Google Search Engine Architecture

1998@Stanford

Paper L1

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 2014 ; distribution outside this group of students is NOT allowed.*

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 could index 24 million pages of total size 147GB in uncompressed form (zlib compressed down to 54GB), index size was 62GB giving a total of 116GB, close 80% of the web-based corpus size (of 147GB). Of those 62GB of the index 4GB is the short index stored in the short barrels, 37GB the full index and the lexicon is less than 1GB, with temporary anchor data using 6.6GB, the docindex 10GB and the link close to 4GB.

```
A.  Google's Search Engine Performance in 1998@Stanford
                            [circa 1998]
   Active crawlers       :  3-4 typically
   Open Connections      : 300
   Web-page read         : 100 concurrently
   Data                  : 600Kbytes/second
B.  Google@Stanford   size statistics (in millions)
   # of Web pages        : 25 (9 days per crawl)
     (Or just 63 hrs to retrieve 11mil of preexisting pages)
   #  of URLs            : 76
   #  of Email addresses : 1.7
   #  of 404 messages    : 1.6
C. Google Indexer Performance
   Indexer Speed         : 54 pages/second
   Sorter  Speed         : 24 hours on 4 PCs
D.  Google Query Speed   : 1-10 seconds/query
E. Data Collected from Web-pages
   Link Data             : fromURL, toURL, anchor-text
                           [25million URLs, 260mil anchors]
   Short Barrel (4.1GB)  : [title and anchor text only]
   Long  Barrel (37GB)   : [all]
F. Google Index Data
   Fetched Information   : 147Gbytes
   Compressed            :  53Gbytes (3 to 1)
   Inverted Index Full   :  41Gbytes
       Short Barrel      :    4.1Gbytes
       Long  Barrel      :   37.0Gbytes
   Anchors               : 6.6Gbytes [260mil]
   Links                 : 3.9Gbytes
   Lexicon               :  .3Gbytes [14mil words]
   DocIndex              : 9.7Gbytes
G. Source:  Brin & Page [ http://www-db.stanford.edu/~backrub/google.html ]
```

Figure 1: Google Search Engine 1998

Some general statistics

The table below shows the search engine development in the past 15-20 years. For the remainder be reminded that 1MB is 1,000,000B and `B` stands for a Byte when it denotes hard-disk drive space and may be 1,048,576B when it refers to main memory. We shall use `bit` for a bit. A lower-case `b` means nothing. If you want to specify 1,048,576B i.e (2^{20} bytes) use the SI system appropriate `1MiB` notation for a mebi-byte, and `1GiB` for a gibi-byte.

		Pages indexed/Corpus size	Queries/day
WWW worm	Mar/Apr 1994	110,000	1,500
Web Crawler	Nov 1997	2,000,000-10,000,000	20,000,000
	2000	1,000,000,000	severalx100,000,000
	2010	1trillion but < 30billion indexed	400,000,000
	2013	about the same	4,500,000,000

1.Corpus size (Google@1998) and URLs. In Google@1998, the corpus was 24million pages and included 80 million URLs. Although the 24 million were approximately 150GB in size and one third of it in compressed form, the index (all data structures) was approximately 62GB.
2. Performance (Google@1998). Google's crawler used to crawl on the average 50 pages per second and fetch all 24 million pages in roughly 9 days. The indexer and sorter used to index 54 pages per second sustained over a 24 hour period using only 4 workstations. The PageRank computation could be completed in few hours.
3. Link information (Google@1998). The 24 million pages included more than 259,000,000 anchors that were indexed, and close to 518,000,000 hyperlinks (some might point to inside a given page), and a total of 322,000,000 links were stored in the LINKS structures.
4. Google@1998: Search, sort, and select. Do not forget that the three action performed by Google relate to the problems of **searching**, **sorting** and `selection` discussed in data structures classes. The `search` component in Google is associated with the **crawling** process. The `indexing process` performs two fundamental opeations: `indexing` performed by the **indexer** and `sorting` performed by the **sorter**. 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

The architecture of a search engine has evolved relative to the simplified view of the previous Subject. The original Google System Architecture is depicted in Figure 2 and we describe below the major components of the Google High-level Architecture.

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

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

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

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

Indexer. Parses documents and generates Google's index.

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

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

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

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).

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).

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).

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

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.

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.

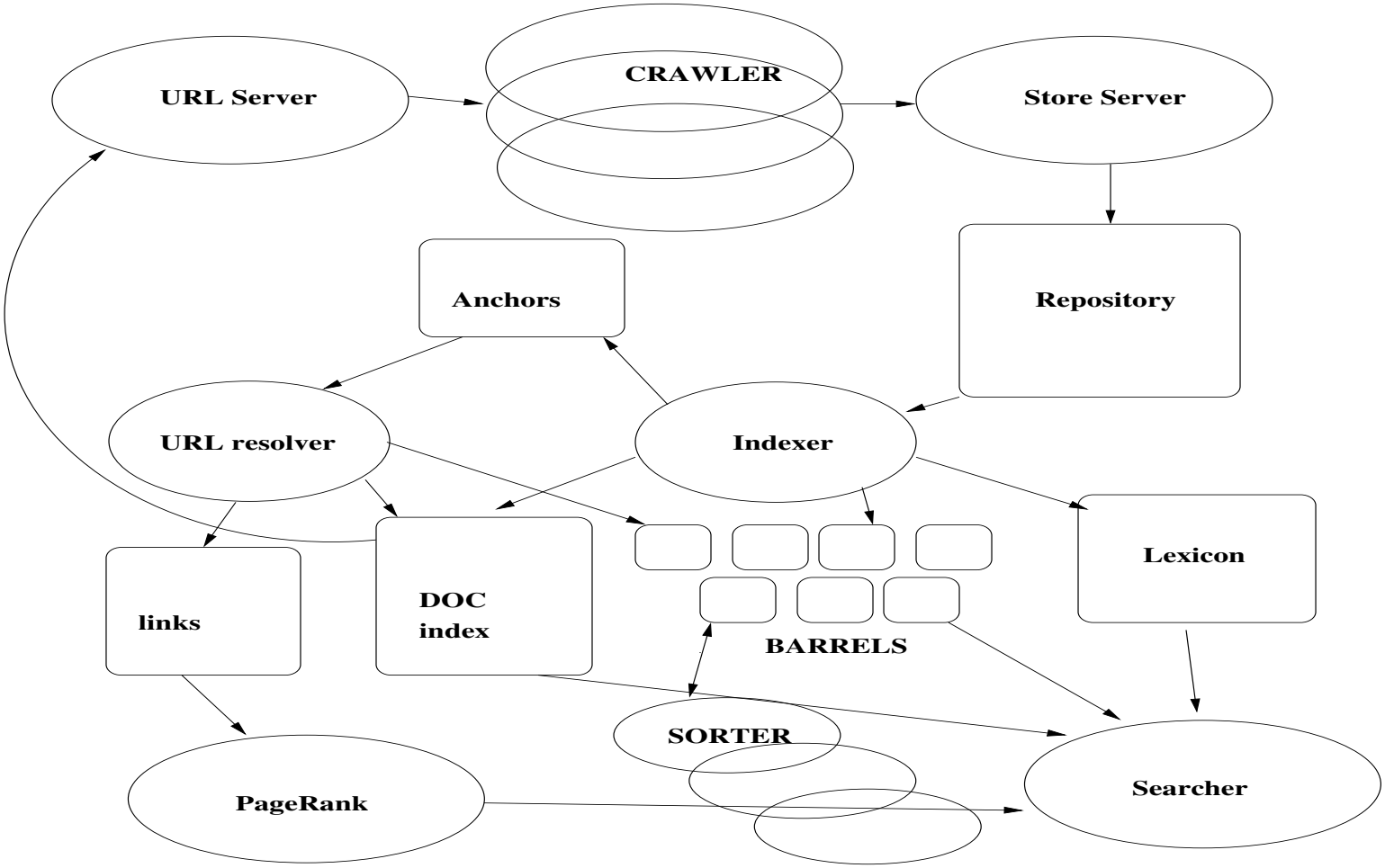

Figure 2: Google System Architecture.

A. URLserver. The URLserver sends list of URLs to the active crawlers (up to 4 at a time in 1997/1998). The URLserver is implemented in Python.(**Action 1.**)

B. 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 connection and using a visit policy based on Depth First Search. In 9 days it could crawl and visit the 24-26 million pages indexed at that time, but after the first run, it could visit 11 million pages in 63 hours. The 24 million pages indexed included 76 million URLs and 1.7 million email addresses. All 4 instances could work on 100 pages per second, feteching roughly 600KiB of data for approximately 6KiB per page.

Action 2 (Crawler) The crawler fetches lists of URLs from the URLserver and then goes to the Web to visit pages and on receipt it sends these pages to the StoreServer.

C. StoreServer. The storeserver compressed the pages received from the crawler using zlib and stores them into the Repository. (**Action 3.**) Google made the decision of using zlib (compressed ratio of 3 to 1) over bzip (ratio of 4 to 1) based on speed considerations than compression performance.

D. Repository. The Repository stores compressed pages received from the STORESERVER and such pages are then made available to the Indexer. (**Action 4.**)

Note: All of Google's index can be built from information stored in the Repository, without accessing the Web or interacting with the crawler.

E. Indexer (indexing functions)

The indexing functions in Google@1998 are performed by the (a) the **Indexer**, and (b) the **Sorter**.

E. Indexer (description). The Indexer performs two tasks: indexing and sorting. Every URL has a checksum (hash) and a docID which is assigned to it by the URLresolver whenever a URL is parsed out of a web-page by the indexer.

TASK A of Indexer.

i. reads information from the Repository,

ii. and uncompresses it,

iii. then parses it,

iv. and generates hitlists that are tuples including the word pasrsed, positional information in document, count information, fontsize, and capitalization, and

v. assigns those hitlists to the (a) the sort barrels that deal with words appearing in titles and anchors (fancy hitlists), and (b) the long/full barrels that deals with words appearing elsewhere in the text.

A.1 Hitlists. A **A.1 hitlist** is a **word occurrence** and thus a document is converted into a set of word occurrences consisting of tuples that are of the form ⟨`docID`, `word`, `Info`⟩. The hitlists form a partially sorted forward index.

A.2 Forward index. The Indexer distributes the hit lists into a partially sorted forward index distributed into long and short barrels. The short barrels contain the hitlists (i.e. words) appearing in anchor or title text, whereas the long/full barrels contain words appearing anywhere else in the text/document parsed.

TASK B of Indexer. The **second task** of the indexer is to parse out all links of every page and store the full URL, or partial URL and accompanying anchor text into `Anchors`. Thus a tuple of the form
⟨`fromURL`, `toURL`, `AnchorText`⟩
is stored in `Anchors`.

Note: All of Google's index can be built from (a) information stored in the Repository, without accessing the Web or interacting with the crawler, (b) by the Indexer, (c) using only the error log of the crawler to filter the information in the Repository that is incomplete or of no use.

F. Sorter (indexing functions)

F. Sorter (TASK C). The Sorter takes information from the barrels (long/full or short) in the form of a partially sorted **forward index** and sorts this information. The output is sent back to the barrels in the form of an **inverted index**. In addition it generates a list of wordID. (**Action 9** part of **TASK C** of the indexing functions.) Thus both the input and the output of the sorter is available in the barrels. The input is usually sorted by possibly docID, and offset, and the output by wordID, possibly docID, and offset.

DumpLexicon (TASK D). The list of wordID generated by the Sorter (Action 9) and the current list of wordIDs maintained in the LEXICON that were generated by the INDEXER are being used to generate a new LEXICON that is up-to-date. (**Action 10** part of **TASK D** of the indexing functions.)

Sorter connectivity. Figure 2 shows the Sorter connected to the barrels only.

Indexer connectivity. The Indexer, however, is connected to the barrels, anchors, repository, docindex, and lexicon.

G. URL Resolver. It reads the `ANCHORS` data structure that was created by the `Indexer` and (i) converts relative URL into absolute URLs, and (ii) absolute URLs into docIDs. (**Action 6**.)

Action 7 by URL Resolver. It uses the anchor text to associate it with the destination URL/docID not with the URL/docID of the page containing it. Thus it puts into the SHORT BARRELS a tuple $\langle$**to_docID**, **AnchorText**$\rangle$ generated from text available in `from_URL / from_docID` . This involves converting URL into docIDs.

Action 8 by URL Resolver. It generates from $\langle$**from_URL**, **to_URL**, **AnchorText**$\rangle$ a link pair that will be stored in LINKS: $\langle$**from_docID**, **to_docID**$\rangle$.

H. Links. It stores pairs of docIDs (not URLs) generated by the URL resolver through Action 8.

I. PageRank. It computes 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.

J. Searcher. It uses the lexicon and the inverted index to answer queries. It uses PageRank to rank results (docID) of queries. It uses DocIndex to convert docID back into URLs, and retrieves files from the Repository to display readable result information (URL, title of result document, context information). 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.

Some data structures

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 be stored in main memory. Whether it is 1990 or 2010, one disc access still takes 8-10msecs i.e. it is slow.

DS1. BigFiles. It is virtual files spanning multiple disks and filesystems that store Google index-related information (web-pages, indexes, etc). They use 64-bit addressing.

DS2. Repository. The Repository stores web-pages retrieved by the crawler in compressed zlib format in the form of packets that maintain a docID, URL length and the URL, and of the page itself in compressed form (along with the length of it).

DS3. DocIndex. The DocIndex contains information about pages in fixed width ISAM (Indexed Sequential Access Method) indexed by `docID`. This facilitates indexing by docID.

One file maintained includes for each document its `docID`, current status information, pointers to the Repository for the document, a document checksum and other statistics.

In addition for a crawled document it contains the URL and title information of it. For non-crawled documents it contains a pointer to a data structure that is a URLlist, a list of all URLs not crawled yet.

A second file is maintained that is sorted by URL checksum and contains this URL checksum and the corresponding `docID`. This 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 then because this second file is sorted by the checksum, a binary search is performed to determine its docID. The URL resolver periodically merges this sorted by URL checksum file with a list of URLs whose docID is to be computed.

DS4. Lexicon. The lexicon is maintained so that it fits in main memory. Its size is thus kept down 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. 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, and the sorter creates a new one merging it with the current lexicon to generate a new lexicon.

Lexicon size and an empirical Law

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

$$v = kn^{\beta}$$

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 $v = 100\sqrt{n}$ we obtain a $v = 14,000,000$. This is consistent with the size of Google's lexicon in 1998.

More data structures: Hitlists

DS5. Hitlists. The hitlists are built by the Indexer while parsing pages stored in compressed form in the Repository. A hitlist is the list of occurrences of words in a given document (and incorporates word offset and other information about a word). Most of the space in the forward or the inverted index deals with hitlist information. In Google a hitlist is stored in hand-optimized form. For a given word a hit is stored in 2 bytes of memory. There are two types of hits: (i) fancy hits and (ii) plain hits.

Fancy hits. A fancy hit is for words appearing in a URL, title, anchor text or meta tags. For such hits 1 bit is a capitalization bit, 3 bits are set to the pattern `111` to indicate it is a fancy hit, and 4 bits show the type of a fancy hit. The remaining 8 bits have one of two interpretations: (a) normal that uses those 8 bits for positional information, and (b) anchor interpretation that uses 4 of those 8 bits as anchor position within an anchor and the remaining 4 bits as a hash of the docID of the document containing the anchor.

Plain hits. A plain hit contains words appearing anywhere in the document/text (including title and anchor). Again 1 bit is a capitalization bit, 3 bits indicate 7 states (other than `111`) of the font-size relative to the rest of the document, and the remaining 12 bits indicate a word-based offset position starting with 1 and going all the way to 4095. If a word appears in a position in the text higher than 4095 then the 4096 index is used to indicate this.

In a hitlist a hit count always precedes the hitlist, the list containing all the relevant hits.

Note that the hit count is 8 or 5 bits depending on whether it is available in the FORWARD index or the INVERTED index.

More data structures: Forward Index and Inverted Index

DS6. Forward Index. The forwards index is distributed over 64 barrels. In each barrel a range of `wordID`s is stored. 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. (Be reminded that the sorter generates such offsets from prior discussion.)

A 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 2 bytes long. 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.

DS7. Inverted Index. The same barrels that store the forward index store the inverted index except that in the latter the tuples of the forward index are sorted by `wordID`.

The lexicon contains information for each `wordID` that includes not only its `wordID` and a pointer to an array that contains the word itself, but also an `ndocs` entry that denotes the number of distinct documents (`docID`s) containing `wordID`, and also a pointer to the barrel associated with this `wordID`. The pointer points to a location of `ndocs` consecutive records. Each such record is a tuple containing a `docID`, the document containing that word, an `nhits` field that records the number of occurrences within `docID` of `wordID`, followed by a variable length record of `2xnhits` bytes recording those `nhits`.

As we have mentioned, there are two sets of barrels, the short barrels storing information related to words appearing in anchors and document titles and long barrels for words appearing anywhere in the document (including anchors and title).

The short barrel(s) is one-tenth of the size of the long barrel(s) and thus indexing takes ten-times longer to work on long barrels than short barrels.

Because of this distinction between short and long barrels, the Google query engine and query process (see next page) relies on the short barrels to generate docIDs first, i.e it relies on title and anchor text; when this is exhausted, then the long barrels are searched.


```
   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
```


Exercises

Practice makes perfect

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?*