Fast Construction of a Word↔Number Index for Large Data

Miloš Jakubíček, Pavel Rychlý, and Pavel Šmerk

Natural Language Processing Centre
Faculty of Informatics, Masaryk University
Botanická 68a, 602 00 Brno, Czech Republic
{jak,pary,xsmerk}@fi.muni.cz

Abstract. The paper presents a work still in progress, but with promising results. We offer a new method of construction of word to number and number to word indices for very large corpus data (tens of billions of tokens), which is up to an order of magnitude faster than the current approach. We use HAT-trie for sorting the data and Daciuk's algorithm for building a minimal deterministic finite state automaton from sorted data. The latter we reimplemented and our new implementation is roughly three times faster and with smaller memory footprint than the one of Daciuk. This is useful not only for building word↔number indices, but also for many other applications, e.g. building data for morphological analysers.

Key words: word to number index, number to word index, finite state automata, hat-trie

1 Introduction

The main area of interest of this work lies in computer processing of large amounts of text (text corpora) with heavy annotation using a corpus management system that provides the user with fast and efficient search in the text data. The primary usage focuses on research in natural language processing, both from a more linguistically motivated or more language engineering oriented perspective, and on the exploitation of these tools in third-party industry applications in the domain of information systems and information extraction.

For any such system to perform well on large data, complex indexing and database management system must be in place – and so is this the case of the Manatee corpus management system which was the subject of our experiments.

Any reasonable indexing of text data by means of individual words (tokens in text) starts with providing a fast word-to-number and number-to-word mapping that allows to build the database indices on numbers, not words. This enables faster comparison, search and sort, and is also much more space efficient.

Aleš Horák, Pavel Rychlý (Eds.): Proceedings of Recent Advances in Slavonic Natural Language Processing, RASLAN 2013, pp. 63–67, 2013. © Tribun EU 2013

In this paper we particularly focus on constructing such word↔number mapping when indexing large text corpora. We first describe the current procedure used within the Manatee corpus management system and discuss its deficiencies when processing very large input data – here by large we refer to text collections containing billions of tokens. Then we present a new implementation exploiting a HAT-trie structure and provide an evaluation showing a significant speedup in building the mapping and henceforth also indexing of the whole text corpus.

2 Word↔number mapping in Manatee

2.1 Lexicon structure

The corpus management system Manatee uses the concept of a *lexicon* for providing the word↔number mapping, thus implementing two basic operations:

- `str2id` – retrieving an ID according to its word string
- `id2str` – retrieving a word according to its ID

The lexicon is constructed from source data when compiling all corpus indices and consists of three data files:

- `.lex` file – a plain text file containing the word strings separated by a NULL byte, in the order of their appearance in the source text.
- `.lex.idx` file – a fixed-size (4 B) integer index containing offsets to the `.lex` file. The `id2str` operation for a given ID n is implemented by retrieving the string offset at the $4 \cdot n^{\text{th}}$ byte in this file and reading at that offset in the `.lex` file (until the first NULL byte).
- `.lex.srt` file – a fixed-size (4 B) integer index containing IDs sorted alphabetically. The `str2id` operation for a given string s is implemented by binary search in this file (retrieving strings for comparison as described above).

2.2 Building the lexicon

When compiling corpus indices, new items are added to the lexicon in the order as they appear in the source texts and the lexicon is used for retrieving the ID of items already added to the lexicon. The system keeps two independent caches to speed up the process: one contains recently used lexicon items, another items that were recently added. As soon as the latter one reaches some threshold size, the cache is cleared – written to the lexicon and the lexicon must be re-sorted. This is a significant time bottleneck and as the lexicon grows, the time spent on its sorting grows rapidly too.

For more than two decades the data sizes of text corpora allowed not to care about the compilation time much, it was mainly the runtime of the database (i.e. querying) that mattered and that was subject to development. As data sizes of current text corpora grow to dozens of billions of tokens [1], the compilation time is being counted in days and starts to be an obstackle for data maintenance. Therefore we considered alternative implementations to overcome this issue.

3 Experiments and results

We demonstrate our results on three sets of corpus data. As can be seen in the Table 1, the sets differ not only in size: Tajik language uses Cyrillic, which means the words are two times longer (counted in bytes) only due to the encoding, and the French corpus from OPUS project[1] obviously uses rather limited vocabulary.

Table 1: Data sets used in the experiments.

data set	size	words	unique	size	language
100M	1148 MB	110 M	1660 k	31 MB	Tajik
1000M	5161 MB	957 M	1366 k	14 MB	French
10000M	69010 MB	12967 M	27892 k	384 MB	English

HAT-trie [2] is a cache-conscious data structure which combines trie and hash and allows sorted access. In general, for indexing the natural language strings, it is among the best solutions regarding both time and space. We used it[2] to create files described in the previous section. Results in the Table 2 show that hat-trie is up to an order of magnitude faster than the current solution `encodevert`.

Table 2: Comparison of encodevert and hat-trie.

	`encodevert`		**hat-trie**		
data set	**time**	**memory**	**time**	**memory**	**output size**
100M	3:11 m	0.44 GB	26.5 s	0.12 GB	44 MB
1000M	23:01 m	0.40 GB	2:21 m	0.04 GB	25 MB
10000M	7:38 h	0.98 GB	44:37 m	0.78 GB	607 MB

If a server is to support concurrent queries to multiple corpora, the indices for these corpora generated by `encodevert` (or now hat-trie) has to be loaded in memory. The last cell in the Table 2 indicates that for very large corpora it can consume a lot of memory, thus we tried to reduce this data. We used Jan Daciuk's `fsa` tools[3] which are able to convert a sorted set of strings to a deterministic acyclic finite state automaton usable for (static) minimal perfect hashing, i.e. string↔number translation, where the number is a rank in the sorted set of strings. We started with version 0.51 compiled with STOPBIT and NEXTBIT options, but because the original tools were rather memory and time consuming, we reimplement it and significantly reduce both time and space

[1] `http://opus.lingfil.uu.se/`, mostly legal texts.

[2] We use a free implementation from `https://github.com/dcjones/hat-trie`.

[3] `www.eti.pg.gda.pl/katedry/kiw/pracownicy/Jan.Daciuk/personal/fsa.html`

required for the automaton costruction (we did not change the output format). The Table 3 compares the results of the original version of `fsa_ubuild` acting on unsorted corpus data and our new approach. The last column shows new sizes of indices.

The last table, Table 4, compares only the original and reimplemented algorithm for sorted data. The hat-trie sort column are the costs of using hat-trie as a data pre-sort. Two results are obvious: firstly, having such an effective sort algorithm, to sort data and then use the algorithm for sorted data is always better than `fsa_ubuild`, secondly, to reduce the used memory it is better to flush sorted data to hard disk before fsa construction, as the time penalty is minimal.

Table 3: Building automata for perfect hashing from unsorted data.

	`fsa_ubuild`		**hat + new fsa**		
data set	**time**	**memory**	**time**	**memory**	**output size**
100M	*failed*		31.7 s	0.09 GB	15 MB
1000M	15:48 m	0.11 GB	2:34 m	0.06 GB	11 MB
10000M	7:44 h	31.01 GB	1:08 h	1.47 GB	363 MB

Table 4: Sorting data and building automata for perfect hashing from sorted data.

	hat-trie sort		`fsa_build`		**new fsa**	
data set	**time**	**memory**	**time**	**memory**	**time**	**memory**
100M	28.4 s	0.06 GB	12.4 s	0.21 GB	4.2 s	0.03 GB
1000M	2:51 m	0.04 GB	5.6 s	0.11 GB	1.8 s	0.03 GB
10000M	59:16 m	0.77 GB	35:15 m	27.07 GB	9:36 m	0.71 GB

4 Future work

The presented results are only preliminary, as it is only a proof of concept, not a final solution. We plan to further reduce both time and space of the automata construction, as well as their final size. The final automaton can be built directly from the input data which would cut the required memory to less than two thirds. The use of UTF-8 labels would reduce the space even further. We also want to employ some variable length encoding of numbers and addresses (similar to [3], but computationally simpler one). We suspect Daciuk's "tree index" used to discovering already known nodes during the automaton construction to be slow for large data and we hope that simple hash will decrease the compilation time significantly at the acceptable expense of some additional space.

Acknowledgements This work has been partly supported by the Ministry of Education of CR within the Lindat Clarin Center LM2010013.

References

1. Pomikálek, J., Rychlý, P., Jakubíček, M.: Building a 70 billion word corpus of English from ClueWeb. In: Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12). (2012) 502–506
2. Askitis, N., Sinha, R.: Hat-trie: a cache-conscious trie-based data structure for strings. In: Proceedings of the thirtieth Australasian conference on Computer science-Volume 62, Australian Computer Society, Inc. (2007) 97–105
3. Daciuk, J., Weiss, D.: Smaller representation of finite state automata. Theoretical Computer Science **450** (2012) 10–21