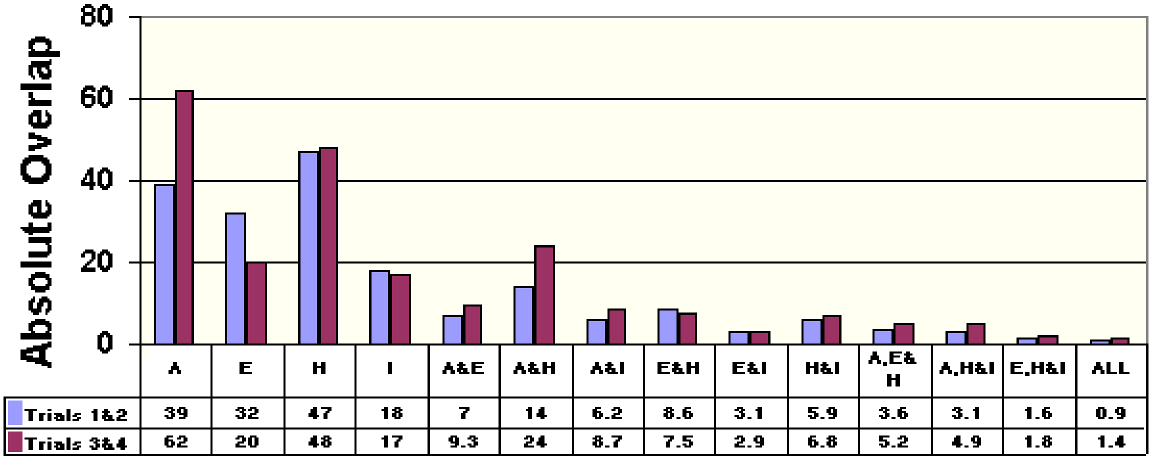
To implement this idea two procedures are needed:

- **sampling** A procedure for picking pages uniformly at random from the index of a particular engine.
- **checking** A procedure for determining whether a particular page is indexed by a particular engine.

Choosing pages uniformly from the entire web is practically infeasible. It would require either collecting all valid URLs on the Web, which requires constructing a better Web crawler than any existing one, or the use of sampling methods that do not explore the entire Web. Such methods, based on random walks are not easily applicable to the Web, since the Web is a directed graph with highly non-uniform degrees and has many small cuts. Little is known about the graph structure of the Web. Hence the length of the random walks required to generate a distribution close to the uniform may be extremely large.

The solution to this used in [4] is to make use of the search engines themselves to generate page samples. First they build a lexicon of the Web and their associated frequencies on the Web. A random URL is generated from a search engine by firing a random query at it and selecting a random URL from the result set. Both disjunctive and conjunctive queries are experimented. All queries tends to introduce a query bias towards large, content rich pages. Also search engines usually returns the first few hundreds matches and in fact may not even compute the remaining matches. The experiment in [4] select the random page from the first 100 pages, thus introduce rank bias. Checking is also query-based. To test a page with a given URL has been indexed by another search engine, first a query meant to strongly identify the page is constructed. Ideally this strong query will bring one exact same page from the tested search engine if that page is indexed in it. However, due to web duplication, still multiple results might prompt. The actual matching requirements can be any of the following:

- **Full URL Comparison** In this case only if the same (normalized) URL is returned by the tested search engine.



| | A | E | H | I | A&E | A&H | A&I | E&H | E&I | H&I | A,E&H | A,H&I | E,H&I | ALL |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 🟦 Trials 1&2 | 39 | 32 | 47 | 18 | 7 | 14 | 6.2 | 8.6 | 3.1 | 5.9 | 3.6 | 3.1 | 1.6 | 0.9 |
| 🟥 Trials 3&4 | 62 | 20 | 48 | 17 | 9.3 | 24 | 8.7 | 7.5 | 2.9 | 6.8 | 5.2 | 4.9 | 1.8 | 1.4 |

Figure 8: Normalized estimates for all intersections (expressed as a percentage of total joint coverage) where A-AltaVista, I-Infoseek, E-Excite, H-HotBot

- **High Similarity** Retrieve all the results pages and compare their content. If they are of high similarity, then the page is deemed as present.
- **Weak URL Comparison** Only a host name comparison is done.
- **Non Zero Result Set** Any URL returned by the strong query is considered a match.

We can see from Figure 8 that in November 1997, only 1.4% of all URLs indexed by the search engines were estimated to be common to all of them. The maximum coverage by any search engine was by AltaVista, which had indexed an estimated 62% of the combined set of URLs. In July 1997, HotBot was the largest, with 47% of the combined set of URLs, and the four engine intersection was only 0.9%.

The ratios we obtained seem consistent with other estimates of the sizes of these search engines at the time. Search Engine Watch reported the following search engine sizes (as of November 5, 1997): AltaVista = 100 million pages, HotBot = 80 million, Excite = 55 million, and Infoseek = 30 million pages. The October 14, 1997 press release from AltaVista also states a 100 million page database. Hence if the size of AltaVista was 100 million pages, then the total coverage of all the search engines should have been about 100 million/0.62, or roughly 160 million pages in November 1997 (see Figure 9). Similarly, the estimate that the four-engine intersection was 1.4% leads us to estimate that only roughly 2.25 million pages were common to all search engines in November 1997.

There is a large interest in finding patterns in and computing statistics of search engine query logs. Silverstein et al. [8] presents their analysis of a very large AltaVista query log. They show that web users type in short queries, mostly look at the first 10 results only, and seldom modify the query. The correlation analysis showed that the most highly correlated items are constituents of phrases. This result indicates it may be useful for search engines to consider search terms as parts of phrases even if the users did not explicitly specify them as such. The query data set contains queries collected over 43 days from 2 August 1998 to 13 September 1998. Table 3 summarizes the data set.

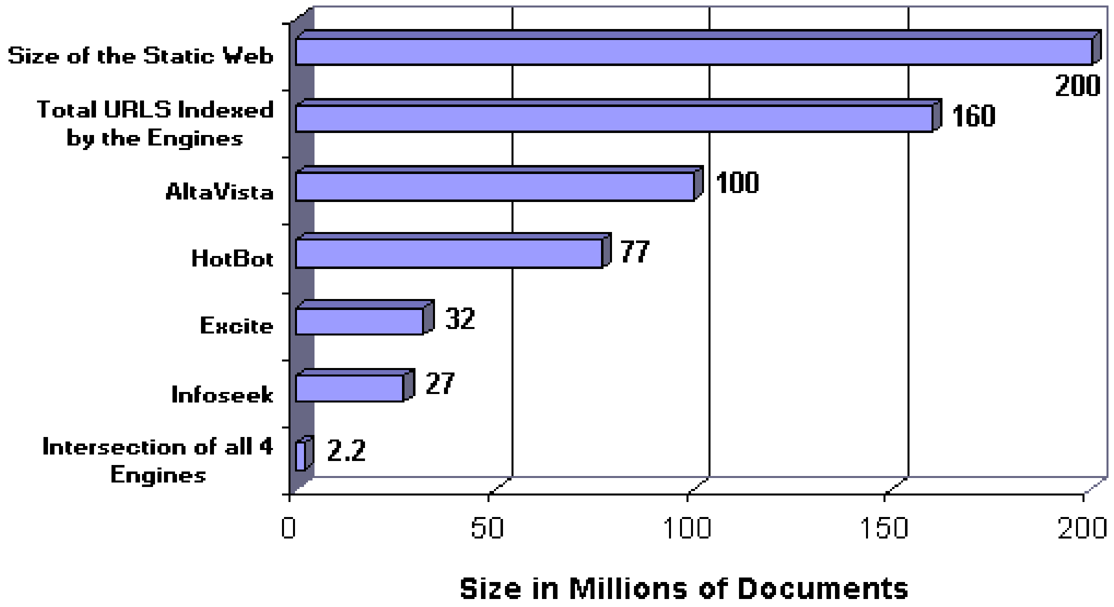


Figure 9: Absolute size estimates for November 1997.

| | |
|---|---|
| Total number of bytes | 300,210,000,000 |
| Total number of requests | 993,208,159 |
| Total number of non-empty requests | 843,445,731 |
| Total number of non-empty queries | 575,244,993 |
| Total number of unique,non-empty queries | 153,645,050 |
| Total number of sessions | 285,474,117 |
| Total number of exact-same-as-before requests | 41,922,802 |

Table 3: Statistics summarizing the query log contents used in the experiments. Empty requests had no query terms. A request consists of either a new query or a new requested result screen. Exact-same-as-before requests had the same query and requested result page as the previous request. The total number of non-empty, unique queries gives the cardinality of the set consisting of all queries.

| | | | |
|---|---|---|---|
| 0 terms in query | 20.6% | max terms in query | 393 |
| 1 terms in query | 25.8% | avg terms in query | 2.35 |
| 2 terms in query | 26.0% | stddev of terms in query | 1.74 |
| 3 terms in query | 15.0% | $>$ 3 terms in query | 12.6% |

Table 4: Statistics concerning the number of terms per query Only distinct queries were used in the count; queries with many result screen requests were not up-weighted. The mean and standard deviation are calculated only over queries with at least one term.

| | | | |
|---|---|---|---|
| 0 operators in query | 79.6% | max operators in query | 958 |
| 1 operators in query | 9.7% | avg operators in query | 0.41 |
| 2 operators in query | 6.0% | stddev of operators in query | 1.11 |
| 3 operators in query | 2.6% | $>$ 3 operators in query | 2.1% |

Table 5: Statistics concerning the number of operators – $+, -$, and, or, not, and near – per query. Only distinct queries were used in the count; queries with many result screen requests were not up-weighted.

Fully 15% of all requests were empty requests, i.e. requests containing no query terms. of the non-empty requests, 32% consisted of a request for a new result screen, while 68% consisted of a request for the first screen of a new query.

1. **Analysis of Individual Queries** Table 4, Table 5 summarize the statistics concerning the terms and operators in single query.

2. **Analysis of Query Duplicates** Data shows that 25 most common queries form fully 1.5% of the total number of queries asked in a 43 day period, despite being only 0.00000016% of the unique queries. The fact that almost two-thirds of all queries are asked only once in a 6 week period indicates that information needs on the web are quite diverse, or at least are specified in diverse ways.

3. **Analysis of Sessions** One of the most striking observation about sessions is most of them are very short. Fully 63.7% of a llsessions consisst of only one request. Result is that average number of queries per session is 2.02 and the average screens per query is 1.39.

However, above analysis did not distinguish between requests issued by humans and requests issued by robots. In some cases, robot-initiated queries in the AltaVista logs resulted in a number

| | | | |
|---|---|---|---|
| query occurs 1 time | 63.7% | max query frequency | 1,551,477 |
| query occurs 2 times | 16.2% | avg query frequency | 3.97 |
| query occurs 3 times | 6.5% | stddev of query freq | 221.31 |
| query occurs $>$ 3 times | 13.6% | | |

Table 6: Statistics concerning how often distinct queries are asked. Only distinct queries were used in the count; queries with many result screen requests were not up-weighted. Percents are of the 154 million unique queries.

| | | | |
|---|---|---|---|
| 1 query per session | 77.6% | max queries per session | 172325 |
| 2 query per session | 13.5% | avg queries per session | 2.02 |
| 3 query per session | 4.4% | stddev of queries/session | 123.40 |
| > 3 queries per session | 4.5% | | |

Table 7: Statistics concerning the characteristics of query modification in sessions

| | | | |
|---|---|---|---|
| 1 screens per query | 85.2% | max screens per query | 78496 |
| 2 screens per query | 7.5% | 2nd most screens | 5108 |
| 3 screens per query | 3.0% | stddev of screens/query | 1.39 |
| > 3 screens per query | 4.3% | avg screens per query | 3.74 |

Table 8: Statistics concerning the characteristics of result screen requests in sessions

of seemingly wierd results. It is impossible to judge how much automated search techniques skewed the results of this study.

## 6 Conclusion

Web information retrieval has achieved many interesting results. Google from Stanford, and Clever from IBM incorporate the latest technology and algorithms for web searching. HITS algorithm and PageRanking algorithm are two most important algorithms behind above search engines. Besides theses algorithms, we also discussed on load balancing, caching,duplicate elimination. Hierarchical directories is another widely used resource. However, with the fast growth of the size of the Web, manual classification (used in Yahoo!) cannot keep up with the pace. TAPER is the automatic classification system built by IBM. The accuracy of it can be as good as 80%. Machine classification has its own limitation. And OpenGrid project proposes extending the HTML standard and enabling any web user to submit their opinions about the pages they browse. These opinions are collected by some search engines to do the proper ranking and classification. We also discussed many other issues for web information retrieval such as shopping agent to deal with heterogeneity. We conclude this report by giving some heuristic approaches to measure the Web and show some experimental results as well. We poposed an online update solution (Codir) which will save the storage space, speed up the update to the database crawled by the search engine.

## References

[1] H. Garcia-Molina A. Tomasic and K.Shoens. Incremental updates of inverted lists for text document retrieval. In *Proceedings of 1994 ACM International Conference on management of Data (SIGMOD94)*, December 1994.

[2] M. S. Manasse A. Z. Broder, S. C. Glassman and G. Zweig. Syntactic clustering of the Web. In *Proc. of the sixth International World Wide Web Conference [WWW6]*, pages 391–404.

[3] AltaVista. http://www.altavista.com.

[4] K. Bharat and A. Z. Broder. A technique for measuring the relative size and overlap of public web search engines. In *Proceedings of the Seventh International World Wide Conference[WWW7]*, pages 379–388.

[5] K. Bharat and A. Z. Broder. A study of host pairs with replicated content. In *Proc. of 8th International Conference on World Wide Web [WWW99]*, May 1999.

[6] Krshna Bharat and Monika Henzinger. Improved algorithms for topic distillation in a hyperlinked environment. In *21st SIGIR Conference on Research and Development in Information Retrieval*, 1998.

[7] Sergey Brin and Lawrenece Page. The anatomy of a large-scale hypertextual web search engine. *The Seventh International World Wide Web Conference*, April 1998.

[8] J. Marais C. Silverstein, M. Henzinger and M. Moricz. Analysis of a very large altavista query log. In *Technical Report 1998-014, Compaq System Research Center, Palo Alto, CA*, 1998.

[9] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. *USENIX Symposium on Internet Technologies and Systems*, 1997.

[10] T. Chiueh and L. Huang. Efficient real-time index updates in text retrieval systems. *ECSL Technical Report 66*, August 1998.

[11] D. Cutting and J. Pedersen. Optimizations for dynamic inverted index maintenance. In *Proceedings of the International Conference on Information Retrieval(SIGIR'90)*, pages 405–411, Jan. 1990.

[12] R. Duda and P. Hart. *Pattern Classification and Scene Analysis.* Wiley, New York, 1973.

[13] J. P. Callan E. W. Brown and W. B. Croft. Fast incremental indexing for full-text information retrieval. In *Proceedings of the 20th International Conference of Very large Databases (VLDB'94)*, pages 192–202, September 1994.

[14] Excite. http://www.excite.com.

[15] C. Faloutsos and H. Jagadish. On b-tree indices for skewed distributions. In *Proceedings of the International Conference of Very Large Databases (VLDB'92)*, pages 363–374, September 1992.

[16] Google. http://www.google.com.

[17] Wired Digital Inc. http://www.hotbot.com.

[18] Dennis Shasha Theodore Johnson. 2q: A low overhead high performance buffer management replacement algorithm. *Proc. 1994 Very Large Data Bases*, 1994.

[19] Hector Garcia-Molina Junghoo Cho and Lawrence Page. Efficient crawling through url ordering. In *Proceedings of the 8th Intenrational World Wide Web Conference, Tornoto, Canada*, May 1999. also at: http://www7.scu.edu.au/programme/fullpapers/1919/com1919.htm.

[20] N. Shivakumar Junghoo Cho and H. Garcia-Molina. Finding replicated web collection. In *Technical Report,(http://www-db.stanford.edu/pub/papers/cho-mirror.ps), Department of Computer Science, Stanford University*, 1999.

[21] Alistair Moffat Justin Zobel and Ron Sacks-Davis. An efficient indexing technique for full-text database systems. In *Proceedings of the 18th Inter. Conf. on VLDB*, pages 352–362, 1992.

[22] J. Kleinberg. Authoritative sources in a hyperlinked environment. In *Proceedings of the ACM-SIAM Symposium n Discrete Algorithms*, 1998.

[23] S. Brin L. Page. The pagerank citation ranking: Bringing order to the web. In *Technical Report,(http://www-db.stanford.edu/ backrub/pageranksub.ps)*, 1998.

[24] U.C. Berkeley Library. http://www.lib.berkeley.edu/teachinglib/guides/internet/toolstables.html. August 1999.

[25] M. Lifantsev. http://www.cs.sunysb.edu/maxim-cgi-bin/opengrid/, Dec 1998.

[26] Evangelos P. Markatos. On caching search engine results. Technical Report 241, Institute of Computer Science, Foundation for Research and Technology-Hellas, GREECE, 1999. Http://www.ics.forth.gr/proj/arch-vlsi/html_papers/TR241.

[27] O. McBryan. Genvl and wwww: Tools for taming the web. *Proc. 1st International World Wide Web Conference*, 1994.

[28] Yungming Shih Michael Cutler and Weiyi Meng. Using the structure of html documents to improve retrieval. In *USENIX Symposium on Internet Technologies and Systems (NSITS'97)*, pages 241–251, December 1997.

[29] M. Perkowitz and O. Etzioni. Life,death and lawfulness on the electronic frontier. In *International Conference on Computer and Human Interaction*, 1997.

[30] J. E. Pitkow and M. Recker. A simple, yet robust caching algorithm based on dynamic access patterns. *Proceedings of the Second International WWW Conference*, 1994.

[31] Open Directory Project. http://www.dmoz.org.

[32] P. Indyk S. Chakrabarti, B. Dom. Enhanced hypertext categorization using hyperlinks. In *ACM SIGMOD 1998.*

[33] R. Agrawal S. Chakrabarti, B. Dom and P.Raghavan. Using taxonomy, discriminants, and signatures for navigating in text databases. In *VLDB 1998*, 1998.

[34] R. Agrawal P. Raghavan S. Chakrabarti, B. Dom. Scalable feature selection, classification and signature generation for organizing large text databases into hierarchical topic taxonomies. In *The VLDB Journal*, 1998.

[35] G. Salton. *Introduction to modern information retrieval.* McGraw-Hill, 1983.

[36] N. Shivakumar and H. Gracia-Molina. SCAM: a copy detection mechanism for digital documents. In *Proc. of 2nd International Conference in Theory and Practice of Digital Libraries*, June 1995.

[37] N. Shivakumar and H. Gracia-Molina. Building a scalable and accurate copy detection mechanism. In *Proc of 1st ACM conference on Digital Libraries (DL'96)*, March 1996.

[38] George Kingsley Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley Press, 1949.