Indexing Google 1T for low-turnaround wildcarded frequency queries

Steinar Vittersø Kaldager
University of Oslo, Department of Informatics
steinavk@ifi.uio.no

Abstract

We propose a technique to prepare the Google 1T n-gram data set for wildcarded frequency queries with a very low turnaround time, making unbatched applications possible. Our method supports token-level wildcarding and – given a cache of 3.3 GB of RAM – requires only a single read of less than 4 KB from the disk to answer a query. We present an indexing structure, a way to generate it, and suggestions for how it can be tuned to particular applications.

1 Background and motivation

The "Google 1T" data set (LDC #2006T13) is a collection of 2-, 3-, 4-, and 5-gram frequencies extracted at Google from around 10^{12} tokens of raw web text. Wide access to web-scale data being a relative novelty, there has been considerable interest in the research community in how this resource can be put to use (Bansal and Klein, 2011; Hawker et al., 2007; Lin et al., 2010, among others).

We are concerned with facilitating approaches where a large number of frequency queries (optionally with token-by-token wildcarding) are made automatically in the context of a larger natural language-based system. Our motivating example is Bansal and Klein (2011) who substantially improve statistical parsing by integrating frequency-based features from Google 1T, taken as indicative of associations between words. In this work, however, parser test data is preprocessed "off-line" to make n-gram queries tractable, hampering the practical utility of this work. Our technique eliminates such barriers to application, making it feasible to answer previously unseen wildcarded frequency queries "on-line", i.e. when parsing new inputs. We devise a structure to achieve this, making each query approximately the cost of a single random disk access, using an in-memory cache of about 3 GB.

Our own implementation will be made available to other researchers as open source.

2 Prior work

Sekine and Dalwini (2010) have built a high-quality "1T search engine" that can return lists of n-gram/frequency pairs matching various types of patterns, but they operate on a wider scale of queries that makes their reported performance ($0.34\,s$ per query) insufficient for our desired use.

Hawker, Gardiner and Bennetts (2007) have explored the same problem and devised a "lossy compression" strategy, deriving from the data set a lookup table fitting in RAM indexed by hashes of entries, with cells corresponding to more than one entry in the n-gram set filled with a "compromise" value appropriate to the application. Although they obtain very fast queries, in our estimation the error introduced by this method would be problematic for our desired use. Furthermore, the authors do not address wildcarding for this strategy.

Talbot and Osborne (2007b; 2007a) have explored applications of Bloom filters to making comparatively small probabilistic models of large n-gram data sets. Though their method too is randomized and subject to false positives, they discuss ways of controlling the error rate.

Finally, several researchers including Bansal and Klein (2011) and Hawker, Gardiner and Bennetts (2007) describe ways of working "off-line", without low-turnaround querying. However, systems built along these lines will be unable to efficiently solve single small problems as they arise.

3 The indexing structure

The Google 1T data set consists of entries for n-grams for $n \in \{1, 2, 3, 4, 5\}$. We have not ap-

Proceedings of the NAACL HLT 2012 Student Research Workshop, pages 17–22,
Montréal, Canada, June 3-8, 2012. ©2012 Association for Computational Linguistics

plied our methods to the unigrams, as these are few enough in number that they can be held in RAM and structured by a standard method such as a hash table.

For the n-grams for $n \in \{2, 3, 4, 5\}$, we use separate carefully tuned and generated B-trees(Bayer and McCreight, 1972), caching nodes near the root in RAM and keeping the rest on disk.

3.1 Preprocessing

We apply *preprocessing* to the Google 1T data in two ways. Firstly, in almost any application of Google 1T it will be desirable to perform preprocessing to discard unimportant details, both in order to obtain a more manageable set of data and to make patterns evident that would otherwise be obscured by data scarcity. We identify and collapse to class tokens IP addresses, email addresses, prefixed hexadecimal numbers, and various kinds of URIs. We also collapse all decimal numeric data by mapping all *digits* to the digit zero.

The preprocessing we apply (which is used to generate the data set described in the rest of this article) reduces the vocabulary size by about 37.6%. It is our belief that, seen as a whole, this preprocessing is quite mild, considering the amount of almost universally unnecessary detail in the input data (e.g. 26% of the "words" begin with a digit).

Secondly, we use preprocessing in an entirely different way, as a brute-force approach to supporting wildcarded queries. The lookup structure constructed does *not* provide any wildcarding features – instead we use the preprocessing phase to add entries for each of the 2^n possible variously-wildcarded queries (all the possible configurations with each position either wildcarded or not) matching each of the n-grams in the data.

After this preprocessing, the wildcard token <*> can be treated just like any other token.

3.2 Dictionary

For cheaper processing and storage, our indexing structure deals in integers, not string tokens. The components of the structure describing this mapping are the *dictionaries*. These are associative arrays that are kept in RAM at runtime.

The main dictionary uniquely maps preprocessed tokens to integers (e.g., `<EMAIL>` $\longrightarrow$ 137). There are fewer than 2^{24} unique tokens in the 1T data set, so each integer requires only 3 bytes of storage.

During generation of the structure, we have found a second "transforming" dictionary useful. This dictionary maps *unpreprocessed* tokens to integers, e.g., `john@example.com` $\longrightarrow$ 137, avoiding string processing entirely. Unlike the normal dictionary, the transforming dictionary describes a many-to-one mapping.

The dictionaries are stored on disk simply as text files, with one line for each key/value pair. The appropriate dictionary is preloaded into an appropriate in-memory Judy array (Baskins, 2004) during initialization, taking up around 300 MB of memory.

The main and the transforming dictionaries have around 8 and 13 million entries respectively.

3.3 Search tree

Our central structure is a search tree, with the keys being fixed-length sequences of integer tokens.

Owing to the static nature of the data set, the tree can be constructed whole. For this reason there is no need to support insertions or deletions, and we do not account for them. Apart from the lack of support for mutation, the structure is a conventional B-tree (Bayer and McCreight, 1972). Our main contribution is identifying what sort of B-tree solves our problem, describing how it can be implemented effectively, and how it practically can be generated when dealing with a very large number of entries.

The tree should be broad to account for the difference in speed between searching within an in-memory node and retrieving the next node from disk. We use a branching factor limit of 127. With parameters like ours the tree will generally have a height (counting the root and the leaves, but not individual n-gram entries) of 5. It will be about half-filled, meaning – due to the generation method outlined in Subsection 4.3 – that the root will have around $\frac{127}{2}$ children. Figure 2 illustrates the pattern – rightmost nodes may have fewer children.

A larger node size for the leaves would mean lower memory requirements at the cost of having to make larger reads from the disk.

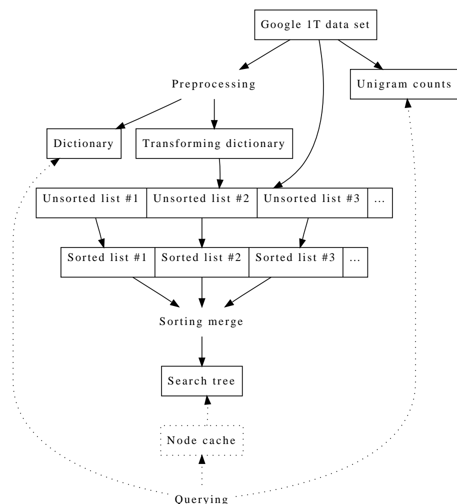

Figure 1: An overview of the steps involved in generating the indexing structure. The dotted portions indicate how it is later used.

4 Generating the structure

4.1 Creating the dictionaries

The dictionaries are created by simply iterating through the 1T vocabulary file, preprocessing and assigning integral labels.

During development we have performed it in Python and in Common Lisp, with the complexity of the preprocessing being on the order of 8 class-recognizer regular expressions and a character replacement pass for digits. One pass over the vocabulary with this setup takes around 18 minutes.

4.2 Creating sorted partial lists

We now seek to generate all the entries to be entered into our structure, ordered lexicographically by the numeric n-tuples that constitute the entry keys.

However, preprocessing portions of the (sorted) raw data set disturbs its ordering and introduces duplicate keys. After wildcarding it is also a concern that the list is very long – about $3.5 \cdot 10^{10}$ entries for the 5-grams after wildcarding and before merging.

As directly sorting a list of this size is impractical, we use an external sorting (Knuth, 1998) technique, dividing the input of length N into sections of K entries, then sort and merge duplicates in each one *separately*, producing $\lceil \frac{N}{K} \rceil$ separately sorted lists.

For sorting and merging, we use nested integer-based Judy arrays. For each batch of input we first fill such a structure – merging duplicate keys as they are encountered – and then traverse it in order, writing a sorted list.

We have found $1.5 \cdot 10^8$ to be a suitable value for K, using about 4.2 GB of memory per list-sorting process. In our development environment we use 10 such processes and produce 160 lists in 130 wall-clock minutes (1233 CPU minutes) for the full data set with full wildcarding.

4.3 Merging and creating the tree

The next step encompasses two subtasks – merging the sorted lists generated into one large sorted list (with duplicate entries merged), and using that large sorted list to generate an indexing tree.

The merging task involves, in our configuration, a P-way merge with $P \approx 160$. We perform this merge using a binary heap for replacement selection in $\log P$ time as outlined in Knuth (1998). Each node in the heap consists of a file handle, one "active" entry (which determines the value of the node), and a read buffer. After being popped from the heap, a node is reinserted if a next entry can be read from the read buffer or the file.

As they emerge in order from the merging section of the program, entries with duplicate keys are merged with their values added.

The tree-building routine receives all the entries to be stored correctly ordered and merged. It proceeds according to the following algorithm:

1. Space is left for the root node at the beginning of the file. We note the offset *after* this space as the "current generation offset". An empty "current node" is created.
2. For each input entry:
   (a) The entry is added as the last entry in the current node.
   (b) If the current node is now full, or if there are no more input entries, it is written to disk and then cleared.
3. We note down the current file offset (as used for writing) as the "next generation offset". We seek back to the current generation offset, and begin reading through the nodes until we reach the next generation offset, obtaining an in-order sequence of all nodes in the current generation

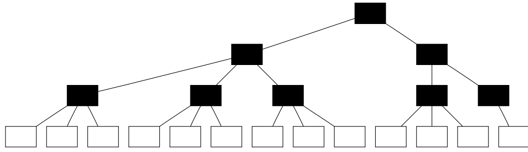

Figure 2: Illustration of the "all but leaves" caching strategy. Filled nodes are kept in memory, unfilled ones are left on disk. The maximal branching factor is 3 here (as compared to 127 in our trees).

(initially all leaf nodes). The sequence is read in lazily.

4. If this sequence is shorter than the number of entries in a node, it is used to construct the root node, which is then written to disk, and the process returns.
5. Otherwise, we repeat the process from the second step, with the following value replacements:
   - The next generation offset becomes the new current generation offset.
   - Each node read in from the file generates a new "input entry", with the key of the first entry of the node as the key, and the file offset pointing to the node as the value.

In our development environment this task currently takes around 283 minutes.

5 Using the indexing structure

5.1 Initialization and caching

The dictionary is loaded into a Judy array in RAM. The unigram counts are loaded into an integer array.

Finally, the upper levels of the trees are loaded into memory to be used as a cache. Since it is possible with only 3.3 GB of RAM, we recommend caching *all* nodes that are not leaves, as seen in Figure 2. Since we use broad trees, the number of leaves we can reach is relatively large compared to the number of internal nodes we need to cache.

5.2 Performing a query

The querying machinery assumes that queries are formulated in terms of integer tokens, and offers an interface to the dictionary so the caller can perform this transformation. This enables the caller to reuse integer mappings over multiple queries, and leaves the querying system loosely coupled to the application-specific preprocessing.

When a query arrives, all the tokens are first mapped to integers (using preprocessing and/or a dictionary). If this process fails for any token, the query returns early with frequency zero.

Otherwise, a conventional B-tree lookup is performed. This entails performing a binary search through the children of each node (with the value of each node considered as the value of its first entry, with entries in leaves identified by keys). In an internal node, after such a search, the node which has been located is loaded (from disk or memory cache) and the process repeats. In a leaf, it is checked whether the match found is *exact*, returning either its associated frequency value or 0 accordingly.

Empirically we have found usage of `lseek(2)` and `read(2)` to be the most performant way to perform the disk reads practically. For threaded applications `mmap(2)` may be more appropriate, as our method would require synchronization.

6 Performance

6.1 Testing setup

The development environment referred to elsewhere, `A`, is a high-performance computer with four eight-core 2.2GHz CPUs, 256 GB of RAM, and a number of 10 000 RPM hard disk drives. We also tested on `B` which is the same system augmented with 500 GB of solid state storage, and `C` which is an off-the-shelf PC with 8 GB of RAM, a 7200 RPM HDD and a single 2.93GHz CPU.

In development and preliminary testing, however, we discovered that the impact of disk caching made straightforward time measurements misleading. As seen in Figure 3, these measurements tended to be drastically affected by accumulation of large parts of the disk structure into cache, and as such showed ever-decreasing query times.

However, we have also observed that the required random disk access (a potentially "far" seek, followed by a read) dominates all other factors in the querying process in terms of cost. Our performance in terms of required random read accesses need not be measured: as noted in Subsection 5.1 we use a caching strategy which makes it self-evident that exactly one read access is required per query. Our

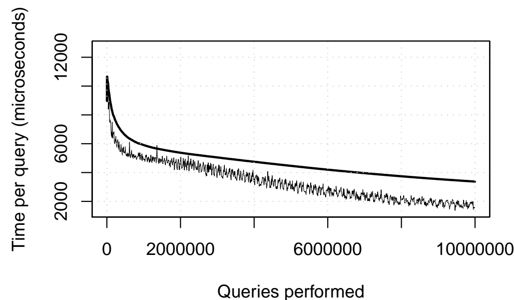

Figure 3: A test run of around 10 000 000 queries in A, illustrating how caching distorts timing information in lengthy tests. The wide line is cumulative average, the narrow one query-time average for the last 1 000 queries. The test run does not reach the stable state of a fully cached file.

performance testing, then, focuses on justifying our assertion that random disk access time is dominant. With this shown, we will have justified random-disk-access-count as a valid way to measure performance, and thus established our chief result.

We generated lists of test queries from the 1T data set with the same distribution as the preprocessed and merged entries in our structure.

6.2 Results

Table 1 shows measurements of time required for queries vs. time required to read a random leaf node (selected from a uniform distribution) without any querying logic. The random-read tests were *interleaved* with the querying tests, alternating batches of 100 queries with batches of 100 random reads. This process was chosen to avoid distorting factors such as differences in fragmentation and size of the area of the disk being read, memory available and used for caching it, as well as variable system load over time.

As can be seen in Table 1, the measurements indicate that time per random read times number of random reads is a very good approximation for time per query. The querying overhead seems to be on the order of 15μs, which is around 5% of the time per node access on the SSD, and less than 0.2% of the access time on the hard drives. It seems justified to measure the performance of our system by random disk access *count*.

	N	$\mu_{\mathtt{R}}$	$\mu_{\mathtt{Q}}$	$\sigma_{\lvert\mathtt{Q}-\mathtt{R}\rvert}$
A	10	6 089.41	6 069.55	260.05
A	100	6 135.54	6 149.60	640.05
A	1 000	6 094.83	6 097.82	477.35
B	10	299.50	313.59	11.09
B	100	298.43	317.14	15.02
B	1 000	308.39	326.00	9.62
C	10	14 763.60	14 924.81	818.90
C	100	14 763.11	14 769.24	634.99
C	1 000	14 776.43	14 708.51	817.47

Table 1: Performance measurements. N is test size, in batches of 100 queries and 100 random node-reads. All measurements in μs. $\mu_{\mathtt{Q}}$ is mean time to make a test query. $\mu_{\mathtt{R}}$ is mean time to read a random leaf node. $\sigma_{\lvert\mathtt{Q}-\mathtt{R}\rvert}$ is the sample standard deviation for the *difference* $Q_i - R_i$ between corresponding samples. (By definition, $\mu_{\lvert\mathtt{Q}-\mathtt{R}\rvert} = \mu_{\mathtt{Q}} - \mu_{\mathtt{R}}$.)

Tree breadth	127
Caching strategy	All but leaves
Total memory use	3.3 GB
Disk accesses per search	1
Leaf size	2 923 bytes
Generation time	431 minutes

Table 2: Vital numbers for our implementation. Generation time is based on adding up the measured wall-clock times reported elsewhere and is of course dependent on our development environment.

We have justified our central assertion that our indexing structure can answer queries using exactly one random disk access per query, as well as the underlying assumption that this is a meaningful way to measure its performance. The performance of our system on any particular hardware can then be estimated from the time the system uses for normal random disk accesses.

In terms of random reads per search, our result is clearly the best *worst-case* result achievable without loading the entire data set into memory: a single disk read (well below the size of a disk sector on a modern disk) per search. Naturally, further improvements could still be made in average-case performance, as well as in achieving equivalent results while using less resources.

The disk space required for the lookup structure

Wildcarding	2	3	4	5	Total
Full	3.6	23	80	221	327
None	3.4	15	24	24	65

Table 3: Disk space, in gigabytes, required for trees with and without wildcarding, by n and in total.

as a whole is summarized in Table 3. The full tree set with full wildcarding requires 327 GB. Wildcarding greatly affects the distribution of the entries: before wildcarding, the 4-grams are in fact more numerous than the 5-grams. Many real applications would not require full wildcarding capabilities.

7 Application adaptation and future work

Our method may be improved in several ways, leaving avenues open for future work.

Firstly and most importantly, it is natural to attempt applying our indexing structure to a real task. The work of Bansal and Klein (2011) has served as a motivating example. Implementing their method with "on-line" lookup would be a natural next step.

For other researchers who wish to use our indexing machinery, it has been made available as free software and may be retrieved at `http://github.com/svk/lib1tquery`.

If wildcarding is not required, a lowering of storage and memory requirements can be achieved by disabling it. This will reduce storage costs to about 21.52% or around 75 GB (and memory requirements approximately proportionally). Generalizing from this, if only *certain kinds* of wildcarded queries will be performed, similar benefits can be achieved by *certain kinds* of wildcarded (or even non-wildcarded) queries. For instance, less than 40% of the structure would suffice to perform the queries used by Bansal and Klein (2011).

Disk and memory efficiency could be improved by applying compression techniques to the nodes, though this is a balancing act as it would also increase computational load.

Furthermore, performance could be increased by using a layered approach that would be able to resolve *some* queries without accessing the disk at all. This is more feasible for an application where information is available about the approximate distribution of the coming queries.

References

Mohit Bansal and Dan Klein. 2011. Web-scale features for full-scale parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1*, HLT '11, pages 693–702, Stroudsburg, PA, USA. Association for Computational Linguistics.

Doug Baskins. 2004. Judy arrays. `http://judy.sourceforge.net/index.html`. (Online; accessed November 18, 2011).

R. Bayer and E. M. McCreight. 1972. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189. 10.1007/BF00288683.

Tobias Hawker, Mary Gardiner, and Andrew Bennetts. 2007. Practical queries of a massive n-gram database. In *Proceedings of the Australasian Language Technology Workshop 2007*, pages 40–48, Melbourne, Australia, December.

Donald E. Knuth. 1998. *The Art of Computer Programming, volume 3: Sorting and Searching*. Addison Wesley, second edition.

Dekang Lin, Kenneth Church, Heng Ji, Satoshi Sekine, David Yarowsky, Shane Bergsma, Kailash Patil, Emily Pitler, Rachel Lathbury, Vikram Rao, Kapil Dalwani, and Sushant Narsale. 2010. New tools for web-scale n-grams. In *Proceedings of the Seventh conference on International Language Resources and Evaluation (LREC'10)*, Valletta, Malta, may.

Satoshi Sekine and Kapil Dalwani. 2010. Ngram search engine with patterns combining token, POS, chunk and NE information. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, Valletta, Malta, may.

D. Talbot and M. Osborne. 2007a. Smoothed bloom filter language models: Tera-scale LMs on the cheap. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 468–476.

David Talbot and Miles Osborne. 2007b. Randomised language modelling for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 512–519, Prague, Czech Republic, June. Association for Computational Linguistics.