Google Search Engine Architecture

1998@Stanford

Paper L1 (pdf is long article, html shorter version)

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

General Statistics

The main statistics of the Google Search Engine Architecture in its original incarnation at Stanford can be summarized in the stats available in Figure 1. Google's view of the Web was a paltry 24M pages of total size 147GiB uncompressed (zlib compressed down to 53GiB), index size was approximately 62GiB for a total of 116GB. This roughly 80% of the web-based corpus size (of 147GiB). Of those 62GiB of the index roughly two-third or 41GiB was the size of the inverted index and the remainder the size of other index-related data structures. As a point of reference in 2015, approximately 1T URLs are crawled, around 30G pages indexed, and roughly 6G queries are satisfied daily (two-thirds of them just by Google).

```
A.  Google's Search Engine Performance in 1998@Stanford
   Active crawlers       :  3-4 typically with 300 open connections/crawler
   Web-page read         : 100/sec concurrently, 600KiB/sec
B.  Google@Stanford   size statistics
   # of Web pages        : 24-25M
   # Crawling (New)      : 9 days per crawl for 25M pages, 50pages/sec
   # Crawling (Existing) : 63hrs for 11M preexisting pages
   #  of URLs/WebServers : 76M URLs and 500,000 Web Servers
   #  of Email addresses : 1.7M Email addresses and 1.6M 404 messages
C. Google Indexer Performance
   Indexer Speed         : 54 pages/second, 2-4weeks total
   Sorter  Speed         : 24 hours on 4 PCs
   PageRank              : < 1 day
D.  Google Query Speed   : 1-10 seconds/query,  1K/day
E. Data Collected from Web-pages
   Link Data             : fromURL, toURL, anchor-text
                           [25M URLs, 260M anchors, 322M links]
   Short Barrel (4GiB)   : [title and anchor text only]
   Long  Barrel (37GiB)  : [all]
F. Google Index Data
   Fetched Information   : 147GiB
   Compressed            :  53GiB (3 to 1)
   Inverted Index Full   :  41GiB
       Short Barrel      :    4GiB
       Long  Barrel      :   37GiB
   Anchors               : 6.6GiB [260M]
   Links                 : 3.9GiB
   Lexicon               :  .3GiB [14M words]
   DocIndex              : 9.7GiB
G. Source:  Brin & Page [ http://www-db.stanford.edu/~backrub/google.html ]
```

Figure 1: Google Search Engine 1998

Note that `B` stands for byte, `bit` for bit, `b` for nothing/nonsense. 1MB may be 1,000,000B if it denotes hard-disk drive capacity and 1,048,576B of main memory. Use instead SI's `1MiB` for 1,048,576B i.e 2^{20}B or `1GiB, 1TiB, 1PiB` if you need more!

1.1 Corpus size (Google@1998) and URLs. In Google@1998, the corpus was 24-25M web-pages and approximately 80M URLs. The cumulative size of the web-pages was 150GiB uncompressed, roughly 50GiB compressed; overal index size (all data structures) was 62GiB.
1.2. Performance (Google@1998). Google's crawler used to crawl on the average 50pages/sec and fetch all 25 million pages in roughly 9 days. The indexer and sorter used to index 54pages/sec sustained over a 24 hour period using only 4 workstations. The PageRank computation could be completed in less than a day.
1.3. Link information (Google@1998). The 25 million pages included more than 260M anchors that were indexed, and close to 520M hyperlinks (some might point to inside a given page) with a total of 322M links stored in the Links structures of Google (to be explained).

Google@1998: Search, sort, and select.

1.4.1 Search. Google's first action was to **search** i.e. **crawl** the web, locate 500K web-servers containing 25M pages and fetch those pages at rates ranging from 50pages/sec, to 100pages/sec for the 3-4 instances of a crawler running at any time. Such a task could be completed in less than 2 weeks (9 days) from scratch or in less than 3 days if preexisting (known) pages were to be refetched.

1.4.2 Sort. The **indexing process** in Google is realized by the `indexer` and the `sorter`. The indexer deals with parsing pages and generating the forward index at a speed of 54pages/sec in 1998 completing its task in roughly 2-4 weeks. The sorter then inverts the forward index using **sorting** to generate the inverted index i.e. the **index** in 24 hours using 4PCs.

1.4.3 Select. The **selection** procedure requires the actions of the **query engine** or what is known in Google terminology (see bottom right of Figure 2) as the **searcher**.

The crawler was implemented in Python but the indexer mainly in C and C++.

Google System Architecture Components

The original Google System Architecture is depicted in Figure 2 and its major components are highlighted below. (A component is a program or data structure.)

2.1 URL server. Provides a list of URLs to be sent to and retrieved by the crawler.

2.2 Crawler. A distributed crawler is used with 3-4 instances running at any time (in 1998-2000).

2.3 StoreServer. Compresses Web-pages (using zlib) and then stores them into the

2.4 Repository, that holds all retrieved pages. The indexer can use the Repository and the error log of the Crawler to restore the index.

2.5 Indexer. Parses documents and generates Google's index.

2.6 Anchors. Contains a list of (possibly relative) URLs retrieved by the Indexer.

2.7 URLresolver. Converts Anchors-related data into absolute URLs and uses this information to generate the

2.8 Links, data that show how URLs reference each other (i.e. depict the link structure of the Web).

2.9 Lexicon. Contains a list of unique words found (14million, 256MBytes of data in 2000). Resides in main memory (and the size of the main memory determines its size).

2.10 DocIndex. Contains information about documents. Every document has a docID which is a hash of its contents, a pointer to its location to the Repository along with other useful information (eg. URL and title).

2.11 Barrels. Contain all hitlists generated by the indexer from the Repository. Google originally used 64 Barrels, one barrel for a group of consecutive words. It contains information about hits i.e. is an inverted index of all words in all documents (every word records also positional, font and TITLE/ANCHOR relative indexing information).

2.12 Sorter. It's the program that sorts the Barrels based on word (actually wordID) to generate the hitlists.

2.13 PageRank. It's the ranking method used by Google to rank documents based on "popularity". Such information is based solely on the graph structure available in Links.

2.14 Searcher. It's the search interface for Google. It parses queries, converts them into a form google understands (e.g. converts words into wordIDs using the Lexicon), accesses the Barrels to determine the documents. The ranking of the output documents is a combination of PageRank and an evaluation of the results of the hits according to the query.

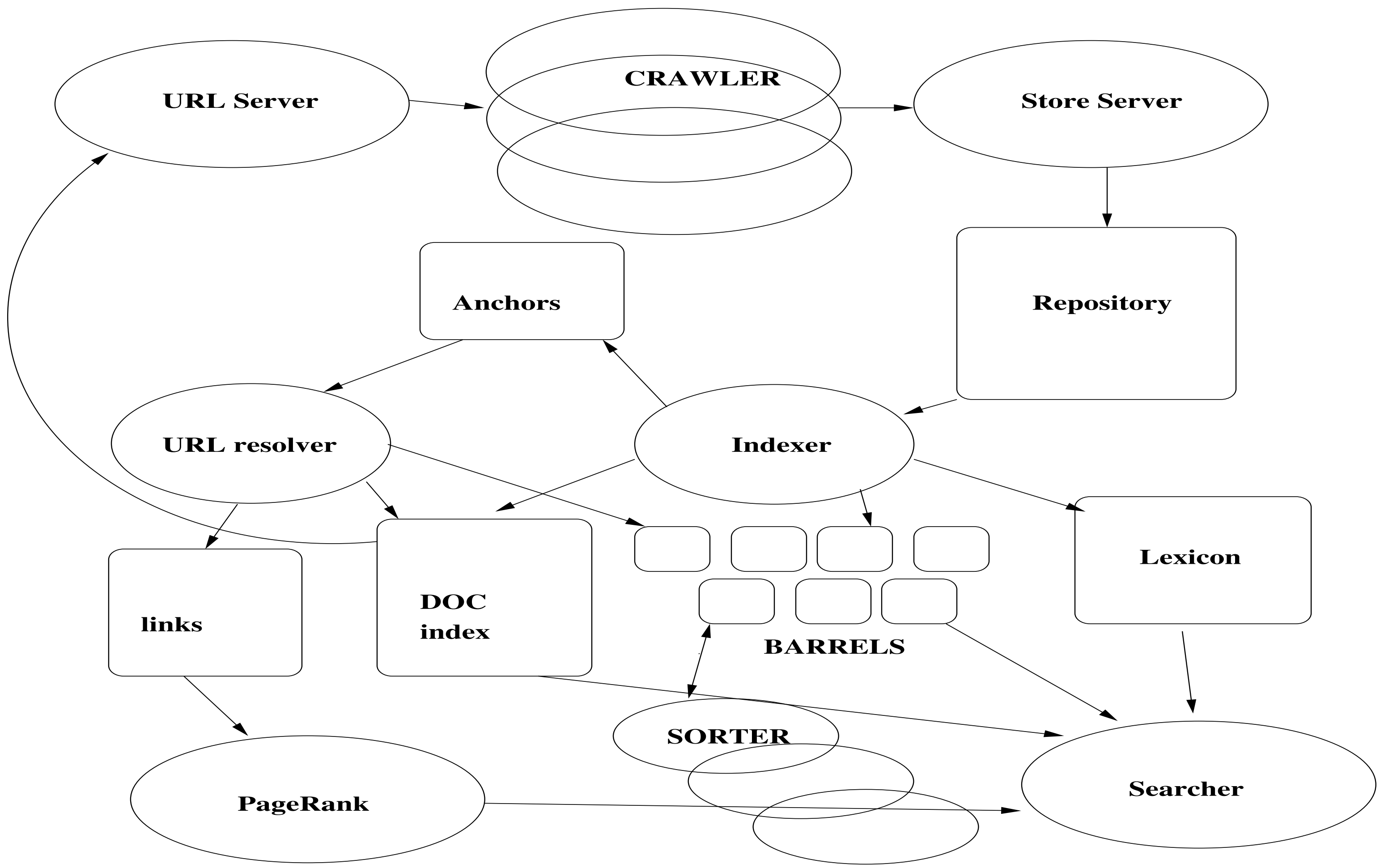

Figure 2: Google System Architecture.

2.1. URLserver. The URL server maintains lists of URL that were supplied to it in previous runs of the crawler(s) by the crawler(s). In subsequent runs, it is the URLserver that schedules what a crawler is going to crawl: it sends lists of URLs to the active crawlers (3-4 active ones at a time in 1997/1998). The URLserver is implemented in Python. The URLserver maintains metadata (data on the "data"/URLs) on these URLs (eg. last time fetched, etc).

ACTION 2.1 (URLserver). The URL server supplies lists of URLs to the crawler(s).

2.2. Crawler. Google's crawler (1998 edition) was a distributed one written mainly in Python, with 3-4 instances of it running at any time, maintaining 300 connections and using a visit policy based on Depth First Search (DFS). In 9 days it could crawl and visit roughly 26M arbitrary pages though it could visit and fetch 11M preexisting pages in 63 hours at a rate of 100pages/sec for the 3-4 instances. The 24-25M pages indexed at the time included 76M URLs, and 1.7M email addresses. All 4 instances of a crawler could work on 100pages/sec, fetching roughly 600KiB/sec.

ACTION 2.2 (Crawler) The crawler receives lists of URLs from the URLserver and then goes to the Web to visit those pages, fetch its contents, and then sends these pages to the StoreServer. Metadata or other info may be transmitted to the URLserver and subsequently stored into the Repository by the StoreServer.

2.3. StoreServer. The storeserver compresses the pages received from the crawler using zlib and stores them into the Repository; metadata related to the files (i.e. data on those data) are also stored there. Google made the decision of using zlib (compression ratio of 3 to 1) over bzip (ratio of 4 to 1) based on speed considerations than compression performance.

ACTION 2.3 (Storeserver) The Storeserver receives pages from the crawler(s), compresses them and stores them into the Repository.

2.4. Repository. The Repository stores compressed pages received by the STORESERVER from the Crawler who fetched them from the Web. They are stored in compressed zlib format in the form of packets that maintain a docID, URL length, the URL, length/size of the compressed page itself, and the raw compressed page. Such pages are then made available to the Indexer. (**Action 2.4.**)

ACTION 2.4 (Repository) The Repository data can be read directly by the Indexer during the indexing process. All of Google's index can be built from information stored in the Repository, without accessing the Web or interacting with the crawler; the error log of the crawler is only used to discard incomplete pages that were not fetched due to errors.

INDEXING PROCESS: 2.5 Indexer and 2.12 Sorter. The indexing process includes the indexing functions realized by Google@1998 and which are performed by the (a) the **Indexer**, and (b) the **Sorter**. Through ACTION 2.4 the Indexer first accesses the decompresses pages stored in the Repository, and using the Error Log of the crawler then generates the forward index. The forward index is a hitlist, i.e. a list of hits, with a hit being an occurrence of a word in a document. A partially sorted forward index generated by the Indexer is then stored into a data structure known as the Barrels (there are long and short Barrels). Subsequently the Sorter would read and invert (i.e sort) the forward index from the Barrels to generate the index also to be stored in the Barrels. Auxiliary functions are being performed by the URL resolver and the PageRank component that ranks links (aka resolved pairs of URLs) and several data structures are created.

2.5 Indexer. The indexer performs two actions:
(a) **ACTION 2.5.1** that generates the forward index stored in the long and short barrels using as input the Error Log of the crawler, and the data in the Repository; these are the only two sources from which the Indexer accepts input; this action creates/populates a variety of other structures including the Lexicon and Doc Index, and
(b) **ACTION 2.5.2** that creates the **Anchors** data structure using the text information found inside a web-page.

Action 2.5.1:Forward Index. The indexer through ACTION 2.4

i. reads a compressed web-page and its metadata from the Repository,

ii. uncompresses and parses it, and

iii. generates the forward index that is hitlists which are tuples that include the word parsed, positional information (word offset) in document, count information, fontsize, capitalization, location/context, and

iv. then assigns those hitlists to the (a) the **short barrels** that deal with words appearing in the **title,meta** tag of an HTML document or the URL or the **anchor** text that points to another document (**fancy hits** of short barrels), and (b) the **long or full barrels** that deal with words appearing everywhere (including the title and anchor related text).

The **Forward index** is partially ordered (by wordID and position).

ACTION 2.5.1 continued: Hitlist is a list of Hits. A **hitlist** consists of **hits**. A hit is a **word occurrence** of a word in a document; a document is converted into a set of word occurrences consisting of tuples that are of the form (excluding docID information a hit in Google used to be 2B long): ⟨`docID`, `wordID`, `offset`, `Info`⟩ . The Info field maintains the fontsize of the word, its capitalization, context such as whether it is in the title of a document, anchor, emphasized, higher font size, etc. **The hitlists form a partially sorted forward index**; for a given document its tuples are ordered by `offset`.

There are two types of hits: (a) fancy hits and (b) plain hits. A **fancy hit** is used for words appearing in a URL, title text, anchor text and meta tag of an html document. One bit is used to indicate capitalization, 3 bits are set to the pattern 111, 4 bits to indicate the type of a fancy hit, and 8 bits are being used to either indicate position information (normal) or anchor-related information: in the latter case 4 bits are indicative of position in the anchor text and the remaining 4 bits is a 4-bit hash of the docID containing the anchor/text. Plain hits use 1 bit for capitalization, 3 bits for fontsize encoding relative to the fontsize of the rest of the document (7 of the 8 states used; the 8-th state is 111 to indicate a fancy hit) and 12 bits for word position offset information (0 through 4094; value 4095 is for larger offsets). In a hitlist a hit count precedes the hitlist. *Note that a hit count is either 5 bits or 8 bits depending on whether it is available in the inverted index or the forward index.*

ACTION 2.5.2: Anchors. The **second action** of the indexer is to parse out all links (i.e. anchors) of every page p and store the full or partial URL associated with a page u pointed by p and the accompanying anchor text associated with u into `Anchors`. Note that if page p contains a link to page u the anchor text will be associated with u not p. Thus a tuple of the form
⟨`from_URL_p`, `to_URL_u`, `AnchorTextAssociatedwith_u`⟩ is stored in `Anchors`. Moreover the text `AnchorTextAssociatedWith_u` will be included in the short barrels of document `to_URL_u`, not `from_URL_p` by the URL resolver in `ACTION 2.7.2` to be described later.

URL, URL checksum, docID, word, wordID. Every URL has a checksum (hash) which is considered a fingerprint of the URL (a "shorter and simpler" identifier of it; a URL can be 70 or more characters/bytes long but its unique checksum is just 16-32B long. In addition a docID is associated with a URL and its URL checksum: it is only 4-8B long. A mapping of one to another is to be maintained somewhere in the index. The docID is assigned to the URL by the URLresolver whenever a URL is parsed out of a web-page by the indexer. Note that the Indexer stores those URLs in incomplete form in the Anchors first, and the URL resolver, resolves the full path to the corresponding URL and stores that finalized information in the DOC index. Likewise, for every word of arbitrary length a shorter wordID is maintained that is few bytes long, usually 3-4B. The mapping between the two is available in the Lexicon.

2.12 Sorter: Inversion and (Inverted) Index
The Sorter takes information from the barrels (short or long/full) in the form of a partially sorted **forward index** and sorts this information. The sorting is by `wordID`, first, then for a given wordID by `docID` and then by `offset` for a given (same) wordID and docID pair. This sorting operation is known as **inversion**. The output of this inversion is stored back into the barrels in the form of an **(inverted) index**. In addition it generates a list of wordIDs. Thus both the input and the output of the sorter is available in the barrels. (Figure 2 shows the Sorter connected to the barrels only.)

ACTION 2.12.1 : Inversion. The sorter uses the forward index of the Barrels as input to invert it (sort it) and the output of this inversion is the (inverted) index also stored into the Barrels.

ACTION 2.12.2 : List of wordID. The sorter generates also a list of wordIDs as an output of the inversion process.

2.11 Barrels. The are 64 Barrels (short or long). The forward index is distributed over those 64 barrels. Each barrel stores a range of `wordID`s. For `wordID`s in a given barrel we don't need to store all the bits of the `wordID` but only an offset from the first `wordID` stored in that barrel. Long barrels store data 10 times larger than those stored in the short barrels.

2.11.1 Forward Index in Barrels. For a given document, the forward index contains multiple variable-length tuples. A tuple will contain a `docID` to identify the current document, followed by multiple entries that consist of a `wordID` to identify the word whose hits will be recorded, an `nhits` record (8 bits) that will record the number of occurrences of the word in `docID` followed by a variable length array of `2 x nhits` bytes since a hit (and there are `nhits` of them) is 2B long. The length of this record is thus `nhits*2+4` bytes. The last tuple for a given `docID` records a `NULL wordID` in the corresponding field to indicate the end of the record related to this `docID`. Information related to the next `docID` is then made available.

2.11.2 (Inverted) Index in Barrels. The same barrels that store the forward index store the (inverted) index except that in the latter the tuples are sorted by `wordID`. The Lexicon records for a given `wordID` the number of documents `ndocs` that contain it. A pointer to the barrel(s), long or short, is also available. This pointer points to an area of `ndocs` consecutive variable length records. Each such record is a tuple containing a `docID`, the document containing that word, an `nhits` field (5bits) that records the number of occurrences within `docID` of `wordID`, followed by a variable length record of `2xnhits` bytes recording those `nhits`. The length of this record is thus `nhits*2+4` bytes.

2.6 Anchors and
2.7. URL Resolver.

During the indexing process and ACTION 2.5.2 the `Indexer` creates and populates the `ANCHORS` data structure. This data structure is being read by the URL Resolver and it (i) converts relative URL into absolute URLs, and (ii) absolute URLs into docIDs.

ACTION 2.7.1.: Absolute URL and docID conversion. Using as input ANCHORS information a relative URL are converted into an absolute URL by the URL resolver. Moreover an absolute URL is also assigned a docID value. This information is maintained in the DOC index.

ACTION 2.7.2.: Short Barrel Update. As explained in ACTION 2.5.2 when in a document p an anchor to document u is found with an associated Anchor text, a tuple
⟨`from_URL_p`, `to_URL_u`, `AnchorTextAssociatedwith_u`⟩
is stored in `Anchors`. In this ACTION 2.7.2 the text associated with u will be included in the short barrels of document `docID_u`, not `docID_p` by the URL resolver. Note that in Barrel we do not store URLs, only docIDs; thus `from_URL_p`, `to_URL_u` will converted into docIDs by the URL resolver first; this is possible because of prior ACTION 2.7.1.

ACTION 2.7.3: Links. As explained in ACTION 2.5.2, the URL resolver read ANCHORS and converts a
⟨`from_URL_p`, `to_URL_u`, `AnchorTextAssociatedwith_u`⟩
into a pair of docIDs that will be stored into Links
⟨**from_docID_p**, **to_docID_u**⟩.

ACTION 2.7.4: URLs with no docID. Periodically, the URL resolver merges a list of URLs that have not been assigned a docID with the list of URLs that have been assigned a docID. This latter file is File3 maintained by DOC index.

2.8. Links. It stores pairs of docIDs (not URLs) generated by the URL resolver through ACTION 2.7.3

Indexing Process: Lexicon

The motivation behind the data structures used by Google is to reduce disk accesses since all of the information maintained by the Index can not (or could not) be stored into main memory. Whether it is 1990 or 2010, one disc access still takes 8-10msecs i.e. it is slow.

2.9 Lexicon. The lexicon is maintained so that it fits into main memory. Its size is thus kept small by not maintaining rare words in main memory. Google's lexicon in 1998 used to maintain in main memory roughly 14 million words. Rare words are attached to the lexicon and maintained in a separate disk file, not in main memory. The lexicon is organized as a `hash table` of pointers to a list of words separated by nulls (i.e. \0 in Unix/C parlance). The indexer generates lists of words (or wordIDs) through the lexicon, and the sorter creates a new one list of wordID as a result of the inversion operation. The merging of these two lists is undertaken by the SEARCHER and generates a new lexicon. Moreover, the lexicon for a given `wordID` maintains the number of ndocs that contain the wordID and a pointer to the barrel(s).

2.9.1 Heaps' Law. This empirical law states that for a corpus containing N words, the number of distinct words n forming its dictionary/vocabulary/lexicon can be approximated by a formula

$$n = kN^{\beta}$$

wher β varies between 0.4 and 0.6 and a typical value if $\beta = 0.5$, and k is a constant that varies between 10 and 100.

Example. For a text with 16 billion words, $k = 100$, $\beta = 1/2$, and thus $n = 100\sqrt{N}$ we obtain a $n = 14,000,000$.

Indexing Process: Doc Index, PageRank, BigFiles

2.10 DocIndex. The DocIndex contains information about documents (web-pages) in fixed width and length (ISAM for Indexed Sequential Access Method). The information is indexed by `docID`.

File1 in DocIndex. One file maintained includes for each document its `docID`, current status information, a pointer to the Repository for the document, a document checksum and other statistics and a pointer to either File2 or File4.

File2 in DocIndex (for crawled pages). File1 for a crawled web-page includes a pointer to this File2. File2 contains the URL of the document whose entry in File1 points to this location (of File2), and its title information. **File4** is a list of URLs (no title info).

File3 in Doc Index (for non-crawled pages). File1 for a non-crawled web-page includes a pointer to this File3. This File3 contains a list of URL checksums and corresponding docIDs. File3 is sorted by URL checksum. It is used to convert a URL into the corresponding `docID`. When Google wants to find the docID of a URL, the URL is first converted into its checksum, and a binary search is performed on File3 on the checksum to retrieve its docID. The URL resolver periodically merges this sorted by URL checksum file with a list of URLs whose docID is to be computed.

2.13 PageRank. It computes the PageRank number for every docID stored in the Links using the Links information. It will provide this information to the Searcher for the set of documents submitted by the Searcher to PageRank.

BigFiles. It is virtual files spanning multiple disks and filesystems that store Google index-related information (web-pages, indexes, barrels, etc). They use 64-bit addressing.

Searching Process: Searcher

2.14 Searcher. It is the query engine of Google. It performs a variety of tasks that includes the following ones.

2.14.a Query satisfaction. It uses the Lexicon and the inverted index (stored in the Barrels) to answer queries. Moreover it uses the Lexicon to offer alternatives if it realizes that a search term might have been misspelled.

2.14.b Rank results. It uses and retrieves PageRank information to rank results of queries. Given that Barrels store docIDs, the ranks are retrieved through the PageRank interface. (Be reminded that Links store link information of docIDs as well.)

2.14.c Context Information. For the average user, a docID means nothing. The Searcher uses DocIndex to convert a docID back into a URL (File1 and File2 of DOC index), and then using this information from DOC Index (File1), it retrieves the compressed file from the Repository to display readable result information such as the URL not docID, title information of the document (File2 of DOC Index), context information i.e. the text that includes the word occurrence appearing as a search term in the query.

2.14.d ACTION 2.14.1 below. The Searcher performs the merging of information that merges the sorted list of word/wordIDs generated by the Sorter and a list of word/wordID available in the current Lexicon and maintained by the indexer into a new Lexicon.

ACTION 2.14.1 : DumpLexicon. The list of wordID/words generated by the Sorter (ACTION 2.12.2) and the current list of wordIDs maintained in the LEXICON by the INDEXER are being merged to generate a new LEXICON that is up-to-date. This merging is performed by the SEARCHER.


```
      Google Query Evaluation Steps

   1. Parse the query.
   2. Convert words into wordIDs.
   3. Seek to the start of the doclist in the short barrel
      for every word.
          Short Barrel: Title and Anchor text of web-pages
          DocList     : For every wordID list of docIDs containing.
   4. Scan through the doclists until there is a document
      that matches all the search terms.
   5. Compute the rank of that document for the query.
   6. 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.
        Full Barrel : All of the page is indexed
                     (10x larger than Short Barrel in 1998).
   7. If we are not at the end of any doclist go to step 4.
   8. Sort the documents that have matched by rank,
       and return the top k.

Source:http://www-db.stanford.edu/~backrub/google.html
```

Figure 3: Google Query Evaluation Process

An example of an information retrieval system is that used by Google. Figure 3 describes the steps undertaken by a Google search engine in answering a query. Although it represents how Google worked in 1998, it is still relevant today. Several steps may have been enhanced and highly optimized since then, but the fundamental underlying operations have little changed.

So far we have given an outline of what web-searching entails and is about. One might ask the following question. If web-searching through a search engine is an information retrieval problem, that has been studied for more than 30 years, what is left to be told about it? Is it an obvious problem or is there something that still needs to be told about it?

Google Query valuation

For every query q and document d_j Google establishes an information retrieval based valuation that depends on five or so factors that includes title, anchor, URL, large-font and small-font occurrence of a word in the query in the document. Each such factor is assigned a weight (say a number between 0 and 1) and that weight is then multiplied by a count that shows how often the word of the query appears in the document as factored. This determines the information-retrieval based value/rank of the document relative to the query. This becomes in Google parlance the **relevance score** or the similarity $s(q, d_j)$ measure. For the document in hand, its PageRank is also determined let us call it $r(d_j)$. The sum of the two scores/ranks determines the final rank of the document d_j relative to query q and is used to order the results in the form that will be presented to the user: $R(q, d_j) = s(q, d_j) + r(d_j)$.

```
To be, or not to be, that is the question:
Whether 'tis Nobler in the mind to suffer
The Slings and Arrows of outrageous Fortune
```

Exercise 1 *Read the paper introduction the Google search engine also available from the following URL.* `http://www-db.stanford.edu/~ backrub/google.html` *Try to answer after reading the paper the following questions whose answers are in the paper or may need to be further searched upon.* **(a)** *What was the size of google's dictionary/vocabulary around 1998? Justify your answer.* **(b)** *What is the number of bits used for* `docID`*? You might need information beyond the paper for that.* **(c)** *On the same one page you used for (a), (b) describe the data structures used by google for indexing.* **(d)** *Does google (circa 1998) use a hash table for the dictionary? What do they use? Why ? What does Google call the dictionary?*