Discussion Report: The Anatomy of a Large scale Hypertextual Web Search Engine (part 2)

Azbayar Demberel
Department of Computer Science
asic@cs.duke.edu

January 24, 2007

1 System design

1.1 Data structures

A good search engine is not just a clever search algorithm. It is a system consisting of several parts that collaborate with each other to produce high quality results efficiently. Google uses several special data structures that are optimized to make the Google parts, namely crawling, indexing and searching, work efficiently.

Google was especially designed to avoid disk seeks whenever possible, as the time to search and fetch data from a disk remained nearly constant throughout the years, whereas the processing speed and I/O rate have steadily increased, and in addition they can always be improved by adding more CPU's or by allocating more bandwidth. Therefore, the design of many Google's data structures reflected this property.

Bigfile Bigfiles are virtual files located on multiple file systems.

Repository Repository is the database of web pages. Google stores full HTML codes of every web pages that it crawls in the repository, compressed by zlib.

Document index The document index stores information about each document. It contains current document status, a pointer into the repository, a document checksum, and various statistics. Also, depending whether the document has been crawled or not, either a pointer to the docinfo or URL list is also stored. The document index is ordered by docID so that the information retrieval can be done quickly.

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 the checksum. Therefore we can quickly find the docID of a particular

URL by computing the URL's checksum and doing a binary search on the checksums.

Lexicon Every possible word (not only English) is assigned a unique ID. ID's are preferred to the real values because they occupy less space in storage, and it is faster to search by numbers than strings. For example let's assume that a word "Database" was assigned an ID 155,015,109. Although 155,015,109 is a huge number, it would require only one comparison to check if a given number equals to it. On the other hand to check if 2 strings are equal we have to do a comparison for every single character, i.e. to check if a given string equals to "Database" we would need to do around 8 comparisons.

Lexicon is essentially a dictionary that maps words to their wordID's. Currently Google's lexicon holds 14 million words and fits in a 256MB main memory.

Hit lists Hit list is a list of occurrences of a particular word in a particular document including its position, font, and capitalization information. Because hit lists are used both in forward and inverted indices, and hence accounts for most of the space, it was important to store it efficiently.

Every word in a web page is categorized into either fancy (if it is a URL, title, anchor text, or meta tag) or plain (everything else). A plain hit consists of: 1 capitalization bit, 3 font size bits, and 12 word position bits. A fancy hit consists of: 1 capitalization bit, 3 font size bits set to 7 to indicate it is a fancy hit, 4 bits to encode the type of fancy hit, and 8 word position bits. 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. To save more space, instead of the actual size, a relative font size (relative to the rest of the document) is stored.

Barrels Barrels are used to store the hit lists. Each barrel holds a set of wordID's. If a documents contains a word that falls into a particular barrel, the docID is recorded in the barrel followed by hit lists for each occurences of the word in the document. Since the words are added to barrels in order of documents, the barrel is sorted on docID's. You can see an example how barrel is stored in the following section.

Forward index Forward index is a mapping of web pages to the words, e.g If we are given a web page, we can find which words this web page contains using the forward index. It is stored in a number of barrels. Hence it is already partially sorted.

Since each barrel contains a certain range of wordID's, instead of the whole wordID, a relative difference from the minimum wordID is stored in the forward index. For example let's assume that a barrel corresponds to a set of wordID's ranging from 100,000,000 to 109,999,999. If we store the actual wordID's we would need 27 bits. Instead if we use relative wordID's from 0

to 9,999,999 we would use 24 bits, saving 3 bits. Google uses 24 bits for the wordID's in the unsorted barrels, leaving 8 bits for the hit list length.

Inverted index Inverted index is the opposite of the forward index. It maps words to web pages, e.g. If we are a given word, we can find which web pages contain that word using the inverted index. For every 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.

Google uses 2 sets of inverted barrels - one set for hit lists which include title or anchor hits and another set for all hit lists. This way, Google first checks the first set of barrels which contain only title and anchor hits and if there are not enough matches within those barrels checks the larger ones.

Anchors file Anchors file contains all the necessary information about links on web pages. It stores information such as where the link points from and to and the text of the link.

1.2 Google architecture

Here is the high level overview of how Google works:

1. First of all, Google needs to obtain all the data from which it will do the search. So a **URL server** creates a list of URL's that need to be fetched and forwards the list to web crawlers.

2. Several distributed **web crawlers** receive the list and start downloading web pages. After they finish downloading web pages, they send them to the store server.

3. The **store server** then compresses the web pages and stores them in the repository.

4. The **indexer** then reads the repository, uncompresses the documents and parses them. Each document is then converted into a set of words called hits. If the hit is an anchor text, the indexer parses out its link and stores it in anchors file with some other information to determine where each link points from and to, and the text of the link. Next, the indexer distributes the hit table table into barrels.

   For example let's assume the docID of www.duke.edu is 13.

   - The word "Admissions" occurs once in www.duke.edu, and it is coded to (i.e. the wordID is) 25320. Let's assume that the position of "Admissions" in the web page is (14,15), the font size is 12 and it is not capitalized.

- The word "Students" occurs twice in www.duke.edu, and it is coded to 13250. Let's further assume that the position of "Students" in the web page are (14,25) and (23,27). Both are not capitalized and the font sizes are 12 and 10 respectively.

Let's assume that this barrel contained a set of wordID's ranging from 0 to 40,000. The barrel would look like the following:

docID	wordID	number of hits	hit1	hit2	...
13	13250	2	14,25; 1; No	23,27; 0; No	
13	25320	1	14,15; 1; No		

Note that instead of the original font sizes 12 and 10, 1 and 0 were stored, showing that the first one was similar to and the second was relatively smaller than the rest of the document.

5. The **URL resolver** will then read the Anchors file, convert relative URL into absolute URL and turn it into a docID. If it is a new URL that hasn't been crawled before, it will be assigned a new docID. 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, showing which web pages have links to which. This database of links will later be used to compute PageRanks.

6. The **sorter** then takes the barrels, inverts it and generates the inverted index.

7. In the meantime, a program called **Dumplexicon** takes the lists of words and their corresponding wordID's to generate a new lexicon which will be used by the searcher.

8. Finally, the **searcher** will use the lexicon to map words into wordID's and use the inverted index together with the pageRank to answer queries.

1.3 Implementation issues

A major complication in the implementation aroused in web crawling. Apart from the obvious challenges of crawling millions of web pages, crawling faced a big obstacle of dealing with social issues. Many people complained about the overload the crawler gave on the web servers; some sent questions like "Wow, you looked at a lot of pages from my web site. How did you like it?"; some were concerned if crawling breached their web sites' security.

The parser also had to deal with a variety of challenges, ranging from HTML typos to parsing non ASCII characters to HTML tags which were nested hundreds deep.

The main difficulty with indexing was the parallelization of the indexing phase as the lexicon needs to be shared. So instead of sharing the lexicon, they wrote a log of all the extra words that were not in a base 14 million word lexicon. That way multiple indexers were able to run in parallel and then the small log file of extra words was processed by one final indexer.

The sorting was done in parallel by simply running multiple sorters, which could process different buckets at the same time. Since the barrels didn't fit into main memory, the sorter further subdivided the barrels into baskets that could fit into memory based on wordID and docID. Then the sorter loaded each basket into memory, sorted it and wrote its contents into the short inverted barrel and the full inverted barrel.

Let us recall that the goal of searching is to provide quality search results efficiently. Much progress has already been made in the "efficiency" of the searches by other commercial search engines. So Google instead focused on the other property - quality of search. So since instead of figuring a way to output the enourmous number of search results quickly, Google put a limit on response time so that once a certain number of results (40,000) were generated, the searcher stops further searches, and outputs the results.

Multi-word searches also caused some complications. In case of multi-word searches, it is important to consider not only the occurence of each word but also the proximity of the words. So in this case 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 were matched up so that nearby hits were matched together. For every matched set of hits, the proximity was computed. The proximity was based on how far apart the hits were in the document and was classified into 10 different value "bins" ranging from a phrase match to "not even close".

2 The search algorithm

Googles uses the following algorithm to evaluate each queries.

1. Parse the query
2. Using the lexicon, convert words into wordID
3. Using the short inverted index, go to the start of the barrels for each word
4. Scan through the barrel until there is a document that matches all the search terms.
5. Using the page rank, compute the rank of that document for the query.
6. If more that 40,000 results have been produced go to step 9
7. If we are in the short barrels and at the end of any doclist, seek to the start of the doclist in the full barrel for every word and go to step 4.
8. If we are not at the end of any doclist go to step 4.
9. Sort the documents that have matched by rank and return the top k

3 Performance

3.1 Quality of search

The quality of search is the most important performance metric of a search engine. By using PageRank, anchor text, and proximity, Google performed better for most searches than commercial search engines. For example when a search on "Bill Clinton" was performed, most commercial search engines returned a lot of irrelevant results and many didn't even return obvious ones like whitehouse.gov, whereas all of the Google's results were of high quality.

3.2 Storage

As one of the main goals of Google was to scale well with the growth of the web, it was important to use storage efficiently. Google used only 53GB's of storage after compression which makes it economically efficient. And when better compression was used, it was possible to reduce the size to mere 7GB's.

3.3 System

To keep the information up to date and quickly make tests in case a major change was made to the system, it is important to have an efficient crawler and indexer. The crawling and downloading the web pages took 9 days, indexing was run in parallel with crawling and sorting took about 24 hours to complete.

3.4 Search

Searching took 1 to 10 seconds to complete, and the majority of the was spent on I/O over NFS. However, as noted earlier, the speed of the search was not a major focus of Google, so it is possible to improve the speed by using optimization techniques such as query caching, distribution, and other algorithmic, software and hardware improvements.

4 Future work

The authors noted that several improvements were needed to be made. For example, extending the research on anchor text and user the text surrounding it, the use of query caching and other techniques so that Google can scale to contain more web pages, and many other. Also one of the more important areas to consider is finding a smart algorithm that decides when to re-crawl, and how to crawl new pages.