Efficient Data Structures for Massive N-Gram Datasets

Giulio Ermanno Pibiri
University of Pisa *and* ISTI-CNR
Pisa, Italy
giulio.pibiri@di.unipi.it

Rossano Venturini
University of Pisa *and* ISTI-CNR
Pisa, Italy
rossano.venturini@unipi.it

ABSTRACT

The efficient indexing of large and sparse N-gram datasets is crucial in several applications in Information Retrieval, Natural Language Processing and Machine Learning. Because of the stringent efficiency requirements, dealing with billions of N-grams poses the challenge of introducing a compressed representation that preserves the query processing speed.

In this paper we study the problem of reducing the space required by the representation of such datasets, maintaining the capability of looking up for a given N-gram within micro seconds. For this purpose we describe compressed, exact and lossless data structures that achieve, at the same time, high space reductions and no time degradation with respect to state-of-the-art software packages. In particular, we present a trie data structure in which each word following a context of fixed length k, i.e., its preceding k words, is encoded as an integer whose value is proportional to the number of words that follow such context. Since the number of words following a given context is typically very small in natural languages, we are able to lower the space of representation to compression levels that were never achieved before. Despite the significant savings in space, we show that our technique introduces a negligible penalty at query time.

CCS CONCEPTS

•**Information systems → Data compression; Language models; Information extraction;**

KEYWORDS

Performance, Data Compression, Elias-Fano, Language Models

1 INTRODUCTION

N-grams are widely adopted for many vital tasks in Information Retrieval, Natural Language Processing and Machine Learning, such as: auto-completion in search engines, spelling correction, similarity search, identification of duplicated documents in databases, automatic speech recognition and machine translation [10, 19, 21, 25, 26, 34]. As an example, query auto-completion is one of the key features that any modern search engine offers to help users formulate their queries. The objective is to predict the query by saving keystrokes: this is implemented by reporting the top-k most frequently-searched N-grams that follow the words typed by the user [2, 25, 26]. The identification of such patterns is possible by traversing a data structure that stores the N-grams as seen by previous user searches. Given the number of users served by large-scale search engines and the high query rates, it is of utmost importance that such data structure traversals are carried out in a handful of microseconds [2, 10, 21, 25, 26]. Another noticeable example is spelling correction in text editors and web search. In their basic formulation, N-gram spelling correction techniques work by looking up every N-gram in the input string in a pre-built data structure in order to assess their existence or return a statistic, e.g., a frequency count, to guide the correction [23]. If the N-gram is not found in the data structure it is marked as a misspelled pattern: in such case correction happens by suggesting the most frequent word that follows the pattern with the longest matching history [10, 21, 23].

At the core of all the mentioned applications lies an efficient data structure mapping N-grams to their associated satellite data, e.g., a frequency count representing the number of occurrences of the N-gram or probability and backoff weights for word-predicting computations [17, 31]. The data structure efficiency should be both in time and space, because modern string search and machine translation systems make very frequent queries over databases containing several billion N-grams that often do not fit in internal memory [10, 21].

While several solutions have been proposed for the indexing and retrieval of N-grams, either based on tries [15] or hashing [24], their practicality is actually limited because of some important inefficiencies that we discuss now. Context information, such as the fact that *relatively few* words may follow a given context, is not currently exploited to achieve better compression ratios. When query processing speed is the main concern, space efficiency is almost completely neglected by not compressing the data structure using sophisticated encoding techniques [17]. Space reductions are usually achieved by lossy quantization of satellite values and/or randomized approaches with false positive allowed [35]. The most space-efficient and lossless proposals still employ binary search over the compressed representation to lookup for a N-gram: this results in a severe inefficiency during query processing because of the lack of a compression strategy with a fast random access operation [31]. To support random access, current methods leverage on block-wise compression with expensive decompression of a block every time an element of the block has to be retrieved. Even hashing schemes, adopted for their constant-time access capabilities, result sub-optimal for static corpora as long as open addressing with linear probing is used [17, 31].

Because a solution that is compact, fast and lossless at the same time is still missing, the aim of this paper is that of addressing the

aforementioned problems by introducing compressed data structures that, despite their space efficiency, support efficient random access to the satellite N-gram values.

Our contributions. We list here our main contributions.

(1) We introduce a compressed trie data structure in which each level of the trie is modeled as a monotone integer sequence that we encode with *Elias-Fano* [12, 13] as to efficiently support random access operations and successor queries over the compressed sequence. Our hashing approach leverages on minimal perfect hash in order to use tables of size *equal* to the number of stored patterns per level, with one random access to retrieve the relative N-gram information.

(2) We describe a technique for lowering the space usage of the trie data structure, by reducing the magnitude of the integers that form its monotone sequences. Our technique is based on the observation that *few* distinct words follow a predefined context, in *any* natural language. In particular, each word following a context of fixed length k, i.e., its preceding k words, is encoded as an integer whose value is proportional to the number of words that follow such context.

(3) We present an extensive experimental analysis that shows our technique offers a significantly better compression with respect to the plain Elias-Fano trie, while only introducing a slight penalty at query processing time. Compared to the state-of-the-art proposals, our data structures outperform all of them for space usage, *without* compromising their time performance. More precisely, the most space-efficient proposals, which are both quantized and lossy, are no better than our trie data structure and up to 5 times slower. Conversely, we are as fast as the fastest competitor, but also retain an advantage of up to 65% in absolute space.

2 RELATED WORK

Two different data structures are mostly used to store large and sparse N-grams language models: trie [15] and hashing [24].

A trie is a tree devised for efficient indexing and search of string dictionaries. The common prefixes shared by the strings are represented once to achieve compact storage. This property makes this data structure useful for storing the N-gram strings in compressed space. In this case, each constituent word of a N-gram is associated a node in the trie and different N-grams correspond to different root-to-leaf paths. These paths must be traversed to resolve a query, which retrieves the string itself or an associated satellite value, e.g., a frequency count. Conceptually, a trie implementation has to store a triplet for any node: the associated word, satellite value and a pointer to each child node. As N is typically very small and each node has many children, tries are of short height and dense. Therefore, these are implemented as a collection of sorted arrays: for each level of the trie, a separate array is built to contain all the triplets for that level, sorted by the words. In this implementation, a pair of adjacent pointers indicates the sub-array listing all the children for a word, which can be inspected by binary search.

Hashing is another way to implement associative arrays: for each value of N a separate hash table stores all grams of order N. At the location indicated by the hash function the following information is stored: a fingerprint value to lower the probability of a false positive and the satellite data for the N-gram. This data structure permits to access the specified N-gram data in expected $O(1)$. Open addressing with linear probing is usually preferred over chaining for its better locality of accesses.

Tries are usually designed for space-efficiency as the formed sorted arrays are highly compressible. However, retrieval for the value of a N-gram involves $O(N)$ searches in the constituent arrays. Conversely, hashing is designed for speed but sacrifices space-efficiency since keys, along with their fingerprint values, are randomly distributed and, therefore, incompressible. Moreover, hashing is a randomized solution, i.e., there is a non-null probability of retrieving a frequency count for a N-gram *not* really belonging to the indexed corpus (false positive). Such probability equals $2^{-\delta}$, where δ indicates the number of bits dedicated to the fingerprint values: larger values of δ yield a smaller probability of false positive but also increase the space of the data structure. We now review how these two different data structural approaches have been used in the literature to implement the state-of-the-art solutions.

The paper by Pauls and Klein [31] proposes trie-based data structures in which the nodes are represented via sorted arrays or with hash tables with linear probing. The trie sorted arrays are compressed using a variable-length block encoding: a configurable radix $r = 2^k$ is chosen and the number of digits d to represents a number in base r is written in unary. The representation then terminates with the d digits, each of which requires exactly k bits. To preserve the property of looking up a record by binary search, each sorted array is divided into blocks of 128 bytes. The encoding is used to compress words, pointers and the positions that frequency counts take in a unique-value array that collect all distinct counts. The hash-based variant is likely to be faster than the sorted array variant, but requires extra table allocation space to avoid excessive collisions.

Heafield [17] improves the sorted array trie implementation with some optimizations. The keys in the arrays are replaced by their hashes and sorted, so that these are uniformly distributed over their ranges. Now lookup time for a record is reduced to $O(\log \log n)$ with high probability by using *interpolation* search [11]. Pointers are compressed using the integer compressor devised in [32]. Values can also be quantized using the *binning* method [14] that sorts the values, divides them into equally-sized bins and then elects the average value of the bin as the representative of the bin. The number of chosen quantization bits directly controls the number of created bins and, hence, the trade-off between space and accuracy.

Talbot and Osborne [35] use Bloom filters [4] with lossy quantization of frequency counts to achieve small memory footprint. In particular, the raw frequency count $f(g)$ of gram g is quantized using a logarithmic codebook, i.e., $\widetilde{f}(g) = 1 + \log_b f(g)$. The scale is determined by the base b of the logarithm: in the implementation b is set to $2^{1/v}$, where v is the quantization range used by the model, e.g., $v = 8$. Given the quantized count $\widetilde{f}(g)$ of gram g, a Bloom filter is trained by entering composite events into the filter, represented by g with an appended integer value j, which is incremented from 1 to $\widetilde{f}(g)$. Then at query time, to retrieve $\widetilde{f}(g)$, the filter is queried with a 1 appended to g. This event is hashed using the k hash functions of the filter: if all of them test positive, then the count is incremented and the process repeated. The procedure terminates as

soon as any of the k hash functions hits a 0 and the previous count is reported. This procedure avoids a space requirement for the counts proportional to the number of grams in the corpus because only the codebook needs to be stored.

The use of the succinct encoding LOUDS [20] is advocated in [37] to implicitly represent the trie nodes. In particular, the pointers for a trie of m nodes are encoded using a bitvector of $2m + 1$ bits. Bit-level searches on such bitvector allow forward/backward navigation of the trie structure. Words and frequency counts are compressed using Variable-Byte encoding [33], with an additional bitvector used to indicate the boundaries of such byte sequences as to support random access to each element.

Because of the importance of strings as one of the most common computerized kind of information, the problem of representing trie-based storage for string dictionaries is among one of the most studied in computer science, with many and different solutions available [9, 18, 29]. Given the properties that N-gram datasets exhibit, generic trie implementations are not suitable for their efficient treatment. However, comparing with the performance of such implementations gives useful insights about the performance gap with respect to a general solution. We mention Marisa [38] as the best and practical general-purpose trie implementation. The core idea is to use Patricia [28] tries to recursively represent the nodes of a Patricia trie. This clearly comes with a space/time trade off.

3 ELIAS-FANO TRIES

In this section we introduce our main result: a compressed trie data structure, based on the *Elias-Fano* representation [12, 13] of monotone integer sequences for its efficient random access and search operations. As we will see, the constant-time random access of Elias-Fano makes it the right choice for the encoding of the sorted-array trie levels, given that we fundamentally need to randomly access the sub-array pointed to by a pair of pointers. Such pair is retrieved in constant time too. Now every access performed by binary search takes $O(1)$ *without* requiring any block decompression, differently from currently employed strategies [31].

We also describe a novel technique to lower the memory footprint of the trie levels by losslessly reducing the entity of their constituent integers. This reduction is achieved by mapping a word ID *conditionally* to its context of fixed length k, i.e., its k preceding words.

3.1 Core data structure

As it is standard, a unique integer ID is assigned to each distinct token (uni-gram) to form the vocabulary V of the indexed corpus. Uni-grams are indexed using a hash data structure that stores for each gram its ID in order to retrieve it when needed in $O(1)$. If we sort the N-grams following the token-ID order, we have that all the successors of gram $w_1^{N-1} = w_1, \ldots, w_{N-1}$, i.e., all grams whose prefix is w_1^{N-1}, form a strictly increasing integer sequence. For example, suppose we have the uni-grams $\langle a, b, c, d \rangle$, which are assigned IDs $\langle 0, 1, 2, 3 \rangle$ respectively. Now consider the bi-grams $\langle aa, ac, bb, bc, bd, ca, cd, db, dd \rangle$ sorted by IDs. The sequence of the successors of a, referred to as the *range* of a, is $\langle a, c \rangle$, i.e., $\langle 0, 2 \rangle$; the sequence of the successors of b, is $\langle b, c, d \rangle$, i.e., $\langle 1, 2, 3 \rangle$ and

so on. Concatenating the ranges, we obtain the integer sequence $\langle 0, 2, 1, 2, 3, 0, 3, 1, 3 \rangle$. In order to distinguish the successors of a gram from others, we also maintain where each range begins in a monotone integer sequence of pointers. In our example, the sequence of pointers is $\langle 0, 2, 5, 7, 9 \rangle$ (we also store a final dummy pointer to be able to obtain the last range length by taking the difference between the last and previous pointer). The ID assigned to a uni-gram is also used as the position at which we read the uni-gram pointer in the uni-grams pointer sequence. Therefore, apart from uni-grams, each level of the trie has to store two integer sequences: one for the representation of the gram-IDs, the other for the pointers.

Among the many integer compressors available in the literature [33], we choose Elias-Fano, which has been recently applied to inverted index compression showing an excellent time/space trade off [30, 36]. We now briefly describe this elegant integer encoding.

Elias-Fano. Given a monotonically increasing sequence $S(n, u)$ of n positive integers upper-bounded by u, i.e., $S[i-1] \leq S[i]$, for any $1 \leq i < n$ with $S[n-1] \leq u$, we write each $S[i]$ in binary using $\lceil \log u \rceil$ bits. Each binary representation is then split into two parts: a *high* part consisting in the first $\lceil \log n \rceil$ most significant bits that we call *high bits* and a *low* part consisting in the other $\ell = \lfloor \log \frac{u}{n} \rfloor$ bits that we similarly call *low bits*. Let us call h_i and ℓ_i the values of high and low bits of $S[i]$ respectively. The Elias-Fano representation of S is given by the encoding of the high and low parts. The array $L = [\ell_0, \ldots, \ell_{n-1}]$ is stored in fixed-width and represents the encoding of the low parts. Concerning the high bits, we represent them in *negated unary* using a bit vector of $n + u/2^\ell \leq 2n$ bits as follows. We start from a 0-valued bit vector H and we set the bit in position $h_i + i$, $\forall i \in [0, n)$. The effect is that now the k-th unary integer m of H indicates that m integers of S have high bits equal to k. Finally the Elias-Fano representation of S is given by the concatenation of H and L and takes, overall, $\text{EF}(S(n, u)) = n \lceil \log \frac{u}{n} \rceil + 2n$ bits.

Despite its simplicity, it is possible to randomly access an integer from a sequence compressed with Elias-Fano *without* decompressing it. The operation is supported using an auxiliary data structure that is built on bit vector H, able to efficiently answer $\text{Select}_1(i)$ queries, that return the position in H of the i-th 1 bit. This auxiliary data structure is *succinct* in the sense that it is negligibly small compared to $\text{EF}(S(n, u))$, requiring only $o(n)$ additional bits [8, 36]. Using the Select_1 primitive, it is possible to implement $\text{Access}(i)$, which returns $S[i]$ for any $1 \leq i < n$, in constant time.

Gram-ID sequences and pointers. While the sequences of pointers are monotonically increasing by construction and, therefore, immediately Elias-Fano encodable, the gram-ID sequences could not. However, a gram-ID sequence can be transformed into a monotone one, though not strictly increasing, by taking range-wise prefix sums: to the values of a range we sum the last prefix sum (initially equal to 0). Then, our exemplar sequence becomes $\langle 0, 2, 3, 4, 5, 5, 8, 9, 11 \rangle$. The last prefix sum is initially 0, therefore the range of a remains the same, i.e., $\langle 0, 2 \rangle$. Now the last prefix sum is 2, so we sum 2 to the values in the range of b, yielding $\langle 3, 4, 5 \rangle$, and so on. In particular, if we sort the vocabulary IDs in decreasing order of occurrence, we make small IDs appear more often than

large ones and this is highly beneficial for the growth of the universe u and, hence, for Elias-Fano whose space occupancy critically depends on it. We emphasize this point again: for each uni-gram in the vocabulary we count the number of times it appears in all gram-ID sequences. Notice that the number of occurrences of a N-gram can be different than its frequency count as reported in the indexed corpus. The reason is that such corpora often do not include the N-grams appearing less than a predefined frequency threshold.

Frequency counts. To represent the frequency counts, we use the unique-value array technique, i.e., each count is represented by its *rank* in an array, one for each separate value of N, that collects all distinct frequency counts. The reason for this is that the distribution of the frequency counts is extremely skewed (see Table 2), i.e., relatively few N-grams are very frequent while most of them appear only a few times. Now each level of the trie, besides the sequences of gram-IDs and pointers, has also to store the sequence made by all the frequency count ranks. Unfortunately, this sequence of ranks is not monotone, yet it follows the aforementioned highly repetitive distribution. Therefore, we assigned to each count rank a codeword of variable length. As similarly done for the gram-IDs, by assigning smaller codewords to more repetitive count ranks, we have most ranks encoded with just a few bits. More specifically, starting from $k = 1$, we first assign all the 2^k codewords of length k before increasing k by 1 and repeating the process until all count ranks have been considered. Therefore, we first assign codewords 0 and 1, then codewords 00, 01, and so on. All codewords are then concatenated one after the other in a bitvector B. Following [16], to the i-th value we give codeword $c = i + 2 - 2^{\ell_c}$, where $\ell_c = \lfloor \log(i + 2) \rfloor$ is the number of bits dedicated to the codeword. From codeword c and its length ℓ_c in bits, we can retrieve i by taking the inverse of the previous formula, i.e., $i = c - 2 + 2^{\ell_c}$. Besides the bitvector for the codewords themselves, we also need to know where each codeword begins and ends. We can use another bitvector for this purpose, say L, that stores a 1 for the starting position of every codeword. A small additional data structure built on L allows efficient computation of Select_1, which we use to retrieve ℓ_c. In fact, $b = \text{Select}_1(i)$ gives us the starting position of the i-th codeword. Its length is easily computed by scanning L upward from position b until we hit the next 1, say in position e. Finally $\ell_c = e - b$ and $c = B[b, e - 1]$.

In conclusion, each level i of the trie stores three sequences: the gram-ID sequence G_i, the pointer sequence P_i and the count ranks sequence R_i. Two exceptions are represented by uni-grams and maximum-order grams, for which gram-ID and pointer sequences are missing respectively.

Lookup. We now describe how to retrieve the frequency count given a gram w_1^m. We first perform m vocabulary lookups to map the gram tokens into its constituent IDs. We write these IDs into an array $W[1, m]$. This preliminary query-mapping step takes $O(m)$. Now, the search procedure basically has to locate $W[i]$ in the i-th level of the trie. Refer to Figure 1, which shows a pictorial representation of a trie of order 3. If $m = 1$, then our search terminates: at the position $k_1 = W[1]$ we read the rank $r_1 = R_1[k_1]$ to finally access $C_1[r_1]$. If, instead, m is greater than 1, the position
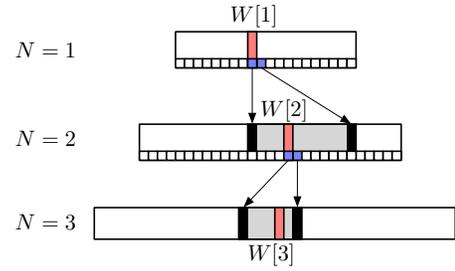

Figure 1: Search for the tri-gram $W[1, 3]$ in a trie data structure of order 3. The light gray shaded area indicates the range of IDs that follow a given gram and its is delimited by the black IDs pointed to by the pointers represented as blue squares.

k_1 is used to retrieve the pair of pointers $\langle P_1[k_1], P_1[k_1 + 1] \rangle$ in constant time, which delimits the range of IDs in which we have to search for $W[2]$ in the second level of the trie. This range is inspected by binary search, taking $O(\log(P_1[k_1 + 1] - P_1[k_1]))$ as each access to an Elias-Fano-encoded sequence is performed in constant time. Let k_2 be the position at which $W[2]$ is found in the range. Again, if $m = 2$, the search terminates by accessing $C_2[r_2]$ where r_2 is the rank $R_2[k_2]$. If m is greater than 2, we fetch the pair $\langle P_2[k_2], P_2[k_2 + 1] \rangle$ to continue the search of $W[3]$ in the third level of the trie, and so on. This search step is repeated for $m - 1$ times in total, to finally return the count $C_m[r_m]$ of w_1^m.

3.2 Context-based identifier remapping

In this subsection we describe a technique that lowers the space occupancy of the gram-ID sequences that constitute, as we have seen, the core of the trie data structure. The high level idea is to map a word w occurring after context w_1^k to an integer whose value is proportional to the number of words that *follow* the context w_1^k, and *not* proportional to the whole vocabulary size $|V|$. Specifically, w is mapped to the position it occupies within its siblings, i.e., all the words following the gram w_1^k. We call this technique *context-based remapping* as each ID is mapped to the position it takes relative to a context. As we will see in the experimental Section 5, the dataset vocabulary can contain several million tokens, whereas the number of words that naturally occur after another is typically very small. Even in the case of stopwords, such as "the" or "are", the number of words that can follow is far less than the whole number of distinct words for any N-gram dataset. This ultimately means that the remapped integers forming the gram-ID sequences of the trie will be much smaller than the original ones, which can indeed range from 0 to $|V| - 1$. Lowering the values of the integers clearly helps in reducing the memory footprint of the levels of the trie because *any* integer compressor takes advantage of encoding smaller integers, since fewer bits are needed for their representation [27, 30]. In our case the gram-ID sequences are encoded with Elias-Fano: from Section 3.1 we know that Elias-Fano spends $\lceil \log \frac{u}{n} \rceil + 2$ bits per integer, thus a number of bits proportional to the average gap between the integers. Remapping the integers as explained can *critically* reduce the universe u of representation, thus lowering the average gap of the sequence.
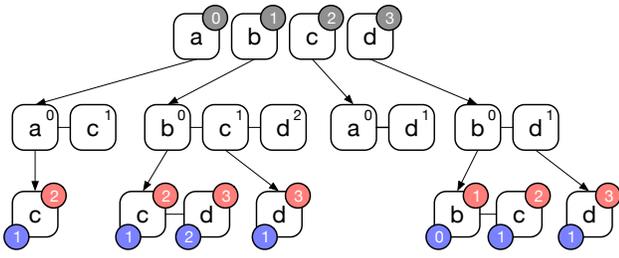
Figure 2: Example of a trie of order 3. **Vocabulary IDs are represented in dark grey while level-**3 **IDs in red. The blue IDs are derived by applying a context-based remapping with context length** 1.

To better understand the power of the idea, we consider now a concrete example. We continue the example from Section 3.1, extended to contain the tri-grams depicted in Figure 2. The Figure shows a trie model of order 3. The gray IDs are the vocabulary IDs and the red ones are the last token IDs of the tri-grams as assigned by the vocabulary. We now explain how the remapped IDs, represented in blue, are derived by the model using our technique with a context of length 1. Consider the tri-gram bcd. The default ID of d is 3. We now rewrite this ID as the position that d takes within the successors of the word preceding it, i.e., c (context 1). As we can see in the Figure, d appears in position 1 within the successors of c, therefore its new ID will be 1. Another example: take dbb. The default ID of b is 1, but it occurs in position 0 within the successors of its parent b, therefore its new ID is 0. As it is clear, the first two levels of the trie act as a *mapper* structure which is used to map the vocabulary ID of the last token of a tri-gram into its new ID. At query time, this mapper structure is used to retrieve the mapped ID that is finally searched in the third level of the trie. Therefore, notice that the technique takes full advantage of the N-gram model represented by the trie structure itself in that it does *not* need any redundancy to retrieve the remapped ID.

We observe that this strategy allows a great deal of flexibility, in that we can choose the length of the context. The example in Figure 2 illustrates how to map tri-grams using a context of length 1: this is clearly the only one possible as the first two levels of the trie must be used to retrieve the mapped ID at query time. However, if we have a gram of order 4, i.e., w_1^4, we can choose to map w_4 as the position it takes within the successors of w_3 (context length 1) or within the successors of $w_2 w_3$ (context length 2). In general, with a gram of order $N \geq 2$ we can choose between $N - 2$ distinct context lengths k, i.e., $1 \leq k \leq N - 2$. This also means that, when using context length k, we have the first $k + 1$ levels of the trie acting as the mapper structure and, hence, must be kept un-mapped. Clearly, the greater the context length we use, the smaller the remapped integers will be. The proper context length to use should take into account the characteristics of the N-gram dataset, in particular the number of grams per order.

To support our discussion, we present some real numbers regarding the effect of the context-based remapping on the average gap of the gram-ID sequences of Europarl, one of the datasets used in the experiments. As uni-grams and bi-grams must be kept un-mapped

as explained, we report the statistics for the trie of order 5 built from the N-grams of the dataset. The average gap $\frac{u}{n}$ of the sequences of order $N \geq 3$ are respectively, without remapping: 2404, 2782 and 2920. With our technique, using a context of length 1, we obtain the following average gaps instead: 213, 480 and 646, i.e., an average reduction of 7.2 (up to 11.3 on tri-grams). With a context of length 2 we obtain: 48 and 101 for 4− and 5-grams respectively, i.e., an average reduction of 43.4 (up to 58 on 4-grams).

Lookup. The context-based remapping comes with an overhead at query time as the search algorithm described in Section 3.1 must map the default vocabulary ID to its remapped ID, before it can be searched in the proper gram sequence. If the remapping strategy is applied with a context of length k, it involves $k(N-k-1)$ additional searches. As an example, by looking at Figure 2, before searching the mapped ID 1 of d for the tri-gram bcd, we have to map the vocabulary ID of d, i.e., 3, to 1. For this task, we search 3 within the successors of c. As 3 is found in position 1, we now know that we have to search for 1 within the successors of bc. On the one hand, the context-based remapping will assign smaller IDs as the length of the context rises, on the other hand it will also spend more time at query processing. In conclusion, we have a space/time trade-off that we explore with an extensive experimental analysis in Section 5.

4 HASHING

Since the indexed N-gram corpus is static, we obtain a *full* hash utilization by resorting to Minimal Perfect Hash (MPH). We indexed all grams of the same order N into a separate MPH table T_N, each with its own MPH function h_N. This introduces a twofold advantage over the linear probing approach used in the literature [17, 31]: use a hash table of size *equal* to the exact number of grams per order (no extra space allocation is required) and avoid the linear probing search phase by requiring one single access to the required hash location. We use the implementation, publicly available at https://github.com/ot/emphf, of MPH as described in [3], which requires 2.61 bits per key on average. At the hash location for a N-gram we store: its 8-byte hash key as to have a false positive probability of 2^{-64} (4-byte hash keys are supported as well) and the position of the frequency count in the unique-value array C_N which keeps all distinct frequency counts for order N. As already motivated, these unique-value arrays, one for each different order of N, are negligibly small compared to the number of grams themselves and act as a direct map from the position of the count to its value. Although these unique values could be sorted and compressed, we do not perform any space optimization as these are too few to yield any improvement but we store them uncompressed and byte-aligned, in order to favor lookup time. We also use this hash approach to implement the vocabulary of the previously introduced trie data structure.

Lookup. Given N-gram g we compute the position $p = h_N(g)$ in the relevant table T_N, then we access the count rank r stored at position p and finally retrieve the count value $C_N[r]$.

		Europarl		YahooV2		GoogleV2	
		bpg	μs × query	bpg	μs × query	bpg	μs × query
	EF	1.97	1.28	2.17	1.60	2.13	2.09
	PEF	1.87 (−4.99%)	1.35 (+5.93%)	1.91 (−12.03%)	1.73 (+8.00%)	1.52 (−28.60%)	1.91 (−8.79%)
CONTEXT-BASED ID REMAPPING — $k=1$	EF	1.67 (−15.30%)	1.58 (+23.86%)	1.89 (−12.92%)	2.05 (+28.07%)	1.91 (−10.24%)	3.03 (+44.61%)
	PEF	1.53 (−22.36%)	1.61 (+25.89%)	1.63 (−24.91%)	2.16 (+35.22%)	1.31 (−38.71%)	2.30 (+9.88%)
$k=2$	EF	1.46 (−25.62%)	1.60 (+25.17%)	1.68 (−22.32%)	2.08 (+30.23%)	—	—
	PEF	1.28 (−34.87%)	1.64 (+28.12%)	1.38 (−36.15%)	2.15 (+34.81%)	—	—

Table 1: Average bytes per gram (bpg) and average Lookup time per query in micro seconds.

5 EXPERIMENTS

We performed our experiments on the following standard datasets: Europarl, which consists in all unpruned N-grams extracted from the english Europarl parallel corpus [22]; YahooV2 [1] and GoogleV2 [5] as the last English version of Web1T [5]. Each dataset comprises all N-grams for $1 \le N \le 5$ and associated frequency counts. Table 2 shows the basic statistics of the datasets.

N	Europarl		YahooV2		GoogleV2	
	n	m	n	m	n	m
1	304 579	4518	3 475 482	23 785	24 357 349	246 490
2	5 192 260	4663	53 844 927	31 711	665 752 080	722 966
3	18 908 249	2975	187 639 522	19 856	7 384 478 110	683 653
4	33 862 651	1744	287 562 409	10 761	1 642 783 634	133 491
5	43 160 518	1032	295 701 337	6167	1 413 870 914	104 025
Total	101 428 257	7147	828 223 677	45 285	11 131 242 087	1 073 473
gzip bpg	6.98		6.45		6.20	

Table 2: Number of N-grams per order (n); distinct frequency counts (m); and gzip average bytes per gram (bpg) for the datasets used in the experiments.

Compared Indexes. We compare the performance of our data structures against the following software packages that use the approaches introduced in Section 2.

- BerkeleyLM implements two trie data structures based on sorted arrays and hash tables to represent the nodes of the trie [31]. Java implementation available at: https://github.com/adampauls/berkeleylm.
- Expgram makes use of the LOUDS succinct encoding [20] to implicitly represent the trie structure and compresses frequency counts using Variable-Byte encoding [37]. C++ implementation available at: https://github.com/tarowatanabe/expgram.
- KenLM implements a trie with interpolation search and a hashing with linear probing [17]. C++ implementation available at: http://kheafield.com/code/kenlm.
- Marisa is a general-purposes string dictionary implementation in which Patricia tries are recursively used to represent the nodes of a Patricia trie [38]. C++ implementation available at: https://github.com/s-yata/marisa-trie.
- RandLM employs Bloom filters with lossy quantization of frequency counts to attain to low memory footprint [35]. C++ implementation available at: https://sourceforge.net/projects/randlm.

Experimental setup. All experiments have been performed on a machine with 16 Intel Xeon E5-2630 v3 cores (32 threads) clocked at 2.4 Ghz, with 193 GBs of RAM, running Linux 3.13.0, 64 bits. Our implementation is in standard C++11 and freely available at https://github.com/jermp/tongrams. The code was compiled with gcc 5.4.1, using the highest optimization settings.

The data structures were saved to disk after construction, and loaded into main memory to be queried. To test the speed of lookup queries, we use a query set of 5 million random N-grams for YahooV2 and GoogleV2 and of 0.5 million for Europarl. In order to smooth the effect of fluctuations during measurements, we repeat each experiment five times and consider the mean. All query algorithms were run on a single core.

5.1 Elias-Fano Tries

Gram-ID sequences. Table 1 shows the average number of bytes per gram including the cost of pointers, and lookup speed per query. The first two rows refers to the trie data structure described in Section 3.1, when the sorted arrays are encoded with Elias-Fano (EF) and partitioned Elias-Fano (PEF) [30]. Subsequent rows indicate the space gains obtained by applying the context-based remapping strategy using EF and PEF for contexts of lengths respectively 1 and 2. In particular, for GoogleV2 we use a context of length 1, as the number of tri-grams roughly takes 66% of the whole number of N-grams in the dataset, thus it would make little sense to optimize only the space of 4- and 5-grams.

As expected, partitioning the gram sequences using PEF yields a better space occupancy. However, though the paper by Ottaviano and Venturini [30] describes a dynamic programming algorithm that finds the partitioning able of minimizing the space occupancy of a monotone sequence, we instead adopt a *uniform* partitioning strategy. Partitioning the sequence uniformly has several advantages over variable-length partitions for our setting. As we have seen in Section 3.1, trie searches are carried out by performing a preliminary random access to the endpoints of the range pointed to by a pointer pair. Then a search in the range follows to determine the position of the gram-ID. Partitioning the sequence by variable-length blocks introduces an additional search over the sequence of partition endpoints to determine the proper block in which the search must continue. While this preliminary search only introduces a minor overhead in query processing for inverted index queries [30] (as it has to be performed once and successive accesses are only directed to forward positions of the sequence), it

is instead the major bottleneck when random access operations are very frequent as in our case. By resorting on uniform partitions, we eliminate this first search and the cost of representation for the variable-length sizes. To speed up queries even further, we also keep the upper bounds of the blocks uncompressed and bit-aligned. We use partitions of 64 integers for bi-gram sequences, and of 128 for all other orders, i.e., for $N \geq 3$. Shrinking the size of blocks has the potential of speeding-up searches over plain Elias-Fano because a successor query has to be resolved over an interval potentially much smaller than a range length (number of successors of a gram). However, excessively reducing the block size will ruin the advantage in space reduction. We use 64-integer blocks for bi-gram sequences because these must be searched several times during the query-mapping phase when the context-based remapping is adopted. In conclusion, as we can see by the second row of Table 1, this sequence organization brings a negligible overhead in query processing speed (less than 8% on Europarl and YahooV2), while maintaining a noticeable space reduction (up to 29% on GoogleV2).

Concerning the efficacy of the context-based remapping, we have that remapping the gram IDs with a context of length $k = 1$ is already able of reducing the space of the sequences by ≈13% on average when sequences are encoded with Elias-Fano, with respect to the EF cost. If we consider a context of length $k = 2$ we *double* the gain, allowing for more than 28% of space reduction *without* affecting the lookup time with respect to the case $k = 1$. As a first conclusion, when space efficiency is the main concern, it is always convenient to apply the remapping strategy with a context of length 2. The gain of the strategy is even more evident with PEF: this is no surprise as the encoder can better exploit the reduced IDs by encoding all the integers belonging to a block with a universe relative to the block and not to the whole sequence. This results in a space reduction of more than 36% on average and up to 39% on GoogleV2. However, as explained in Section 3, the remapping strategy comes with a penalty at query time as we have to map an ID before it can be searched in the proper gram sequence. On average, we found that 30% more time is spent with respect to the Elias-Fano baseline. Notice that PEF does *not* introduce any time degradation with respect to EF with context-based remapping: it is actually faster on GoogleV2.

Frequency counts. For the representation of frequency counts we compare three different encoding schemes: the first one refers to the strategy described in Section 3 that assigns variable-length codewords (CW) to the ranks of the counts and keeps track of codewords length using a binary vector (BV), the other two schemes transform the sequence of count ranks into a non-decreasing sequence by taking its prefix sums (PF) and then applies EF or PEF. Table 3 shows the average number of bytes per count for these different strategies. The reported space also includes the space for the storage of the arrays containing the distinct counts for each order of N. As already pointed out, these take a negligible amount of space because the distribution of frequency counts is highly repetitive (see Table 2). The percentages of PS + EF and PS + PEF are done with respect to the first row CW + BV.

The time for retrieving a count was pretty much the same for all the three techniques. Prefix-summing the sequence and apply EF does not bring any advantage over the codeword assignment

	Europarl	YahooV2	GoogleV2
CW + BV	0.36	0.47	1.46
PS + EF	0.35 (−1.59%)	0.62 (+32.46%)	1.59 (+9.17%)
PS + PEF	0.30 (−16.65%)	0.51 (+8.67%)	1.30 (−11.03%)
VLBC	0.76 (+155.63%)	0.79 (+55.86%)	1.32 (+1.44%)
PLAIN	1.63 (+444.74%)	2.00 (+294.17%)	2.63 (+102.43%)
VByte + BV	3.21 (+975.40%)	3.32 (+554.66%)	—

Table 3: Average bytes per count (bpc).

technique because its space is practically the same on Europarl but it is actually larger on both YahooV2 (by up to 32%) and GoogleV2. These two reasons together place the codeword assignment technique in net advantage over EF. PEF, instead, offers a better space occupancy of more than 16% on Europarl and 10% on GoogleV2. Therefore, in the following we assume this representation for frequency counts, except for YahooV2, where we adopt CW + BV.

We also report the space occupancy for the counts representation of BerkeleyLM and Expgram which, differently from all other competitors, can also be used to index frequency counts. BerkeleyLM COMPRESSED variant uses the variable-length block coding (VLBC) explained in Section 2 to compress count ranks, whereas the HASH variant stores uncompressed count ranks, referred to as PLAIN in the table, using the minimum number of bits necessary for their representation (see Table 2). Expgram, instead, does not store count ranks but directly compress the counts themselves using Variable-Byte encoding with an additional binary vector as to be able of randomly accessing the counts sequence. The available RAM of our test machine (192 GBs) was not sufficient to successfully build Expgram on GoogleV2. The same holds for KenLM and Marisa, as we are going to see in the next Section. Therefore, we report its space for Europarl and YahooV2. We first observe that rank-encoding schemes are far more advantageous than compressing the counts themselves, as done by Expgram. Moreover, none of the these techniques beats the three ones we previously introduced, except for the BerkeleyLM COMPRESSED variant which is ≈10% smaller on GoogleV2 with respect to CW + BV. However, notice that this gap is completely closed as soon as we adopt the combination PS + PEF.

Before concluding the subsection, we use the analysis to fix two different trie data structures that respectively privilege space efficiency and query time: we call them PEF-RTrie and PEF-Trie. For the PEF-RTrie variant we use PEF for representing the gram-ID sequences; PEF for the counts on Europarl and GoogleV2 but CW + BV for YahooV2. We also use the maximum applicable context length for the context-based remapping technique, i.e., 2 for Europarl and YahooV2; 1 for GoogleV2. For the PEF-Trie variant we choose a data structure using PEF for representing gram-ID sequences and the codeword assignment technique for the counts, without remapping. The corresponding size breakdowns are shown in Figures 3c and 3d respectively. Pointer sequences take very little space for both data structures, while most of the difference lies, not surprisingly, in space of the gram-ID sequences. The timing
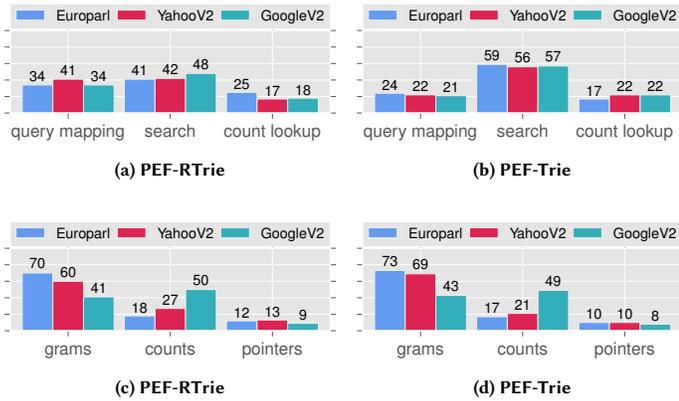
Figure 3: Trie data structures timing (a-b) and size (c-d) breakdowns in percentage on the tested datasets.

breakdowns in Figures 3a and 3b clearly highlight, instead, how the context-based remapping technique rises the time we spend in the query-mapping phase, during which the IDs are mapped to their reduced IDs.

5.2 Hashing

We build our MPH tables using 8-byte hash keys, as to yield a false positive rate of 2^{-64}. For each different value of N we store the distinct count values in an array, uncompressed and byte-aligned using 4 bytes per distinct count on Europarl and YahooV2; 8 bytes on GoogleV2.

For all the three datasets, the number of bytes per gram, including also the cost of the hash function itself (0.33 bytes per gram) is 8.33. The number of bytes per count is given by the sum of the cost for the ranks and the distinct counts themselves and is equal to 1.41, 1.74 and 2.43 for Europarl, YahooV2 and GoogleV2 respectively. Not surprisingly, the majority of space is taken by the hash keys: clients willing to reduce this memory impact can use 4-byte hash keys instead, at the price of a higher false positive rate (2^{-32}). Therefore, it is worth observing that spending additional effort in trying to lower the space occupancy of the counts only results in poor improvements as we pay for the high cost of the hash keys.

The constant-time access capability of hashing makes gram lookup extremely fast, by requiring on average $\frac{1}{3}$ of a micro second per lookup (exact numbers are reported in Table 4). In particular, all the time is spent in computing the hash function itself and access the relative table location: the final count lookup is totally negligible.

5.3 Overall comparison

In this subsection we compare the performance of our selected trie-based solutions, PEF-RTrie and PEF-Trie, as well as our minimal perfect hash approach against the competitors introduced at the beginning of the section. The results of the comparison are shown in Table 4, where we report the space taken by the representation of the gram-ID sequences and lookup time per query in micro seconds. For the trie data structures, the reported space also includes the cost

of representation for the pointers. We compare the space of representation for the N-grams excluding their associated information because this varies according to the chosen implementation: for example, KenLM can only store probability and backoffs, whereas BerkeleyLM can be used to store either counts or probabilities. For those competitors storing frequency counts, we already discussed their count representation in subsection 5.1. Expgram, KenLM and Marisa require too much memory for the building of their data structures on GoogleV2, therefore we mark as empty their entry in the table for this dataset.

Except for the last two rows of the table in which we compare the performance of our MPH table against KenLM probing (P.), we write for each competitor two percentages indicating its score against our selected trie data structures PEF-Trie and PEF-RTrie, respectively. Let us now examine each row one by one. In the following, unless explicitly indicated, numbers refer to average values over the different datasets.

BerkeleyLM COMPRESSED (C.) variant results ≈21% larger than our PEF-RTrie implementation and slower by more than 70%. It gains, instead, an advantage of roughly 9% over our PEF-Trie data structure, but it is also more than 2 times slower. The HASH variant uses hash tables with linear probing to represent the nodes of the trie. Therefore, we test it with a small extra space factor of 3% for table allocation (H.3) and with 50% (H.50), which is also used as the default value in the implementation, as to obtain different time/space trade-offs. Clearly the space occupancy of both hash variants do not compete with the ones of our proposals as these are from 3 to 7 times larger, but the $O(1)$-lookup capabilities of hashing makes it faster than a sorted array trie implementation: while this is no surprise, notice that our PEF-Trie data structure is anyway competitive as it is actually faster on GoogleV2.

Expgram is ≈13.5% larger than PEF-Trie and also 2 and 5 times slower on Europarl and YahooV2 respectively. Our PEF-RTrie data structure retains an advantage in space of 60% and it is still significantly faster: of about 72% on Europarl and 4.3 times on YahooV2.

KenLM is the fastest trie language model implementation in the literature. As we can see, our PEF-Trie variant retains 70% of its space with a negligible penalty at query time. Compared to PEF-RTrie, it results a little faster, i.e., ≈15%, but also 2.3 and 2.5 times larger on Europarl and YahooV2 respectively.

We also tested the performance of Marisa even though it is not a trie optimized for language models as to understand how our data structures compare against a general-purpose string dictionary implementation. We outperform Marisa in both space and time: compared to PEF-RTrie, it is 2.7 times larger and 38% slower; with respect to PEF-Trie it is more than 90% larger and 70% slower.

RandLM is designed for small memory footprint and returns approximated frequency counts when queried. We build its data structures using the default setting recommended in the documentation: 8 bits for frequency count quantization and 8 bits per value as to yield a false positive rate of $\frac{1}{256}$. While being from 2.3 to 5 times slower than our exact and lossless approach, it is quite compact because the quantized frequency counts are recomputed on the fly using the procedure described in Section 2. Therefore, while its space occupancy results even larger with respect to our grams representation by 61%, it is still no better than the whole space of our PEF-RTrie data structure. With respect to the whole

	Europarl		YahooV2		GoogleV2	
	bpg	μs × query	bpg	μs × query	bpg	μs × query
PEF-Trie	1.87	1.35	1.91	1.73	1.52	1.91
PEF-RTrie	1.28	1.64	1.38	2.15	1.31	2.30
BerkeleyLM C.	1.70 (−8.89%) (+32.90%)	2.83 (+108.88%) (+72.70%)	1.69 (−11.41%) (+22.04%)	3.48 (+101.84%) (+61.70%)	1.45 (−4.87%) (+10.83%)	4.13 (+116.57%) (+79.76%)
BerkeleyLM H.3	6.70 (+258.81%) (+423.40%)	0.97 (−28.46%) (−40.85%)	7.82 (+310.38%) (+465.36%)	1.13 (−34.35%) (−47.41%)	9.24 (+507.79%) (+608.07%)	2.18 (+13.95%) (−5.42%)
BerkeleyLM H.50	7.96 (+326.03%) (+521.45%)	0.97 (−28.49%) (−40.88%)	9.37 (+391.32%) (+576.87%)	0.96 (−44.27%) (−55.35%)	—	—
Expgram	2.06 (+10.18%) (+60.73%)	2.80 (+106.61%) (+70.82%)	2.24 (+17.36%) (+61.68%)	9.23 (+435.33%) (+328.87%)	—	—
KenLM T.	2.99 (+60.11%) (+133.56%)	1.28 (−5.47%) (−21.84%)	3.44 (+80.39%) (+148.52%)	1.94 (+12.32%) (−10.01%)	—	—
Marisa	3.61 (+93.09%) (+181.66%)	2.06 (+52.00%) (+25.67%)	3.81 (+99.60%) (+174.98%)	3.24 (+87.96%) (+50.58%)	—	—
RandLM	1.81 (−3.06%) (+41.41%)	4.39 (+224.20%) (+168.04%)	2.02 (+6.18%) (+46.29%)	5.08 (+194.35%) (+135.82%)	2.60 (+70.73%) (+98.90%)	9.25 (+384.54%) (+302.19%)
MPH	8.33	0.26	8.33	0.32	8.33	0.37
KenLM P.3	9.40 (+12.87%)	0.43 (+62.60%)	9.41 (+13.03%)	0.38 (+20.08%)	—	—
KenLM P.50	16.91 (+103.11%)	0.31 (+16.83%)	16.92 (+103.25%)	0.34 (+7.84%)	—	—

Table 4: Average bytes per gram (bpg) and average Lookup time per query in micro seconds.

space of PEF-Trie, it retains instead an advantage of ≈15.6%. This space advantage is, however, compensated by a loss in precision and a much higher query time (up to 5 times slower on GoogleV2).

The last two rows of Table 4 regard the performance of our MPH table with respect to KenLM PROBING. As similarly done for BerkeleyLM H., we also test the PROBING data structure with 3% (P.3) and 50% (P.50) extra space allocation factor for the tables. While being larger as expected, the KenLM implementation makes use of expensive hash key recombinations that yields a slower random access capability with respect to our minimal perfect hashing approach.

Perplexity. Besides the efficient indexing of frequency counts, our data structures can also be used to map *N*-grams to language model probabilities and backoffs. As done by KenLM, we also use the binning method [14] to quantize probabilities and backoffs, but allowing any quantization bits ranging from 2 to 32. Uni-grams values are stored unquantized to favor query speed: as vocabulary size is typically very small compared to the number of total *N*-grams, this has a minimal impact on the space of the data structure. Our trie implementation is *reversed* as to permit a more efficient computation of sentence-level probabilities, with a *stateful* scoring function that carries its state on from a query to the next as similarly done by KenLM and BerkeleyLM.

For the perplexity benchmark we used the query dataset, publicly available at http://www.statmt.org/lm-benchmark, that contains 306 688 sentences, for a total of 7 790 011 tokens [6]. We used Expgram utilities to build modified Kneser-Ney [7] 5-gram language models from the counts of Europarl and YahooV2 that have an OOV (out of vocabulary) rate of, respectively, 16% and 1.82% on

the test query file. As Expgram only builds quantized models using 8 quantization bits for both probabilities and backoffs, we also use this number of quantization bits for our tries and KenLM trie.

	Europarl		YahooV2	
	bpg	μs × query	bpg	μs × query
PEF-Trie	3.48	0.25	3.64	0.38
PEF-RTrie	2.91	0.28	3.06	0.43
BerkeleyLM C.	6.50 (+87.03%) (+123.47%)	1.19 (+371.79%) (+322.22%)	6.39 (+75.72%) (+109.21%)	1.08 (+187.45%) (+152.17%)
BerkeleyLM H.3	9.36 (+169.17%) (+221.61%)	0.84 (+233.63%) (+198.58%)	8.75 (+140.41%) (+186.23%)	0.74 (+95.77%) (+71.75%)
BerkeleyLM H.50	12.31 (+254.00%) (+322.97%)	0.35 (+39.00%) (+24.39%)	12.01 (+230.05%) (+292.95%)	0.30 (−19.39%) (−29.28%)
Expgram	4.15 (+19.33%) (+42.59%)	3.83 (+1424.87%) (+1264.67%)	5.80 (+59.41%) (+89.79%)	14.05 (+3637.90%) (+3179.16%)
KenLM T.	4.58 (+31.80%) (+57.48%)	0.23 (−8.00%) (−17.66%)	5.04 (+38.53%) (+64.93%)	0.39 (+4.57%) (−8.26%)
RandLM	4.01 (+15.42%) (+37.90%)	6.48 (+2477.95%) (+2207.12%)	3.86 (+6.03%) (+26.24%)	6.25 (+1561.20%) (+1357.33%)
MPH	9.92	0.15	9.94	0.24
KenLM P.3	14.77 (+48.90%)	0.32 (+106.38%)	14.84 (+49.24%)	0.30 (+24.82%)
KenLM P.50	21.48 (+116.59%)	0.10 (−36.37%)	21.57 (+116.89%)	0.15 (−40.16%)

Table 5: Perplexity benchmark results reporting average number of bytes per gram (bpg) and micro seconds per query using modified Kneser-Ney 5-gram language models built from Europarl and YahooV2 counts.

For all data structures, BerkeleyLM truncates the mantissa of floating-point values to 24 bits and then stores indices to distinct probabilities and backoffs. RandLM was build, as already said, with the default parameters recommended in the documentation.

Table 5 shows the results of the benchmark. As we can see, the PEF-Trie data structure is as fast as the KenLM trie while being more than 30% more compact on average; the PEF-RTrie variant *doubles* the space gains with negligible loss in query processing speed ($\approx$13% slower). We instead significantly outperform all other competitors in both space and time, including the BerkeleyLM H.3 variant. In particular, notice that we are also smaller than RandLM which is randomized and, therefore, less accurate. The query time of BerkeleyLM H.50 is smaller on YahooV2; however, it also uses from 3 up to 4 times the space of our tries.

The last two rows of the table are dedicated to the comparison of our MPH table with KenLM PROBING. While our data structure stores quantized probabilities and backoffs, KenLM stores uncompressed values for all orders of N. We found out that storing unquantized values results in indistinguishable differences in perplexity while unnecessarily increasing the space of the data structure, as it is apparent in the results. The expensive hash key recombinations necessary for random access are avoided during perplexity computation for the left-to-right nature of the query access pattern. This makes, not surprisingly, a linear probing implementation actually faster, by 38% on average, than a minimal perfect hash approach when a large multiplicative factor is used for tables allocation (P.50). The price to pay is the double of the space. On the other hand, the P.3 variant is larger (by 50%) and slower (by 60% on average).

6 CONCLUSION AND FUTURE WORK

We have presented highly compact and fast indexes for N-grams datasets that achieve substantial performance improvements over state-of-the-art approaches. Our trie data structure represents each level of the trie with Elias-Fano, preserving the query processing speed of the fastest implementation in the literature. We have also described how to reduce its memory footprint by introducing a context-based remapping for vocabulary tokens. This technique offers, on average, an additional 28% of space reduction with a context of length 1 and 35% with a context of length 2, with only a slight penalty at query processing speed.

Some interesting research problems naturally arise from the ideas described in the paper. We mention some of them. Is the frequency-based assignment of vocabulary tokens optimal for Elias-Fano? An interesting, yet hard, research problem could focus on devising an ID-assignment strategy with proven guarantees of optimality for Elias-Fano and for our context-based remapping technique. Additional efforts could also be spent on making trie searches even faster, e.g., by exploiting SIMD processor special instructions.

ACKNOWLEDGMENTS

REFERENCES

[1] 2006. Yahoo! N-Grams, version 2.0. http://webscope.sandbox.yahoo.com/catalog.php?datatype=l.
[2] Ziv Bar-Yossef and Naama Kraus. 2011. Context-sensitive query auto-completion. In *WWW*. 107–116.
[3] Djamal Belazzougui, Paolo Boldi, Giuseppe Ottaviano, Rossano Venturini, and Sebastiano Vigna. 2014. Cache-oblivious peeling of random hypergraphs. In *DCC*. 352–361.
[4] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM*. 422–426.
[5] Thorsten Brantz and Alex Franz. 2006. The Google Web 1T 5-Gram Corpus. http://storage.googleapis.com/books/ngrams/books/datasetsv2.html. In *Linguistic Data Consortium, Philadelphia, PA, Technical Report LDC2006T13*.
[6] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. 2014. One billion word benchmark for measuring progress in statistical language modeling. In *INTERSPEECH*. 2635–2639.
[7] Stanley F. Chen and Joshua Goodman. 1996. An empirical study of smoothing techniques for language modeling. In *ACL*. 310–318.
[8] David Clark. 1996. *Compact Pat Trees*. Ph.D. Dissertation. University of Waterloo.
[9] David R. Clark and J. Ian Munro. 1996. Efficient suffix trees on secondary storage. In *SODA*. 383–391.
[10] Bruce Croft, Donald Metzler, and Trevor Strohman. 2009. *Search Engines: Information Retrieval in Practice* (1st ed.). Addison-Wesley Publishing Company.
[11] Erik D. Demaine, Thouis Jones, and Mihai Pătraşcu. 2004. Interpolation search for non-independent data. In *SODA*. 529–530.
[12] Peter Elias. 1974. Efficient Storage and Retrieval by Content and Address of Static Files. In *JACM*. 246–260.
[13] Robert Mario Fano. 1971. On the number of bits required to implement an associative memory. In *Memorandum 61, Computer Structures Group, MIT*.
[14] Marcello Federico and Nicola Bertoldi. 2006. How many bits are needed to store probabilities for phrase-based translation?. In *WMT*. 94–101.
[15] Edward Fredkin. 1960. Trie memory. In *Communications of the ACM*. 490–499.
[16] Kimmo Fredriksson and Fedor Nikitin. 2007. Simple compression code supporting random access and fast string matching. In *WEA*. 203–216.
[17] Kenneth Heafield. 2011. KenLM: Faster and smaller language model queries. In *WMT*. 187–197.
[18] Steffen Heinz, Justin Zobel, and Hugh E. Williams. 2002. Burst tries: a fast, efficient data structure for string keys. In *TOIS*. ACM, 192–223.
[19] Samuel Huston, Alistair Moffat, and W. Bruce Croft. 2011. Efficient indexing of repeated n-grams. In *WSDM*. 127–136.
[20] Guy Jacobson. 1989. Space-efficient Static Trees and Graphs. In *FOCS*. 549–554.
[21] Dan Jurafsky and James H. Martin. 2014. *Speech and language processing*. Pearson.
[22] Philipp Koehn. 2005. Europarl: A parallel corpus for statistical machine translation. http://www.statmt.org/europarl. In *MT summit*. 79–86.
[23] Karen Kukich. 1992. Techniques for automatically correcting words in text. In *CSUR*. 377–439.
[24] Ted G. Lewis and Curtis R. Cook. 1988. Hashing for dynamic and static internal tables. In *Computer*. 45–56.
[25] Bhaskar Mitra and Nick Craswell. 2015. Query auto-completion for rare prefixes. In *CIKM*. 1755–1758.
[26] Bhaskar Mitra, Milad Shokouhi, Filip Radlinski, and Katja Hofmann. 2014. On user interactions with query auto-completion. In *SIGIR*. 1055–1058.
[27] Alistair Moffat and Lang Stuiver. 2000. Binary Interpolative Coding for Effective Index Compression. In *IRJ*. 25–47.
[28] Donald R. Morrison. 1968. PATRICIA: practical algorithm to retrieve information coded in alphanumeric. In *JACM*. 514–534.
[29] Gonzalo Navarro, Ricardo A. Baeza-Yates, Erkki Sutinen, and Jorma Tarhio. 2001. Indexing methods for approximate string matching. In *IEEE Data Eng. Bull.* 19–27.
[30] Giuseppe Ottaviano and Rossano Venturini. 2014. Partitioned Elias-Fano Indexes. In *SIGIR*. 273–282.
[31] Adam Pauls and Dan Klein. 2011. Faster and Smaller N-gram Language Models. In *ACL*. 258–267.
[32] Bhiksha Raj and Ed Whittaker. 2003. Lossless Compression of Language Model Structure and Word Identifiers. In *ICASSP*. 388–391.
[33] David Salomon. 2007. *Variable-length Codes for Data Compression*. Springer.
[34] Jangwon Seo and W. Bruce Croft. 2008. Local text reuse detection. In *SIGIR*. 571–578.
[35] David Talbot and Miles Osborne. 2007. Randomised language modelling for statistical machine translation. In *ACL*. 512–519.
[36] Sebastiano Vigna. 2013. Quasi-succinct indices. In *WSDM*. 83–92.
[37] Taro Watanabe, Hajime Tsukada, and Hideki Isozaki. 2009. A succinct n-gram language model. In *IJCNLP*. 341–344.
[38] Susumu Yata. 2011. Prefix/Patricia trie dictionary compression by nesting Prefix/Patricia tries. In *NLP*.