Handling Massive *N*-Gram Datasets Efficiently

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

Two fundamental problems concern the handling of large n-gram language models: *indexing*, that is, compressing the n-grams and associated satellite values without compromising their retrieval speed, and *estimation*, that is, computing the probability distribution of the n-grams extracted from a large textual source.

Performing these two tasks *efficiently* is vital 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 simultaneously achieve high space reductions and no time degradation with respect to the state-of-the-art solutions and related software packages. In particular, we present a compressed trie data structure in which each word of an n-gram following a *context* of fixed length k, that is, 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, allowing the indexing of billions of strings. Despite the significant savings in space, our technique introduces a negligible penalty at query time.

Specifically, the most space-efficient competitors in the literature, which are both quantized and lossy, do not take less than our trie data structure and are up to 5 times slower. Conversely, our trie is as fast as the fastest competitor but also retains 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 by 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 times on the total runtime of the previous approach.

CCS Concepts: • **Information systems** → **Language models**; **Data compression**; **Information extraction**;

Additional Key Words and Phrases: Efficiency, scalability, algorithm engineering

ACM Reference format:
Giulio Ermanno Pibiri and Rossano Venturini. 2019. Handling Massive *N*-Gram Datasets Efficiently. *ACM Trans. Inf. Syst.* 37, 2, Article 25 (February 2019), 41 pages.
https://doi.org/10.1145/3302913

This work was partially supported by the BIGDATAGRAPES project (grant agreement no. 780751), which received funding from the European Union's Horizon 2020 research and innovation program under the Information and Communication Technologies program, and by the PEGASO project (POR FSE 2014-2020).

Authors' address: G. E. Pibiri and R. Venturini, University of Pisa, Largo Bruno Pontecorvo 3, 56127 Pisa, Italy and ISTI-CNR, Via Giuseppe Moruzzi, 1, 56127, Pisa, Italy; email: giulio.pibiri@di.unipi.it, rossano.venturini@unipi.it.

1046-8188/2019/02-ART25 $15.00
https://doi.org/10.1145/3302913

1 INTRODUCTION

An *n-gram* is a sequence of n tokens, where n ranges from 1 to N, that is, a small constant, for example, $N = 5$. 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). 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, 34, 35], spelling correction [32], similarity search [31], identification of text reuse and plagiarism [26, 46], automatic speech recognition [28], and machine translation [23, 41], to mention some of the most notable.

For example, query auto-completion is one of the key features that (virtually) 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 keywords typed by the user [2, 34, 35]. The identification of such patterns is possible by traversing a data structure that stores the n-grams seen during 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 data structure traversals are carried out in a handful of microseconds [2, 14, 28, 34, 35]. 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 prebuilt data structure in order to assess their existence or return a statistic, for example, a frequency count, to guide the correction [32]. If the n-gram is not found in the data structure, it is marked as a misspelled pattern: in that case, the correction happens by suggesting the most frequent word that follows the pattern with the longest matching history [14, 28, 32].

At the core of these example applications lies an *efficient* data structure mapping the n-grams to their associated satellite values, for example, the frequency counts representing the number of occurrences of the n-grams in some domain of interest or probability/back-off weights for word-predicting computations [23, 41]. The efficiency of the data structure should be both in time *and* space, because modern string search and machine translation systems make many queries over databases containing several billion n-grams that often do not fit in internal memory [14, 28]. 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 is necessary. While several solutions have been proposed for the indexing and retrieval of n-grams based on *tries* [21] or *hashing* [33], 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. When query processing speed is the main concern, space efficiency is almost completely neglected by not compressing the data structure using sophisticated encoding techniques [23]. In fact, space reductions are usually achieved by either lossy quantization of satellite values or by randomized approaches with false positives allowed [50]. The most space-efficient and lossless proposals in the literature still employ binary search over compressed representation to search for an n-gram: this results in a severe limitation during query processing because of the lack of a compression strategy with a fast random-access operation [41]. 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 accessed. Finally, storing n-grams in hash tables with linear probing results in prohibitive space usage since the tables are allocated with significant extra empty space (e.g., 30%–50%) to allow fast random access [23, 41].

Since a solution that is compact, fast, and lossless at the same time is still elusive, the first objective of this article is to address 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 this problem as *indexing* n-gram datasets.

The other related problem that we study in this article is 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 an efficient, compressed index that maps the n-grams extracted from a collection to their probabilities. Clearly, the way that these probabilities are computed depends on the chosen model. The problem of estimation has received a lot of attention throughout the years: 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 [29]—and, in particular, their modified version introduced by Chen and Goodman [10]—have gained popularity thanks to their relatively low-perplexity performance. This makes modified Kneser-Ney the de-facto choice for language model toolkits. In fact, all of the following software libraries, widely used in both academia and industry (e.g., Google [5, 8] and Facebook [11]), support modified Kneser-Ney smoothing: KenLM [23, 24], BerkeleyLM [41], RandLM [50], Expgram [54], MSRLM [39], SRILM [49], IRSTLM [20], and the recent approach based on suffix trees by Shareghi et al. [47, 48]. For these reasons, Kneser-Ney is the model that we consider in this work and that we describe in Section 2.

The current limitation of the the abovementioned software libraries is that estimation of such models occurs in internal memory; as a result, they are not able to scale to the dimensions that we consider in this work. An exception is represented by the work of Heafield et al. [24] (and included as part of the library KenLM), which 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 other toolkits [24]. Therefore, our work aims at improving on the I/O efficiency of this approach.

Our Contributions. We list here the contributions of this article.

(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* [17, 18] as to efficiently support random-access operations and successor queries over the compressed sequence. As a side contribution, we adopt a hashing approach that leverages on *minimal perfect hash* in order to use tables of size equal to the number of stored n-grams and spend one random access to retrieve the corresponding n-gram satellite value.
(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 levels. 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, that is, its preceding k words, is encoded as an integer whose value is proportional to the number of words that follow this 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 introducing only a slight penalty at query-processing time. Specifically, the most space-efficient proposals in the literature, which are both quantized and lossy, do not take less space than our trie data structure and are up to 5× slower. Conversely, our trie data structure is as fast as the fastest competitor but also retains 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 [24] 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 these 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 the 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 state-of-the-art approach.

(5) We introduce many optimizations to further enhance the runtime of our algorithm, such as asynchronous CPU and I/O threads, parallel least-significant-digit (LSD) radix sort, block-wise compression, and multi-threading. With an extensive experimental analysis conducted over billions of strings, we study the behavior of our algorithm 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 runtime by 1.6×, on average, making our algorithm 4.5× faster than the state-of-the-art solution.

Article Organization. Although the two problems that we address in this article—that is, indexing and estimation—are strictly correlated, we address them one after the other (Sections 4 and 5, respectively) in order to introduce the material in an incremental way. We show the experimental evaluation right after the description of our techniques for each problem rather than deferring it to the end of the article. We believe that this form is the most suitable to documenting the achieved results. We intend for each section to be an independent unit of exposition. In light of these considerations, the article is structured as follows.

Section 2 explains the notation used in the article and introduces the relevant background, such as the the Kneser-Ney smoothing technique. Section 3 presents related work on the state-of-the-art solutions for the two problems that we tackle in the article. Section 4 treats the problem of indexing. In particular, Section 4.1 describes our compressed trie data structure, whereas Section 4.2 describes the hash-based index. The efficiency of these data structures is validated in Section 4.3 with a rich set of experiments. Section 5 treats the problem of estimation. We discuss our improved estimation algorithm in Section 5.1 and validate its performance in Section 5.2 by also introducing many optimizations. We present our conclusions in Section 6.

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 of interest. The primary 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: compute $\mathbb{P}(w_n|w_1^{n-1})$ for all $w_1^n \in \mathcal{S}$. Using informal words, we would say that the goal is to *predict* the "next" word after a given context.

In what follows, let us indicate with w_i^j the *sequence* of words $w_i \cdots w_j$, for any $1 \leq i \leq j$, which is equal to ε, the *empty* string, whenever $i < j < 0$.

The conditional probability $\mathbb{P}(w_n|w_1^{n-1})$ is equal to $\prod_{k=1}^{n} \mathbb{P}(w_k|w_1^{k-1})$, that is, all contexts of length $1, 2, \ldots, n-1$ contribute to the final computed value. Therefore, computing the probability *exactly* is inefficient in both time and memory requirements when n is large. To make this task feasible and *efficient*, n-gram language models are adopted. Unless otherwise specified, throughout the article we consider datasets of n-grams consisting of words. Since we impose that $1 \leq n \leq N$, where N is a small constant (e.g., $N = 5$), dealing with strings of this form permits one to work with a context of *at most* $N-1$ preceding words. This ultimately implies that the aforementioned probability $\mathbb{P}(w_n|w_1^{n-1}) = \prod_{k=1}^{n} \mathbb{P}(w_k|w_1^{k-1})$ can be approximated with $\prod_{k=1}^{n} \mathbb{P}(w_k|w_{k-N-1}^{k-1})$.

Now, the way that each N-gram probability $\mathbb{P}(w_k|w_{k-N-1}^{k-1})$ is computed depends on the chosen language model.

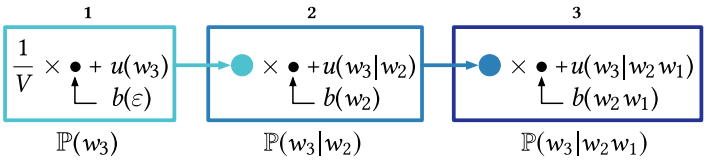


Fig. 1. The Kneser-Ney interpolated probabilities for a 3-gram, calculated in a bottom-up fashion, from left (1-gram) to right (3-gram).

2.1 Modified Kneser-Ney Smoothing

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 an n-gram backoff-smoothed language model, the probability of w_n with context w_1^{n-1} is computed according to the following recursive equation:

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

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

Among these many models, the *modified* version of Kneser-Ney smoothing [29], introduced by Chen and Goodman [9], was shown to have the best performance in terms of perplexity score. As already mentioned in Section 1, modified Kneser-Ney is the de-facto choice for language modeling and all major software packages support it. For these reasons, we adopt Kneser-Ney in this work.

Under this model, the conditional probability $\mathbb{P}(w_n|w_1^{n-1})$ is computed as

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

Refer to Figure 1 for an example of a 3-gram probability computation. Note that all lower-order probabilities are interpolated together and $u(w_n|w_1^{n-1})$ and $b(w_1^{n-1})$ are, respectively, the normalized probability and context backoff for n-gram w_1^n:

$$u(w_n|w_1^{n-1}) = \frac{a(w_1^n) - D_n(a(w_1^n))}{\sum_x a(w_1^{n-1}x)} \tag{2}$$

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

We now explain the quantities used in the above equations. The quantity $a(w_1^n)$ is called the *modified count* for the n-gram w_1^n and is equal to either the raw occurrence count $c(w_1^n)$ of w_1^n in the text when $n = N$ or to $|\{x : xw_1^n\}|$ when $n < N$, that is, the number of distinct words to the left of w_1^n (also called the *left extensions* of w_1^n). The quantity $N_k(w_1^{n-1}\bullet) = |\{x : a(w_1^{n-1}x) = k\}|$ represents the number of n-grams having context w_1^{n-1} and modified count equal to k, whereas $N_{3+}(w_1^{n-1}\bullet)$ is equal to $|\{x : a(w_1^{n-1}x) \geq 3\}|$, that is, the number of n-grams having a modified count greater than or equal to 3.

The recursion shown in Equation (1) terminates when unigrams are interpolated with the probability of the *unknown word*, which is uniformly distributed by assumption: $\mathbb{P}(w_n) = u(w_n) + b(\varepsilon) \times \frac{1}{V}$, where V denotes the size of the vocabulary, that is, the number of distinct words appearing in the textual collection used for estimating the language model. Note that $b(\varepsilon)/V$ is a constant quantity that depends on the used textual collection.

Last, following [9, 10], closed-form discounts $D_n(k)$ are computed according to

$$D_n(k) = \begin{cases} 0 & \text{if } k = 0 \\ k - (k+1) \times \frac{t_{n,1} t_{n,k+1}}{(t_{n,1}+2t_{n,2}) t_{n,k}} & \text{if } k = 1, 2, 3 \\ D_n(3) & \text{otherwise,} \end{cases} \tag{4}$$

with the smoothing statistic $t_{n,k}$ representing the total number of n-grams in the corpus with modified count k, that is, $t_{n,k} = |\{w_1^n : a(w_1^n) = k\}|$ for $k = 1, 2, 3$ and 4.

3 RELATED WORK

In this section, we review the solutions proposed in the literature for the two problems that we address in the article:, indexing and estimation.

3.1 Indexing

We first discuss the classic data structures used to represent 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-gram datasets: *tries* [21] and *hash tables* [33].

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 an 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, for example, 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 shallow and wide. Therefore, they are implemented as a collection of (few) sorted arrays: for each level of the trie, a separate array is built to contain all of the triplets for that level, sorted by the words. In this implementation, a pair of adjacent pointers indicates the subarray listing all of the children for a word, which can be inspected by binary search.

Hashing is another way to implement associative arrays: for each value of n from 1 to N, a separate hash table stores all of the n-grams. At the location indicated by the hash function, we store a fingerprint value to lower the probability of a false positive (typically, the 4B or 8B hash of the n-gram itself) and the satellite data for the n-gram.

This data structure permits access of 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 an 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 (usually expensive) fingerprint values, are randomly distributed and, therefore, incompressible. Furthermore, hashing is a randomized solution, that is, there is a non-null probability of retrieving a frequency count for an n-gram *not* actually belonging to the indexed corpus (false positive). Such probability equals 2^{-b}, where b indicates the number of bits dedicated to the fingerprint values: larger values of b yield a smaller probability of false positive but also increase the space of the data structure.

Pauls and Klein [41] propose trie-based data structures with nodes represented via sorted arrays or hash tables with linear probing. The sorted arrays are compressed using a variable-length block encoding, consisting of the following steps: (1) a configurable radix $r = 2^k$ is chosen; (2) the number

of digits, d, needed to represent a number in base r is written in unary; and (3) the d digits are written in dk bits. To preserve the property of looking up a record by binary search, each sorted array is divided into blocks of 128B. 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 faster than the sorted array variant, but requires extra table allocation space to avoid excessive collisions.

Heafield [23] 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 $O(\log\log m)$[1] with high probability by using *interpolation* search [15]. Records in each sorted array are minimally sized at the bit level, improving the memory consumption over [41]. Pointers are compressed using the integer compressor devised in [44]. Values can also be quantized using the *binning* method [19] 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 [50] use Bloom filters [4] with lossy quantization of frequency counts to achieve a small memory footprint. The raw frequency count f_g of the n-gram g is quantized using a logarithmic codebook, that is, $\widetilde{f_g} = 1 + \log_b f_g$. 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, for example, $v = 8$. Given the quantized count $\widetilde{f_g}$ of 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 $\widetilde{f_g}$. In order to retrieve $\widetilde{f_g}$ at query time, the filter is queried with a 1 appended to g. This event is hashed using the k hash functions of the filter: if all of them test positive, then the count is incremented and the process is repeated. The procedure terminates as soon as any of the k hash functions hits a 0 and the previous count is reported (after conversion to a linear count). 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 use of the succinct encoding Level-Order Unary-Degree Sequence (LOUDS) [27] is advocated in the work by Watanabe et al. [54] to implicitly represent the trie nodes. The pointers for a trie of m nodes are encoded using a bitvector of $2m + 1$bits. Bit-level searches on the bitvector allow forward/backward navigation of the trie structure. Words and frequency counts are compressed using variable-byte encoding [45, 51], with an additional bitvector used to indicate the boundaries of the byte sequences to support random access to each element. Shareghi et al. [47, 48] also consider the use 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 [23].

The problem of representing trie-based storage for general-purpose string dictionaries is among one of the most studied in computer science, with many different solutions available [12, 25, 38]. It goes without saying that, given the properties that n-gram datasets exhibit, generic trie implementations are *not* suitable for their efficient treatment. However, comparison with the performance of these implementations gives useful insights into the performance gap with respect to a general

[1]Unless otherwise specified, all logarithms are in base 2 and $\log x = \log_2 x$ for any $x > 0$.

solution. We mention Marisa [56] as the best and most practical general-purpose trie implementation. The main idea is to use Patricia tries [37] 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 Estimation

In this section, we first discuss the related work concerning the estimation of language models, then we describe the state-of-the-art algorithm devised by Heafield et al. [24].

The use of the Map+Reduce paradigm for the problem has been advocated in [5]. As reported in that article, 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. [39] (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 the saving of some resources during estimation, it also imposes a significant burden during the most efficiency-demanding use of language models, which is query processing [10, 23]. We, instead, prefer to follow the approach of [24], which performs all of the steps of estimation to permit the building of an efficient, static, compressed index over the computed model.

The works by Stolcke [49] (SRILM), Federico et al. [20] (IRSTLM), Pauls and Klein [41] (BerkeleyLM), and Watanabe et al. [54] (Expgram) build Kneser-Ney language models in internal memory without resorting to sophisticated software optimizations and data compression techniques. As a result, they are not able to scale to the dimensions that we consider in this work.

Heafield et al. [24] (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-the-art solution to the problem. Specifically, it takes 25.4% and 7.7% of CPU time and RAM of SRILM and 16.4% and 16.6% of CPU and RAM of IRSTLM [24], respectively.

The recent approach by Shareghi et al. [47] resorts to compressed suffix trees to compute the Kneser-Ney probabilities on the fly. The experimental analysis reported in the article is compared with SRILM and shows that the approach is comparable in building time with SRILM indexes but several orders of magnitude (e.g., 1000×) slower to query. In [48], the same authors improved over their previous work [47] by precomputing some modified counts to speed up the on-the-fly calculation of the Kneser-Ney probabilities. Although precomputing 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 above, we aim at improving on the I/O efficiency of the 3-Sort approach by Heafield et al. [24] (KenLM) that we describe in detail in the following.

3.2.1 The 3-Sort Algorithm. 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; thus, we necessarily need to rely on disk usage.
(2) The estimate is performed without pruning; thus, the minimum occurrence count for an n-gram is 1.
(3) The compressed index built over the n-gram strings must reside in internal memory to allow fast query processing (e.g., for perplexity-score computations and machine translation).

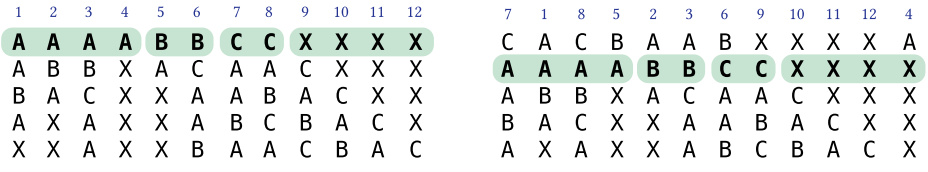


Fig. 2. A block of twelve 5-grams sorted in *suffix* order (a) and sorted in *context* order (b).

Since the sorted orders defined over a set of N-grams are central to the description of the algorithm that we are going to consider, we define them as follows. Consider a set of N-grams[2] as the one illustrated in Figure 2. The set is put into sorted order by sorting the N-grams on their words, as considered in a specific order. If the specific order is $N, N-1, \ldots, 1$, that is, we sort the N-grams from their last word up to the first, then the block is *suffix*-sorted: the last word is primary (Figure 2(a)). If the considered order is $N-1, N-2, \ldots, 1, N$, then the block is *context*-sorted: the penultimate word is primary (Figure 2(b)).

The algorithm consists of four streaming passes over the data that we are going to detail next: (1) counting, (2) adjusting counts, (3) normalization, and (4) interpolation and joining. 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 $c(w_1^N)$ for all distinct N-grams in the text (with order exactly N) by streaming through the textual corpus, using a window of size N words that slides by one word at a time. 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 4B vocabulary identifiers and unigram strings are written to disk as plain text. Their 8B Murmur hash is retained in internal memory. The occurrence counts, represented as 8B numbers, are accumulated in an open-addressing hash table with linear probing: the counts are finally 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 counts.* All blocks sorted in suffix order are merged into a single block B_N. This step aims at computing the modified counts $a(w_1^n)$ for the n-grams w_1^n that is equal to $|\{x : xw_1^n\}|$, which is the number of distinct words to the left of w_1^n.
By streaming through B_N sorted in suffix order, it is sufficient to compare consecutive entries to decide whether to write the record $\langle w_1^n, a(w_1^n)\rangle$ to a new block B_n or increment the currently computed $a(w_1^n)$. During the same pass, smoothing statistics $t_{n,k}$ are collected and discount coefficients $D_n(k)$ are calculated as in Equation (4).

(3) *Normalization.* This step computes normalized probabilities and backoffs according to Equations (2) and (3), respectively . For this purpose, the blocks B_n, $1 < n \leq N$, produced during the previous step, are sorted in context order such that, for each context w_1^{n-1}, the entries $w_1^{n-1}x$ are consecutive. Also in this case, a streaming pass through each block 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 (see Figure 1) is needed to perform interpolation. The computed backoffs are

[2]Throughout the article, whenever we need to show some examples, we consider an n-gram as consisting of n capital letters rather than words.

saved twice on disk, also as bare values without keys, one file per order $1 \leq n < N$ to facilitate the next step of interpolation and joining.

(4) *Interpolation and joining.* The last streaming step performs interpolation of all orders to compute the final Kneser-Ney probability, as in Equation (1). 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 1 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, for $1 \leq n < N$ (N-grams do not have backoff).

4 COMPRESSED INDEXES

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

4.1 Elias-Fano Tries

In this section, we present a compressed trie data structure based on the *Elias-Fano* representation [17, 18] 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 sorted-array trie levels given that we (fundamentally) need to randomly access the subarray pointed to by a pair of pointers. The pair is retrieved in constant time as well. Now, every access performed by binary search takes $O(1)$ without requiring any block decompression in contrast to other strategies [41].

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 identifier (ID in the following) conditionally to its context of fixed length k, that is, its k preceding words.

4.1.1 Data Structure. As it is standard, a unique integer ID is assigned to each distinct word to form the vocabulary of the indexed n-gram corpus. The vocabulary is implemented using a hash data structure that stores for each unigram its ID in order to retrieve it when needed in $O(1)$. If we sort the n-grams following the token-ID order, we have that all of the successors of gram $w_1^{n-1} = w_1 \cdots w_{n-1}$, that is, all grams whose prefix is w_1^{n-1}, form a strictly increasing integer sequence. For example, suppose that we have the unigrams {A, B, C, D}, which are assigned IDs $\{0, 1, 2, 3\}$, respectively. Now, consider the bigrams {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 in this article, is $\langle$A, C$\rangle$, that is, $\langle 0, 2\rangle$, because A and C are prefixes by A to form the bigrams AA and AC; the sequence of the successors of B is $\langle$B, C, D$\rangle$, that is, $\langle 1, 2, 3\rangle$, and so on. Concatenating all ranges, we obtain the integer sequence $\langle 0, 2|1, 2, 3|0, 3|1, 3\rangle$, which we can see in Figure 3(b) (the thick vertical bars, depicted in dark blue in Figure 3(b). They are not actually part of the sequence: they are shown to better highlight the different ranges. In order to distinguish the successors of an n-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).

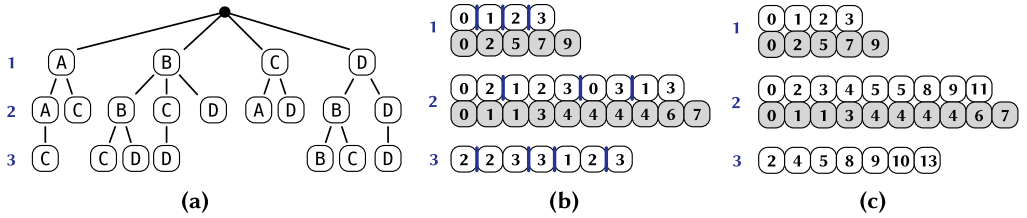


Fig. 3. In (a), we show an example of a trie of order 3, representing the set of grams {A, AA, AAC, AC, B, BB, BBC, BBD, BC, BCD, BD, C, CA, CD, D, DB, DBB, DBC, DDD}. In (b), we see the sorted-array representation of the trie, where each vocabulary token is assigned a distinct integer ID. Last, in (c), we show the final representation of the trie, where each sorted array has been transformed into a monotone sequence by computing the prefix sums of the ranges marked with the thick bars in (b). The shaded arrays represent the pointers.

The ID assigned to a unigram is also used as the position at which we read the unigram pointer in the unigram pointer sequence.

Therefore, apart from unigrams that are stored in a hash table, each level of the trie is composed of two integer sequences: one for the representation of the gram IDs and the other for the pointers. Figure 3 shows a graphical representation of what we described.

We have therefore reduced the problem of representing a trie to the problem of compressing (a few) integer sequences. While many integer compressors are available in the literature, in this work, we adopt Elias-Fano (EF) [17, 18], along with its *partitioned* variant (PEF) [40], which has been applied to inverted index compression showing an excellent space/time trade-off [40, 42, 52].

We now quickly state the salient features of this elegant integer encoding and point to the recent survey by Pibiri and Venturini [43] for a more detailed description.

Elias-Fano encodes a monotonically increasing sequence $S(m, u)$ of m positive integers drawn from a universe of size u in $\mathsf{EF}(\mathcal{S}(m, u)) \leq m\lceil \log \frac{u}{m} \rceil + 2m$ bits, which permits random access of an integer in constant time without decompressing the whole sequence. Again, see the survey in [43] for a description of the algorithm for random access and references therein. The *partitioned* Elias-Fano variant, proposed by Ottaviano and Venturini [40], splits the sequence into variable-sized partitions to enhance compression effectiveness. Clearly, the partitioned sequence organization introduces a level of indirection when resolving a random access, because a first search must be spent in the first level to identify the block in which the searched integer is located. We will return to and stress this point in Section 4.3.

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 may not be, as we can see from Figure 3(b). 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 add the last prefix sum (initially equal to 0). Then, our example sequence $\langle 0, 2|1, 2, 3|0, 3|1, 3\rangle$ 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, that is, $\langle 0, 2\rangle$. Now, the last prefix sum is 2; thus, we sum 2 to the values in the range of B, yielding $\langle 3, 4, 5\rangle$. Now, the last prefix sum is 5; thus, we sum 5 to the values in the range of C, yielding $\langle 5, 8\rangle$. Finally, the last prefix sum is 8; therefore, we sum 8 to the values in the range of D, obtaining $\langle 9, 11\rangle$. The final trie resulting from this transformation is shown in Figure 3(c).

If we sort the vocabulary IDs in decreasing order of occurrence, we make small IDs appear more often than large ones, which 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 unigram in the vocabulary, we count the number of times that it appears in all gram-ID sequences and assign IDs to vocabulary tokens in decreasing order of occurrence[3].

Frequency Counts. To represent the frequency counts, we use the *unique-value array* technique: each count is represented by its *index* in an array $C[n]$, $1 \leq n \leq N$, that collects all *distinct* frequency counts for the n-grams. This technique is widely used in data compression whenever the distribution of the represented values is extremely skewed, as it is in our case for the frequency counts of the n-grams: relatively few n-grams are very frequent while most appear only a few times. As we can better see in Table 1 (Section 4.3), the number of distinct counts is very small compared with the number of n-grams themselves; thus, the space for the arrays $C[n]$, $1 \leq n \leq N$, is negligible.

Now, each level of the trie, in addition to the sequences of gram-IDs and pointers, also has to store the sequence made by all frequency-count indexes. Unfortunately, this sequence of indexes is not monotone, yet it follows the aforementioned highly repetitive distribution. To exploit such repetitiveness, we assigned to each index a codeword of variable length. As similarly done for the gram-IDs, by assigning smaller codewords to more repetitive indexes, we have most indexes encoded in just a few bits. More specifically, starting from $k = 1$, we first assign all 2^k codewords of length k before increasing k by 1 and repeating the process until all indexes 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 [22], to the ith index 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 c. From codeword c and its length ℓ_c in bits, we can retrieve i by taking the inverse of the previous formula, that is, $i = c - 2 + 2^{\ell_c}$. In addition to the bitvector for the codewords themselves, we 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 the select_1 primitive that we use to retrieve ℓ_c. In fact, $b = \text{select}_1(i)$ gives us the starting position of the ith 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, a trie is conceptually represented by an array of levels, $levels[1, N]$, where each $levels[n]$ stores, for $1 \leq n \leq N$: the gram-ID sequence $levels[n].ids$, the sequence of frequency-count indexes $levels[n].indexes$, and the pointer sequence $levels[n].pointers$, with the only exceptions of 1-grams and N-grams, for which gram-ID and pointer sequences are missing, respectively.

Lookup. We now describe how the lookup operation is implemented, that is, how to retrieve the frequency count given an n-gram w_1^n. The corresponding pseudo-code is illustrated in Figure 4. We first perform n vocabulary lookups to map the n-gram tokens into their constituent IDs. We write these IDs into an array $ids[1, n]$ (Lines 2–4 in Figure 4(a)). This preliminary query-mapping step takes $\Theta(n)$ because each vocabulary lookup is performed in $O(1)$. Now, the search procedure has to locate $ids[i]$ in the ith level of the trie (Lines 3–6 in Figure 4(b)), as follows. If $n = 1$, then our search terminates: at the position $p = ids[1]$, we read the index $i = levels[1].indexes[p]$ to finally return $C[1][i]$. If, instead, n is greater than 1, the position p is used to retrieve the pair of pointers $(b, e) = (levels[1].pointers[p], levels[1].pointers[p + 1])$ in constant time, which delimits the range of IDs in which we have to search for $ids[2]$ in the second level of the trie. This range is inspected by binary search with the operation **find**, taking $O(\log(e - b))$ because each access to an EF-encoded sequence is performed in constant time. Now p is updated to be the position in $levels[2].ids$ at which

[3]Note that the number of occurrences of an n-gram can be different from its frequency count as reported in the dataset. The reason is that such datasets often do not include the n-grams appearing less than a predefined frequency threshold.

```
1 lookup(w_1^n)
2     ids[1, n] = [0, 0]
3     for i = 1; i ≤ n; i = i + 1
4         ids[i] = vocab.lookup(w_i)
5     p = search(ids, 1, n, false)
6     i = levels[n].indexes[p]
7     return C[n][i]
```

(a)

```
1 search(ids, i, j, remapping)
2     b = 0, e = 0, p = ids[i]
3     for k = 1; k ≤ j − i; k = k + 1
4         b = levels[k].pointers[p]
5         e = levels[k].pointers[p + 1]
6         p = find(levels[k + 1].ids, b, e, ids[k + i])
7     return p − (b if remapping == true else 0)
```

(b)

Fig. 4. The **lookup** and **search** functions. The **find**(A, b, e, x) function, used in the **search** pseudo-code, finds the integer x in the range $A[b, e)$ and returns its position in A.

$ids[2]$ is found in the range. Again, if $n = 2$, the search terminates by accessing $C[2][i]$, where i is now the index $levels[2].indexes[p]$. If n is greater than 2, we fetch the pair ($levels[2].pointers[p]$, $levels[2].pointers[p + 1]$) to continue the search of $ids[3]$ 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 $C[n][i]$ of w_1^n.

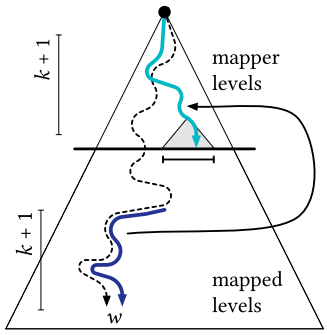
4.1.2 Context-Based Identifier Remapping. In this section, we describe a novel technique that lowers the space occupancy of the gram-ID sequences that constitute, as we have seen, the main component of the trie data structure.

The key 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 the context, and not bounded by the total vocabulary size V. Specifically, w is mapped to the position that it occupies within its siblings, that is, the words following the k-gram w_1^k. We call this technique *context-based identifier remapping* because each ID is remapped to the position that it takes relative to a context.

Figure 5(a) 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 dark-blue one in the figure) is searched along the same path occurring in the first $k + 1$ levels of the trie (e.g., the light-green one in the figure). This can be graphically interpreted as if the dark-blue path were projected onto the light-green path in order to search w along its sibling IDs that are the ones occurring after the k-gram w_1^k (the small dark-gray triangle in the figure). We note that this projection is always possible, that is, we are guaranteed to find any subpath 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 5(a) 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 other $N - k - 1$ levels are the ones formed by the remapped IDs.

The salient feature of the 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 (e.g., an additional data structure) 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, for an n-gram dataset comprising all n-grams for $n = 1, \ldots, N$ with $N \geq 2$, we can choose between $N - 2$ distinct context lengths k, that is, $1 \leq k \leq N - 2$. Clearly, the greater the context length we use, the smaller the remapped IDs will be but the searches will take longer. 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 n-grams* per order.

In what follows, we explain 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 Section 4.3, the dataset vocabulary can contain several million



(a)

	k	3-grams	4-grams	5-grams
Europarl	0	2404	2782	2920
	1	213 (11×)	480 (6×)	646 (5×)
	2	2404	48 (58×)	101 (29×)
YahooV2	0	7350	7197	7417
	1	753 (10×)	1461 (5×)	1963 (4×)
	2	7350	104 (69×)	249 (30×)
GoogleV2	0	4050	6631	6793
	1	1025 (4×)	2192 (3×)	2772 (2×)
	2	4050	221 (30×)	503 (14×)

(b)

Fig. 5. In (a), we depict the action performed by the context-based identifier remapping strategy. The last word ID w of any subpath of length $k + 1$, for example, the dark-blue one, is replaced with the position that it takes within its sibling IDs. These sibling IDs are found at the end (the gray triangle) of the search of w along the same path, for example, the light-green one, in the first $k + 1$ levels of the trie. In (b), we show the effect of 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 $k = 0, 1, 2$.

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 (reasonably large) 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 [36, 40]. In our case, the gram-ID sequences are encoded with Elias-Fano: from Section 4.1.1, 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 of the strategy is illustrated in Figure 5(b) which shows how the average gap of the gram-ID sequences of the datasets that we used in the experiments (see also Table 1, Section 4.3) is affected by the context-based remapping. As unigrams and bigrams constitute the mapper levels, these are kept unmapped: we show the statistics for the mapped levels, that is, 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, the technique with a context of length 1 achieves an average reduction of 7.2× (up to 11.3× on tri-grams). With a context of length 2, instead, we obtain an average reduction of 43.4× (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, it is, on average, 3× with a context of length 1 and 16.75× with a context of length 2.

Lookup. The described remapping strategy comes with an overhead at query time because the lookup algorithm illustrated in Figure 4 must map a default vocabulary ID to its remapped ID before it can be searched in the proper trie level.

Specifically, if the remapping strategy is applied with a context of length k, it involves $k \times (N - k - 1)$ additional searches in the trie levels. As an example, by looking at Figure 6, before searching the mapped ID 1 of D for the tri-gram BCD, we have to map the vocabulary ID of D, that is, 3, to

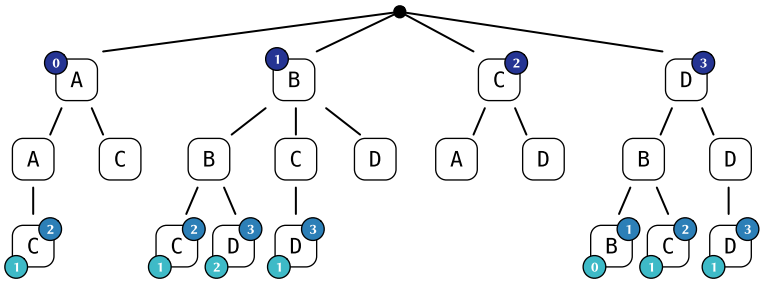


Fig. 6. Example of a trie of order 3, representing the set of grams {A, AA, AAC, AC, B, BB, BBC, BBD, BC, BCD, BD, CA, CD, DB, DBB, DBC, DDD}. From top to bottom, we show the vocabulary IDs in darkest blue, level-3 IDs in blue, and the light-green IDs (those in the left-hand bottom corner), which are derived by applying a context-based remapping with context length 1.

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. Therefore, we have a space/time trade-off that we explore with an extensive experimental analysis in Section 4.3. The pseudo-code for the lookup operations with context-based remapping is illustrated in Figure 7. Note that, in comparison with the pseudo-code in Figure 4(a), the remapping technique uses an array to store the remapped IDs (Line 3) and an additional `for` loop (Lines 7 and 8).

Example. To better understand how the remapping algorithm works, we now consider a small didactic example. We continue with the example trie from Section 4.1.1 and represented in Figure 6. The darkest-blue IDs are the vocabulary IDs and the blue ones are the last token IDs of the tri-grams as assigned by the vocabulary. We now explain how the remapped IDs, represented in light green, 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, that is, the C because we are using a context of length 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 6 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 an n-gram of order 4, w_1^4, we can choose to map w_4 as the position that it takes within the successors of w_3 (context of length 1) or within the successors of $w_2 w_3$ (context of length 2).

4.2 Hashing

Since the indexed n-gram corpus is static, we obtain a *full* hash utilization by resorting to Minimal Perfect Hash (MPH). We index all n-grams (of the same order n) into a separate MPH table, $levels[n]$, each with its own MPH function h_n. This introduces a twofold advantage over the linear probing approach used in the literature [23, 41]: 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 only 2.61b per key, on average.

```
1  lookup(w_1^n, k)
2      ids[1, n] = [0, 0]
3      remapped_ids[1, n] = [0, 0]
4      for i = 1; i ≤ n; i = i + 1
5          id = vocab.lookup(w_i)
6          ids[i] = remapped_ids[i] = id
7      for i = k + 1; i ≤ n; i = i + 1
8          remapped_ids[i]
               = search(ids, i − k, i, true)
9      p = search(remapped_ids, 1, n, false)
10     i = levels[n].indexes[p]
11     return C[n][i]
```

Fig. 7. The **lookup** function with context-based remapping of order k.

At the hash location for an n-gram, we store its 8B hash key to have a false-positive probability of 2^{-64} (4B hash keys are possible as well) and the index of the frequency count in the unique-value array $C[n]$ that keeps all distinct frequency counts for order n. 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 speed. We also use this hash approach to implement the vocabulary of the previously introduced trie data structure.

Lookup. Given the n-gram w_1^n, we compute the position $p = h_n(w_1^n)$ in the relevant table $levels[n]$. We then access the count index i stored at position p and, finally, retrieve the count value $C[n][i]$.

4.3 Experiments

In this section, we first present experiments to validate the effectiveness of our compressed data structures in relation to the corresponding query processing speed. We then 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 of all unpruned n-grams extracted from the English Europarl parallel corpus [30], available at http://www.statmt.org/europarl.
- YahooV2 is a collection of English n-grams with a minimum frequency count equal to 2, extracted from a corpus of 14.6 million documents crawled from more than 12,000 sites during 2006 [1]. 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. These standard datasets are also suitable to test our data structures on different corpora sizes: starting from the left of the table, each dataset has roughly 10× 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 discussed in Section 3.1: BerkeleyLM by Pauls

Table 1. Number of n-grams and Distinct Frequency Counts for the Datasets Used in the Experiments

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	101,428,257	7147	828,223,677	45,285	11,131,242,087	1,073,473

and Klein [41] (Java code at https://github.com/adampauls/berkeleylm); Expgram by Watanabe et al. [54] (C++ code at https://github.com/tarowatanabe/expgram); KenLM by Heafield [23] (C++ code at http://kheafield.com/code/kenlm); Marisa by Yata [56] (C++ code at https://github.com/s-yata/marisa-trie); and RandLM by Talbot and Osborne [50] (C++ code 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 2.4GHz, with 193GB of RAM, running Linux 3.13.0, 64b. Our implementation is in standard C++11 and compiled with gcc 5.4.1 with the highest optimization settings, that is, with compilation flags `-O3` and `-march=native`. To ensure a fair comparison with the other competitors, we used the same compiler and optimization flags for all C++ implementations.

Template specialization was preferred over inheritance to avoid the virtual method call overhead, which can be disruptive for the very fine-grained operations that 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 Single Instruction Multiple Data (SIMD) 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 by 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 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 query algorithms were run on a single core.

4.3.1 Elias-Fano Tries. In this section, we test the efficiency of the trie data structure described in Section 4.1.

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 refer to the trie data structure described in Section 4.1.1, when the sorted arrays are encoded with EF and PEF [40]. Subsequent rows indicate the space gains obtained by applying the context-based remapping strategy using EF and PEF for contexts of lengths 1 and 2, respectively. For GoogleV2, 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 (together) 27% of the dataset.

Table 2. Average Bytes Per Gram (bytes/gram) and Average lookup Time Per Query in Micro Seconds (μsec/query)

			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 (−5%)	1.35 (+6%)	1.91 (−12%)	1.73 (+8%)	1.52 (−29%)	1.91 (−9%)
CONTEXT-BASED ID REMAPPING	$k = 1$	EF	1.67 (−15%)	1.58 (+24%)	1.89 (−13%)	2.05 (+28%)	1.91 (−10%)	3.03 (+45%)
		PEF	1.53 (−22%)	1.61 (+26%)	1.63 (−25%)	2.16 (+35%)	1.31 (−39%)	2.30 (+10%)
	$k = 2$	EF	1.46 (−26%)	1.60 (+25%)	1.68 (−22%)	2.08 (+30%)	—	—
		PEF	1.28 (−35%)	1.64 (+28%)	1.38 (−36%)	2.15 (+35%)	—	—

The bytes/gram cost also includes the space of representation for the pointer sequences.

As expected, partitioning the gram sequences using PEF yields a better space occupancy. Though Ottaviano and Venturini [40] describe a dynamic programming algorithm to find the partitioning that minimizes the space occupancy of a monotone sequence, we instead adopt a uniform partitioning strategy. Partitioning the sequence uniformly has several advantages over variable-length partitions for our setting. As we have seen in Section 4.1.1, 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 introduces only a minor overhead in query processing for inverted index queries [40] (as it has to be performed once and successive accesses are directed only 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 to 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 8 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 show the one for the Europarl dataset. The dashed black line illustrates how the average lookup time varies when all gram-ID sequences are partitioned using the same block size. The figure suggests using partitions of 64 integers for bi-gram sequences and of 128 for all other orders, that is, for $N \geq 3$, given that the space usage remains low without increasing the query processing speed much. With this choice of block sizes, the increase in space consumption with respect to the optimal partitioning 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 8. However, excessively reducing the block size may ruin the advantage in space reduction. Therefore, it is convenient to use small block sizes for the most traversed sequences, for example, the bi-gram sequences, that must be searched several times during the query-mapping phase when 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).

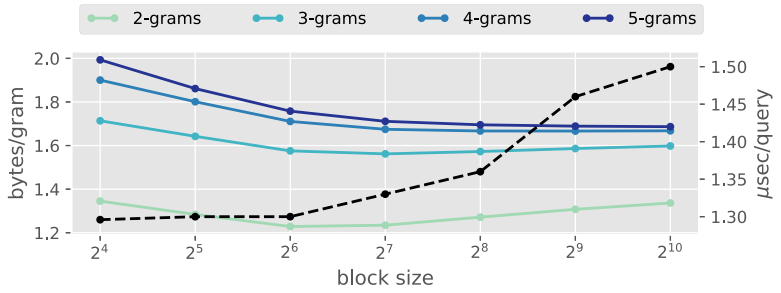


Fig. 8. Bytes per gram (left vertical axis) and μs per query (right vertical axis, black dashed line) by varying block size in PEF uniform on the gram-ID sequences of Europarl.

Context-Based Identifier Remapping. Concerning the effectiveness of context-based remapping, we can see from Table 2 that remapping the gram IDs with a context of length $k = 1$ is already able to reduce 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 $k = 1$. The lookup speeds for $k = 1$ and $k = 2$ are about the same because the number of successors for a bi-gram is very small, on average (as already noted with the numbers shown in Figure 5(b)). Therefore, a search among very few successors (case for $k = 2$) results in an almost negligible time overhead with respect to the case of one single search (case for $k = 1$).

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 of 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 Section 4.1.2, 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. Note 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 refers to the strategy described in Section 4.1.1 that assigns variable-length codewords to the ranks of the counts and keeps track of codeword 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 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, that is, Variable-len. codewords.

The time for retrieving a count was about the same for all three techniques. Prefix-summing the sequence and applying EF does not provide 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%

Table 3. Average Bytes Per Count for Different Techniques

	Europarl	YahooV2	GoogleV2
Variable-len. codewords	0.36	**0.47**	1.46
Prefix sums + EF	0.35 (−2%)	0.62 (+33%)	1.59 (+9%)
Prefix sums + PEF	**0.30** (−17%)	0.51 (+9%)	**1.30** (−11%)
Variable-len. block coding	0.76 (+156%)	0.79 (+56%)	1.32 (+1%)
Packed	1.63 (+445%)	2.00 (+294%)	2.63 (+102%)
VByte	3.21 (+975%)	3.32 (+555%)	—

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, in contrast to all other competitors, can also be used to index frequency counts. BerkeleyLM COMPRESSED variant uses the Variable-len. block-coding mechanism explained in Section 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 does not store count ranks; instead, it directly compresses the counts themselves using variable-byte encoding (VByte) with an additional binary vector to be able to randomly access the counts sequence. The available RAM of our test machine (193GB) 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 these techniques beats the three techniques that we previously introduced except for the BerkeleyLM COMPRESSED variant, which is ≈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. Now we use the analysis done so far 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; we use 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, that is, 2 for Europarl and YahooV2 and 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 Figure 9(c) and Figure 9(d), 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). Instead, the timing breakdowns in Figure 9(a) and Figure 9(b) clearly highlight how the context-based remapping technique raises the time that we spend in the query-mapping phase, during which the IDs are mapped to their reduced IDs. In this case, the two phases of query mapping (given by vocabulary lookups plus context-based remapping) and search are almost the same, while in the PEF-Trie the search phase dominates.

4.3.2 Hashing. We build our MPH tables using 8B hash keys 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 4B per distinct count on Europarl and YahooV2 and 8B on GoogleV2.

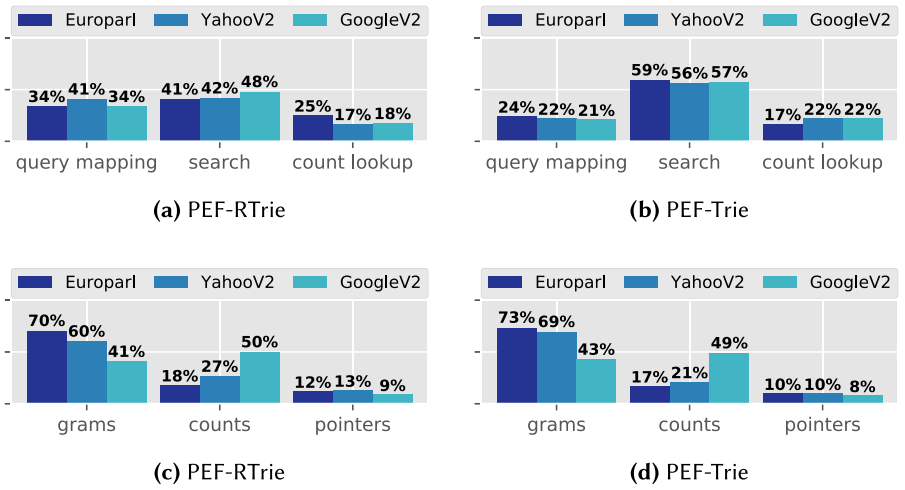


Fig. 9. Trie data structures timing (a-b) and size (c-d) breakdowns in percentage on the tested datasets. For the timing breakdowns we distinguish the three phases of query mapping, ID-search and final count lookup. For the space breakdowns we distinguish, instead, the contribution of gram-ID, count and pointer sequences.

For all three datasets, the number of bytes per gram, including the cost of the hash function itself (0.33B 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 4B 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 results in poor improvements, as we pay for the high cost of the hash keys.

The constant-time access capability of hashing makes gram lookup extremely fast by requiring, on average, 1/3 of a microsecond per lookup (exact numbers are reported in Table 4). All the time is spent in computing the hash function itself and accessing the relative table location: the final count lookup is completely negligible.

4.3.3 Overall Comparison. We now compare the performance of our selected trie-based solutions, the PEF-RTrie and PEF-Trie, as well as our minimal perfect hash approach against the competitors mentioned at the beginning of the experiments. 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 microseconds. 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 store only probabilities and backoffs, whereas BerkeleyLM can be used to store either counts or probabilities. For those competitors storing frequency counts, we discussed their count representation in Section 4.3.1. Expgram, KenLM, and Marisa require too much memory for building 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.

Table 4. Average Bytes Per Gram (bytes/gram) and Average lookup Time Per Query in Microseconds Per Query (μsec/query)

	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	
Berk. C.	1.70	(−9%)	2.83	(+109%)	1.69	(−11%)	3.48	(+102%)	1.45	(−5%)	4.13	(+117%)
		(+33%)		(+73%)		(+22%)		(+62%)		(+11%)		(+80%)
Berk. H.3	6.70	(+259%)	0.97	(−29%)	7.82	(+310%)	1.13	(−34%)	9.24	(+508%)	2.18	(+14%)
		(+423%)		(−41%)		(+465%)		(−47%)		(+608%)		(−5%)
Berk. H.50	7.96	(+326%)	**0.97**	(−29%)	9.37	(+391%)	**0.96**	(−44%)	—		—	
		(+522%)		(−41%)		(+577%)		(−55%)				
Expgram	2.06	(+10%)	2.80	(+107%)	2.24	(+17%)	9.23	(+435%)	—		—	
		(+61%)		(+71%)		(+62%)		(+329%)				
KenLM T.	2.99	(+60%)	1.28	(−6%)	3.44	(+80%)	1.94	(+12%)	—		—	
		(+134%)		(−22%)		(+149%)		(−10%)				
Marisa	3.61	(+93%)	2.06	(+52%)	3.81	(+100%)	3.24	(+88%)	—		—	
		(+182%)		(+26%)		(+175%)		(+51%)				
RandLM	1.81	(−3%)	4.39	(+224%)	2.02	(+6%)	5.08	(+194%)	2.60	(+71%)	9.25	(+385%)
		(+41%)		(+168%)		(+46%)		(+136%)		(+99%)		(+302%)
MPH	**8.33**		**0.26**		**8.33**		**0.32**		**8.33**		**0.37**	
KenLM P.3	9.40	(+13%)	0.43	(+63%)	9.41	(+13%)	0.38	(+20%)	—		—	
KenLM P.50	16.91	(+103%)	0.31	(+17%)	16.92	(+103%)	0.34	(+8%)	—		—	

For our data structures, that is, PEF-Trie and PEF-RTrie, the bytes/gram cost also includes the space of representation for the pointer sequences. Berk. is short for BerkeleyLM.

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 (Berk.) COMPRESSED (C.) variant results are 21% larger than our PEF-RTrie implementation and slower by more than 70%. It gains, instead, an advantage of roughly 9% over our PEF-Trie data structure, but it is also more than 2 times slower. The HASH variant uses hash tables with linear probing to represent the nodes of the trie. Therefore, we test it with a small extra space factor of 3% for table allocation (H.3) and with 50% (H.50), which is also used as the default value in the implementation, to obtain different time/space trade-offs. Clearly, the space occupancy of both hash variants do not compete with those 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, note that our PEF-Trie data structure is competitive anyway, 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: about 70% 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 with the PEF-RTrie data structure, KenLM's trie is slightly faster, 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 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 a small memory footprint and returns approximated frequency counts when queried. We build its data structures using the default setting recommended in the documentation: 8b for frequency count quantization and 8b per value to yield a false-positive rate of $\frac{1}{256}$. While being from 2.3 to 5 times slower than our exact and lossless approach, it is quite compact because the quantized frequency counts are recomputed on the fly using the procedure described in Section 3.1. However, its space is even larger than that of our n-gram representation by 61%. It is also larger 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 show 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 yield a slower random-access capability with respect to our minimal perfect hashing approach.

Perplexity Benchmark. In addition to 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 [19] to quantize probabilities and backoffs and allow any quantization bits ranging from 2 to 32. Uni-gram 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 to permit a more efficient computation of sentence-level probabilities, with a state 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, which 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 out of vocabulary (OOV) rate of, respectively, 16% and 1.82% on the test query file. As Expgram builds quantized models using only 8 quantization bits for both probabilities and backoffs, we also use this number of quantization bits for our tries and the KenLM trie. For all data structures, BerkeleyLM truncates the mantissa of floating-point values to 24b and then stores indices to distinct probabilities and backoffs. RandLM was built, 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, note that our index is also smaller than that of 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 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 that storing unquantized values results in indistinguishable differences in perplexity while unnecessarily increasing the space of the data structure, as 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. Not surprisingly, this makes a linear probing implementation actually faster, by 38% on average,

Table 5. Perplexity Benchmark Results Reporting Average Number of Bytes Per Gram (bytes/gram) and Microseconds Per Query (μsec/query) Using Modified Kneser-Ney 5-gram Language Models Built From Europarl and YahooV2 Counts

	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	
Berk. C.	6.50	(+87%)	1.19	(+372%)	6.39	(+76%)	1.08	(+188%)
		(+124%)		(+322%)		(+109%)		(+152%)
Berk. H.3	9.36	(+169%)	0.84	(+234%)	8.75	(+140%)	0.74	(+96%)
		(+222%)		(+199%)		(+186%)		(+72%)
Berk. H.50	12.31	(+254%)	0.35	(+39%)	12.01	(+230%)	**0.30**	(−19%)
		(+323%)		(+24%)		(+293%)		(−29%)
Expgram	4.15	(+19%)	3.83	(+1425%)	5.80	(+59%)	14.05	(+3638%)
		(+43%)		(+1265%)		(+90%)		(+3179%)
KenLM T.	4.58	(+32%)	**0.23**	(−8%)	5.04	(+39%)	0.39	(+5%)
		(+58%)		(−18%)		(+65%)		(−8%)
RandLM	4.01	(+15%)	6.48	(+2478%)	3.86	(+6%)	6.25	(+1561%)
		(+38%)		(+2207%)		(+26%)		(+1357%)
MPH	**9.92**		0.15		**9.94**		0.24	
KenLM P.3	14.77	(+49%)	0.32	(+106%)	14.84	(+49%)	0.30	(+25%)
KenLM P.50	21.48	(+117%)	**0.10**	(−36%)	21.57	(+117%)	**0.15**	(−40%)

Berk. is short for BerkeleyLM.

than a minimal perfect hash approach when a large multiplicative factor is used for table allocation (P.50). The price to pay is the doubling of the space, however. On the other hand, the P.3 variant is larger (by 50%) and slower (by 60% on average).

5 FAST ESTIMATION

The problem that we tackle in this section isestimating a modified Kneser-Ney language model (see background in Section 2.1), that is, computing the probability and backoff penalty for every n-gram, $1 \leq n \leq N$, extracted from a large textual source.

5.1 Improved Construction: The 1-Sort Algorithm

From the description given in Section 3.2, we observe that the runtime 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 10. 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 answers this question positively by designing an algorithm that requires only one sorting step in external memory.

The 1-Sort algorithm that we are going to describe performs three steps: (1) counting (Section 5.1.1); (2) adjusting counts (Section 5.1.2); and (3) in a single, last, pass—normalization and interpolation (Section 5.1.3), joining and index construction (Section 5.1.4).

In what follows, we detail these steps, showing how to save two steps of sorting in external memory.

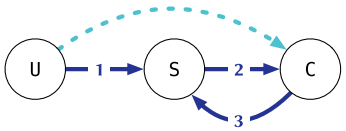


Fig. 10. Sorting passes performed between N-grams: unsorted (U), suffix-sorted (S), and context-sorted (C). Solid arrows illustrate the path followed by the 3-Sort algorithm; the dashed arrow illustrates the one followed by the 1-Sort algorithm.

5.1.1 Counting. This first step is performed similarly to the counting step of 3-Sort. A window of N words slides by one word at a time to scan the input text completely. We maintain an in-memory block of bytes to accommodate as many N-grams as possible, that is, 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 8B frequency count. In order to tell whether an N-gram was already seen or not during the scanning of the input, we associate a 4B 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 of comparing the extracted N-gram string with the $4N$ bytes stored in the block starting from the byte at position $k \times (4N + 8)$. If the comparison yields equality, then we increment the corresponding count; 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 operations 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.

5.1.2 Adjusting Counts. All blocks written to disk by the counting step are merged during this step 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 $D_n(k)$, for $k = 1, \ldots, 4$ (Equation (4)).

Because smoothing statistics and, thus, discount coefficients depend on the modified counts of the n-grams, the key ingredient that we develop in this section 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.

Specifically, the records written by the counting step are merged and accumulated in an in-memory block $block[1, m]$ of m records. When the block fills up, we run the algorithm over the block and then write it to disk. We repeat the process until the whole input B_N is processed completely. At the end of the process, we use Equation (4) to compute the discount coefficients $D_n(k)$.

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. Because the next step of estimation is normalization and requires context order, computing discount coefficients directly over the strings sorted in context order has the benefit of avoiding sorting from suffix to context. We are, therefore, eliminating the sorting step 2 shown in Figure 10.

Exploiting the Completeness of the N-gram Strings. First, observe that since estimation is done without pruning by assumption *and* N-grams are extracted using a window of size N that

	1	2	3	4	5	6	7	8	9	10	11	12
5	**C**	A	**C**	B	A	A	B	X	X	X	X	A
4	**A**	A	**A**	A	B	B	**C**	**C**	X	X	X	X
3	**A**	B	**B**	X	A	C	**A**	**A**	C	X	X	X
2	B	A	C	X	X	A	**A**	**B**	A	C	X	X
1	A	X	A	X	X	A	B	C	B	A	C	X

Fig. 11. The left extensions (words in blue) of AC must be found in the region highlighted by the light-green rectangle, that is, the run of entries whose context of length 1 is equal to A.

slides by one word at a time, *the strings in B_N cover the input text completely*. This means that all 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 11 and consider the first 5-gram ABAAC in the context-sorted block. For example, we know that its substring BAA must appear at positions 1, 2, and 3 of some other 5-grams (the ones in positions 7, 1, and 2, respectively). In particular, we know that its prefix of length 4, that is, ABAA, will be matched at position 2 in some other 5-gram (in this case, the second one, i.e., XABAA).

This observation means that all lower-order n-grams 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-grams, for $1 \leq n < N$.
(2) Let C_{n-1} be the context of length $n-1$ of an n-gram w_1^n. The number of distinct left extensions, that is, the distinct words appearing to the left of w_1^n, can be computed by *scanning the N-grams whose context of length $n-1$ is equal to C_{n-1}*[4].

By exploiting these two properties, we now explain the linear-time algorithm for computing the distinct left extensions in context order.

Computing Distinct Left Extensions in Context Order. For ease of explanation, let us consider an N-gram w_1^N as composed by three pieces, in this 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, that is, $a(C_{n-1}w_N)$. 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 (these entries $C_{n-1}w_N$ are clearly consecutive in suffix order). However, from fact (2), we know that *every left extension must necessarily appear to the left of the context* C_{n-1}; thus, we need to scan only the entries having context C_{n-1}.

The quantity $a(C_{n-1}w_N)$ is computed using a direct-address table of size $\Theta(V)$, called *statistics* in the pseudo-code shown in Figure 12(b), in which we store, for each distinct w_N, the last seen left word (*left*) and the number of distinct left words seen so far (*count*).

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 (*right*) 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

[4]Observe that we could compute the left extensions for an n-gram by directly scanning the N-grams having w_1^n as a context of length n. Again, consider the example in Figure 11. We could scan the N-grams in positions 7 and 8 to compute the distinct left extensions (words in blue) of the bi-gram AC instead of the ones in positions 1, 2, 3, and 4. The problem with this approach is that we would not be able to compute the desired quantity for $(N-1)$-grams because, obviously, a context of length $N-1$ cannot be extended to the left. Moreover, consider the first 5-gram ABAAC. Since interpolation produces the probabilities for all of its suffixes, that is, for C, AC, AAC, and BAAC, we need the modified counts for *these* suffixes and not for its contexts A, AA, BAA, and ABAA that we could have computed with the other approach.

```
1  compute_left_extensions(block, m)
2      p_1^N = block[1]                    ▷ previous record
3      for i = 1; i ≤ m; i = i + 1
4          w_1^N = block[i]
5          right = w_N
6          for n = 1; n < N; n = n + 1
7              if n != 1 and w_{N-n}^{N-1} != p_{N-n}^{N-1}
8                  ++ranges[n]
9                  for k = 1; k ≤ 4; k = k + 1
10                     T[n][k] += R[n][k]
11                     R[n][k] = 0
12             left = w_{N-n-1}
13             update(n, left, right)
14         p_1^N = w_1^N
15         k = c(w_1^N)
16         if k ≤ 4
17             ++T[N][k]
```

(a)

```
1  update(n, left, right)
2      s = statistics[n][right]
3      k = s.count
4      ℓ = s.left
5      if n != 1
6          if not_seen(n, right)
7              k = 0
8              ℓ = -1                  ▷ invalid word ID
9      if ℓ != left
10         ℓ = left
11         k = k + 1
12         if k == 1
13             ++R[n][1]
14         else
15             if 1 < k ≤ 5
16                 ++R[n][k]
17                 --R[n][k − 1]
```

(b)

Fig. 12. The **compute_left_extensions** and **update** functions.

```
1  not_seen(n, right)
2      s = statistics[n][right]
3      r = s.range
4      if r != ranges[n]
5          r = ranges[n]
6          return true
7      return false
```

Fig. 13. The **not_seen** function, which checks whether the *right* word was not seen in the current *range*.

do nothing. This update step takes $O(1)$ worst case and it is coded in the **update** function shown in Figure 12(b). We are sure to count correctly the number of left extensions because left words are seen in sorted order.

Figure 11 shows an example for the bi-gram AC. In this case, we have that C_{n-1} = A; thus, we need to scan all (consecutive) N-grams having an A as a context of length 1. These N-grams are those spanned by the light-green rectangle in Figure 11. In this example, AC can be extended to the left with words A and B as depicted in blue in the picture; thus, a(AC) = 2. Also observe that these two words, A and B, correspond to the children of the bi-gram CA in the reverse trie representation of the block shown in the upper part of Figure 16. We will return to this point in Section 5.1.4, when we discuss how to lay out efficiently the reverse trie.

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 n-grams $C_{n-1}x$.

When the context C_{n-1} changes (Line 7 in the pseudo-code of Figure 12(a)), then we would need a fast way to set all counts in the table to zero. Instead, we do not reinitialize the table explicitly, which would cost $\Theta(V)$ time, but we associate each context with an increasing identifier, as follows.

We store in the table *statistics* an identifier for each distinct word w_N, called *range* in the function **not_seen** of Figure 13, which represents the identifier of the range in which the word w_N was last

seen. We also keep track of the current range identifiers in an array $ranges[1, N-2]$. Now, 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 update its range identifier accordingly (Figure 13 and Lines 6–8 in the pseudo-code of Figure 12(b)).

Before concluding, there are two corner cases that we must mention for completeness: that of N-grams and that of 1-grams. The former must be mentioned because N-grams do not have modified counts; rather, their counts are equal to the raw frequency counts written in the input block B_N (Lines 15–17 in Figure 12(a)). The latter needs to be mentioned because their context is empty and we do not have to reinitialize their counts in the table when we switch range (`if` at Line 5 in the pseudo-code in Figure 12(b)).

Collecting Smoothing Statistics. We finally describe how we collect the smoothing statistics $t_{n,k}$ for $k = 1, \ldots, 4$ by using the introduced algorithm. For each order n, we maintain an array $R[1, 4]$, where $R[k]$ will store the quantity $|\{w_1^n : a(w_1^n) = k\}|$. A trivial solution scans the table used by the algorithm whenever we change context and updates the counters accordingly. This approach is clearly infeasible in terms of runtime. Instead, we can update each $R[k]$ in $O(1)$ on the fly during the **`update`** function of the algorithm, as follows. Whenever we increment the occurrence of w_N from k to $k+1$ (Line 11 in Figure 12(b)), we just have to check the value of k: if $k = 1$, then we only increment $R[1]$; otherwise, if $1 < k \leq 5$, then we increment $R[k]$ and decrement $R[k-1]$ (Lines 12–17 in the pseudo-code of Figure 12(b)).

Whenever we change context, the local counts accumulated in R are first combined with the global ones in another array T and are then reinitialized (Lines 9–11 in the pseudo-code in Figure 12(a)). Also, this recombining step takes constant time.

Finally, from the computed smoothing statistics, we can calculate the discount coefficients D_n using Equation (4). These are kept in an array $D[1, k]$, one for each order $1 \leq n \leq N$ and $k = 1, 2, 3$.

5.1.3 Normalization and Interpolation. The linear-time algorithm that computes the modified counts directly over a context-sorted block of N-grams can also be used to calculate pseudo-probabilities and backoff values using Equations (2) and (3), respectively, by just scanning B_N and using a direct-address table of size $\Theta(V)$ to read the modified counts.

Refer to the pseudo-code in Figure 14(a). In order to interpolate all of the different orders, we produce pseudo-probabilities and backoffs for all n-grams sharing the same context, starting from order 2 up to N. This guarantees that as soon as we compute $u(w_N|w_{N-n-1}^{N-1})$ for $2 \leq n < N$, we can directly interpolate it with $\mathbb{P}(w_N|w_{N-n}^{N-1})$ that has been already computed. Therefore, the function **`write`** in Figure 14(b) normalizes and interpolates all n-grams sharing the same context (there are *size* of them at each iteration of the loop). We now discuss some details about the pseudo-code.

We accumulate the interpolated probabilities of the n-grams sharing the same context in an array called *probabilities* and read them sequentially when needed to perform interpolation by using another array of *offsets*.

The body of the function consists of three loops. The loop in the Lines 4 to 6 calculates the numerator of the backoff for the context. The loop in Lines 8 to 11 calculates the denominator for normalized probabilities and backoffs. Finally, the loop in Lines 13 to 20 calculates the interpolated probabilities. As already observed, the case for the N-grams in Line 22 is identical to the general case for $n < N$, with the only difference being that the N-grams' counts are not modified but are the raw occurrence counts as seen in the text (see Figure 17(b)). Finally, for ease of presentation, Line 17 assumes that the uni-grams' probabilities are stored in the array *probabilities*[1]. Actually, a uni-gram probability $\mathbb{P}(w_n)$ can be computed in $O(1)$ when needed as illustrated in the pseudo-code in Figure 15; therefore, we do not need to buffer them into memory.

```
1  last(block, m)
2      iterators[1, N] = [0, 0]
3      while iterators[N] < m
4          for n = 2; n ≤ N; n = n + 1
5              i = iterators[n]
6              p_1^N = block[i]
7              size = 0
8              while i < m
9                  w_1^N = block[i]
10                 if w_{N-n}^{N-1} == p_{N-n}^{N-1}
11                     size = size + 1
12                     right = w_N
13                     left = w_{N-n-1}
14                     update(n, left, right)
15                 else
16                     break
17                 p_1^N = w_1^N
18                 i = i + 1
19             write(n, size)
```

(a)

```
1  write(n, size)
2      i = iterators[n], j = offsets[n]
3      b = 0, d = 0
4      for k = 1; k ≤ 3; k = k + 1
5          b += R[n][k] × D[n][k]
6          R[n][k] = 0
7      if n < N
8          for ℓ = i − size; ℓ < i; ℓ = ℓ + 1
9              w_1^N = block[ℓ]
10             if not_seen(n, w_N)
11                 d += statistics[n][w_N].count
12         b = b/d
13         for ℓ = i − size; ℓ < i; ℓ = ℓ + 1
14             w_1^N = block[ℓ]
15             k = statistics[n][w_N].count
16             u = (k − D[n][k])/d
17             p = u + b × probabilities[n − 1][j]
18             probabilities[n].add(p)
19             j = j + 1
20             lines 1-4 of Figure 17a
21     else
22         lines 1-13 of Figure 17b
23     offsets[n + 1] = 0
```

(b)

Fig. 14. The **last** step of estimation and the **write** function that performs normalization, interpolation, and indexing.

```
1  unigram_prob(w_n)
2      k = statistics[1][w_n].count
3      u = (k − D[1][k])/m_2
4      p = u + b(ε)/V
5      return p
```

Fig. 15. Final interpolated probability for the uni-gram w_n. The denominator for the quantity u is equal to the number of bi-grams in the text, called m_2.

In conclusion, normalization and interpolation are carried on as explained for the 3-Sort algorithm (see Section 3.2.1) 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, 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 tsorting from context to suffix* in order to perform interpolation, thus eliminating the sorting step 3 of Figure 10. Summing up, given that we have formerly shown how to save the sorting from suffix to context as well (Section 5.1.2), we have completely eliminated the round-trip of 3-Sort mentioned at the beginning of Section 5.1.

5.1.4 Joining and Indexing. We now show how to perform the two remaining steps of estimation: first, the joining of probabilities with backoff values and, second, the building of the reverse trie data structure during the same pass.

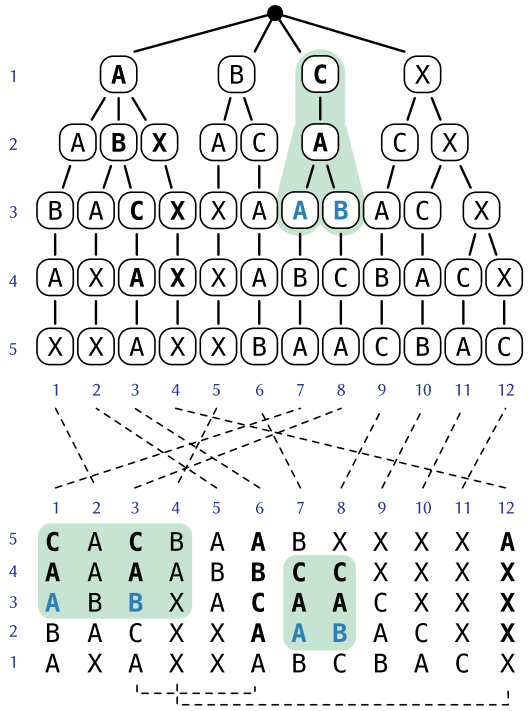


Fig. 16. The 5-gram block sorted in context order of Figure 11 in relation to its reverse trie representation. The bottom level of the trie, [X, X, A, X, X, B, A, A, C, B, A, C], is obtained by permuting the first words of the strings in the context-sorted block, [A, X, A, X, X, A, B, C, B, A, C, X], according to the lexicographic position of their last words, [C, A, C, B, A, A, B, X, X, X, X, A]. The left extensions (words in blue) of AC correspond to the children of CA in the reverse trie representation.

We recall that 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 Section 4.1.1. In particular, it is the *reverse* trie variant, such as the one depicted in Figure 16, because it optimizes the left-to-right pattern of lookups performed by perplexity scoring (see the perplexity benchmark in Section 4.3.3).

For this problem, we exploit the property already mentioned in Section 5.1.2: *every N-gram prefix of length $N-1$ must be matched at position 2 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 the first guarantee to build the reverse trie data structure and the second one to perform joining of probabilities with backoffs. By looking at Figure 16, which shows the context-sorted block of Figure 11 in relation to its reverse trie representation, we can graphically visualize these two guarantees. Let us discuss them separately.

Regarding guarantee (1), we can immediately see that the first 4 levels of the trie are, indeed, the contexts of length 4 of the 5-grams in the context-sorted block. For example, the prefix of length 4 of ACBAC, that is, ACBA, is found in the 6th string; the one of XXXAB is found in the 12th string

instead (following the dashed lines at the bottom of Figure 16). Note that we always find the match at position 2; thus, the first 4 levels of the trie store those prefixes. In general, we have that the first $N-1$ levels of the reversed trie *are the prefixes* of size $N-1$ of the context-sorted N-grams and, therefore, can be efficiently built directly from the context-sorted N-grams without having to sort the N-grams in suffix order.

Regarding guarantee (2), consider the first 5-gram ABAAC. Since interpolation produces the probabilities for all of the suffixes—that is, for C, AC, AAC, and BAAC—*we compute the backoffs for their contexts*—that is, $b(\varepsilon)$, b(A), b(AA), and b(BAA), respectively—which appear in sorted order in the block. Refer to Figure 1 for a graphical example. Backoffs are, therefore, computed in suffix order and can be written directly in the corresponding trie nodes.

Now that we know how to efficiently build the first $N-1$ levels of the reversed trie and perform joining, we are left to consider two problems: first, how to handle the bottom level of the trie and, second, how to write the interpolated probabilities in the nodes of the trie. Note that regarding the first problem, we cannot build the bottom level of the trie directly because a context of length $N-1$ does not extend to the left. Regarding the second problem, interpolation produces the probabilities for the *suffixes* but we would need the ones for the *contexts* in order to write them in the trie as we can do for the backoffs. We clarify this latter point by continuing the example for ABAAC. We interpolate its constituent n-grams in the following (suffix) order: C, AC, AAC, BAAC, and ABAAC, but we would actually need the probabilities for the contexts A, AA, BAA, and ABAA in order to write them in the suffix trie (as done for the backoffs).

Exploiting the Relation Between Context and Suffix Order. To efficiently solve these two remaining problems, we exploit the following property that establishes the relation between context and suffix order: *A context-sorted block can be sorted efficiently in suffix order by considering the order on the last word only*, because the prefixes of length $N-1$ are already sorted.

In turn, this property implies that the following: *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 16 by the dashed lines linking the context-sorted 5-grams with the corresponding root-to-leaf paths in the reverse trie. For example, consider the first 5-gram ABAAC. We know that this 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 in the bottom level is the lexicographic position of the last word, that is, the C. Since the lexicographic position of the C is 7 within all of the last words of the 5-grams (4 As and 2 Bs first), A is placed in position 7 in the last level of the trie (follow the dashed line from position 1 in the context-sorted block to position 7 in the trie).

In order to place word identifiers and probabilities in the correct position, we use a *count-indexing technique*. For each vocabulary word, we maintain the number of times that it appears as the last word of an N-gram in a direct-address table of size $\Theta(V)$. Prefix-summing such counts (shifted by one position to the right) gives us in $O(1)$, for each distinct word identifier w_N, the position in the array that represents the bottom level of the trie at which we have to write the first occurrence of w_N. Given this position, we write the integer w_N in $O(1)$ and increment the position in the table by one. Note that this is the same procedure used by counting sort; thus, the correctness of the approach follows automatically (see Section 8.2 of [13]). It requires only V integer counters, which we store in an array *positions*$[1, N]$.

Let us consider a complete example. Refer to Figure 16 and the pseudo-code in Figure 17(b). For the uni-grams A, B, C, and X, we count how many times they appear as last words of the N-grams and obtain the following counts $[4, 2, 2, 4]$, because A and X appear 4 times each while B and C

```
1 if not_seen(n, w_N)
2     pos = positions[n][w_N]
3     levels[n][pos].prob = p
4     pos = pos + 1
```

(a)

```
1  for ℓ = i − size; ℓ < i; ℓ = ℓ + 1
2      w_1^N = block[ℓ]
3      d += c(w_1^N)
4  b = b/d
5  for ℓ = i − size; ℓ < i; ℓ = ℓ + 1
6      w_1^N = block[ℓ]
7      k = c(w_1^N)
8      u = (k − D[n][k])/d
9      p = u + b × probabilities[N − 1][j]
10     pos = positions[N][w_N]
11     levels[N][pos].prob = p
12     levels[N][pos].word = w_1
13     pos = pos + 1
```

(b)

Fig. 17. The pseudo-code that illustrates how to perform indexing for the case $n < N$ in (a) and for the case $n = N$ in (b). The two listings complete the pseudo-code in Figure 14(b).

appear twice each. Now, we prefix sum the counts[5], obtaining [5, 7, 9, 13], and we shift them one position to the right, obtaining the following initial $positions[5][4] = [1, 5, 7, 9]$.

Now, consider the first 5-gram in the context-sorted block, ABAAC. Since its last word is C, we look at its initial position in the array, which is 7, and we know that we have to place its first word, A, at position 7 in the last level of the trie. This is done in Line 9 of the pseudo-code. As a matter of fact, the 7th string in the reverse trie of Figure 16 is exactly ABAAC. Then, we know that the second occurrence of C (last word of ACBAC) will give us position 7 + 1 = 8. Thus, we will write an A in position 8. Let us now consider the second N-gram, XABAA. The position associated to A is 1; thus, we have to write the first word X at position 1. We repeat the process for all N-grams in the context-sorted block: following the dashed lines of Figure 16, it is easy to see that the last level of the trie can be built correctly by the introduced algorithm. The corresponding pseudo-code is illustrated in Figure 17(b) and it represents the case for $n = N$ in the **write** pseudo-code in Figure 14(b) (Line 22).

The same technique is also used to place the final probabilities in the correct trie nodes for all orders $1 < n \leq N$. Let us consider a full example for $n = 2$ in order to explain how this is possible. For the uni-grams A, B, C, and X, we obtain the following counts [3, 2, 1, 2]. In fact, although A appears 4 times, it appears in only 3 distinct contexts, to the right of the bi-grams AA, BA (that appears twice) and XA. Instead, B appears twice: once to the right of AB and to the right of CB. As done before, prefix-summing and shifting the counts, we obtain the initial $positions[2][4] = [1, 4, 6, 7]$. Now, consider the first 5-gram ABAAC. When we produce the final interpolated probability for AC, we have to write it in the second level of the trie in position 6 as given by the corresponding counter in the array. Again, we can immediately verify that the (6+1)th root-to-leaf path in the trie is the one spelling out CA. For the second 5-gram XABAA, instead, we have to write the probability of AA at position 1 in the second level of the trie.

The examples above can be easily extended to any other order $2 < n \leq N$. In this case, the corresponding pseudo-code is illustrated in Figure 17(a) and completes the **write** function coded in Figure 14(b) (Line 20).

[5]And also sum 1 because our examples use 1-based indexes.

Table 6. Number of n-grams for the Datasets Used in the Experiments

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	1,264,975,112	3,314,051,350	5,058,407,983

Finally, we also have to write the pointers for each node of the trie. However, observe that a pointer represents the number of successors of a given n-gram; thus, pointers are the same as the modified counts. Therefore, pointers require no extra effort (and are not shown in the pseudo-code for simplicity).

5.2 Experiments

The experiments discussed in this section first analyze the runtime of our solution, the 1-Sort algorithm, then introduce optimizations, and, finally, consider the comparison against the 3-Sort approach.

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

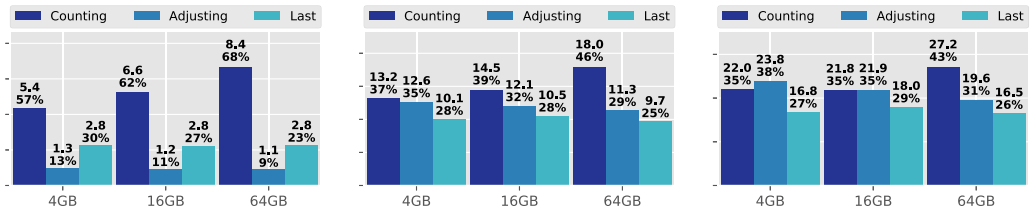
- 1BillionWord is the concatenation of all the news files contained in the training directory of the dataset described in [7] and is publicly available at http://www.statmt.org/lm-benchmark.
- Wikipedia17 is a recent Wikipedia dump, collected from October to December 2017 and publicly available at https://dumps.wikimedia.org/enwiki/latest.
- ClueWeb09 is a sampling of 5 million pages drawn from the ClueWeb 2009 TREC Category B test collection, consisting of English webpages 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, as already done for experiments presented in Section 4.3. 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, 64b. 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++14, whose code is freely available at https://github.com/jermp/tongrams. As our competitor, we use the C++ implementation of 3-Sort as provided by the authors of [24] and available at http://kheafield.com/code/kenlm. We refer to this implementation as KenLM, which is the lead toolkit for language modeling [23].

Both implementations were compiled with gcc 5.4.0, using the highest optimization setting, that is, with compilation flags `-O3` and `-march=native`.



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

Fig. 18. Time in minutes spent at each step of estimation by using different amounts of internal memory.

5.2.1 Preliminary Analysis. As a first set of experiments, we show the runtime of our algorithm by varying the amount of internal memory and by inspecting the CPU and I/O activity.

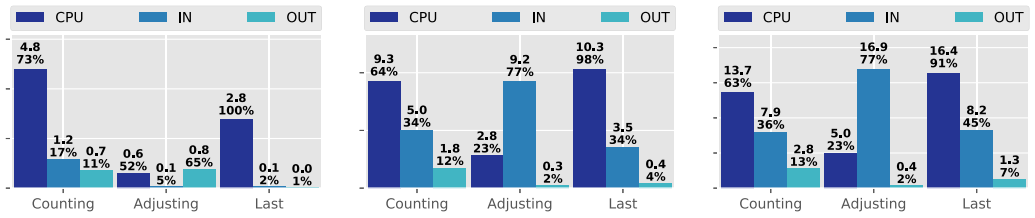
Varying the Amount of Internal Memory. We show the runtime of our algorithm at each step of estimation by varying the allowed amount of internal memory among 4GB, 16GB, and the maximum available RAM, 64GB. This experiment aims at showing what steps are the most expensive and fixes the amount of internal memory that we will use for the subsequent analysis. The plots in Figure 18 illustrate the results. Above each bar, we report two numbers: the first indicating the number of minutes spent during the step and the second indicating the percentage with respect to the total runtime of the algorithm. This grand total measures the time of the whole estimation process, that is, 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, we can observe that, not surprisingly, the size of the language model has a significant impact not only on the total runtime but also on which step becomes the most expensive. In fact, while on the 1BillionWord dataset, the Counting and the Last steps contribute to more than 80% of the total runtime 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 runtime is almost evenly distributed across the three steps. Note that, in particular, 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 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 runtime because using larger hash sets also means sorting larger blocks of N-grams. Observe that the total runtime of Counting (slightly) increases by increasing the amount of memory. However, as we have discussed above, using more memory for sorting implies fewer 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 buffers does not impact the runtime.

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 of 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 runtime of each step. Under a different perspective, this analysis is



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

Fig. 19. Time in minutes spent by CPU computation and I/O activity at each step of estimation.

useful to understand how disk usage is impacted by the size of the language model. The plots in Figure 19 illustrate such impact, that is, 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 this reason, we use asynchronous threads to handle I/O operations so that while the CPU is performing internal processing, data is read or written to disk simultaneously [16]. 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 pay only the maximum of the two. Consequently, the sum of three percentages for CPU, IN, and OUT time for a given step in Figure 19 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 runtime of Counting, whereas the OUT time is completely overlapped with CPU processing.

The total runtime 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 for the disk for input operations, thus occurring in more disk seeks [53]. 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 Section 5.1.2 for computing the left extensions over a context-sorted block is very small compared with the overall runtime 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 the disk, the CPU is processing the previous block. Therefore, we have a good overlap between CPU and reading time from the disk. This is possible because disk reads are issued to a single source, that is, 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.

5.2.2 Optimizing Our Solution. In this section, 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 runtime of the Counting step, it is important to guarantee a good overlap between input scanning and sorting in order to pay only the maximum of the two latencies and not the sum of the two. For this reason, we use LSD *radix sort* [13] 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 32b numbers; thus, N passes of counting sort, that is, one for each word index j, $j = N-1, 0, 1, \ldots, N-2$, 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 jth word identifier.

Thanks to this strategy and by using all available cores on our test machine ($K = 4$), the time for the Counting step improves substantially[6] 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×).

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 runtime 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* (FC) [55] representation of an 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, that is, as many as can be possibly 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 word identifiers that we have to copy from the previous record (in context order) and s is the remaining part of the string (the suffix). 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 28B per record of the uncompressed version, we pass to an average of 9B per record on Wikipedia17 and to 9.75B per record on ClueWeb09. As a net result, the Adjusting step on Wikipedia17 and ClueWeb09 runs 2× and 1.5× faster, respectively. 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, note that the CPU time rises as

[6]During our experimentation, 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 an N-gram block of 8GB, the gnu::parallel_sort takes 30s while our parallel LSD radix sort takes 16.4s.

Table 7. The Effect of Compressing Blocks During the Adjusting Step on Wikipedia17 and ClueWeb09 Datasets

(a) Wikipedia17

	CPU	IN	total	bytes/gram
Uncompressed	**2.81**	9.24	12.05	28.00
FC bit-aligned	5.77 (0.5×)	**0.10** (97×)	5.86 (2×)	**9.00** (3×)
FC byte-aligned	3.94 (0.7×)	1.22 (8×)	**5.03** (2.4×)	11.00 (2.5×)

(b) ClueWeb09

	CPU	IN	total	bytes/gram
Uncompressed	**4.98**	16.91	21.89	28.00
FC bit-aligned	9.29 (0.5×)	5.25 (3×)	14.55 (1.5×)	**9.75** (3×)
FC byte-aligned	7.61 (0.7×)	**4.23** (4×)	**11.55** (2×)	11.65 (2.4×)

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.

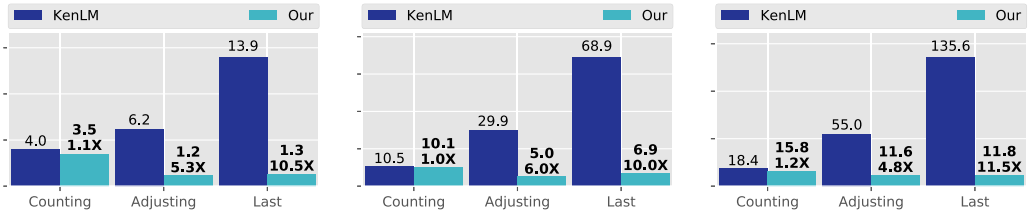
well, roughly 2×, owing to decoding from a compressed stream: we trade CPU time for less reading from disk.

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 slightly worse compression (2.5×, on average, compared with 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 runtime of the step (see Figure 19(a)).

Last: Processing N-gram Blocks in Parallel. As discussed in Section 5.2.1, 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 step described in Section 5.1.3 on its own block (the `last` function in the pseudo-code of Figure 14(a)), 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 be written by a worker thread. For this problem, we use a 2-step algorithm: 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 that $K = 4$; thus, we use 3 worker threads and 1 reader thread. On 1BillionWord, we reduce the runtime from 2.8 to 1.33 minutes (2.1×), on Wikipedia17 from 10.53 to 6.85 minutes (1.54×), and on ClueWeb09 from 18 to 11.8 minutes (1.52×).

5.2.3 Overall Comparison. In this final section, 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 20. The plots strictly confirm the thesis of this article. The round-trip performed by 3-Sort, that is, the sorting from suffix to context and then back from context to suffix (see Figure 10), results in a severe



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

Fig. 20. Time in minutes spent by KenLM and our algorithm at each step of estimation.

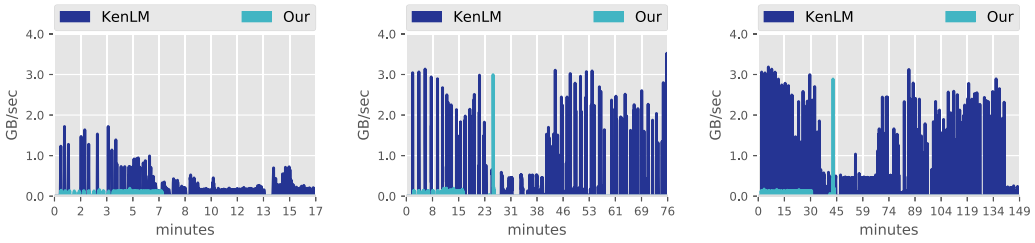
penalty on the total runtime of the estimation process: our improved 1-Sort algorithm exploits the properties of the extracted N-gram strings in order to completely avoid the round-trip. Overall, this makes our approach run 4×, 4.9×, and 5.3× faster than KenLM, respectively, on 1BillionWord, Wikipedia17, and ClueWeb09. Let us now discuss each step separately.

As already commented in Section 5.1.1, the first step of Counting is performed similarly by the two algorithms; this is why the corresponding runtimes 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.

During the Adjusting step, our approach computes the modified counts in context order 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 recomputes the modified counts during the process of normalization itself, we need to handle only 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 to be, on average, 5.3× slower during this step than our approach.

During the Last step, normalization, interpolation, and indexing are performed (Sections 5.1.3 and 5.1.4). Again, we can observe an average speedup 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 [24]). To ensure fairness, the indexing time for KenLM is measured by excluding the time to write and parse the intermediate (ARPA) file on disk: in any event, it is a significant amount of the total runtime 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 exploit 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 Equations (2) and (3). Again, recall that we need to tackle only 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 21 shows the number of GB written per second on disk for each dataset. We collect the



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

Fig. 21. Gigabytes per second written on disk by KenLM and our algorithm.

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 match the results shown in Figure 20, that is, not surprisingly, the improvement in runtime is directly proportional to the quantity of data written to disk. In fact, the area below the curve of our algorithm is ≈6× less than the one of KenLM (20.4GB vs. 124.5GB) on 1BillionWord; ≈5× less on Wikipedia17 (63.6GB vs. 310.7GB), and ≈5.8× less on ClueWeb09 (88GB vs. 514GB).

6 CONCLUSIONS

In this article, we studied in depth the two problems lying at the core of language model applications: providing random access to n-gram probabilities and estimating such probabilities from large textual collections. We focused on solving these two problems *efficiently*.

Concerning the problem of indexing, we presented highly compact and fast indexes that achieve substantial performance improvements over the state-of-the-art approaches. In particular, our trie data structure exploits the succinctness of the Elias-Fano encoding by preserving the query processing speed of the fastest implementation in the literature. We also introduced a context-based remapping technique for vocabulary tokens to further compress the trie data structure. On average, this technique improves compression by 28% with a context of length 1 and by 35% with a context of length 2, with only a slight penalty at query-processing speed.

Concerning the problem of estimation, we described 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 the compressed trie data structure indexing the strings in *suffix* order. Our algorithms run 4.5× faster than the fastest state-of-the-art approach. The improved performance of the algorithm derives from the exploitation of the properties of the extracted N-gram strings that relate context and suffix order and that are neglected by competitive approaches.

ACKNOWLEDGMENTS

We thank Roberto Trani for his assistance with the textual collections of Wikipedia17 and ClueWeb09.

REFERENCES

[1] 2006. Yahoo! N-Grams, version 2.0. Retrieved January 16, 2019 from 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'11)*. 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'14)*. 352–361.
[4] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM* 13, 7 (1970), 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'07)*. 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*, A. Neustein and J. A. Markowitz (eds.). Springer. 197–229.
[9] Stanley Chen and Joshua Goodman. 1996. An empirical study of smoothing techniques for language modeling. In *Association for Computational Linguistics (ACL)*. 310–318.
[10] Stanley Chen and Joshua Goodman. 1999. An empirical study of smoothing techniques for language modeling. In *Computer Speech and Language*, 13, 4 (1999), 359–394.
[11] Wenlin Chen, David Grangier, and Michael Auli. 2015. Strategies for training large vocabulary neural language models. In *Preprint arXiv:1512.04906*.
[12] David R. Clark and J. Ian Munro. 1996. Efficient suffix trees on secondary storage. In *Symposium on Discrete Algorithms (SODA'96)*. 383–391.
[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). MIT Press.
[14] Bruce Croft, Donald Metzler, and Trevor Strohman. 2009. *Search Engines: Information Retrieval in Practice* (1st ed.). Addison-Wesley Publishing Company.
[15] Erik D. Demaine, Thouis Jones, and Mihai Pătraşcu. 2004. Interpolation search for non-independent data. In *Symposium on Discrete Algorithms (SODA'04)*. 529–530.
[16] Roman Dementiev, Lutz Kettner, and Peter Sanders. 2008. STXXL: Standard template library for XXL data sets. *Software, Practice and Experience* 38, 6 (2008), 589–637.
[17] Peter Elias. 1974. Efficient storage and retrieval by content and address of static files. *Journal of the ACM* 21, 2 (1974), 246–260.
[18] Robert Mario Fano. 1971. On the number of bits required to implement an associative memory. In *Memorandum 61, Computer Structures Group, MIT*. MIT, Cambridge, MA.
[19] 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'06)*. 94–101.
[20] Marcello Federico, Nicola Bertoldi, and Mauro Cettolo. 2008. IRSTLM: An open source toolkit for handling large scale language models. In *INTERSPEECH*. 1618–1621.
[21] Edward Fredkin. 1960. Trie memory. *Communications of the ACM* 3, 9 (1960), 490–499.
[22] Kimmo Fredriksson and Fedor Nikitin. 2007. Simple compression code supporting random access and fast string matching. In *Workshop on Experimental Algorithms (WEA'07)*. 203–216.
[23] Kenneth Heafield. 2011. KenLM: Faster and smaller language model queries. In *Workshop on Statistical Machine Translation (WMT'11)*. 187–197.
[24] 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.
[25] Steffen Heinz, Justin Zobel, and Hugh E. Williams. 2002. Burst tries: A fast, efficient data structure for string keys. *Transactions on Information Systems* 20, 2 (2002), 192–223.
[26] 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'11)*. 127–136.
[27] Guy Jacobson. 1989. Space-efficient static trees and graphs. In *Foundations of Computer Science (FOCS'89)*. 549–554.
[28] Dan Jurafsky and James H. Martin. 2014. *Speech and Language Processing*. Pearson.
[29] Reinhard Kneser and Hermann Ney. 1995. Improved backing-off for m-gram language modeling. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP'95)*, Vol. 1. 181–184.
[30] Philipp Koehn. 2005. Europarl: A parallel corpus for statistical machine translation, http://www.statmt.org/europarl. In *MT Summit*. 79–86.
[31] Grzegorz Kondrak. 2005. N-gram similarity and distance. In *International Symposium on String Processing and Information Retrieval (SPIRE)*. Springer, 115–126.

[32] Karen Kukich. 1992. Techniques for automatically correcting words in text. *ACM Computing Surveys* 24, 4 (1992), 377–439.
[33] Ted G. Lewis and Curtis R. Cook. 1988. Hashing for dynamic and static internal tables. *Computer* 21, 10 (1988), 45–56.
[34] Bhaskar Mitra and Nick Craswell. 2015. Query auto-completion for rare prefixes. In *International Conference on Information and Knowledge Management (CIKM'15)*. 1755–1758.
[35] 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'14)*. 1055–1058.
[36] Alistair Moffat and Lang Stuiver. 2000. Binary interpolative coding for effective index compression. *Information Retrieval Journal* 3, 1 (2000), 25–47.
[37] Donald R. Morrison. 1968. PATRICIA: Practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM* 514–534.
[38] Gonzalo Navarro, Ricardo A. Baeza-Yates, Erkki Sutinen, and Jorma Tarhio. 2001. Indexing methods for approximate string matching. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*. 19–27.
[39] Patrick Nguyen, Jianfeng Gao, and Milind Mahajan. 2007. MSRLM: A scalable language modeling toolkit. In *Microsoft Research MSR-TR-2007-144. 2007.*
[40] Giuseppe Ottaviano and Rossano Venturini. 2014. Partitioned Elias-Fano indexes. In *International Conference on Research and Development in Information Retrieval (SIGIR'14)*. 273–282.
[41] Adam Pauls and Dan Klein. 2011. Faster and smaller *N*-gram language models. In *Association for Computational Linguistics (ACL)*. 258–267.
[42] 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'17)*. 615–624.
[43] Giulio Ermanno Pibiri and Rossano Venturini. 2018. Inverted index compression. In *Encyclopedia of Big Data Technologies* (2018), 1–8.
[44] 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'03)*. 388–391.
[45] David Salomon. 2007. *Variable-length Codes for Data Compression*. Springer.
[46] Jangwon Seo and W. Bruce Croft. 2008. Local text reuse detection. In *International Conference on Research and Development in Information Retrieval (SIGIR'08)*. 571–578.
[47] 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'15)*. 2409–2418.
[48] 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* 4, 1 (2016), 477–490.
[49] Andreas Stolcke. 2002. SRILM - an extensible language modeling toolkit. In *International Conference on Spoken Language Processing (ICSLP'02)*. 901–904.
[50] David Talbot and Miles Osborne. 2007. Randomised language modelling for statistical machine translation. In *Association for Computational Linguistics (ACL)*. 512–519.
[51] Larry H. Thiel and H. S. Heaps. 1972. Program design for retrospective searches on large data bases. *Information Storage and Retrieval* 8, 1 (1972), 1–20.
[52] Sebastiano Vigna. 2013. Quasi-succinct indices. In *International Conference on Web Search and Data Mining (WSDM)*. 83–92.
[53] Jeffrey Scott Vitter. 1998. External memory algorithms. In *European Symposium on Algorithms (ESA'98)*. 1–25.
[54] Taro Watanabe, Hajime Tsukada, and Hideki Isozaki. 2009. A succinct *N*-gram language model. In *International Joint Conference on Natural Language Processing (IJCNLP'09)*. 341–344.
[55] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann.
[56] Susumu Yata. 2011. Prefix/Patricia trie dictionary compression by nesting Prefix/Patricia tries. In *International Conference on Natural Language Processing (NLP'11)*.

Received June 2018; revised November 2018; accepted December 2018