Google for the Linguist on a Budget

András Kornai and Péter Halácsy

Budapest University of Technology Media Research Center
{kornai,hp}@mokk.bme.hu

Abstract

In this paper, we present GLB, yet another open source and free system to create and exploit linguistic corpora gathered from the web. A simple, robust web crawl algorithm, a multi-dimensional information retrieval tool, and a crude parallelization mechanism are proposed, especially for researchers working in resource-limited environments.

Introduction

The GLB (Google for the Linguist on a Budget) project grew out of the realization that the current open source search engine infrastructure, in particular the `nutch/lucene/hadoop` effort, is in many ways inadequate for the creation, refinement, and testing of language models (both statistical and rule-based) on large-scale web corpora, especially for researchers working in resource-limited environments such as startup companies and academic departments unlikely to be able to devote hundreds, let alone thousands, of servers to any project.

Section 1 describes `nut`, a simple, robust web crawl algorithm designed with the needs of linguistic corpora gathering in mind. Section 2 details `luc`, an information retrieval tool that facilitates querying along multiple dimensions. We leave `had`, a crude parallelization mechanism sufficient for load balancing dozens (or perhaps hundreds) of CPUs and offering fine control over rerunning versions of different processing steps, to the concluding Section 3.

Many other ways out of the budget predicament have been proposed, and in the rest of this Introduction we discuss these briefly, not so much to criticize these approaches as to highlight the design criteria that emerged from considering them. First, what do we mean by being on a budget? The Google search appliance (GSA) starts at $30,000, which puts it (barely) within reach of grants to individual investigators, and certainly within the reach of better endowed academic departments. Unfortunately, the GSA is an entirely closed system, the internals cannot be tweaked by the investigators, and the whole appliance model is much better suited for a relatively static document collection than for rapid loading of various corpora. Also, the size limitations (maximum of 500k documents) make the GSA too small for typical corpus-building crawls, and the query language is not flexible enough to handle many of the queries that arise in linguistic practice. There is no breaking out of separate software and hardware costs in the GSA, and as our project is providing free (as in beer) and open source (LGPL) software, our goal was to design algorithms that run well on any (x86-) 64-bit system with 8-16 GB memory and 5-10 TB attached storage – today such systems are available at a quarter of the cost of the GSA.

Another, in many ways more attractive, approach is to rely on the Google API, Alexa, or some similar easily accessible search engine cache. Methods of building corpora by selective querying of major search engines have been pioneered by Ghani (2001), and a set of very useful bootstrapping scripts was made available by Baroni and Bernardini (2004). But being parasitic on a major search engine has its own risks. Many of these were discussed in Kilgarriff (2007) and require no elaboration here, but there are issues, in particular *integration, query depth,* and *replicability,* which are worth further discussion.

First, there are many corpora which may be licensed to the researcher but are not available on crawlable pages (and thus are not indexed by the host engine at all). Such corpora, including purpose-built corpora collected by the researchers themselves, can be extremely relevant to the investigation at hand, and the integration of results from the web-based and the internal corpora is a central issue. This applies to the community-based solution proposed by Kilgarriff as well, inasmuch as researchers are often bound by licenses and other contractual obligations that forbid sharing their data with the rest of the community, or even uploading it to the Sketch Engine CorpusBuilder.

Second, with the leading search engine APIs, deeper querying of the sort provided by the Linguist's Search Engine (Resnik and Elkiss, 2005) or the IMS Corpus Workbench (see `http://www.ims.uni-stuttgart.de/projekte/CorpusWorkbench`) is impossible, a matter we shall return to in the concluding Section. Finally, owing to the ever-changing nature of the web, the work is never replicable. This is quite acceptable for brief lexicographic safaris where the objective is simply to find examples, but in the context of system building and regression testing replicability is essential. The main design requirements for GLB stemming from these considerations are as follows. The system must

1. run on commodity hardware (less that $15k per node)
2. hold a useful number of pages (one billion per node)
3. provide facilities for logging, checkpointing, repeating, and balancing subtasks
4. have a useful throughput (one million queries per day)
5. not be a drain on external resources/goodwill

There are various tradeoffs among these requirements that are worth noting. Trivially, relaxing the budget constraint (1) could lead to more capable systems in regards to (2) and (4), but the proposed system is already at the high end

of what financially less well endowed researchers, departments, and startups can reasonably afford. In the other direction, as long as the reliability of storage is taken out of the equation (a terabyte non-redundant disk space is now below $1k), memory becomes the limiting factor, and the same design, deployed on 500m or just 100m items, becomes proportionally less memory-intensive, so running the system on a modern laptop with 4GB memory is feasible. As described in Section 2, GLB does not mandate storage of web pages as such, the items of interest may also be sentences or words. For smaller corpora (in the 1m page range) it may make perfect sense to change the unit of indexing from pages to words and, if disk space is available, to store more information about a unit than the raw text, e.g. to precompute the morphological analysis of each word (or even a full or partial syntactic parse, see some speculative remarks at the end of Section 3). Finally, we note that the design goal of 1m queries per day (12 queries/sec) may be too ambitious if all reads are taken on the same non-redundant disks: while in principle this is well within the speed and latency capabilities of ordinary disk drives, in practice a drive may not stand up against sustained use of this intensity for long. However, those who cannot afford high quality SANs may also be in less of a need to issue millions of queries.

1. Nut

Replicability means that pages once crawled and deemed useful must be kept around forever, otherwise later versions of some processing step cannot be run on the same data as the earlier version, which would throw into question whether improvements are due to improvements in the processing algorithm itself or simply to better data. This is not to say that all pages must be in the scope of all queries, just that a simple, `berkdb`-style list of what was included in which experiment must be preserved. This is in sharp contrast to full-function crawler databases, which manage information about when a host and a particular page was last crawled, when it was created/last changed, how many in-links it has, etc.

In general, neither link structure nor recency matters a great deal for a linguistic corpus, as made plain by the fact that the typical (gigaword) corpora in common use are composed of literary and news text that are entirely devoid of links and are, for the news portion, several years outdated. The exhaustiveness of a crawl is also a secondary concern, since there are far more pages than we can expect to be able to analyze in any depth. This means that it is sufficient to download any page just once, and we can have near-zero tolerance toward buggy, intermittent pages: connection timeouts and errorful http responses are sufficient reason never to go to the page again. Also, the simplest breadth-first algorithm has as good a chance to turn up linguistically relevant pages as the more complex approaches taken in large-scale crawlers.

Among the public domain crawlers, `heritrix` (see `http://crawler.archive.org`) has been successfully utilized by Baroni and Kilgarriff (2006) to create high quality gigaword corpora, achieving a crawl throughput of 35 GB/day. Our own experience with `heritrix`, `nutch`, and `larbin` was that sustained rates in this range are difficult to maintain. We had the best results the WIRE crawler (Castillo, 2005), 8-10 GB/day sustained throughput for domains outside `.hu` and nearly twice that for `.hu` (the crawls were run from Budapest, see Halácsy et al 2008).

Our main loop is composed of three stages: management, fetching, and parsing. Since most of the time is spent fetching, interleaving the steps could save little, and would entail concurrency overhead. We manage three data sets: downloaded URLs, forbidden URLs (those that have already displayed some error), and forbidden hosts (those with dns resolution error, no route to host, host unreachable). We do not manage at all, let alone concurrently, link data, recency of crawl per host, or URL ordering. This simplifies the code enormously, and eliminates nearly all the performance problems that plague `heritrix, nutch, larbin` and other highly developed crawlers where clever management of such data is the central effort. To speed up name resolution (host,ip) pairs already resolved are stored in a simple hash table, and we ignore issues of hosts with multiple IPs and the existence of CNAMEs. The three lists we maintain are read into memory once and written on disk for the next stage, so nothing is ever overwritten. As a matter of fact, it is sufficient for the fetcher to simply append to the list of downloaded URLs on disk, since duplicate elimination (which is not a big issue here) can happen as part of building the hash table on the next cycle.

The bulk of the time is spent fetching, and the efficiency of the fetcher is due essentially to the tightly written `ocamlnet` library, which was designed for high performance from the ground up. We use asynchronous, non-blocking I/O throughout, with callbacks that mesh well with the functional paradigm. We keep a maximum of N (in the range 1000-2000) connections open. Just as the `WIRE` and `larbin` (Ailleret, 2003) crawlers, we use `GNU ADNS` (Jackson and Finch, 2006), an asynchronous-capable DNS client library to resolve IP address of unknown hosts. We keep every resolved IP cached, ignoring changes and TTL issues entirely. Asynchronous name resolution improves speed by a factor of 10. Since the fetcher runs in a single process (with OS-level callbacks), the downloaded HTML file is simply appended to the tail of a large batch. In case of errors (including the case when mime type is not text/html) the URL is placed on the forbidden list. Because charset-normalization is a step that cannot always be performed by standard libraries, we prefer to save out the charset information that is given in the http together with the original text and perform the conversion at a later stage. This facility would actually be a very useful addition in crawlers like `WIRE` or `larbin` which perform charset-normalization at download time, especially if the target is a less commonly taught language where the standard conversion libraries are not mature.

The parse step locates `<a href=` and pulls out the following quoted string, normalizing this using the base URL of the page. URLs containing angled brackets, question marks, or space/tab/newline are discarded. It is the responsibility of the management stage to detect duplicates, filter out the forbidden URLs and hosts, and to organize the next pass search in a manner that puts less load on smaller sites,

leveraging the built in ability of `ocamlnet` to serialize requests to a single host.

Altogether, the effort to tailor the crawl to the need of linguists pays off in notably improved throughput: instead of the 35 GB/day reported in Baroni and Kilgarriff (2006), `nut` has a sustained throughput of over 330 GB/day. This number is largely delimited by bandwidth availability at the Budapest Institute of Technology: `nut` is three times as fast (over 20 GB/hour) at night than during the day (8 GB/hour).

2. Luc

In search engine work the assumption that the fundamental unit of retrieval is the document (downloaded page) is rarely questioned. Yet in many classical IR/IE applications, books are broken up into chapters to be ranked (and returned) separately, and in question answering it is generally necessary to pinpoint information even more precisely, breaking documents down to the section, paragraph, or even sentence level. In many linguistic applications the objects of interest are the sentences, but for purposes of morphological analysis we are also interested in systems capable of responding to queries by single words or morphemes. For the smallest elements it is tempting to keep the entire dataset in main memory, but this would entail a drastic loss of efficiency for corpora that go beyond a single DVD: under more realistic query loads the system would page itself to death.

The `luc` IR subsystem of GLB stands neutral on the size or composition of the retrieved unit, but it assumes that in the typical (non-cached) case it will take at least one disk seek to get to it. At the 2GHz clock speeds and 10ms seek latencies typical of contemporary hardware, one can easily invert a 100x100 matrix the time it takes to fetch a single disk block. Thus the name of the game is to minimize the seeks, which means that all information about a retrieval unit that is relevant for speeding up queries must be precomputed and stored in an index kept in memory. `Luc` limits the size of the indexes to 4GB with the idea that at any given time two copies (a working copy and one under update/refresh) must stay in main memory. Since a billion retrieval units (see our goal 2) will require four-byte pointers (seek offsets), the 4GB limit on indexes is very tight, leaving no room for auxiliary indexes or meta-information stored with the offset. But if such information cannot be stored with the document pointer, how can it be accessed?

The key idea is to use the pointer itself, or more precisely, the location of the pointer in memory, to encode this information. We assume a small set of k dimensions, each dimension taking values in the [0,1] interval. Typical features that could be encoded in such numbers include the *page rank* of a document, the *authority* of the site it comes from, the *recency* of the document, its normalized *length*, and so on. In practice, none of these scales requires the granularity provided by 64-bit floats, and there are many quantization techniques we can use to arrive at a more compressed but still useful representation. Without loss of generality, we can assume that in any dimension values are limited to integers in the 0 to M_i range for $i = 1 \ldots k$.

There are important retrieval keys, such as the presence of a word w in a document, which require some encoding to fit into the `luc` model. We rank words by DF (and within a single DF, lexicographically) to arrive at a canonical ordering: in a typical gigaword corpus there will be on the order of a million different words. A single document will be indexed as many times as it has different words, so a gigaword corpus will require perhaps a hundred million pointers (but not more, since the per-document token multiplicities are collapsed).

The entire index is conceptualized as a single k-dimensional array with static bounds M_i. The main advantage of this view is that pointers to documents that should come early on the postings list are located close to the origin, and are accessible as $k-1$ dimensional slices of the original array. For example, if our query involves the terms *plane, of, immanence*, it is the last word which has the highest IDF, and query execution may begin by fetching the contents of the subarray that has the kth coordinate fixed at the value assigned to this word. Since the index array is very sparse, the key to fast execution is to compress it by kd-tree techniques.

In the `luc` model the impact of the different dimensions of classification on memory usage is similar to the impact that building a secondary array would have, but this fact is carefully hidden from the retrieval routines. For example, if we wish our posting lists to contain not just words, but POS-tagged words, the number of pointers per document grows (assuming that not every token of a type gets tagged the same way), and this impacts the size of the tree that supports the sparse array. Once the meta-information stored with a retrieval unit grows beyond 4 bytes, either index size cannot be kept at 4 GB or the number of retrieval units per node must be curtailed. Either way, the design aims squarely at what is likely to be the sweet spot in the memory price/performance curve for the next decade or so, with 8-16 GB DIMMs already reasonably cheap today and 64-128 GB machines likely to be commodity by 2020.

3. Conclusions

GLB is work in progress. `Nut`, the best developed component, is already in the performance tuning stage. It is currently capable of 50-200 URLs/sec, (20 MB/s download bandwidth, more than what our network can sustain), which we consider satisfactory for a single node, and large-grain parallelization in `had` style is not complicated. At the time of this writing `nut` still ignores `robots.txt`, but once this antisocial behavior is fixed it will be ready for release (planned by the time of the meeting) under LGPL.

`Luc` is in a more preliminary stage, especially as we strive to optimize query execution. The design described above is really optimized for the situations where the bulk of the subselection work is carried by the partial ordering that is encoded in any coordinate dimension. This works well for IDF, recency, and all other examples described in the main text, but falls short of the ideal of matching subtree-like patterns in syntactic descriptions (parse structures) that is explored in LSE. Realistically, we do not believe we can keep as much information as a parse tree in memory for each sentence and still maintain high performance characteristics, but this is largely a question of encoding parse information efficiently in an array-based system.

While our current goal is to first support regular expression queries composed of lexical entries and POS tags (i.e. the kind of queries familiar from IMS), and to respond to the more complex LSE-type queries based on a regexp 'stapler' (Bangalore, 1999), it is tempting to speculate how one would go about supporting complex syntactic queries from the get-go. The key issue is to encode syntactic relationships in their own dimensions: for example, in a system where "parse" means identifying the deep cases (Fillmore, 1968; Fillmore, 1977), a separate dimension would be required for each deep case, and even this would only help encoding main clause syntax. Encoding subordination and coordination would require further additions, and so would modifiers, possessives, and other issues considered critical in parsing. The effective balance between complicating the storage structure and query execution time needs to be tested carefully, and it may well turn out to be the case that stapling (which amounts to query-time discarding of false positives) is more effective than precomputing these relationships at load time.

Finally, `had` is still in the early design stage. Again, budget considerations are paramount: we expect neither thousands of highly capable processors nor exabyte storage to be available to GLB users. In fact, we expect no more than some form of shared disk space (e.g. NFS crossmounts or AFS). Tasks are expected to run on a single node for no more than a few hours. Each node will run a demon that can start a single task, and with the volume of task-related transactions staying well below a thousand per hour a single, central batch distributor is sufficient. We expect a rudimentary but usable system to be available together with the first release of `nut`.

Acknowledgments

We thank Dániel Varga (Budapest Institute of Technology) for performing measurements on other crawlers, and the anonymous referees for their penetrating remarks – responding to the issues they raised improved the draft significantly.

4. References

Sebastien Ailleret. 2003. Larbin: Multi-purpose web crawler.

Srinivas Bangalore. 1999. Explanation-based learning and finite state transducers: Application for parsing lexicalized tree-adjoining grammars. In Andras Kornai, editor, *Extended finite state models of language*, pages 160–192. Cambridge University Press.

Marco Baroni and Silvia Bernardini. 2004. Bootcat: Bootstrapping corpora and terms from the web. In *Proceedings of Language Resources and Evaluation Conference (LREC04)*, pages 1313–1316. European Language Resources Association.

Marco Baroni and Adam Kilgarriff. 2006. Large linguistically-processed Web corpora for multiple languages. In *Companion Volume to Proceedings of the European Association of Computational Linguistics*, pages 87–90, Trento.

Carlos Castillo. 2005. *Effective Web Crawling*. PhD Thesis, Department of Computer Science, University of Chile, Santiago.

Charles Fillmore. 1968. The case for case. In E. BachandR. Harms, editor, *Universals in linguistics theory*, pages 1–90. Holt and Rinehart, New York.

Charles Fillmore. 1977. The case for case reopened. In P. ColeandJ. M. Sadock, editor, *Syntax and Semantics 8: Grammatical relations*, pages 59–82. Academic Press, New York.

Rayid Ghani. 2001. Combining labeled and unlabeled data for text classification with a large number of categories. *ICDM, First IEEE International Conference on Data Mining (ICDM'01)*, 01:597–.

Péter Halácsy, Andrá1s Kornai, Péter Németh, and Dániel Varga. 2008. Parallel creation of gigaword corpora for medium density languages – an interim report. In *Proceedings of Language Resources and Evaluation Conference (LREC08)*, page to appear. European Language Resources Association.

Ian Jackson and Tony Finch. 2006. Gnu adns – advanced, easy to use, asynchronous-capable DNS client library and utilities.

Adam Kilgarriff. 2007. Googleology is bad science. *Computational Linguistics*, 33(1):147–151.

Philip Resnik and Aaron Elkiss. 2005. The linguist's search engine: an overview. In *ACL '05: Proceedings of the ACL 2005 on Interactive poster and demonstration sessions*, pages 33–36, Morristown, NJ, USA. Association for Computational Linguistics.