The Anatomy of A Large-Scale Hyper Textualweb Search Engine

[1]S. Manisha, [2]A. Monisha, [3]G. Nandhini, [4]S. Priyadharcini.
[1,2,3,4] Students of Electronics and Communication Engineering,
Kings College of Engineering,
Punalkulam

Abstract:- **In this paper, we present Google, a prototype of a large-scale search engine which makes heavy use of the structure present in hypertext. Google is designed to crawl and index the Web efficiently and produce much more satisfying search results than existing systems. The prototype with a full text and hyperlink database of at least 24 million pages is available at http://google.stanford.edu/ To engineer a search engine is a challenging task. Search engines index tens to hundreds of millions of web pages involving a comparable number of distinct terms. They answer tens of millions of queries every day. Despite the importance of large-scale search engines on the web, very little academic research has been done on them. Furthermore, due to rapid advance in technology and web proliferation, creating a web search engine today is very different from three years ago. This paper provides an in-depth description of our large-scale web search engine -- the first such detailed public description we know of to date. Apart from the problems of scaling traditional search techniques to data of this magnitude, there are new technical challenges involved with using the additional information present in hypertext to produce better search results. This paper addresses this question of how to build a practical large-scale system which can exploit the additional information present in hypertext. Also we look at the problem of how to effectively deal with uncontrolled hypertext collections where anyone can publish anything they want.**

Keywords:- World Wide Web, Search Engines, Information Retrieval, PageRank, Google

I. INTRODUCTION

The web creates new challenges for information retrieval. The amount of information on the web is growing rapidly, as well as the number of new users inexperienced in the art of web research. People are likely to surf the web using its link graph, often starting with high quality human maintained indices such as Yahoo! or with search engines. Human maintained lists cover popular topics effectively but are subjective, expensive to build and maintain, slow to improve, and cannot cover all esoteric topics. Automated search engines that rely on keyword matching usually return too many low quality matches. To make matters worse, some advertisers attempt to gain people's attention by taking measures meant to mislead automated search engines. We have built a large-scale search engine which addresses many of the problems of existing systems. It makes especially heavy use of the additional structure present in hypertext to provide much higher quality search results. We chose our system name, Google, because it is a common spelling of googol, or 10^{100} and fits well with our goal of building very large-scale search engines.

1.1 Web Search Engines -- Scaling Up: 1994 -2000

Search engine technology has had to scale dramatically to keep up with the growth of the web. In 1994, one of the first web search engines, the World Wide Web Worm (WWWW) [McBryan 94] had an index of 110,000 web pages and web accessible documents. As of November, 1997, the top search engines claim to index from 2 million (WebCrawler) to 100 million web documents (from Search Engine Watch). It is foreseeable that by the year 2000, a comprehensive index of the Web will contain over a billion documents. At the same time, the number of queries search engines handle has grown incredibly too. In March and April 1994, the World Wide Web Worm received an average of about 1500 queries per day. In November 1997, Altavista claimed it handled roughly 20 million queries per day. With the increasing number of users on the web, and automated systems which query search engines, it is likely that top search engines will handle hundreds of millions of queries per day by the year 2000. The goal of our system is to address many of the problems, both in quality and scalability, introduced by scaling search engine technology to such extraordinary numbers.

1.2. Google: Scaling with the Web

Creating a search engine which scales even to today's web presents many challenges. Fast crawling technology is needed to gather the web documents and keep them up to date. Storage space must be used efficiently to store indices and, optionally, the documents themselves. The indexing system must process hundreds of gigabytes of data efficiently. Queries must be handled quickly, at a rate of hundreds to thousands per second.

These tasks are becoming increasingly difficult as the Web grows. However, hardware performance and cost have improved dramatically to partially offset the difficulty. There are, however, several notable exceptions to this progress such as disk seek time and operating system robustness. In designing Google, we have considered both the rate of growth of the Web and technological changes. Google is designed to scale well to extremely large data sets. It makes efficient use of storage space to store the index. Its data structures are optimized for fast and efficient access (see section 4.2). Further, we expect that the cost to index and store text or HTML will eventually decline

relative to the amount that will be available (see Appendix B). This will result in favorable scaling properties for centralized systems like Google.

1.3 Design Goals

1.3.1 Improved Search Quality

Our main goal is to improve the quality of web search engines. In 1994, some people believed that a complete search index would make it possible to find anything easily. According to Best of the Web 1994 -- Navigators, "The best navigation service should make it easy to find almost anything on the Web (once all the data is entered)." However, the Web of 1997 is quite different. Anyone who has used a search engine recently, can readily testify that the completeness of the index is not the only factor in the quality of search results. "Junk results" often wash out any results that a user is interested in. In fact, as of November 1997, only one of the top four commercial search engines finds itself (returns its own search page in response to its name in the top ten results). One of the main causes of this problem is that the number of documents in the indices has been increasing by many orders of magnitude, but the user's ability to look at documents has not. People are still only willing to look at the first few tens of results.Because of this, as the collection size grows, we need tools that have very high precision (number of relevant documents returned, say in the top tens of results). Indeed, we want our notion of "relevant" to only include the very best documents since there may be tens of thousands of slightly relevant documents. This very high precision is important even at the expense of recall (the total number of relevant documents the system is able to return). There is quite a bit of recent optimism that the use of more hypertextual information can help improve search and other applications [Marchiori 97] [Spertus 97] [Weiss 96] [Kleinberg 98]. In particular, link structure [Page 98] and link text provide a lot of information for making relevance judgments and quality filtering. Google makes use of both link structure and anchor text (see Sections 2.1 and 2.2).

1.3.2 Academic Search Engine Research

Aside from tremendous growth, the Web has also become increasingly commercial over time. In 1993, 1.5% of web servers were on .com domains. This number grew to over 60% in 1997. At the same time, search engines have migrated from the academic domain to the commercial. Up until now most search engine development has gone on at companies with little publication of technical details. This causes search engine technology to remain largely a black art and to be advertising oriented (see Appendix A). With Google, we have a strong goal to push more development and understanding into the academic realm. Another important design goal was to build systems that reasonable numbers of people can actually use. Usage was important to us because we think some of the most interesting research will involve leveraging the vast amount of usage data that is available from modern web systems. For example, there are many tens of millions of searches performed every day. However, it is very difficult to get this data, mainly because it is considered commercially valuable. Our final design goal was to build an architecture that can support novel research activities on large-scale web data. To support novel research uses, Google stores all of the actual documents it crawls in compressed form. One of our main goals in designing Google was to set up an environment where other researchers can come in quickly, process large chunks of the web, and produce interesting results that would have been very difficult to produce otherwise. In the short time the system has been up, there have already been several papers using databases generated by Google, and many others are underway. Another goal we have is to set up a Spacelab-like environment where researchers or even students can propose and do interesting experiments on our large-scale web data.

II. SYSTEM FEATURES

The Google search engine has two important features that help it produce high precision results. First, it makes use of the link structure of the Web to calculate a quality ranking for each web page. This ranking is called PageRank and is described in detail in [Page 98]. Second, Google utilizes link to improve search results.

2.1 Page Rank: Bringing Order to the Web

The citation (link) graph of the web is an important resource that has largely gone unused in existing web search engines. We have created maps containing as many as 518 million of these hyperlinks, a significant sample of the total. These maps allow rapid calculation of a web page's "PageRank", an objective measure of its citation importance that corresponds well with people's subjective idea of importance. Because of this correspondence, Page Rank is an excellent way to prioritize the results of web keyword searches. For most popular subjects, a simple text matching search that is restricted to web page titles performs admirably when Page Rank prioritizes the results (demo available at google.stanford.edu). For the type of full text searches in the main Google system, Page Rank also helps a great deal. **.**1.1 Description of Page Rank Calculation Academic citation literature has been applied to the web, largely by counting citations or back links to a given page. This gives some approximation of a page's importance or quality. Page Rank extends this idea by not counting links from all pages equally, and by normalizing by the number of links on a page. PageRank 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. We usually set d to 0.85. There are more details about d in the next section. 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:

$$PR(A) = (1-d) + d\,(PR(T1)/C(T1) + \ldots + PR(Tn)/C(Tn))$$

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.

There are many other details which are beyond the scope of this paper.

2.1.2 Intuitive Justification

PageRank can be thought of as a model of user behavior. We 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 Page Rank. 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 only add the damping factor *d* 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. We have several other extensions to Page Rank, again see [Page 98].Another intuitive justification is that a page can have a high Page Rank if there are many pages that point to it, or if there are some pages that point to it and have a high Page Rank. 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!. Home page 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. Page Rank handles both these cases and everything in between by recursively propagating weights through the link structure of the web.

2.2 Anchor Text

The text of links is treated in a special way in our search engine. Most search engines associate the text of a link with the page that the link is on. In addition, we associate 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, anchors 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, 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 [McBryan 94] especially because it helps search non-text information, and expands the search coverage with fewer downloaded documents. We use anchor propagation mostly because anchor text can help provide better quality results. Using anchor text efficiently is technically difficult because of the large amounts of data which must be processed. In our current crawl of 24 million pages, we had over 259 million anchors which we indexed.

2.3 Other Features

Aside from PageRank and the use of anchor text, Google has several other features. First, it has location information for all hits and so it makes extensive use of proximity in search. Second, Google keeps track of some visual presentation details such as font size of words. Words in a larger or bolder font are weighted higher than other words. Third, full raw HTML of pages is available in a repository.

III.RELATED WORK

Search research on the web has a short and concise history. The World Wide Web Worm (WWWW) [McBryan 94] was one of the first web search engines. It was subsequently followed by several other academic search engines, many of which are now public companies. Compared to the growth of the Web and the importance of search engines there are precious few documents about recent search engines [Pinkerton 94]. According to Michael Mauldin (chief scientist, Lycos Inc) [Mauldin], "the various services (including Lycos) closely guard the details of these databases". However, there has been a fair amount of work on specific features of search engines. Especially well represented is work which can get results by post-processing the results of existing commercial search engines, or produce small scale "individualized" search engines. Finally, there has been a lot of research on information retrieval systems, especially on well controlled collections. In the next two sections, we discuss some areas where this research needs to be extended to work better on the web.

3.1 Information Retrieval

Work in information retrieval systems goes back many years and is well developed [Witten 94]. However, most of the research on information retrieval systems is on small well controlled homogeneous collections such as collections of scientific papers or news stories on a related topic. Indeed, the primary benchmark for information retrieval, the Text Retrieval Conference [TREC 96], uses a fairly small, well controlled collection for their benchmarks. The "Very Large Corpus" benchmark is only 20GB compared to the 147GB from our crawl of 24 million web pages. Things that work well on TREC often do not produce good results on the web. For example, the standard vector space model tries to return the document that most closely approximates the query, given that both query and document are vectors defined by their word occurrence. On the web, this strategy often returns very short documents that are the query plus a few words. For example, we have seen a major search engine return a page containing only "Bill Clinton Sucks" and picture from a "Bill Clinton" query. Some argue that on the web, users should specify more accurately what they want and add more words to their query. We disagree vehemently with this position. If a user issues a query like "Bill Clinton" they should get reasonable results since there is a enormous amount of high quality information available on this topic. Given examples like these, we believe that the standard information retrieval work needs to be extended to deal effectively with the web.

3.2 Differences Between the Web and Well Controlled Collections

The web is a vast collection of completely uncontrolled heterogeneous documents. Documents on the web have extreme variation internal to the documents, and also in the external meta information that might be available. For example, documents differ internally in their language (both human and programming), vocabulary (email addresses, links, zip codes, phone numbers, product numbers), type or format (text, HTML, PDF, images, sounds), and may even be machine generated (log files or output from a database). On the other hand, we define external meta information as information that can be Another big difference between the web and traditional well controlled collections is that there is virtually no control over what people can put on the web. Couple this flexibility to publish anything with the enormous influence of search engines to route traffic and companies which deliberately manipulating search engines for profit become a serious problem. This problem that has not been addressed in traditional closed information retrieval systems. Also, it is interesting to note that metadata efforts have largely failed with web search engines, because any text on the page which is not directly represented to the user is abused to manipulate search engines. There are even numerous companies which specialize in manipulating search engines for profit.

IV.SYSTEM ANATOMY

First, we will provide a high level discussion of the architecture. Then, there is some in-depth descriptions of important data structures. Finally, the major applications: crawling, indexing and searching will be examined in depth.

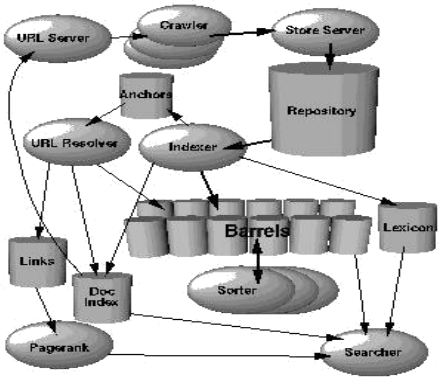
4.1 Google Architecture Overview

In this section, we will give a high level overview of how the whole system works as pictured in Figure 1. Further sections will discuss the applications and data structures not mentioned in this section. Most of Google is implemented in C or C++ for efficiency and can run in 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, 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 inferred about a document, but is not contained within it. Examples of external meta information include things like reputation of the source, update frequency, quality, popularity or usage, and citations. Not only are the possible sources of external meta information varied, but the things that are being measured vary many orders of magnitude as well. For example, compare the usage information from a major homepage, like Yahoo's which currently receives millions of page views every day with an obscure historical article which might receive one view every ten years. Clearly, these two items must be treated very differently by a search engine.

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 (this is a simplification, see Section 4.2.5), 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.

4.2 Major Data Structures

Google's data structures are optimized so that a large document collection can be crawled, indexed, and searched with little cost. Although, CPUs and bulk input output rates have improved dramatically over the years, a disk seek still requires about 10 ms to complete. Google is designed to avoid disk seeks whenever possible, and this has had a considerable influence on the design of the data structures.



4.2.1 BigFiles

Big Files are virtual files spanning multiple file systems and are addressable by 64 bit integers. The allocation among multiple file systems is handled automatically. The

Big Files package also handles allocation and deallocation of file descriptors, since the operating systems do not provide enough for our needs. Big Files also support rudimentary compression options.

4.2.2 Repository

The repository contains the full HTML of every web page. Each page is compressed using zlib (see RFC1950). The choice of compression technique is a tradeoff between speed and compression ratio. We chose zlib's speed over a significant improvement in compression offered by bzip. The compression rate of bzip was approximately 4 to 1 on the repository as compared to zlib's 3 to 1 compression. In the repository, the documents are stored one after the other and are prefixed by docID, length, and URL as can be seen in Figure 2. The repository requires no other data structures to be used in order to access it. This helps with data consistency and makes development much easier; we can rebuild all the other data structures from only the repository and a file which lists crawler errors.

4.2.3 Document Index

The document index keeps information about each document. It is a fixed width ISAM (Index sequential access mode) index, ordered by docID. 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. This design decision was driven by the desire to have a reasonably compact data structure, and the ability to fetch a record in one disk seek during a search 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. This batch mode of update is crucial because otherwise we must perform one seek for every link which assuming one disk would take more than a month for our 322 million link dataset.

4.2.4 Lexicon

The lexicon has several different forms. One important change from earlier systems is that the lexicon can fit in memory for a reasonable price. In the current implementation we can keep the lexicon in memory on a machine with 256 MB of main memory. The current lexicon contains 14 million words (though some rare words were not added to the lexicon). It is implemented in two parts -- a list of the words (concatenated together but separated by nulls) and a hash table of pointers. For various functions, the list of words has some auxiliary information which is beyond the scope of this paper to explain fully.

4.2.5 Hit Lists

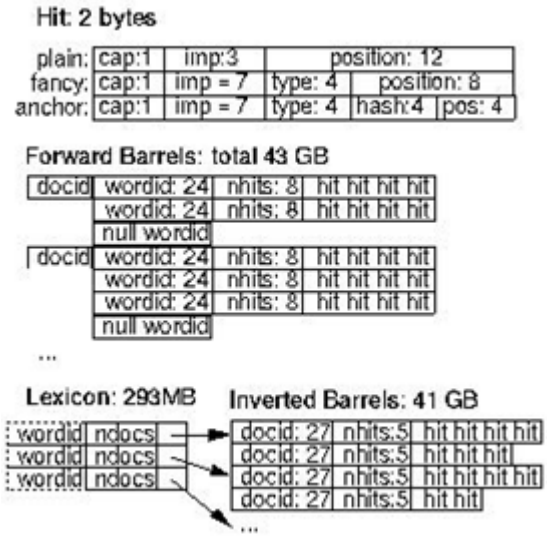
A hit list corresponds to a list of occurrences of a particular word in a particular document including position, font, and capitalization information. Hit lists account for most of the space used in both the forward and the inverted indices. Because of this, it is important to represent them as efficiently as possible. We 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 we 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 hits are shown in Figure 3.Our 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 us some limited phrase searching as long as there are not that many anchors for a particular word. We expect to update the way that anchor hits are stored to allow for greater resolution in the position and docIDhash fields. We use 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.The length of a hit list is stored before the hits themselves. To save space, the length of the hit list is combined with the wordID in the forward index and the docID in the inverted index. This limits it to 8 and 5 bits respectively (there are some tricks which allow 8 bits to be borrowed from the wordID). If the length is longer than would fit in that many bits, an escape code is used in those bits, and the next two bytes contain the actual length.

4.2.6 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 hitlists which correspond to those words. This scheme requires slightly more storage because of duplicated doc IDs 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, we store each word ID as a relative difference from the minimum word ID that falls into the barrel the wordID is in. This way, we can use just 24 bits for the wordID's in the unsorted barrels, leaving 8 bits for the hit list length.

4.2.7 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. Also, this makes development much more difficult in that a change to the ranking function requires a rebuild of the index. We 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. 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.



4.3 Crawling the Web

Running a web crawler is a challenging task. There are tricky performance and reliability issues and even more importantly, there are social issues. Crawling is the most fragile application since it involves interacting with hundreds of thousands of web servers and various name servers which are all beyond the control of the system.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 (we typically ran about 3). 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 a 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. It uses asynchronous IO to manage events, and a number of queues to move page fetches from state to state.It turns out that running a crawler which connects to more than half a million servers, and generates tens of millions of log entries generates a fair amount of email and phone calls. Because of the vast number of people coming on line, there are always those who do not know what a crawler is, because this is the first one they have seen. Almost daily, we receive an email something like, "Wow, you looked at a lot of pages from my web site. How did you like it?" There are also some people who do not know about the robots exclusion protocol, and think their page should be protected from indexing by a statement like, "This page is copyrighted and should not be indexed", which needless to say is difficult for web crawlers to understand. Also, because of the huge amount of data involved, unexpected things will happen. For example, our system tried to crawl an online game. This resulted in lots of garbage messages in the middle of their game! It turns out this was an easy problem to fix. But this problem had not come up until we had downloaded tens of millions of pages. Because of the immense variation in web pages and servers, it is virtually impossible to test a crawler without running it on 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. Systems which access large parts of the Internet need to be designed to be very robust and carefully tested. Since large complex systems such as crawlers will invariably cause problems, there needs to be significant resources devoted to reading the email and solving these problems as they come up.

4.4 Indexing the Web

Parsing -- Any parser which is designed to run on the entire Web must handle a huge array of possible errors. These range from typos in HTML tags to kilobytes of zeros in the middle of a tag, non-ASCII characters, HTML tags nested hundreds deep, and a great variety of other errors that challenge anyone's imagination to come up with equally creative ones. For maximum speed, instad of using YACC to generate a CFG parser, we use flex to generate a lexical analyzer which we outfit with its own stack. Developing this parser which runs at a reasonable speed and is very robust involved a fair amount of work.very count is converted into a count-weight. Count-weights increase linearly with counts at first but quickly taper off so that more than a certain count will not help. We take the dot product of the vector of count-weights with the vector of type-weights to compute an IR score for the document. Finally, the IR score is combined with PageRank to give a final rank to the document.

Indexing Documents into Barrels -- After each document is parsed, it is encoded into a number of barrels. Every word

is converted into a wordID by using an in-memory hash table -- the lexicon. New additions to the lexicon hash table are logged to a file. Once the words are converted into wordID's, their occurrences in the current document are translated into hit lists and are written into the forward barrels. The main difficulty with parallelization of the indexing phase is that the lexicon needs to be shared. Instead of sharing the lexicon, we took the approach of writing a log of all the extra words that were not in a base lexicon, which we fixed at 14 million words. That way multiple indexers can run in parallel and then the small log file of extra words can be processed by one final indexer.

Sorting -- In order to generate the inverted index, the sorter takes each of the forward barrels and sorts it by wordID to produce an inverted barrel for title and anchor hits and a full text inverted barrel. This process happens one barrel at a time, thus requiring little temporary storage. Also, we parallelize the sorting phase to use as many Machines as we have simply by running multiple sorters, which can process different buckets at the same time. Since the barrels don't fit into main memory, the sorter further subdivides them into baskets which do fit into memory based on wordID and docID. Then the sorter, loads each basket into memory, sorts it and writes its contents into the short inverted barrel and the full inverted barrel.

4.5 Searching

The goal of searching is to provide quality search results efficiently. Many of the large commercial search engines seemed to have made great progress in terms of efficiency. Therefore, we have focused more on quality of search in our research, although we believe our solutions are scalable to commercial volumes with a bit more effort. Google considers each hit to be one of several different types (title, anchor, URL, plain text large font, plain text small font, ...), each of which has its own type-weight. The type-weights make up a vector indexed by type. Google counts the number of hits of each type in the hit list. Then

For a multi-word search, the situation is more complicated. Now multiple hit lists must be scanned through at once so that hits occurring close together in a document are weighted higher than hits occurring far apart. The hits from the multiple hit lists are matched up so that nearby hits are matched together. For every matched set of hits, a proximity is computed. The proximity is based on how far apart the hits are in the document (or anchor) but is classified into 10 different value "bins" ranging from a phrase match to "not even close". Counts are computed not only for every type of hit but for every type and proximity. Every type and proximity pair has a type-prox-weight. The counts are converted into count-weights and we take the dot product of the count-weights and the type-prox-weights to compute an IR score. All of these numbers and matrices can all be displayed with thesearch results using a special debug mode. These displays have been very helpful in developing the ranking system.

4.5.2 Feedback

The ranking function has many parameters like the type-weights and the type-prox-weights. Figuring out the right values for these parameters is

something of a black art. In order to do this, we have a user feedback mechanism in the search engine. A trusted user may optionally evaluate all of the results that are returned. This feedback is saved Then when we modify the ranking function, we can see the impact of this change on all previous searches which were ranked. Although far from perfect, this gives us someidea of how a change in the ranking function affects the search results.

V. RESULTS AND PERFORMANCE

The most important measure of a search engine is the quality of its search results.While a complete user evaluation is beyond the scope of this paper, our own experience with Google has shown it to produce better results than the major commercial search engines for most searches. As an example which illustrates the use of PageRank, anchor text, and proximity, Figure 4 shows Google's results for a search on "bill clinton". These results demonstrates some of Google's features. The results are clustered by server. This helps considerably when sifting through result sets. A number of results are from the whitehouse.gov domain which is what one may reasonably expect from such a search. Currently, most major commercial search engines do not return any results from whitehouse.gov, much less the right ones. Notice that there is no title for the first result. This is because it was not crawled. Instead, Google relied on anchor text to determine this was a good answer to the query. Similarly, the fifth result is an email address which, of course, is not crawlable. It is also a result of anchor text. All of the results are reasonably high quality pages and, at last check, none were broken links. This is largely because they all have high PageRank. The PageRanks are the percentages in red along with bar graphs. Finally, there are no results about a Bill other than Clinton or about a Clinton other than Bill. This is because we place heavy importance on the proximity of word occurrences. Of course a true test of the quality of a search engine would involve an extensive user study or results analysis which we do not have room for here. Instead, we invite the reader to try Google for themselves at http://google.stanford.edu.

Query: bill Clinton

http://www.whitehouse.gov/
100.00%(no date) (0K)
http://www.whitehouse.gov/
Office of the President
99.67%(Dec 23 1996) (2K)
http://www.whitehouse.gov/WH/EOP/OP/html/OP_Home.html
Welcome To The White House
99.98% (Nov 09 1997) (5K)
http://www.whitehouse.gov/WH/Welcome.html
Send Electronic Mail to the President

99.86% (Jul 14 1997) (5K)
http://www.whitehouse.gov/WH/Mail/html/Mail_President.html
mailto:president@whitehouse.gov

99.98%
mailto:President@whitehouse.gov
99.27%
The "Unofficial" Bill Clinton
94.06%(Nov 11 1997) (14K)
http://zpub.com/un/un-bc.html
Bill Clinton Meets The Shrinks
86.27% (Jun 29 1997) (63K)
http://zpub.com/un/un-bc9.html
President Bill Clinton - The Dark Side
97.27% (Nov 10 1997) (15K)
http://www.realchange.org/clinton.htm
$3 Bill Clinton
94.73% (no date) (4K)
http://www.gatewy.net/~tjohnson/clinton1.html

5.1 Storage Requirements

Aside from search quality, Google is designed to scale cost effectively to the size of the Web as it grows. One aspect of this is to use storage efficiently. Table 1 has a breakdown of some statistics and storage requirements of Google. Due repository a relatively cheap source of useful data.the search engine requires a comparable amount of storage, about 55 GB.

5.2 System Performance

It is important for a search engine to crawl and index efficiently. This way information can be kept up to date and major changes to the system can be tested relatively quickly. For Google, the major operations are Crawling, Indexing, and Sorting. It is difficult to measure how long crawling took overall because disks filled up, name servers crashed, or any number of other problems which stopped the system. In total it took roughly 9 days to download the 26 million pages (including errors). However, once the system was running smoothly, it ran much faster, downloading the last 11 million pages in just 63 hours, averaging just over 4 million pages per day or 48.5 pages per second. We ran the indexer and the crawler simultaneously. The indexer ran just faster than the crawlers. This is largely because we spent just enough time optimizing the indexer so that it would not be a bottleneck. These optimizations included bulk updates to the document index and placement of critical data structures on the local disk. The indexer runs at roughly 54 pages per second. The sorters can be run completely in parallel; using four machines, the whole process of sorting takes about 24 hours.

5.3 Search Performance

Improving the performance of search was not the major focus of our research up to this point. The current version of Google answers most queries in between 1 and 10 seconds. This time is mostly dominated by disk IO over NFS (since disks are spread over a number of machines). Furthermore, Google does not have any optimizations such as query caching, subindices on common terms, and other common optimizations. We intend to speed up Google considerably through distribution and hardware, software, and algorithmic improvements. Our target is to be able to handle several hundred queries per second. Table 2 has some sample query times from the current version of Google. They are repeated to show the speedups resulting from cached IO.

VI.CONCLUSIONS

Google is designed to be a scalable search engine. The primary goal is to provide high quality search results over a rapidly growing World Wide Web.
Google employs a number of techniques to improve search quality including page rank, anchor text, and proximity information. Furthermore, Google is a complete architecture for gathering web pages, indexing them, and performing search queries over them.

6.1 Future Work

A large-scale web search engine is a complex system and much remains to be done. Our immediate goals are to improve search efficiency and to scale to approximately 100 million web pages. Some simple improvements to efficiency include query caching, smart disk allocation, and subindices. We must have smart algorithms to decide what old web pages should be recrawled and what new ones should be crawled. Work toward this goal has been done in [Cho 98]. One promising area of research is using proxy caches to build search databases, since they are demand driven. We are planning to add simple features supported by commercial search engines like boolean operators, negation, and stemming. However, other features are just starting to be explored such as relevance feedback and clustering (Google currently supports a simple hostname based clustering).

REFERENCES

[1] Best of the Web 1994 -- Navigators http://botw.org/1994/awards/navigators.html Bill Clinton Joke of the Day: April 14, 1997.
[2] http://www.io.com/~cjburke/clinton/970414.html. Bzip2 Homepage
[3] Google Search Engine ttp://google.stanford.edu/ [Abiteboul 97] Serge Abiteboul and Victor Vianu, *Queries and Computation on the Web*. Proceedings of the International Conference on Database Theory. Delphi, Greece 1997.
[4] [Cho 98] Junghoo Cho, Hector Garcia-Molina, Lawrence Page. *Efficient Crawling Through URL Ordering.* Seventh International Web Conference (WWW 98). Brisbane, Australia, April 14-18, 1998.
[5] [McBryan 94] Oliver A. McBryan. GENVL and *WWWW: Tools for Taming the Web. First International Conference on the World Wide Web.* CERN, Geneva (Switzerland), May 25-26-27 1994. http://www.cs.colorado.edu/home/mcbryan/mypapers/www94.ps.