Handling Massive N-Gram Datasets Efficiently

GIULIO ERMANNO PIBIRI and ROSSANO VENTURINI, University of Pisa and ISTI-CNR, Italy

Abstract. This paper deals with the two fundamental problems concerning the handling of large n-gram language models: *indexing*, that is compressing the n-gram strings and associated satellite data without compromising their retrieval speed; and *estimation*, that is computing the probability distribution of the strings from a large textual source. Performing these two tasks efficiently is fundamental for several applications in the fields of Information Retrieval, Natural Language Processing and Machine Learning, such as auto-completion in search engines and machine translation.

Regarding the problem of indexing, 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 solutions and related software packages. In particular, we present a compressed 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 lower the space of representation to compression levels that were never achieved before. Despite the significant savings in space, our technique introduces a negligible penalty at query 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 in the literature, that are both quantized and lossy, are not smaller 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.

Regarding the problem of estimation, we present a novel algorithm for estimating *modified Kneser-Ney* language models, that have emerged as the de-facto choice for language modeling in both academia and industry, thanks to their relatively low perplexity performance. Estimating such models from large textual sources poses the challenge of devising algorithms that make a parsimonious use of the disk. The state-of-the-art algorithm uses three sorting steps in external memory: we show an improved construction that requires only one sorting step thanks to exploiting the properties of the extracted n-gram strings. With an extensive experimental analysis performed on billions of n-grams, we show an average improvement of 4.5× on the total running time of the state-of-the-art approach.

Authors' address: Giulio Ermanno Pibiri; Rossano Venturini, University of Pisa and ISTI-CNR, Pisa, Italy, giulio.pibiri@di.unipi.it, rossano.venturini@unipi.it.

1 INTRODUCTION

The use of n-grams is wide and vital for many tasks in Information Retrieval, Natural Language Processing and Machine Learning, such as: auto-completion in search engines [2, 37, 38], spelling correction [34], similarity search [33], identification of text reuse and plagiarism [28, 48], automatic speech recognition [30] and machine translation [25, 44], to mention some of the most notable.

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, 37, 38]. 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, 15, 30, 37, 38]. 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 [34]. 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 [15, 30, 34].

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/backoff weights for word-predicting computations [25, 44]. The efficiency of the data structure should come 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 [15, 30]. To reduce the memory-access rate and, *hence*, speed up the execution of the retrieval algorithms, the design of an efficient *compressed* representation of the data structure appears as mandatory. While several solutions have been proposed for the indexing and retrieval of n-grams, either based on *tries* [23] or *hashing* [35], their practicality is actually limited because of some important inefficiencies that we discuss below.

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 [25]. In fact, space reductions are usually achieved by either: lossy quantization of satellite values, or by randomized approaches with false positive allowed [52]. 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 [44]. 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. Finally, hashing schemes based on open addressing with linear probing result extremely large for static corpora as long as the tables are allocated with $30 - 50\%$ extra space to allow fast random access [25, 44].

Since a solution that is compact, fast and lossless at the same time is still missing, the first aim of this paper is that of addressing the aforementioned inefficiencies by introducing compressed data structures that, despite their small memory footprint, support efficient random access to the satellite n-gram values. We refer to such problem as the on of *indexing* n-gram datasets.

The other correlated problem that we study in this paper is the one of computing the probability distribution of the n-grams extracted from large textual collections. We refer to this second problem as the one of *estimation*. In other words, we would like to create and efficient, compressed, index

that maps the n-grams of a large text to its probability of occurrence in the text. Clearly, the way such probability is computed depends on the chosen model. This is an old problem and has received a lot of attention: not surprisingly, several models have been proposed in the literature, such as Laplace, Good-Turing, Katz, Jelinek-Mercer, Witten-Bell and Kneser-Ney (see [9, 10] and references therein for a complete description and comparison).

Among the many, *Kneser-Ney* language models [31] and, in particular, their *modified* version introduced by Chen and Goodman [9], have gained popularity thanks to their relatively low-perplexity performance. This makes modified Kneser-Ney the de-facto choice for language model toolkits. The following software libraries, widely used in both academia and industry (e.g., Google [5, 8] and Facebook [11]), all support modified Kneser-Ney smoothing: KenLM [25], BerkeleyLM [44], RandLM [52], Expgram [57], MSRLM [42], SRILM [51], IRSTLM [21] and the recent approach based on suffix trees by Shareghi et al. [49, 50]. For such reasons, Kneser-Ney is the model we consider in this work too and that we review in Section 4.

The current limitation of the mentioned software libraries is that estimation of such models occurs in internal memory and, as a result, these are not able to scale to the dimensions we consider in this work. An exception is represented by the work of Heafield, Pouzyrevsky, Clark, and Koehn [26] (KenLM) that contributed an estimation algorithm involving three steps of sorting in external memory. Their solution embodies the current state-of-art solution to the problem: the algorithm takes, on average, as low as 20% of the CPU and 10% of the RAM of the cited toolkits [26]. Therefore, our work aims at improving upon the I/O efficiency of this approach.

1.1 Our 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* [18, 19] 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 to demonstrate that 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. Our data structures outperform all proposals at the state-of-the-art for space usage, *without* compromising their time performance. More precisely, the most space-efficient proposals in the literature, that 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.

(4) We design a faster estimation algorithm that requires only one step of sorting in external memory, as opposed to the state-of-the-art approach [26] that requires three steps of sorting. The result is achieved by the careful exploitation of the properties of the extracted n-gram strings. Thanks to such properties, we show how it is possible to perform the whole estimation on the *context*-sorted strings and, yet, be able to efficiently lay out a reverse trie data structure, indexing such strings in *suffix* order. We show that saving two steps of sorting in external memory yields a solution that is 2.87× faster on average than the fastest algorithm proposed in the literature.

(5) We introduce many optimizations to further enhance the running time of our proposal, such as: asynchronous CPU and I/O threads, parallel LSD radix sort, block-wise compression and multi-threading. With an extensive experimental analysis conducted over large textual datasets, we study the behavior of our solution at each step of estimation; quantify the impact of the introduced optimizations and consider the comparison against the state-of-the-art. The devised optimizations further improve the running time by 1.6× on average, making our algorithm 4.5× faster than the state-of-the-art solution.

1.2 Paper organization

Although the two problems we address in this paper, i.e., indexing and estimation, are strictly correlated, we treat them one after the other in order to introduce the whole material in an incremental way without burdening the exposition. In particular, we show the experimental evaluation right after the description of our techniques for each problem, rather than deferring it to the end of the paper. We believe this form is the most suitable to convey the results that we want to document with this paper. In our intention, each section of this document is an independent unit of exposition. Based on the following observations, the paper is structured as follows.

Section 2 fixes the notation and provides some basic notions about the n-grams. More detailed background will be provided when needed in the relevant (sub-)sections of the paper.

Section 3 treats the problem of indexing. Subsection 3.1 reviews the standard data structures used to index n-gram datasets in compressed space and how these are used by the proposals in the literature. Subsections 3.2 and 3.3 describe our compressed data structures, whose efficiency is validated in Subsection 3.4 with a rich set of experiments.

Section 4 treats the problem of estimation. After reviewing the Kneser-Net smoothing technique in Subsection 4.1, we describe the state-of-the-art approach in Subsection 4.2 because we aim at improving the efficiency of that algorithm. We present our improved estimation process in Subsection 4.3 and test its performance in Subsection 4.4. We conclude the paper in Section 5.

2 BACKGROUND AND NOTATION

A *language model* (LM) is a probability distribution $\mathbb{P}(\mathcal{S})$ that describes how often a string $w_1^n = w_1 \cdots w_n$ drawn from the set $\mathcal{S}$ appears in some domain on interest. The central goal of a language model is to compute the probability of the word w_n given its preceding history of $n-1$ words, called the *context*, that is: $\mathbb{P}(w_n|w_1^{n-1})$ for all $w_1^n \in \mathcal{S}$. Informally, the goal is to *predict* the "next" word following a given context.

When *efficiency* is the main concern, n-gram language models are adopted. A *n-gram* is a sequence of at most n tokens. A token can be either a single character or a word, the latter intended as a sequence of characters delimited by a special symbol, e.g., a whitespace character. Unless otherwise specified, throughout the paper we consider n-grams as consisting of words. Since we impose that $1 \leq n \leq N$, where N is a small constant, (e.g., typically $N = 5$), dealing with strings of this form permits to work with a context of *at most* $N-1$ preceding words. This ultimately implies that the aforementioned probability $\mathbb{P}(w_n|w_1^{n-1}) = \prod_{k=1}^{n+1} \mathbb{P}(w_k|w_1^{k-1})$ can be approximated with $\prod_{k=1}^{n+1} \mathbb{P}(w_k|w_{k-N-1}^{k-1})$. The way each N-gram probability $\mathbb{P}(w_k|w_{k-N-1}^{k-1})$ is computed depends on the chosen model.

Several models have been proposed in the literature, such as Laplace, Good-Turing, Katz, Jelinek-Mercer, Witten-Bell and Kneser-Ney (see [9, 10] and references therein for a complete description and comparison). For a n-gram backoff-smoothed language model, the probability of w_n with

context w_1^{n-1} is assigned according to the following recursive equation

$$\mathbb{P}(w_n|w_1^{n-1}) = \begin{cases} \mathbb{P}(w_n|w_1^{n-1}) & \text{if } n\text{-gram } w_1^n \in \mathcal{S} \\ b(w_1^{n-1})\mathbb{P}(w_n|w_2^{n-1}) & \text{otherwise} \end{cases}$$

that is: if the model has enough information we use the full distribution $\mathbb{P}(w_n|w_1^{n-1})$, otherwise we *backoff* to the lower-order distribution $\mathbb{P}(w_n|w_2^{n-1})$ with penalty $b(w_1^{n-1})$.

Clearly, the bigger the language model the more accurate the computed probability will be. In other words, predictions will be more accurate when more n-grams are used to estimate the probability of a word following a given context. Therefore, we would like to handle as many n-grams as possible: this paper describes techniques to handle several billions of n-grams. Such n-gram strings are extracted from *text*, from any of its different incarnations, e.g., web pages, novels, code fragments and scientific articles, by adopting a *sliding-window* approach. A window of n words, for $1 \leq n \leq N$, slides over a text counting the number of times such n words appear in the text. This counting process is usually implemented using a hash data structure, whose keys are the distinct n-gram strings and the values the accumulated frequency counts: if the extracted n-gram is not already present in the table, a new entry is allocated with associated value 1; otherwise the corresponding value is incremented by 1. This process is repeated for different widow sizes over huge text corpora: this gives birth to colossal datasets in terms of number of distinct strings. As a concrete example, if all distinct n-grams for the values of n ranging from 1 to 5 are extracted from the Agner Fog's manual *Optimizing software in* C++ [22], we obtain the following numbers of distinct n-grams: 8761 1-grams, 38 900 2-grams, 61 516 3-grams, 70 186 4-grams and 73 187 5-grams. Thus more than 250 thousands distinct grams for already 164 pages written in English. Google did the same but on approximately 8 million books, or 6% of all books ever published [36], yielding a dataset of more than 11 billion N-grams (see also Table 1). This motivates and helps understanding the need for efficient data structures, in both memory footprint and access speed, able to manage such quantity of strings.

3 COMPRESSED INDEXES

The problem we tackle in this section of the paper is the one of representing in compressed space a dataset of n-gram strings and their associated values, being either frequency counts (integers) or probabilities (floating points). Given a n-gram string, the compressed data structure should allow fast random access to the corresponding associated value by means of the operation Lookup.

3.1 Related Work

In this subsection we first discuss the classic data structures used to represent efficiently large n-gram datasets, highlighting the advantages/disadvantages of these approaches in relation to the structural properties that n-gram datasets exhibit. Next, we consider how these approaches have been adopted by different proposals in the literature. Two different data structures are mostly used to store large and sparse n-grams datasets: *tries* [23] and *hash tables* [35].

Tries. A trie is a tree data structure devised for efficient indexing and search of string dictionaries, in which 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 (few) 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.

Hash tables. Hashing is another way to implement associative arrays: for each value of n from 1 to 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 (typically the 4 or 8-byte hash of the n-gram itself) and the satellite data for the n-gram. This data structure permits to access the specified n-gram data in expected constant time. 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 exactly n searches in the constituent arrays. Conversely, hashing is designed for speed but sacrifices space-efficiency since its 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.

State-of-the-art. The paper by Pauls and Klein [44] 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 [25] 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 finding a word ID in a trie level of size m can be done in[1] $O(\log \log m)$ with high probability by using *interpolation* search [16]. Records in each sorted arrays are minimally sized at the bit level, improving the memory consumption over [44]. Pointers are compressed using the integer compressor devised in [46]. Values can also be quantized using the *binning* method [20] 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 [52] 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

[1]Unless otherwise specified, all logarithms are in base 2 and $\log x = \lceil \log_2(x+1) \rceil$, $x \geq 0$.

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 one-sided error of the filter and the training scheme ensure that the actual quantized count cannot be larger than the reported value. As the counts are quantized using a logarithmic-scaled codebook, the count will be incremented only a small number of times. The quantized logarithmic count is finally converted back to a linear count.

The use of the succinct encoding LOUDS (Level-Order Unary-Degree Sequence) [29] is advocated in [57] 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 [47, 53], with an additional bitvector used to indicate the boundaries of such byte sequences as to support random access to each element. The paper also discusses the use of block-wise compression (basically gzip on blocks of 8 KB) though it is not used in the implementation for time efficiency reasons. Shareghi et al. [49, 50] also consider the usage of succinct data structures to represent *suffix trees* that can be used to compute Kneser-Ney probabilities on-the-fly. Experimental results indicate that the method is practical for large-scale language modeling although significantly slower to query than leading toolkits for language modeling [25].

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 [13, 27, 41]. It goes without saying that, 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 [59] as the best and practical general-purpose trie implementation. The core idea is to use Patricia tries [40] to recursively represent the nodes of a Patricia trie. This clearly comes with a space/time trade off: the more levels of recursion are used, the greater the space saving but also the higher the retrieval time.

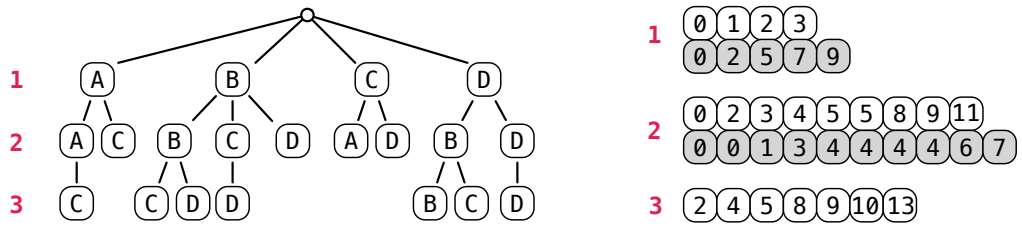
3.2 Elias-Fano Tries

In this subsection we present our main result: a compressed trie data structure, based on the *Elias-Fano* representation [18, 19] 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 [44].

We also introduce 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.

3.2.1 Core Data Structure. This subsection contains the core description of the compressed trie data structure: we dedicate one paragraph to each of its main building components, i.e., how the grams, satellite data and pointers are represented; how searches are implemented.

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



(a) (b)

Fig. 1. On the left (a): example of a trie of order 3, representing the set of grams {A, AA, AAC, AC, B, BB, BBC, BBD, BC, BCD, BD, CA, CD, DB, DBB, DBC, DDD}. On the right (b): the sorted-array representation of the trie. Light-gray arrays represent the pointers.

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[2] {A, B, C, D}, which are assigned IDs $\{0, 1, 2, 3\}$ respectively. Now consider the bi-grams {AA, AC, BB, BC, BD, CA, CD, DB, DD} sorted by IDs. The sequence of the successors of A, referred to as the *range* of A, is ⟨A, C⟩, i.e., $\langle 0, 2\rangle$; the sequence of the successors of B, is ⟨B, C, D⟩, i.e., $\langle 1, 2, 3\rangle$ and so on. Figure 1 shows a graphical representation of what described. 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 that are stored in a hash table, each level of the trie is composed by two integer sequences: one for the representation of the gram-IDs, the other for the pointers. Now, what we need is an efficient encoding for integer sequences. Among the many integer compressors available in the literature (see the book by Salomon [47] for a complete overview), we choose Elias-Fano (along with its partitioned variant [43]), which has been recently applied to inverted index compression showing an excellent time/space trade off [43, 45, 55]. We now describe this elegant integer encoding.

Elias-Fano. Given a monotonically increasing sequence $S(m, u)$ of m positive integers drawn from a universe of size u (i.e., $S[i-1] \leq S[i]$, for any $1 \leq i < m$, with $S[m-1] < u$), we write each $S[i]$ in binary using $\lceil \log u \rceil$ bits. The binary representation of each integer is then split into two parts: a *low* part consisting in the right-most $\ell = \lceil \log \frac{u}{m} \rceil$ bits that we call *low bits* and a *high* part consisting in the remaining $\lceil \log u \rceil - \ell$ bits that we similarly call *high bits*. Let us call ℓ_i and h_i the values of low and high bits of $S[i]$ respectively (notice that, given $\mathcal{S}[i]$: $h_i = \mathcal{S}[i] \gg \ell$ and $\ell_i = \mathcal{S}[i] \& ((1 \ll \ell) - 1)$, where $\ll$ and $\gg$ are the left and right shift operators respectively, & the bitwise AND). The Elias-Fano representation of S is given by the encoding of the high and low parts. The array $L = [\ell_0, \ldots, \ell_{m-1}]$ is written explicitly in $m\lceil \log \frac{u}{m} \rceil$ bits and represents the encoding of the low parts. Concerning the high bits, we represent them in *negated unary*[3] using a

[2] Throughout this subsection we consider, for simplicity, a n-gram as consisting of n capital letters.

[3] The negated unary representation of an integer x is the bitwise NOT of its unary representation $U(x)$. As an example: $U(5) = 000001$ and $\text{NOT}(U(5)) = 111110$.

bit vector of $m + 2^{\lfloor \log m \rfloor} \leq 2m$ bits as follows. We start from a 0-valued bit vector H and set the bit in position $h_i + i$, for all $i = 0, \ldots, m-1$. Finally the Elias-Fano representation of S is given by the concatenation of H and L and overall takes

$$\mathsf{EF}(\mathcal{S}(m,u)) = m\left\lceil \log \frac{u}{m} \right\rceil + 2m \text{ bits.} \quad (1)$$

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 $\mathsf{Select}_1(i)$ 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 $\mathsf{EF}(\mathcal{S}(m,u))$, requiring only $o(m)$ additional bits [12, 54]. Using the Select_1 primitive, it is possible to implement $\mathsf{Access}(i)$, which returns $\mathcal{S}[i]$ for any $0 \leq i < m$, in $O(1)$. We basically have to re-link together the high and low bits of an integer, previously split up during the encoding phase. While the low bits ℓ_i are trivial to retrieve as we need to read the range of bits $[i\ell, (i+1)\ell)$ from L, the high bits deserve a bit more care. Since we write in negated unary how many integers share the same high part, we have a bit set for every integer of S and a zero for every distinct high part. Therefore, to retrieve the high bits of the i-th integer, we need to know how many zeros are present in $H[0, \mathsf{Select}_1(i))$. This quantity is evaluated on H in $O(1)$ as $\mathsf{Rank}_0(\mathsf{Select}_1(i)) = \mathsf{Select}_1(i) - i$. Finally, linking the high and low bits is as simple as: $\mathsf{Access}(i) = ((\mathsf{Select}_1(i) - i) \ll \ell) | \ell_i$, where $\ll$ is the left shift operator and $|$ the bitwise OR.

Partitioned Elias-Fano. The crucial characteristic of the Elias-Fano space bound (1) is that it only depends on two parameters, i.e., the length m and universe u of the sequence, which poorly describe the sequence itself. If the sequence presents regions of close identifiers, i.e., formed by integers that slightly differ from one another, Elias-Fano fails to exploit such natural clusters. Clearly, we would obtain a better space usage if such regions were encoded separately. Partitioning the sequence into chunks to better adapt to such regions of close identifiers is the key idea of the partitioned Elias-Fano representation (PEF in the following) [43].

The core idea is as follows. We partition a sequence $\mathcal{S}(m,u)$ into m/b chunks, each of b integers. The first level L of the representation is made up of the last elements of each chunk, i.e., $L = [\mathcal{S}[b-1], \mathcal{S}[2b-1], \ldots, \mathcal{S}[m-1]]$. This level is encoded with Elias-Fano. The second level is represented by the encoding of the chunks themselves. The main reason for introducing this two-level representation, is that now the elements of the j-th chunk are encoded with a smaller universe, i.e., $L[j] - L[j-1] - 1$. This is, however, a *uniform-partitioning* strategy that may be suboptimal, since we cannot expect clusters of integers be aligned to such boundaries. As the problem of choosing the best possible partition is posed, an algorithm based on dynamic programming is presented in [43] which, in $O(m \log_{1+\epsilon} \frac{1}{\epsilon})$ time, yields a partition whose cost (i.e., the space taken by the encoded sequence) is at most $(1+\epsilon)$ times away from an optimal one, for any $\epsilon \in (0,1)$. To support variable-size partitions, another sequence E is maintained in the first level of the representation, which encodes (again with Elias-Fano) the sizes of the chunks in the second level.

This sequence organization introduces a level of indirection when resolving the queries, because a first search must be spent in the first level of the representation to identify the block in which the searched ID is located. We will return to and stress this point in the experimental Subsection 3.4.

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

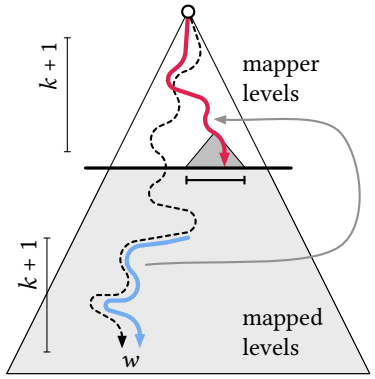
sum is initially 0, therefore the range of A remains the same, i.e., $\langle 0, 2\rangle$. Now the last prefix sum is 2, so we sum 2 to the values in the range of B, yielding $\langle 3, 4, 5\rangle$, and so on. In particular, if we sort the vocabulary IDs in decreasing order of occurrence, we make 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 1), 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, 10, 11, 000 and so on. All codewords are then concatenated one after the other in a bitvector B. Following [24], 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 k of the trie stores three sequences: the gram-ID sequence G_k, the count ranks sequence R_k and the pointer sequence P_k. 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 the Lookup operation is supported, i.e., how to retrieve the frequency count given a gram w_1^n for some $1 \leq n \leq N$. We first perform n vocabulary lookups to map the gram tokens into its constituent IDs. We write these IDs into an array $W[1..n]$. This preliminary query-mapping step takes $O(n)$. Now, the search procedure basically has to locate $W[i]$ in the i-th level of the trie.

If $n = 1$, 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, n is greater than 1, the position 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 $n = 2$, the search terminates by accessing $C_2[r_2]$ where r_2 is the rank $R_2[k_2]$. If n is greater than 2, we fetch



(a)

	k	3-grams	4-grams	5-grams
Europarl	0	2404	2782	2920
	1	213 (×11.28)	480 (×5.79)	646 (×4.52)
	2	2404	48 (×57.95)	101 (×28.91)
YahooV2	0	7350	7197	7417
	1	753 (×9.76)	1461 (×4.93)	1963 (×3.78)
	2	7350	104 (×69.20)	249 (×29.79)
GoogleV2	0	4050	6631	6793
	1	1025 (×3.95)	2192 (×3.03)	2772 (×2.45)
	2	4050	221 (×30.00)	503 (×13.50)

(b)

Fig. 2. The action performed by the context-based identifier remapping strategy. The last word ID w of any sub-path of length $k + 1$, e.g., the blue one, is replaced with the position it takes within its sibling IDs. These sibling IDs are found at the end (the dark gray triangle) of the search of w along the *same* path, e.g., the red one, in the first $k + 1$ levels of the trie. Effect of the context-based remapping on the average gap (ratio between universe and size) of the gram-ID sequences of the datasets used in the experiments, with context length $k = 0, 1, 2$.

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 $n - 1$ times in total, to finally return the count $C_n[r_n]$ of w_1^n.

3.2.2 Context-based Identifier Remapping. In this subsection we describe a novel technique that lowers the space occupancy of the gram-ID sequences that constitute, as we have seen, the main component of the trie data structure.

The idea is to map a word w occurring after the context w_1^k to an integer whose value is bounded by the number of words that *follow* such context, and *not* bounded by the total vocabulary size $|V|$. Specifically, w is mapped to the position it occupies within its siblings, i.e., the words following the gram w_1^k. We call this technique *context-based identifier remapping* because each ID is re-mapped to the position it takes relatively to a context.

Figure 2a shows a representation of the action performed by the remapping strategy: the last word ID w of any sub-path of length $k + 1$ (e.g., the blue one in the figure) is searched along the *same* path occurring in the first $k + 1$ levels of the trie (e.g., the red one in the figure). This can be graphically interpreted as if the blue path were projected to the red path in order to search w along its sibling IDs, that are the ones occurring after the gram w_1^k (the small dark gray triangle in the figure). We stress that this projection is *always possible*, i.e., we are guaranteed to find any sub-path of length $k + 1$ in the first $k + 1$ levels of the trie, because of the sliding-window extraction process described in Section 2. Figure 2a also highlights that using a context of length k will partition the levels of the trie into two categories: the so-called *mapper* levels and the *mapped* levels. The first $k + 1$ levels of trie act, in fact, as a mapper structure whose role is to map any word ID through searches; all the other $n - k - 1$ levels are the ones formed by the remapped IDs.

The salient feature of our strategy is that it takes full advantage of the n-gram model represented by the trie structure itself in that it does *not* need any redundancy to perform the mapping of IDs, because these are mapped by means of searches in the first $k + 1$ levels of the trie. The strategy also allows a great deal of flexibility, in that we can choose the length k of the context. In general,

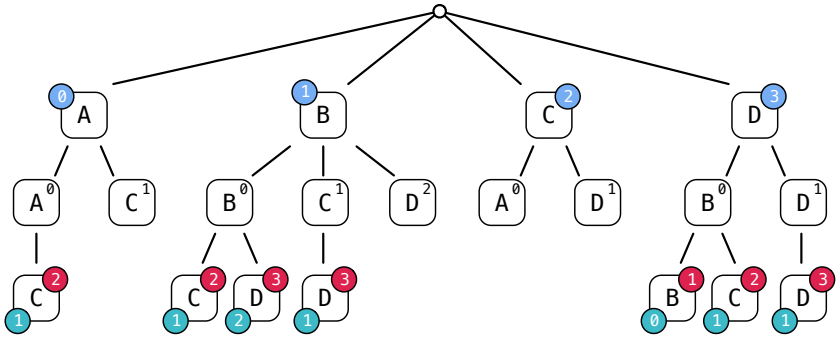


Fig. 3. Example of a trie of order 3, representing the set of grams {A, AA, AAC, AC, B, BB, BBC, BBD, BC, BCD, BD, CA, CD, DB, DBB, DBC, DDD}. Vocabulary IDs are represented in blue while level-3 IDs in red. The green IDs are derived by applying a context-based remapping with context length 1.

with a n-gram dataset of order $N \geq 2$, we can choose between $N-2$ distinct context lengths k, i.e., $1 \leq k \leq N-2$. Clearly, the greater the context length we use, the smaller the remapped IDs will be but the more the searches will take. The choice of 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.

In what follows we motivate *why* the introduced remapping strategy offers a valuable contribution to the overall space reduction of the trie data structure, throughout some didactic and real examples. As we will see in the experimental Subsection 3.4, 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 [39, 43, 45]. In our case the gram-ID sequences are encoded with Elias-Fano: from Subsection **3.2.1**Ãň, equation (1), we know that Elias-Fano spends $\lceil \log \frac{u}{m} \rceil + 2$ bits per integer, thus a number of bits proportional to the average gap u/m between its values. The remapping strategy reduces the universe u of representation, thus lowering the average gap and space of the sequence.

This effect is illustrated by the numbers in Figure 2b that shows how the average gap of the gram-ID sequences of the datasets we used in the experiments (see also Table 1) is affected by the context-based remapping. As uni-grams and bi-grams constitute the mapper levels, these are kept un-mapped: we show the statistic for the mapped levels, i.e., the third, fourth and fifth, of a trie of order 5 built from the n-grams of the datasets. For each dataset we did the experiment for context lengths 0, 1 and 2. As we can see by considering Europarl, our technique with a context of length 1 achieves an average reduction of 7.2 times (up to 11.3 on tri-grams). With a context of length 2, instead, we obtain an average reduction of 43.4 times (up to 58 on 4-grams). Very similar considerations and numbers hold for the YahooV2 dataset as well. The reduction on the GoogleV2 dataset is less dramatic instead, being on average of 3 times with context-length 1 and of 16.75 times with context-length 2.

Example. To better understand how the remapping algorithm works, we consider now a small didactic example. We continue with the example from Subsection **3.2.1** and represented in Figure 3.

The blue 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 greed, 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, 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. The example in Figure 3 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).

Lookup. The described remapping strategy comes with an overhead at query time as the search algorithm described in Subsection **3.2.1** 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 $k \times (N - k - 1)$ additional searches. As an example, by looking at Figure 3, 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 Subsection 3.4.

3.3 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 [25, 44]: 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 publicly available implementation of MPH as described in [3] and available at https://github.com/ot/emphf. This implementation 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 $C_n[r]$.

n	Europarl		YahooV2		GoogleV2	
	n-grams	counts	n-grams	counts	n-grams	counts
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 n-grams	101 428 257	7147	828 223 677	45 285	11 131 242 087	1 073 473
gzip	6.98		6.45		6.20	

Table 1. Number of n-grams and distinct frequency counts for the datasets used in the experiments. We also report the average bytes per gram achieved by gzip as a useful baseline for comparison.

3.4 Experiments

In this subsection, we first present experiments to validate the effectiveness of our compressed data structures in relation to the corresponding query processing speed; then we compare our proposals against several solutions available in the state-of-the-art.

Datasets. We performed our experiments on the following standard datasets.

- Europarl consists in all unpruned n-grams extracted from the English Europarl parallel corpus [32], available at: http://www.statmt.org/europarl.
- YahooV2 [1] is a collection of English n-grams with minimum frequency count equal to 2, extracted from a corpus of 14.6 million documents crawled from more than 12 000 sites during 2006. The dataset is available at: http://webscope.sandbox.yahoo.com/catalog.php?datatype=l.
- GoogleV2 is the latest English version of Web1T [6], whose n-grams have a minimum frequency count of 40. This collection roughly corresponds to 6% of the books ever published. The dataset is available at: http://storage.googleapis.com/books/ngrams/books/datasetsv2.html.

Each dataset comprises all n-grams for $1 \leq n \leq N = 5$ and associated frequency counts. Table 1 shows the basic statistics of the datasets. We choose these datasets in order to test our data structures on different corpora sizes: starting from the left of Table 1 each dataset has roughly 10 times the number of n-grams of the previous one.

Compared indexes. We compare the performance of our data structures against the following software packages that use the approaches introduced in Subsection 3.1.

- BerkeleyLM implements two trie data structures based on sorted arrays and hash tables to represent the nodes of the trie [44]. The code is written in Java and available at: https://github.com/adampauls/berkeleylm.
- Expgram makes use of the LOUDS succinct encoding [29] to implicitly represent the trie structure, while the frequency counts are compressed using VByte encoding [57]. The code is written in C++ and available at: https://github.com/tarowatanabe/expgram.
- KenLM implements a trie with interpolation search and a hashing with linear probing [25]. The code is written in C++ and 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 [59]. The code is written in C++ and available at: https://github.com/s-yata/marisa-trie.

		Europarl		YahooV2		GoogleV2	
		bytes/gram	μsec/query	bytes/gram	μsec/query	bytes/gram	μsec/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 2. Average bytes per gram (bytes/gram) and average Lookup time per query in micro seconds (μsec/query). The bytes/gram cost also includes the space of representation for the pointer sequences.

- RandLM employs Bloom filters with lossy quantization of frequency counts to attain to low memory footprint [52]. The code is written in C++ and available at: https://sourceforge.net/projects/randlm.

Experimental setting and methodology. 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 compiled with gcc 5.4.1 with the highest optimization settings. Template specialization has been preferred over inheritance to avoid the virtual method call overhead, which can be disruptive for the very fine-grained operations we consider. Except for the instructions to count the number of bits set in a word (popcount), and to find the position of the least significant bit (number of trailing zeroes), no special processor feature was used. In particular, we did not add any SIMD (Single Instruction Multiple Data) instruction to our code.

The data structures were saved to disk after construction, and loaded into main memory to be queried. For the scanning of input files we used the posix_madvice system, called with the parameter POSIX_MADV_SEQUENTIAL to instruct the kernel to optimize the sequential access to the mapped memory region. The implementation of our data structures, as well as the utilities to prepare the datasets for indexing and unit tests, is freely available at: https://github.com/jermp/tongrams.

To test the speed of Lookup queries, we use a query set consisting of 5 million n-grams for YahooV2 and GoogleV2 and of 0.5 million for Europarl, drawn at random from the entire datasets. In order to smooth the effect of fluctuations during measurements, we repeat each experiment five times and consider the mean. The shown query results are, therefore, average times. All query algorithms were run on a single core.

3.4.1 Elias-Fano Tries. In this subsection we test the efficiency of our trie data structure. As already done for the description in Subsection 3.2.1, we dedicate one paragraph to the validation of each of the main building components of the trie, as well as to the introduced performance optimizations.

Gram-ID sequences. Table 2 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 Subsection 3.2.1, when the sorted arrays are encoded with Elias-Fano (EF) and partitioned Elias-Fano (PEF) [43]. 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. For GoogleV2

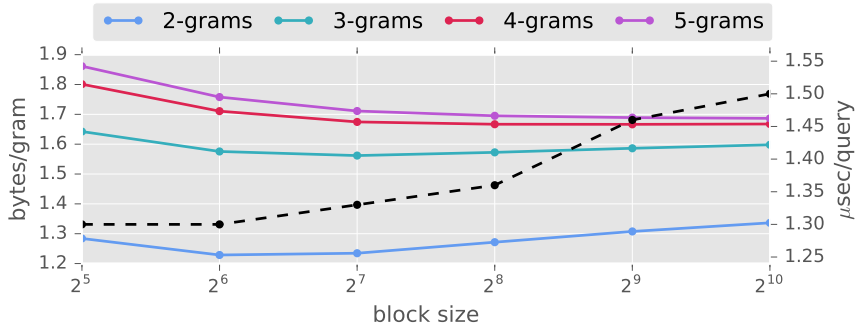


Fig. 4. Bytes per gram (left vertical axis) and μs per query (right vertical axis, black dashed line) by varying block size in PEF uniform on the gram-ID sequences of Europarl.

we use a context of length 1, as the tri-grams alone roughly constitute 66% of the whole the dataset, thus it would make little sense to optimize only the space of 4- and 5-grams that take 27.46% of the dataset.

As expected, partitioning the gram sequences using PEF yields a better space occupancy. Though the paper by Ottaviano and Venturini [43] describes a dynamic programming algorithm that finds the partitioning able of minimizing the space occupancy of a monotone sequence (we refer to this scheme as PEF-OPT in the following), 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 Subsection **3.2.1**, 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 [43] (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.

As the problem of deciding the optimal block size is posed, Figure 4 shows the space/time trade-off obtained by varying the block size on the gram-ID sequences. The plots for YahooV2 and GoogleV2 datasets exhibit the same shape, therefore we report the one for Europarl. The dashed black line illustrates how the average Lookup time varies when *all* the gram-ID sequences are partitioned using the same block size. The figure suggests to use partitions of 64 integers for bi-gram sequences, and of 128 for all other orders, i.e., for $N \geq 3$, given that the space usage remains low without increasing much the query processing speed. With this choice of block sizes, the loss in space with respect to PEF-OPT is small and equal to 3.32% for Europarl; 5.29% for YahooV2 and 7.33% for GoogleV2.

Shrinking the size of blocks speeds up searches over plain Elias-Fano because a successor query has to be resolved over an interval potentially much smaller than a range length. This behavior is clearly highlighted by the shape of the black dashed line of Figure 4. However, excessively reducing the block size may ruin the advantage in space reduction. Therefore it is convenient to use small

	Europarl	YahooV2	GoogleV2
Variable-len. codewords	0.36	0.47	1.46
Prefix sums + EF	0.35 (−1.59%)	0.62 (+32.46%)	1.59 (+9.17%)
Prefix sums + PEF	0.30 (−16.65%)	0.51 (+8.67%)	1.30 (−11.03%)
Variable-len. block-coding	0.76 (+155.63%)	0.79 (+55.86%)	1.32 (+1.44%)
Packed	1.63 (+444.74%)	2.00 (+294.17%)	2.63 (+102.43%)
VByte	3.21 (+975.40%)	3.32 (+554.66%)	—

Table 3. Average bytes per count for different techniques.

block sizes for the most traversed sequences, e.g., the bi-gram sequences, that indeed 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 2, there is *no* practical difference between the query processing speed of EF and PEF: this latter 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).

Context-based identifier remapping. 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.

Regarding the query processing speed, as explained in Subsection 3.2.2, 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, by looking at Table 2, 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 Subsection 3.2.1 that assigns variable-length codewords to the ranks of the counts and keeps track of codewords length using a binary vector (Variable-len. codewords); the other two schemes transform the sequence of count ranks into a non-decreasing sequence by taking its prefix sums and then applies EF or PEF (Prefix sums + EF/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 1). The percentages of Prefix sums + EF/PEF are done with respect to the first row of the table, i.e., Variable-len. codewords.

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

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 Variable-len. codewords.

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-len. block-coding mechanism explained in Subsection 3.1 to compress count ranks, whereas the HASH variant stores bit-packed count ranks, referred to as Packed in the table, using the minimum number of bits necessary for their representation (see Table 1). Expgram, instead, does not store count ranks but directly compress the counts themselves using Variable-Byte encoding (VByte) with an additional binary vector as to be able of randomly accessing the counts sequence. The available RAM of our test machine (193 GBs) was not sufficient to successfully build Expgram on GoogleV2. The same holds for KenLM and Marisa, as we are going to see next. 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 $\approx$10% smaller on GoogleV2 with respect to Variable-len. codewords. However, note that this gap is completely bridged as soon as we adopt the combination Prefix sums + PEF.

Time and space breakdowns. 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 (the R stands for *remapped*) and PEF-Trie. For the PEF-RTrie variant we use PEF for representing the gram-ID sequences; Prefix sums + PEF for the counts on Europarl and GoogleV2 but Variable-len. codewords 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 Variable-len. codewords for the counts, *without* remapping.

The corresponding size breakdowns are shown in Figures 5c and 5d respectively. Pointer sequences take very little space for both data structures (approximately 10.3%), while most of the difference lies, not surprisingly, in the space of the gram-ID sequences (roughly 70% for Europarl and YahooV2; 40% for GoogleV2). The timing breakdowns in Figures 5a and 5b 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. In such case, the two phases of query mapping and search are almost the same, while in the PEF-Trie the search phase dominates.

3.4.2 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.

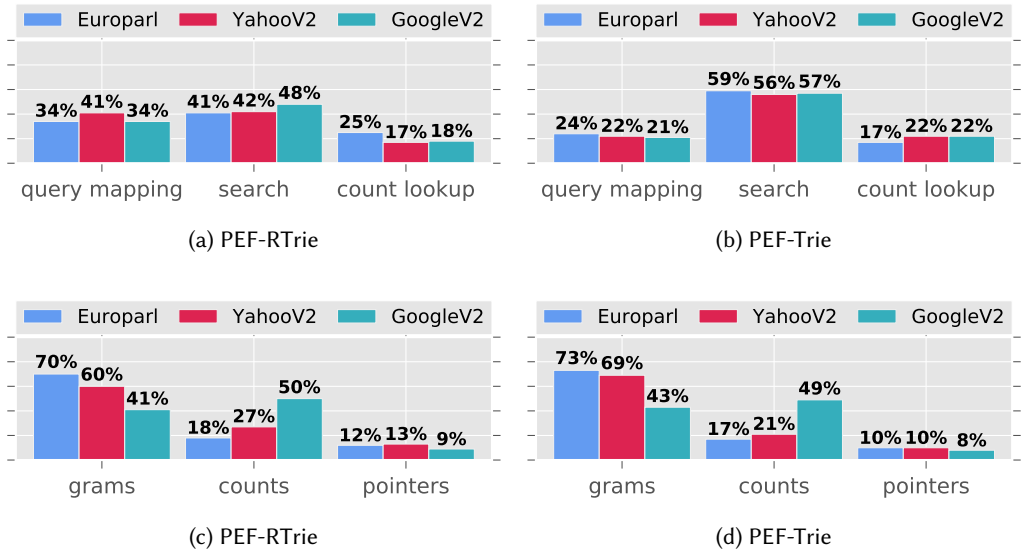


Fig. 5. Trie data structures timing (a-b) and size (c-d) breakdowns in percentage on the tested datasets. For the timing breakdowns we distinguish the three phases of query mapping, ID-search and final count lookup. For the space breakdowns we distinguish, instead, the contribution of gram-ID, count and pointer sequences.

The constant-time access capability of hashing makes gram lookup extremely fast, by requiring on average 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 completely negligible.

3.4.3 Overall Comparison. In this subsection we compare the performance of our selected trie-based solutions, i.e., the PEF-RTrie and PEF-Trie, as well as our minimal perfect hash approach against the competitors introduced at the beginning of this subsection. 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 average 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 probabilities 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 3.4.1. 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 discussion, unless explicitly stated, the numbers cited as percentages 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

	Europarl		YahooV2		GoogleV2	
	bytes/gram	μsec/query	bytes/gram	μsec/query	bytes/gram	μsec/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 (bytes/gram) and average Lookup time per query in micro seconds per query (μsec/query). For our data structures, i.e., PEF-Trie and PEF-RTrie, the bytes/gram cost also includes the space of representation for the pointer sequences.

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 Subsection 3.1. Therefore, while its space occupancy results even larger with respect

	Europarl		YahooV2	
	bytes/gram	μsec/query	bytes/gram	μsec/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 (bytes/gram) and micro seconds per query (μsec/query) using modified Kneser-Ney 5-gram language models built from Europarl and YahooV2 counts.

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 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.

We finally compare the *total* space occupancy, as given by the sum of the space of gram-ID sequences, frequency counts and pointers, of our trie data structures against the gzip baseline reported in Table 1. The total average bytes per represented n-gram for PEF-Trie are 2.17, 2.38 and 2.82 on the three datasets Europarl, YahooV2 and GoogleV2 respectively. Table 1 shows that gzip takes, instead, 6.98, 6.45 and 6.2 bytes per gram. This means that our PEF-Trie is 3.2×, 2.7× and 2.2× smaller than gzip and it does also support efficient search of individual n-grams. Finally, our PEF-RTrie is 4.4×, 3.5×, 2.4× smaller.

Perplexity benchmark. 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 [20] 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 standard query dataset publicly available at http://www.statmt.org/lm-benchmark, that contains 306 688 sentences, for a total of 7 790 011 tokens [7]. We used the utilities of Expgram to build modified Kneser-Ney [9, 10] 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. 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, whereas the PEF-RTrie variant *doubles* the space gains with negligible loss in query processing speed (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, however, the double of the space. On the other hand, the P.3 variant is larger (by 50%) and slower (by 60% on average).

4 FAST ESTIMATION

The problem we tackle in this section of the paper is the one of computing the *Kneser-Ney* probability and backoff penalty for every n-gram, $1 \leq n \leq N$, extracted from a large textual source.

4.1 Preliminaries and Related Work

Since the sorted orders defined over a set of n-grams are central to the description of the algorithms we are going to consider, we now define them. Consider a set of n-grams. The set is put into sorted order by sorting the n-grams on their words, as considered in a *specific* order. If this specific order is $N, N-1, \ldots, 1$, i.e., we sort n-grams from the last word up to the first, then the block is *suffix*-sorted: last word is primary. If the considered order is $N-1, N-2, \ldots, 1, N$, then the block is *context*-sorted: penultimate word is primary.

During the estimation process, we deal with the following assumptions:

(1) the uncompressed n-gram strings with associated satellite values, $1 \leq n \leq N$ do *not* fit in internal memory and we necessarily need to rely on disk usage;

(2) the estimate is performed *without* pruning [26], thus the minimum occurrence count for an n-gram is 1;
(3) the compressed index built over the n-gram strings (e.g., the trie presented in Subsection 3.2) *must* reside in internal memory to allow fast query processing (perplexity and machine translation) [25, 44, 45].

Modified Kneser-Ney smoothing. The *modified* version of Kneser-Ney smoothing [31] was introduced by Chen and Goodman [10] and uses, instead of a single discount value, multiple discounts D_n for all n-grams w_1^n having occurrence count $c(w_1^n)$ equal to 1, 2 and 3. Under the Kneser-Ney model, the conditional probability is computed recursively according to

$$\mathbb{P}(w_n|w_1^{n-1}) = u(w_n|w_1^{n-1}) + b(w_1^{n-1})\mathbb{P}(w_n|w_2^{n-1}) \tag{2}$$

that is, all lower-order probabilities are *interpolated* together, where $u(w_n|w_1^{n-1})$ and $b(w_1^{n-1})$ are, respectively, the normalized probability and context backoff for n-gram w_1^n

$$u(w_n|w_1^{n-1}) = \frac{c(w_1^n) - D_n(c(w_1^n))}{\sum_x c(w_1^{n-1}x)} \tag{3}$$

$$b(w_1^{n-1}) = \frac{\sum_{k=1}^{2} D_n(k)N_k(w_1^{n-1}\bullet) + D_n(3)N_{3+}(w_1^{n-1}\bullet)}{\sum_x c(w_1^{n-1}x)} \tag{4}$$

where $N_k(w_1^{n-1}\bullet) = |\{x : c(w_1^{n-1}x) = k\}|$ represents the number of n-grams having context w_1^{n-1} and count equal to k; $N_{3+}(w_1^{n-1}\bullet) = |\{x : c(w_1^{n-1}x) \geq 3\}|$. Recursion terminates when uni-grams are interpolated with the probability of the unknown word $\langle \mathsf{unk} \rangle$ (uniformly distributed by assumption): $\mathbb{P}(w_n) = u(w_n) + b(\varepsilon) \times \mathbb{P}(\langle \mathsf{unk} \rangle)$, where $\mathbb{P}(\langle \mathsf{unk} \rangle) = \frac{1}{V}$, where we denote by ε the empty string and by V the size of the vocabulary.

Following [9, 10], closed-form discounts $D_n(k)$ are computed as

$$D_n(k) = \begin{cases} 0, & k = 0 \\ k - (k+1)\frac{t_{n,1}t_{n,k+1}}{(t_{n,1}+2t_{n,2})t_{n,k}}, & k = 1, 2, 3 \\ D_n(3), & \text{otherwise} \end{cases} \tag{5}$$

with the smoothing statistic $t_{n,k}$ representing the number of n-grams with count k, i.e., $t_{n,k} = |\{w_1^n : c(w_1^n) = k\}|$ for $k = 1, 2, 3$ and 4.

State-of-the-art. The use of the Map+Reduce paradigm for the problem has been advocated in [5]. As reported in the paper, estimation involved hundreds of machines for a few days. Our work does not consider distributed computations, rather it shows how to let the estimation process scale well on the cores of a single target machine. Nguyen et al. [42] (MSRLM) also considered estimation on a single machine, using a parallel merge sort implementation. However, part of the estimation process is delayed until query-time: while this allows to save some resources during estimation, it also imposes a significant burden during the most efficiency-demanding use of language models, that is query processing [9, 25]. We, instead, prefer to follow the approach of [26] that performs all steps of estimation as to permit the building of an efficient, static, compressed index over the computed model.

The works done by Stolcke [51] (SRILM), Federico et al. [21] (IRSTLM), Pauls and Klein [44] (BerkeleyLM) and Watanabe et al. [57] (Expgram) build Kneser-Ney language models in internal memory without resorting on sophisticated software optimizations and data compression techniques: as a result, are not able to scale to the dimensions we consider in this work.

Heafield, Pouzyrevsky, Clark, and Koehn [26] (KenLM) contributed an estimation algorithm involving three steps of sorting in external memory. Their solution, referred to as the 3-Sort algorithm in the following, significantly outperforms the approaches that we have mentioned above, making it the state-of-art solution to the problem. Indeed, as the authors reported in the paper [26], their algorithm takes, on average, as low as 20% of the CPU and 10% of the RAM of both SRILM and IRSTLM.

Shareghi et al. [49] resort on compressed suffix trees to compute on-the-fly the Kneser-Ney probabilities. The experimental analysis reported in the paper compares against SRILM and shows that such approach is comparable in building time with SRILM indexes but several orders of magnitude (e.g., 1000×) slower to query. In [50] the same authors improved over their previous work [49] by pre-computing some modified counts to speed up the on-the-fly calculation of the Kneser-Ney probabilities. Although pre-computing allows for significant improvement at query time (by up to 2500× faster than the previous solution) at the price of a larger index construction time (70% more time), the resulting language model is still 5× slower than KenLM.

For the reasons discussed in this paragraph, we aim at improving upon the I/O efficiency of the 3-Sort approach of KenLM that we describe in details in Subsection 4.2.

4.2 The 3-Sort algorithm

In this section we review the algorithm devised by Heafield, Pouzyrevsky, Clark, and Koehn [26] since our work aims at improving its I/O efficiency. As already reported, the algorithm is the fastest implementation of modified Kneser-Ney smoothing up to date, as it takes takes 25.4% and 7.7% of, respectively, CPU time and RAM of SRILM; 16.4% and 16.6% of CPU and RAM of IRSTLM [26].

As an overview, the algorithm consists in four streaming passes over the data, that we are going to detail next: (1) counting, (2) adjusting counts, (3) normalization and (4) interpolation. Since all n-grams, $1 < n \leq N$, are sorted between these steps in the next-step desired order, thus *three* times in total, we refer to this approach as the 3-Sort algorithm.

(1) **Counting.** The first step computes the unpruned occurrence counts of all N-grams (with order exactly N) by streaming through the textual corpus. Lower-order n-grams are not counted since raw occurrence counts for N-grams are sufficient to derive smoothing statistics. In particular, N-gram tokens are replaced with 4-byte vocabulary identifiers and uni-gram strings are written to disk as plain text. Their 8-byte Murmur hash is retained in internal memory. The occurrence counts, represented as 8-byte numbers, are accumulated in an open-addressing hash table with linear probing: the counts are written to disk in a suffix-sorted block as records of the form $\langle w_1^N, c(w_1^N)\rangle$ whenever the table reaches a specified amount of internal memory.

(2) **Adjusting.** All blocks sorted in suffix order are merged together in a single block B_N. For $1 \leq n < N$, the *modified count* $c(w_1^n)$ for w_1^n is equal to $|\{x : xw_1^n\}|$, that is, informally, the number of distinct words to the left of w_1^n. These are also called the *left extensions* of w_1^n. By streaming through B_N it is sufficient to compare consecutive entries to decide whether to write the record $\langle w_1^n, c(w_1^n)\rangle$ to a new block B_n or increment the currently computed $c(w_1^n)$. During the same pass, smoothing statistics $t_{n,k}$ are collected and discount coefficients $D_n(k)$ are calculated as in formula (5).

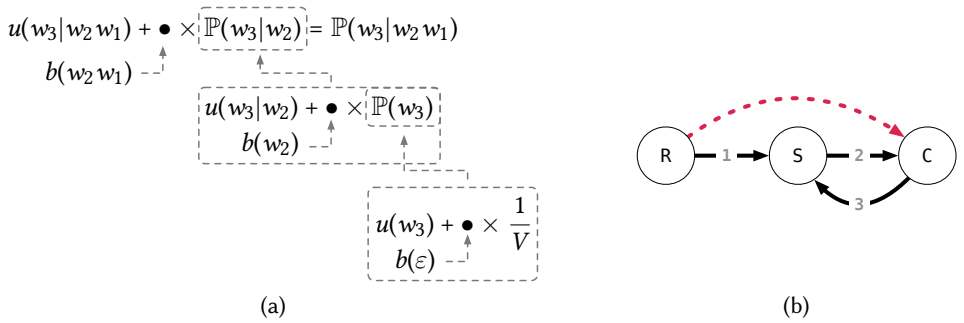


Fig. 6. On the left (a): The Kneser-Ney interpolated probabilities for a 3-gram, calculated in a bottom-up fashion (from 1-gram to 3-gram). On the right (b): Sorting passes performed between random order (R), suffix order (S) and context order (C). Black arrows describe the path followed by the 3-Sort algorithm; the red, dashed, arrow the one followed by the 1-Sort algorithm.

(3) **Normalization.** This step computes normalized probabilities and backoffs according to, respectively, formulas (3) and (4). For such purpose, the blocks B_n, $1 < n \leq N$, produced during the previous step of Adjusting, are sorted in context order such that, for each context w_1^{n-1}, the entries $w_1^{n-1}x$ are consecutive. Again, a streaming pass through each B_n suffices to emit records of the form $\langle w_1^n, u(w_n|w_1^{n-1}), b(w_1^{n-1})\rangle$. The information stored in the record, enclosed in the red rectangles in Figure 6a, is one needed to perform interpolation. The computed backoffs are saved twice on disk, also as bare values without keys, one file per order $1 \leq n < N$ to facilitate the next step of joining.

(4) **Interpolation and joining.** The last streaming step performs interpolation of all orders to compute the final Kneser-Ney probability as in equation (2). The blocks B_n are sorted again in suffix order so that $\mathbb{P}(w_n)$ is computed before it is needed to compute $\mathbb{P}(w_n|w_{n-1})$, which in turn is computed before $\mathbb{P}(w_n|w_{n-2}w_{n-1})$, and so on. Figure 6a offers a pictorial representation of this bottom-up process for a 3-gram. Note that the backoffs for the contexts that are needed for interpolation were saved in-line with the string w_1^n during the previous step. Also note that since normalization streamed through the blocks sorted in context order, the backoffs were saved to disk in suffix order. Therefore, during this step the two quantities $\mathbb{P}(w_n|w_1^{n-1})$ and $b(w_1^n)$ are joined together, for $1 \leq n < N$ (N-grams do not have backoff).

4.3 Improved construction: the 1-Sort algorithm

In this section we introduce our main result that is an estimation algorithm for unpruned, modified, Kneser-Ney language models which substantially improve upon the I/O efficiency of 3-Sort by requiring *only one* sorting in external memory. As apparent from the description in Subsection 4.2, the running time of 3-Sort is dominated by the cost of sorting in external memory, which is paid *three times* in total: (1) from extraction order (unsorted) to suffix order, (2) from suffix order to context order and then (3) from context order to, again, suffix order. This round-trip is the performance bottleneck of 3-Sort and it is graphically represented in Figure 6b. The natural question is whether it is possible to avoid the round-trip and perform the whole estimation by exploiting a single ordering over the N-gram strings. This section of the paper answers positively to such question.

In what follows we detail the steps performed by our algorithm in comparison with 3-Sort and, thus, show how to save two steps of sorting. As a general overview, the algorithm performs three

steps: (1) counting N-grams; (2) computing discount coefficients; (3) normalization, interpolation, joining and index construction in a single pass.

4.3.1 Counting. This first step is performed similarly to the counting step of 3-Sort. In particular, we allocate an in-memory block of bytes able of accommodating the largest number of N-grams as possible, i.e., without taking more space than the amount of RAM specified by the user. Specifically, the block stores records of the form $\langle w_1^N, c(w_1^N)\rangle$, each taking $4N$ bytes for its vocabulary identifiers, plus an 8-byte frequency count. In order to tell whether a N-gram was already seen or not during the scanning of the input, we associate a 4-byte identifier to each distinct N-gram by resorting to an open-addressing hash set.

If a cell of the set is not empty and contains the identifier $k \geq 0$, our probe consists in comparing the extracted N-gram string with the $4N$ bytes stored in the block starting from position $k \times (4N+8)$. If the comparison yields equality, then we increment the corresponding counter, otherwise we advance to the next probe position. If any probed cell is found to be empty, then we write there the next available identifier (equal to the number of distinct seen N-grams) and append a new record to the in-memory block.

As soon as we completely fill the block, we use a parallel thread to sort and write it to disk, thus hash deduplication of the text and I/O happen simultaneously. The key difference of this step with respect to the one of 3-Sort, lies in the fact that we sort the blocks in *context* order instead of suffix order. The reason for this choice will become clear as we proceed in the description of the subsequent steps.

4.3.2 Adjusting. All blocks written to disk by the previous step are merged together to obtain a single block B_N, listing all distinct N-grams sorted in context order. During the process of merging the blocks, we collect the smoothing statistics $t_{n,k}$ in order to use the closed-form estimate of discount coefficients as in formula (5). Because smoothing statistics and, thus, discount coefficients, depend on the modified counts of the n-grams, the key ingredient we develop in this subsection is a linear-time algorithm that computes the modified counts of all n-grams for $1 \leq n < N$ *by scanning the context-sorted block* B_N. In particular, the merged records are accumulated in an in-memory block before writing them to disk. When the block fills up, we run this algorithm over the block. We repeat the process until the input block B_N is processed completely. At the end of the process, we use formula (5) to compute the discount coefficients.

Before illustrating the algorithm for computing the modified counts over the context-sorted block B_N, we first discuss its immediate advantage and then introduce the property of N-grams that the algorithm exploits. Recall that 3-Sort computes the modified counts of the n-grams by scanning B_N as sorted in suffix order (Subsection 4.2). Because the next step of estimation is normalization and it requires context order, computing discount coefficients *directly* over the strings sorted in context order has the benefit of *avoiding to sort from suffix to context*. We are, therefore, eliminating the sorting step 2 of Figure 6b.

Exploiting the completeness of N-gram strings. First of all, observe that since estimation is done without pruning by assumption *and* N-grams are extracted using a window of size N that slides by one word at a time, *the strings in* B_N *cover the input text completely*. This means that all the substrings of length $1 \leq n < N$ of each N-gram occur as substrings of some other N-gram in B_N. Refer to Figure 7 and consider the first 5-gram ABAAC in the context-sorted block at the bottom of the picture. For example, we know that its sub-string BAA must appear at positions 0, 1 and 2 of some other N-grams (the ones of index 6, 0 and 1, respectively). In particular we know that its *prefix* of length 4, i.e., ABAA will be matched at position 1 in some other N-gram (the one of index 1, in this case). We will return to this point later on, in Subsection 4.3.3.

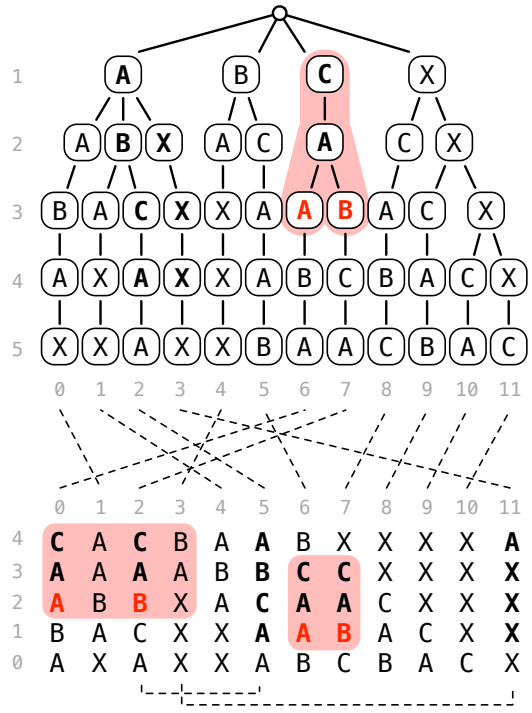


Fig. 7. An example of a 5-gram block sorted in context order and the relation with its reverse trie representation. The bottom level of the trie is obtained by permuting the *first* words of the strings in the context-sorted block, according to the lexicographic position of their *last* words. The left extensions (words in red) of AC must be found in the region highlighted by the red rectangle, which is the run of entries whose context of length 1 is equal to A. These correspond to the children of CA in the trie representation.

This observation means that all lower-order n-grams strings are *implicitly* contained in the single source block B_N. Two important facts are direct consequences of this property.

(1) A sorted scan of the n-grams can be performed by just scanning B_N, without the need of replicating on disk all other n-gram strings, for $1 \leq n < N$, as done by 3-Sort during the Adjusting step.
(2) The number of distinct left extensions, i.e., the distinct words appearing to the left, of a n-gram w_1^n can be computed by scanning the N-grams whose context of length $n-1$ is equal to w_1^{n-1}.

About the fact (2), observe that we could compute the left extensions for a n-gram by directly scanning the N-grams having that n-gram as context of length n. Consider the example in Figure 7. We could scan the entries of index 6 and 7 to compute the distinct left extensions (words in red) of the bi-gram AC, instead of the entries of index 0, 1, 2 and 3. The problem with this approach is that we would not be able to compute the quantity for $(N-1)$-grams because, obviously, a context of length $N-1$ can not be extended to the left. Moreover, consider now the first 5-gram ABAAC. Since interpolation produces the probabilities for all its suffixes, i.e., for C, AC, AAC and BAAC, we need the modified counts for *that* suffixes and *not* for its contexts A, AA, BAA and ABAA that we could have computed with the other approach.

Computing distinct left extensions in context order. We now introduce the linear-time algorithm for computing the distinct left extensions in context order. For ease of explanation, let us consider a N-gram w_1^N as composed by three pieces, in order: P, C_{n-1} and w_N, where C_{n-1} is the context of length $n-1$ and P is the remaining prefix. Our aim is to compute the number of distinct words w_{N-n-1} to the left of the n-gram $C_{n-1}w_N$, because this quantity will be its adjusted count $c(C_{n-1}w_N)$. Since B_N is sorted in context order, the entries PC_{n-1} are consecutive for every context C_{n-1}, but entries $C_{n-1}w_N$ could not (entries $C_{n-1}w_N$ are consecutive in suffix order). However, we know that every left extension must necessarily appear to the left of the context C_{n-1}, and thus we need to only scan the entries having such context.

The quantity $c(C_{n-1}w_N)$ is computed using a direct-address table of size $\Theta(V)$ in which we store, for each distinct w_N, the last seen left word and the number of distinct left words seen so far. As long as context C_{n-1} remains the same during the scan of the block, we look at the table entry corresponding to w_N and consider its last seen left word: if different from w_{N-n-1} then we increment its count by one and update the last seen left word with the current one, otherwise we do nothing. This update step takes $O(1)$ worst-case. We are sure to count correctly the number of left extensions because left words are seen in sorted order. Figure 7 shows an example for the bi-gram AC. In this case its context of length $n-1$ is the uni-gram A. We are sure to find all the distinct left extensions of AC in the range of entries having A as context of length 1, i.e., the ones spanned by the light red rectangle. In particular AC can be extended to the left with words A and B, as depicted in red in the picture, thus $c(\mathsf{AC}) = 2$. Observe that these corresponds to the children of the bi-gram CA in the *reverse* trie representation [25, 45] of the block in the upper part of the picture. The trie stores the strings in suffix order. In other words, the node spelling out the bi-gram CA will store two pointers (one for A and one for B). Again, we will return to this point when we will discuss how to lay out efficiently the reverse trie, in Subsection 4.3.3.

At the end of the scan of all entries with the same context C_{n-1}, it is therefore guaranteed that the table contains the modified counts for all the n-grams $C_{n-1}x$. When the context C_{n-1} changes, then we would need a fast way of zeroing all counts in the table. Instead, we do not re-initialize the table explicitly which would cost $\Theta(V)$ time, but we associate each context an increasing identifier. During the update step we first check the context identifier for the current word w_N: if different from the current one, we set its count in the table to zero and increase by one its range identifier.

Before concluding, there are two corner (simplified) cases that we must mention for completeness: the one of N-grams and the one of 1-grams. The former because N-grams do not have modified counts, rather their counts are equal to the raw frequency counts written in the block B_N; the latter because their context is empty and we do not have to re-initialize their counts in the table when we switch range. This concludes the description of the linear-time algorithm that uses $\Theta(V)$ space to compute the modified counts of all n-grams over a context-sorted block of N-grams.

Collecting smoothing statistics. We are left to describe how we collect the smoothing statistics $t_{n,k}$ for $k \in [1, 4]$ by using the introduced algorithm. For each order n, we maintain an array T of 4 counters, where $T[k-1]$ will store the quantity $|\{w_1^n : c(w_1^n) = k\}|$. A trivial solution scans the table of size $\Theta(V)$ used by the algorithm whenever we change context and update the counters accordingly. This approach is clearly unfeasible in terms of running time. Instead, we can update each $T[k-1]$ in $O(1)$ on-the-fly, during the updating step of the algorithm, as follows. Whenever we increment the occurrence of w_N from k to $k+1$, we just have to check k: if $1 < k \leq 4$ then we increment $T[k-1]$ and decrement $T[k-2]$; if $k = 1$ then we only increment $T[0]$ while if $k > 4$ we only decrement $T[3]$. Whenever we change context, these local counts are first summed to the global ones and then re-initialized.

4.3.3 All in the last step. In the last step of estimation, the algorithm performs normalization, interpolation, joining and indexing at the same time, i.e., *without* requiring a different scan, nor a sorting step, for each phase.

The output of this last step is the compressed, static, trie index that maps the extracted n-gram strings to their Kneser-Ney probabilities and backoffs, described in Subsection 3.2.1. In particular it is the *reverse* trie variant, such as the one depicted in Figure 7, because it optimizes the left-to-right pattern of lookups performed by perplexity scoring (see Subsection 3.4.3) [25, 44, 45].

Normalization and interpolation. In Subsection 4.3.2 we have shown how we can compute the modified counts over the block B_N sorted in context order. Thanks to this tool, we can therefore calculate pseudo probabilities and backoff values using formulas (3) and (4) respectively, by just scanning B_N and using a direct-address table of size $\Theta(V)$ to read the modified counts.

In order to interpolate all different orders, we produce pseudo probabilities and backoffs for all n-grams sharing the same context, starting from order 1 up to N. This guarantees that as soon as we compute $u(w_N|w_{N-n-1}^{N-1})$, we can directly interpolate it with $\mathbb{P}(w_N|w_{N-n-2}^{N-1})$ that has been already computed. Normalization and interpolation are, therefore, carried on as explained in Subsection 4.2, but without requiring two separate sorting passes over the N-gram strings. Another crucial difference is that the two phases are performed during the same scan of only one block, i.e., B_N, and we do not need to jointly iterate through N distinct files, one for each value of n, as done by 3-Sort.

The net result is that we *avoid to sort from context to suffix* in order to perform interpolation, thus eliminating the sorting step 3 of Figure 6b. Summing up, given that we have formerly shown how to save the sorting from suffix to context too (Subsection 4.3.2), we have completely eliminated the round-trip of 3-Sort mentioned at the beginning of Subsection 4.3.

Joining and indexing. We now show how to join probabilities and backoffs values, and how to build the reverse trie during the same pass. For this purpose, we exploit the property already mentioned in Subsection 4.3.2, that is: *every prefix of length $N-1$ must be matched at position 1 in some other N-gram.* This property gives us two important guarantees.

(1) The first $N-1$ levels of the reverse trie can be built by streaming through the N-grams in context order.
(2) Backoffs are emitted in suffix order.

In the following we exploit these two guarantees to build the trie and perform joining, respectively.

By looking at Figure 7, we can graphically visualize these two points. Regarding point (1), we see that the first 4 levels of the trie are indeed the contexts of length 4 of the N-gram strings in the context-sorted block. For example, the prefix of length 4 of **ACBA**C is found in the N-gram of index 5; the one of **XXXA**B in the N-gram of index 11 instead. Notice that we *always* find the match at position 1, thus the first 4 levels of the trie store such prefixes. Regarding point (2), consider the first N-gram ABAAC. Since interpolation produces the probabilities for all the suffixes, i.e., for C, AC, AAC and BAAC, we compute the backoffs for their contexts, i.e., for $b(\varepsilon)$, b(A), b(AA) and b(BAA), which appear in sorted order in the block. Refer to Figure 6a too, for a graphical example. Backoffs are, therefore, computed in suffix order and can be written directly in the corresponding trie nodes.

Exploiting the relation between context and suffix order. Now that we have discussed how to handle the first $N-1$ levels of the trie and perform joining, we are left to consider its bottom level and how we write the probabilities in the nodes of the trie. In fact, notice that: regarding point

n	1BillionWord	Wikipedia17	ClueWeb09
1	2 438 616	5 681 625	4 291 588
2	43 179 094	141 639 447	236 626 867
3	203 793 974	587 261 939	977 038 965
4	427 172 514	1 115 647 651	1 710 815 581
5	588 390 914	1 463 820 688	2 129 634 982
total n-grams	1 264 975 112	3 314 051 350	5 058 407 983

Table 6. Number of n-grams for the datasets used in the experiments.

(1), since a context of length $N-1$ does not extend to the left, we can not build the bottom level directly; regarding point (2), interpolation produces the probabilities for the suffixes but we rather would need the ones for the contexts, in order to write them in the trie, as already done for the backoffs. We clarify this latter point by continuing the above example for ABAAC. We interpolate its constituent n-grams: C, AC, AAC, ecc, but we would actually need the probabilities for A, AA, BAA, ecc, in order to write them in the suffix trie as already done for the backoffs.

To address these two problems, we exploit the following property, that established the relation between context and suffix order. *A context-sorted block can be sorted efficiently in suffix order by considering the last word only*, because the prefixes of length $N-1$ are already sorted by definition. This property implies that: *the bottom level of the trie can be built by placing the first words of the strings of the context-sorted block in the lexicographic positions of their last words*. Thanks to this property, although the algorithm operates over the strings sorted in context order, it is still able to efficiently lay out the strings in suffix order.

The relation is depicted in Figure 7 by the dashed lines. For example, consider the first N-gram ABAAC. We know that such string will terminate with A (first word) in the bottom level of the trie. The position at which we have to place this first word is the lexicographic position of the last word, i.e., the C. Since the lexicographic position of the C is 6 within all the last words of the N-grams, A is placed in position 6.

In order to place word identifiers and probabilities in correct position, we use a count-indexing technique, instead of storing the permutation explicitly that would cost $m_N \log m_N$ bits of space, where m_N is the total number of N-grams. For each vocabulary word, we maintain the number of times it appears as last word of a N-gram in a direct-address table of size V. Prefix-summing such counts (shifted by one position to the right) gives us in $O(1)$, for each distinct word identifier i, the position in the array, that represents the bottom level of the trie, at which we have to write the first occurrence of i. Given such position, we write the integer i in $O(1)$ and increment the position in the table by one. This is the same procedure as used by counting sort and requires only V integer counters. In the example of Figure 7, we obtain the initial positions $[0, 4, 6, 8]$ for, respectively, A, B, C and X. Thus we know that the first occurrence of, say, B, will be placed in the bottom level at position 4; the second occurrence at position 5. The same technique is also used to place the final interpolated probabilities in the correct trie nodes for all other orders $1 < n < N$.

Finally, we also have to write the pointers for each node of the trie. As already observed in Subsection 4.3.2, a pointer represents the number of successors of a given n-gram, thus these are *the same* as the modified counts. In conclusion, the computation of the pointers requires no extra effort.

4.4 Experiments

The experiments we now show have the purpose of: first characterizing the running time of our solution, i.e., the 1-Sort algorithm; introducing optimizations and finally considering the comparison against the 3-Sort approach.

Datasets. We performed our experiments using the following textual collections in the English language.

- 1BillionWordthat is the concatenation of all the news files contained in the training directory of the dataset described in [7] and publicly available at: http://www.statmt.org/lm-benchmark;
- Wikipedia17that is the latest Wikipedia dump, collected from October to December 2017 and publicly available at: https://dumps.wikimedia.org/enwiki/latest;
- ClueWeb09 that is a sampling of 5 million pages drawn from the ClueWeb 2009 TREC Category B test collection, consisting of English web pages crawled between January and February 2009, available at: http://www.lemurproject.org/clueweb09.

From each dataset we removed all non-ASCII characters and markup tags. We use the (standard) value of $N = 5$ in every experiment. The datasets are of increasing size, reported as the number of n-grams in Table 6: this will be useful to show the behavior of our solution by varying the size of the input.

Experimental setting and methodology. All experiments have been performed on a machine with 4 Intel i7-7700 cores clocked at 3.6GHz, with 64GB of RAM DDR3, running Linux 4.4.0, 64 bits. RAM is clocked at 2.133GHz. The machine is equipped with a mechanical disk of 3TB WDC WD30EFRX-68E, with standard page size of 4KB.

We implemented the 1-Sort algorithm in standard C++11: the source code will be released upon publication of the paper. As our competitor, we use the C++ implementation of 3-Sort as provided by the authors of [26] and available at http://kheafield.com/code/kenlm. We refer to this implementation as KenLM, which is the lead toolkit for language modeling [25]. As a matter of fact, KenLM provides the fastest estimation algorithm, significantly outperforming the previous approaches [26]: it takes on average 20% of the CPU and 10% of the RAM of SRILM and IRSTLM. This is also confirmed by other recent experiments, showing KenLM to be up to 10× faster to build for the typical values of $n \leq 5$ than approaches based on compressed suffix trees [50].

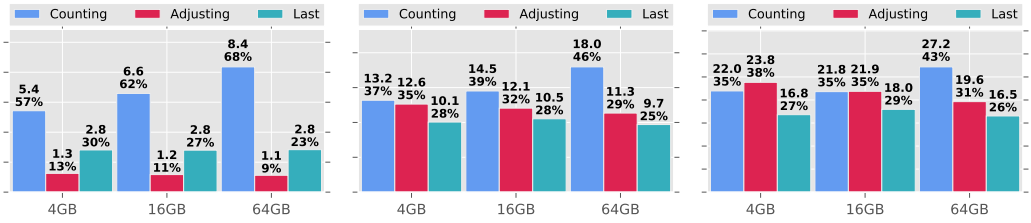
All code compiled with gcc 5.4.0, using the highest optimization setting.

4.4.1 Preliminary analysis.

As a first set of experiments we show the running time of our algorithm by varying the amount of internal memory and by expecting the CPU and I/O activity.

Varying the amount of internal memory. As a first experiment, we show the running time of our algorithm at each step of estimation, by varying the allowed amount of internal memory. We show the results using three values: 4GB, 16GB and the maximum available RAM, 64GB. This experiment aims at showing which steps are the most expensive and fix the amount of internal memory that we will use for the subsequent analysis. The plots in Figure 8 illustrate the results. Above each bar, we report two numbers: the first indicating the number of minutes spent during the step, the second indicating the percentage with respect to the total running time of the algorithm. This grand total measures the time of the whole estimation process, i.e., the time it takes from the scanning of the input text to the flushing on disk of the compressed index built over the extracted strings. Some considerations are in order.

First of all, we can observe that, not surprisingly, the size of the language model has a significant impact not only on the total running time but also on which step becomes the most expensive.



(a) 1BillionWord (b) Wikipedia17 (c) ClueWeb09

Fig. 8. Time spent at each step of estimation by using different amounts of internal memory.

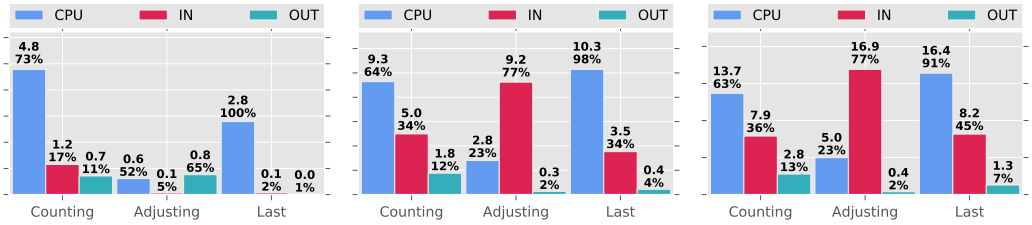
In fact, while on the 1BillionWord dataset the Counting and Last step contribute for more than 80% of the total running time and the Adjusting step has a quite low impact, the trend changes significantly on the larger datasets. In fact, on Wikipedia17 and ClueWeb09 the total running time is almost evenly distributed across the three steps. Notice, in particular, that the time for Adjusting rises significantly. This is due to the number of N-gram blocks written to disk during the Counting step and that are merged together during the Adjusting step. On the smaller dataset 1BillionWord, we have relatively few blocks to merge, thus Adjusting is performed quickly. Clearly, using more internal memory helps in lowering the number of blocks to merge and, thus, reducing the time for Adjusting.

We also observe that the step of Counting and the Last one do not vary much when more memory is available. Concerning the Counting step, more memory is not useful to lower the running time because using larger hash sets also means sorting larger blocks of N-grams. Indeed, observe that the total running time of Counting (slightly) increases by increasing the amount of memory. However, as we have discussed above, using more memory for sorting implies fewer of blocks to merge, thus internal memory size has an impact only on the Adjusting step. For the open-address hash set implementation that we use in the Counting step, we experimented with linear probing, quadratic probing and double hashing. No significant difference among the three strategies was observed, thus we prefer linear probing for its better locality of accesses. Concerning the last step, we need to scan the merged N-gram file once. We use a standard buffered-scan approach using blocks of 64MB by default. Using larger blocks does not impact the running time.

Since similar observations also hold true for KenLM, we choose the middle value of 16GB for all datasets as the quantity of memory we use for all the following experiments.

Inspecting CPU and I/O activity. It is now interesting to quantify the impact that CPU and I/O operations have on the total running time of each step. Under a different perspective, this analysis is also useful to understand and *how* disk usage is impacted by the size of the language model. The plots in Figure 9 illustrate such impact, i.e., the time spent by CPU and I/O at each step by using the amount of RAM that we fixed before (16GB).

Dealing with external memory poses the challenge of trying to avoid CPU idle time by overlapping CPU computation with I/O activity. For such reason, we use asynchronous threads to handle input/output operations, so that while the CPU is performing internal processing, data is read or written to disk simultaneously [17]. This is a feature of particular importance for on-disk programs such as the ones we are considering, given the huge discrepancy in speed of modern processors and (mechanical) disks. Clearly, a perfect overlapping between CPU and I/O time would mean to only pay the maximum of the two. Consequently, the sum of three percentages for CPU, IN and



(a) 1BillionWord (b) Wikipedia17 (c) ClueWeb09

Fig. 9. Time spent by CPU computation and I/O activity at each step of estimation.

OUT time for a given step in Figure 9, may exceed 100% because these are handled by different threads. Let us now consider each step in order.

During the Counting step, while the reader thread is scanning the input and probing the hash set, the writer thread is asynchronously sorting the previous N-gram block and flushing it to disk. While sorting is strictly CPU-bound because it is performed in memory, the scanning of the input text imposes some CPU idle time as apparent for the plots of the larger datasets Wikipedia17 and ClueWeb09. However, probing the hash set and sorting contribute to most of the time spent during the Counting step. In fact, the plots report that the sum of CPU and IN percentages yields almost the whole running time of Counting, whereas the OUT time is completely overlapped with CPU processing.

The total running time of the Adjusting step is, instead, dominated by the cost of reading the blocks from the disk. This is no surprise given that multiple input streams are contending the disk for input operations, thus incurring in more disk seeks [56]. As a result, on the larger datasets Wikipedia17 and ClueWeb09 we can see the IN time taking 77% of the total: this causes the CPU utilization to drop down to roughly 23%, by experiencing idle time. Indeed, the time taken by the algorithm described in Subsection 4.3.2 is negligible compared to the overall running time of the step and contributes to a small percentage of the CPU: it is just 0.42, 1.2 and 1.8 minutes on 1BillionWord, Wikipedia17 and ClueWeb09 respectively. The remaining part of the CPU is spent by iterating through the fetched block of N-grams and comparing records during the merging process.

During the Last step, while the reader thread is loading a block from disk, the CPU is processing the previous block. Therefore, we have a good overlap between CPU and reading time from disk. This is possible because disk reads are issued to a single source, i.e., the merged N-gram file, thus we avoid the disk seeks experienced during the Adjusting step. As a result, all time is spent by the CPU.

4.4.2 Optimizing our solution. In this subsection we devise and quantify the impact of one performance optimization for each step of estimation.

Counting: implementing a parallel radix sort. In order to lower the total running time of the Counting step, it is important to guarantee a good overlap between input scanning and sorting in order to only pay the maximum of the two latencies and not the sum of the two. For this reason, we use LSD (least-significant-digit) *radix sort* [14], instead of the general-purpose std::sort. This sorting algorithm is the right choice in our setting because each N-gram is a (short) string of exactly N word identifiers, thus N passes of counting sort, i.e., one for each word index j, $j = N-1, 0, 1, \ldots, N-2$,

(a) Wikipedia17

	CPU	IN	total	bytes/gram
Uncompressed	2.81	9.24	12.05	28.00
FC bit-aligned	5.77 (0.49×)	0.10 (97.08×)	5.86 (2.06×)	9.00 (3.11×)
FC byte-aligned	3.94 (0.71×)	1.22 (7.60×)	5.03 (2.40×)	11.00 (2.54×)

(b) ClueWeb09

	CPU	IN	total	bytes/gram
Uncompressed	4.98	16.91	21.89	28.00
FC bit-aligned	9.29 (0.54×)	5.25 (3.22×)	14.55 (1.50×)	9.75 (2.87×)
FC byte-aligned	7.61 (0.65×)	4.23 (4.00×)	11.55 (1.89×)	11.65 (2.40×)

Table 7. The effect of compressing blocks during the Adjusting step, on Wikipedia17 and ClueWeb09 datasets. The table reports: the time in minutes spent by computation (CPU), reading from disk (IN) and globally (total) and the average bytes per gram achieved by the different implementations.

are sufficient (and necessary) to sort a block in context order. The time complexity to sort a block of m N-grams is $\Theta((m + V) \times N)$, which is $\Theta(m \times N)$ given that $V = O(m)$.

Moreover, each step of counting sort on column index j is implemented in parallel, as follows. Let K be the number of threads used for sorting. We allocate a table $C[K + 1][V]$ of counters, where $C[t + 1]$ will store the number of occurrences of each word identifier in the partition of $\Theta(\frac{m}{K})$ records assigned to thread t. Then each thread t, for $0 \leq t < K$, runs in parallel and increments by one the entry $C[t + 1][i]$ whenever it encounters the word identifier i. Now, prefix-summing the counters by a *column-major scan* of C transforms each entry $C[t][i]$ into the (sorted) position in the output block at which thread t has to write the record having i as its j-th word identifier.

Thanks to this strategy and by using all the available cores on our test machine ($K = 4$), the time for the Counting step improves substantially because sorting N-gram blocks becomes completely overlapped with input scanning and probing of the hash set: from 6.6 minutes we pass to 3.5 minutes on 1BillionWord (1.88×); from 14.5 to 10 minutes on Wikipedia17 (1.45×); from 21.8 to 15.8 on ClueWeb09 (1.38×). In our experiments, we found out that this parallel implementation of radix sort is also roughly 1.8× faster on average than gnu::parallel_sort. As an example, to sort a N-gram block of 8GB, the gnu::parallel_sort takes 30 seconds while our parallel LSD radix sort takes 16.4 seconds.

Adjusting: compressing N-gram blocks. The high cost of reading the N-gram files from disk during the Adjusting step suggests that all efforts spent in enhancing its running time should be devoted in reducing the loading time from disk, because lowering the CPU cost will result in a negligible improvement. For this reason we compress the N-gram blocks created during the Counting step. Compressing the blocks has the potential of reducing the time spent in reading from disk because more (compressed) N-grams are transferred from disk to memory during an input operation.

What we need is a compressed stream representation that supports fast sequential decoding. We adapt a *front-coding* [58] representation of a N-gram block, as follows. We fix a window size in bytes (64MB by default, in our implementation) and compress as many records $\langle w_1^N, c(w_1^N)\rangle$ as possible,

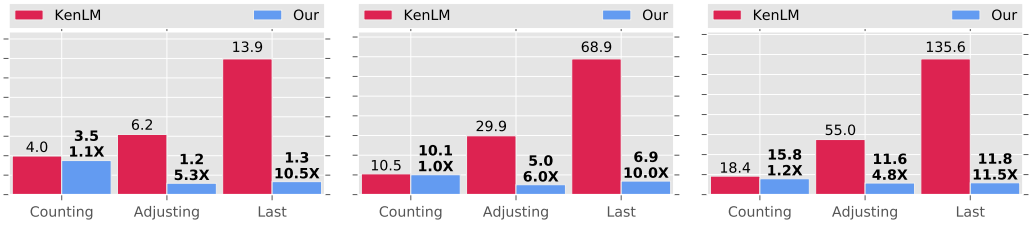
i.e., that can be contained in the window. When encoding/decoding a window, we maintain the following invariant: a record is either written uncompressed, or compressed with respect to the previous one. In particular, a record is encoded as a pair $\langle \ell, s \rangle$, where ℓ is the number of words identifier we have to copy from the previous record (in context order) and s is the remaining part of the string. The first record of each window is written uncompressed.

We can use the minimum number of bits or bytes to represent each word identifier and frequency count. We refer to such strategies as, respectively, FC bit-aligned and FC byte-aligned, whose impact is evaluated in Table 7. As we can see from the data reported in the table, the bit-aligned version offers a 3× space reduction: from 28 bytes per record of the uncompressed version, we pass to an average of 9 bytes per record on Wikipedia17 and to 9.75 bytes per record on ClueWeb09. As a net result, the Adjusting step on Wikipedia17 and ClueWeb09 runs 2× and 1.5× faster. Indeed, we can observe that the input time decreases significantly: it is almost 100× smaller on Wikipedia17 and more than 3× smaller on ClueWeb09. However, notice that the CPU time rises as well, roughly 2×, due to decoding from a compressed stream: we trade CPU time for less reading from disk. The byte-aligned version, FC byte-aligned, avoids the many bit-level instructions to decode a record. Not surprisingly, we can see that this strategy is actually faster than the bit-aligned version by 25% on average, while only allowing a slight worse compression (2.5× on average compared to 3×). In conclusion, compressing the N-gram blocks with byte-aligned front-coding yields an improvement of 2.4× and 1.9× on Wikipedia17 and ClueWeb09 datasets, respectively. Therefore, for the rest of the experiments we use the FC byte-aligned representation of the blocks. On the smaller dataset 1BillionWord, however, compressing the blocks does not yield an appreciable improvement since input time from disk takes a negligible fraction of the total running time of the step (see Figure 9a).

Last: processing N-gram blocks in parallel. As discussed in Subsection 4.4.1, the last step of estimation is CPU-bound. Thus, we can use multi-threading to speed up the execution of the step. If K is the chosen parallelism degree, we use 1 reader thread to load the next $K - 1$ blocks from the merged N-gram file and $K - 1$ worker threads to process these blocks in parallel. While each worker thread independently executes the algorithm described in Subsection 4.3.3 on its own block, the reader thread asynchronously loads the next $K - 1$ blocks in memory. The main challenge of this approach lies in computing the partition of each level of the trie that has to written by a worker thread. For such purpose, we adopted a similar partitioning strategy to the one described in the previous subsection: in a first phase, each worker thread computes the number of distinct n-grams in its own block; in a second phase these counts are combined to obtain the offsets of the global partition of the trie. Although the first phase is performed in parallel, it has an impact on the achieved scalability.

On our test machine, we have $K = 4$, thus we use 3 worker threads and 1 reader thread. On 1BillionWord we reduce the running time from 2.8 to 1.33 minutes (2.1×); on Wikipedia17 from 10.53 to 6.85 minutes (1.54×); on ClueWeb09 from 18 to 11.8 minutes (1.52×).

4.4.3 Overall comparison. In this subsection we compare the performance of our solution, featuring all the optimizations that we have discussed before, against the state-of-the-art implementation of 3-Sort, that is KenLM. The first comparison plots we show are illustrated in Figure 10. The plots strictly confirm the thesis of this paper. The round-trip performed by 3-Sort, i.e., the sorting from suffix to context and then back from context to suffix (see Figure 6b), results in a severe penalty on the total running time of the estimation process: our improved 1-Sort algorithm exploits the properties of the extracted N-gram strings in order to *completely avoid* the round-trip. Overall, this makes our approach run 4×, 4.9× and 5.3× faster than KenLM, respectively on 1BillionWord, Wikipedia17 and ClueWeb09. Let us now discuss each step separately.



(a) 1BillionWord (b) Wikipedia17 (c) ClueWeb09

Fig. 10. Comparison between KenLM and our algorithm at each step of estimation.

As already commented in Subsection 4.3.1, the first step of Counting is performed similarly by the two algorithms and this is the reason why the corresponding running times are comparable. In fact, both algorithms use a separate thread to sort the previously-formed block in parallel and flushing it to disk while input scanning takes place at the same time. Both implementations also use open-addressing with linear probing. The key difference lies in the fact that we sort in context order, whereas KenLM adopts suffix order. Another crucial difference is that our solution compresses the blocks to reduce the merging time in the next step, which KenLM does not do.

During the Adjusting step, our approach computes the modified counts in context order as described in Subsection 4.3.2 on every output block formed during the merging process. KenLM does the same but over suffix-sorted blocks, thus it has to write back to disk *each n-gram, for* $1 < n \leq N$, along with its own modified count, in context order. Since our approach re-computes the modified counts during the process of normalization itself, we only need to handle the N-grams and merge their blocks. Instead, KenLM has to finally merge the blocks or all n-grams written to disk. Although it exploits multiple threads (one for each order), the additional writes to disk and sorting operations cause KenLM be on average 5.3× slower during this step than our approach.

During the last step, normalization, interpolation and indexing are performed (Subsection 4.3.3). Again, we can observe an average speed-up of 10.6×. Since our algorithm builds a compressed reverse trie index during the same step, we also sum to the time of KenLM the time it takes to build the same data structure, because the current implementation does not build the index during the same pass (although the possibility is advocated in the paper [26]). To ensure fairness, the indexing time for KenLM is measured by excluding the time to write and parse the intermediate (ARPA) file on disk: it is anyway a significant amount of the total running time of KenLM, equal to 7, 31 and 61 minutes for, respectively, 1BillionWord, Wikipedia17 and ClueWeb09. Apart from indexing, the rest of the time is spent in sorting again from context to suffix order, as needed for interpolation. Both normalization and interpolation phases of KenLM exploits multi-threading, by using separate threads for each value of n. In particular, two threads are used to compute the denominators and numerators of the quantities in formulas (3) and (4). Again, recall that we only need to tackle N-grams because we consider the other n-gram strings implicitly, thus our implementation uses multiple threads for in-memory processing and a thread to asynchronously feed the CPU with input.

Output volume. Another way of visualizing the comparison between our solution and KenLM is to measure the number of bytes read/written per second from/to the disk by the two algorithms. Figure 11 shows the number of GB written per second on disk for the Wikipedia17 dataset. The shapes for the other datasets are similar, as well as the shape for the input volumes. We collect

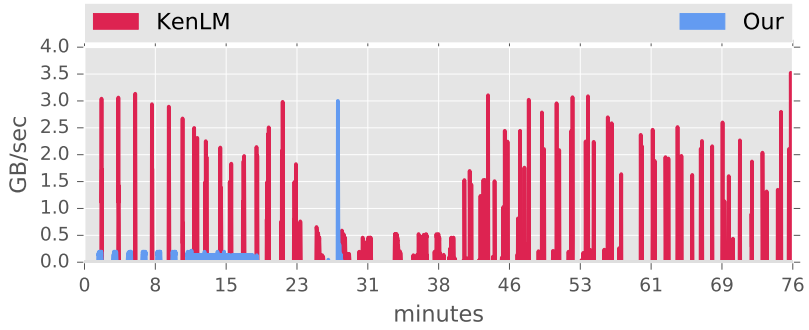


Fig. 11. Gigabytes per second written on disk by KenLM and our algorithm when processing the Wikipedia17 dataset.

the statistic using the Linux utility pidstat with time interval of 1 second and matching the name of the executed task. The volume for our construction also includes the one spent when flushing the compressed index to disk, whereas the volume for KenLM does not because the current implementation builds the index with a separate program.

The plots strictly matches the results shown in Figure 10, where an average improvement of 4.5× in running time is exhibited. Not surprisingly the improvement in running time is directly proportional to the quantity of data written to disk. In fact, the area below the curve of our algorithm is ≈5× less on Wikipedia17 (63.6GB vs. 310.7GB) than the one of KenLM. We obtained similar results on the other datasets: ≈6× less (20.4GB vs. 124.5GB) on 1BillionWord and ≈5× less on ClueWeb09 (88.1GB vs. 433.7GB).

5 CONCLUSIONS

In this paper we studied in depth the two correlated problems lying at the core of language models applications, i.e., indexing n-grams and estimating language models by computing probability and backoff values for each n-gram extracted form large textual collections. We focused on solving these two problems *efficiently*.

Concerning the problem of indexing, we 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.

Concerning the problem of estimation, we have presented a novel algorithm that estimates unpruned, modified, Kneser-Ney language models in external memory. Our approach sorts the extracted N-gram strings once, in *context* order, and outputs a compressed trie data structure, indexing the strings in *suffix* order. Our construction is, on average, 4.5× faster than the fastest state-of-the-art algorithm. The improved performance of the algorithm derives from the exploitation of the properties on the extracted N-gram strings that are neglected by competitive approaches. Thanks to such properties, it is possible to operate over the context-sorted strings and, yet: (1) compute the Kneser-Ney modified counts in linear time and only taking space proportional to the vocabulary; (2) efficiently lay out the popular reverse trie data structure.

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 *International World Wide Web Conference (WWW)*. 107–116.

[3] Djamal Belazzougui, Paolo Boldi, Giuseppe Ottaviano, Rossano Venturini, and Sebastiano Vigna. 2014. Cache-oblivious peeling of random hypergraphs. In *International Data Compression Conference (DCC)*. 352–361.

[4] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM (CACM)*. 422–426.

[5] Thorsten Brants, Ashok C Popat, Peng Xu, Franz J Och, and Jeffrey Dean. 2007. Large language models in machine translation. In *International Conference Empirical Methods on Natural Language Processing (EMNLP)*. 858–867.

[6] 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*.

[7] 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.

[8] Ciprian Chelba and Johan Schalkwyk. 2013. Empirical Exploration of Language Modeling for the google.com Query Stream as Applied to Mobile Voice Search. In *Mobile Speech and Advanced Natural Language Solutions (MSANLS)*. 197–229.

[9] Stanley Chen and Joshua Goodman. 1999. An empirical study of smoothing techniques for language modeling. In *Computer Speech and Language (CSL)*, Vol. 13. 359–394.

[10] Stanley F. Chen and Joshua Goodman. 1996. An empirical study of smoothing techniques for language modeling. In *Association for Computational Linguistics (ACL)*. 310–318.

[11] Wenlin Chen, David Grangier, and Michael Auli. 2015. Strategies for Training Large Vocabulary Neural Language Models. In *Preprint arXiv:1512.04906*.

[12] David Clark. 1996. *Compact Pat Trees*. Ph.D. Dissertation. University of Waterloo.

[13] David R. Clark and J. Ian Munro. 1996. Efficient suffix trees on secondary storage. In *Symposium on Discrete Algorithms (SODA)*. 383–391.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3-rd Edition)*. MIT Press.

[15] Bruce Croft, Donald Metzler, and Trevor Strohman. 2009. *Search Engines: Information Retrieval in Practice* (1st ed.). Addison-Wesley Publishing Company.

[16] Erik D. Demaine, Thouis Jones, and Mihai Pătraşcu. 2004. Interpolation search for non-independent data. In *Symposium on Discrete Algorithms (SODA)*. 529–530.

[17] Roman Dementiev, Lutz Kettner, and Peter Sanders. 2008. STXXL: standard template library for XXL data sets. In *Software, Practice and Experience (SPE)*, Vol. 38. 589–637.

[18] Peter Elias. 1974. Efficient Storage and Retrieval by Content and Address of Static Files. In *Journal of the ACM (JACM)*. 246–260.

[19] Robert Mario Fano. 1971. On the number of bits required to implement an associative memory. In *Memorandum 61, Computer Structures Group, MIT*.

[20] Marcello Federico and Nicola Bertoldi. 2006. How many bits are needed to store probabilities for phrase-based translation?. In *Workshop on Statistical Machine Translation (WMT)*. 94–101.

[21] Marcello Federico, Nicola Bertoldi, and Mauro Cettolo. 2008. IRSTLM: an open source toolkit for handling large scale language models. In *INTERSPEECH*. 1618–1621.

[22] Agner Fog. 2014. *Optimizing software in C++: an optimization guide from Windows, Linux and Mac platforms*. Technical University of Denmark.

[23] Edward Fredkin. 1960. Trie memory. In *Communications of the ACM (CACM)*. 490–499.

[24] Kimmo Fredriksson and Fedor Nikitin. 2007. Simple compression code supporting random access and fast string matching. In *Workshop on Experimental Algorithms (WEA)*. 203–216.

[25] Kenneth Heafield. 2011. KenLM: Faster and smaller language model queries. In *Workshop on Statistical Machine Translation (WMT)*. 187–197.

[26] Kenneth Heafield, Ivan Pouzyrevsky, Jonathan H Clark, and Philipp Koehn. 2013. Scalable Modified Kneser-Ney Language Model Estimation. In *Association for Computational Linguistics (ACL)*. 690–696.

[27] Steffen Heinz, Justin Zobel, and Hugh E. Williams. 2002. Burst tries: a fast, efficient data structure for string keys. In *Transactions on Information Systems (TOIS)*. 192–223.

[28] Samuel Huston, Alistair Moffat, and W. Bruce Croft. 2011. Efficient indexing of repeated n-grams. In *International Conference on Web Search and Data Mining (WSDM)*. 127–136.

[29] Guy Jacobson. 1989. Space-efficient Static Trees and Graphs. In *Foundations of Computer Science (FOCS)*. 549–554.

[30] Dan Jurafsky and James H. Martin. 2014. *Speech and language processing*. Pearson.

[31] Reinhard Kneser and Hermann Ney. 1995. Improved backing-off for m-gram language modeling. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Vol. 1. 181–184.
[32] Philipp Koehn. 2005. Europarl: A parallel corpus for statistical machine translation, http://www.statmt.org/europarl. In *MT summit*. 79–86.
[33] Grzegorz Kondrak. 2005. N-gram similarity and distance. In *International symposium on string processing and information retrieval (SPIRE)*. Springer, 115–126.
[34] Karen Kukich. 1992. Techniques for automatically correcting words in text. In *ACM Computing Surveys (CSUR)*. 377–439.
[35] Ted G. Lewis and Curtis R. Cook. 1988. Hashing for dynamic and static internal tables. In *Computer*. 45–56.
[36] Yuri Lin, Jean-Baptiste Michel, Erez Lieberman Aiden, Jon Orwant, Will Brockman, and Slav Petrov. 2012. Syntactic annotations for the google books ngram corpus. In *Association for Computational Linguistics (ACL)*. 169–174.
[37] Bhaskar Mitra and Nick Craswell. 2015. Query auto-completion for rare prefixes. In *International Conference on Information and Knowledge Management (CIKM)*. 1755–1758.
[38] Bhaskar Mitra, Milad Shokouhi, Filip Radlinski, and Katja Hofmann. 2014. On user interactions with query auto-completion. In *International Conference on Research and Development in Information Retrieval (SIGIR)*. 1055–1058.
[39] Alistair Moffat and Lang Stuiver. 2000. Binary Interpolative Coding for Effective Index Compression. In *Information Retrieval Journal (IRJ)*. 25–47.
[40] Donald R. Morrison. 1968. PATRICIA: practical algorithm to retrieve information coded in alphanumeric. In *Journal of the ACM (JACM)*. 514–534.
[41] Gonzalo Navarro, Ricardo A. Baeza-Yates, Erkki Sutinen, and Jorma Tarhio. 2001. Indexing methods for approximate string matching. In *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*. 19–27.
[42] Patrick Nguyen, Jianfeng Gao, and Milind Mahajan. 2007. MSRLM: a scalable language modeling toolkit. In *Microsoft Research MSR-TR-2007-144.2007*.
[43] Giuseppe Ottaviano and Rossano Venturini. 2014. Partitioned Elias-Fano Indexes. In *International Conference on Research and Development in Information Retrieval (SIGIR)*. 273–282.
[44] Adam Pauls and Dan Klein. 2011. Faster and Smaller N-gram Language Models. In *Association for Computational Linguistics (ACL)*. 258–267.
[45] Giulio Ermanno Pibiri and Rossano Venturini. 2017. Efficient Data Structures for Massive N-Gram Datasets. In *International Conference on Research and Development in Information Retrieval (SIGIR)*. 615–624.
[46] Bhiksha Raj and Ed Whittaker. 2003. Lossless Compression of Language Model Structure and Word Identifiers. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. 388–391.
[47] David Salomon. 2007. *Variable-length Codes for Data Compression*. Springer.
[48] Jangwon Seo and W. Bruce Croft. 2008. Local text reuse detection. In *International Conference on Research and Development in Information Retrieval (SIGIR)*. 571–578.
[49] Ehsan Shareghi, Matthias Petri, Gholamreza Haffari, and Trevor Cohn. 2015. Compact, efficient and unlimited capacity: Language modeling with compressed suffix trees. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2409–2418.
[50] Ehsan Shareghi, Matthias Petri, Gholamreza Haffari, and Trevor Cohn. 2016. Fast, Small and Exact: Infinite-order Language Modelling with Compressed Suffix Trees. *Transactions of the Association of Computational Linguistics (TACL)* 4, 1 (2016), 477–490.
[51] Andreas Stolcke. 2002. SRILM - an extensible language modeling toolkit. In *International Conference on Spoken Language Processing (ICSLP)*. 901–904.
[52] David Talbot and Miles Osborne. 2007. Randomised language modelling for statistical machine translation. In *Association for Computational Linguistics (ACL)*. 512–519.
[53] Larry H Thiel and HS Heaps. 1972. Program design for retrospective searches on large data bases. *Information Storage and Retrieval (ISR)* 8, 1 (1972), 1–20.
[54] Sebastiano Vigna. 2008. Broadword implementation of rank/select queries. In *Workshop on Experimental Algorithms (WEA)*. 154–168.
[55] Sebastiano Vigna. 2013. Quasi-succinct indices. In *International Conference on Web Search and Data Mining (WSDM)*. 83–92.
[56] Jeffrey Scott Vitter. 1998. External memory algorithms. In *European Symposium on Algorithms (ESA)*. 1–25.
[57] Taro Watanabe, Hajime Tsukada, and Hideki Isozaki. 2009. A succinct n-gram language model. In *International Joint Conference on Natural Language Processing (IJCNLP)*. 341–344.
[58] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. 1999. *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann.
[59] Susumu Yata. 2011. Prefix/Patricia trie dictionary compression by nesting Prefix/Patricia tries. In *Inernational Conference on Natural Language Processing (NLP)*.