Tightly Packed Tries: How to Fit Large Models into Memory, and Make them Load Fast, Too

Ulrich Germann
University of Toronto and
National Research Council Canada
germann@cs.toronto.edu

Eric Joanis
National Research Council Canada
Eric.Joanis@cnrc-nrc.gc.ca

Samuel Larkin
National Research Council Canada
Samuel.Larkin@cnrc-nrc.gc.ca

Abstract

We present *Tightly Packed Tries* (TPTs), a compact implementation of read-only, compressed trie structures with fast on-demand paging and short load times.

We demonstrate the benefits of TPTs for storing n-gram back-off language models and phrase tables for statistical machine translation. Encoded as TPTs, these databases require less space than flat text file representations of the same data compressed with the *gzip* utility. At the same time, they can be mapped into memory quickly and be searched directly in time linear in the length of the key, without the need to decompress the entire file. The overhead for local decompression during search is marginal.

1 Introduction

The amount of data available for data-driven Natural Language Processing (NLP) continues to grow. For some languages, language models (LM) are now being trained on many billions of words, and parallel corpora available for building statistical machine translation (SMT) systems can run into tens of millions of sentence pairs. This wealth of data allows the construction of bigger, more comprehensive models, often without changes to the fundamental model design, for example by simply increasing the n-gram size in language modeling or the phrase length in phrase tables for SMT.

The large sizes of the resulting models pose an engineering challenge. They are often too large to fit entirely in main memory. What is the best way to organize these models so that we can swap information in and out of memory as needed, and as quickly as possible?

This paper presents *Tightly Packed Tries* (TPTs), a compact and fast-loading implementation of read-only trie structures for NLP databases that store information associated with token sequences, such as language models, n-gram count databases, and phrase tables for SMT.

In the following section, we first recapitulate some basic data structures and encoding techniques that are the foundations of TPTs. We then lay out the organization of TPTs. Section 3 discusses compression of node values (i.e., the information associated with each key). Related work is discussed in Section 4. In Section 5, we report empirical results from run-time tests of TPTs in comparison to other implementations. Section 6 concludes the paper.

2 Fundamental data structures and encoding techniques

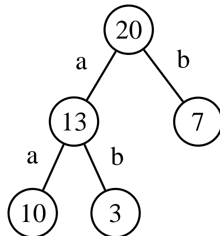
2.1 Tries

Tries (Fredkin, 1960), also known as *prefix trees*, are a well-established data structure for compactly storing sets of strings that have common prefixes. Each string is represented by a single node in a tree structure with labeled arcs so that the sequence of arc labels from the root node to the respective node "spells out" the token sequence in question. If we augment the trie nodes with additional information, tries can be used as indexing structures for databases that rely on token sequences as search keys. For the remainder of this paper, we will refer to such additional

Proceedings of the NAACL HLT Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing, pages 31–39,
Boulder, Colorado, June 2009. ©2009 Association for Computational Linguistics

total count	20
a	13
aa	10
ab	3
b	7

(a) Count table



(b) Trie representation

field	32-bit	64-bit
index entry: token ID	4	4
index entry: pointer	4	8
start of index (pointer)	4	8
overhead of index structure node value	x	y
total (in bytes)	$12 + x$	$20 + y$

(c) Memory footprint per node in an implementation using memory pointers

0	13	*offset of root node*
1	10	*node value of* 'aa'
2	0	*size of index to child nodes of* 'aa' in bytes
3	3	*node value of* 'ab'
4	0	*size of index to child nodes of* 'ab' in bytes
5	13	*node value of* 'a'
6	4	*size of index to child nodes of 'a' in bytes*
7	a	*index key for* 'aa' *coming from* 'a'
8	4	*relative offset of node* 'aa' $(5 - 4 = 1)$
9	b	*index key for* 'ab' *coming from* 'a'
10	2	*relative offset of node* 'ab' $(5 - 2 = 3)$
11	7	*node value of* 'b'
12	0	*size of index to child nodes of* 'b' in bytes
13	20	*root node value*
14	4	*size of index to child nodes of root in bytes*
15	a	*index key for* 'a' *coming from root*
16	8	*relative offset of node* 'a' $(13 - 8 = 5)$
17	b	*index key for* 'b' *coming from root*
18	2	*relative offset of node* 'b' $(13 - 2 = 11)$

(d) Trie representation in a contiguous byte array. In practice, each field may vary in length.

Figure 1: A count table (a) stored in a trie structure (b) and the trie's sequential representation in a file (d). As the size of the count table increases, the trie-based storage becomes more efficient, provided that the keys have common prefixes. (c) shows the memory footprint per trie node when the trie is implemented as a mutable structure using direct memory pointers.

information as the *node value*. Figure 1b shows a count table (Figure 1a) represented as a trie.

Tries offer two main advantages over other indexing structures, e.g., binary search trees. First, they are more compact because overlapping prefixes are stored only once. And second, unless the set of keys is extremely small, lookup is faster than with binary search trees. While the latter need time logarithmic in the number of keys, trie-based search is linear in the length of the search key.

2.2 Representing tries in memory

Mutable trie implementations usually represent tries as collections of fixed-size records containing the node value and a pointer or reference to an indexing structure (henceforth: *index*) that maps from arc or token labels to the respective child nodes. Links to child nodes are represented by object references or C-style memory pointers. To simplify the discussion, we assume in the following that the code consistently uses pointers. Since integers are faster to compare and require less space to store than character strings, token labels are best represented as integer IDs. With typical vocabulary sizes ranging from hundreds of thousands to several million distinct items, 32-bit integers are the data type of choice to store token IDs.[1]

This type of implementation offers flexibility and fast lookup but has two major drawbacks. First, load times are significant (cf. Tables 1 and 3). Since each node is created individually, the entire trie must be traversed at load time. In addition, all the information contained in the database must be copied explicitly from the OS-level file cache into the current process's memory.

Second, these implementations waste memory, especially on 64-bit machines. Depending on the architecture, memory pointers require 4 or 8 byes of memory. In theory, a 64-bit pointer allows us to address 16 exabytes (16 million terabytes) of memory. In practice, 20 to 30 bits per 64-bit pointer will remain unused on most state-of-the-art computing equipment.

The use of 32-bit integers to represent token IDs also wastes memory. Even for large corpora, the size

[1] 16 bits have been used occasionally in the past (Clarkson and Rosenfeld, 1997; Whittaker and Raj, 2001) but limit the vocabulary ca. 64 K tokens.

of the token vocabulary is on the order of several million distinct items or below. The Google 1T web n-gram database (Brants and Franz, 2006), for example, has a vocabulary of only ca. 13 million distinct items, which can be represented in 24 bits, letting 8 bits go to waste if IDs are represented as 32-bit integers.

An alternative is to represent the trie in a single contiguous byte array as shown in Figure 1d. For each node, we store the node value, the size of the index, and the actual index as a list of alternating token IDs and byte offsets. Byte offsets are computed as the distance (in bytes) between the first byte of the child node and the first byte of its parent. The trie is represented in post-order because this is the most efficient way to write it out to a file during construction. For each node, we need to store the byte offsets of its children. When we write tries to file in post-order, this information is available by the time we need it. The only exception is the root node, whose offset is stored at the beginning of the file in a fixed-length field and updated at the very end.

This representation scheme has two advantages. First, since node references are represented as relative offsets within the array, the entire structure can be loaded or mapped (cf. Section 2.5) into memory without an explicit traversal. And secondly, it allows symbol-level compression of the structure with local, on-the-fly decompression as needed.

2.3 Trie compression by variable-length coding

Variable-length coding is a common technique for lossless compression of information. It exploits the uneven distribution of token frequencies in the underlying data, using short codes for frequently occurring symbols and long codes for infrequent symbols. Natural language data with its Zipfian distribution of token frequencies lends itself very well to variable-length coding. Instead of using more elaborate schemes such as Huffman (1952) coding, we simply assign token IDs in decreasing order of frequency. Each integer value is encoded as a sequence of digits in base-128 representation. Since the possible values of each digit (0–127) fit into 7 bits, the eighth bit in each byte is available as a flag bit to indicate whether or not more digits need to be read. Given the address of the first byte of a compressed integer representation, we know when to stop reading subsequent bytes/digits by looking at the flag bit.[2]

TPTs use two variants of this variable-length integer encoding, with different interpretations of the flag bit. For "stand-alone" values (node values, if they are integers, and the size of the index), the flag bit is set to 1 on the last digit of each number, and to 0 otherwise. When compressing node indices (i.e., the lists of child nodes and the respective arc labels), we use the flag bit on each byte to indicate whether the byte belongs to a key (token ID) or to a value (byte offset).

2.4 Binary search in compressed indices

In binary search in a sorted list of key-value pairs, we recursively cut the search range in half by choosing the midpoint of the current range as the new lower or upper bound, depending on whether the key at that point is less or greater than the search key. The recursion terminates when the search key is found or it has been determined that it is not in the list.

With compressed indices, it is not possible to determine the midpoint of the list precisely, because of the variable-length encoding of keys and values. However, the alternation of flag bits between keys and values in the index encoding allows us to recognize each byte in the index as either a 'key byte' or a 'value byte'. During search, we jump *approximately* to the middle of the search range and then scan bytes backwards until we encounter the beginning of a key, which will either be the byte at the very start of the index range or a byte with the flag bit set to '1' immediately preceded by a byte with the flag bit set to '0'. We then read the respective key and compare it against the search key.

2.5 Memory mapping

Memory mapping is a technique to provide fast file access through the OS-level paging mechanism. Memory mapping establishes a direct mapping between a file on disk and a region of virtual memory,

[2] This is a common technique for compact representation of non-negative integers. In the Perl world it is know as BER (Binary Encoded Representation) compressed integer format (see the chapter `perlpacktut` in the Perl documentation). Apache Lucene and Hadoop, among many other software projects, also define variable-length encoded integer types.

often by providing direct access to the kernel's file cache. Transfer from disk to memory and vice versa is then handled by the virtual memory manager; the program itself can access the file as if it was memory. There are several libraries that provide memory mapping interfaces; we used the *Boost Iostreams* C++ library.[3] One nice side-effect of memory mapping the entire structure is that we can relegate the decision as to when to fall back on disk to the operating system, without having to design and code our own page management system. As long as RAM is available, the data will reside in the kernel's file cache; as memory gets sparse, the kernel will start dropping pages and re-loading them from disk as needed. In a computer network, we can furthermore rely on the file server's file cache in addition to the individual host's file cache to speed up access.

2.6 Additional tweaks

In order to keep the trie representation as small as possible, we shift key values in the indices two bits to the left and pad them with two binary flags. One indicates whether or not a node value is actually stored on the respective child node. If this flag is set to 0, the node is assumed to have an externally defined default value. This is particularly useful for storing sequence counts. Due to the Zipfian distribution of frequencies in natural language data, the lower the count, the more frequent it is. If we define the threshold for storing counts as the default value, we don't need to store that value for all the sequences that barely meet the threshold.

The second flag indicates whether the node is terminal or whether it has children. Terminal nodes have no index, so we don't need to store the index size of 0 on these nodes. In fact, if the value of terminal nodes can be represented as an integer, we can store the node's value directly in the index of its parent and set the flag accordingly.

At search time, these flags are interpreted and the value shifted back prior to comparison with the search key.

To speed up search at the top level, the index at the root of the trie is implemented as an array of file offsets and flags, providing constant time access to top-level trie nodes.

[3] Available at `http://www.boost.org`.

3 Encoding node values

Information associated with each token sequence is stored directly in a compact format "on the node" in the TPT representation. Special reader functions convert the packed node value into whatever structure best represents the node value in memory. In this section, we discuss the encoding of node values for various sequence-based NLP databases, namely sequence count tables, language models, and phrase tables for SMT.

3.1 Count tables

The representation of count tables is straightforward: we represent the count as a compressed integer. For representing sequence co-occurrence counts (e.g., bilingual phrase co-occurrences), we concatenate the two sequences with a special marker (an extra token) at the concatenation point.

3.2 Back-off language models

Back-off language models (Katz, 1987) of order n define the conditional probability $\mathrm{P}(w_i \,|\, w_{i-n+1}^{i-1})$ recursively as follows.

$$\mathrm{P}(w_i \,|\, w_{i-n+1}^{i-1}) = \begin{cases} \bar{\mathrm{P}}(w_i \,|\, w_{i-n+1}^{i-1}) & \text{if found} \\ \beta(w_{i-n+1}^{i-1}) \cdot \bar{\mathrm{P}}(w_i \,|\, w_{i-n+2}^{i-1}) & \text{otherwise} \end{cases} \tag{1}$$

Here, $\bar{\mathrm{P}}(w_i \,|\, w_{i-n+1}^{i-1})$ is a smoothed estimate of $\mathrm{P}(w_i \,|\, w_{i-n+1}^{i-1})$, $\beta(w_{i-n+1}^{i-1})$ is the *back-off weight* (a kind of normalization constant), and w_{i-n+1}^{i-1} is a compact notation for the sequence $w_{i-n+1}, \ldots, w_{i-1}$.

In order to retrieve the value $\bar{\mathrm{P}}(w_i \,|\, w_{i-n+1}^{i-1})$, we have to retrieve up to n values from the data base. In the worst case, the language model contains no probability values $\bar{\mathrm{P}}(w_i \,|\, \text{context})$ for any context but back-off weights for all possible contexts up to length $n-1$. Since the contexts $w_{i-n+1}^{i-1}, \ldots, w_{i-1}^{i-1}$ have common suffixes, it is more efficient to organize the trie as a backwards suffix tree (Bell *et al.*, 1990), that is, to represent the context sequences in right-to-left order in the trie. On each node in the trie, we store the back-off weight for the respective context, and the list of possible successor words and their conditional probabilities. The SRI language modeling toolkit (Stolcke, 2002) organizes its trie structure in the same way.

Probability values and back-off weights are stored via value IDs that are assigned in decreasing order of value frequency in the model and encoded as compressed integers. The list of successor words and their probability IDs is represented in the same way as the nodes' indices, i.e., as a sorted list of ⟨*word ID, probability value ID*⟩ pairs in compressed format.

3.3 Phrase tables for SMT

Phrase tables for phrase-based SMT list for every source phrase a number of target phrases and for each phrase pair a number of numerical scores that are usually combined in a linear or log-linear model during translation.

To achieve a very compact representation of target phrases, we organize all target phrases in the table in a "bottom-up" trie: instead of storing on each node a list of arcs leading to children, we store the node's label and its parent. Each phrase can thus be represented by a single integer that gives the location of the leaf node; we can restore the respective phrase by following the path from the leaf to the root.

Phrase pair scores are entropy-encoded and stored with variable-length encoding. Since we have several entropy-encoded values to store for each phrase pair, and several phrases for each source phrase, we can achieve greater compression with optimally sized "bit blocks" instead of the octets we have used so far. By way of a historical accident, we are currently still using indicator bits on each bit block to indicate whether additional blocks need to be read; a more principled approach would have been to switch to proper Huffman (1952) coding. The optimal sizes of the bit blocks are calculated separately for each translation table prior to encoding and stored in the code book that maps from score IDs to actual scores.

4 Related work

The challenges of managing huge models have been addressed by a number of researchers in recent years.

4.1 Array offsets instead of memory pointers

The CMU-Cambridge language modeling toolkit (Clarkson and Rosenfeld, 1997) represents the context trie in contiguous arrays of fixed-size node records, where each array corresponds to a certain "layer" of the trie. Instead of memory pointers, links between nodes are represented by offsets into the respective array. With some additional bookkeeping, the toolkit manages to store array offsets in only 16 bits (see Whittaker and Raj (2001) for details). Quantization of probability values and back-off weights is used to reduce the amount of memory needed to store probability values and back-off weights (see Section 4.4 below).

4.2 Model filtering

Many research systems offer the option to filter the models at load time or offline, so that only information pertaining to tokens that occur in a given input is kept in memory; all other database entries are skipped. Language model implementations that offer model filtering at load time include the SRILM toolkit (Stolcke, 2002) and the *Portage* LM implementation (Badr *et al.*, 2007). For translation tables, the *Moses* system (Koehn *et al.*, 2007) as well as *Portage* offer model filtering (*Moses:* offline; *Portage:* offline and/or at load time). Model filtering requires that the input is known when the respective program is started and therefore is not feasible for server implementations.

4.3 On-demand loading

A variant of model filtering that is also viable for server implementations is on-demand loading. In the context of SMT, Zens and Ney (2007) store the phrase table on disk, represented as a trie with relative offsets, so that sections of the trie can be loaded into memory without rebuilding them. During translation, only those sections of the trie that actually match the input are loaded into memory. They report that their approach is "not slower than the traditional approach", which has a significant load time overhead. They do not provide a comparison of pure processing speed ignoring the initial table load time overhead of the "traditional approach".

IRSTLM (Federico and Cettolo, 2007) offers the option to use a custom page manager that relegates part of the structure to disk via memory-mapped files. The difference with our use of memory mapping is that IRSTLM still builds the structure in memory and then swaps part of it out to disk.

4.4 Lossy compression and pruning

Large models can also be reduced in size by lossy compression. Both SRILM and IRSTLM offer tools for language model pruning (Stolcke, 1998): if probability values for long contexts can be approximated well by the back-off computation, the respective entries are dropped.

Another form of lossy compression is the quantization of probability values and back-off weights. Whittaker and Raj (2001) use pruning, quantization and difference encoding to store language model parameters in as little as 4 bits per value, reducing language model sizes by to 60% with "minimal loss in recognition performance." Federico and Bertoldi (2006) show that the performance of an SMT system does not suffer if LM parameters are quantized into 256 distinct classes (8 bits per value).

Johnson *et al.* (2007) use significance tests to eliminate poor candidates from phrase tables for SMT. They are able to eliminate 90% of the phrase table entries without an adverse effect on translation quality.

Pruning and lossy compression are orthogonal to the approach taken in TPTs. The two approaches can be combined to achieve even more compact language models and phrase tables.

4.5 Hash functions

An obvious alternative to the use of trie structures is the use of hash functions that map from n-grams to slots containing associated information. With hash-based implementations, the keys are usually not stored at all in the database; hash collisions and therefore lookup errors are the price to be paid for compact storage. This risk can be controlled by the design of the hash function. Talbot and Brants (2008) show that Bloomier filters (Chazelle *et al.*, 2004) can be used to create perfect hash functions for language models. This guarantees that there are no collisions between existing entries in the database but does not eliminate the risk of false positives for items that are not in the database.

For situations where space is at a premium and speed negotiable (e.g., in interactive context-based spelling correction, where the number of lookups is not in the range of thousands or millions per second), Church *et al.* (2007) present a compressed trigram model that combines Stolcke (1998) pruning with Golomb (1966) coding of inter-arrival times in the (sparse) range of hash values computed by the hash function. One major drawback of their method of storage is that search is linear in the total number of keys in the worst case (usually mediated by auxiliary data structures that cache information).

Since hash-based implementations of token sequence-based NLP databases usually don't store the search keys, it is not possible to iterate through such databases.

4.6 Distributed implementations

Brants *et al.* (2007) present an LM implementation that distributes very large language models over a network of language model servers. The delay due to network latency makes it inefficient to issue individual lookup requests to distributed language models. As Brants *et al.* point out: "Onboard memory is around 10,000 times faster" than access via the network. Instead, requests are batched and sent to the server in chunks of 1,000 or 10,000 requests.

5 Experiments

We present here the results of empirical evaluations of the effectiveness of TPTs for encoding n-gram language models and phrase tables for SMT. We have also used TPTs to encode n-gram count databases such as the Google 1T web n-gram database (Brants and Franz, 2006), but are not able to provide detailed results within the space limitations of this paper.[4]

5.1 Perplexity computation with 5-gram language models

We compared the performance of TPT-encoded language models against three other language model implementations: the SRI language modeling toolkit (Stolcke, 2002), IRSTLM (Federico and Cettolo, 2007), and the language model implementation currently used in the *Portage* SMT system (Badr *et al.*, 2007), which uses a pointer-based implementation but is able to perform fast LM filtering at load time. The task was to compute the perplexity of a text of

[4]Bottom line: the entire Google 1T web n-gram data base fits into about 16 GB (file/virtual memory), compared to 24 GB as *gzip*-compressed text files (file only).

Table 1: Memory use and runtimes of different LM implementations on a perplexity computation task.

		file/mem. size (GB)				1st run (times in sec.)					2nd run (times in sec.)				
		file	virt.	real	b/ng[1]	ttfr[2]	wall	usr	sys	cpu	ttfr	wall	usr	sys	cpu
full model loaded	**SRILM[3]**	5.2	16.3	15.3	42.2	940	1136	217	31	21%	846	1047	215	30	23%
	SRILM-C[4]	5.2	13.0	12.9	33.6	230	232	215	14	98%	227	229	213	14	98%
	IRST	5.1	5.5	5.4	14.2	614	615	545	13	90%	553	555	544	11	100%
	IRST-m[5]	5.1	5.5	1.6	14.2	548	744	545	8	74%	547	549	544	5	100%
	IRST-Q[6]	3.1	3.5	3.4	9.1	588	589	545	9	93%	551	553	544	8	100%
	IRST-Qm	3.1	3.5	1.4	9.1	548	674	546	7	81%	548	549	544	5	99%
	Portage	8.0	10.5	10.5	27.2	120	**122**	90	15	85%	110	112	90	14	92%
	TPT	**2.9**	**3.4**	**1.4**	**7.5**	**2**	127	**2**	2	**2%**	**1**	**2**	**1**	**1**	98%
filtered[7]	**SRILM**	5.2	6.0	5.9		111	112	90	12	91%	99	99	90	9	99%
	SRILM-C	5.2	4.6	4.5		112	113	93	11	91%	100	105	93	8	99%
	Portage	8.0	4.5	4.4		120	122	75	11	70%	80	81	74	7	99%

Notes: [1] **B**ytes per **n-g**ram (Amount of virtual memory used divided by total number of n-grams). [2] **T**ime **t**o **f**irst **r**esponse (first value returned). This was measured in a separate experiment, so the times reported sometimes do not match those in the other columns exactly. [3] Node indices stored in hashes. [4] "Compact" mode: node indices stored in sorted arrays instead of hashes. [5] Uses a custom paging mechanism to reduce memory requirements; [6] Values are quantized into 256 discrete classes, so that each value can be stored in 1 byte. [7] Models filtered on evaluation text at load time.

Table 2: Language model statistics.

	Gigaword	Hansard
unigrams	8,135,668	211,055
bigrams	47,159,160	4,045,363
trigrams	116,206,275	6,531,550
4-grams	123,297,762	9,776,573
5-grams	120,416,442	9,712,384
file size (ARPA format)	14.0 GB	1.1 GB
file size (ARPA .gz)	3.7 GB	225 MB

10,000 lines (275,000 tokens) with a 5-gram language model trained on the English Gigaword corpus (Graff, 2003). Some language model statistics are given in Table 2.

We measured memory use and total run time in two runs: the first run was with an empty OS-level file cache, forcing the system to read all data from the hard disk. The second run was immediately after the first run, utilizing whatever information was still cached by the operating system. All experiments were run successively on the same 64-bit machine with 16 GB of physical memory.[5] In order to eliminate distortions by variances in the network and file server load at the time the experiments were run, only locally mounted disks were used.

The results of the comparison are shown in Table 1. SRILM has two modi operandi: one uses hashes to access child nodes in the underlying trie implementation, the other one (SRILM-C) sorted arrays. The "faster" hash-based implementation pushes the architecture beyond its limitations: the system starts thrashing and is therefore the slowest by a wide margin.

The most significant bottleneck in the TPT implementation is disk access delay. Notice the huge difference in run-time between the first and the second run. In the first run, CPU utilization is merely 2%: the program is idle most of the time, waiting for the data from disk. In the second run, the file is still completely in the system's file cache and is available immediately. When processing large amounts of data in parallel on a cluster, caching on the cluster's file server will benefit all users of the respective model, once a particular page has been requested for the first time by any of them.

Another nice feature of the TPT implementation is the short delay between starting the program and being able to perform the first lookup: the first n-gram probability is available after only 2 seconds.

The slightly longer wall time of TPLMs ("tightly packed language models") in comparison to the *Portage* implementation is due to the way the data file is read: *Portage* reads it sequentially, while TPLMs request the pages in more or less random order, resulting in slightly less efficient disk access.

[5]Linux kernel version 2.6.18 (SUSE) on an Intel® Xeon® 2.33 GHz processor with 4 MB cache.

Table 3: Model load times and translation speed for batch translation with the *Portage* SMT system.

# of sentences per batch	Baseline			TPPT + Baseline LM			TPLM + Baseline PT			TPPT + TPLM		
	load time	w/s[1]	w/s[2]	load time	w/s[1]	w/s[2]	load time	w/s[1]	w/s[2]	load time[3]	w/s[1]	w/s[2]
47	210s	5.4	2.4	16s	5.0	4.6	178s	5.9	2.67	< 1s	5.5	5.5
10	187s	5.5	0.8	16s	5.1	3.6	170s	5.6	0.91	< 1s	5.6	5.6
1	—	—	—	15s	5.0	1.0	154s	5.5	0.12	< 1s	5.3	5.2

Baseline: *Portage*'s implementation as pointer structure with load-time filtering.
TP: Tightly packed; **PT:** phrase table; **LM:** language model
[1] words per second, excluding load time (pure translation time after model loading)
[2] words per second, including load time (bottom line translation speed)

5.2 TPTs in statistical machine translation

To test the usefulness of TPTs in a more realistic setting, we integrated them into the *Portage* SMT system (Sadat *et al.*, 2005) and ran large-scale translations in parallel batch processes on a cluster. Both language models and translation tables were encoded as TPTs and compared against the native *Portage* implementation. The system was trained on ca. 5.2 million parallel sentences from the Canadian Hansard (English: 101 million tokens; French: 113 million tokens). The language model statistics are given in Table 2; the phrase table contained about 60.6 million pairs of phrases up to length 8. The test corpus of 1134 sentences was translated from English into French in batches of 1, 10, and 47 or 48 sentences.[6]

Translation tables were not pre-filtered a priori to contain only entries matching the input. Pre-filtered tables are smaller and therefore faster to read, which is advantageous when the same text is translated repeatedly; the set-up we used more closely resembles a system in production that has to deal with unknown input. *Portage* does, however, filter models at load time to reduce memory use. The total (real) memory use for translations was between 1 and 1.2 GB, depending on the batch job, for all systems.

Table 3 shows the run-time test results. Ignoring model load times, the processing speed of the current *Portage* implementation and TPTs is comparable. However, when we take into account load times (which must be taken into account under realistic conditions), the advantages of the TPT implementation become evident.

[6]The peculiar number 47/48 is the result of using the default batch size used in minimum error rate training of the system in other experiments.

6 Conclusions

We have presented Tightly Packed Tries, a compact implementation of trie structures for NLP databases that provide a good balance between compactness and speed. They are only slightly (if at all) slower but require much less memory than pointer-based implementations. Extensive use of the memory-mapping mechanism provides very short load times and allows memory sharing between processes. Unlike solutions that are custom-tailored to specific models (e.g., trigram language models), TPTs provide a general strategy for encoding all types of NLP databases that rely on token sequences for indexing information. The novelty in our approach lies in the compression of the indexing structure itself, not just of the associated information. While the underlying mechanisms are well-known, we are not aware of any work so far that combines them to achieve fast-loading, compact and fast data structures for large-scale NLP applications.

References

Badr, G., E. Joanis, S. Larkin, and R. Kuhn. 2007. "Manageable phrase-based statistical machine translation models." *5th Intl. Conf. on Computer Recognition Systems (CORES)*. Wroclaw, Poland.

Bell, T. C., J. G. Cleary, and I. H. Witten. 1990. *Text Compression*. Prentice Hall.

Brants, T. and A. Franz. 2006. "Web 1T 5-gram Version 1." LDC Catalogue Number LDC2006T13.

Brants, T., A. C. Popat, P. Xu, F. J. Och, and J. Dean. 2007. "Large language models in machine trans-

lation." *EMNLP-CoNLL 2007*, 858–867. Prague, Czech Republic.

Chazelle, B., J. Kilian, R. Rubinfeld, and A. Tal. 2004. "The Bloomier filter: An efficient data structure for static support lookup tables." *15th Annual ACM-SIAM Symposium on Discrete Algorithms*. New Orleans, LA, USA.

Church, K., T. Hart, and J. Gao. 2007. "Compressing trigram language models with Golomb coding." *EMNLP-CoNLL 2007*, 199–207. Prague, Czech Republic.

Clarkson, P. R. and R. Rosenfeld. 1997. "Statistical language modeling using the CMU-Cambridge toolkit." *EUROSPEECH 1997*, 2707–2710. Rhodes, Greece.

Federico, M. and N. Bertoldi. 2006. "How many bits are needed to store probabilities for phrase-based translation?" *Workshop on Statistical Machine Translation*, 94–101. New York City.

Federico, M. and M. Cettolo. 2007. "Efficient handling of n-gram language models for statistical machine translation." *Second Workshop on Statistical Machine Translation*, 88–95. Prague, Czech Republic.

Fredkin, E. 1960. "Trie memory." *Communications of the ACM*, 3(9):490–499.

Golomb, S. W. 1966. "Run-length encodings." *IEEE Transactions on Information Theory*, 12(3):399–401.

Graff, D. 2003. "English Gigaword." LDC Catalogue Number LDC2003T05.

Huffman, D. A. 1952. "A method for the construction of minimum-redundancy codes." *Proceedings of the IRE*, 40(9):1098–1102. Reprinted in *Resonance* 11(2).

Johnson, H., J. Martin, G. Foster, and R. Kuhn. 2007. "Improving translation quality by discarding most of the phrasetable." *EMNLP-CoNLL 2007*, 967–975. Prague, Czech Republic.

Katz, S. M. 1987. "Estimation of probabilities from sparse data for the language model component of a speech recognizer." *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 35(3):400–401.

Koehn, P., H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens, C. Dyer, O. Bojar, A. Constantin, and E. Herbst. 2007. "Moses: Open source toolkit for statistical machine translation." *ACL 2007 Demonstration Session*. Prague, Czech Republic.

Sadat, F., H. Johnson, A. Agbago, G. Foster, R. Kuhn, J. Martin, and A. Tikuisis. 2005. "PORTAGE: A phrase-based machine translation system." *ACL Workshop on Building and Using Parallel Texts*, 133–136. Ann Arbor, MI, USA. Also available as NRC-IIT publication NRC-48525.

Stolcke, A. 1998. "Entropy-based pruning of backoff language models." *DARPA Broadcast News Transcription and Understanding Workshop*, 270–274. Lansdowne, VA, USA.

Stolcke, A. 2002. "SRILM — an extensible language modeling toolkit." *Intl. Conf. on Spoken Language Processing*. Denver, CO, USA.

Talbot, D. and T. Brants. 2008. "Randomized language models via perfect hash functions." *ACL 2008*, 505–513. Columbus, Ohio.

Whittaker, E. W. D. and B. Raj. 2001. "Quantization-based language model compression." *EUROSPEECH 2001*, 33–36. Aalborg, Denmark.

Zens, R. and H. Ney. 2007. "Efficient phrase-table representation for machine translation with applications to online MT and speech translation." *NAACL-HLT 2007 2007*, 492–499. Rochester, New York.