Ph.D. Thesis

Space and Time-Efficient Data Structures for Massive Datasets

by

Giulio Ermanno Pibiri

Supervisor
Rossano Venturini

Referee
Daniel Lemire

Referee
Simon Gog

Ph.D.
in
Computer Science

Department of Computer Science
University of Pisa, Italy

2018

Abstract

This thesis concerns the design of compressed data structures for the efficient storage of massive datasets of integer sequences and short strings. The studied problems arise in several real-world scenarios, such as inverted index engineering, thus a consistent part of the thesis focuses on the experimental analysis of the proposed, practical, solutions. All the code used in the experimentation is publicly available in the hope of being useful for further research.

For the problem of representing in compressed space the sorted integer sequences of inverted indexes, we propose three different solutions showing new interesting space/time trade-offs.

The first proposal shows that *clustering* the inverted lists yields more compact indexes at the expense of a moderate slowdown in query processing speed. The core idea is that clusters of *similar* inverted lists, i.e., the ones sharing many integers, are compressed together, thus permitting to reduce their redundancy.

The second proposal describes an optimal, linear-time, algorithm that splits an inverted list in variable-size partitions in order to minimize its space when the *Variable-Byte* encoding is used to compress its partitions. The proposed algorithm is fast and yields indexes that are more than twice smaller than regular Variable-Byte, without compromising their speed efficiency.

The third proposal exploits the repetitiveness of the d-gapped inverted lists to achieve compact storage and very fast decoding speed. Specifically, we show that a *dictionary* of repeated integer patterns can be built to compress the inverted lists. This permits to represent the inverted lists as *references* to the dictionary's patterns, thus allowing the two-fold advantage of: encoding many integers with a single reference identifier; decoding many integers with a single dictionary access.

We then take into account strings of at most n words, i.e., n-grams and address the two fundamental problems of: *indexing* large and sparse n-gram datasets; *estimating* language models from a large textual collection.

For the problem of indexing, we introduce a compressed trie data structure where the identifier of a word appearing after a *context* of k preceding words is represented as an integer whose value is proportional to the number of words that can follow *only* such context. Since the number of words following a given

context is small for natural languages, the data structure achieves very compact storage and fast random access to the n-gram satellite values.

For the problem of estimation, we design an algorithm for estimating Kneser-Ney language models in external memory, which have emerged as the de-facto choice for language modelling. We show that exploiting the relation between *context* and *suffix* order of the extracted n-gram strings significantly improves the running time of the algorithm.

From a theoretical perspective, we also study the fascinating problem of representing *dynamic* integer dictionaries in succinct space. Many solutions for this problem are known to implement the operation of insertion, deletion and access to individual integers in optimal time, yet without posing any space bound on the problem. We exhibit, instead, a data structure that matches the optimal time bounds for the operations and, at the same time, achieves almost optimal compressed space by resorting on the succinctness of the *Elias-Fano* integer encoding.

Acknowledgments

I come writing this page after a long journey of three years of academic research that undoubtedly changed me in a professional way, as well as personal. This page is meant to express my sincere gratitude to all people I have interacted with and made this change possible.

The first words of gratitude are dedicated to Rossano Venturini, who has proven to be a great mentor and the best supervisor. He taught me how to do high-quality research, how to not give up in critical moments and how to always push forward my limits and expectations. Moreover, I am excited to know that we will have a lot more to share in the future.

I am also grateful to the referees for all the useful comments and suggestions.

A special thank is dedicated to the people from the HPC-lab, at the ISTI-CNR of Pisa. This goes especially to Nicola Tonellotto, Franco M. Nardini and Raffaele Perego for their kind support.

I am very grateful for the opportunity I had to visit both Japan and Australia during my last year of Ph.D., therefore a special thank goes to Yasuo Tabei for having hosted me at Riken AIP and to Alistair Moffat and Matthias Petri for having welcomed me at the University of Melbourne. I also have to thank Shane Culpepper (RMIT University) for having contributed in funding my visit to Australia.

These three years would not have been the same without the support of all my dear friends in Pisa, Florence, Tokyo and Melbourne. This is dedicated to: Marco, Luca, Giovanna, Laura, Alessandro, Shunsuke, Bonnie, Adrián, Joanna, Brian, Takahiro, Ryo, Cham and many others (this list is by no means exhaustive!).

Last but not least words of gratitude are for where heart is. Giovanni, Isabella and Fiammetta are the best family one could ever dream on. Their unconditional love, wise advices and support are my life-guide, forever. The formidably strength and respect they provide me is a unique gift. These are the people with whom it is worth spending a life, therefore this thesis is dedicated to them.

Florence, Italy
October, 2018

Giulio Ermanno Pibiri

Contents

1 Introduction

The incredibly fast growth of hardware and software technologies in recent years has radically changed the way data is stored and processed. On the one hand, advancements in hardware that are now in widespread use, such as multicore architectures, GPUs and larger internal memories, permit the treatment of datasets far beyond the capabilities of a decade ago. On the other hand, the ubiquitous availability of devices like smart phones, tablets and personal computers, have caused a critical explosion of information to be stored, analyzed and indexed. As a result, nowadays computations are more data-intensive than ever. This evidence imposes a severe limitation: the increase of information does *not scale* with technology. Therefore, we will always reach the point at which hardware can only help but *not* solve our problems. Indeed Niklaus Wirth observed that: "*Software is getting slower more rapidly than hardware becomes faster*" [167].

The main reason lies in the fact that current computer architectures have a *hierarchical memory system*, i.e., multiple layers with different access latency and size (Table 1). Layers that are closer (lower) to the processor are small but are extremely fast. Conversely, higher layers are much bigger but also much slower. A typical organization, shown in Figure 1, includes one or more level of caches (L1, L2 and, often, L3), an internal memory (RAM) and a disk (referred to as external memory). There are still orders of magnitude of difference in accessing speed between consecutive memory layers, as we can see from Table 1a (the numbers reported in the table are taken from *Latency Numbers Every Programmer Should Know* [46], see also [54]). For example, while a L1 cache reference may only costs 1 nanosecond, a RAM reference costs 100 nanoseconds and a random disk read a few (3-4) milliseconds.

Differently, processors (indicated as the CPUs in Figure 1) are fast: a single *cache miss*, i.e., the time spent in fetching (say) 64 bytes from main memory to L1 cache, could suffice to execute *hundreds* of CPU operations. This discrepancy in speed between modern processor and memory technology is known in the literature as the *I/O bottleneck* and it ultimately implies the following facts:

- it makes sense to *trade instructions for memory reads/writes*;
- *cache-aware* algorithms are substantially faster than non cache-aware ones;
- *if the data fits in main memory* and no disk usage is needed, the computation can be speeded up by several order of magnitudes.

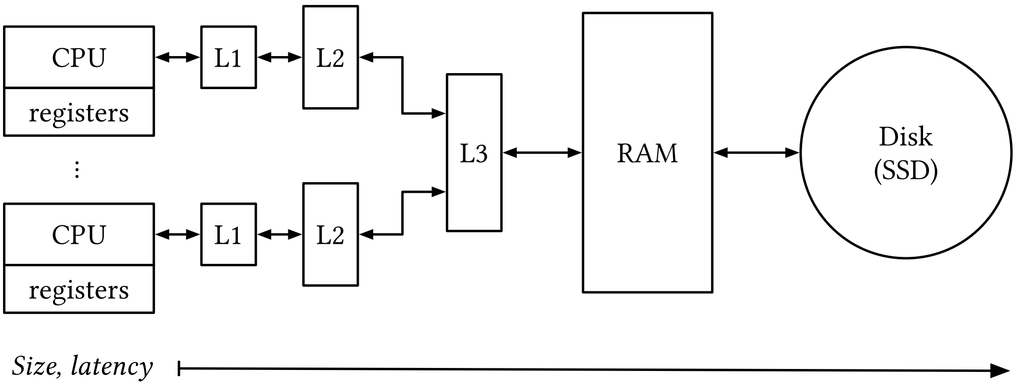


Figure 1: The computer memory hierarchy. From left to right, the memory size increases but latency increases as well.

Event	10^{-9} secs
L1 cache reference	1
L2 cache reference	4
RAM reference	100
SSD random read	16,000
Disk random read	3,000,000

(a)

Memory type	Size	
registers	64	bits
L1	32	KB
L2	1	MB
L3	30	MB
RAM	64	GB
Disk	1	TB

(b)

Table 1: Example of latency time in nanoseconds for various hardware events in **(a)** and memory sizes for a server machine in **(b)**.

In this scenario, *data compression* appears as mandatory since it can introduce a twofold advantage over non-compressed data: transfer more bytes from slower memory levels into faster ones and, *hence*, improve the speed of algorithms. In other terms, more data can be stored within the same memory budget, thus enabling the handling of larger datasets. This could make the difference for the very same algorithm to work in internal memory (compressed case) or in external memory (uncompressed case). However, we stress the fact that data compression is only advantageous as long as the decompression/access time is (a lot) faster than the time needed to fetch the data itself from slower memories. In fact, while general-purpose compression utilities such as gzip permit to compress virtually any source of data, applying data compression to data structures require a non-trivial engineering effort, especially when fast random access is needed.

These considerations highlight the challenge addressed in this thesis: how to design *compressed* data structures that support *fast* access to data. In particular, we consider two types of data: *integer sequences* and *strings*, because these arise in many practical applications, such as in the ones that we are going to discuss next.

The motivation for this work finds its origin in the observation that the engineering of data structures, e.g., inverted indexes, tries, B-trees and hash tables just to name a few, is what boosts the availability and wealth of the information around us, making its access more economical by saving computer resources. In particular, as we have already argued above, the study of this engineering has noticed that there is a strict correlation between time and space optimizations, now more than ever. The impact of such optimizations is broad, far reaching and, not least, usually implies substantial economic gains. To better understand what we mean, we discuss some meaningful real-world examples in the following.

Consider a hash table T with two hash functions, h_1 and h_2. During a search, if the key k is not found at the position $p_1 = h_1(k)$ we have to jump unconditionally to position $p_2 = h_2(k)$ which may not be in the cache. This algorithm is likely to incur in many cache misses. To improve its spatial locality, we could try to store the key in any location available in $T[p_1, p_1 + s]$ if $T[p_1]$ is already occupied (if all such positions are occupied, we try the same strategy but starting at position p_2). If the span s is chosen such that the space taken by $T[p_1, p_1 + s]$ fits in a cache line, accessing the table at position p_1 will likely fetch into the cache the searched key. Moreover, if the keys are stored compressed, we can afford larger values of s, thus improving the spatial locality *and* access time. This optimization effectively doubled the speed of RocksDB (Facebook) Cuckoo Hashing Table Format [84].

Firefly is the Dropbox's full-text search engine. During 2016, Firefly's engineers saw the query search latency deteriorating from 250 ms to 1000 ms as they kept adding new users to the system [82]. They found that the reason was an increase in I/O operations per second on the machines hosting their inverted index. In fact, the index was not loaded in internal memory because its space was too big, rather it was accessed directly from the disk. They solved the problem by compressing the index (by 33% using a variable-byte scheme called *Varint*) as this greatly reduced the I/O pressure on the server machines, bringing them back to 250 ms. In particular, the energetic cost of a massive disk usage is the main cost in modern storage architectures. This is also why, on July 14, 2016, Dropbox announced a new encoding algorithm for images, *Lepton* [83], which saves 22% of space without any loss and able of decompressing at 15 MB/sec. Indeed, as the post reports [83], Lepton is saving multiple petabytes of space and, thus, reducing the number of machines as well as the electricity bill.

S2 Geometry [85] is the geographic information systems developed at Google that represents all data on a three-dimensional sphere by first mapping the Earth surface onto the fix faces of a cube and, then, placing all points on a face along a space-filling curve (in their implementation, along a Hilbert curve). Using a space-filling curve implies that points that are close on the Earth surface will be placed close together in a two-dimensional space, e.g., a sequence of integers. Again, a higher spatial locality of accesses can be achieved in this case with improved performance of proximity queries in spatial databases, such as Google Maps.

Another crucial application which has demanded exceptionally fast random access and good storage requirements is the representation of *language models* that are, essentially, dictionaries of (short) strings. Their usage is widespread in information systems and applications, such as automatic speech recognition, machine translation and auto-completion in search engines just to name a few [90]. Two noticeable examples are Google Translate and Apple Siri. In all such cases, storing the dictionary in compressed format has become fundamental since larger language models can be loaded in internal memory, therefore improving the application's accuracy and speed. This is especially true for battery-limited devices, such as mobile phones and tablets, where performing a network request is too costly and slow.

All these examples confirm the importance of compressing large real-world datasets and operating on them with efficient algorithms. For this reason, space-efficiency and fast access are the two key design goals of the work presented in this thesis. In particular, we will show that these two objectives, typically considered to be opposite by nature, can now be combined in a single solution.

To this end, particular care has been put in providing a *practical implementation* of the ideas described in the thesis that, besides the theoretical analysis of the algorithms, is the only one measuring a concrete improvement over the state-of-the-art. The *hiding constants* effect of the asymptotic notation can, in fact, abstract away the real space cost of a data structure or disrupt the efficiency of a carefully-designed algorithm. For example, compressed datasets often exhibit less space with a $O(n \log n)$-bit data structure rather than with a $o(n \log n)$-bit one or, similarly, a constant-time algorithm can be slower than a (small) optimized binary search.

1.1 Thesis organization and contributions

We describe here how the thesis is structured. After introducing the needed background in Chapter 2, we articulate our contributions in seven chapters that are briefly summarized below. We conclude the thesis with Chapter 10, discussing some interesting open problems and future research directions.

Integer Dictionaries in Compressed Space. The efficient maintenance of a dynamic set on n integer keys is among the most studied problems in Computer Science. While some solutions are known to spend an optimal amount of time per operations, whether time optimality can be preserved in the *compressed space* regime is still an interesting open problem.

Therefore, the problem we consider in Chapter 3 is the one of representing in compressed space a *dynamic ordered set* $\mathcal{S}$ of n integer keys drawn from a universe of size $u \geq n$. We show a data structure representing $\mathcal{S}$ with $n\lceil \log \frac{u}{n} \rceil + 2n + o(n)$ bits of space and supporting random access in $O(\log n / \log \log n)$ worst-case, insertions and deletions in $O(\log n / \log \log n)$ amortized and predecessor in $O(\min\{1 + \log \frac{u}{n}, \log \log n\})$ worst-case time. The mentioned time bounds are optimal [65, 134, 136].

The contents of this chapter are based on [127].

Clustered Inverted Indexes. State-of-the-art encoders for inverted indexes compress each posting list *individually*. Encoding *clusters* of inverted lists offers the possibility of reducing the redundancy of the lists while maintaining a noticeable query processing speed.

In Chapter 4 we propose a new index representation based on clustering the collection of inverted lists and, for each created cluster, building an ad-hoc *reference list* with respect to which all lists in the cluster are encoded. We describe an inverted lists clustering algorithm tailored for our encoder and an efficient method for building the reference list for a cluster.

The extensive experimental analysis indicates that significant space reductions are indeed possible, beating the best state-of-the-art encoders.

The contents of this chapter are based on [126].

Optimal Variable-Byte Encoding. The ubiquitous *Variable-Byte* encoding is considered one of the fastest compressed representation for integer sequences. However, its compression ratio is usually not competitive with other more sophisticated encoders, especially when the integers to be compressed are small that is the typical case for inverted indexes.

In Chapter 5 we show that the compression ratio of Variable-Byte can be improved by 2× by adopting a partitioned representation of the inverted lists. This makes Variable-Byte surprisingly competitive in space with the best bit-aligned encoders, *hence* disproving the folklore belief that Variable-Byte is space-inefficient for inverted index compression.

Despite the significant space savings, we show that our optimization almost comes for free, given that: we introduce an optimal partitioning algorithm that, by running in linear time and with low constant factors, does not affect indexing time; we show that the query processing speed of Variable-Byte is preserved with an extensive experimental analysis.

The contents of this chapter are based on [130].

Dictionary-based Decoding. In Chapter 6 we continue our exploration about effective *and* efficient inverted index representation by developing a new compression approach, based on a *dictionary of integer sequences*.

Conversely from the standard approaches, ours adopts a *fixed-to-fixed* decoding algorithm. The core idea is that each unit of decoding consumes one b-bit integer codeword, and causes a fixed-length copying operation from an internal codebook of size 2^b – the *dictionary* – to the output buffer. The elegance of this approach means that decoding is fast; and yet, as we demonstrate with our experiments, it also provides state-of-the-art compression effectiveness.

The contents of this chapter are based on [125][1].

Comparing Inverted Index Representations. In Chapter 7 we offer a conclusive experimental comparison between the techniques introduced in chapters 4, 5 and 6. We also test the many encoders for inverted indexes described in the background Section 2.2, in order to provide an extensive comparison against the state-of-the-art.

We adopt the same datasets used for the experiments in chapters 4, 5 and 6, whose description and statistics are reported below. Whenever needed in the experimental analysis of the individual chapters, we will point the reader to this subsection and the relevant table.

- Gov2 is the TREC 2004 Terabyte Track test collection, consisting in roughly 25 million .gov sites crawled in early 2004. The documents are truncated to 256 KB.

- ClueWeb09 is the ClueWeb 2009 TREC Category B test collection, consisting in roughly 50 million English web pages crawled between January and February 2009.

[1]This work was done while the author was visiting the School of Computing and Information Systems at the University of Melbourne, Australia. Therefore, some of the writing and pictures of the paper have been made in collaboration with the other two co-authors.

Collection	Lists	Postings	Documents
Gov2	35,636,425	5,742,630,292	24,622,347
ClueWeb09	92,094,694	15,857,983,641	50,131,015

Table 2: Basic statistics for the Gov2 and ClueWeb09 test collections.

Standard text preprocessing was performed on the collections. For each document, the body text was extracted using Apache Tika[2], and the words lowercased and stemmed using the `Porter2` stemmer. Identifiers were assigned to documents according to the lexicographic order of their URLs [148]. Table 2 reports the basic statistics for the two collections.

Compressed Indexes for N-Gram Strings. In Chapter 8 we address the problem of reducing the space required by the representation of datasets of short string (namely, n-grams), maintaining the capability of looking up for a given string within micro seconds.

In particular, we present a compressed trie data structure in which each word following a context of fixed length k, i.e., its preceding k words, is encoded as an integer whose value is proportional to the number of words that follow such context. Since the number of words following a given context is typically very small in natural languages, we lower the space of the representation to compression levels that were never achieved before. Despite the significant savings in space, our technique introduces a negligible penalty at query time. Compared to the state-of-the-art proposals, our data structures outperform all of them for space usage, without compromising their time performance. efficient proposals in the literature, that are both quantized and lossy, are not smaller than our trie data structure and up to 5 times slower. Conversely, we are as fast as the fastest competitor, but also retain an advantage of up to 65% in absolute space.

The contents of this chapter are based on [128].

Language Models Estimation. The problem we tackle in Chapter 9 is the one of computing the *Kneser-Ney* probability and backoff penalty for every n-gram, $1 \le n \le N$, extracted from a large textual source. Estimating such models from large texts poses the challenge of devising efficienct external memory algorithms, i.e., that make a parsimonious use of the disk.

The state-of-the-art algorithm uses three sorting steps in external memory: we show an improved construction that requires only *one* sorting step thanks to exploiting the properties of the extracted n-gram strings. With an extensive exper-

[2] http://tika.apache.org

imental analysis performed on billions of n-grams, we show an average improvement of $4.5\times$ on the total running time of the state-of-the-art approach.

The contents of this chapter are based on [131].

2 Background and Tools

In this thesis, we will address problems concerning the design of compressed data structures and the analysis of the algorithms operating on them. Therefore, the aim of this chapter is the one of introducing the notation and tools that we will exploit throughout the chapters of the thesis.

2.1 Basic concepts

Let $[n]$ indicate the ordered set of integers $\{0, \ldots, n-1\}, \forall n > 0$. Unless otherwise specified, all logarithms are in base 2, i.e., $\log x = \log_2 x$, $x > 0$, and we assume that $0 \log 0 = 0$. Given a set $\mathcal{X}$, let $|\mathcal{X}|$ be its *cardinality*.

Let $B(x)$ represent the binary representation of the integer x, and $U(x)$ its *unary* representation, that is a run of x zeros plus a final one: $0^x 1$. The *negated* unary representation of x is the bitwise NOT of $U(x)$, i.e., $1^x 0$.

2.1.1 Model of computation

Every algorithm is designed in the context of a model of computation. We analyze the algorithms described in this thesis with a classic model of computation, called *RAM* (Random Access Machine). This theoretical model tries to capture the behavior of a real computer, by using two components:

- a *CPU* with a set of elementary operations, all executed in constant time worst-case;

- a *set of memory cells* that we indicate with the integers in $[u]$. Each cell can be read/written by the CPU in constant time worst-case[1]. We will assume u to be a power of two, i.e., if $u = 2^w$ then w bits are transferred from memory to the CPU in $O(1)$. The bit width w is called the *memory word* size.

  The only assumption we make on w is that it is at least n, our input problem size, i.e., $w = \Omega(\log n)$ bits. This assumption is actually realistic because if w were not $\Omega(\log n)$, then we would not even be able to index all the elements in our input, i.e., $n > u$. The assumption also reveals the power and

[1] In practice, memory access on hardware is of *logarithmic* complexity due to adress translation from virtual to physical [91, 70].

generality of the model: w changes with n and so does the modelled CPU (*trans-dichotomous* RAM [66]).

Under this model of computation, the time complexity of an algorithm is measured as the number of memory words it reads/writes and, similarly, the space of a data structure is measured as the number of words it consists in.

However, it is our responsibility to not abuse the RAM model, pretending it includes some instructions that it should not. For illustrative purposes, think of an elementary sorting instruction. If such instruction were part of the model instruction set, then we would be able to sort in constant time by just using one instruction! This is, of course, a very unrealistic assumption [42]. Therefore, we assume the instruction set coincides with the one of the C programming language [93], that includes: arithmetic operations, bitwise operations, data movements and control structures such as loops (`for` and `while`) and conditionals (`if-and-else`).

2.1.2 Sequences, arrays and bitvectors

Many times in this thesis we will mention *lists*, *arrays*, *vectors* and *strings*. Since these are all equivalent from a mathematical point of view, we first provide a definition of such objects, commonly referred to as *sequences* and, then, fix some conventions that we will adopt throughout the thesis.

Definition 1 — Let Σ be a set called the *alphabet*. Then for any set $\mathcal{X}$ of $n = |\mathcal{X}|$ symbols drawn from Σ, we define a *sequence* as the function $\mathcal{S} : [n] \rightarrow \mathcal{X}$ defined as $i \mapsto x_i \in \mathcal{X}$.

Sequences and arrays. In other terms, a sequence $\mathcal{S}$ of length n is a function associating the integer indexes in $[n]$ with elements of a set $\mathcal{X}$.

There are many possible ways of implementing a sequence in a programming language, e.g., in C++ or Java. Perhaps, the easiest one is to use an *array*, that is a contiguous piece of memory containing a collection of objects of the same type and supporting random access to individual elements in constant time. In the thesis, whenever we mention the term *array* (or *vector*), we implicitly refer to this computerized representation and not to an abstract mathematical entity.

Thinking of a sequence as implemented with an array, we will refer to a sequence $\mathcal{S}$ of length n as $\mathcal{S}[0, n)$ and to its i-th element as $\mathcal{S}[i]$, for any $0 \leq i < n$. Similarly, for every $0 \leq i < j < n$, $\mathcal{S}[i, j]$ refers to the *sub-sequence* starting with element $\mathcal{S}[i]$ and ending with element $\mathcal{S}[j]$ (including it).

When the elements of a sequence are integers and the sequence is sorted, we call *universe* an integer $u \geq \mathcal{S}[n-1]$ and indicate such sequence with $\mathcal{S}(n, u)$.

With this definition and notation, it is now easy to refer to other objects without confusion. For example, we call *string* a sequence where its elements are characters from a given alphabet Σ (often identified with set of ASCII symbols[2]). Similarly a *text* can be thought to be a "long" string.

Bitvectors. A particular case of interest for us is the one with $\Sigma = \{0, 1\}$. In this case, we call any sequence a *bitvector* (often referred to as *bitarray* or *bitmap* in the literature). For example, the following $\mathcal{S}[0, 40)$ is a bitvector of 40 bits.

$$\mathcal{S} = \underset{32}{01001001}\quad \underset{24}{00011001}\quad \underset{16}{01001011}\quad \underset{8}{10100110}\quad \underset{0}{10001101}$$

Bitvectors can be implemented with arrays of integer numbers, with the binary representation of an integer representing a contiguous range of bits. For example, the above bitvector could be implemented with an array of 5 8-bit numbers, that are $\langle 141, 166, 75, 25, 73\rangle$ because $B(141) = 10001101$, $B(166) = 10100110$, $B(75) = 01001011$, $B(25) = 00011001$ and $B(73) = 01001001$. Similarly, with an array of 2 32-bit numbers, we will have $\langle 424388237, 73\rangle$. In such cases, bit-wise operations (binary and/or and left/right shift) may be needed to read and write ranges of bits.

Given two bitvectors $\mathcal{B}_1$ and $\mathcal{B}_2$, we indicate with $\mathcal{B}_1\mathcal{B}_2$ their concatenation.

Operations. Given a sequence $\mathcal{S}$ over a generic alphabet Σ, we define the following operations for any $0 \le i < n$.

- Access(i), that returns the i-th element of $\mathcal{S}$, i.e., $\mathcal{S}[i]$.
- Rank$_c(i)$, that returns the number of elements equal to c in $\mathcal{S}[0, i)$.
- Select$_c(i)$, that returns the position of the i-th element equal to c in $\mathcal{S}$ or just -1 if there is no such element in $\mathcal{S}$.

If the symbols of $\mathcal{S}$ are integer numbers and $\mathcal{S}$ is sorted, we will also use the following two operations.

- Predecessor$(x) = \max\{\mathcal{S}[i] : \mathcal{S}[i] < x\}$ or -1 if no such element exists.
- Successor$(x) = \min\{\mathcal{S}[i] : \mathcal{S}[i] \ge x\}$ or -1 if no such element exists.

For example, if $\mathcal{S}$ is $\langle 12, 14, 22, 35, 46\rangle$, then Predecessor$(10) = -1$, Successor$(15) = 22$, Predecessor$(40) = 35$, Successor$(244) = -1$, ecc.

Often we need to iterate *sequentially* through the elements of a sequence. In this case, we use the operation Next that takes no parameters and returns the next

[2] http://www.ascii-code.com

integer of the sequence after the last returned. Notice that this actually needs to keep track of the position of the last returned integer. We, therefore, assume that this information is saved somewhere (e.g., in a concrete C++ implementation, in the state of an iterator object) so that the last returned integer (and its position) is available for other queries on the sequence. This holds true for the operations Predecessor and Successor as well.

For example, continuing the example for $\mathcal{S} = \langle 12, 14, 22, 35, 46 \rangle$, consider the following sequence of operations: Successor(17), Next(), Next(). The results returned, in order, will be 22, 35 and 46 because Successor(17) $= \mathcal{S}[2] = 22$ and moving from position 2 twice will return the integers $\mathcal{S}[3] = 35$ and $\mathcal{S}[4] = 46$.

We will exploit *binary* rank and select in this thesis, that is, if $\mathcal{S}$ is a bitvector: $\text{Rank}_b(i)$ returns number of bits equal to b in $\mathcal{S}[0, i)$ and $\text{Select}_b(i)$ returns the position of the i-th b bit. For example, if $\mathcal{S}$ is

$$\mathcal{S} = \begin{array}{ccccccccccccc} 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} \end{array}$$

then $\text{Rank}_0(6) = 3$, $\text{Select}_0(4) = 10$, $\text{Rank}_1(4) = 2$, $\text{Select}_1(11) = -1$, ecc.

2.1.3 Information-theoretic lower bound

We often ask how many bits are needed to *optimally* represent a combinatorial object. In order to answer such question, we need a tool from Information Theory.

Given a set $\mathcal{X}$ of combinatorial objects, the minimum amount of bits we need to *uniquely* identify each element of $\mathcal{X}$ is $\lceil \log |\mathcal{X}| \rceil$ bits. This quantity is what we call an *information-theoretic lower bound.* This means that, generally speaking, there is no hope of using less bits to represent $\mathcal{X}$ then the quantity given by such bound.

But here is the trick, we shall say. In fact, if we *restrict* the scope of the objects in $\mathcal{X}$ by making some assumptions on them, we can lower the bound. As an example, consider the set $\mathcal{X}$ of bitvectors of length u. Then we have that $|\mathcal{X}| = 2^u$ and therefore we need at least u bits. Instead, if we restrict our attention to the class of bitvectors of length u *having exactly* n *bits set*, say $\mathcal{B}(n, u)$, then we have $|\mathcal{B}(n, u)| = \binom{u}{n}$, i.e., all possible ways we can set n bits starting from an all-zero bitvector of length u. Therefore, for the bitvectors in $\mathcal{B}(n, u)$ the information-theoretical lower bound is $\lceil \log \binom{u}{n} \rceil$ bits. Now, by doing some math (i.e., using Newton's coefficient formula and Stirling's factorial approximation), we have $\lceil \log \binom{u}{n} \rceil \simeq u \log u - n \log n + (u - n) \log \frac{1}{u-n} - O(\log u)$ and, by adding and subtracting $(u - n) \log u$, we finally get $\lceil \log \binom{u}{n} \rceil \simeq n \log \frac{u}{n} + (u - n) \log \frac{u}{u-n} - O(\log u)$ bits. This function is symmetric and has a maximum in $n = u/2$, therefore that we can concentrate our attention to the values of the function for $0 \le n \le u/2$ [145], obtaining

$$\mathcal{B}(n, u) = n \log \frac{u}{n} + O(n) \text{ bits}, \tag{1}$$

which represents a more practical way of describing the minimum number of bits we need to encode a bitvector in $\mathcal{B}(n, u)$.

2.1.4 Succinct data structures

Succinct data structures were originally introduced by Jacobson [88] in his doctoral thesis and constitute a specific class of compressed data structures. In particular, if an object (e.g., a tree, a graph or a sequence) requires *at least* m bits to be represented, then a succinct representation for this object will use $o(m)$ additional bits to implement all the operations needed for this object and the total space occupancy will be $m + o(m)$ bits. This quantity is referred to as a *succinct bound*. For example, many solutions are known to succinctly encode *static* binary trees and planar graphs [87]. Informally, we could say that the object is first described using the minimum amount of bits and, then, a “small” redundancy (a lower-order term) is added to the representation to implement the operations.

However, as already observed in Chapter 1, a *theoretical* negligible redundancy of $o(m)$ bits does not always imply a *practically* negligible redundancy too. This is due the *hiding constants* effect of the Big-Oh asymptotic notation, as also observed in other works [73, 160, 70].

2.1.5 Entropy

The father of Information Theory, Claude E. Shannon, left us a powerful tool that he names *entropy* [145]. He was concerned with the problem of defining the *information content* of a discrete random variable $\mathbf{x} : \Sigma \to \mathbb{R}$, with distribution $p_c = \mathbb{P}\{\mathbf{x} = c\}, \forall c \in \Sigma$. He defined the entropy of $\mathbf{x}$ as

$$\mathcal{H}(\mathbf{x}) = -\sum_{c \in \Sigma} p_c \log p_c \text{ bits}.$$

The quantity $-\log p_c$ bits is also called the *self-information* of $c \in \Sigma$. $\mathcal{H}(\mathbf{x})$ represents the number of bits we need to encode each value of Σ.

Let now $\mathcal{S}$ be a string of n characters drawn from an alphabet Σ. Let also n_c denote the number of times the character c occurs in $\mathcal{S}$. Assuming empirical frequencies as probabilities [121] (the larger is n, the better the approximation), i.e.,

$p_c \approx n_c/n$, we can consider $\mathcal{S}$ as a random variable assuming value c with probability p_c. In this setting, the entropy of a string $\mathcal{S}$ is

$$\mathcal{H}_0(\mathcal{S}) = -\frac{1}{n}\sum_{c\in\Sigma} n_c \log\frac{n_c}{n} \text{ bits.}$$

This quantity is also knows as the 0-th order *empirical* entropy of $\mathcal{S}$. The quantity $n\mathcal{H}_0(\mathcal{S})$ gives a theoretic lower bound on the average number of bits we need to represent $\mathcal{S}$, and consequently to the output size of *any* compressor that encodes each symbol of $\mathcal{S}$ with a fixed-length codeword.

2.2 Representation of integer sequences

A key ingredient of the data structures we will design in this thesis is the representation of integer sequences. Therefore, this section overviews the most important techniques used to compress integer numbers and sequences [129].

The most classical solution is to assign each integer a *self-delimiting* (or *uniquely-decodable*) variable-length codeword, so that the whole integer sequence is the result of the concatenation of the codewords of all its integers. Clearly, the aim of such encoders is to assign the smallest codewords as possible in order to minimize the number of bits used to represent the sequence.

2.2.1 Elias- Gamma and Delta

The two codes we now describe have been introduced by Elias [57] in the '60.

Given an integer $x > 0$, $\gamma(x) = 0^{|B(x)|-1}B(x)$, where $|B(x)| = \lceil\log(x+1)\rceil$. Therefore, $|\gamma(x)| = 2\lceil\log(x+1)\rceil - 1$ bits. For example, $\gamma(11) = 0001011$ because the binary representation of 11, that is 1011, has length 4 bits and, therefore, we prefix it by $4-1=3$ zeroes.

Decoding $\gamma(x)$ is simple: first count the number of zeroes up to the one, say there are n of these, then read the following $n+1$ bits and interpret them as x.

The key inefficiency of γ lies in the use of the unary code for the representation of $|B(x)|-1$, which may become very large for big integers. The δ code replaces the unary part $0^{|B(x)|-1}$ with $\gamma(|B(x)|)$, i.e., $\delta(x) = \gamma(|B(x)|)B(x)$. Notice that, since we are representing with γ the quantity $|B(x)| = \lceil\log(x+1)\rceil$ which is guaranteed to be greater than zero, δ can represent value zero too. The number of bits required by $\delta(x)$ is $|\gamma(|B(x)|)| + |B(x)|$, which is $2\lceil\log(\lceil|B(x)|+1\rceil)\rceil + \lceil\log(x+1)\rceil - 1$.

As an example, consider $\delta(113) = 001111110001$. The last part, 1110001, is $B(113)$ whose length is 7. Therefore we prefix $B(113)$ by $\gamma(7) = 00111$, which is the first part of the encoding.

The decoding of δ codes follows automatically from the one of γ codes.

2.2.2 Golomb-Rice

In the early '70, Rice [140] introduced a parameter code that can better compress the integers in a sequence if these are highly concentrated around some value. This code is actually a special case of the Golomb code [72], hence its name.

The Rice code of x with parameter k, $R_k(x)$, consists in two pieces: the *quotient* $q = \lfloor \frac{x-1}{2^k} \rfloor$ and the *remainder* $r = x - q \times 2^k - 1$. The quotient is encoded in unary, i.e., with $U(q)$, whereas the remainder is written in binary with k bits. Therefore, $|R_k(x)| = q + k + 1$ bits. Clearly, the closer 2^k is to the value of x the smaller the value of q: this implies a better compression and faster decoding speed. As an example, consider $R_5(113) = 000110000$. Because $\lfloor \frac{112}{2^5} \rfloor = 3$, we first write $U(3) = 0^3 1$. Then we write in binary the remainder $113 - 3 \times 2^5 - 1 = 16$, using 5 bits. If we would have adopted $k = 6$, then $R_6(113) = 01110000$. Notice that we are saving one bit with respect to $R_5(113)$ since $2^6 = 64$ is closer to 113 than $2^5 = 32$. In fact, the quotient in this case is 1 rather than 3.

Given the parameter k and the constant 2^k that is computed ahead, decoding Rice codes is simple too: count the number of zeroes up to the one, say there are q of these, then multiply 2^k by q and finally add the remainder, by reading the next k bits. Finally, add one to the computed result.

2.2.3 Variable-Byte

The codes described in the previous subsections are *bit-aligned* as they do not represent an integer using a multiple of a fixed number of bits, e.g., a byte. The decoding speed can be slowed down by the many operations needed to decode a single integer. This is a reason for preferring *byte-aligned* or even word-aligned codes when decoding speed is the main concern.

Variable-Byte (VByte), first described by Thiel and Heaps [155] in 1972, is the most popular and simplest byte-aligned code: the binary representation of a non-negative integer is split into groups of 7 bits which are represented as a sequence of bytes. In particular, the 7 least significant bits of each byte are reserved for the data whereas the most significant (the 8-th), called the *continuation bit*, is equal to 1 to signal continuation of the byte sequence. The last byte of the sequence has its 8-th bit set to 0 to signal, instead, the termination of the byte sequence. The main advantage of VByte codes is decoding speed: we just need to read one byte at a time until we found a value smaller than 2^7. Conversely, the number of bits to encode an integer cannot be less than 8, thus VByte is only suitable for large numbers and its compression ratio may not be competitive with the one of bit-aligned codes for small integers.

As an example, consider VByte(65,790) = $\underline{1}0000100\underline{1}0000001\underline{0}1111110$, where we underline the control bits. Also notice the padding bits in the first byte starting from the left, inserted to align the binary representation of the number to a multiple of 8 bits. VByte uses $\lceil \lceil \log(x+1) \rceil / 7 \rceil \times 8$ bits to represent the integer x.

Several enhancements were proposed in the literature to improve the (sequential) decoding speed of VByte. In order to reduce the probability of a branch misprediction - that leads to a higher throughput and helps keeping the CPU pipeline fed with useful instructions - the control bits can be grouped together. For example, if we assume that the largest represented integer fits into four bytes (which is reasonable, given that we often represent d-gaps), we have to distinguish between only four different lengths, thus two bits are sufficient. In this way, groups of four integers require one control byte. This optimization was introduced in Google's Varint-GB [45], which is faster than the original VByte format to decode.

Using SIMD. SIMD (Single-Instruction-Multiple-Data) is a computer organization that exploits the independence of multiple data objects to execute a single instruction on these objects simultaneously. Specifically, a single instruction (e.g., arithmetic or boolean) is executed for every element of a *vector*, that is a large(r) machine register containing, say, 128 or 256 bits of data, that packs multiple elements together. SIMD is now widely used to accelerate the execution of many data-intensive tasks, e.g., graphic-related applications, and (usually) an optimizing compiler is able of "vectorizing" code snippets to make them run faster.

Working with byte-aligned codes opens the possibility of exploiting the parallelism of SIMD instructions to further enhance decoding speed. This is the approach taken by the recent proposals Varint-G8IU [151], Masked-VByte [132] and Stream-VByte [101].

Varint-G8IU [151] uses a format similar to the one of Varint-GB: one control byte is used to describe a variable number of integers in a data segment of exactly eight bytes, therefore each group can contain between two and eight compressed integers. In Chapter 7 we will see an improvement in sequential decoding speed of ≈2× with respect to the scalar VByte decoding.

Masked-VByte [132] works, instead, directly on the original VByte format. The decoder first gathers the most significant bits of consecutive bytes using a dedicated SIMD instruction. Then using previously-built look-up tables and a shuffle instruction, the data bytes are permuted to obtain the decoded integers.

Stream-VByte [101] separates the encoding of the control bits from the data bits by writing them into separate streams. This organization permits to decode multiple control bits simultaneously and, therefore, reduce branch mispredictions that can stop the CPU pipeline execution when decoding the data stream.

2.2.4 Packed

A relatively simple approach to improve both compression ratio and decoding speed is to encode a *block* of integers, instead of the whole sequence. This line of work finds its origin in the so-called *frame-of-reference* [71] and dates back to the late '90.

Once the sequence has been partitioned into blocks (of fixed or variable length), then each block is encoded separately. An example of this approach is *binary packing* [10, 100], where blocks of fixed length are used, e.g., 128 integers. Given a block $\mathcal{S}[i, j]$, we can simply represent its integers using $\lceil \log(\mathcal{S}[j] - \mathcal{S}[i] + 1) \rceil$ bits by subtracting the lower bound $\mathcal{S}[i]$ from their values. Plenty of variants of this simple approach has been proposed [149, 49, 100].

Simple. Among some of the simplest binary packing strategies, Simple-9 [9], Simple-16 [173] and Simple-8b [10] combine very good compression ratio and high decompression speed. The key idea is to try to pack as many integers as possible in a memory word (32 or 64 bits). As an example, Simple-9 uses 4 *selector* bits and 28 *data* bits. The selector bits provide information on how many elements are packed in the data segment using equally-sized codewords. A selector 0000 may correspond to 28 1-bit integers; 0001 to 14 2-bits integers; 0010 to 9 3-bits integers (1 bit unused), and so on. The four bits distinguish from 9 possible configurations. Similarly, Simple-16 has 16 possible configurations. Simple-8b, instead, uses 64-bit words with fixed 4-bit selectors.

QMX. The QMX mechanism introduced by Trotman [156] packs as many integers as possible into 128-bit words (Quantities) and stores the selectors (eXtractors) separately in a different stream. The selectors are compressed (Multipliers) with RLE (Run-Length Encoding), that is with a stream of couples $(value, length)$. For example, if $\mathcal{S} = \langle 12, 12, 12, 5, 7, 7, 7, 7, 9, 9 \rangle$, the corresponding RLE representation is $\langle (12, 3), (5, 1), (7, 4), (9, 2) \rangle$.

2.2.5 PForDelta

The biggest limitation of block-based strategies is their space-inefficiency whenever a block contains at least one large value, because this forces the compressor to use a universe of representation as large as this value. This has been the main motivation for the introduction of *PForDelta* (PFOR) [176]. The idea is to choose a proper value k for the universe of representation of the block, such that a large fraction, e.g., 90%, of its integers can be written in k bits each. All integers that do not fit in k bits, are treated as *exceptions* and encoded separately using another compressor. This strategy is called *patching*.

More precisely, two configurable parameters are chosen: a b value (base) and a universe of representation k, so that most of the values fall in the range $[b, b + 2^k - 1]$ and can be encoded with k bits each by shifting (delta-encoding) them in the range $[0, 2^k - 1]$. To mark the presence of an exception, we also need a special *escape* symbol, thus we have $[0, 2^k - 2]$ available configurations.

As an example, consider the sequence $\langle 3, 4, 7, 21, 9, 12, 5, 16, 6, 2, 34\rangle$. By using $b = 2$ and $k = 4$, we obtain $\langle 1, 2, 5, *, 7, 10, 3, *, 4, 0, *\rangle\langle 21, 16, 34\rangle$, where we use the special symbol $*$ to represent an exception that is written in a separate sequence, reported to the right part of the example.

The *optimized* variant Opt-PFOR [170], which selects for each block the values of b and k that minimize its space occupancy, has been demonstrated to be more space-efficient and only slightly slower than the original PFOR [170, 100].

2.2.6 Elias-Fano

The encoder we now describe was independently proposed by Elias [56] and Fano [58] in the '70, hence its name.

Encoding. Given a sorted sequence $\mathcal{S}(n, u)$, we write each $\mathcal{S}[i]$ in binary using $\lceil \log u \rceil$ bits. The binary representation of each integer is then split into two parts: a *low* part consisting in the right-most $\ell = \lceil \log \frac{u}{n} \rceil$ bits that we call *low bits* and a *high* part consisting in the remaining $\lceil \log u \rceil - \ell$ bits that we similarly call *high bits*. Let us call ℓ_i and h_i the values of low and high bits of $\mathcal{S}[i]$ respectively. The Elias-Fano encoding of $\mathcal{S}(n, u)$ is given by the encoding of the high and low parts.

The integers $L = [\ell_0, \ldots, \ell_{n-1}]$ are written explicitly in $n\lceil \log \frac{u}{n} \rceil$ bits and they represent the encoding of the low parts. Concerning the high bits, we represent them in *negated unary* using a bitvector of $n + 2^{\lfloor \log n \rfloor} \leq 2n$ bits as follows. We start from a 0-valued bitvector H and set the bit in position $h_i + i$, for all $0 \leq i < n$. The effect is that now the k-th unary integer m of H indicates that m integers of

input	**3**	**4**	**7**	**13**	**14**	**15**	**21**	**25**	**36**	**38**		**54**	**62**
	0	0	0	0	0	0	0	0	1	1	**1**	1	1
high	0	0	0	0	0	0	1	1	0	0	**0**	1	1
	0	0	0	1	1	1	0	1	0	0	**1**	0	1
	0	1	1	1	1	1	1	0	1	1		1	1
low	1	0	1	0	1	1	0	0	0	1		1	1
	1	0	1	1	0	1	1	1	0	0		0	0
H	1110			1110			10	10	110		0	10	10
L	001100111			101110111			101	001	100110			110	110

Figure 2: The sequence $\langle 3, 4, 7, 13, 14, 15, 21, 25, 36, 38, 54, 62\rangle$ as encoded with Elias-Fano. The *missing high bits* are shown in bold.

$\mathcal{S}$ have high bits equal to k, $0 \leq k \leq \lfloor \log n \rfloor$. Finally the Elias-Fano representation of $\mathcal{S}$ is given by the concatenation of H and L and overall takes[3]

$$\mathsf{EF}(\mathcal{S}(n, u)) \leq n \left\lceil \log \frac{u}{n} \right\rceil + 2n \text{ bits.} \tag{2}$$

While we can opt for an arbitrary split into high and low parts, ranging from 0 to $\lceil \log u \rceil$, it can be shown that $\ell = \lceil \log \frac{u}{n} \rceil$ minimizes the overall space occupancy of the encoding [56]. Moreover, as the information-theoretic lower bound (see Section 2.1.3) for a monotone sequence of n elements drawn from a universe of size u is $\lceil \log \binom{u+n}{n} \rceil \approx n \log \frac{u+n}{n} + n \log e$ bits, it can be shown that less than half a bit is wasted per element by the Elias-Fano space bound [56].

Figure 2 shows a graphical example of encoding for the sequence $\mathcal{S}(12, 62) = \langle 3, 4, 7, 13, 14, 15, 21, 25, 36, 38, 54, 62\rangle$. The *missing high bits* embody the representation of the fact that using $\lfloor \log n \rfloor$ bits to represent the high part of an integer, we have *at most* $2^{\lfloor \log n \rfloor}$ distinct high parts because not all of them could be present. In Figure 2, we have $\lfloor \log 12 \rfloor = 3$ and we can form up to 8 distinct high parts. Notice that, for example, no integer has high part equal to 101 which are, therefore, "missing" high bits.

Since we set a bit for every $0 \leq i < n$ in H and each h_i is extracted in $O(1)$ time from $\mathcal{S}[i]$, it follows that $\mathcal{S}$ gets encoded with Elias-Fano in $\Theta(n)$ time.

Access. Despite the space-efficiency of the encoding, it is possible to support Access *without* decompressing the whole sequence. The operation is implemented by using an auxiliary data structure that is built on the bitvector H and that is able to efficiently answer Select_1 queries (see Section 2.1.2). This auxiliary data

[3]Using the same choice of Elias for ceilings and floors, we arrive at the slightly different bound of at most $n\lfloor \log \frac{u}{n} \rfloor + 3n$ bits.

structure is succinct in the sense that it is negligibly small in asymptotic terms compared to $\mathsf{EF}(\mathcal{S}(n,u))$, requiring only $o(n)$ additional bits [108, 161].

Using the Select_1 primitive, it is possible to implement Access in $O(1)$. We basically have to re-link together the high and low bits of an integer, previously split up during the encoding phase. The low bits ℓ_i are trivial to retrieve as we need to read the range of bits $L[i \times \ell, (i+1) \times \ell)$. The retrieval of the high bits deserve, instead, a bit more care. Since we write in negated unary how many integers share the same high part, we have a bit set for every integer in $\mathcal{S}$ and a zero for every distinct high part. Therefore, to retrieve the high bits of the i-th integer, we need to know how many zeros are present in $H[0, \mathsf{Select}_1(i))$. This quantity is evaluated on H in $O(1)$ as $\mathsf{Rank}_0(\mathsf{Select}_1(i)) = \mathsf{Select}_1(i) - i$ (notice that Rank_0 is not needed at all). Finally, linking the high and low bits is as simple as: $\mathsf{Access}(i) = ((\mathsf{Select}_1(i) - i) \ll \ell) \mid \ell_i$, where $\ll$ is the left shift operator and $\mid$ is the bitwise OR.

Successor. The query Successor(x) is supported in $O(1 + \log \frac{u}{n})$ time[4], as follows. Let h_x be the high bits of x. Then for $h_x > 0$, $i = \mathsf{Select}_0(h_x) - h_x + 1$ indicates that there are i integers in S whose high bits are less than h_x. On the other hand, $j = \mathsf{Select}_0(h_x + 1) - h_x$ gives us the position at which the elements having high bits greater than h_x start. The corner case $h_x = 0$ is handled by setting $i = 0$. These two preliminary operations take $O(1)$. Now we have to conclude our search in the range $\mathcal{S}[i, j]$, having *skipped* a potentially large range of elements that, otherwise, would have required to be compared with x. We finally determine the successor of x by binary searching in this range which may contain up to u/n integers. The time bound follows.

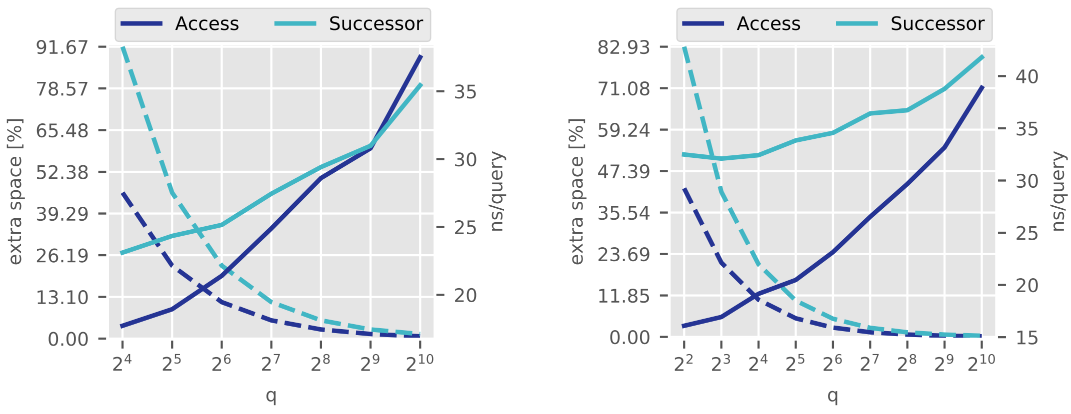
As an example, consider Successor(30). Since $h_{30} = 3$, we have $i = \mathsf{Select}_0(3) - 3 + 1 = 7$ and $j = \mathsf{Select}_0(4) - 3 = 8$. Therefore we conclude our search in the range $\mathcal{S}[7, 8]$ by returning Successor(30) $= \mathcal{S}[8] = 36$.

The algorithm for Predecessor(x) runs in a similar way. In particular, it could be that Predecessor(x) lies before the interval $\mathcal{S}[i, j]$: in this case $\mathcal{S}[i-1]$ is the element to return.

Implementing the binary Select primitive. As the binary Select primitive is fundamental to enable fast Elias-Fano codes (and many other succinct data structures as well), we now discuss how it can be implemented efficiently on top of a bitvector by using little extra space. Also, see the paper by Vigna [161] and references therein for an in-depth discussion.

While there exists a large body of research on selection that has developed *optimal*, i.e., constant-time, algorithms requiring tiny space [38], such solutions are rather complicated and not practically appealing for their high constant costs. The

[4] We report the bound as $O(1 + \log \frac{u}{n})$, instead of $O(\log \frac{u}{n})$, to cope with the case $n = u$.



(a) Dense **(b)** Sparse

Figure 3: Query time (solid lines) and extra space (dashed lines) for the Elias-Fano representation of a sequence of one million integers, having an average gap of 2 **(a)** and 2000 **(b)**.

solution we describe here is, instead, very simple and of practical use. It actually dates back to the original work by Elias [56].

Let n be the size of the bitvector and m the number of bits set. We write the position of the next bit we would reach after reading $k \times q$ unary codes, where q is a fixed quantum and $k = 0, \ldots, \lfloor m/q \rfloor$. These $\lfloor m/q \rfloor$ sampled positions are called *forward pointers*. The algorithm for $\text{Select}_1(i)$ first fetches the $(\lfloor i/q \rfloor)$-th pointer, say p, and then completes exhaustively by reading i % q unary codes starting from position p in the bitvector.

Concerning the Elias-Fano representation of a sequence, we can use the introduced algorithm on top of bitvector H to support Select_1 for random Access and, with a straightforward adaptation, also Select_0 for Successor. In Figure 3 we show how the needed extra space and query time change by varying q, for a sequence of 1 million integers whose average gap between the integers is 2 (*dense* case) and 2000 (*sparse* case). The benchmark numbers were obtained with an Intel i7-7700 processor clocked at 3.6 GHz, on Linux 4.4.0 (64 bits). The code was written in C++ and compiled in release mode with gcc 7.3.0 using compilation flags `-O3` and `-march=native`.

Clearly, smaller values of q permit less unary reads, hence resulting in a faster execution, but cost more extra space. As a suitable space/time trade-off configuration for all the experiments in the thesis, we choose $q = 2^8$ for Access and $q = 2^9$ for Successor given that in correspondance of these values the extra cost is almost the same for both operations, being 2.86% for the *dense* case and 0.65% for the *sparse* case.

2.2.7 Partitioned Elias-Fano

One of the most relevant characteristics of the Elias-Fano space Formula 2 is that it only depends on two parameters, i.e., the length n and universe u of the sequence $\mathcal{S}(n, u)$. As we are going to see in the next chapters, integer sequences often present groups of very similar numbers and Elias-Fano fails to exploit such natural clusters because it uses a number of bits per integer equal to $\lceil \log(u/n) \rceil + 2$, thus proportional to the logarithm of the *average gap* (i.e., u/n) between the integers.

Therefore, partitioning the sequence into blocks to better adapt to the distribution of the gaps between the integers, is the key idea of the *partitioned Elias-Fano* (PEF) representation devised in [120].

The core idea is as follows. The sequence $\mathcal{S}(n, u)$ is partitioned into k blocks of variable length. The first level of representation stores two sequences compressed with plain Elias-Fano: (1) the sequence made up of the last elements $\{u_1, \ldots, u_k\}$ of the blocks, the so-called *upper-bounds* and (2) the prefix-summed sequence of the sizes of the blocks. The second level is formed, instead, by the representation of the blocks themselves, that can be again encoded with Elias-Fano. The main advantage of this two-level representation, is that now the integers in the i-th block are encoded with a smaller universe, i.e., $u_i - u_{i-1} - 1$, $i > 0$, thus improving the space with respect to the original Elias-Fano representation.

Encoding. More precisely, each block in the second level is encoded with one among three different strategies. As already stated, one of them is Elias-Fano. The other two additional strategies come into play to overcome the space inefficiencies of Elias-Fano when representing *dense* blocks.

Let consider a block and call b its size, m its universe respectively. Vigna [161] first observed that as b approaches m the space bound $b\lceil \log(m/b) \rceil + 2b$ bits becomes close to $2m$ bits. In other words, the closer b is to m, the denser the block. However, we can always represent the block with m bits by writing the *characteristic vector* of the block, that is a bitvector where the i-th bit is set if the integer i belongs to the block.

Therefore, besides Elias-Fano, two additional encodings can be chosen to encode the block, according on the relation between m and b. The first one addresses the extreme case in which the block covers the whole universe, i.e., when $b = m$: in such case, the first level of the representation (upper-bound and size of the block) trivially suffices to recover each element of the block which is, therefore, encoded with zero bits. The second case is used whenever the number of bits used by the Elias-Fano representation of the block is larger than m bits: by doing the math, it is not difficult to see that this happens whenever $b > m/4$. In this case we can encode the block with its characteristic bitvector using m bits.

The dynamic programming algorithm. Splitting the sequence $\mathcal{S}(n, u)$ into equally-sized block is clearly sub-optimal, since we cannot expect clusters of similar integers to be aligned with uniform partitions. For such reason, an algorithm based on dynamic programming is presented in [120] to find, in $\Theta(n)$ time and space, a partition whose cost in bits (i.e., the space taken by the partitioned encoded sequence) is at most $(1+\epsilon)$ times away from the optimal one, for any $0 < \epsilon < 1$. We now describe such algorithm.

The problem of determining the partition of minimum encoding cost can be seen as the problem of determining the path of minimum cost (shortest) in a complete, weighted and directed acyclic graph (DAG) $\mathcal{G}$. This DAG has n vertices, one for each integer of $\mathcal{S}$, and $\Theta(n^2)$ edges where the cost $w(i, j)$ of edge (i, j) represents the number of bits needed to represent $\mathcal{S}[i, j]$. Each edge cost $w(i, j)$ is computed in $O(1)$ by just knowing the universe and size of the chunk $\mathcal{S}[i, j]$, as explained above.

Since the DAG is complete, a simple shortest path algorithm is inefficient already for medium sized inputs. However, it could be used on a pruned DAG $\mathcal{G}_\epsilon$, which is obtained from $\mathcal{G}$ and has the following crucial properties: (1) the number of edges is $O(n \log_{1+\epsilon} \frac{U}{F})$ for any given $\epsilon \in (0, 1)$; (2) its shortest path distance is at most $(1+\epsilon)$ times the one of the original DAG $\mathcal{G}$. U represents the encoding cost of $\mathcal{S}$ when no partitioning is performed; F represents the fixed cost that we pay for each partition. Precisely, for each partition we have to write its universe of representation, its size and a pointer to its second-level Elias-Fano representation. F can be, therefore, safely upper bounded with $2 \log u + \log n$.

The pruning step retains all the edges (i, j) from $\mathcal{G}$ that satisfy the following two properties for any $i = 0, \ldots, n-1$, $j > i$: (1) there exists an integer $h \geq 0$ such that $w(i, j) \leq F \times (1+\epsilon)^h < w(i, j+1)$; (2) (i, j) is the last edge outgoing from node i, i.e., $j = n$. The edges is $\mathcal{G}_\epsilon$ are the ones that better approximate the value $(1+\epsilon)^h F$ from below because the edge costs are monotone. Such edges are called $(1+\epsilon)$-maximal edges. Since for each $h \geq 0$ it must be $F \times (1+\epsilon)^h \leq U$, there are at most $\log_{1+\epsilon} \frac{U}{F}$ edges outgoing from each node of $\mathcal{G}_\epsilon$, thus in conclusion $\mathcal{G}_\epsilon$ has $O(n \log_{1+\epsilon} \frac{U}{F})$ edges. Now the dynamic programming recurrence can be solved in $\mathcal{G}_\epsilon$ in $O(n \log_{1+\epsilon} \frac{U}{F})$ admitting a solution whose cost is at most $(1+\epsilon)$ times larger than the optimal one [61].

We can further reduce this complexity to $O(n \log_{1+\epsilon} \frac{1}{\epsilon})$ conserving the same approximation guarantee as follows. Let $\epsilon_1 \in [0, 1)$ and $\epsilon_2 \in [0, 1)$ two approximation parameters. We first retain from $\mathcal{G}$ all the edges whose cost is no more than $L = \frac{F}{\epsilon_1}$, then we apply the pruning strategy as described above with approximation parameter ϵ_2. The obtained graph has now $O(n \log_{1+\epsilon_2} \frac{L}{F}) = O(n \log_{1+\epsilon_2} \frac{1}{\epsilon_1})$ edges, which is $O(n)$ as soon as ϵ_1 and ϵ_2 are fixed. It can be shown [120] that the shortest path distance is no more than $(1+\epsilon_1)(1+\epsilon_2) \leq (1+\epsilon)$ times the one in $\mathcal{G}$ by setting

$\epsilon_1 = \epsilon_2 = \frac{\epsilon}{3}$. Note that the complexity $O(n \log_{1+\epsilon} \frac{1}{\epsilon})$ becomes $\Theta(n)$ as soon as ϵ is constant.

Finally, to generate the pruned DAG from $\mathcal{G}$, we employ $q = O(\log_{1+\epsilon} \frac{1}{\epsilon})$ windows $W_1, \ldots, W_q$, one for each possible exponent $h \geq 0$ such that $F \times (1+\epsilon)^h \leq L$. Each sliding window covers a different portion of S and it slides over the sequence. We generate the q maximal edges outgoing from node i on-the-fly as soon as the shortest path algorithm visit this node. Initially all windows start and end at position 0. Every time the algorithm visits the next node i, we advance the starting position of each window by one position and the ending position j until $w(i, j)$ exceeds the value $F \times (1+\epsilon)^j$.

In practice. The parameters ϵ_1 and ϵ_2 deeply affect index construction time, as they directly control the number of edges considered by the algorithm. Therefore, whenever mentioned in this thesis, we refer to PEF as the configuration that uses $\epsilon_1 = 0.03$ and $\epsilon_2 = 0.3$ that were chosen in the original work (and implementation) by Ottaviano and Venturini [120], to tradeoff between index space and construction time.

2.2.8 Binary interpolative coding

Binary interpolative coding (BIC) [115] is a recursive algorithm that first encodes the integer in the middle of the sequence and then recursively applies this encoding step to both halves of the sequence. At each step of recursion, the algorithm knows the reduced ranges that will be used to write the middle elements in fewer bits during the next recursive calls.

More precisely, consider the range $\mathcal{S}[i, j]$. The encoding step writes the quantity $\mathcal{S}[m] - low - m + i$ using $\lceil \log(hi - low - j + i) \rceil$ bits, where: $\mathcal{S}[m]$ is the range middle element, i.e., the integer at position $m = (i + j)/2$; low and hi are respectively the lower bound and upper bound of the range $\mathcal{S}[i, j]$, i.e., two quantities such that $low \leq \mathcal{S}[i]$ and $hi \geq \mathcal{S}[j]$. The algorithm proceeds recursively, by applying the same encoding step to both halves: $\mathcal{S}[i, m]$ and $\mathcal{S}[m + 1, j]$ by setting $hi = \mathcal{S}[m] - 1$ for the left half and $low = \mathcal{S}[m] + 1$ for the right half.

At the beginning, the encoding algorithm starts with $i = 0$, $j = n - 1$, $low = \mathcal{S}[0]$ and $hi = \mathcal{S}[n - 1]$. These quantities must be also known at the beginning of the decoding phase. Apart from the initial lower and upper bound, all the other values of low and hi are *computed on-the-fly* by the algorithm.

Figure 4 shows an encoding example for the same sequence of Figure 2. We can interpret the encoding as a pre-order visit of the binary tree formed by the recursive calls the algorithm performs. Therefore, the encoded elements in the example will be, in order: $\langle 7, 2, 0, 0, 18, 5, 3, 16, 1, 7 \rangle$. Moreover, notice that whenever the al-

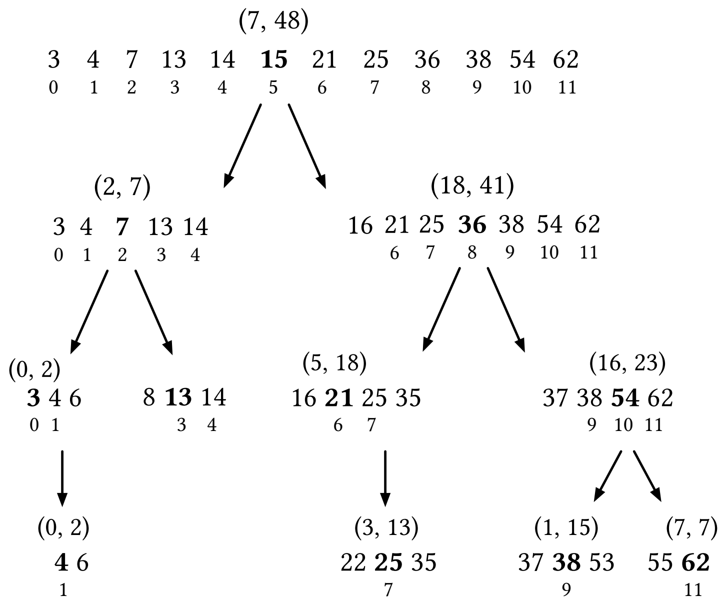


Figure 4: Binary Interpolative coding example for the same sequence of Figure 2, i.e., $\langle 3, 4, 7, 13, 14, 15, 21, 25, 36, 38, 54, 62\rangle$. In bold we highlight the middle element currently encoded: above each element we report a pair, where the first value indicates the element actually encoded and the second value the universe of representation.

gorithm works on a range $\mathcal{S}[i, j]$ of *consecutive integers*, such as the range $\mathcal{S}[3, 4] = [13, 14]$ in the example, i.e., the ones for which the condition $\mathcal{S}[j] - \mathcal{S}[i] = j - i$ holds, it stops the recursion by emitting no bits at all. The condition is again detected at decoding time and the algorithm implicitly decodes the run $\mathcal{S}[i], \mathcal{S}[i] + 1, \mathcal{S}[i] + 2, \ldots, \mathcal{S}[j]$. This property makes BIC extremely space-efficient whenever the encoded sequences feature *clusters* of very close integers [168, 170, 149]. The key inefficiency of BIC is, however, the decoding speed which is highly affected by the recursive nature of the algorithm.

2.2.9 Asymmetric numeral systems

Asymmetric Numeral Systems (ANS) is a family of *entropy* coders, originally developed by Jarek Duda [55]. ANS combines the excellent compression ratio of Arithmetic coding with a decompression speed comparable with the one of Huffman. It is now widely used in commercial applications, like Facebook ZSTD, Apple LZFSE and in the Linux kernel.

The basic idea of ANS is to represent a sequence of symbols with a natural number x. If each symbol s belongs to the binary alphabet $\Sigma = \{0, 1\}$, then appending s to the end of the binary string representing x will generate a new integer $x' = 2x + s$. This coding is optimal whenever $\mathbb{P}(0) = \mathbb{P}(1) = 1/2$. ANS generalizes

$$
\begin{aligned}
\mathcal{D}_1 &= \{house, dog, red, boy, people\} \\
\mathcal{D}_2 &= \{dog, boy, people, hungry\} \\
\mathcal{D}_3 &= \{people, boy, red\} \\
\mathcal{D}_4 &= \{hungry, house, people, sun, red\}
\end{aligned}
$$

(a)

$$
\begin{aligned}
boy &\longrightarrow \langle 1, 2, 3 \rangle \\
dog &\longrightarrow \langle 1, 2 \rangle \\
house &\longrightarrow \langle 1, 4 \rangle \\
hungry &\longrightarrow \langle 2, 4 \rangle \\
people &\longrightarrow \langle 1, 2, 3, 4 \rangle \\
red &\longrightarrow \langle 1, 3, 4 \rangle \\
sun &\longrightarrow \langle 4 \rangle
\end{aligned}
$$

(b)

Table 3: In **(a)**, a collection of four textual documents, represented as sets of terms and, in **(b)**, the corresponding inverted lists.

this concept by adapting it to a general distribution of symbols $\{p_s\}_{s \in \Sigma}$. In particular, appending a symbol s to x increases the information content from $\log x$ bits to $\log x - \log p_s = \log(x/p_s)$ bits, thus the new generated natural number will be $x' \approx x/p_s$.

2.3 Inverted indexes

Consider a collection of textual *documents*, where each document can be regarded as being a (multi) set of terms, like the ones shown in Table 3a. Now, for each distinct term t appearing in the collection we build the integer sequence $\mathcal{S}_t$ listing in sorted order all the identifiers of the documents (docIDs in the following) in which the term appears. The sequence $\mathcal{S}_t$ is called the *inverted list* (or *posting list*) of the term t and the collection of inverted lists for all the distinct terms is called the *inverted index*. Table 3b shows the inverted index built for the textual collection in Table 3a.

Inverted lists can store additional information about each term, such as the set of positions in which the term appears in the documents (in *positional* indexes) and the number of occurrences of the term in the documents (term *frequency*) [175, 109, 31]. In this thesis we consider the docID-*sorted* version of the inverted index, i.e., all inverted lists are monotonically increasing. Furthermore, we ignore additional information about each term except the term frequencies, which are stored in separate sequences. For example, consider the inverted list $boy \longrightarrow \langle 1, 2, 3 \rangle$ in Table 3b and call $f_{boy,k}$ the number of times the term *boy* appears in document k. Suppose that $f_{boy,1} = 2$, $f_{boy,2} = 1$ and $f_{boy,3} = 1$. Then the sequence of the frequencies will be $\langle 2, 1, 1 \rangle$, which is *not* sorted by construction.

The inverted index is the data structure at the core of nowadays large-scale search engines, social networks and storage architectures [175, 109, 31] and, differently from our toy example in Table 3, it stores million of documents. We mention now some noticeable examples.

Classically, inverted indexes are used to support full-text search in databases [109]. Then, identifying a set of documents containing all the terms in a user query reduces to the problem of *intersecting* the inverted lists associated to the terms in the query, as we are going to better see in the next subsection. Likewise, an inverted list can be associated to a user in a social network (e.g., Facebook) and stores the sequence of all friend identifiers of the user [44]. Moreover, database systems based on SQL often precompute the list of row identifiers matching a specific frequent predicate over a huge table, in order to speed up the execution of a query involving the conjunction of, possibly, many predicates [80, 139]. Also, finding all occurrences of twig patterns in XML databases can be done efficiently by resorting on an inverted index [27]. In recent years, a vast number of key-value stores has emerged, e.g., Apache Ignite, Redis, InfinityDB, BerkeleyDB and many others. Common to all such architectures is the organization of data elements falling into the same bucket due to an hash collision: the list of all such elements is recorded, which is nothing but an inverted list [47].

2.3.1 Query processing

The inverted index owes its popularity to the efficient resolution of *queries*, expressed as a set of terms $\{t_1, \ldots, t_k\}$ combined with a query operator. The simplest operators are boolean AND and OR such that, for example, the query q = AND(t_i, t_j) means that the index has to report all the docIDs of the documents containing term t_i *and* term t_j. This operation ultimately boils down to *intersecting* the two inverted lists $\mathcal{S}_{t_i}$ and $\mathcal{S}_{t_j}$, because the docIDs belonging to the intersection of the two lists are the ones corresponding to documents in which t_i and t_j *co-occur*. Back to our example in Table 3, given the query q = AND(*hungry*, *dog*), the index has to fetch the inverted lists of the two terms, i.e., $\langle 2, 4\rangle$ and $\langle 1, 2\rangle$, and compute their intersection. In this case, the intersection contains the docID 2 and, as we can see in Table 3a, the document $\mathcal{D}_2$ is the only one containing *both hungry* and *dog*. We can easily generalize the above example to an arbitrary number of query terms, as well as with other query operators.

However, as already stated, the inverted lists manipulated by search engines are very long and computing the intersection must be done efficiently. This is especially true considering that million of users could be using the inverted index at the same time (think of, for example, the Google search engine). For these reasons, an efficient algorithm for computing the intersection between two (or more) sorted integer sequences is of utmost importance. The efficiency of this algorithm relies on the query $\mathsf{Successor}_t(x)$, which returns the smallest integer $z \in \mathcal{S}_t$ such that $z \geq x$ (see Section 2.1.2). In the context of inverted indexes, the Successor query is often called NextGEQ (next Greater-then or EQual-to). The NextGEQ is used because it permits to *skip* over the sequences to be intersected.

Suppose we have to compute $\mathsf{AND}(t_i, t_j) = \mathcal{S}_{t_i} \cap \mathcal{S}_{t_j}$, and suppose that $\mathcal{S}_{t_i}$ is smaller than $\mathcal{S}_{t_j}$. We search for the first docID $x = \mathcal{S}_{t_i}[0]$ in $\mathcal{S}_{t_j}$ by means of $\mathsf{NextGEQ}_{t_j}(x)$: if the docID z returned by the search is equal to x then it is a member of the intersection and we can just repeat this search step for the next docID of $\mathcal{S}_{t_i}$; otherwise z gives us the *candidate* docID to be searched next, indeed allowing to *skip the searches* for all docIDs in between x and z. In fact, since we have that $z \geq x$, $\mathsf{NextGEQ}_{t_j}(y)$ will be equal to z also for *all* $x < y < z$, thus none of such docIDs can be a member of the intersection.

This strategy processes the two inverted list *concurrently*, keeping them aligned by docID. This behaviour is known as *document-at-a-time* and is the one that we will adopt in this thesis, since it is the most natural for docID-sorted indexes. Another popular strategy is, instead, *term-at-a-time* that scans each inverted list *separately* to build the results set.

The algorithm coded in Figure 5 illustrates what we have just explained for any arbitrary number k of query terms. In particular, it returns the *number of* docIDs shared between the inverted lists, rather than returning the set of docIDs themselves (that is, anyway, a simple extension). In the pseudo code we adopted the convention of regarding a sequence as an object of a programming language and use the dot notation `obj.foo(args)` to mean that the `foo` method is invoked on the object `obj` with arguments `args`.

In many applications, it is also preferable to associate to each document matching the query a *score* value, indicating the *relevance* of the document to the query. This score is typically computed as a function of the term frequency in the document and few other statistics. For example, a common relevance score is BM25 [141]. Furthermore, to limit the number of query results, only the k documents scoring better are usually taken into account (top-k retrieval).

For example, WAND [25] is a popular strategy that augments the inverted index by storing for each term its maximum impact to the score. More precisely, WAND processes a query in *phases* by maintaining a priority queue storing the top-k documents seen so far along with a cursor for each term in the query scanning the corresponding inverted list. During each phase, WAND estimates an upper bound to the scores that can be reached by the documents currently pointed to by the cursors. In this way, it is possible to either determine a new candidate to be inserted in the priority queue or move forward one cursor, potentially skipping several documents in its inverted list.

```
intersect({S_0, ..., S_{k-1}})
    results = 0
    candidate = S_0[0]
    i = 1
    while candidate < # of documents
        for ; i < k; i = i + 1
            z = S_i.NextGEQ(candidate)
            if z ! = candidate
                candidate = z
                i = 0
                break
        if i == k
            results = results + 1
            candidate = S_0.Next()
            i = 1
    return results
```

Figure 5: The intersection algorithm.

2.3.2 Compression

Because of the huge quantity of indexed documents and heavy query loads, *compressing* the inverted index has become indispensable because, as motivated in Chapter 1, it can introduce a two-fold advantage over a non-compressed representation: feed faster memory levels with more data and, *hence*, speed up the query processing algorithms. As a result, the design of algorithms that *compress the index effectively while maintaining a noticeable decoding speed* is an old problem in Computer Science, that dates back to more than 50 years ago, and still a very active field of research.

Many representation for inverted lists are known, each exposing a different compression ratio vs. query processing speed trade-off. All the techniques we have described in Section 2.2 can be used to compress the inverted lists and, as a matter of fact, most of them were proposed exactly for this purpose. Therefore, we point the reader to Section 2.2 for a detailed overview of such mechanisms.

In particular, since we are dealing with inverted lists that are *monotonically increasing* by construction, we can subtract from each element the previous one (the first integer is left as it is), making the sequence be formed by integers greater than zero known as *delta*-gaps (or just *d*-gaps). This popular *delta encoding* strategy helps in reducing the number of bits for the codes. Most of the literature on inverted index compression assumes this sequence form.

Here, we mention an important property that the inverted lists exhibit and that most of the encoders described in Section 2.2 exploit to achieve good compression effectiveness. Indeed some of the proposals described in this thesis strictly depends on such property.

Property 1 — Inverted lists often contain *clusters* of close docIDs, e.g., *runs* of consecutive integers, that are far more compressible than highly scattered regions.

The reason for the presence of such clusters is that the indexed documents themselves tend to be clustered, i.e., there are subsets of documents sharing the very *same* set of terms. As a meaningful example, consider all the Web pages belonging to a certain domain (e.g., all the pages hosted by `di.unipi.it`): since their topic is likely to be the same, they are also likely to share a lot of terms.

Consequently, the compressors benefit a lot from docID-reordering strategies that focus on re-assigning the docIDs in order to form larger clusters of docIDs. An amazingly simple strategy, but very effective, for Web pages is to assign identifiers to documents according to the lexicographical order of their (reversed) URLs [148]. A recent approach has instead adopted a recursive graph bisection algorithm to find a suitable re-ordering of docIDs [53]. In this model, the input graph is a bipartite graph in which one set of vertices represents the terms of the index and the other set represents the docIDs. A graph bisection identifies a permutation of the docIDs and, thus, the goal is the one of finding, at each step of recursion, the bisection of the graph which minimizes the size of the graph compressed using delta-encoding.

2.4 N-Gram language models

A *language model* (LM) is a probability distribution $\mathbb{P}(\mathcal{W})$ that describes how often a string $w_1^n = w_1 \cdots w_n$ drawn from the set $\mathcal{W}$ 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 $w_1^n \in \mathcal{W}$. Using informal words, we would say that the goal is to *predict* the "next" word following a given context.

In what follows, let us indicate with w_i^j the *sequence* of words $w_i \cdots w_j$, for any $1 \le i \le j$, that 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 $\prod_{k=1}^{n} \mathbb{P}(w_k|w_1^{k-1})$, i.e., all contexts of length $1, 2, \ldots, n-1$ contribute to the final computed value. Therefore, computing such 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.

An *n-gram* is a sequence of n tokens. A token can be either a single character or a word, the latter intended as a sequence of characters delimited by a special symbol, e.g., a whitespace character. Unless otherwise specified, throughout the thesis 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., typically $N = 5$), dealing with strings of this form permits to work with a context of *at most* $N - 1$ preceding words. This ultimately implies that the aforementioned probability $\mathbb{P}(w_n|w_1^{n-1}) = \prod_{k=1}^{n} \mathbb{P}(w_k|w_1^{k-1})$ can be approximated with $\prod_{k=1}^{n} \mathbb{P}(w_k|w_{k-N-1}^{k-1})$.

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

Two fundamental problems are central to the handling of large and sparse n-gram language models:

- *estimation*, that is computing the probability distribution of the strings from a large textual source; and
- *indexing*, that is compressing the extracted n-gram strings and associated satellite data without compromising their retrieval speed.

Performing these two tasks efficiently is fundamental for several applications in the fields of Information Retrieval, Natural Language Processing and Machine Learning, such as: auto-completion in search engines [14, 113, 112], spelling correction [97], similarity search [96], identification of text reuse and plagiarism [144, 81], automatic speech recognition [90] and machine translation [76, 122], to mention some of the most notable.

For example, query auto-completion is one of the key features that any modern search engine offers to help users formulate their queries. The objective is to predict the query by saving keystrokes: this is implemented by reporting the top-k most frequently-searched n-grams that follow the words typed by the user [14, 113, 112]. The identification of such patterns is possible by traversing a data structure that stores the n-grams as seen by previous user searches. Given the number of users served by large-scale search engines and the high query rates, it is of utmost importance that such data structure traversals are carried out in a handful of microseconds [90, 14, 43, 113, 112]. Another noticeable example is spelling correction in text editors and web search. In their basic formulation, n-gram spelling correction techniques work by looking up every n-gram in the input string in a pre-built data structure in order to assess their existence or return a statistic, e.g., a frequency count, to guide the correction [97]. If the n-gram is not found in the data structure it is marked as a misspelled pattern: in such case correction happens by suggesting the most frequent word that follows the pattern with the longest matching history [90, 97, 43].

2.4.1 Estimation

Several language models have been proposed in the literature, such as Laplace, Good-Turing, Katz, Jelinek-Mercer, Witten-Bell and Kneser-Ney (see [36] and references therein for a complete description and comparison). For a n-gram backoff-smoothed language model, the probability of w_n with context w_1^{n-1} is assigned according to the following recursive equation

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

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

Clearly, the bigger the language model the more accurate the computed probability will be. In other words, predictions will be more accurate when more n-grams are used to estimate the probability of a word following a given context. Therefore, we would like to handle as many n-grams as possible: Chapter 8 and 9 describe techniques to handle several *billions* of n-grams.

The n-gram strings are extracted from *text*, from any of its different incarnations, e.g., web pages, books, code fragments and scientific articles, by adopting a *sliding-window* approach. A window of n words, for $1 \leq n \leq N$, slides over a text counting the number of times such n words appear in the text. This counting process is usually implemented using a hash data structure, whose keys are the distinct n-gram strings and the values are the accumulated frequency counts: if the extracted n-gram is not already present in the table, a new entry is allocated with associated value 1; otherwise the corresponding value is incremented by 1. This process is repeated for different widow sizes over huge text corpora: this gives birth to colossal datasets in terms of number of distinct strings.

As a concrete example, consider Table 4: we have already more than 250 thousands distinct grams for 164 pages written in English. Applying this process on approximately 8 million books, or 6% of all books ever published [105], resulted in a huge dataset of more than 11 billion N-grams (see also Table 28 at page 138).

n	# of n-grams
1	8761
2	38,900
3	61,516
4	70,186
5	73,187
total	252,550

Table 4: Number of distinct n-grams in the Agner Fog's manual *Optimizing software in* C++ [62].

2.4.2 Indexing

In this subsection we discuss the classic data structures used to represent efficiently large n-gram datasets, highlighting the advantages/disadvantages of these approaches in relation to the structural properties that n-gram datasets exhibit.

Two different data structures are mostly used to store n-grams datasets: *tries* [63] and *hash tables* [102].

Tries. A *trie* is a tree data structure devised for efficient indexing and search of string dictionaries, in which the *common prefixes* shared by the strings are represented *once* to achieve compact storage. This property makes this data structure useful for storing the n-gram strings in compressed space.

In this case, each constituent word of a n-gram is associated to 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 looks up for a string and, if present, retrieves the associated satellite value, e.g., a frequency count. Conceptually, a trie implementation has to store a *triplet* for any node: the associated word, satellite value and a pointer to each child node. As n is typically very small and each node has many children, tries are shallow and wide. Therefore, these are implemented as a collection of (few) *sorted* arrays: for each level of the trie, a separate array is built to contain all the triplets for that level, sorted by the words. In this implementation, a pair of adjacent pointers indicates the sub-array listing all the children for a word, which can be efficiently inspected by binary search.

Hash tables. Hashing is another way to implement associative arrays: for each value of n from 1 to N a separate hash table stores all grams of order n. At the location indicated by the hash function the following information is stored: a fingerprint value to lower the probability of a false positive (typically the 4 or 8-byte hash of the n-gram itself) and the satellite data for the n-gram. This data structure permits to access the specified n-gram data in expected constant time. Open addressing with linear probing is usually preferred over chaining for its better locality of accesses.

Tries are usually designed for space-efficiency as the formed sorted arrays are highly compressible. However, retrieval for the value of a n-gram involves exactly n searches in the constituent arrays. Conversely, hashing is designed for speed but sacrifices space-efficiency since its keys, along with their fingerprint values, are randomly distributed and, therefore, incompressible. Moreover, hashing is a *randomized* solution, i.e., there is a non-null probability of retrieving a frequency count for a n-gram *not* really belonging to the indexed corpus (false positive). Such probability equals 2^{-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.

3 Integer Dictionaries in Compressed Space

The efficient maintenance of a dynamic set on n integer keys is among the most studied problems in Computer Science (see the introduction to parts III and V of [42]). Many solutions for this problem are known to require an optimal amount of time per operation within polynomial space. For example, any self-balancing search tree data structure, e.g., AVL or Red-Black tree, solves the problem optimally in the comparison model [42], by implementing all operations in $O(\log n)$ worst-case time and using linear space. However, by exploiting the fact that the stored integers are drawn from a bounded universe of size $u \geq n$, the problem is known to admit more efficient solutions in terms of asymptotic time complexity while still retaining linear space [42, 136, 157, 165, 65, 66]. In this case, classical examples include the van Emde Boas tree [157, 158, 159], x/y-fast trie [165] and the fusion tree [66], that was the first data structure able to surpass the information-theoretic lower bound of $O(\log n)$, by exhibiting an improved running time of $O(\log_w n)$ per operation on a RAM with word size $w = \Theta(\log u)$ bits. Some efforts have been spent in trying to reduce the space requirements of the representation [75, 108, 142] but known compressed solutions do not closely match the information-theoretic lower bound of the underlying integer set. Whether time optimality can be preserved in the compressed space regime is still an interesting open problem. Therefore, the problem we consider in this chapter is the one of representing in compressed space a dynamic ordered set $\mathcal{S}$ of n integer keys drawn from a universe of size $u \geq n$.

In this chapter we show that it is possible to *preserve the optimal bounds* for the operations under *almost optimal space* requirements. The key ingredient of our data structures is the *Elias-Fano* representation of monotone integer sequences [56, 58] that we have described in details in Section 2.2.6. Here, we just recall that Elias-Fano encodes a monotone integer sequence $\mathcal{S}(n, u)$ in $\mathsf{EF}(\mathcal{S}(n, u)) \leq n\lceil\log\frac{u}{n}\rceil + 2n$ bits of space and supports the Access operation in $O(1)$ worst-case time. The query Predecessor$(x) = \max\{y \in \mathcal{S} : y < x\}$ is supported in $O(1 + \log\frac{u}{n})$ worst-case time. Throughout the paper we adopt the classical nomenclature and discuss the Predecessor query as it is well known that the twin Successor query can be answered in a similar way.

The natural question is whether it is possible to extend the *static* Elias-Fano representation to *dynamic* scenarios, in which integers can also be inserted (deleted) in (from) $\mathcal{S}$. To this end, we consider the case in which the n integers of $\mathcal{S}$ are drawn from a *polynomial universe* of size $u = n^\gamma$, for any $\gamma = \Theta(1)$. This is the classical operational setting considered by Fredman and Saks [65] in the so-called *List Representation Problem*, i.e., how to maintain $\mathcal{S}$ subject to the operations Access, Insert and Delete.

In order to characterize the asymptotic complexity of the data structures described in the paper and review the literature, we use the RAM model as described in Section 2.1.1 that is: we consider words of size $w = \Theta(\log u)$ bits and allow arithmetic operations on w bits in $O(1)$ time. We also adopt the usual trans-dichotomous assumption [66], making w grow with n as needed.

Our contributions. In this chapter we show how to represent the sequence $\mathcal{S}(n,u)$ with $\mathsf{EF}(\mathcal{S}(n,u)) + o(n)$ bits of space, hence introducing a *sublinear* space overhead with respect to its static Elias-Fano representation, and how:

- static predecessor queries can be supported in $O(\min\{1 + \log \frac{u}{n}, \log\log n\})$ worst-case time (note that the first term of the bound, i.e., $O(1 + \log \frac{u}{n})$, is optimal for polynomial universes of size $u = n^{\gamma}$ when γ is $1 \leq \gamma \leq 1 + \log\log n/\log n$);

- to maintain $\mathcal{S}(n,u)$ in an append-only fashion, i.e., by inserting the integers in sorted order in constant time;

- to maintain $\mathcal{S}(n,u)$ in a fully dynamic way, supporting random access in $O\left(\frac{\log n}{\log\log n}\right)$ worst-case, insertions and deletions in $O\left(\frac{\log n}{\log\log n}\right)$ amortized and predecessor in $O(\min\{1 + \log \frac{u}{n}, \log\log n\})$ worst-case time.

The mentioned time bounds are optimal [65, 134, 136].

3.1 Related work

We organize the discussion of the related work in two parts: the first concerns the review of the results about the static predecessor problem, whereas the second one describes the results closest to our work for the maintenance of a dynamic integer set.

The static predecessor problem. We could solve the static predecessor problem in $O(1)$ worst-case by storing the results to every possible query using *perfect hashing* [64] in $O(u)$ words of space. In order to not trivialize the problem, we assume to have a polynomial space budget, i.e., we deal with a data structure occupying $O(n^{O(1)})$ memory words.

Ajtai [7] proved the first $\omega(1)$ lower bound for polynomial space, claiming that $\forall w$, $\exists n$ that gives $\Omega(\sqrt{\log w})$ query time. Only ten years later Beame and Fich

[15, 16] proved two strong bounds for any *cell-probe* data structure[1]. They proved that $\forall w$, $\exists n$ that gives $\Omega(\log w/\log\log w)$ query time and that $\forall n$, $\exists w$ that gives $\Omega(\sqrt{\log n/\log\log n})$ query time. They also gave a static data structure achieving $O(\min\{\log w/\log\log w, \sqrt{\log n/\log\log n}\})$ which is, therefore, optimal. Building on a long line of research, Pătraşcu and Thorup [134, 135] finally proved that

$$\Theta\Big(\min\Big\{\log_w n, \log\frac{w-\log n}{a}, \frac{\log\frac{w}{a}}{\log(\frac{a}{\log n}\log\frac{w}{a})}, \frac{\log\frac{w}{a}}{\log(\log\frac{w}{a}/\log\frac{\log n}{a})}\Big\}\Big) \tag{3}$$

is an *optimal* space/time trade-off for a *static* data structure taking $m = n2^a w$ bits of space, where $a = \log\frac{m}{n} - \log w$. This lower bound holds for cell-probe, RAM, trans-dichotomous RAM, external memory and communication game models. The first branch of the trade-off indicates that, whenever we are in RAM or external memory with one integer fitting in one memory word, fusion trees [66] are optimal, as these require $O(\log_w n) = O(\log n/\log w)$ query time. The second branch holds for *polynomial universes*, i.e., whenever $u = n^\gamma$, for any $\gamma = \Theta(1)$. In such case we have that $w = \Theta(\log u) = \gamma\log n$, therefore y-fast tries [165] and van Emde Boas trees [157, 158, 159] are optimal with query time $O(\log\log n)$. Finally, the last two branches of the trade-off treat the case for *super-polynomial universes*. In particular, the third branch matches the lower bound by Beame and Fich [15, 16] that requires $n^{O(1)}$ words of space. Finally, the fourth branch improves the latter space occupancy, showing that $n^{1+1/\exp(\log^{1-\epsilon}\log u)}$ words are sufficient for any $\epsilon > 0$.

Dynamic problems. We now review the most important results concerning the maintenance of a *dynamic* set of integers, following the chronological order of their proposal. We will reuse some of these results later on in this chapter.

The van Emde Boas tree is a recursive data structure that maintains $\mathcal{S}$ in $O(u)$ words of space and supports the operations: Search. which tests whether a given integer is present or not in $\mathcal{S}$, Insert, Delete and Predecessor all in $O(\log w)$ worst-case time [157, 158, 159]. Willard [165] improved the space bound to $O(n)$ words by introducing the *y-fast trie* that supports Search and Predecessor queries in $O(\log w)$ worst-case time, Insert and Delete in amortized $O(\log w)$ time.

The work by Fredman and Saks [65] is useful to understand which lower bounds apply to the problem we consider in this chapter. They proved that the *List Representation Problem*, i.e., maintaining $\mathcal{S}$ under Access, Insert and Delete, can be solved in $\Omega(\log n/\log\log n)$ amortized time per operation if $w \le \log^\gamma n$ for some γ.

[1] In the cell-probe computational model, described by Yao [171], computation is for free given that we only take into account *word reads*. It is not a very realistic model of computation, but it is useful to prove lower bounds because it is a stronger model than RAM and trans-dichotomous RAM.

No space bound is posed on such problem. Their lower bound does not apply to dynamic predecessor queries and holds for the cell-probe computational model [171]. Extending the result to the dynamic predecessor problem, they proved that any cell-probe data structure representing $\mathcal{S}$ using $(\log u)^{O(1)}$ bits per memory cell and $n^{O(1)}$ worst-case time for insertions, requires $\Omega(\sqrt{\log n/\log\log n})$ worst-case query time. They also proved that on a RAM, the dynamic predecessor problem can be solved in $O(\min\{\log\log n \cdot \log w/\log\log w, \sqrt{\log n/\log\log n}\})$, using $O(n)$ words. This bound was matched by Andersson and Thorup [8] with the so-called *exponential search tree*. This data structure has an optimal bound of $O(\sqrt{\log n/\log\log n})$ worst-case time for searching and updating $\mathcal{S}$, using polynomial space.

Raman, Raman, and Rao [138] also addressed the List Representation Problem[2] for arrays of length n by providing two solutions. Their first data structure supports Access in $O(1)$ and Insert/Delete in $O(n^{\epsilon})$ worst-case time for any fixed positive $\epsilon < 1$; the second data structure implements all the three operations in $O(\log n/\log\log n)$ amortized time. Both data structures use $o(n)$ bits of redundancy and the time bounds are optimal.

Fredman and Willard [66] showed that dynamic predecessor queries can be answered in $O(\log n/\log\log n)$ time by using the *fusion tree*. This data structure is a B-tree with branching factor $B = \Theta(\log n)$ that stores in each internal node a *fusion node*, a small data structure able of answering predecessor queries in $O(1)$ for sets up to $w^{1/5}$ integers. Updating a fusion node takes, however, $O(B^4)$ time. The overall space of the data structure is $O(n)$ words. The work by Pătraşcu and Thorup [136] has recently shown that it is possible to "dynamize" the fusion node, in order to support Insert and Delete in $O(1)$. As a result, they have proposed a data structure representing $\mathcal{S}$ in $O(n)$ words and optimal [15, 16] $O(\log n/\log w)$ running time for the following operations: Insert, Delete, Predecessor, Successor, Rank and Select. Pătraşcu and Thorup [134] also proved that y-fast tries and van Emde Boas trees have an optimal query time for the dynamic predecessor problem too, as long as polynomial universes are considered.

We conclude this section by mentioning few additional results, that will be useful in the following. Bille et al. [20] recently combined the static solution of Demaine and Pătraşcu [51] with the one by Pătraşcu and Thorup [136] to support *dynamic prefix sums* over an array of size n in optimal $O(\log n/\log(w/\delta))$ time per operation and linear space, where δ is the number of bits needed to encode the quantity that we sum to the elements of the array.

Though not devised for integer sets, the *extended CRAM* (Compressed Random Access Memory) data structure described by Jansson, Sadakane, and Sung [89] allows a string $\mathcal{S}$ of length n to be stored using its k-th order empirical entropy,

[2] In their paper [138], the authors refer to the List Representation Problem of Fredman and Saks [65], as the *Dynamic Array Problem*. Also, the operation Access is named Index.

$nH_k(\mathcal{S})$, plus a redundancy of $O(n \log \sigma (k \log \sigma + (k+1) \log\log n)/\log n)$ bits for every $0 \leq k < \log_\sigma n$, where σ is the size of the alphabet. Insertions/deletions of characters and Access to any consecutive $\log_\sigma n$ bits are all supported in optimal $O(\log n/\log\log n)$ worst-case time. We will exploit the part of this work dedicated to the memory management (Appendix A). Finally, Grossi, Raman, Rao, and Venturini [74] improved the previous space bound by using $nH_k(\mathcal{S}) + O\left(n\frac{\log\log n}{\log_\sigma n}\right)$ bits and maintaining the asymptotic optimality for all operations.

The paper by Navarro and Nekrich [118] illustrates a data structure supporting Access, Rank and Select queries, as well as insertions/deletions of symbols of $\mathcal{S}$ in optimal $O\left(\frac{\log n}{\log\log n}\right)$ time and taking $nH_0(\mathcal{S}) + O(n + \sigma(\log \sigma + \log^{1+\epsilon} n))$ bits of space. Of particular interest for our purposes, is the data structure described in Appendix A.1 concerning the organization of data in small blocks. The high-level idea is to maintain a tree of constant height with node degree $\log^\delta n$, for some $0 < \delta < 1$, and leaves containing $o(\log n)$ elements each. As each internal node can fit in one machine word, the tree supports basic search operations in $O(1)$ time by using a small pre-computed table. In order to handle small blocks of sorted integers, we will make use of a similar data structure in Section 3.4.

3.2 Static predecessor queries in optimal time

In this section we are interested in determining the optimal running time of the query Predecessor for the Elias-Fano space bound of $\mathsf{EF}(\mathcal{S}(n,u)) \leq n\lceil \log \frac{u}{n} \rceil + 2n$ bits in Equation 2. As already mentioned, our focus is on polynomial universes, i.e., $u = n^\gamma$ for any $\gamma = \Theta(1)$, for which the second branch of the time/space trade-off in Equation 3 is optimal.

The following theorem shows that adding $o(n)$ bits of redundancy to $\mathsf{EF}(\mathcal{S}(n,u))$ is enough to support Predecessor queries in optimal time.

Theorem 1 — There exists a data structure representing an ordered set $\mathcal{S}(n,u)$ of n integers drawn from a polynomial universe of size $u = n^\gamma$, for any $\gamma = \Theta(1)$, that takes $\mathsf{EF}(\mathcal{S}(n,u)) + o(n)$ bits of space and supports: Access in $O(1)$ worst-case and Predecessor queries in optimal $O(\min\{1 + \log \frac{u}{n}, \log\log n\})$ worst-case time.

Discussion. The parameter a in (3) is equal to $\log(\lceil \log \frac{u}{n} \rceil + 2)$ for our space budget and, given that $w = \Theta(\log u) = \gamma \log n$ bits, the second branch of the trade-off becomes

$$\log \frac{w - \log n}{a} = \log \frac{(\gamma - 1)\log n}{\log(\lceil (\gamma-1)\log n \rceil + 2)} = O(\log\log n),$$

that shows that the running time of Predecessor for y-fast tries and van Emde Boas trees is optimal within the Elias-Fano space bound.

However, such bound of $O(\log\log n)$ only depends on n, whereas the plain Elias-Fano bound for Predecessor of $O(1+\log\frac{u}{n})$ depends on *both* n and u. On the other hand, the relation $u = n^{\gamma}$ relates the two parameter by means of the constant $\gamma = \Theta(1)$. It is clear that varying γ only one of the two bounds is optimal. Indeed, comparing $1+\log\frac{u}{n}$ with $\log\log n$, we have that $1+\log\frac{u}{n} \leq \log\log n$ whenever $u \leq \frac{n}{2}\log n$, i.e., when $n^{\gamma-1} \leq \frac{1}{2}\log n$. From this last condition we derive that the plain Elias-Fano is faster than van Emde Boas whenever $1 \leq \gamma \leq 1+\frac{\log\log n}{\log n}$. In this case the static Elias-Fano representation does *not* need to be augmented. When, instead, $\gamma > 1+\frac{\log\log n}{\log n}$, the query time $O(\log\log n)$ is optimal and *exponentially better* than plain Elias-Fano. Therefore, $O(\min\{1+\log\frac{u}{n}, \log\log n\})$ is an accurate characterization of the Predecessor time bound, within $\mathsf{EF}(\mathcal{S}(n,u)) + o(n)$ bits of space.

Data structure. We are now left to describe a data structure that matches the bound of $O(\log\log n)$ and uses $\mathsf{EF}(\mathcal{S}(n,u)) + o(n)$ bits of space. We divide $\mathcal{S}$ into $\lceil n/\log^2 u\rceil$ blocks of $\log^2 u$ integers each (the last block may contain less integers). We can solve Predecessor queries in a block in $O(\log\log u) = O(\log\log n)$ time by applying binary search. Now, we need a data structure on top of $\mathcal{S}$ that allows us to identify the proper block in $O(\log\log n)$ time.

Call the *first* element of a block its *lower bound*. We attach to $\mathcal{S}$ an y-fast trie storing the lower bounds of the blocks. More precisely, each leaf in the y-fast trie holds the lower bound of a block and its position in $\mathcal{S}$. The integers stored in the y-fast trie are $\lceil n/\log^2 u\rceil$, therefore the space of the trie is $O(\frac{n}{\log^2 u}\log u) = o(n)$ bits. To identify the block where the predecessor of x is located, we answer a partial Predecessor(x) query among the integers stored in the y-fast trie in $O(\log\log n)$ worst-case time. The position p in $\mathcal{S}$ of the block's lower bound, associated to the identified partial answer, indicates that the search must continue in the block $\mathcal{S}[p, \min\{p+\log^2 u, n\})$.

Observe again that the time bound for Predecessor is always at most $O(\log\log n)$ except when $1 \leq \gamma \leq 1+\frac{\log\log n}{\log n}$: in this case, the plain Elias-Fano representation beats the time bound of $O(\log\log n)$. Therefore, in what follows we report the bound as $O(\min\{1+\log\frac{u}{n}, \log\log n\})$ but discuss the case for $\gamma > 1+\frac{\log\log n}{\log n}$.

3.3 Extensible representation

When the integers are inserted in sorted order, we obtain an efficient *extensible* representation because these can only be added at the end of $\mathcal{S}$ by using an Append operation. This is a scenario of practical interest as it is the operational setting of append-only inverted indexes, e.g., the one of Twitter [29].

In this section we aim at illustrating the following theorem.

Theorem 2 — There exists a data structure representing an ordered set $\mathcal{S}(n, u)$ of n integers, drawn from a polynomial universe of size $u = n^\gamma$, for any $\gamma = \Theta(1)$, that takes $\mathsf{EF}(\mathcal{S}(n, u)) + o(n)$ bits of space and supports: Append in $O(1)$ amortized, Access in $O(1)$ worst-case and Predecessor queries in optimal $O(\min\{1 + \log \frac{u}{n}, \log \log n\})$ worst-case time.

Discussion. We maintain an array B of size m in which integers are appended uncompressed. This array acts as a buffer, which is periodically encoded with Elias-Fano in $\Theta(m)$ time and overwritten. Each compressed representation of the buffer is appended to an array of blocks encoded with Elias-Fano. More precisely, when B becomes full we encode with Elias-Fano its corresponding *differential* buffer, i.e., the buffer whose values are $B[i] - B[0]$, $0 \leq i < m$. Each time the buffer is compressed, we append in another array C the pair $(base, low) = (B[0], \lceil \log(B[m-1]/m) \rceil)$, i.e., the buffer lower bound value ($base$) and the number of bits (low) needed to encode the average gap of the Elias-Fano representation of the buffer.

Space analysis. Apart from the space taken by the compressed Elias-Fano blocks, the space of the data structure is given by the following contributions:

- $(m + 1) \log u$ bits for the buffer B of uncompressed integers and its size;
- $O(\frac{n}{m} \log n)$ bits for pointers to rank/select data structures, low and high bit arrays;
- $O(\frac{n}{m} \log u)$ bits for the array C.

Summing up, the redundancy is $O((m + \frac{n}{m} + 1) \times \log u)$ bits. We set $m = \log^2 u$ and, as done in Section 3.2, we index the buffer lower bounds in an y-fast trie. More precisely, each leaf of the fast trie stores a buffer lower bound and the index of the compressed block to which the lower bound belongs to. The values stored in the y-fast trie are $\lceil n/\log^2 u \rceil$, thus taking $o(n)$ bits of space. The redundancy $O((m + \frac{n}{m} + 1) \times \log u)$ bits becomes $o(n)$ bits for $n = \omega(\log^3 u)$, which is already satisfied by requiring that $\gamma = \Theta(1)$.

To analyze the space taken by the representation of the blocks, we use the property that splitting a block encoded with Elias-Fano into two sub-blocks *never* increases the cost of representation of the block. This is possible because each sub-block can be encoded with a universe *relative* to the sub-block, which is smaller than the original block universe, by subtracting to each integer the lower bound (first value) of the sub-block[3]. Doing so, we say that each sub-block has been "re-mapped" relatively to its universe. The following property can be easily extended to work with an arbitrary number of splits.

Property 2 — Given a monotone sequence $\mathcal{S}$ of n integers, let $\mathcal{S}[i, j)$ indicate the range of $\mathcal{S}$ delimited by endpoints i and j. Then for any i, k and j such that $0 \leq i < k < j < n$, we have that $\mathsf{EF}(\mathcal{S}[i,k)) + \mathsf{EF}(\mathcal{S}[k,j)) \leq \mathsf{EF}(\mathcal{S}[i,j))$.

Proof. Let m and u be respectively size and universe of the sub-sequence $\mathcal{S}[i, j)$, and, similarly, let m_1, m_2, u_1, u_2 be the sizes and universes of the two sub-sequences $\mathcal{S}[i,k)$ and $\mathcal{S}[k,j)$ respectively. We have that $m = m_1 + m_2$ and $u = u_1 + u_2$. From Section 2.2.6, we know that $\mathsf{EF}(\mathcal{S}[i,j))$ takes $m\phi + m + \lceil \frac{u}{2^\phi} \rceil$ bits. Similarly $\mathsf{EF}(\mathcal{S}[i,k)) = m_1\phi_1 + m_1 + \lceil \frac{u_1}{2^{\phi_1}} \rceil$ bits and $\mathsf{EF}(\mathcal{S}[k,j)) = m_2\phi_2 + m_2 + \lceil \frac{u_2}{2^{\phi_2}} \rceil$ bits. $\mathsf{EF}(\mathcal{S}[i,k))$ and $\mathsf{EF}(\mathcal{S}[k,j))$ are minimized by setting $\phi_1 = \lfloor \log \frac{u_1}{m_1} \rfloor$ and $\phi_2 = \lfloor \log \frac{u_2}{m_2} \rfloor$ respectively [56], therefore, by replacing ϕ_1 and ϕ_2 with ϕ, we have that $\mathsf{EF}(\mathcal{S}[i,k)) + \mathsf{EF}(\mathcal{S}[k,j)) \leq m_1\phi + m_2\phi + m_1 + m_2 + \lceil \frac{u_1}{2^\phi} \rceil + \lceil \frac{u_2}{2^\phi} \rceil = m\phi + m + \lceil \frac{u}{2^\phi} \rceil = \mathsf{EF}(\mathcal{S}[i,j))$. ■

The above property guarantees that the space taken by the blocks encoded with Elias-Fano can be safely upper bounded by $\mathsf{EF}(\mathcal{S}(n,u))$ so that the overall space of the data structure is at most $\mathsf{EF}(\mathcal{S}(n,u)) + o(n)$ bits.

Operations. The operations are supported as follows. Since we compress the buffer each time it fills up (by taking $\Theta(m)$ time), Append is performed in $O(1)$ amortized time. Appending new integers in the buffer accumulates a credit of $O(\log^2 u)$ which pays (by a large margin) the amortized cost $O(\log\log u)$ of inserting a buffer lower bound into the y-fast trie. To Access the i-th integer, we retrieve the element x in position $i - k \times m$ from the compressed block of index $k = \lfloor \frac{i}{m} \rfloor$. This is done in $O(1)$ worst-case time, since we know how many low bits are required to perform the access by reading $C[k].low$. We finally return the integer $x + C[k].base$. Predecessor queries are supported similarly as in the description of Theorem 1. Given the integer x, we first resolve a partial Predecessor(x) query in the y-fast trie to identify the index k of the compressed block in which the predecessor is located. Then we return $C[k].base$ + Predecessor($x - C[k].base$) by binary searching the block of index k in $O(\log\log u) = O(\log\log n)$ worst-case time.

[3]Or by subtracting the upper bound (last value) of the *preceding* sub-block. In this chapter, we prefer to subtract the lower bounds because it slightly simplifies the design of the data structures.

Further considerations. From Theorem 2, the following lemma easily follows.

Lemma 1 — There exists a data structure representing an ordered set $\mathcal{S}(n,u)$ of $n = \omega(\log^2 u)$ integers drawn from a universe of size u that takes $\mathsf{EF}(\mathcal{S}(n,u)) + o(n)$ bits of space and supports Append and Access operations in $O(1)$ worst-case time.

Without using the y-fast trie we are able to achieve a worst-case running time for the Append operation in Lemma 1 by using a classical de-amortization argument (note that, however, Predecessor queries are not supported in optimal time anymore). We maintain two buffers, B_1 and B_2, instead of only one. When one is full we use the other to store the elements that must be appended. Suppose B_1 is full. For each of the successive m Append operations, we compress one element from B_1 and append the new integer in B_2. These two steps require $O(1)$ worst-case time each.

3.4 Dynamic representation

In this section we describe how to exploit the space efficiency of the Elias-Fano representation in order to obtain a compressed *dynamic* data structure that supports random access, insertions, deletions and predecessor queries in optimal time, taking $\mathsf{EF}(\mathcal{S}(n,u)) + o(n)$ bits of space.

As already discussed in the related work, Fredman and Saks [65] proved that $O\left(\frac{\log n}{\log\log n}\right)$ amortized time is optimal for any data structure maintaining a set of integers subject to Access, Insert and Delete (List Representation Problem). Their result holds when $w \le \log^{\gamma} n$ for some γ, which covers the case for polynomial universes, that is, $u = n^{\gamma}$, since $\gamma \le \log^{\gamma-1} n$, for any $\gamma \ge 1$ and $n \ge 2$. We operate, therefore, in the same setting as Theorems 1 and 2, considering integers drawn from a polynomial universe of size $u = n^{\gamma}$, for any $\gamma = \Theta(1)$. In this setting, Pătraşcu and Thorup [134] showed that the $O(\log\log n)$ query time of y-fast tries and van Emde Boas trees is optimal for the dynamic predecessor problem too.

We will now show the following theorem which claims we can attain to the optimal operational bounds mentioned above in compressed space.

Theorem 3 — There exists a data structure representing an ordered set $\mathcal{S}(n,u)$ of n integers drawn from a polynomial universe of size $u = n^{\gamma}$, for any $\gamma = \Theta(1)$, that takes $\mathsf{EF}(\mathcal{S}(n,u)) + o(n)$ bits of space and supports: Access in $O(\log n/\log\log n)$ worst-case; Insert and Delete in $O(\log n/\log\log n)$ amortized; Predecessor queries in $O(\min\{1 + \log\frac{u}{n}, \log\log n\})$ worst-case. These time bounds are optimal.

In what follows, we first describe the layout of the data structure, then analyse its space and time complexities.

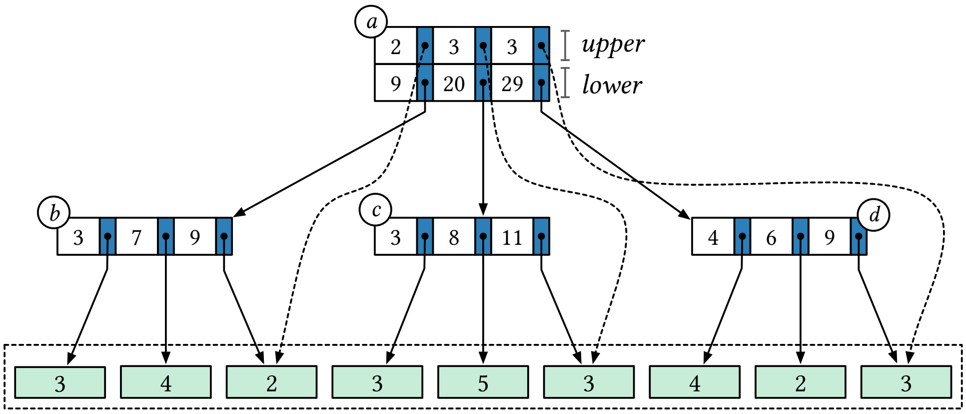


Figure 6: A graphical example of the tree data structure described in Lemma 2, indexing a collection C of 9 sorted blocks, represented as gray rectangles. The numbers in the blocks indicate their sizes. The number of integers stored in the sub-tree rooted in the (only) internal node a of the example are, respectively, 9, 11 and 9, which are kept in prefix sums as the sequence $\langle 9, 20, 29\rangle$ and stored in the lower part of the node. For each child, we also store the pointer (dashed arrow) to the right-most block indexed in its sub-tree and its size. In the example, this information is stored in the upper part of the node a: the sequence of the children sizes is $\langle 2, 3, 3\rangle$. The leaves of the tree (nodes b, c and d in the example) store, instead, the sizes of the blocks in prefix sums and pointers to the blocks.

3.4.1 A BASIC TOOL: SORTED BLOCKS IN SUCCINCT SPACE

We begin our description by showing how to handle a dynamic collection of mini blocks in succinct space, which is a key tool to obtain the full dynamic data structure. This result builds on an idea from [118], Appendix A.1.

Let C be a collection of $k = O(\operatorname{polylog} n)$ blocks of sorted integers, with the following properties. The blocks of C form a *total order*, i.e., $u_j \leq f_{j+1}$, for all $j = 1, \ldots, k-1$, where f_j and u_j indicate, respectively, the first and last element of the j-th block in the total order. Each block supports random access to its elements and is of size $\Theta(b) = \rho b$ with $\frac{1}{2} \leq \rho \leq 2$ and $b = O(\operatorname{polylog} n)$.

Lemma 2 — The total order of the blocks of C can be maintained by using a data structure that takes $O(\operatorname{polylog} n \times \log\log n)$ bits of space and supports the following operations in $O(\log\log n)$ worst-case time: Search(x) ,which returns a pointer to the block containing the integer x; Access(i), which returns the i-th integer of the total order; Insert and Delete of a block.

Data structure. Pointers of $O(\log\log n)$ bits to blocks are stored in the leaves of a τ-ary tree $\mathcal{T}$, with $\tau = \Theta(\log^{\sigma} n)$ for some $0 < \sigma < 1$. Given that we have $O(\operatorname{polylog} n)$ leaves, the height of $\mathcal{T}$ is constant and equal to $O(1/\sigma)$. Therefore, $\mathcal{T}$ operates as a B-tree, in which internal nodes have $\Theta(\tau) = \rho\tau$ children.

Logically, we divide the information stored at each *internal node* of the tree into two levels of representation. For each of the two levels we store $\Theta(\tau)$ pairs, where the i-th pair maintains information about the sub-tree rooted in the i-th child. The pairs are stored following the order of the upper bounds of the blocks indexed in the sub-trees rooted in the node's children. In the lower level, each pair contains a pointer to the sub-tree rooted in the child and the size of such sub-tree. The $\Theta(\tau)$ children sizes are kept in prefix sums to enable binary search. In the upper level, each pair contains a pointer to the *right-most* block indexed in the sub-tree rooted in its child and the size of such sub-tree. Clearly, each *leaf* only stores the lower level. Refer to Figure 6 for a pictorial example.

Space analysis. Each node uses $O(\tau \times (\log\log n + \log \text{polylog}\, n)) = O(\tau \log\log n) = o(\log n)$ bits, thus fitting in (less than) a machine word. The space taken by whole data structure is, therefore, $O(\tau^{O(1/\sigma)} \log\log n) = O(\text{polylog}\, n \times \log\log n)$ bits.

Operations. To support Search(x), i.e., determining the block where the integer x is comprised, we percolate $\mathcal{T}$, locating the correct child at each node in $O(\log\tau) = O(\log\log n)$ by binary searching on blocks' upper bounds. Specifically, if the upper bounds of the i-th block is needed for comparison for some $1 \leq i \leq \Theta(\tau)$, we access the block following the pointer (to the right-most block) of the i-th pair stored in the upper level of the node and we retrieve the upper bound in $O(1)$, given that we also know the size of the block.

When we have to insert (delete) an integer, we identify the proper block of the total order in (from) which the integer must be inserted (deleted) in $O(\log\log n)$ time (as described for the Search operation) and update the pairs along the path from the root in constant time, as these pairs fits in $o(\log n)$ bits overall. If a split or merge of a block happens, it is handled as usual and solved in a constant number of $O(1)$-time operations.

During an Access(i) query, we follow the proper root-to-leaf path in $\mathcal{T}$. The traversal of the data structure does not need to access the blocks directly, but instead uses their sizes to determine the correct child at each level. By binary searching the sizes, we traverse the data structure in $O(\log\log n)$ time[4]. During the traversal of the path we also compute the sum Δ of the sizes of the preceding blocks by summing to the current value of Δ, at each level, the value stored in the $(j-1)$-th pair of the lower level if the j-th child is traversed. Finally we retrieve the $(i-\Delta)$-th integer from the identified block in $O(1)$, as the blocks of the collection C support random access by assumption.

[4]Since the sizes fit in less than a machine word, we could identify the correct child at each level in $O(1)$ by using a small universal table.

3.4.2 Data structure

Let ℓ be $\log n/\log\log n$. We logically divide the sorted sequence $\mathcal{S}(n,u)$ into mini blocks of $\Theta(\ell) = \rho\ell$ integers each. We organize the dynamic layout into two levels.

(1) **Lower level.** We group $O(\log^2 n)$ consecutive mini blocks together and index such collection using the data structure $\mathcal{T}$ described in Lemma 2. In the following, we refer to this collection as a "block" and say that $\mathcal{T}$ stores a "block of $O(\log^2 n)$ mini blocks". The set $\{\mathcal{T}_j\}_{j=1}^{k'}$, with $k' = n/O(\ell\log^2 n)$, of all such data structures forms the *lower level* of the dynamic layout. Each $\mathcal{T}_j$ also stores the lower bound f_j of its block and the number of low bits required by its Elias-Fano representation in $\Theta(\log u)$ bits, so that we can subtract f_j to all the integers belonging to the mini blocks of $\mathcal{T}_j$.

(2) **Upper level.** The set $\{f_j\}_{j=1}^{k'}$ of all the lower bounds of the blocks are indexed using an y-fast trie. The sizes of the blocks are maintained, instead, using the dynamic prefix sums data structure described in [20], i.e., a B-tree in which each node stores a dynamic prefix sums data structure operating on a small set of integers in $O(1)$ time. In particular, we use the operation Update(i,Δ) as described in [20], which sums to the i-th integer of the data structure the quantity Δ (that fits in δ bits) and runs in optimal $O(\log n/\log(w/\delta))$ worst-case time. In our setting this operation is supported in $O(\ell)$ given that $\delta = \Delta = 1$.

These two data structures, i.e., the y-fast trie and the dynamic prefix sums data structure, form the *upper level* of the dynamic layout. We will indicate them with $\mathcal{Y}$ and $\mathcal{P}$, respectively. The j-th leaf of $\mathcal{Y}$ and $\mathcal{P}$ stores a $O(\log n)$-bit pointer to the data structure $\mathcal{T}_j$ in the lower level.

To handle the memory allocation for the mini blocks, we employ a different technique to manage the high and low part of their Elias-Fano representation. Recall from Section 2.2.6 that, given a sequence $\mathcal{S}(n,u)$, the high part of $\mathsf{EF}(\mathcal{S}(n,u))$ consists in a bitvector of at most $2n$ bits, whereas the low part is given by a vector of n integers, each taking $\lceil\log\frac{u}{n}\rceil$ bits. In our case, the high part of each mini block requires at most $2\ell = O(w)$ bits and is stored using the data structure of Theorem 6 from [89] that allows to address and allocate the high part of a mini block in $O(1)$ worst-case time. The low part of a mini block is instead stored using the data structure of Corollary 3 from [138] that supports Access in $O(1)$ and Insert/Delete in $O(\ell^\epsilon)$ worst-case time for any fixed positive $\epsilon < 1$.

3.4.3 Space analysis

The space required by the introduced layout will be clearly given by the contribution of: 1. the data structures $\mathcal{Y}$ and $\mathcal{P}$ used in the upper level and the data structures $\mathcal{T}$ of Lemma 2 used in the lower level; 2. the cost of representation of the mini blocks encoded with Elias-Fano; 3. the overhead given by the mini blocks memory management.

In the following we separately analyze each contribution.

Lower and upper levels. The space taken by the data structures $\mathcal{Y}$ and $\mathcal{P}$ in the upper level is $O(\frac{n}{\ell \log^2 n} \times \log u) = o(n)$ bits. All the data structures $\mathcal{T}$ of Lemma 2 require $O(\frac{n}{\ell \log^2 n} \times \log^2 n \log \log n) = o(n)$ bits too.

Mini blocks. We now analyze the space taken by the encoding of the mini blocks. Since the universe of representation of a mini block could be as large as the one of its comprising block, i.e., u, storing the lower bounds of the mini blocks in order to use reduced universes (as already done for the blocks), would require $O(\frac{n}{\ell} \log u)$ bits, which is not $o(n)$ bits. In what follows we show that it is not necessary to re-map the mini blocks using Property 2, hence these are kept encoded with the universe relative to their comprising block, if we carefully set the number of bits required to represent each *low part* in the Elias-Fano space bound (Equation 2). As pointed out previously, each low part in the Elias-Fano representation of a sequence $\mathcal{S}(n, u)$ is encoded using $\lceil \log \frac{u}{n} \rceil$ bits, which is the number of bits needed to encode the average gap u/n of $\mathcal{S}$. The number of bits for the average gap of a block is therefore $\lceil m \rceil = \lceil \log \frac{u}{\ell \log^2 n} \rceil$.

The idea is to choose a number of bits $\lceil m' \rceil$ for the encoding of the average gap of the mini blocks such that $\lceil m' \rceil = \lceil m \rceil$ for a sufficiently long sequence of p insertions/deletions. After p insertions/deletions have been performed, we rebuild the mini blocks using $\lceil m \rceil$ bits for the average gap. In other words, we want to guarantee that encoding the mini blocks with $\lceil m' \rceil$ bits for the average gap, instead of $\lceil m \rceil$, does *not* introduce any extra space. Since m' lies in the interval $[l, r] = [\log \frac{u}{\ell \log^2 n+p}, \log \frac{u}{\ell \log^2 n-p}]$, m' must be chosen in order to satisfy $\lceil m \rceil - 1 < m' < \lceil m \rceil$, which indeed implies $\lceil m' \rceil = \lceil m \rceil$. Precisely, we satisfy this condition by fixing $m' = m \pm \theta$ with $\lceil m \rceil - l < \pm\theta < \lceil m \rceil - r + 1$. To derive this condition, we distinguish three possible cases.

(1) $[l, r] \subset [\lceil m \rceil - 1, \lceil m \rceil)$. In this case the condition $\lceil m \rceil - 1 < m' < \lceil m \rceil$ is already satisfied. The other two cases below are symmetric.

(2) $\lceil l \rceil = \lceil m \rceil - 1$. In this case we set $m' = m + \theta$. To let $\lceil m \rceil - 1 < m' < \lceil m \rceil$ holds, θ must be at least $\lceil m \rceil - l$ and at most $\lceil m \rceil + 1 - r$.

(3) $\lceil r \rceil = \lceil m \rceil + 1$. In this case we set $m' = m - \theta$. To let $\lceil m \rceil - 1 < m' < \lceil m \rceil$ holds, θ must be at least $r - \lceil m \rceil - 1$ and at most $l - \lceil m \rceil$.

Cases (2) and (3) together yield the condition $\lceil m \rceil - l < \pm\theta < \lceil m \rceil - r + 1$.

Finally, we have to determine the proper number p of insertions/deletions before triggering the rebuilding of the mini blocks in order to attain to optimal insert/delete amortized time $O(\ell)$. As blocks are of size $\Theta(\ell \log^2 n)$, p is chosen to be $O(\log^2 n)$.

Memory allocation. The techniques used to manage the memory allocation for the mini blocks introduce an overall redundancy of $o(n)$ bits. Precisely, the data structure of Theorem 6 from [89] has an overhead of $O(w^4 + \frac{n}{\log n} \log^2 w) = o(n)$ bits, while the one of Corollary 3 from [138] uses $o(n)$ bits by choosing a proper positive $\epsilon < 1$.

In conclusion, by the above discussion and the use of Property 2, the space taken by the mini blocks can be safely upper bounded by $\mathsf{EF}(\mathcal{S}(n, u))$ and the redundancy sums up to $o(n)$ bits, so that the whole data structure requires $\mathsf{EF}(\mathcal{S}(n, u)) + o(n)$ bits of space.

3.4.4 Operations

In this subsection we describe how the operations of Theorem 3, namely Access, Insert, Delete and Predecessor queries are supported. As stated before, we will use ℓ as a short-hand for $\log n / \log\log n$.

- To Access the i-th integer, we first resolve Search(i) on $\mathcal{P}$ in $O(\ell)$: Search(i) = j indicates that the j-th block contains the i-th integer given that Sum($j-1$) $< i \leq$ Sum(j), where Sum(j) equals the sum of the sizes of the first j blocks. We then follow the pointer stored in the j-th leaf of $\mathcal{P}$, which points to the data structure $\mathcal{T}_j$. We finally access the integer x of index $i -$ Sum($j-1$) from $\mathcal{T}_j$ in $O(\log\log n)$ and return $x + f_j$. The overall complexity is, therefore, $O(\ell)$.

- To Insert (Delete) an integer x, we perform the following steps.
  (1) Identify the proper data structure $\mathcal{T}_j$ by resolving a partial Successor(x) query on $\mathcal{Y}$ in $O(\log\log n)$ and following the pointer retrieved at the identified leaf of $\mathcal{Y}$.
  (2) Identify the correct mini block by Search($x - f_j$) in $\mathcal{T}_j$ in $O(\log\log n)$.
  (3) Insert (Delete) $x - f_j$ in $\mathcal{T}_j$ by rebuilding the proper mini block in $\Theta(\ell)$.
  (4) Update $\mathcal{P}$ in $O(\ell)$.

During the third step, split or merge of a mini block can happen and it is handled in $O(\ell)$ worst-case time by the data structure $\mathcal{T}_j$; rebuilding of the mini blocks can happen as pointed out in the previous section and it is handled in $O(\ell)$ amortized time. If splitting (merging) of a block is necessary, the lower bound of the block is inserted (removed) from $\mathcal{Y}$ in $O(\log \log n)$ time. The overall complexity is, therefore, $O(\ell)$ amortized.

- The query Predecessor(x) is supported as follows. We identify the proper data structure $\mathcal{T}_j$ in $O(\log \log n)$ by answering a partial Predecessor(x) query on $\mathcal{Y}$ and following the pointer retrieved at the identified leaf of $\mathcal{Y}$. Then we identify the proper mini block by Search($x - f_j$) in $\mathcal{T}_j$ in $O(\log \log n)$ time. We finally return f_j + Predecessor($x - f_j$) by binary searching on the identified mini block. The overall complexity is $O(\log \log n)$ worst-case.

3.4.5 A further consideration

Observe that the data structure described in Section 3.4 allows us to support all operations in time $O(\log \log u)$ when *non*-polynomial universes are considered, i.e., when n and u are not necessarily related by the formula $u = n^\gamma$ for any $\gamma = \Theta(1)$. In this setting, the data structure of Lemma 2 will take $O(\text{polylog}\, u \times \log \log u)$ bits and operate in $O(\log \log u)$ time. In order to guarantee an overall redundancy of $o(n)$ bits, we let mini blocks be of size $\Theta((\log \log u)^2)$ and group $O(\log^2 u)$ consecutive mini blocks into a block. The high part of a mini block fits into one machine word, whereas we can insert/delete a low part in $O((\log \log u)^{2\epsilon})$ for Corollary 3 of [138], which is $O(\log \log u)$ as soon as $\epsilon < \frac{1}{2}$. Therefore, the following corollary matches the asymptotic time bounds of y-fast tries and van Emde Boas trees but in almost optimally compressed space.

Corollary 1 — There exists a data structure representing an ordered set $\mathcal{S}(n, u)$ of n integers drawn from a universe of size u that takes $\mathsf{EF}(\mathcal{S}(n, u)) + o(n)$ bits of space and supports: Access and Predecessor queries in $O(\log \log u)$ worst-case, Insert and Delete in $O(\log \log u)$ amortized time.

4 Clustered Inverted Indexes

As we have motivated in Section 2.3, representing the posting lists of document identifiers in compressed space while attaining to efficient query processing is *the* fundamental challenge of any Information Retrieval system. This is especially true nowadays, given the size of textual corpora and stringent query efficiency requirements. Achieving both objectives is generally hard, since they are conflicting in nature: a great deal of compression usually sacrifices fast retrieval; on the contrary, high speed algorithms benefit from an augmented index representation [25, 30]. A vast amount of literature describes different space/time trade-offs: see Section 2.2 for a description of the encoders proposed for inverted index compression.

Here, we notice that such encoders represent each inverted list *individually* and, thus, none of those techniques exploits the redundancy that may exist between *two or more* lists. Indeed the compression effectiveness of such encoders would be better if *clusters* of inverted lists were encoded together instead of separately since this offers the possibility of reducing the redundancy of the lists. Understanding how to reduce such redundancy to save index space and, at the same time, achieve very fast retrieval time is the issue addressed in this chapter.

As a matter of fact, inverted indexes naturally present some amount of redundancy. The reason is that the document identifiers (docIDs in the following) of "similar" documents, i.e., the ones sharing a lot of terms, will be stored in the inverted lists of the terms they share. More precisely, consider a document having docID d in which terms t_1 and t_2 occur. Then the inverted lists of *both* t_1 *and* t_2 will contain d. Figure 7 gives a graphical evidence of this fact. The picture shows how many times the 300 most frequent docIDs appear in an example cluster of 1000 inverted lists belonging to the Gov2 dataset (see Section 1.1 for its description and Table 2 for its statistics). Values are plotted along a Hilbert curve to better highlight the regions of docIDs having similar frequencies. The intensity of the color represents the degree of repetitiveness of a docID.

Now, generalizing to an arbitrary number of terms and documents, consider a set of terms $\{t_1, \ldots, t_k\}$ that occurs in m documents having identifiers $\{d_1, \ldots, d_m\}$. If the k terms *always co-occur* in the considered documents, then the set of integers $\{d_1, \ldots, d_m\}$ would be present in *each* of the posting lists of $\{t_1, \ldots, t_k\}$. In this special case, reducing the redundancy of the lists has an obvious solution: the redundant set $\{d_1, \ldots, d_m\}$ is encoded just once and each posting list stores a reference to it.

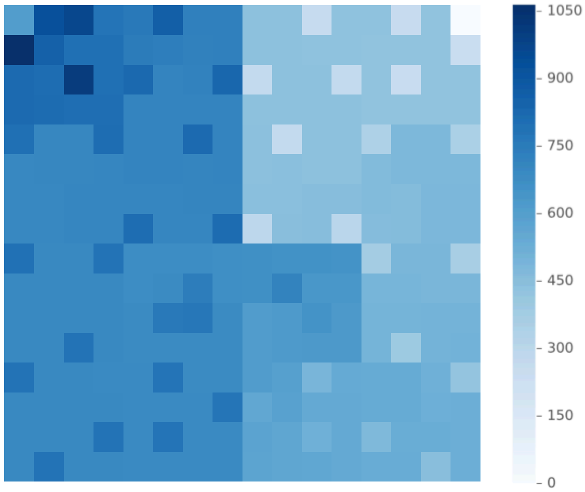


Figure 7: The 300 most frequent docIDs for a set of 1000 random inverted lists drawn from the dataset Gov2. We plot their frequencies along a Hilbert curve in order to highlight the regions of redundant postings. The color scale on the right indicates the intensity of frequency.

Unfortunately, it is *very unlikely* that the k terms appear in all documents. In general, it is more likely to have: very few terms co-occurring in all documents and several subsets of terms co-occurring in subsets of documents. Indeed notice that only few squares in Figure 7 are *much* darker than others, meaning that relatively few docIDs appear in all the lists of the cluster. Most docIDs have, instead, intermediate frequencies and constitute, therefore, the fundamental data source to be represented effectively, i.e., for which a space-efficient solution should be designed.

In this scenario, the problem of reducing the redundancy of the inverted lists is much more difficult, due to the following two different aspects.

- We do not know what and how many terms should be clustered together in order to maximize the number of co-occuring terms.
- It is not obvious how to compactly represent the redundant docIDs shared by the clustered inverted lists to save index space.

Here is, in short, our main idea. Suppose that we build a *meta-list* $\mathcal{R}$ by selecting a subset of docIDs belonging to the lists in a cluster C. Then the integers belonging to the intersection of list $\mathcal{S} \in C$ with $\mathcal{R}$ can be rewritten as the positions they occupy in $\mathcal{R}$ (see also Figure 8). The list $\mathcal{R}$ will be called the cluster *reference list.* If u denotes the last element of $\mathcal{S}$, each docID in the intersection is now drawn from a universe of size $|\mathcal{R}|$ instead of u, i.e., the number of bits necessary to represent each docID is now $\lceil \log |\mathcal{R}| \rceil$ instead of $\lceil \log u \rceil$. Then, if $\mathcal{R}$ is chosen such that the

list	intersection	PEF	CPEF	
1	68%	18.25 KB	15.27 KB	(−16%)
2	67%	18.92 KB	15.50 KB	(−18%)
3	68%	19.92 KB	16.92 KB	(−15%)
4	59%	19.71 KB	17.54 KB	(−11%)
5	66%	20.19 KB	17.16 KB	(−15%)

Table 5: Example of our method, CPEF, applied to five lists drawn from the Gov2 cluster of Figure 7. Last column shows the percentage of space reduction achieved with respect to partitioned Elias-Fano, PEF. This reduction depends on the percentage of intersection, shown in the first column, between the reference list of the cluster and the individual lists.

condition $|\mathcal{R}| \ll u$ holds true, the universe of the intersection is highly reduced and can be, therefore, encoded with much fewer bits. The representation of each list is finally compressed using a state-of-the-art encoder to save index space: in this work, we use PEF (see Section 2.2.7 and paper [120]) to support efficient query processing.

To better illustrate the potential of this technique, we build the reference list for the cluster of inverted lists illustrated in Figure 7, using the algorithm that we will discuss later in Section 4.2.2. This reference list contains 880,638 docIDs and its PEF representation takes 0.54 MB of space. In Table 5 we report the space occupancy for some inverted lists drawn from the Gov2 cluster of Figure 7, compressed using our technique (referred to as *clustered* PEF, or CPEF in the following) and compared against the space taken by PEF. We can see from this example that we are able to substantially improve the compression effectiveness over the PEF representation. Notice, in particular, how the gain strictly depends on the corresponding *percentage of intersection* with the reference list.

Therefore in the following, we are interested in the problem of determining the partition of the set of inverted lists into clusters and the choice of each cluster reference list such that the overall encoding cost is minimum. This is a difficult optimization problem. Indeed we will show that the problem of selecting docIDs to build the reference list such that the encoding cost of the cluster is minimized is NP-hard, already when considering a special (and simplified) case of the problem.

Our contributions. We discuss here the main contributions of this chapter.

We introduce a novel list representation designed to exploit the redundancy of inverted indexes by encoding a portion of a list with respect to a carefully built reference list. The representation is fully compressed using PEF, as to support very fast random access and search operations.

We describe an algorithm to cluster posting lists which is tailored for the introduced list representation. Moreover, we show how the optimization problem of selecting the proper reference meta-list for each cluster can be solved by an heuristic approach of selecting the postings by frequency within their cluster. This heuristic is fast and helps controlling the building time of the indexes.

We finally present a rich experimental analysis to show how to obtain interesting trade-offs by varying the size of the reference list. We then compare our technique with the state-of-the-art in Chapter 7.

4.1 Related work

We point the reader to Sections 2.2 and 2.3 for general background on integer compressors and inverted indexes. Here, we review the approaches most similar to our work in terms of index representation and clustering algorithms.

Index representations. Lam et al. [98] explore the possibility of exploiting the redundancy of inverted indexes by encoding two lists together. Their work proposes two encoding schemes: Mixed Union (MU) and Separated Union (SU). MU stores the union of two posting lists with two additional bits per posting, indicating whether the posting belong to the first posting list (bits 10), to the second (bits 01) or to both (bits 11). SU splits the representation of the union in three segments: the intersection between the two lists and the two residual parts. The terms that should be paired together are chosen by solving a *maximum-weight matching* problem on the graph $\mathcal{G} = (V, E)$ built as follows. Each node of V is a term; edge $(t_i, t_j) \in E$ is labeled with a value indicating how many bits would be saved with the pairing of terms t_i and t_j.

Another *multi-term indexing* strategy appeared in the work by Chaudhuri et al. [32]. Their solution builds, beside the traditional inverted index, a multi-term inverted index where each entry is composed by terms co-occurring frequently in query logs. As they observed that the distribution of terms is highly skewed in query logs, the aim of their proposal is to boost query processing speed at the price of the extra space needed to deal with another index.

Broder et al. [26] also noticed that many document collections, such as Web pages, e-mails and newsgroups can be highly repetitive (e.g., up to 45% of the Web pages are duplicates or nearly duplicates). They proposed a tree-based encoding for such collections, where each node of the tree contains one document and its children are documents that share its content. This encoding allows to index shared content just once. The experimental assessment discussed in the paper shows that significant space savings are possible while preserving the traditional ability of inverted indexes of handling free-text queries.

Recently, an approach based on generating a *context-free grammar* from the inverted index has been proposed by Zhang et al. [174]. The core idea is to identify common patterns, i.e., repeated sub-sequences of docIDs, and substitute them with symbols belonging to the generated context-free grammar. Although the reorganized posting lists and grammar can be suitable to different encoding schemes, the authors preferred Opt-PFOR [170]. The experimental analysis indicates that good space reductions are possible (as reported by the authors, around 8.8%) compared to state-of-the-art encoding strategies with competitive query processing performance. By exploiting the fact that the identified common patterns can be directly placed in the final result set, decoding speed can also be improved.

Clustering algorithms. The classical categorization of clustering algorithms divides them into two broad classes, dual in nature: *partitioning* and *agglomerative*. Partitioning algorithms work by partitioning the whole set of objects $\mathcal{X}$ according to an objective function f, until the desired number of clusters is reached or a stopping criterion is satisfied. On the other hand, agglomerative algorithms start by considering as many initial singleton clusters as the number of objects in $\mathcal{X}$. Then two or more clusters are merged together according to f. This process produces a hierarchy of clusters that may be subject to further refinement.

Among the first class, k-*means* [6] is the most popular and used clustering algorithm. In its simplest formulation [107]:

(1) k objects are drawn at random from $\mathcal{X}$ and considered as *centroids*;

(2) all other objects are assigned to the closest centroid, according to a distance function D;

(3) centroids are updated to be the mean of all objects in their clusters;

(4) repeat from step 2. until centroids "stop moving".

Hierarchical clustering algorithms include, instead: Single-Link, Average-Link and Complete-Link. We point the reader to the survey by Xu and Wunsch [169] for an exhaustive overview on the subject. Although such algorithms are supposed to produce superior clusters in terms of cluster quality [150], their quadratic complexity often prevents from practical use. This has been the main reason of the success of k-means which is elegant, simple and fast [150, 123, 12]: its complexity is $O(kd|\mathcal{X}|i)$, where d is the dimensionality of the clustered objects and i is the number of needed iterations to converge. Since usually k and d are much less than $|\mathcal{X}|$, its complexity is very appealing in practice.

Many variants of the regular k-means algorithm have been proposed in the literature [150, 123, 12]. Steinbach, Karypis, and Kumar [150] introduced a *bisecting* variant of regular k-means that computes two initial clusters and recurse on them until k clusters are formed. They proved bisecting k-means performs even better than the classical one, because it produces relatively uniformly sized clusters. We will use this approach in our own clustering algorithm. Pelleg and Moore [123] improved on the classical formulation that needs the user to provide the number of clusters. Instead of a single value for k, their algorithm takes as input a possible range of values. In essence, the algorithm starts with k equal to the lower bound of the provided range and keep adding centroids until the upper bound is attained. Adding a new centroid implies splitting an existing one in two: the decision on which centroid to split is based on the *bayesian information criterion*. During the whole process, the centroid set that achieves the highest score is stored and finally output.

As the question regarding which seeds to choose for the initialization step of k-means is posed, Arthur and Vassilvitskii [12] proposed to select k seeds at random, one at a time, from a *non-uniform* distribution. Specifically, the first centroid c is picked uniformly at random, then the probability that the point x becomes the next centroid is $dist(x, c)^2 / \sum_{y \in \mathcal{S} \setminus \{x\}} dist(y, c)^2$. The process is repeated until k seeds are chosen. The key drawback of such approach lies in its inner sequential nature, since the k seeds must be chosen sequentially thus requiring k passes over the data. This issue was tackled by Bahmani et al. [13], providing an efficient parallel implementation of the above procedure. As clear from the above formula, the greater the distance of an object to the just chosen centroid, the higher the probability of selecting that object. The intuition, confirmed by their experimental analysis, is that a "good" initial choice of centroids will place them far apart from each other. We will use this initialization procedure in our own algorithm.

4.2 Representing a set of inverted lists

In this section we introduce our novel index representation for a set of posting lists. The key idea is to exploit the implicit redundancy of the inverted lists to reduce space, via a *universe reduction* technique.

Encoding. Let $\mathcal{L}$ indicate the set of all inverted lists in the index. Let $\mathcal{R}$ be a list not necessarily belonging to $\mathcal{L}$ and $C \subseteq \mathcal{L}$ a subset of lists (a cluster). Then the integers belonging to the intersection of list $\mathcal{S} \in C$ with $\mathcal{R}$ can be rewritten as the positions they occupy in $\mathcal{R}$. If u denotes the universe of $\mathcal{S}$, the intersection is now encoded with a universe of size $m = |\mathcal{R}|$ instead of u. If $\mathcal{R}$ is chosen such that the condition $m \ll u$ holds, we can reduce the universe of the intersection which can

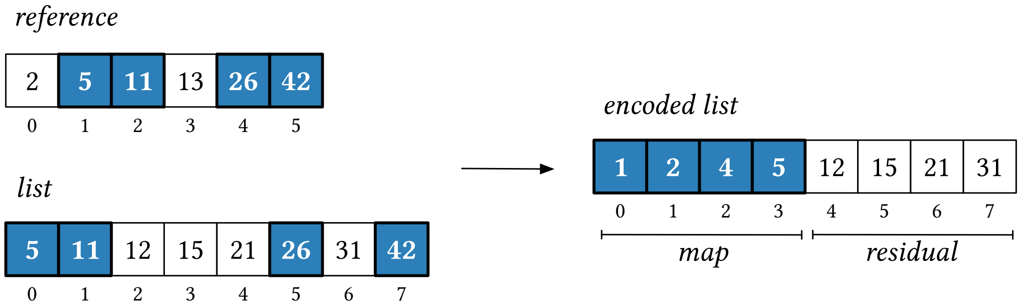


Figure 8: List representation example: shaded boxes mark the integers falling in the intersection between list and reference (*map*); all the others form the residual part (*residual*).

be, therefore, encoded with much fewer bits. The list $\mathcal{R}$ will be called the *reference* list for the cluster C. Figure 8 shows an encoding example.

This strategy introduces a partition of each list into two segments: a *map* part made up of all rewritten integers falling in the intersection with the reference, and a *residual* part consisting in all other integers. Our list representation is the juxtaposition of these two segments.

Specifically, suppose $\mathcal{S}$ and $\mathcal{R}$ share k integers. Using the space bound of plain Elias-Fano (equation 2), the space taken by $\mathcal{S}$ in this new representation is

$$\mathsf{EF}(\mathcal{S}(n,m)) \le k\left\lceil \log \frac{m}{k} \right\rceil + (n-k)\left\lceil \log \frac{u}{n-k} \right\rceil + 2n \text{ bits.} \tag{4}$$

Not surprisingly, the greater the number of integers shared by $\mathcal{S}$ and $\mathcal{R}$, the better the space of our encoding. Notice that, though the average gap $\lceil \log(u/(n-k)) \rceil$ will become larger, the overall residual cost will decrease too, since $(n-k)\lceil \log(u/(n-k)) \rceil$ is monotonically decreasing for reasonably large values of u, as it is likely to be in practice. For example, substituting $m = u/8$ and $k = n/2$ in Equation 4, the resulting space is $n\lceil \log(u/n) \rceil + 2n - n/2$ bits, thus saving 0.5 bits per integer with respect to plain Elias-Fano.

Note that the described solution is *general*, in the sense that it allows for any encoding strategy to be used for the reference, map and residual lists. As already stated and motivated, in our implementation we adopt PEF, to encode both map and residual segments of each list, as well as references. Furthermore, observe that nothing prevents from recursively applying the very same encoding strategy to the residual segment of each list. However, implementing such a recursive encoder is much more complicated and may prevent from practicality.

Searching. The cursor operations, NextGEQ and Next, can be efficiently implemented by means of the same operations performed by three cursors, each operating on map, residual and reference list separately. To answer a NextGEQ query

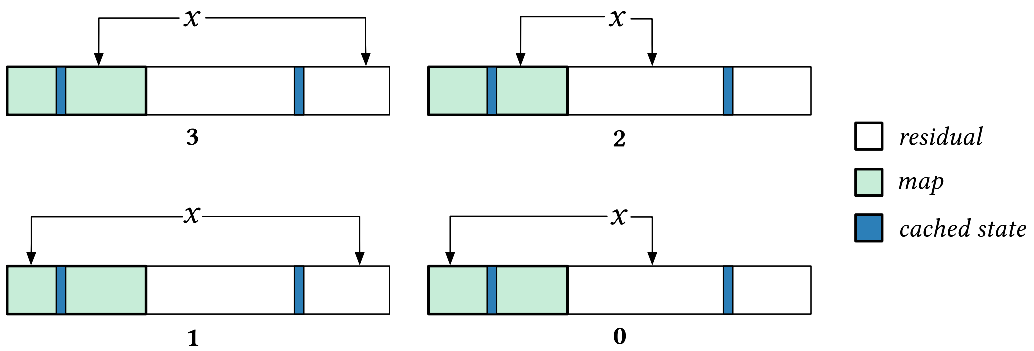


Figure 9: NextGEQ algorithm operating on the clustered list representation. Depending on the relation between the searched value x and the cached state of our cursor, the algorithm may perform **3**, **2**, **1** or even **0** NextGEQs. Below each case, we report the number of performed operations.

on our list organization, a naïve implementation will first perform a NextGEQ on the residual followed by another one on the map to finally return the minimum of the two. Given an integer x, notice that the map may only contain its position p_x within the reference. Therefore a NextGEQ(x) operation on the map actually involves a first r_x = NextGEQ(x) on the reference followed by NextGEQ(p_x) on the map. This second NextGEQ(p_x) is necessary to verify if the searched element p_x is actually contained in the map. If it is, then r_x is the value to return, otherwise a final random-access operation must be performed on the reference to retrieve $\mathcal{R}$[NextGEQ(p_x)].

The outlined algorithm will *always* execute three NextGEQs. We argue that this complexity can be alleviated if we cache a state in our cursor implementation. In particular, we save the last accessed values in map and residual: call them y_m and y_r respectively. Depending on the relationship between the given lower bound x, y_m and y_r, it is possible to write the algorithm such that we do not always need three NextGEQs but just two, one or even none of them. Figure 9 offers a pictorial representation of such relations. In the following description we show in brackets the number of NextGEQs performed.

If $x > y_m$ then a NextGEQ on the map is performed (2). At this point we also check if $x > y_r$. If so, a NextGEQ on the residual is performed and we return the minimum of the two values (3). If $x \leq y_m$ instead, we can directly check if $x > y_r$. If so a NextGEQ on residual is performed (1). Eventually, if both previous conditions are not satisfied, i.e., $x \leq y_m$ and $x \leq y_r$, then no NextGEQ are needed and we can just return the minimum between y_m and y_r (0). This completes the description on the NextGEQ algorithm operating on the new list representation. The Next procedure can be implemented similarly.

At this point it is clear that the reference length is a crucial parameter for our encoder. As already mentioned, the larger the number of integers shared by the reference and a list, the better the space usage but, conversely, the greater the number of integers that need to be un-mapped during a search operation (two NextGEQs and one Access in the worst case), affecting the retrieval efficiency. We will stress this evident trade-off in Section 4.3 with an extensive experimental analysis.

Finally, observe that frequency lists need to reflect the partition into map and residual segments too. Therefore, first we store all frequencies for the map, then all frequencies for the residual. When resolving an $x' = \text{NextGEQ}(x)$ query we need to know the frequency of the term in document x', we access the corresponding frequency list at position $k + j_r$ if x' falls into the residual or at position j_m if it falls into the map, where j_m and j_r denote the position of cursors operating on map and residual respectively.

The optimization problem. We argue that two intuitive, yet fundamental, problems naturally arise from the given representation.

- Which lists to group together in set C under the same reference. Since the greater the intersection of a list with its reference the better our encoding, we would like to collect together lists sharing a lot of integers so that a given reference is able to "cover" a greater portion of the lists in C. This problem reduces to the problem of clustering the index posting lists to maximize the number of integers shared by the lists in a cluster.

- Choosing which integers to put into the reference list reduces to an optimization problem. Indeed, our goal is that of selecting the reference list for a given set C such that the space taken by the encoding of C is minimized.

These two considerations allow us to formally state the problem we are considering. Let $\text{CPEF}(C_i, \mathcal{R}_i)$ be the cost, in bits, of our encoding applied to the lists in set C_i, using reference $\mathcal{R}_i$. We want to solve the following optimization problem.

Problem 1 — Determine the partition of $\mathcal{L}$ in c clusters, i.e., $\{C_1, \ldots, C_c\}$ subject to $\mathcal{L} = \cup_{i=1}^{c} C_i$, $C_i \cap C_j = \emptyset \ \forall i \neq j$ and each C_i non-empty, and the choice of $\mathcal{R}_i$, $|\mathcal{R}_i| > 0$, for each cluster C_i, such that

$$\sum_{i=1}^{c} \text{CPEF}(C_i, \mathcal{R}_i) \quad (5)$$

is minimum.

This problem is difficult. In fact, suppose to have already computed the partition $\{C_1, \ldots, C_c\}$. Now consider a cluster C_i and the problem of choosing a reference of size k such that the encoding cost of C_i is minimum when each list is encoded with *plain* Elias-Fano. This is a *k-reference selection problem* (RSP_k). Even in this simplified setting, the following theorem holds.

Theorem 4 — RSP_k is NP-hard.

Proof. We use the hardness of the *k-Clique Problem* (CP_k) [69], in which the input is an undirected graph $\mathcal{G} = (V, E)$ and the output is a clique of k nodes (if one exists). In particular, the variant of CP_k for which we are asked to find, for a given $0 < \epsilon < 1$, a clique of size $\epsilon|V|$ is NP-complete for any choice of ϵ [69]. In our reduction we are interested in the variant where k equals $n/4$, i.e., $\epsilon = 1/4$.

Consider an instance $\mathcal{G} = (V, E)$ of CP_k. V is the set of n vertices that, without loss of generality, we can indicate with $1, \ldots, n$. E is the set of m undirected edges. The instance of RSP_k is obtained from $\mathcal{G}$ by letting vertices in V be the set of postings and by including in C the m posting lists, one for each edge $(u, v) \in E$, formed by the two postings u and v.

Call R the optimal solution for RSP_k on C. The encoding of each posting list with respect to R has only the following three possible costs

(1) $2\left\lceil \log \frac{k}{2} \right\rceil + 4$ bits when both postings are in R;

(2) $2\left\lceil \log \frac{n}{2} \right\rceil + 4$ bits when both postings are not in R;

(3) $\lceil \log k \rceil + \lceil \log n \rceil + 4$ bits when one posting is in R and the other is not.

Since k is $n/4$, the costs in (2) and (3) coincide and (1) is always the smallest encoding cost. Thus, $\mathcal{R}$ must be such that the number of posting lists whose encoding cost is 1. is maximized. This is equivalent of saying that $\mathcal{R}$ is such that it maximizes the number of edges that have both vertices in R, by the one-to-one correspondence between edges of $\mathcal{G}$ and posting lists in C. Thus, if $\mathcal{G}$ has a k-clique, $\mathcal{R}$ is formed by the vertices of this k-clique. ■

As it is not obvious which is the strategy that selects the best clusters and references so that the encoding of the clusters is minimum, it is convenient to consider the following three-step modelling, in which the first two steps are solved by an heuristic approach.

(1) Clustering $\mathcal{L}$ into $\{C_1, \ldots, C_c\}$, where c is unknown, to group together lists sharing as many docIDs as possible.

(2) For each cluster C_i select the reference $\mathcal{R}_i$, $|\mathcal{R}_i| > 0$, such that $\mathsf{CPEF}(C_i, \mathcal{R}_i)$ is minimum. We will illustrate an efficient, heuristic, algorithm for this problem.

(3) Encode each $C_i \in \{C_1, \ldots, C_c\}$ with PEF.

In what follows, we discuss each of these three steps in details. Although the three steps are strictly correlated and, therefore, cannot be completely separated from each other, this modelling yields an efficient algorithm for an approximate solution to the original problem.

4.2.1 Clustering

In this subsection we describe the algorithm that we use to cluster the inverted lists. We use a modified version of k-means [6]. We adopt a k-means-based approach because it is the only able to scale to the dimensions we are dealing with (see Table 2 for the basic statistics of the tested collections). More precisely, although other clustering approaches, e.g., Single-Link or Average-Link [169] may produce superior clusters in terms of cluster quality [150], their *quadratic* complexity prevents from practical use. As inverted lists may contain from tens of thousands to millions of postings, either an approximate distance metric is used or a lower-complexity algorithm is used. This is the main reason for the success of k-means and, indeed, the reason for its choice as our clustering algorithm.

Overview. We start with a high-level overview of the algorithm. In order to avoid to supply the apriori number of k clusters to the algorithm, we use the *bisecting* approach by Steinbach, Karypis, and Kumar [150]: we execute an instance of 2-means and recurse on the two children clusters. More specifically, we maintain a deque Q of clusters to be split, initially containing the fictitious cluster formed by the entire dataset. At each step of recursion a cluster is picked from the top of Q and an instance of 2-means is executed on it. Each of the two children is then inserted to the back of Q if it needs refinement. Whenever a cluster does not need any further splitting, it is inserted in a list L of "final" clusters. When Q is empty, the algorithm terminates. The number of created clusters is the length of list L. This skeleton describes a *divisive* and *hierarchical* clustering algorithm.

Details. Instead of recurring on the largest of the two children (as done in [150]), we adopt an ad-hoc criterion that meets the requirement of the encoding phase that will follow the clustering step. Since the cost of our encoding comprises the cost for the reference lists as well, we intuitively would like to create as few clusters as possible. The problem is that, in general, the bigger the cluster, the longer the reference. However, encoding lists with respect to a very long reference will produce a negligible universe reduction, thus dwarfing the quality of our encoder. Therefore, we decide that a cluster needs further splitting if its current reference is greater than a *user-defined threshold*. This threshold defines the maximum length of the reference that the algorithm builds for each cluster. The current reference size of a cluster is estimated using a fast, heuristic approach that we will describe in Section 4.2.2. The number of documents u in the collection clearly represents an upper bound on the possible values of this threshold. The experiments regarding how the cluster quality varies for different values of the threshold are presented and discussed in Section 4.3. In particular, we will determine the best choice of maximum reference size in terms of encoding cost, i.e., the number of bits per posting.

The other meaningful point to describe is the choice of the two seeds. We use the randomized approach described by Arthur and Vassilvitskii [12]: the first seed c is drawn uniformly at random from the set $\mathcal{X}$ of approximately equally-sized posting lists, then another list x is chosen with probability $dist(x, c)^2 / \sum_{y \in \mathcal{X} \setminus \{x\}} dist(y, c)^2$. We follow this approach since we want the two clusters to be well far-apart from each other. Notice that a single pass over the cluster lists suffices for this task.

The last detail we illustrate is how the distance of a list from a cluster centroid is computed. It seems natural to use a similarity measure that accounts for how many integers of a sequence $\mathcal{S}$ are shared with centroid $\mathcal{C}$. Using set notation, a modified Jaccard coefficient $sim = |\mathcal{S} \cap \mathcal{C}| / |\mathcal{S}| \in [0, 1]$ provides us this quantity. We argue that this similarity measure suffers from several problems.

First of all, *not all postings* should be considered for intersection but only a subset of $\mathcal{S}$. This is a direct consequence of the fact that we are using PEF to encode map and residual segments in our list representation. Recall that the j-th chunk of b docIDs can be encoded in three different ways according to the relation between b and its universe u_j. In particular whenever a chunk is encoded with its characteristic bit vector or with 0 bits, it is never advantageous to represent some of its elements with respect to the reference because the chunk will be broken in pieces, each encoded with a larger number of bits. This implies that we have to exclude from S all docIDs except the ones belonging to chunks encoded with Elias-Fano.

The other problem is that $sim = |\mathcal{S} \cap \mathcal{C}| / |\mathcal{S}|$ completely ignores the distribution of docIDs in the posting lists. In fact, $|\mathcal{S} \cap \mathcal{C}|$ could be large just because of docIDs: that are shared between $\mathcal{S}$ and $\mathcal{C}$ but *not with all other lists* in the cluster; or occurring very frequently in the whole collection and, therefore, in almost each list. These issues are tackled by maintaining two counts for each posting: a *local count* keeping track of how many times the posting occurs in the lists of the cluster and a *global count* that weights each posting for its own frequency in the whole collection. Notice that the combination of local and global measures is the same solution adopted by the so-called *vector-space model* [109, 31]. If we provide analogous definitions of term-frequency (tf) and inverse-document-frequency (idf) in the inverse domain, which is made up of all posting lists of the document collection, then we can treat each posting list as a vector of real numbers. More precisely, we define as *document-frequency* $\mathrm{df}(x, \mathcal{S})$ the frequency of docID x in inverted list $\mathcal{S}$. This measure corresponds to the tf count in the documents' domain. This count depends on how the posting lists are built. In our case each df is just 1, but it could be greater depending on the occurrences of a docID in a posting list. A real-life example of this scenario is the Twitter inverted index: a docID is appended multiple times to the posting list of a term if that term occurs multiple times in the indexed tweet [29].

The corresponding of the idf count in the inverse domain is the *inverse-term-frequency* (itf), which accounts for how many times x is appearing in whole collection: $\mathrm{itf}(x) = \log(|\mathcal{T}|/|\{\mathcal{S} \in \mathcal{L} : x \in \mathcal{S}\}|)$, where $\mathcal{T}$ is the collection lexicon.

Now each posting list $\mathcal{S}$ is seen as a vector $S[1, u]$, with $S[i] = \mathrm{itf}(i)$ if integer i belongs to $\mathcal{S}$ or 0 otherwise. In what follows, we implicitly refer to a list $\mathcal{S}$ by means of its itf vector S. As distance function we use $dist = 1 - \cos(S, C)$, where $\cos(S, C) = \sum_{i=1}^{u}(S[i] \times C[i])/\sqrt{||S||^2 \times ||C||^2}$ is the *cosine similarity* between the centroid C and the vector S. Whenever a list is added to its closest cluster, we immediately update the centroid to be the sum of the newly added list and the centroid itself. In this way, the centroid C of each cluster takes into account the number of times the posting i occurs within the cluster (local count), that is exactly $C[i]/\mathrm{itf}(i)$ times.

4.2.2 Reference selection

Consider the cluster C_i and the optimization problem of synthesizing the reference $\mathcal{R}_i$ such that $\mathsf{CPEF}(C_i, \mathcal{R}_i)$ is minimum. We have shown in Theorem 4 that the simplified RSP_k is NP-hard. If m is the number of integers in C_i, i.e., the sum of the lengths of its posting lists, and n the number of its distinct integers, then an optimal solution can be computed in $\Theta(m^n)$ time and $O(n)$ space. In fact, for each sorted subset $\mathcal{R}_i$ of the distinct integers of C_i we should keep track of $\arg\min_{\mathcal{R}_i} \mathsf{CPEF}(C_i, \mathcal{R}_i)$. Since there are $\sum_{k=1}^{n}\binom{n}{k} = 2^n - 1$ possible ways of choosing R_i and encoding takes linear time in the number of postings, the time complexity follows. This is clearly unfeasible.

As a general overview, our approach selects postings that will end up in the reference from a set that we call the set of *candidate postings*. Let c indicate the cardinality of this set. As already noted in Section 4.2.1, only postings belonging to blocks encoded with Elias-Fano should be considered as possible candidates, otherwise there is the risk of "breaking" a block encoded with much fewer bits.

Frequency-based selection. An effective method to select the postings for the reference is based on their frequencies within the cluster, i.e., how many times they occur in the posting lists belonging to the cluster. We just need to sort the set of candidate postings according to their frequencies and add to the reference the top-k most frequent postings, where k is the reference wanted dimension. The intuition behind this heuristic is that if the reference is composed by postings occurring in most lists, then it should have a good "coverage" property. On the other hand, including postings occurring only a few times, would be beneficial for few lists too while, at the same time, be detrimental for *all* other lists: the reference will grow, therefore expanding the universe of the representation of *each* map.

As the experimental Section 4.3 will show next, this heuristic performs well in practice especially for larger values of reference size. Its most important advantage is its speed: a single pass over the lists suffices to build frequency counts that are used as the sort-criterion. Time and space complexities are, respectively, $O(m)$ and $O(c)$.

4.2.3 Encoding

Each cluster C_i gets encoded with respect to its reference list $\mathcal{R}_i$ according to the representation we have detailed at the beginning of Section 4.2.

In particular, only postings belonging to Elias-Fano encoded blocks are considered for intersection with the reference, all others form the residual part. Recall from Section 2.2.7 that a block in PEF is encoded with one among three different representations, according to the relation between its universe and size: 1. Elias-Fano; 2. the block's characteristic bit vector; 3. not encoded at all, thus taking 0 bits. Since we can calculate the Elias-Fano partitions of a sequence in linear time using the dynamic programming algorithm by Ottaviano and Venturini [120] and determine in $O(1)$ which is the type of encoder used for each block, map and residual segments are computed in time proportional to the length of the sequence.

Considering all clusters, the overall complexity of the encoding step is, therefore, linear in the number of postings in the inverted index.

4.2.4 Index layout

We now describe our index organization, starting with a high-level picture: our index is made up of three large bit vectors, that we call *document*, *frequency* and *reference* bit streams. They represent the index posting, frequency and reference lists respectively.

Each bit stream results from the concatenation of the bit vectors that individually represent posting, frequency and reference lists. The three bit streams are *aligned*: the i-th frequency list is associated to the i-th document list. Document lists are stored according to cluster-identifier order, i.e., the first lists in the *docu-*

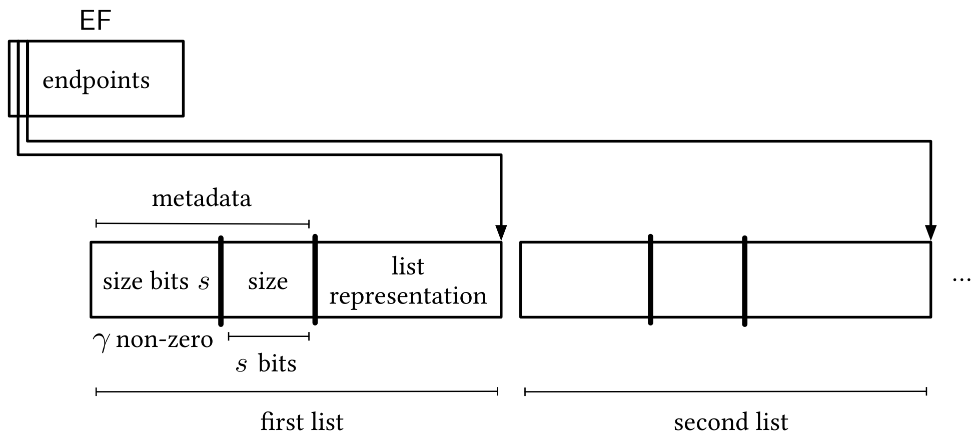
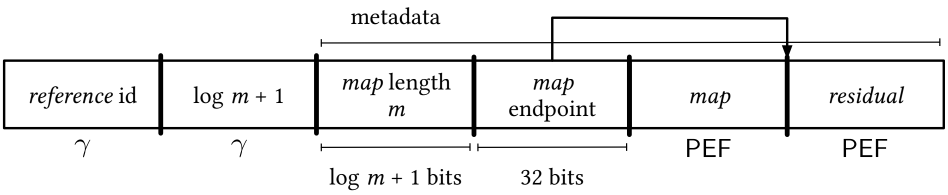


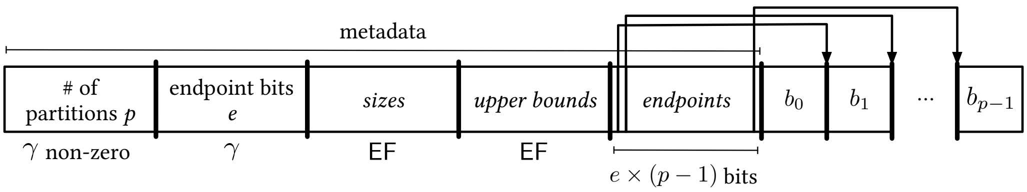
Figure 10: Bit stream and endpoints.

ments stream are the ones belonging to the first cluster, then the ones belonging to the second cluster follow and so on. Except for frequency lists, each list representation in the stream is enriched with a metadata information storing the size of the list. First we write in γ code the number of bits necessary to represent the size, then the size value is written uncompressed using this number of bits. Notice that since lists are non-empty we can subtract 1 to the γ encoding of the size (γ non-zero). Since we use PEF to encode both map and residual segments of our posting list organization, we store list sizes in order to distinguish the different metadata sections of the partitioned Elias-Fano representation that we discuss later. Finally, in order to be able to access each sequence, we store the positions of the bit stream at which sequences end in an Elias-Fano encoded list of endpoints. In this way, as random access operations are supported efficiently with Elias-Fano, we can access each sequence in $O(1)$ within compressed space. Figure 10 shows how each bit stream is organized. Now we discuss how a single list is represented.

CPEF layout. Our posting list organization contains a metadata header section before the actual representation of its map and residual segments. This metadata section is structured as follows. First of all, we need to store the identifier of the reference list with respect to which the list is encoded. Such identifier ranges from 0 to the number of possible clusters minus 1 and is represented in γ. Then we need to record the length of the map segment, say m. We first store the quantity $\log m + 1$ in γ, then m using $\log m + 1$ bits. We finally need to know where the first stored segment (the map) ends to be able to distinguish between the two segments. The last metadata information is, therefore, the number of bits s of the map segment written uncompressed in 32 bits. Map and residual segments are stored one after the other, both encoded as a partitioned Elias-Fano sequence that we describe next. Figure 11a shows our *clustered sequence* organization. In this case, the size of the



(a) Clustered Elias-Fano sequence



(b) Partitioned Elias-Fano sequence

Figure 11: Clustered and partitioned Elias-Fano sequence organizations. Below each block we report how it is encoded: γ, fixed-width, Elias-Fano (EF), partitioned Elias-Fano (PEF). In picture **(a)**, m represents the length of the map segment; in picture **(b)**, p indicates the number of partitions whereas e the number of bits needed to encode a pointer to a block.

whole list in the stream is necessary to derive the size of the two segments, which are needed to correctly search their PEF representation.

As to be able to encode the frequency lists using Elias-Fano, each of them is transformed into a monotonically increasing integer sequence by computing its prefix sums. Since we only need to randomly access such lists, we can subtract i to the i-th frequency value and save space. Notice that we do not need to store the size of each frequency list in the bit stream, given that frequency and document streams are aligned: the size of the i-th frequency list will be the size of the i-th list of the stream.

PEF layout. We conclude this subsection by explaining the PEF layout. Refer to Figure 11b. As usual, a metadata section precedes the representation of the blocks $\{b_0, \ldots, b_{p-1}\}$. We first write the number of partitions using γ non-zero. Then we write the number of bits for each block endpoint: if m is the length of the bitvector resulting from the concatenation of all blocks, then we need $e = \log m + 1$ bits. Two Elias-Fano encoded blocks follow, representing blocks' sizes and upper bounds respectively. The Elias-Fano representation of a sequence is self-delimiting because we can compute exactly the number of bits of the representation if we only know the length and universe of the sequence [161]. In this case, both *sizes* and *upper bounds* blocks have length equal to the number of partitions; the universe of *sizes* is the length of the sequence whereas the universe of *upper bounds* is just the number of documents in the collection. The endpoints are just stored in fixed-width fashion. The concatenation of all encoded blocks terminates the list representation.

4.3 Experiments

Datasets. We performed our experiments on the datasets described in Section 1.1, whose statistics are summarized in Table 2 (page 7).

Experimental setting and methodology. All tests have been performed on a machine with 16 Intel Xeon E5-2630 v3 cores (32 threads) clocked at 2.4 Ghz, with 64 GB RAM, running Linux 3.13.0, 64 bits. Hardware caches have the following sizes: 32 KB (L1), 256 KB (L2) and 20 MB (L3). Levels L1 and L2 are private to each core, while L3 is shared among all the 8 cores on one socket. In addition, we have repeated all the experiments on a second machine equipped with 4 Intel i7-4790K cores (8 threads) clocked at 4 Ghz, with 32 GB RAM, running Linux 4.2.0, 64 bits to confirm the results. This second machine has the same cache sizes of the other, except for the last shared level (L3) which is 8 MB.

All the code is implemented in standard C++11, based on the popular ds2i project[1] and compiled with gcc 5.3.0 with the highest optimization settings. We have preferred template specialization over inheritance to avoid virtual method call overhead, which can be disruptive for very fine-grained operations, such as the ones we consider in the following. Except for the instructions to count the number of bits set in a word and to find the position of the least significant bit, no special processor feature was used. In particular, we did not add any SIMD instruction to our code.

The indexes were saved to disk after construction, and memory-mapped to perform the queries. To test the speed of the indexes, we use a random sampling of 1000 queries, respectively from TREC 2005 and 2006 Efficiency Track topics. In order to smooth the effect of fluctuations during measurements, we repeat each experiment three times and consider the mean. All query algorithms were run on a single core and query times are reported in milliseconds.

Source code. `https://github.com/jermp/clustered_elias_fano_indexes`

[1] `https://github.com/ot/ds2i`

	T	u	$\frac{u}{2}$	$\frac{u}{4}$	$\frac{u}{8}$	$\frac{u}{16}$	$\frac{u}{32}$
Gov2	Number of clusters	2	5	25	70	150	264
	Minutes	5	10	23	32	34	37
	Bits per posting	2.70	2.67	2.67	2.65	2.71	2.79
ClueWeb09	Number of clusters	2	31	88	174	302	461
	Minutes	24	105	127	138	153	170
	Bits per posting	4.60	4.62	4.54	4.50	4.75	4.87

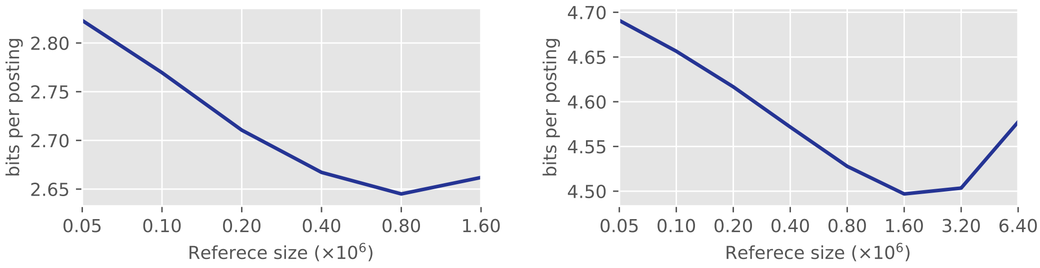
Table 6: Number of clusters, clustering time in minutes and number of bits per posting by varying the reference size threshold T.

4.3.1 Clustering

For the clustering step, instead of considering only the posting lists containing more than a given number of postings (as done, for example, in [53]), we sort the lists by length in descending order and discard the k smallest lists such that the sum of their postings is 10% of the postings of the original collections. The posting lists excluded from clustering are encoded with PEF.

This simple pruning strategy allows us to significantly reduce the number of processed terms from millions to tens of thousands while concentrating our effort on most of the postings, because the distribution of terms occurrences is *highly skewed*: relatively few lists are very long while the majority being very short. In fact, for the Gov2 dataset we retain only 17,959 inverted lists out of more than 35 millions; for the ClueWeb09 datasets we retain, instead, 30,972 inverted lists out of more than 92 millions. As already said, for both datasets, these lists account for the 90% of the postings in the collections.

Table 6 shows the number of created clusters, clustering time in minutes and the number of bits per posting by varying the reference size threshold. We recall from Section 4.2.1 that this threshold is the free parameter of our clustering algorithm and represents the maximum length of the reference that is built for each cluster. This threshold is expressed as the ratio between the universe collection u (number of documents) and a constant ranging from 1 to 32. As we can see, the smaller the reference a cluster can synthesize, the greater the number of iterations performed by the algorithm and, consequently, the number of clusters and clustering time. In particular, clustering time for ClueWeb09 is approximately four times the one of Gov2 since ClueWeb09 has roughly two times the number of posting lists and universe of Gov2. In the next subsection we discuss how the number of bits per posting has been derived.



(a) Gov2 **(b)** ClueWeb09

Figure 12: Bits per posting of Gov2 and ClueWeb09 by varying the reference size.

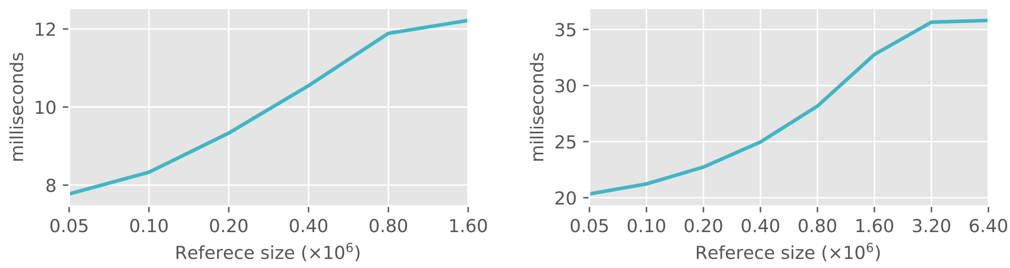
4.3.2 Space/time trade-offs

In this subsection we present a detailed analysis on *how and why* varying the reference size of the encoder can yield interesting space/time trade-offs. During the analysis we also experimentally determine the values of reference size that give the best trade-offs.

Varying the reference size. As already pointed out in Section 4.2, the reference selection step for each cluster deeply affects the quality of the representation in terms of both space usage and speed, as we are going to motivate next. Intuitively, dealing with small references implies that the fraction of remapped postings is small too on average and this is less beneficial for index space. Conversely, as references grow in dimension, space is gradually reduced but accessing the representation of the references becomes the major bottleneck at query time. The introduced trade-off is evident: smaller references yield faster but bigger indexes while longer ones slower but smaller indexes.

To test this impact, we build several indexes for the two test collections, varying the reference size from 50,000 to 1,600,000 for Gov2 and to 6,400,000 for ClueWeb09, doubling its size each time. These sizes represent the *maximum* reference sizes that the algorithm is permitted to build.

Now, before concentrating the analysis on the mentioned space/time trade-off, we choose the clustering that yields the smallest encoding cost in terms of bits per posting. To help our decision we consider Table 6. The reported number of bits per posting has been obtained by encoding each cluster with the largest reference size, i.e., 1,600,000 for Gov2 and 6,400,000 for ClueWeb09. For both datasets, a value of threshold equal to $u/8$ yields the most compact indexes. Therefore, in what follows, all experiments have been done using such value for the clustering algorithm.



(a) Gov2 **(b)** ClueWeb09

Figure 13: Timings for AND queries by varying the reference size on Gov2 and ClueWeb09, using the query set TREC 06.

Reference size ($\times 10^6$)		0.1	0.2	0.4	0.8	1.6	3.2
Gov2	space	−1.71%	−3.82%	−5.53%	−6.45%	−6.52%	—
	time	+7.15%	+20.07%	+35.69%	+52.92%	+57.18%	—
ClueWeb09	space	−0.71%	−1.58%	−2.48%	−3.40%	−4.21%	−4.56%
	time	+4.38%	+11.79%	+22.76%	+38.50%	+61.12%	+75.28%

Table 7: Space and time percentages for CPEF with respect to the minimum value of reference size 50,000.

Figure 12 illustrates the number of bits per posting by varying the reference size. These plots have to be read together with the ones in Figure 13, which show mean query times for AND queries, performed on ClueWeb09 collection using the TREC 06 query log. The two plots together confirm the trade-off that we explained before: as references grow in size, the number of bits per posting decreases but the mean query time increases as well.

Choosing the trade-off points. We now determine three values of reference size as representatives of the different space/time trade-offs we can obtain, in order to concentrate our analysis on these selected points. To help our decision, consider Table 7. The table reports the percentages of space and time with respect to the values in correspondence of reference size 50,000.

For both datasets, a value of reference size equal to 100,000 looses a negligible factor in query processing speed but space reductions are very poor too. Therefore, we choose this value of reference size as representative of the faster query time with respect to the other trade-off points we are going to choose. On the other hand, larger values of reference size optimize space sacrificing query processing speed. For Gov2 values of 800,000 and 1,600,000 offer practically the same space reduction but the former achieves better speed. The same holds for ClueWeb09 as

	Gov2		ClueWeb09	
PEF	2.94		4.80	
CPEF@MIN	2.77	(−6%)	4.66	(−3%)
CPEF@MID	2.71	(−8%)	4.57	(−5%)
CPEF@MAX	2.65	(−10%)	4.50	(−6%)

(a) clustered

	Gov2		ClueWeb09	
PEF	4.10		5.85	
CPEF@MIN	3.94	(−4%)	5.70	(−3%)
CPEF@MID	3.88	(−6%)	5.61	(−4%)
CPEF@MAX	3.81	(−7%)	5.54	(−5%)

(b) total

Table 8: Bits per document in the selected trade-off points on the clustered collections in **(a)** and on the whole collections in **(b)**.

	TREC 05		TREC 06	
PEF	3.7		6.1	
CPEF@MIN	5.3	(+43%)	8.3	(+36%)
CPEF@MID	5.9	(+59%)	9.3	(+52%)
CPEF@MAX	7.8	(+111%)	11.9	(+95%)

(a) Gov2

	TREC 05		TREC 06	
PEF	14.6		17.7	
CPEF@MIN	17.7	(+21%)	21.2	(+20%)
CPEF@MID	20.6	(+41%)	25.0	(+41%)
CPEF@MAX	29.1	(+99%)	35.6	(+101%)

(b) ClueWeb09

Table 9: Timings in milliseconds for AND queries for Gov2 and ClueWeb09, using the query logs TREC 05 and TREC 06.

well but for values 3,200,000 and 6,400,000. Values that fall in between these two extreme points trade-off between space and time.

We now select our trade-off points such that two of them optimize either space or time, the third one tries to grab the best from both. From the above discussion, we choose the following points for ClueWeb09: MIN = 100,000; MID = 400,000; MAX = 3,200,000. For Gov2 we choose instead: MIN = 100,000; MID = 200,000; MAX = 800,000. In the following analysis, we concentrate on such points.

Table 8 and Table 9 show the bits per posting achieved in correspondence of such trade-off points and the query processing speed for AND queries, respectively, with percentages calculated with respect to PEF as a single reference point. A full experimental comparison is shown in Chapter 7.

As we can see from Table 8, a good space reduction can be achieved when using longer reference sizes, up to 10% and 6% respectively on the clustered Gov2 and ClueWeb09. When considering the whole collections, the space reductions are smaller instead because Elias-Fano is space-inefficient on smaller lists for which the ratio between universe of representation and size is very high.

Also notice from Table 9 that, small reference sizes introduce a moderate slowdown of around 40% and 30% on average for Gov2 and ClueWeb09 respectively, whereas it becomes $\approx 2\times$ in correspondence of the MAX point.

	L1	L2	L3	intersection	ms/query
PEF	174	36	25	–	17.7
CPEF@MIN	216	47	31	9.78%	21.2
CPEF@MID	267	58	39	18.93%	25.0
CPEF@MAX	444	89	54	44.37%	35.6

Table 10: Cache misses ($\times 10^6$) at the three cache levels, average intersection with reference list and mean query time for AND queries on ClueWeb09 using the query set TREC 06.

4.3.3 Analysis

This subsection is dedicated to show why the reference size sensibly affects the query processing speed of our proposal. As already argued, clustering the index inevitably brings a penalty at query time. The penalty comes precisely from the *cache misses* induced by accessing the references. The reason is that as references grow in dimension the number of integers that have been encoded with respect to them grows as well, therefore needing more reference accesses to be decoded. In particular, cache misses are spent in the first level of the partitioned Elias-Fano representation and are due to *chunk-switching* operations, i.e., whenever the distance from two consecutive searched values (jump entity) exceeds the current chunk size.

Reference cache misses. To quantify the impact of the cache misses introduced by the reference lists, we report in Table 10 the number of cache misses in million ($\times 10^6$) for AND queries on ClueWeb09, using the TREC 06 query log. Fourth column of the table reports the average intersection of posting lists with the reference while fifth column reports the sum of the space of the references accessed during the queries over the total space of the references. Cache misses have been collected with the perf Linux tool (version 3.13.11-ckt35). Levels L1 and L2 are private while L3 is shared among all the 8 cores on one socket.

As evident, we increase the number of misses at all levels going from MIN to MAX, confirming the shape of Figure 13. Most importantly, the average percentages of reference intersection reported in the fourth column are proportional to the cache misses at L3 as claimed before. From MIN to MID we have an increase of cache misses at L3 of 27.18% which corresponds to an increase of reference intersection of about 1.93 times. Instead, from MIN to MAX the increase of reference intersection is 4.54 times, therefore we should expect a corresponding increasing of cache misses of about 63.94% which is 53.9 million of cache misses, practically the same as the value reported at MAX.

	p_1	p_2	p_3	partial NextGEQs	ms/query
MIN	0.85	0.04	0.11	1.26	21.2
MID	0.61	0.09	0.29	1.66	25.0
MAX	0.32	0.23	0.45	2.13	35.6

Table 11: Mean number of partial NextGEQ operations and corresponding empirical frequencies p_k for AND queries on ClueWeb09 using the query set TREC 06.

We have repeated the test for the second machine, having the same data cache dimensions except for last level which is 8 MB, again shared among all cores. Results were practically the same. The reason is that all query algorithms have *no temporal locality*: even though some reference cached blocks could be reused for other queries, they will be inevitably deallocated and refetched when needed. Therefore, having a bigger cache will not bring a performance improvement at query time, unless we explicitly decide to keep in cache as many reference blocks as possible. Notice that, however, for the illustrated example, this would only be possible for the MIN point for which the cache is able to contain the whole reference working set as reported in the fifth column of Table 10.

The last column of the table reports the mean query time. Again, we confirm that query processing speed sensibly depends on the number of cache misses. By the values reported in the third column, we should expect to have a slowdown, with respect to PEF, of roughly 24%, 57.8% and 118.5% for MIN, MID and MAX respectively. Indeed mean query times report slowdown factors of 19.7%, 41.2% and 101.2%.

Partial NextGEQs. As discussed in Section 4.2, our posting list organisation may require up to three NextGEQ operations, each operating on the map, residual and reference sequences. We argue that, while this case arises in practice, it is very pessimistic and not the most frequent one. In order to avoid confusion, let us call *partial* a NextGEQ resolved on map, residual or reference and *full* the NextGEQ on the whole clustered list. Beside the worst case, a full NextGEQ may need 2, only 1 or even 0 partial NextGEQ. In Table 11 we report the mean number of partial NextGEQ operations, where p_k represents the probability of performing k partial NextGEQs, $k = 1, 2, 3$ (the probability of performing no NextGEQ, i.e., $k = 0$, is minimal and does not contribute to the calculation of the mean value).

The mean number of partial NextGEQs clearly increases for growing values of reference size. More precisely, we notice an increment proportional to the values presented in Table 10. In fact, the mean number of *extra* partial NextGEQs performed in the three points is 25%, 68% and 113% respectively, while the number of cache misses in excess are 24%, 57.8% and 118.5%. Finally notice that these values are also confirmed by mean query time values that report slowdown factors of 19.7%, 41.2% and 101.2%.

Moreover, it is interesting to notice how each p_i changes in relation to the reference size. In particular, p_1 is decreasing because as the reference grows, map segments grow as well thus reducing the entity of residuals and, consequently, the number of partial NextGEQs performed on them.

Terms per cluster. The crucial parameter affecting the query processing speed is, therefore, the *mean number of accessed reference lists per query*. Intuitively, if all the terms of a query belong to the same cluster, the number of accessed reference lists is just one.

To give a practical evidence of this fact, we conduct the following experiment. For each query we evaluate the ratio between the number of terms and the number of distinct clusters. Taking the average of these quantities among all queries gives us the *mean number of terms per cluster* within a query, indicated with r in the following.

r	1.07	1.48	2.01
MIN	21.2	19.5	14.1
MID	25.0	23.3	17.4
MAX	35.6	33.5	25.2

Table 12: Timings for AND queries by varying terms per cluster ratio on ClueWeb09 using the query set TREC 06.

If ρ is the mean number of terms per query in a query set, then $r \in [1, \rho]$: when $r = 1$, it means that, for each query, all terms belong to distinct clusters; on the other hand, when $r = \rho$ then all terms belong to the same cluster. We test the speed of AND queries on three sampling of 1000 queries from TREC 06, having respectively r equal to 1.07, 1.48 and 2.01. Table 12 illustrates the result. As we can see, when r increases the number of reference lists accessed per query decreases and so does the mean query time. For the other query sets, i.e., TREC 06 for ClueWeb09 and TREC 05 for both Gov2 and ClueWeb09, it is not possible to obtain sufficiently diversified values for r because it is concentrated in the interval $[1.06, 1.09]$.

5 Optimal Variable-Byte Encoding

Variable-Byte [155, 166] (henceforth, VByte) is the most popular and used byte-aligned code. Section 2.2.3 explains how this elegant integer encoding works and Section 2.3 provides general background on inverted indexes. In particular, we recall that VByte owes its popularity to its *sequential decoding speed* and, indeed, it is the fastest representation up to date for integer sequences. For this reason, it is widely adopted by well-known companies as a key database design technology to enable fast query processing. We mention some noticeable examples. Google uses VByte extensively: for compressing the posting lists of inverted indexes [45] and as a binary wire format for its protocol buffers [1]. IBM DB2 employs VByte to store the differences between successive record identifiers [19]. Amazon patented an encoding scheme, based on VByte and called Varint-G8IU (Section 2.2.3), which uses SIMD to perform decoding faster [151]. Many other storage architectures rely on VByte to support fast full-text search, like Redis [2], UpscaleDB [3] and Dropbox [82].

On the other hand, the main drawback of VByte lies in its byte-aligned nature, which means that the number of bits needed to encode an integer cannot be less than 8. For this reason, VByte is only suitable for large numbers. However, as we have explained in Section 2.3.2 with Property 1, the inverted lists are notably known to exhibit a *clustering effect*, i.e., these present regions of close identifiers that are far more compressible than highly scattered regions [115, 120, 129]. Such natural clusters are present because the indexed data itself tends to be very similar. As a simple example, consider all the Web pages belonging to the same site: these are likely to share a lot of terms. Also, the values stored in the columns of databases typically exhibit high locality: that is why column-oriented databases can achieve very good compression and high query throughput [5].

The key point is that efficient inverted index compression should exploit as much as possible the clustering effect of the inverted lists. VByte currently fails to do so and, as a consequence, it is believed to be space-inefficient for inverted indexes.

The motivating experiment. As an illustrative example, consider the following two sequences: $\langle 1, 2, 3, 4, 5\rangle$ and $\langle 127, 254, 318, 408, 533\rangle$. To reduce the values of the integers, VByte compresses the differences between successive values, known as *delta*-gaps or *d*-gaps, i.e., the sequences $\langle 1, 1, 1, 1, 1\rangle$ and $\langle 127, 127, 64, 90, 125\rangle$ respectively (the first integer is left as it is). Now, it is easy to see that VByte will

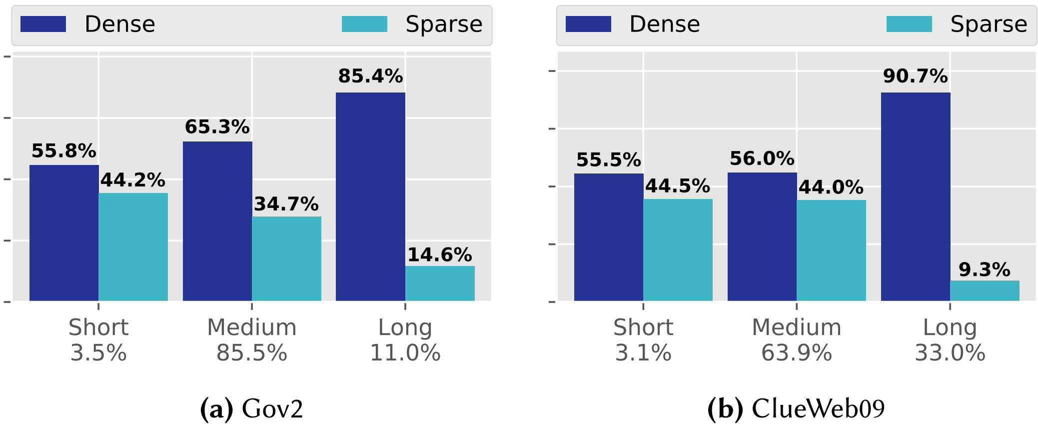


Figure 14: Percentage of integers belonging to dense and sparse regions of the posting lists for the Gov2 and ClueWeb09 datasets. The inverted lists have been clustered by size into three categories: Short (size < 10K), Medium (10K ≤ size < 7M) and Long (size ≥ 7M). Below each category we also indicate the percentage of integers belonging to its inverted lists.

use 5 bytes to encode *both* sequences, but the first one can be compressed much better, with just ≈log 5 bits. To better highlight how this behaviour can deeply affect compression effectiveness, we consider the statistic shown in Figure 14. This statistic reports the percentage of postings belonging to *dense* and *sparse* regions of the lists for the datasets Gov2 and ClueWeb09 (see the statistics in Table 2, Section 1.1). More precisely, the plot originated from the following experiment: we divided each inverted list into chunks of 128 integers and we considered as *sparse* a chunk where VByte yielded the best space usage with respect to the *characteristic bit-vector* representation of the chunk (if u is the last element in the chunk, we have the i-th bit set in a bitmap of size u for all integers i belonging to the chunk), regarded to as the *dense* case. We also clustered the inverted lists by their sizes, in order to show where dense and sparse regions are most likely to be present.

The experiment clearly shows that *we have a majority of dense regions*, thus explaining why in this case VByte is not competitive with respect to bit-aligned encoders and, thus, motivating the need for introducing a better encoding strategy that can adapt to such distribution without compromising the query processing speed of VByte. We can also conclude that such optimization is likely to pay off because the majority of integers, i.e., 85% for Gov2 and 64% for ClueWeb09, concentrate in the lists of medium size (thanks to the Zipfian distribution of words in text), where indeed *more than half* of them belong to dense chunks.

Problem statement. In this chapter we study the problem of partitioning a monotone integer sequence $\mathcal{S}(n, u)$, of size n and universe u, to improve its compression by adopting a *2-level* representation. This data structure stores $\mathcal{S}$ as a sequence of partitions $L_2[\mathcal{S}_1, \ldots, \mathcal{S}_k]$ that are concatenated in the second level L_2. The first level L_1 stores, instead, a fix amount of bits, say F, for each partition $\mathcal{S}_i$, needed to describe its size n_i and largest element u_i. Clearly, F can be safely upper bounded by $O(\log u)$ bits. This representation has several important advantages over a shallow representation:

(1) it permits to choose the most suitable encoder for each partition, given its size and upper bound, hence improving the overall index space;

(2) each partition $\mathcal{S}_i$ can be represented in a smaller universe, i.e., $u_i - u_{i-1} - 1$, by subtracting to all its elements the base value $u_{i-1} + 1$, thus contributing to further reduction in space;

(3) it allows a faster access to the individual elements of $\mathcal{S}$, since we can first locate the partition to which an element belongs to and, then, conclude the search in *that* partition only.

Now, the natural arising problem is *how* to choose the lengths and encoders for each partition in order to minimize the space of $\mathcal{S}$. As already noted, the problem is not trivial since we cannot expect dense regions of the lists being always aligned with fix-sized partitions. While a dynamic programming recurrence offers an optimal solution to this problem in $\Theta(n^2)$ time and $O(n)$ space by just considering the cost of all possible splittings, this approach is clearly unfeasible already for modest sizes of the input. Therefore, we need an efficient algorithm such as the one we describe in this chapter.

Our contributions. We discuss here the main contributions of this chapter.

We disprove the folklore belief that VByte is too large to be considered space-efficient for compressing inverted indexes, by exhibiting an improved compression ratio of 2×. The result is achieved by partitioning the inverted lists into blocks and representing each block with the most suitable encoder, chosen among VByte and the characteristic bit-vector representation. Partitioning the lists has the potential of adapting to the distribution of the integers in the lists, such as the ones shown in Figure 14, by adopting VByte for the sparse regions where larger d-gaps are likely to be present.

Since we cannot expect the dense regions of the lists be always aligned with uniform boundaries, we consider the optimization problem of minimizing the space of representation of an inverted list of size n by representing it with variable-length partitions. To solve the problem efficiently, we introduce an algorithm that finds the *optimal* partitioning in $\Theta(n)$ time and $O(1)$ space.

We remark that the state-of-the-art dynamic programming algorithm in [120] can be used as well to find an $(1+\epsilon)$-optimal solution in $O(n \log_{1+\epsilon} \frac{1}{\epsilon})$ time and $O(n)$ space for any $\epsilon \in (0, 1)$, but it is noticeably slower than our approach and approximated, rather than exact.

Although we conduct our experimental analysis using VByte, we will see that our optimal algorithm can be actually applied to *any point-wise encoder*, that is whenever the chosen encoder needs a number of bits to represent an integer that solely depends on the value of the integer and not on the universe and size of the chunk to which it belongs to.

We conduct an experimental analysis to demonstrate the effectiveness and efficiency of our algorithm on the standard datasets Gov2 and ClueWeb09 described in Section 1.1, Table 2. More precisely, when compared to the un-partitioned VByte indexes, the optimally-partitioned counterparts are: (1) significantly smaller, by 2× on average; (2) only marginally slower at computing boolean conjunctions (by only 5% on the tested architecture); (2) even faster to build on large datasets thanks to the introduced fast partitioning algorithm and improved compression ratio.

We compare the performance of the optimally-partitioned VByte indexes against several state-of-the-art encoders in Chapter 7.

5.1 Related work

The simplest partitioning strategy is to fix the length B of every partition, e.g., $B = 128$ integers, and split the list $\mathcal{S}(n, u)$ into $\lceil n/B \rceil$ blocks (the last partition could be potentially smaller than B integers). We call this partitioning strategy, *uniform*. The advantage of this representation is simplicity, since no expensive calculation is needed prior to encoding. However, we cannot expect this strategy to yield the most compact indexes because the highly clustered regions of inverted lists could be likely broken by such fix-sized partitions.

This is the main motivation for introducing optimization algorithms that try to find the best partitioning of the list, thus minimizing its space of representation. Silvestri and Venturini [149] obtained a $O(n \times h)$ construction time, where n is the length of the inverted list and h its longest partition. Ferragina *et al.* [61] improve the result in [28] by computing a partitioning whose cost is guaranteed to be at most $(1+\epsilon)$ times away from the optimal one, for any $\epsilon \in (0, 1)$, in $O(n \log_{1+\epsilon} n)$ time. Their approach can be applied to any encoder E whose cost in bits can be computed (or, at least, estimated) in constant-time, for any portion of the input.

Dynamic programming: slow and approximated. The core idea of this approach is to not consider *all* possible splittings, but only the ones whose cost is able of amortizing the fix cost F. Inspired by the ideas in [61], Ottaviano and Venturini [120] obtained a running time of $O(n \log_{1+\epsilon} \frac{1}{\epsilon})$, and preserved the same approximation guarantees. We point the reader to Section 2.2.7 and to the original paper [120] for the description of the dynamic programming algorithm. Here, we recall that it finds a solution whose encoding cost is at most $(1+\epsilon)$ away from the optimal one in $O(n \log_{1+\epsilon} \frac{1}{\epsilon})$ time and $O(n)$ space, for any $\epsilon \in (0,1)$. Note that the time complexity is linear as soon as ϵ is constant.

Despite the *theoretical* linear-time complexity for a constant ϵ, the main drawback of the algorithm lies in the high constant factor. For example, even by setting $\epsilon = 0.03$ (as done in the experimental evaluation of the original paper), we obtain a hidden constant of $\log_{1+0.03} 33.33 \simeq 118.63$, which results in a noticeable cost in practice. Although enlarging ϵ can reduce the constant at the price of reducing the compression efficacy, this remains the bottleneck for the building step of large inverted indexes.

5.2 Optimal partitioning in linear time: fast and exact

The interesting research question we now pose is whether there exist an algorithm that finds an *exact* solution, rather than approximated, in linear time and with *low* constant factors. This chapter answers positively to this question by showing that if the cost function of the chosen encoder is *point-wise*, i.e., the number of bits need to represent a single posting solely depends on such posting and not on the universe and size of the partition it belongs to, the problem of determining and optimal partition admits an *exact* and fast solution in $\Theta(n)$ time and $O(1)$ space.

In the following, we first overview and discuss our solution by explaining the intuition that lies at its core, then we give the full technical details along with a proof of optimality and the relative pseudo-code.

5.2.1 Overview

We are interested in computing the partitioning of $\mathcal{S}$ whose encoding cost is minimum by using *two* different encoders that take into account the relation between the size and universe of each partition. We already motivated the potential of this strategy by commenting on Figure 14, which shows the distribution of the integers in dense and sparse regions of the inverted lists. Let us consider the partition $\mathcal{S}[i,j)$, $0 \leq i < j \leq n$, of relative universe $m = \mathcal{S}[j-1] - \mathcal{S}[i-1] - 1$ and size $b = j - i$. Intuitively, when b gets closer to u the partition becomes denser, vice versa, it becomes sparser whenever b diverges from m. Thus the encoding cost $C(\mathcal{S}[i,j))$ is chosen to be the minimum between $\mathrm{B}(S[i,j)) = u$ bits (dense case) and

$\mathsf{E}(\mathcal{S}[i,j))$ bits (sparse case), where B is the characteristic bit-vector of $\mathcal{S}[i,j)$ and E is the chosen point-wise encoder for sparse regions.

In Section 2.2 we described some encoders that are *point-wise*, such as, for example, VByte, Elias- γ and δ and Golomb-Rice. Other encoders, such as Elias-Fano (and partitioned Elias-Fano), BIC and PForDelta are *not* point-wise, since a different number of bits could be needed to represent the *same* integer when belonging to partitions having different characteristics, namely different length and universe. To clarify what we mean, consider the following exemplar sequence

$$\mathcal{S}[0,10) = \langle 8, 9, 10, 11, 12, 36, 37, 38, 39, 40 \rangle$$

Let us now compare the behaviour of Elias-Fano (non point-wise) and VByte (point-wise). By performing no splitting, Elias-Fano will use $\lceil \log(40/10) \rceil + 2 = 4$ bits to represent every posting. By performing the splitting $[0,5)[5,10)$, the first five values will be represented with 4 bits each, but the next five values will be represented with $\lceil \log(40-12-1)/5 \rceil + 2 = 5$ bits each. Instead, by performing the splitting $[0,6)[6,10)$, the first six values will use 5 bits each, while the next four only 2 bits each. Thus, performing different cuts change the cost of representation of the *same* postings for a non point-wise encoder, such as Elias-Fano. Instead, it is immediate to see that VByte will encode each element with 8 bits, regardless any partitioning. Obviously, we do not have to try many (or even, all) different cuts in order to know the best number of bits we should use to represent an element.

The intuition. The above example gives us an intuitive explanation of why it is possible to design a light-weight approach for a point-wise encoder E: we can compute the number of bits needed to represent a partition of $\mathcal{S}$ with E by just scanning its elements and summing up their costs, *knowing that performing a splitting will not change their cost of representation* nor, therefore, the one of the partition. This means that as long as the cost $\mathsf{E}(\mathcal{S}[0,j))$, for some $0 < j \leq n$, is less than $\mathsf{B}(\mathcal{S}[0,j))$ we know that $\mathcal{S}[0,j)$ will be represented optimally with E. Therefore, we can safely keep scanning the sequence until the difference in cost between $\mathsf{E}(\mathcal{S}[0,j))$ and $\mathsf{B}(\mathcal{S}[0,j))$ becomes more than F bits. At this point, it means that E is wasting more than F bits with respect to B, thus we should stop encoding with E the current partition because we can afford to pay the fix cost F and continue the encoding with B. Now, the crucial question we would like to answer is: at which position $k < j$ should we stop encoding with E and switch to B? The answer is simple: we should stop at the position $k < j$ at which we saw the *maximum* difference between the costs of E and B, because splitting in *any* other point will yield a larger encoding cost. In other words, k represents the position at which E *gains most* with respect to B, so we will be wasting bits by splitting before or after position k. Observe that we must also require such gain be more than F bits,

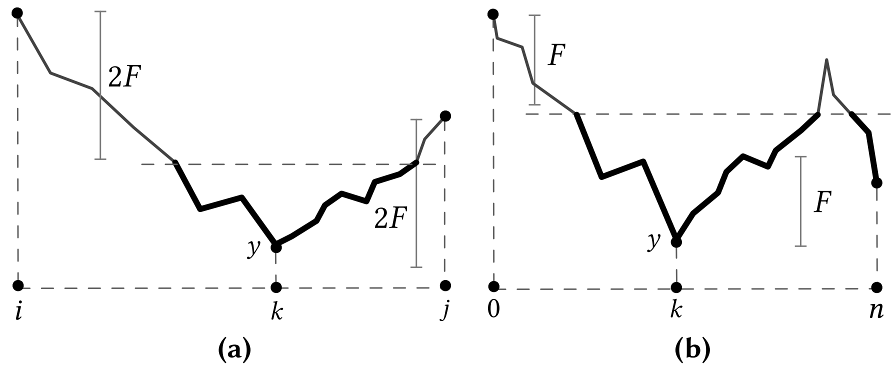


Figure 15: In case **(a)**, we should split $\mathcal{S}[i, j)$ in correspondence of position k because there the gain is the minimum among all points whose gain is below $2F$ from $\mathcal{S}[i]$ *and* $\mathcal{S}[j]$; in case **(b)** we should *not* split the sequence because, although an increase in gain of F bits follows, we do not have a sufficiently high gain up to $\mathcal{S}[n-1]$ to amortize the cost of the splitting.

otherwise switching encoder will actually cause a waste of bits. In other terms, we say that in such case the gain would not be sufficient to amortize the fix cost of the partition, meaning that we should *not* split the sequence yet.

In conclusion, we encode $\mathcal{S}[0, k)$ with E and know that the elements $\mathcal{S}[k, j)$ will be now best represented with B, rather than with E. Figure 15 offers a pictorial representation of how the difference between the encoding costs of E and B, referred to as the *gain* function, changes during the scan of $\mathcal{S}$. When the function is decreasing, it means that E is winning over B, i.e., its encoding cost is less; conversely, when B is more efficient than E, the function is increasing.

After encoding the first partition $\mathcal{S}[0, k)$, the process repeats: (1) we keep scanning $\mathcal{S}$ until B loses more than $2F$ bits with respect to E; (2) we encode with B the elements in $\mathcal{S}[k, k')$ if the maximum gain of B with respect to E, seen at position k', is greater than $2F$ bits. We keep alternating compressors until the end of the sequence.

Before sketching a compact pseudo-code of our algorithm, we first express some considerations. First of all note that, for all partitions except the first, we need to amortize *twice* the fix cost, because we could potentially merge the last formed partition with the current one, thus, in order to be beneficial, the difference in the cost of the two encoders must be larger than $2F$ bits. Again, refer to Figure 15a for an example. Also, for illustrative purposes, in the previous discussion we have assumed that the first partition is best encoded with E: clearly, B could be better at the beginning but the algorithm will work in the very same way.

5.2.2 The algorithm

In the most general terms, call L the encoder used to represent the *last* encoded partition and C the *current* one. These will be either E or B. We also indicate with the same letters the costs in bits of their representation of the current partition. Finally, let g^* indicate the best gain of C with respect to L. At a high level, the skeleton of our algorithm looks as follows.

(1) Encode the first partition.

(2) If $|\mathsf{C} - \mathsf{L}|$ and g^* are greater than $2F$ bits, encode the current partition with C and swap the roles of C and L.

(3) Repeat step 2. until the end of the sequence.

(4) Encode the last partition.

In the above pseudo-code, the encoding of the first and last partitions its treated separately because these must amortize a fix cost of F bits instead of $2F$ bits, because we do not have any partition before and after, respectively (see Figure 15b).

It is immediate to see that the described approach can be implemented by using $O(1)$ space because we only need to keep the difference between the costs of E and B (plus some cursor variables), and that it runs in $\Theta(n)$ time because we calculate the cost in bits of each integer exactly once. We have, therefore, eliminated the linear-space complexity of *any* dynamic programming approach because we do not need to maintain the costs of the shortest path ending in each position of $\mathcal{S}$. Moreover, the introduced algorithm has very low constant factors in the time complexity, since it just performs few comparisons and updates of some variables for each integer of $\mathcal{S}$.

5.2.3 Technical discussion

Let $g : \mathbb{N} \cup \{0\} \to \mathbb{Z}$ be the *gain* function, defined as

$$g(\mathcal{S}[j]) = \sum_{i=0}^{j-1} \Big[\mathsf{E}(\mathcal{S}[i]) - \mathsf{B}(\mathcal{S}[i])\Big], \text{ for } j = 0, \ldots, n-1.$$

In order to describe the properties of our solution, we first need the following definition.

Definition 2 — Given $\mathcal{S}[i, j)$, $0 \le i < j \le n$, the integer $y \in \mathcal{S}[i, j)$ is the *point dominating* $\mathcal{S}[i, j)$ for the encoder E, if

$$y = \arg\min_{i<k\le j} g(\mathcal{S}[k]) \text{ such that } g(\mathcal{S}[i]) - g(y) > T, \tag{6}$$

where $T = F$ if $i = 0$ or $2F$ otherwise, and $\mathcal{S}[j]$ satisfies one of the following:

$$g(\mathcal{S}[j]) - g(y) > 2F, \text{ or} \tag{7}$$
$$g(z) - g(y) > F, \text{ for all } z \ge \mathcal{S}[j]. \tag{8}$$

Notice that the dominating point could not exist for any sub-sequence $\mathcal{S}[i, j)$, but if it exists and $\mathsf{E}(x) \neq \mathsf{B}(x)$ for any $x \in \mathcal{S}[i, j)$, it must be unique. Clearly, the definition of dominating point for encoder B is symmetric to Definition 2.

The above definition explains that, given $\mathcal{S}[i, j)$, we can *always improve* its cost of representation by splitting the interval in correspondence of the dominating point y if it exists, otherwise we should *not* split $\mathcal{S}[i, j)$. It is easy to see that the dominating point in $\mathcal{S}[i, j)$ is the point in which the difference of the costs between the two compressors is *maximized*, thus it will be only beneficial to split in this point rather than any other point, as we explained previously.

It is also easy to see why we should search the dominating point among the ones whose gain is at least T bits less than $g(\mathcal{S}[i])$. The threshold T is set to the minimum amount of bits needed to amortize the cost of switching from one compressor to the other. Consider Figure 15a and suppose we are encoding with B before $\mathcal{S}[i]$ and after y. If we compress with E the partition $\mathcal{S}[i, k)$, we are switching encoder twice, thus the gain in y must be at least $2F$ bits less than $g(\mathcal{S}[i])$ to be able of amortizing the cost for two switches. In Figure 15b, instead, we have no partition before $\mathcal{S}[0]$, thus we strive to amortize the cost for a single switch.

Now, let $p(x) \in [0, n)$ be the position of integer x in $\mathcal{S}[0, n)$, i.e., $\mathcal{S}[p(x)] = x$. *Our strategy consists in splitting the sequence in correspondence of the dominating points*, as defined above. More precisely, the solution $\mathcal{P} = \langle p_1, \ldots, p_k \rangle$ output by this strategy can be described by the following recursive equation

$$p_i = p(y_i), (y_{i-1}, y_i, y_{i+1}), \text{ for } i = 1, \ldots, k, \tag{9}$$

where $y_0 = \mathcal{S}[0]$, $y_{k+1} = \mathcal{S}[n-1]$ and notation $(\mathcal{S}[i], y, \mathcal{S}[j])$ means that y is the point dominating $\mathcal{S}[i, j)$. In other words, any $p_i \in \mathcal{P}$, except for the first and the last, is the dominating point of the interval whose endpoints are dominating points as well.

Notice that, by definition, there cannot be two adjacent dominating points that are relative to the same encoder, but they must be relative to different encoders.

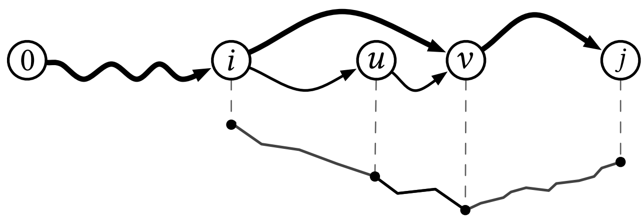


Figure 16: Path of minimum cost till position j (thick black line) and its representation in terms of gain function; u is the position of the point dominating $\mathcal{S}[i, j)$.

In fact, suppose we have a dominating point y for E: it means that either we have seen an increase in gain of $2F$ after y and, thus, the dominating point after y (if found) will be relative to B, or the gain never goes under y and a dominating point after y if not found. This means that $\mathcal{P}$ alternates the choice of compressors, i.e., a partition encoded with E is delimited by two partitions encoded with B and vice versa (except for the first and last). We call such behaviour, *alternating*.

In particular, our strategy will encode with compressor E all partitions ending with a dominating point for E (and starting with a dominating point for B, since $\mathcal{P}$ alternates the compressors). The same holds for B. As already pointed out, the only exception is made for the last partition $\mathcal{S}[p_k, n)$, because $\mathcal{S}[n-1]$ cannot be a dominating point by definition (no increase or decrease in gain is possible after the end of the sequence). In this case, the strategy selects the compressor that yields the minimum cost over $\mathcal{S}[p_k, n)$.

Since a *feasible solution* to the problem is just either a singleton partition or consists in any sequence of strictly increasing positions, we argue that $\mathcal{P}$ is a feasible solution. This follows automatically by the definition of dominating point because such points are different one another and, therefore, their positions strictly increasing. If no dominating points exist, then $\mathcal{P}$ will be empty: it is a feasible solution too and indicates that $\mathcal{S}$ should not be cut (singleton partition). We now show the following lemma.

Lemma 3 — $\mathcal{P}$ is *optimal.*

Proof. As already noted in Section 2.2.7, an optimal solution to the problem can be thought as a path of minimum cost in the DAG whose vertices are the positions of the integers of $\mathcal{S}$ and $C(i, j) = C(\mathcal{S}[i, j])$ for any edge (i, j). Thus, suppose that $\mathcal{P}$ is not a shortest path and let $\mathcal{P}^* = [p_1^*, \ldots, p_m^*]$ be the shortest path sharing the longest common prefix with $\mathcal{P}$. Refer to Figure 16 for a graphical representation: i is the largest position shared by $\mathcal{P}^*$ and $\mathcal{P}$. We want to show that we can replace the edge (i, v), $v \in \mathcal{P}^*$, with the path $(i, u)(u, v)$, $u \in \mathcal{P}$, without changing the cost of $\mathcal{P}^*$, therefore extending the longest common prefix up to node $u < v$ (the case for $u > v$ is symmetric). We argue that this is only possible if $\mathcal{P}$ is optimal, otherwise it would mean that $\mathcal{P}^*$ is not a shortest path sharing a common longest prefix with $\mathcal{P}$, which is absurd by assumption.

First note that both edges (i, v) and (i, u) must be encoded with the same compressor. In fact, suppose that these are not, for example (i, v) is encoded with B and (i, u) with E. Since $v \in \mathcal{P}^*$, we know that it is optimal to encode with B until v. However, the fact that u is a dominating point for E implies that $\mathsf{B}(i, u) > \mathsf{E}(i, u)$, which is absurd because $u < v$ and B is optimal until v. Therefore, both edges must use the same encode. Assume that it is E (the case for B is symmetric).

The fact that v belongs to the optimal solution $\mathcal{P}^*$ means that if we split the edge into two (or more) pieces, we cannot decrease the cost, i.e., $\mathsf{E}(i, v) \leq \mathsf{E}(i, k) + \mathsf{B}(k, v) + F$, $\forall i \leq k \leq v$. Since E is point-wise, we have $\mathsf{E}(i, v) - \mathsf{E}(i, k) = \mathsf{E}(k, v)$ and thus, by imposing $k = u$, we obtain (1) $\mathsf{E}(u, v) \leq \mathsf{B}(u, v) + F$. Vice versa, the fact that u is a dominating point for E means that from u to v the cost is higher if we keep the same encoder, i.e., $\mathsf{E}(i, v) \geq \mathsf{E}(i, u) + \mathsf{B}(u, v) + F$. Again, by exploiting the fact that E is point-wise, we have (2) $\mathsf{E}(u, v) \geq \mathsf{B}(u, v) + F$. Conditions (1) and (2) together imply that it must be $\mathsf{E}(u, v) = \mathsf{B}(u, v) + F$, thus we have no change in the cost of $\mathcal{P}^*$ by performing the exchange, which contradicts our assumption that $\mathcal{P}$ was not optimal.

■

We are now left to present a detailed algorithm that computes $\mathcal{P}$, i.e., that iteratively finds all the dominating points of $\mathcal{S}$ according to equation 9. The argue that the function **`optimal_partitioning`** shown in Figure 18 does the job. Before proving that the algorithm is correct, let us explain the meaning of the variables used in the pseudo-code. Call ℓ the last added position to $\mathcal{P}$. Variables i and j keep track of the positions of the points dominating the interval $\mathcal{S}[\ell, k)$ for, respectively, B and E encoders. Likewise, *max* and *min* are the gains in correspondence of positions i and j; g indicates the gain at step k, for $k = 0, \ldots, n-1$.

Lemma 4 — The algorithm shown in Figure 18 is *correct.*

Proof. We want to show that the array $\mathcal{P}$ returned by **`optimal_partitioning`** contains all the positions of the dominating points, as recursively described by equation 9. We proceed by induction on the elements of $\mathcal{P}$.

The main loop in lines 6-17:

(1) computes the gain g at step k (line 7);

(2) updates the variables i, max (lines 9-10) and j, min (lines 14-15);

(3) add new positions to $\mathcal{P}$ (lines 11-12 and 16-17).

Correctness of points (1) and (2) is immediate: the crucial one to explain is the third.

```
1  optimal_partitioning(S)
2      P = ∅
3      T = F
4      i = j = g = 0
5      min = max = 0
6      for k = 0; k < n; k = k + 1
7          g = g + E(S[k]) − B(S[k])
8          if g is non-decreasing
9              if g > max
10                 max = g, i = k + 1
11             if min < −T and min − g < −2F
12                 update(min, max, j, i)
13         else
14             if g < min
15                 min = g, j = k + 1
16             if max > T and max − g > 2F
17                 update(max, min, i, j)
18     close()
19     return P
```

Figure 18: The **optimal_partitioning** algorithm.

The `if` statements in lines 11 and 16 check whether positions i and j are the positions of a dominating point in $\mathcal{S}[\ell, k)$, i.e., whether $\mathcal{S}[i]$ and $\mathcal{S}[j]$ satisfy Definition 2. Since the `if` statements are symmetric, we proved the correctness of the first one for non-decreasing values of g (line 11).

```
update(g0, g1, p0, p1)
    P.append(p0)
    T = 2F
    p1 = k + 1
    g = g − g0
    g0 = 0
    g1 = g
```

Figure 17: The **update** algorithm used in the pseudo code of Figure 18.

We first check whether the *min* gain, as seen so far, is sufficient to be the one of a dominating point for E as required by equation 6. At the beginning of the algorithm, the current interval starts at $i = 0$ and $T = F$, therefore $g(\mathcal{S}[0]) = 0$ in equation 6 and the test $min < -T$ is correct. If $min < -T$ is true, then we also check if we have a sufficiently large increase in gain at the current step k with respect to the previously seen *min* gain according to condition 7.

Again, it is immediate to see that the test $min - g < -2F$ checks such condition and, therefore, it is correct. If both previous conditions are satisfied, then j is the position of the dominating point for E in the first interval $\mathcal{S}[0, k)$ by Definition 2.

```
1 close()
2     if max > F and max − g > F
3         update(max, min, i, j)
4     if min < −F and min − g < −F
5         update(min, max, j, i)
6     if g > 0
7         i = n, update(max, min, i, j)
8     else
9         j = n, update(min, max, j, i)
```

Figure 19: The **close** algorithm used in the pseudo code of Figure 18.

If so, we can execute the **update** code, show in Figure 17, which adds j to $\mathcal{P}$ and sets T to $2F$ according to Definition 2. Moreover, it updates the gain g to maintain the invariant that its value is always relative to the current interval, which now begins at position j. In fact: since we have seen an increase of $2F$ bits, the *max* gain in $\mathcal{S}[j, k)$ must be the current gain g, whereas the *min* gain is 0 because g is non-decreasing. Thus, the first point is computed correctly.

Now, assume that we have added m points to $\mathcal{P}$ and that the last added is for encoder E. We want to show that the next point will be dominating for encoder B. As explained before, whenever we add a dominating point for E to $\mathcal{P}$, it means that we have seen an increase of $2F$ bits with respect to the last added position, i.e., position $k + 1$ is the one of a point that satisfies equation 6 for encoder B. Therefore the $(m + 1)$-th point added to $\mathcal{P}$ will be dominating for B.

To conclude, we have to explain what happens at the end of the algorithm. Refer to the **close** function, illustrated in Figure 19. Lines 1-2 (3-4) check condition 8: if successful, then *max* (*min*) is the next dominating point for B (E) and, since compressors must alternate each other, we close the encoding of the sequence with the other compressor in lines 5-8, that is E (B); if both unsuccessful, i.e., no dominating point is found, then it means that the remaining part of the sequence should not be cut and, thus, encoded with a single compressor in lines 5-8.

■

In conclusion, since we consider each element of $\mathcal{S}$ once and use a constant number of variables, Lemma 3 and Lemma 4 imply the following result.

Theorem 5 — A monotone integer sequence of size n can be partitioned *optimally* in $\Theta(n)$ time and $O(1)$ space, whenever its partitions are represented with a *point-wise* encoder and *characteristic bit-vectors*.

5.3 Experiments

Datasets. We performed our experiments on the datasets described in Section 1.1, whose statistics are summarized in Table 2 (page 7).

Experimental setting and methodology. All the experiments were run on a machine with 4 Intel i7-4790K CPUs (8 threads) clocked at 4.00GHz and with 32GB of RAM DDR3, running Linux 4.13.0 (Ubuntu 17.10), 64 bits. The implementation of our partitioned indexes is in standard C++14 and it is a flexible template library allowing *any* point-wise encoder to be used, provided that its interface exposes a method to compute the cost in bits of a single integer in constant time. The source code was compiled with gcc 7.2.0 using the highest optimization setting.

To test the building time of the indexes we measure the time needed to perform the whole building process, that is: 1. accessing the inverted lists from disk; 2. encoding them in main memory; 3. saving the whole index data structure back to a file on disk. Since the process is mostly I/O bound, we make sure to avoid disk caching effects by clearing the disk cache before building the indexes.

To test the query processing speed of the indexes, we memory map the index data structures on disk and compute boolean conjunctions over a set of random queries drawn from TREC 05 and TREC 06 Efficiency Track topics. We repeat each experiment three times to smooth fluctuations in the measurements and consider the mean value. The query algorithm runs on a single core and timings are reported in milliseconds.

In all the experiments, we use the value $F = 64$ bits for partitioning the inverted lists. For block-based methods, we use blocks of 128 postings, with trailing elements encoded with BIC as it is standard for all methods in the ds2i framework.

Source code. `https://github.com/jermp/opt_vbyte`

Organization. The aim of this section is to measure the space improvement, indexing time and query processing speed of indexes that are optimally partitioned by the algorithm described in Section 5.2. Since we adopt VByte as exemplar point-wise encoder, the next subsection compares the performance of all the encoders in the VByte family in order to chose the most convenient for the subsequent experiments. Then, we measure the benefits of applying our optimization algorithm on the chosen VByte encoder, by comparing the corresponding partitioned index against the un-partitioned counterpart.

	space [GB]	docs [bpi]	freqs [bpi]	building [min]	AND query [ms/query]
Varint-GB	15.06	11.15	9.77	10.60	0.88
Varint-G8IU	14.00	10.43	9.00	18.00	0.84
Masked-VByte	12.64	9.53	8.02	10.50	0.90
Stream-VByte	15.06	11.15	9.77	10.60	0.86

(a) Gov2

	space [GB]	docs [bpi]	freqs [bpi]	building [min]	AND query [ms/query]
Varint-GB	42.23	11.43	9.80	46.50	5.32
Varint-G8IU	39.43	10.84	8.99	65.80	5.10
Masked-VByte	35.63	9.91	8.01	45.50	5.52
Stream-VByte	42.23	11.43	9.80	46.50	5.30

(b) ClueWeb09

Table 13: The performance of the Variable-Byte family on Gov2 and ClueWeb09: space in GB; average number of bits (bpi) for documents (docs) and frequencies (freqs); index building time in minutes and query processing time for TREC 05 AND queries in milliseconds.

5.3.1 The Variable-Byte family

To help us in deciding which VByte encoder to chose for our subsequent analysis, we consider Table 13. The data reported in the table illustrates how different VByte stream organizations actually impact index space.

Stream-VByte takes exactly the same space of Varint-GB because it stores the very same control and data bytes but concatenated in separate streams. For this reason, we will refer to both versions as Varint-GB. As we can see, the original VByte format (referred to as Masked-VByte in Table 13 because it uses this algorithm to perform decoding) is the most space-efficient among the family. This is not surprising given the distribution plotted in Figure 14 (page 76): it means that the majority of the encoded d-gaps falls in the interval $[0, 2^7)$, otherwise the compression ratio of VByte would have been worse than the one of Varint-GB and Varint-G8IU.

As an example, consider the sequence of d-gaps $\langle 132, 233, 246, 178 \rangle$. VByte uses 8 bytes to represent such sequence, whereas Varint-GB uses 1 control byte and

4 data bytes, thus 5 bytes in total. When all values are in $[0, 2^7)$, VByte uses 4 bytes instead of 5 as needed by Varint-GB. For this reason, the space usage of Varint-GB and Varint-G8IU is worse than the one of VByte: it is 16% and 15% on Gov2 and ClueWeb09 respectively for Varint-GB; more than 10% for Varint-G8IU. The control byte of Varint-G8IU stores a bit for every of the 8 data bytes: a 0 bit means that the corresponding byte completes a decoded integer, whereas a 1 bit means that the byte is part of a decoded integer or it is wasted. Thus, Varint-G8IU compress worse than plain VByte due to the wasted bytes. Finally notice that Varint-GB is slightly worse than Varint-G8IU because it uses 10 bits per integer instead of 9 for all integers in $[0, 2^8)$. In fact, the difference between these two encoders in less than 1 bit on both Gov2 and ClueWeb09.

The speed of the encoders is actually very similar for all alternatives. Among the ones taking less space, i.e., Varint-G8IU and Masked-VByte, the spread between the fastest and the slowest alternative is as little as $6 \div 8\%$. The same holds true for the building of the indexes where, as expected, the plain VByte is the fastest and Varint-G8IU is 40% slower on average.

In conclusion, for the reasons discussed above, i.e., better space occupancy, fastest index building time and competitive speed, we adopt the original VByte stream organization and the Masked-VByte algorithm by Plaisance, Kurz and Lemire [132] to perform sequential decoding.

5.3.2 Optimized Variable-Byte indexes

In this subsection, we compare the optimally-partitioned VByte indexes against the un-partitioned indexes and the ones obtained by using other partitioning strategies, like the ϵ-optimal based on dynamic programming (see Section 2.2.7) and the uniform one.

The result of the comparison shows that our optimally-partitioned VByte indexes are 2× smaller than the original, un-partitioned, counterparts; can be built 2.6× faster than dynamic programming and offer the strongest guarantee, i.e., an exact solution rather than an ϵ-approximation; despite the significant space savings, they are as fast as the original VByte indexes.

Index space. Table 14 shows the results concerning the space of the indexes. Compared to the case of un-partitioned indexes, we observe gains ranging from 51% up to 222%, with a net factor of 2× improvement with respect to the original VByte format. For the uniform partitioning we used partitions of 128 integers, for both documents and frequencies. As we can see, this simple strategy already produces significant space savings: it is 43.3% and 25.6% better on doc sequences for Gov2 and ClueWeb09 respectively; 58.7% and 66.3% better on freq sequences. This is be-

	space [GB]		docs [bpi]		freqs [bpi]		building [min]	
VByte optimal	5.68		4.87		3.04		10.50	
VByte	12.64	(+122.74%)	9.53	(+95.75%)	8.02	(+163.92%)	10.10	(−3.81%)
VByte uniform	6.26	(+10.22%)	5.41	(+11.05%)	3.31	(+8.92%)	11.30	(+7.62%)
VByte ϵ-optimal	5.73	(+0.93%)	4.93	(+1.21%)	3.05	(+0.49%)	26.70	(+154.29%)

(a) Gov2

	space [GB]		docs [bpi]		freqs [bpi]		building [min]	
VByte optimal	17.88		6.54		2.48		28.50	
VByte	35.63	(+99.26%)	9.90	(+51.52%)	8.01	(+222.39%)	43.30	(+51.93%)
VByte uniform	19.95	(+11.58%)	7.37	(+12.73%)	2.69	(+8.54%)	29.30	(+2.81%)
VByte ϵ-optimal	18.15	(+1.53%)	6.66	(+1.84%)	2.50	(+0.68%)	72.30	(+153.68%)

(b) ClueWeb09

Table 14: Space in GB, average number of bits (bpi) for documents (docs) and frequencies (freqs) and index building time in minutes.

cause most d-gaps are actually very small but *any* un-partitioned VByte encoder needs at least 8 bits per d-gap. In fact, notice how the average bits per integer on the doc sequences becomes less than 8.

We remark that the ϵ-optimal algorithm based on dynamic programming was proposed for Elias-Fano, whose cost in bits can be computed in $O(1)$: we adapt the dynamic programming recurrence in order to use it for VByte too. As approximation parameters we used the same as in the experiments of the original paper [120], i.e., we set $\epsilon_1 = 0.03$ and $\epsilon_2 = 0.3$. The computed approximation could be possibly large by enlarging such parameters, while our algorithm finds an exact solution. However, we notice that the ϵ-approximation is good and our optimal solution is slightly better: precisely by 1.2% and 1.84% on the doc sequences of Gov2 and ClueWeb09, respectively. Compared to uniform, the optimal partitioning pays off: indeed it produces a further saving of 11% on average, thus confirming the need for an optimization algorithm.

Index building time. Although the un-partitioned variant would be the fastest to build in internal memory because the posting lists are compressed in the same pass in which these are read from disk, the writing of the data structure to the disk imposes a considerable overhead because of the high memory footprint of

		Gov2		ClueWeb09	
TREC 05	VByte optimal	0.89		5.70	
	VByte	0.90	(+1.37%)	5.56	(−2.54%)
	VByte uniform	0.94	(+5.07%)	5.90	(+3.45%)
	VByte ϵ-optimal	0.92	(+2.70%)	5.89	(+3.34%)
TREC 06	VByte optimal	2.12		8.96	
	VByte	2.12	(+0.02%)	8.35	(−6.90%)
	VByte uniform	2.22	(+4.98%)	9.02	(+0.60%)
	VByte ϵ-optimal	2.24	(+5.77%)	9.17	(+2.31%)

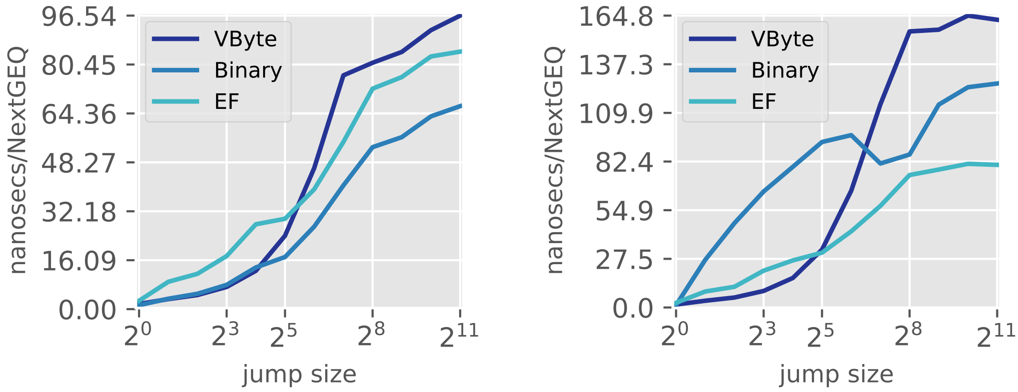
Table 15: Timings for AND queries in milliseconds per query.

the un-partitioned index. Notice how this factor becomes dramatic for the larger dataset ClueWeb09, resulting in a 50% overhead. This also causes the simple uniform strategy be not faster at all, being actually a little slower (by 5% on average). Despite the linear-time complexity as soon as ϵ is constant, the ϵ-optimal solution has a noticeable CPU cost due to the high constant factor, as we motivated in Section 5.2. The optimal solutions has instead low constant factors and, as a result, is faster than the dynamic programming approach by more than 2.6× on average on both Gov2 and ClueWeb09.

AND queries. Table 15 illustrates the results. The striking result of the experiment is that, despite the significant space reduction (2× improvement, see Table 14), the partitioned indexes are as fast as the un-partitioned ones on both the Gov2 and ClueWeb09 datasets.

It is, therefore, important to provide a careful explanation of such result. The answer is provided by understanding the plots in Figure 20, along with the ones in Figures 14 (page 76) and 21. In particular, Figure 20 illustrates the average nanoseconds spent per NextGEQ query by VByte, the binary vector representation and Elias-Fano (EF) too (as a useful reference point that we will exploit in Chapter 7 for a full comparison) on a sequence of one million integers and with an average gap of: **(a)** 2.5, as a *dense* case and **(b)** 1850 as a *sparse* case. As the dense case illustrates, the binary vector representation is as fast as VByte for all jumps of entity less then or equal to 8, and becomes actually faster for longer jumps.

Moreover, the distribution of the jump sizes plotted in Figure 21 indicates us that, whenever executing AND queries, the number of jumps of size less than 16 accounts for ≈90% of the jumps performed by NextGEQ. Furthermore, the distri-



(a) Dense **(b)** Sparse

Figure 20: Nanoseconds per NextGEQ query for **(a)** dense and **(b)** sparse sequences of one million integers, having an average gap of 2.5 and 1850, respectively. These values mimic the ones for the Gov2 datasets, that are 2.125 and 1852. The ones for the ClueWeb09 dataset are 2.137 and 963, and the plots have a similar shape.

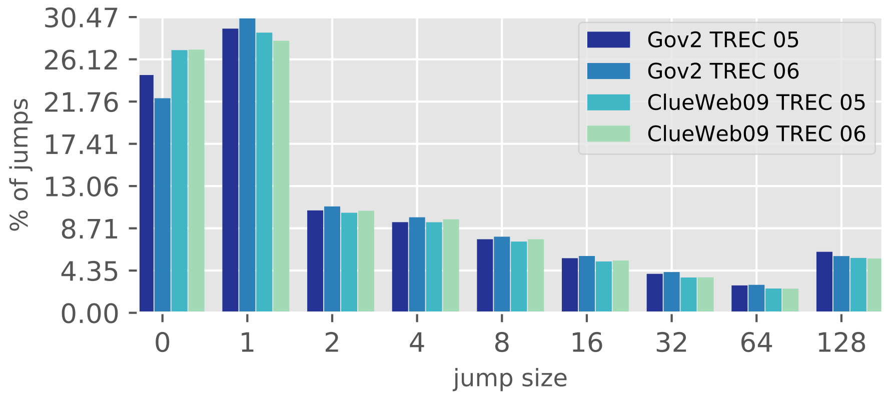


Figure 21: When the difference between two consecutive accessed positions is d, NextGEQ is said to make a jump of size d. The distribution of the jump sizes is divided into buckets of exponential size: all sizes between 2^{d-1} and 2^d belong to bucket d. The plot shows the jumps distribution, in percentage, for the query logs used in the experiments, when performing AND queries.

bution plotted in Figure 14 tells us that the majority of blocks are actually encoded with their characteristic bit-vector, thus explaining why the partitioned indexes exhibit no penalty against the un-partitioned counterparts.

However, VByte tends to be slower on longer jumps because of its block-wise organization: since a posting list is split into blocks of 128 postings that are encoded separately, a block must be completely decoded even for accessing a single integer, which is not uncommon for boolean conjunctions. Moreover, since *d*-gaps values are encoded, we need to access the elements by a linear scan of the block after decoding in order to compute their prefix sums. When the accessed elements per block are very few, even using SIMD instructions to perform decoding results in a slower query execution. Conversely and as expected, the binary vector representation is inefficient for the *sparse* regions since potentially many bits need to be scanned to perform a query, but still faster than VByte whenever the jump size becomes larger than 64 because it allows skipping over the bit stream by keeping samples of the bit set positions.

6 Dictionary-based Decoding

The many approaches devised for inverted index compression described in Section 2.2 can be broadly categorized into two classes: *fixed-to-variable* and *variable-to-fixed*.

In a *fixed-to-variable* arrangement, a fixed number of integers is encoded using a variable number of bits. The simplest incarnation of this approach is to split an inverted list into blocks and then represent each integer of a block in $\lceil \log max \rceil$ bits, where *max* is the maximum element of the block. A more sophisticated example is the well-known *patched frame of reference* (PFOR) mechanism, where a bit-width is chosen that covers most of the values in the block, but not necessarily all of them, and any values that require more than that many bits (referred to as *exceptions*) are represented using a secondary *patching* mechanism.

Conversely, in a *variable-to-fixed* arrangement, a variable number of integers is fit into a single machine word, namely 32 or 64 bits. A noticeable example of this approach is the Simple family, i.e., Simple9, Simple16 and Simple8b. For example, Simple9 uses 4 *header* bits and 28 *data* bits. The header bits provide information on how many integers are packed in the data segment using equally-sized codewords. The four header bits can distinguish 9 different combinations: 28 1-bit integers, 14 2-bit integers, 9 3-bit integers (1 bit unused), 7 4-bit integers, and so on.

Our contributions. In this chapter, we develop a novel *fixed-to-fixed* compression approach, based on a Dictionary of INTeger sequences, that we call DINT. The core idea is that each unit of decoding consumes one b-bit integer codeword, and causes a fixed-length copying operation from an internal codebook of size 2^b – the *dictionary* – to the output buffer. The simplicity of this approach means that decoding is fast; and yet, as we demonstrate with our experiments, DINT provides almost state-of-the-art compression effectiveness. Therefore, the improved combination of efficiency and effectiveness provides an important new reference point in the available spectrum of known trade-off options.

We introduce many refinements to our basic implementation of the mechanism to provide further compression effectiveness, *without* hurting efficiency, namely sequential decoding speed. The devised optimizations include: packed-dictionary arrangement with possible exploitation of strings overlap, optimal block-level encoding and the use of multiple dictionaries to better adapt to the distribution of the encoded integer sequences. We defer the comparison of our technique with the state-of-the-art in Chapter 7.

6.1 Related work

We point the reader to Section 2.3 for general background on inverted indexes and to Section 2.2 for a detailed description of many of the integer encoders devised for efficient and effective index compression.

Here, we review dictionary-based approaches for compression.

Dictionary-based compression. In recent work, Martinez et al. [110] introduce a dictionary-based approach that they call *plurally parsable*. Starting with a probability distribution over an alphabet of symbols, and an assumption of a memoryless source, they construct a set of strings with which to populate a dictionary of some target size, and then use a greedy parsing approach to render any input sequence into a stream of integer dictionary offsets. Their dictionary is allowed to contain prefixes of other entries, and with entries capped at some maximum length.

Table 16a gives an example of a plurally parsable dictionary, assuming an input alphabet of $\{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$ and dictionary width $\ell = 4$ and dictionary size $2^b = 8$. Each entry in the dictionary contains $\ell + 1$ entries, as many as ℓ of which are the corresponding string, and the last one of which is the number of symbols in that string. Using this dictionary, the example string aaabaabcaaaab would be greedily parsed as $\langle \mathsf{aaa}, \mathsf{b}, \mathsf{aa}, \mathsf{b}, \mathsf{c}, \mathsf{aaaa}, \mathsf{b} \rangle$, and coded as the sequence of $b = 3$-bit integers $\langle 5, 1, 4, 1, 2, 7, 1 \rangle$ using a total of 21 bits. Note how all three of the bs, and the c as well, are coded as sequences of length one. In the development below we refer to these instances as being *singletons*. The dictionary does not force the bs to be coded as singletons, but the left-to-right greedy parsing of the input has resulted in that happening. Singletons are relatively costly, because each of them requires a full codeword in the compressed stream.

Martinez et al. [110] use a final $(\ell + 1)$-th column as a way of accelerating decoding. Given a sequence of 3-bit integers $\langle c_i \rangle$, rather than executing a loop that counts the right number of symbols to be written to the output buffer and, in doing so, tests a guard at every iteration, the decoding process copies the full $\ell + 1$ symbols (including the length on the string) from the c_i-th table entry to the output buffer in a single fixed operation, and then increases the output pointer by the amount indicated by the $(\ell + 1)$-th stored value. Because one of the levels of conditionals is eliminated in this iterative cycle, branch mis-predictions are reduced, and higher decoding speeds can be achieved [110].

To build the dictionary Martinez et al. [110] describe a process that tentatively assigns strings to the dictionary based on their zero-order probability of occurrence based on the symbol frequencies, and then iteratively refines those estimates, converging to a set of variable length strings that provide the best coverage. They build a suite of such dictionaries for different initial symbol distributions, and then use them to losslessly code 64×64-pixel blocks of grey-scale image data, with a

Index	String				Size
0	a	-	-	-	1
1	b	-	-	-	1
2	c	-	-	-	1
3	d	-	-	-	1
4	a	a	-	-	2
5	a	a	a	-	3
6	a	b	-	-	2
7	a	a	a	a	4

(a)

Index	String				Size
0	-	-	-	-	1
1	a	-	-	-	1
2	b	-	-	-	1
3	a	a	-	-	2
4	a	b	-	-	2
5	b	a	-	-	2
6	a	a	a	a	4
7	a	a	a	b	4

(b)

Table 16: Two examples of plurally parsable dictionaries of width $\ell = 4$ over the alphabet $\{a, b, c, d\}$. The last column provides the length of each string and is also stored as part of the dictionary. In **(b)**, index zero is a reserved vocabulary entry for a *rare symbol exception*.

matching dictionary selected for each block, and indicated to the decoder via a selector at the start of the block.

Hoobin et al. [79] and Liao et al. [104] have also considered dictionary-based compression options as applied to the text of large document collections; and Zhang et al. [174] have sought to apply that same *relative Lempel Ziv* (RLZ) approach to index data.

6.2 Dictionay-based compression for inverted indexes

In this section, we describe the salient features of our *fixed-to-fixed* approach, namely: (1) how to exploit a dictionary of integer sequences to effectively represent the sequences of docIDs and frequencies and (2), how to attain to fast decompression speed.

6.2.1 Dictionary structure

Our starting point is to consider a *rectangular* dictionary structure, such as the one illustrated in Figure 16, that is a 2-dimensional array of $2^b \times (\ell + 1)$ integers. This simple organization allows random access to any dictionary string in $O(1)$. We now detail its structure.

As we have already introduced and motivated in the previous chapters, there are two factors that make inverted index data highly distinctive. First, very long runs of consecutive docIDs, i.e., runs of 1s (almost always the most frequent symbol in the alphabet) create opportunities for the use of a *frequent symbol exception* (Property 1 in Section 2.3.2), whereby long repetitive sequences are handled outside the normal regime. For example, as we can see from Figure 22, up to 41% of the docID gaps of Gov2 are runs, and handling these economically is a key requirement to attain to good compression effectiveness. Second, the alphabet for docID gaps is very large, into the millions, and it is impossible to consider providing a codeword for every symbol, even as a singleton. Instead, use must be made of *rare symbol exceptions*, a special code that indicates that the next symbol must be fetched from a secondary stream of uncompressed integers.

Figure 22, shows an example of repeated frequent subsequences occurring in a typical extract of 2,048 docID gaps. Each colored rectangle represents a sequence of docID gaps that occurs many times across the index and can be represented as a codeword relative to a dictionary of 65,536 such sequences. Only 3 out of the 2048 docID gaps (or 0.146%) in this fragment are sufficiently rare that they must be coded in full, rather than via the dictionary.

Rare symbol exceptions. To see the use of rare symbol exceptions, consider the dictionary shown in Table 16b, in which only two singleton codes are provided. Using this table the same example string aaabaabcaaaab would be parsed as $\langle$aaab, aa, b, –, c, aaaa, b$\rangle$, and coded as the sequence of integers $\langle 7, 3, 2, 0, \mathrm{c}, 6, 2\rangle$ using a total of $6 \times 3 + 1 \times 2 = 20$ bits, where it is assumed that the rare symbol exception needed for c (that follows a codeword of 0 in the message stream) requires two bits over the alphabet of four symbols. In this small example an overall slight reduction in cost arises, primarily because of the presence of the string aaab in the dictionary. But the general principle is valid: the greater the number of *long* sequences that can be included in the dictionary, the better the compression rate that we can hope to achieve.

The large symbol alphabet used in index compression means that it makes sense to employ multiple exception codes: 0 to indicate that the corresponding patching symbol is a b-bit value between 1 and 2^b; 1 to indicate that the associated patching value is a $2b$-bit value in the range $2^b + 1$ to 2^{2b}; and so on. For example, when $b = 16$, two rare-exception codes cover the space of 32-bit integers; and four rare-symbol exception codewords are employed if $b = 16$ and the input is regarded as being the space of 64-bit integers.

Frequent symbol exceptions. To handle long runs of 1s, further exception codes are added, covering sequences of length $B, B/2, B/4, \ldots, 2\ell$. The first of these covers

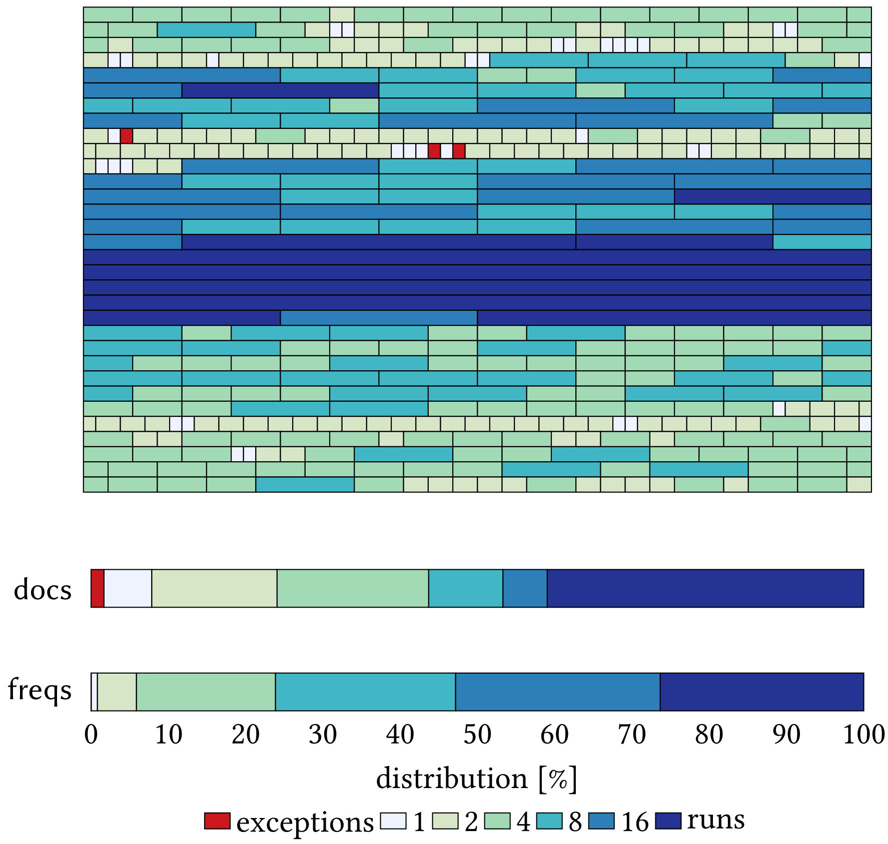


Figure 22: Analysis of a typical sequence of 2048 docID gaps drawn from Gov2. Long runs of 1s are shown in the darkest blue color, the other shades represent frequently-occurring subsequences of length 2, 4, 8 and 16. The light blue squares represent single docID gaps that are frequent, but not as part of longer subsequences and the three red squares are single docID gaps that are rare across the collection, i.e., *exceptions* (not included in the dictionary). Finally, we show the complete distribution for both docs and freqs sequences of Gov2.

an entire block that is all 1s very economically; and short runs of (only) ℓ 1s can be covered by a regular non-exception codeword if required.

For a $b = \ell = 16$ configuration, there will thus be six dictionary slots reserved for exceptions (two rare symbol exception codes, and four frequent symbol exception codes), leaving 65,530 codewords for regular dictionary entries.

Frequency estimation. The set of 2^b sequences making up the dictionary should be tailored to the data being compressed. Intuitively, we would like to store in the dictionary a highly frequent substring of integer, since all the occurrences of such substring will be encoded with just b bits.

To count sub-sequence frequencies, we employ an *interval sampling* approach, examining the source sequence at uniform intervals of $L = 2^k \geq \ell$ and extract-ing samples of each length $\ell' \in \{1, 2, 4, \ldots, \ell\}$ at that point. The frequency of a sequence of length ℓ' is incremented by L/ℓ'. For example, if $\ell' = 2$ and $L = 8$, a two-symbol prefix is extracted every 8 symbols in the input sequence, and that two-symbol combination has its frequency incremented by four. To reduce the counting time L can be made relatively large, e.g., $L = 1024$, and to reduce the space required by the data structure accumulating the counts, a reservoir-based approach can be employed [162, 103]. Both of these techniques produce estimates of the sequence frequencies and not exact counts. But having exact counts would not necessarily be any more useful, since any particular factor parsed from the source sequence might include part or all of other dictionary strings, affecting those counts.

Throughout all the experimental analysis in Section 6.4, exhaustive sampling with $L = \ell'$ is used.

Construction. We now consider an heuristic algorithm for selecting a set of 2^b sequences with which to populate the dictionary. In the description, we indicate with $f(\mathcal{S})$ the number of times the sequence $\mathcal{S}$, of length $|\mathcal{S}|$, has been estimated to occur.

If some sequence $\mathcal{S}$ is selected to enter the dictionary, then it seems likely that both its first half, denoted $\mathcal{S}_1$, of length $|\mathcal{S}|/2$, and its second half, denoted $\mathcal{S}_2$, also of length $|\mathcal{S}|/2$, with $\mathcal{S} = \mathcal{S}_1\mathcal{S}_2$, will already be in the dictionary. This is because (assuming interval sampling) $f(\mathcal{S}_1) \geq f(\mathcal{S})$ and (via symmetry, but not guaranteed) that $f(\mathcal{S}_2) \geq f(\mathcal{S})$. If $\mathcal{S}_1$ and $\mathcal{S}_2$ are in the dictionary, then the saving generated by adding $\mathcal{S}$ to the dictionary is only $f(\mathcal{S})$, since one codeword will be used for each instance of $\mathcal{S}$, rather than two. The same argument can be applied inductively, with the base case arising when singletons are being considered. The true cost of not including them in the dictionary is simply the difference in cost generated by the use of a rare symbol exception.

Hence, the heuristic we consider for populating the dictionary is that of *decreasing static frequency* (DSF), i.e., choosing the sequences with the highest $f(\mathcal{S})$ estimates, regardless of length. As a secondary sort key, to break ties, we sort by decreasing length $|\mathcal{S}|$. Note that this inductive argument is also why we focus on a restricted set of target lengths, i.e., $\{1, 2, \ldots, \ell/2, \ell\}$ with $\ell = 2^k$ for some k, that are powers of two.

We also explored *adaptive* selection heuristics, dynamically updating the $f(\mathcal{S})$ count for sequences that were prefixes and suffixes of longer strings when they were committed to the dictionary, the idea being to maintain more precise frequency estimates for a better sequence selection. Small gains in compression effectiveness were observed in some test situations, but small losses in others; and overall we were unable to consistently outperform the DSF method. Mechanisms

	docs			freqs		
	dictionary	codewords	integers	dictionary	codewords	integers
exceptions	0.00	17.72	1.66	0.00	0.13	0.01
1	21.30	24.13	6.18	1.46	6.32	0.80
2	36.86	31.67	16.22	13.45	19.91	5.05
4	31.03	19.15	19.62	39.07	35.47	17.99
8	8.55	4.69	9.61	34.84	22.98	23.31
16	2.24	1.40	5.74	11.18	13.06	26.49
runs	0.01	1.24	40.97	0.01	2.13	26.35

Table 17: Percentage of integers, codewords, and dictionary entries associated with each target size for the docIDs and freqs sequences of Gov2, parameters $b = 16$, $\ell = 16$.

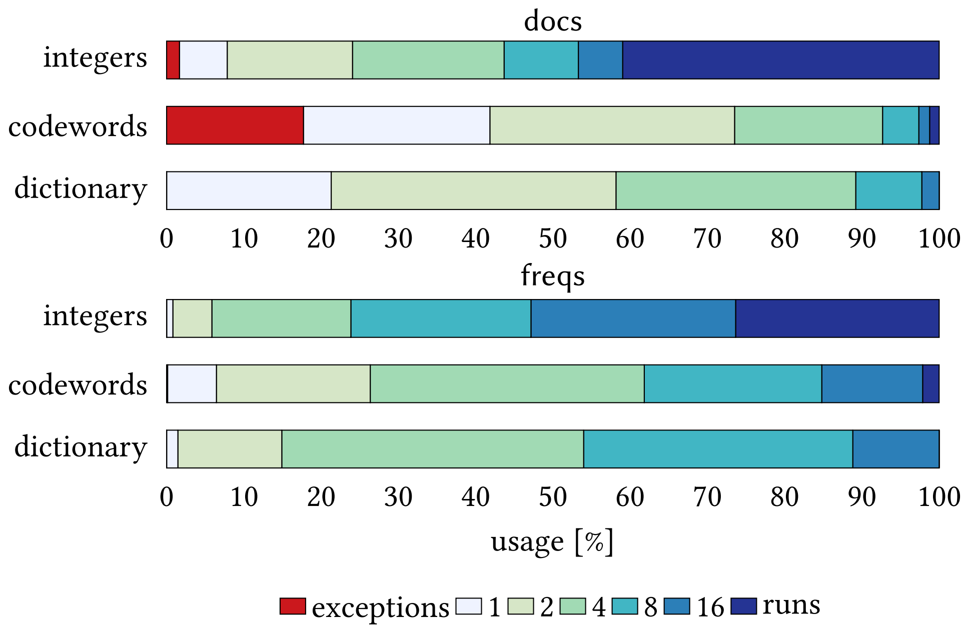


Figure 23: The same data from Table 17 represented as breakdowns.

```
copy(D, c, output)
    begin = c × (ℓ + 1)
    copy 4 × ℓ bytes starting from D[begin] to output
    end = begin + ℓ
    size = D[end]
    return size
```

Figure 24: The **copy** algorithm that performs the copy of a dictionary sequence to the output stream. Note that ℓ is a constant know at compile time and not a variable.

for populating the dictionary will be a target for future work, noting that the problem we face here has parallels in the Re-Pair compression technique [99], which has also been used for index compression [40]; and that Apostolico and Lonardi [11] have also considered the question of identifying useful subsequences.

Performance. Figure 23 provides a summary of the patterns already illustrated in Figure 22, and shows the distribution of target lengths in the raw index, in the compressed index, and in the dictionary respectively. For example, around 18% of the compressed codewords are rare symbol exceptions for less than 1.7% of the docIDs in the actual Gov2 index; whereas up to 41% of the docIDs can be handled by frequent symbol exceptions, consuming just 1.24% of the actual codewords. The freqs dictionary matches are longer than in the docIDs stream, leading to higher compression rates *and* faster decoding speed, as we are going to explain next.

6.2.2 Decoding algorithm

The standard unit of access is a single block of B integers, and we employ $B = 256$ throughout this investigation. That is, each postings list is partitioned into fixed-length blocks, with any remaining elements represented using a secondary mechanism. Fewer than 5% of the postings are coded in this manner. Each block of integers is represented as a set of one or more codewords, each of these being a b-bit binary code.

The action of the decoding algorithm is described in Figure 25: it is assumed that $b = \ell = 16$, with codewords 0 and 1 indicating rare symbol exceptions, and codewords $2, 3, 4, 5$ indicating frequent symbol exceptions, as described in the previous subsection. Small changes might be required if b or ℓ are varied, but note that the rare symbol exception codewords and frequent symbol exception codewords should always be grouped together in the code space, so that a single conditional is sufficient to reach the dominant case, that of a standard codeword referring to a symbol sequence in the dictionary.

```
decode(D, input, output)
    for i = 0; i < B;
        c = get_16bits(input)
        size = 1
        if c > 2
            size = copy(D, c, output)
        else
            e = 0
            if c == 1
                e = get_32bits(input)
            else
                e = get_16bits(input)
            copy e to output
        i = i + size
```

Figure 25: The **decode** algorithm that decodes a block of B integers (assuming $b = 16$).

6.3 Further improvements

In this section, we illustrate several improvements to our initial design presented in Section 6.2. The first two improvements are targeted to enhance the speed of our proposal, whereas the second two to improve its space effectiveness.

6.3.1 Packed dictionary structure

The rectangular dictionary described in Section 6.2 and shown in Table 16 is potentially expensive in terms of space, especially if there are relatively few targets of length ℓ, or if there is significant overlap between prefixes and suffixes of different targets. Therefore, we consider ways of reducing the space required by the dictionary, noting that the smaller the space required, the more likely it is to be retained primarily in cache, thus enhancing the speed of decoding.

To reduce the memory required by the dictionary the packed form shown in Figure 26 can be employed. Now the target lengths are separated from the sequences, and more than one target can indicate the same start position in the single consolidated dictionary string. Doing so both allows the unused trailing symbols to be avoided, and also allows targets that are prefixes of each other to share space. The *length*[] component of each dictionary entry is instead stored as one-byte field within each dictionary offset in the array *start*[], allowing sequences that have the same starting point to be distinguished, an arrangement that is valid provided that

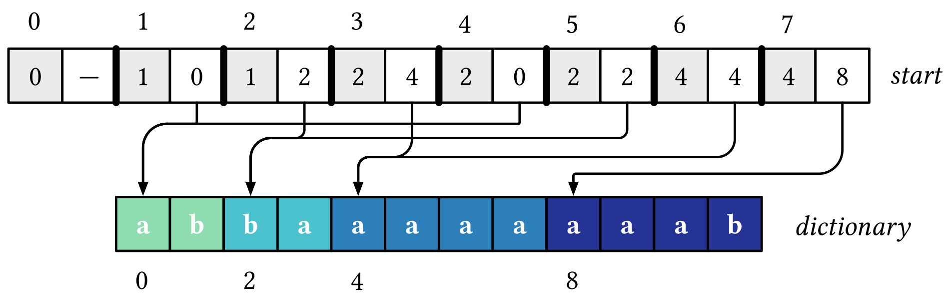


Figure 26: Packed layout for the dictionary shown in Table 16b, with $\ell = 4$ and $b = 3$. The first number in each element in *start*[] is the sequence length. For example, when the input codeword is 3 the four integers in *dictionary*$[4, \ldots, 7]$ are copied to the output, and then the output pointer is incremented by two. All trailing don't-care symbols have been trimmed, and dictionary sequences have been removed if they are a prefix of another longer sequence. For aesthetic purposes, in this simple example no provision has been made for frequent symbol exceptions.

$b \leq 24$ and $\ell < 256$. The indirection via *start*[] means that one additional array dereferencing operation is required in each innermost loop in Algorithm 25, plus a mask/shift sequence to extract the two parts of *start*$[c]$, where c is a codeword, but the net cost is moderate and might be warranted by the space savings. In rectangular form (Table 16), an $\ell = 16$ and $b = 16$ dictionary requires $4 \times 2^{16} \times (16 + 1)$ bytes, that is 4.45 MB; in packed form (see Section 6.4.2, Table 22) that requirement can be reduced to around 1 MB.

6.3.2 Exploiting strings overlap

Further savings are also possible, beyond those offered by prefix matches and trailing don't-cares. For example, with strategic reordering and overlapping, the twelve-symbol *dictionary*[] array in Figure 26 could be further condensed to just six symbols baaaab, since every target listed in Table 16b appears within it as a sub-sequence. The problem of identifying a minimal-length super-sequence in which every one of an original set of supplied sequences occurs as a sub-sequence is NP-hard [68]. But simple greedy approximation algorithms can provide solutions that are within a constant factor of being optimal [22, 92].

The approach we employ here considers the initial set of sequences, determines the longest possible match between a prefix of one sequence and a suffix of another, and replaces the two sequences with their lapped concatenation. Each such step reduces the number of sequences in the set by one, and ensures that a single sequence emerges in which every one of the original sequences is embedded. For

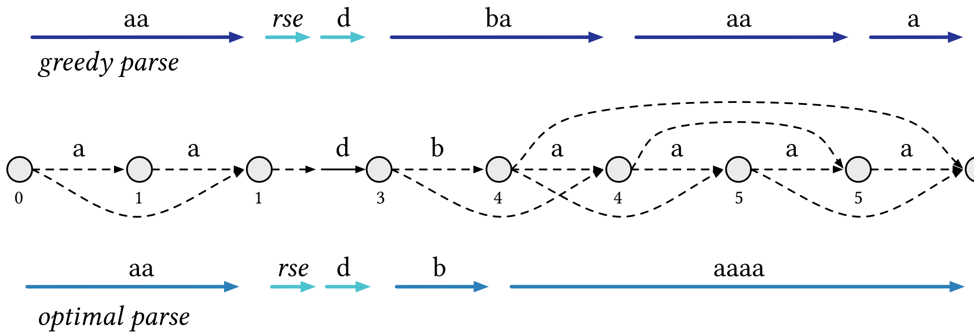


Figure 27: Example in which optimal parsing requires fewer codewords than greedy parsing. The sequence aadbaaaa is being represented relative to the dictionary shown in Table 16b. The cost below each node is the length of the shortest path from the origin to that point. The parse shown at the bottom of the graph has a cost of 5; whereas the greedy parse shown above requires 6 codewords. In both cases it is assumed that d requires a rare symbol exception (*rse*) codeword, followed by a patch codeword that identifies the symbol required.

example, starting with the sequences in Table 16b, the first cycle combines aaaa and aaab to form aaaab.

6.3.3 Optimal block parsing

Dictionary-based compression implementations typically make use of *greedy left-to-right parsing*, with the longest matching dictionary entry employed at each coding step. But in the situation considered here, it is straightforward to identify an *optimal parse* for each block, because each dictionary sequence or frequent symbol exception has a cost of one 16-bit integer, and each rare symbol exception has a fixed cost that depends only on its value.

Figure 27 shows how the prefixes of the block correspond to nodes in an acyclic directed graph, and how the dictionary entries are edges that extend one prefix of the block to a longer one. Within this graph the shortest path from source to sink describes an optimal parse, and can be computed via a left-to-right iterative labeling process that assigns the source with cost zero, and then pushes tentative costs ahead from each node via the edges that emanate from it. The proof of optimality is obvious in this case. The small number of edges that are possible at each node (a maximum of $\log \ell + 1$ if target lengths are restricted to powers of two) makes this process only moderately slower than the more usual greedy approach, with a complexity of $O(n \log \ell)$ that is $\Theta(n)$ when ℓ is a constant.

6.3.4 Multiple dictionaries

Following the example of Moffat and Petri [114], it is also possible for multiple dictionaries to be used. For example, if the input symbols are assumed to be integers between 1 and $2^{32} - 1$, then the use of 32 dictionaries allows each block to be handled within a *context* established by $\lfloor \log \mathit{max} \rfloor$, where *max* is the largest value in the block. Stratifying the blocks according to their maximum value offers clear benefits: blocks in which $\mathit{max} < 4$ are likely to generate quite different dictionaries from those arising when, say, $\mathit{max} < 1024$, even though 1s are likely to still be the most common symbol.

There is, however, a cost. Each additional dictionary must be stored during decoding operations, and both adds to the memory cost, and also adds to the likelihood of cache misses. For this reason, other, less costly, categorizations might also be desirable. In Section 6.4 we make use of the mapping $\mathit{context} = \lceil \log \log \mathit{max} \rceil$ (taking $\log 0 = 0$ when $\mathit{max} = 1$), creating a set of six different contexts $[0, 5]$ with limiting values $2, 4, 16, 256, 65536$, and 2^{32}.

Once the suite of dictionaries has been created, the encoder either uses the same mapping to determine which context to use when encoding each block, or carries out an exhaustive search over all contexts to identify the one that minimizes the compressed size. Either way, each encoded block is prefixed by a *selector* indicating which dictionary to use. In the current DINT implementation the selector is (slightly wastefully) stored as a one-byte integer.

There is a second way in which multiple dictionaries might be employed, and that is through the use of alternative combinations of b and ℓ. For example (see Table 18) it might be beneficial to consider both $b = 16$ and $b = 8$ dictionaries for each context, anticipating (say) that blocks in which *context* is small might be handled more compactly by a 256-element dictionary and the corresponding 8-bit codewords. Again, the selector is used to indicate which context is in use in any particular block. No extra memory space is required by this option, since the $b = 8$ dictionary for any context is an exact prefix of the $b = 16$ dictionary.

As already noted, when multiple contexts are in use memory consumption might become an issue. If so, rather than store each dictionary separately, a set of distinct *start*[] arrays can be used to index a single shared *dictionary*[] array (see Figure 26), with the complete set of contexts' sequences stored overlapped using the heuristic already described.

6.4 Experiments

Datasets. We performed our experiments on the datasets described in Section 1.1, whose statistics are summarized in Table 2 (page 7).

Experimental setting and methodology. All experimentation is based on the ds2i framework[1], with methods implemented using C++14 and compiled with gcc 7.2.0 using the highest optimization setting. The test machine is a server equipped with 512 GB RAM and an Intel Xeon 6144 processor employing 32 KB of L1 cache, 1024 KB of L2 cache, and 25344 KB of L3 cache.

All compression results given in this section are for complete indexes without stopping or other reduction mechanisms being applied, and cover all postings, with sizes given in GB and rates given in bits per integer (bpi). All compression effectiveness results include the overhead of the corresponding dictionaries, accounted as part of the total index size in GB. We use blocks of 256 postings, with trailing elements encoded with BIC as it is standard for all methods in the ds2i framework.

Source code. https://github.com/jermp/dint

	Dictionary width		
	$\ell = 4$	$\ell = 8$	$\ell = 16$
$b = 8$	4.84	4.68	4.65
$b = 16$	5.50	4.68	4.49

(a) Gov2

	Dictionary width		
	$\ell = 4$	$\ell = 8$	$\ell = 16$
$b = 8$	17.16	16.63	16.61
$b = 16$	19.51	16.66	16.09

(b) ClueWeb09

Table 18: Total index size in GB for a complete document-level index (docIDs and freqs combined) for different combinations of bits per codeword b and dictionary width ℓ.

6.4.1 Initial exploration

Choosing alignment. To establish the likely parameter combination for a full implementation, we carried out a preliminary exploration of the parameters b (bits per codeword) and ℓ (maximum length of dictionary entries, in number of integers), using the Gov2 and ClueWeb09 dataset, with measured index sizes in GB. As an initial exploration of the technique, greedy block parsing is adopted for this

[1] https://github.com/ot/ds2i

	Variable-length		Constant-length	
	docs	freqs	docs	freqs
instructions ($\times 10^9$)	53.63	35.02	41.72	28.35
instructions/cycle	1.16	1.13	1.28	1.24
cache-misses ($\times 10^7$)	10.77	9.06	8.21	7.60
branch-misses (%)	3.40	2.79	2.24	0.35
nanoseconds/int	1.82	1.08	1.12	0.73

Table 19: Performance counts reported by the `perf` Linux utility, when decoding the index sequences of Gov2, comparing variable-length copying and constant-length copying for $\ell = 16$.

experiment whose results are shown in Table 18. The trend on both datasets is very similar: all combinations of $b = 8$ provide compression effectiveness levels that are comparable with the ones achieved for $b = 16$, and actually being almost the same for $\ell = 8$.

Baseline compression rates for other methods on these datasets are provided in Section 7.1: here we report, as a single reference point, that a VByte index occupies 12.47 GB for Gov2 and 35.14 GB for ClueWeb09. As can be seen in Table 18, all combinations of b and ℓ are capable of yielding relatively good compression effectiveness once suitable dictionaries have been identified, with the $b = 16$ and $\ell = 16$ combination slightly better than the other arrangements. Therefore, for the rest of the experimental analysis we are going to adopt the configuration $b = \ell = 16$.

Copying fixed-length strings. To further motivate *fixed-length* dictionary-based decoding, Table 19 shows statistics collected using the Linux `perf` utility, when decoding the sequences of the Gov2 dataset. In the left pair of columns, the copying process is executed via a loop that copies the strictly needed number of symbols from the dictionary to the output; in the right pair of columns, a constant ℓ symbols are always copied, with ℓ fixed a compilation time. Copying a fixed number of bytes allows better exploitation of the instruction cache, and leads to a higher instruction throughput (instructions/cycle) with fewer cache- and branch-misses.

Overall, the time taken to extract each decoded integer decreases to around *two-thirds* of what it would be if the copying process was controlled by a loop, a substantial gain in speed.

Decoding speed analysis. Two parameters affect DINT's sequential decoding time: the average number of decoded integers per codeword, say m, and the cost of the copy operation that is associated with each codeword, say c. Decoding

	docs			freqs		
	m	predicted	actual	m	predicted	actual
$\ell = 4$, $c = 2.495$	4.25	0.59	0.82	5.02	0.50	0.50
$\ell = 8$, $c = 4.177$	4.63	0.90	0.91	7.05	0.59	0.56
$\ell = 16$, $c = 5.342$	4.69	1.14	1.12	7.87	0.68	0.73

Table 20: Average number of decoded integers per codeword, m, when decoding the sequences of Gov2, using a rectangular dictionary. The average decoding time per codeword is indicated with c and expressed in nanoseconds per integer. The comparison between predicted and actual sequential decoding time is reported.

more values per codeword increases throughput; on the other hand, copying fewer words during a single decoding operation is faster. This means that using smaller (larger) values of ℓ decreases (increases) the cost of a single copy operation, but also that more (fewer) operations are needed to decode the sequence. To quantify this behaviour, Table 20 reports measured values for c and for m as a function of ℓ, using the docIDs and freqs sequences of Gov2. The predicted decoding time is calculated as c/m nanoseconds/integer. As we can see, the predicted decoding time per integer is a very accurate estimate of the measured one.

6.4.2 Multi-context operation

Space enhancements. From the starting point fixed in Table 18, i.e., the configuration that uses $b = \ell = 16$ and greedy block parsing, we now apply the several refinement introduced in Section 6.3. The impact of these improvements is evaluated in Table 21.

The second row of the table adopts the optimal block parsing. Although a space improvement is always achieved, notice that this is not so high as one could expect. Why? Remember from Section 6.3.3, that the optimal block-parsing algorithm has to consider a small number of edges for each distinct node during the traversal of the DAG illustrated in Figure 27. This number is $\log \ell + 1$, thus for $b = 16$ only 5 edges have to be considered. This causes the number of possible sub-optimal, i.e., "wrong", choices that the greedy parsing can make to be very limited.

The third row shows the additional compression gains that result when a total of six dictionaries are used per stream, conditioned on the largest value *max* in each block via the mapping $context = \lceil \log \log max \rceil$. A one-byte selector is required per block (partially eroding the gain). In the fourth row, we also allow the choice of representing the codewords of the blocks with $b = 8$ bits (still with $\ell = 16$)

	Gov2			ClueWeb09		
	space [GB]	docs [bpi]	freqs [bpi]	space [GB]	docs [bpi]	freqs [bpi]
DINT	4.49	4.21	1.98	16.09	5.97	2.09
+ optimal parsing	4.43	4.16	1.94	15.85	5.90	2.04
+ 6 contexts	4.39	4.10	1.93	15.60	5.76	2.05
+ $b = 8, 16$	4.27	4.04	1.83	15.24	5.67	1.96
+ exhaustive search	4.24	4.01	1.81	15.14	5.64	1.94
+ entropy	3.75	3.47	1.75	13.54	4.96	1.87

Table 21: Total index size in GB and compression rate in bits per integer (bpi) for docs and freqs. The first row uses DINT with greedy parsing; four enhancements are then added. In the last row, the dictionary codewords are assumed to be input to an optimal entropy coder.

		docs		freqs	
		MB	ns/int	MB	ns/int
Single	rectangular	4.456	1.12	4.456	0.73
	packed	1.070	0.88	1.849	0.64
	overlapped	8.722	0.95	1.459	0.69
Multiple	rectangular	22.855	1.66	22.380	1.04
	packed	8.046	1.20	10.121	0.80
	overlapped	6.792	1.33	8.540	0.90

Table 22: Dictionary space in MB for different schemes and corresponding decoding speeds for Gov2.

whenever this provides an advantage in space. The choice between the $b = 8$ and $b = 16$ dictionary is made by test-compressing the block using each. In the fifth row, an exhaustive search using test-compression is made on a per-block basis, slowing decoding time, but not affecting decoding throughput in any way.

We finally report the last row of the table the binary entropy of the written codewords to provide a useful (but unrealistic) lower bound on the space usage of the indexes.

	Gov2				ClueWeb09			
	docs [bpi]	freqs [bpi]	docs [ns/int]	freqs [ns/int]	docs [bpi]	freqs [bpi]	docs [ns/int]	freqs [ns/int]
DINT Time-Opt	4.16	1.94	0.88	0.64	5.90	2.04	1.29	0.78
DINT Space-Opt	4.01	1.81	1.20	0.80	5.64	1.94	1.66	0.94

Table 23: Space usage and sequential decoding time for the two selected DINT configurations.

Dictionary data structures. Table 22 shows, instead, the space usage in MB of the different dictionary data structures described, i.e., rectangular, packed and overlapped, in relation to their decoding speed.

Decoding using a packed dictionary is faster than decoding via a rectangular dictionary because of its more compact memory footprint that lowers the number of cache misses. This is especially true when multiple dictionaries are used. Overlapping the strings to further save space loses the alignment property of the packed arrangement, and slightly increases decoding cost. Use of multiple contexts leads to slightly better compression, but slows decoding throughput because of the increased memory and greater number of cache misses.

Choosing a trade-off configuration. In this concluding paragraph we are interested in selecting two configurations of our technique that we will use for a full comparison again the state-of-the-art in Chapter 7.

The first configuration optimizes space at the expense of decoding speed. In this case, we choose a DINT configuration with multi-packed dictionaries, optimal block parsing, choice of $b = 8$ or $b = 16$ when advantageous and exhaustive search. This space configuration corresponds to the fifth row in Table 21.

The second configuration optimizes decoding speed but sacrifices memory footprint. In this case, we choose instead a DINT configuration with optimal block parsing and a single-packed dictionary.

Space and time requirements for both configurations are finally summarized in Table 23.

7 Comparing Inverted Index Representations

In chapters 4, 5 and 6 we described three different techniques to effectively represent the sorted integer sequences of inverted indexes. The aim of this chapter is the one of providing a comparison between such techniques and the many other competitors described in the background Section 2.2, using the same experimental setting, i.e., testing machine, datasets and methodology.

Inverted index representations. The experiments in this chapter test the effectiveness and efficiency of 14 *different encoders* for inverted indexes. Three of them are the ones described in chapters 4, 5 and 6, i.e., CPEF, Opt-VByte and DINT respectively. We point the reader to the individual chapters for the description of such techniques. The other eleven methods are the ones described in the background Section 2.2, including: VByte without the use of SIMD instructions to accelerate decoding speed and the SIMD-ized variants Varint-GB, Varint-G8IU, Masked-VByte and Stream-VByte; Simple16; QMX; Opt-PFOR; PEF; BIC; ANS.

Datasets. We performed our experiments on the datasets described in Section 1.1, whose statistics are summarized in Table 2 (page 7).

Experimental setting and methodology. Experiments are based on the ds2i framework[1], implemented using C++14 and compiled with gcc 7.2.0 using the highest optimization setting, that is with compilation flags `-march=native` and `-O9`. The test machine is a server equipped with 512 GB RAM and an Intel Xeon 6144 processor with 32 KB of L1 cache, 1024 KB of L2 cache, and 25344 KB of L3 cache.

All compression results are for the complete indexes without stopping or other reduction mechanisms being applied, and cover all postings, with sizes given in GB and rates given in bits per integer (bpi). We use blocks of 256 postings, with trailing elements encoded with BIC as it is standard for all methods in the ds2i framework.

To test the sequential decoding speed of the different proposals, we encoded the docs and freqs sequences in separate streams and decoded all the lists larger than 4096 integers, using a single core. Decoding longer lists favour the decoding throughput of the encoders using SIMD instructions. The reported timings are given in nanoseconds per integer.

To test the query processing speed of the indexes, we memory map the index data structures on disk and compute the boolean conjunctions over a set of queries

[1] `https://github.com/ot/ds2i`

drawn from TREC 05 and TREC 06 Efficiency Track topics. We use a random set of *selective* and *non-selective* queries for each query log, drawn following the methodology of Ottaviano and Venturini [120]. We define as *selective* a query whose ratio between the number of document returned by the conjunction of its terms and the disjunction of its terms is *small*: in our experiments, we used 0.5%. Viceversa, a *non-selective* query is a query whose ratio is *not small*: for this case, we used 10%. We repeat each experiment three times to smooth fluctuations in the measurements and consider the mean value. The query algorithm runs on a single core and timings are reported in milliseconds per query.

Organization. This chapter is organized as follows. In the first three sections we discuss the results of the experiments by showing the related tables and commenting on the *individual trends* for: the space of the indexes (Section 7.1), the sequential decoding speed (Section 7.2) and the query processing speed (Section 7.3). In the conclusive Section 7.4 we finally consider *all the three aspects* of space, decoding and query processing speed as plotted together in trade-off curves, to individuate the techniques that embody the best space/time trade-offs.

7.1 Index space

Table 24 shows the space taken by the different methods, with percentages calculated with respect to the BIC technique (first row), which is traditionally the most compact representation for inverted lists. The two DINT configurations used are the ones identified in Table 23 (see details in Chapter 6). Also, the used CPEF configuration is the one optimizing the space at the expense of retrieval speed (see details in Chapter 4).

The general trend, on both datasets, is that CPEF, BIC, ANS and DINT Space-Opt are the most space-efficient, with DINT Time-Opt and PEF coming second. Among the byte-aligned methods including QMX, Simple16 and Opt-VByte, the latter is the smallest. The other encoders in the Variable-Byte family, i.e., VByte, Varint-GB and Varint-G8IU, are much larger: among them, the simplest VByte stream organization is the most effective for the reasons that we explained in Section 5.3.1. We now discuss some details.

In particular on Gov2, BIC and ANS are the best methods to represent the docs sequences, while DINT Space-Opt is the best to represent the freqs sequences. Notice that this latter characteristic makes DINT Space-Opt to come very close to the overall space of BIC: it is only 3% larger than BIC and actually 14% smaller on the freqs sequences (beating ANS as well), that is a remarkable result. On ClueWeb09 instead, BIC maintains its usual supremacy on the freqs sequences too, in this case DINT Space-Opt is 12% larger.

	space [GB]		docs [bpi]		freqs [bpi]	
BIC	4.13		3.58		2.12	
DINT Space-Opt	4.24	(+3%)	4.01	(+12%)	1.81	(−14%)
DINT Time-Opt	4.43	(+7%)	4.16	(+16%)	1.94	(−9%)
PEF	4.65	(+13%)	4.10	(+15%)	2.38	(+12%)
CPEF	4.43	(+7%)	3.81	(+7%)	2.37	(+12%)
ANS	3.96	(−4%)	3.59	(+1%)	1.92	(−9%)
Opt-PFOR	4.96	(+20%)	4.48	(+25%)	2.38	(+12%)
Simple16	5.73	(+39%)	5.06	(+42%)	2.86	(+35%)
QMX	6.37	(+54%)	5.58	(+56%)	3.23	(+53%)
VByte	12.47	(+202%)	9.28	(+160%)	8.02	(+279%)
Opt-VByte	5.68	(+38%)	4.87	(+36%)	3.04	(+44%)
Varint-GB	15.24	(+269%)	11.12	(+211%)	10.04	(+374%)
Varint-G8IU	13.49	(+227%)	10.02	(+180%)	8.71	(+311%)

(a) Gov2

	space [GB]		docs [bpi]		freqs [bpi]	
BIC	13.55		4.93		1.85	
DINT Space-Opt	15.13	(+12%)	5.64	(+14%)	1.94	(+5%)
DINT Time-Opt	15.85	(+17%)	5.90	(+20%)	2.04	(+10%)
PEF	15.94	(+18%)	5.85	(+18%)	2.20	(+19%)
CPEF	15.33	(+13%)	5.54	(+12%)	2.19	(+18%)
ANS	13.90	(+3%)	5.06	(+3%)	1.95	(+6%)
Opt-PFOR	17.15	(+26%)	6.18	(+25%)	2.41	(+30%)
Simple16	18.68	(+38%)	6.63	(+34%)	2.74	(+48%)
QMX	22.22	(+64%)	7.57	(+53%)	3.58	(+94%)
VByte	35.14	(+159%)	9.66	(+96%)	8.01	(+333%)
Opt-VByte	17.88	(+32%)	6.54	(+32%)	2.48	(+34%)
Varint-GB	42.61	(+214%)	11.40	(+131%)	10.03	(+442%)
Varint-G8IU	38.21	(+182%)	10.43	(+111%)	8.78	(+375%)

(b) ClueWeb09

Table 24: Total index size in GB and compression rate in bits per integer (bpi) for docs and freqs sequences.

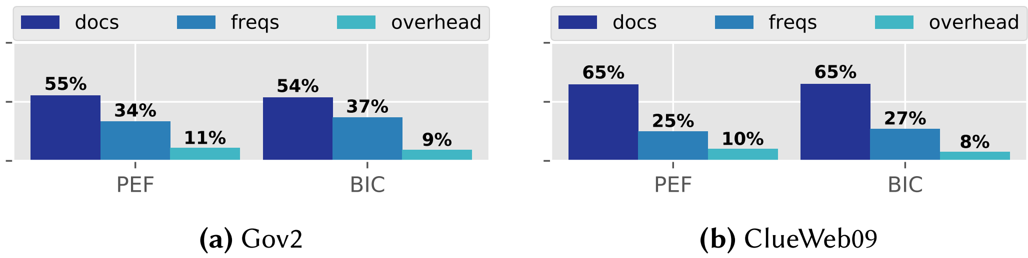


Figure 28: Index space breakdowns in percentages for PEF and BIC representations.

On both datasets, the two DINT configurations are better than PEF especially thanks to their effectiveness in representing the freqs sequences. Notice also that the better compression on the frequencies allows the DINT Space-Opt to win over the CPEF proposal.

The space required by the Opt-VByte technique is comparable (but smaller) with the one of Simple16, with QMX being, instead, considerably larger. The optimization on the space usage of VByte does actually pay off, reducing the gap in compression from BIC from 202% to 38% on Gov2 and from 159% to 32% on ClueWeb09.

As already commented, the other encoders in the Variable-Byte family take roughly 3× the space of the smallest encoders.

Space breakdowns. It is also informative to show space breakdowns of the index layouts in order to better highlight how the three different components of the indexes, i.e., docs, freqs and query-processing overhead constitute the overall space.

We distinguish between encoders (1) working with variable-length partitions, such as PEF, CPEF and Opt-VByte, and (2) encoders working with fixed-length partitions, such as BIC, DINT and the many others mentioned at the beginning of the chapter. For the first class of encoders, the query-processing overhead is due to the cost of the variable-length partitions themselves for which we need to maintain the sizes, upper-bounds and endpoints (also refer to Figure 11 at page 66). For the second class, the overhead is represented by the blocks' upper-bounds and endpoints. In Figure 28 we show the space breakdowns for PEF and BIC, chosen as representatives of the first and second class respectively.

By looking at the plots, we see that the space breakdowns for the two indexes are very similar. In particular, observe that the query-processing overhead is around 10%, with variable-length partitions costing a little more. As expected and discussed, the most expensive components are the sequences of docs whereas the ones of freqs feature more regularities that favour compression: the ratio between the space of docs and freqs is 1.5 ÷ 1.6 for Gov2 and 2.4 ÷ 2.6 for ClueWeb09.

	Gov2				ClueWeb09			
	docs		freqs		docs		freqs	
VByte	1.08		0.76		1.22		0.69	
DINT Time-Opt	0.64	(−40%)	0.51	(−33%)	1.00	(−18%)	0.62	(−10%)
DINT Space-Opt	0.93	(−14%)	0.70	(−8%)	1.44	(+18%)	0.83	(+21%)
PEF	2.37	(+120%)	2.43	(+221%)	3.02	(+148%)	2.46	(+259%)
CPEF	4.22	(+291%)	3.67	(+385%)	5.37	(+341%)	3.83	(+460%)
Opt-PFOR	1.21	(+12%)	0.89	(+17%)	1.75	(+43%)	1.09	(+59%)
BIC	7.25	(+573%)	7.05	(+832%)	8.81	(+623%)	8.01	(+1069%)
ANS	5.72	(+431%)	6.01	(+695%)	8.11	(+565%)	7.44	(+987%)
QMX	0.87	(−20%)	0.86	(+14%)	1.31	(+7%)	1.22	(+78%)
Simple16	1.12	(+4%)	0.93	(+23%)	1.52	(+24%)	1.02	(+49%)
Opt-VByte	0.59	(−45%)	0.42	(−44%)	0.85	(−30%)	0.35	(−49%)
Varint-GB	0.57	(−47%)	0.52	(−32%)	0.67	(−45%)	0.49	(−28%)
Varint-G8IU	0.45	(−58%)	0.38	(−49%)	0.52	(−57%)	0.43	(−37%)
Masked-VByte	0.47	(−57%)	0.41	(−46%)	0.56	(−54%)	0.42	(−38%)
Stream-VByte	0.42	(−61%)	0.40	(−47%)	0.47	(−62%)	0.42	(−39%)

Table 25: Sequential decoding throughput in nanoseconds per integer by scanning all the docs and freqs sequences larger than 4096 postings.

7.2 Decoding speed

Table 25 reports the sequential decoding speed of the various methods, with percentages calculated with respect to the VByte method that does not use SIMD instructions. We express the percentages with respect to this method because it is considered to be the fastest in the literature (as already discussed in chapters 2 and 5), and we can also evaluate the impact of the SIMD instructions.

The general trend is that, not surprisingly, the VByte family is the fastest *when* the SIMD instructions are used: an integer can be decoded in roughly half of a nanosecond (and sometimes in even less than that, e.g., on the freqs sequences), representing a ≈2× improvement when such instructions are not used. We now discuss some details.

The DINT Time-Opt and Opt-VByte methods approach the highest speed. Notice that, however, we did *not* add any SIMD instructions to the the DINT's decoding algorithm but the adopted fixed-copy approach makes it very fast. It also outperforms the QMX mechanism that uses them. Moreover, the DINT's decod-

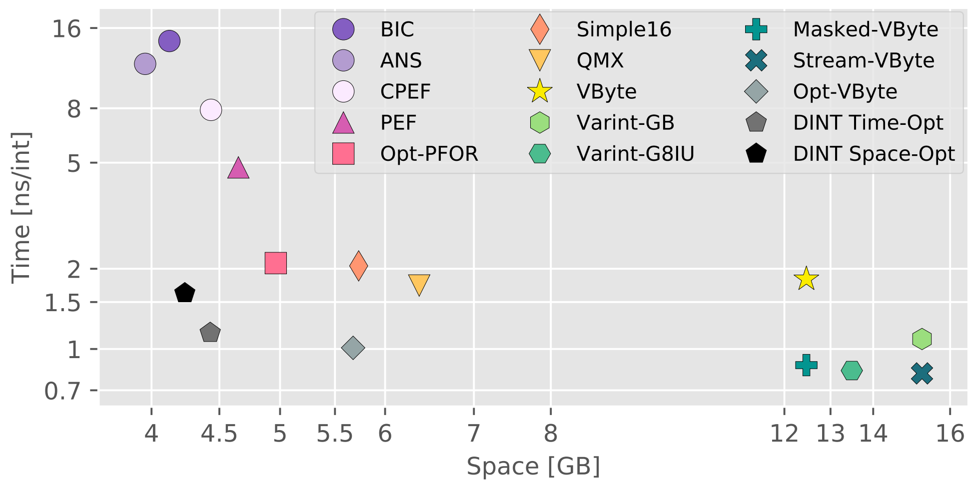


Figure 29: Effectiveness/efficiency plot for Gov2 concerning sequential decoding speed. The horizontal scale shows the total index size in GB, whereas the vertical scales sums the per-posting docs and freqs nanoseconds per integer. Both scales are logarithmic.

ing speed for the freqs sequences is practically as fast as the one of the reported SIMD-ized approaches (see the relevant Section 6.4.1, Table 20, for the details).

Instead, the encoders decoding an integer in the range 1 ÷ 1.5 nanoseconds include: VByte, QMX, Simple16, Opt-PFOR and DINT Space-Opt. Notice that also the DINT Space-Opt version is competitive in speed, being actually faster than QMX (on the frequencies) and Simple16. PEF and BIC are the slowest in raw sequential decoding, with PEF being roughly 3× faster than BIC. Although the current implementation of CPEF and ANS is not (yet) optimized for raw sequential decoding speed, we report their timings for completeness: ANS codes perform similarly to BIC; the CPEF mechanism is faster than ANS but still roughly twice slower than PEF due to the additional accesses to the lists of reference.

Figure 29 plots the timings shown in Table 25 for the Gov2 dataset in relation to the space taken by each method.

7.3 Query speed

Table 26 and 27 report the average time for AND queries in milliseconds, respectively for selective and non-selective queries drawn from the TREC 05 and TREC 06 logs. Figure 30 and 31 plot the timings shown in such tables for the Gov2 dataset in relation to the space taken by each method. In what follows, we discuss the two tables separately.

Selective queries. On selective queries, PEF is the fastest method thanks to the powerful skipping capabilities of Elias-Fano when performing the operation NextGEQ (see Section 2.2.6), reaching (or even beating) the speed of the fastest SIMD-ized decoders. A similar consideration applies to the Opt-VByte technique for the reasons explained in details in Section 5.3.2: in particular, as we can see from the plot illustrated in Figure 20b, for all the jump sizes less than 32, VByte is 2× faster than Elias-Fano, while this advantage vanishes for the longer jumps thanks to the mentioned skipping abilities of Elias-Fano. However, we know that this advantage is shrunk because jumps larger than 32 are not very frequent on the tested query logs, as depicted by the distribution of Figure 21.

Competitive approaches, but still slightly slower in this case, are QMX, VByte, Simple16 and Opt-PFOR. Also the DINT Time-Opt approach is fast, being 22% slower than PEF on average.

Lastly, the most space-efficient methods, i.e., CPEF, ANS and BIC, are considerably slower than PEF. As already reported in Section 4.3.2, although the CPEF mechanism imposes a penalty of roughly 50% over PEF, it is still more than 2× and 3× faster than ANS and BIC respectively.

Non-selective queries. Differently from the selective case, the trend changes considering the non-selective queries. In this case, the methods that are faster at raw sequential decoding perform better, e.g., the ones in the VByte family.

Notice, in fact, how the DINT mechanism approaches the speed of the fastest methods (also confirming the trend shown in Table 25) and being on average faster than PEF by 30%. Opt-VByte has a similar behaviour to the one of PEF (and slightly faster on ClueWeb09) because the sparse blocks of the lists are sequentially decoded with the SIMD-ized Masked-VByte that is faster than Elias-Fano for such task (see details in Section 5.3).

Again, CPEF, ANS and BIC still remain the slowest methods, with CPEF and ANS being comparable to each other and BIC much slower.

	Gov2				ClueWeb09			
	TREC 05		TREC 06		TREC 05		TREC 06	
PEF	1.10		2.42		6.67		10.10	
DINT Time-Opt	1.29	(+17%)	2.90	(+20%)	8.30	(+24%)	12.70	(+26%)
DINT Space-Opt	1.56	(+42%)	3.57	(+48%)	9.39	(+41%)	14.15	(+40%)
CPEF	2.00	(+82%)	4.30	(+78%)	12.10	(+81%)	18.40	(+82%)
Opt-PFOR	1.65	(+50%)	3.80	(+57%)	10.40	(+56%)	15.20	(+50%)
BIC	6.80	(+518%)	15.20	(+528%)	40.00	(+500%)	60.00	(+494%)
ANS	5.00	(+355%)	11.00	(+355%)	29.00	(+335%)	41.00	(+306%)
QMX	1.25	(+14%)	2.60	(+7%)	7.20	(+8%)	11.00	(+9%)
Simple16	1.70	(+55%)	3.80	(+57%)	10.60	(+59%)	15.90	(+57%)
VByte	1.40	(+27%)	3.00	(+24%)	8.60	(+29%)	12.80	(+27%)
Opt-VByte	0.96	(−13%)	2.17	(−10%)	7.00	(+5%)	9.10	(−10%)
Varint-GB	1.13	(+3%)	2.60	(+7%)	7.30	(+9%)	11.00	(+9%)
Varint-G8IU	1.10	(+0%)	2.46	(+2%)	6.70	(+0%)	10.00	(−1%)
Masked-VByte	1.12	(+2%)	2.56	(+6%)	6.90	(+3%)	10.30	(+2%)
Stream-VByte	1.17	(+6%)	2.80	(+16%)	6.70	(+0%)	10.40	(+3%)

Table 26: Timings in milliseconds per query for selective AND queries using the TREC 05 and TREC 06 query logs.

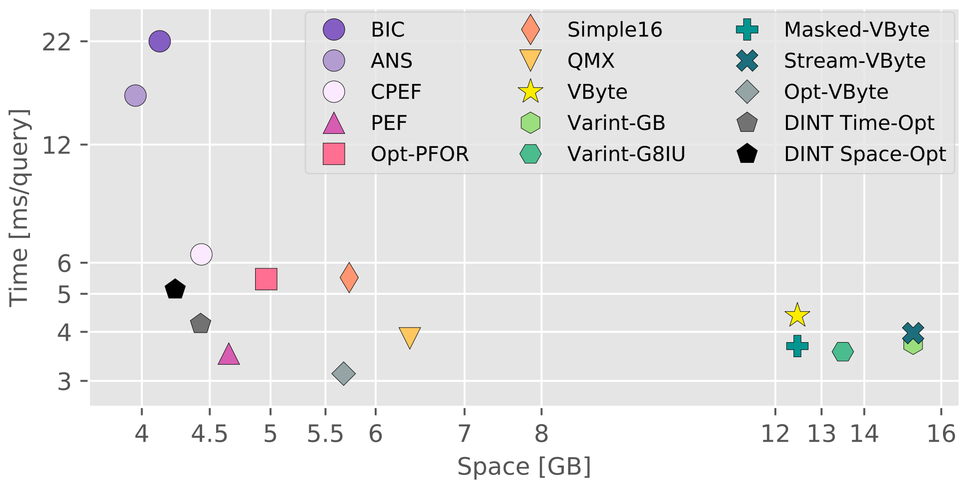


Figure 30: Effectiveness-efficiency plots for Gov2, concerning selective query processing. The horizontal scale shows the total index size in GB, whereas the vertical scales sums the timings on TREC 05 and TREC 06. Both scales are logarithmic.

	Gov2				ClueWeb09			
	TREC 05		TREC 06		TREC 05		TREC 06	
PEF	4.88		8.76		17.10		21.20	
DINT Time-Opt	3.30	(−32%)	6.40	(−27%)	12.10	(−29%)	15.50	(−27%)
DINT Space-Opt	3.81	(−22%)	7.41	(−15%)	13.31	(−22%)	17.04	(−20%)
CPEF	7.00	(+43%)	15.30	(+75%)	31.00	(+81%)	38.40	(+81%)
Opt-PFOR	4.80	(−2%)	9.40	(+7%)	17.20	(+1%)	22.00	(+4%)
BIC	12.00	(+146%)	23.00	(+163%)	42.00	(+146%)	53.00	(+150%)
ANS	7.90	(+62%)	15.20	(+74%)	31.00	(+81%)	39.30	(+85%)
QMX	3.28	(−33%)	6.50	(−26%)	11.40	(−33%)	14.80	(−30%)
Simple16	4.10	(−16%)	8.00	(−9%)	15.46	(−10%)	18.50	(−13%)
VByte	3.60	(−26%)	7.10	(−19%)	12.36	(−28%)	16.00	(−25%)
Opt-VByte	4.89	(+0%)	8.70	(−1%)	15.80	(−8%)	19.00	(−10%)
Varint-GB	3.27	(−33%)	6.43	(−27%)	11.25	(−34%)	14.55	(−31%)
Varint-G8IU	3.22	(−34%)	6.40	(−27%)	11.04	(−35%)	14.20	(−33%)
Masked-VByte	3.28	(−33%)	6.57	(−25%)	11.16	(−35%)	14.35	(−32%)
Stream-VByte	3.24	(−34%)	6.37	(−27%)	11.05	(−35%)	14.40	(−32%)

Table 27: Timings in milliseconds per query for non-selective AND queries using the TREC 05 and TREC 06 query logs.

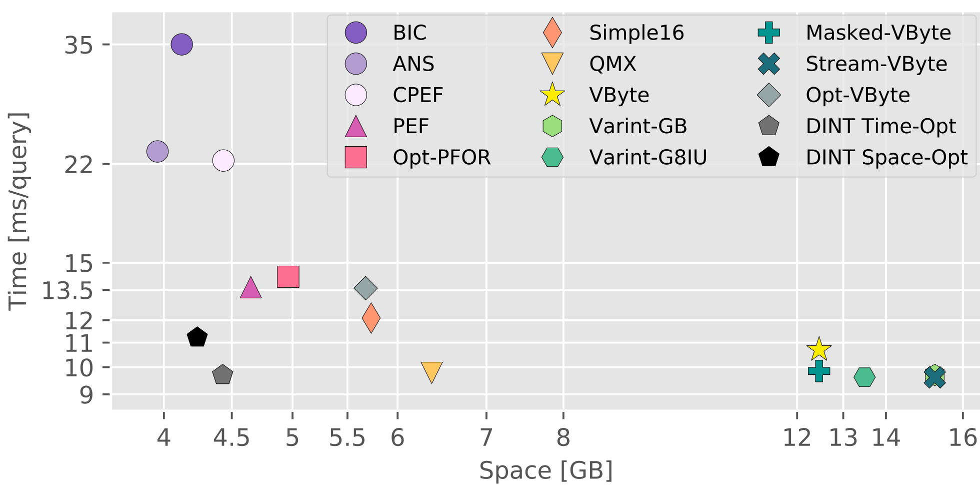


Figure 31: Effectiveness-efficiency plots for Gov2, concerning non-selective query processing. The horizontal scale shows the total index size in GB, whereas the vertical scales sums the timings on TREC 05 and TREC 06. Both scales are logarithmic.

7.4 Conclusions

We now consider all the three aspects of space, sequential decoding speed and query processing speed together by commenting on the trade-off curves plotted in figures 29, 30 and 31. The curves for the ClueWeb09 dataset are similar to the ones illustrated for Gov2.

Concerning sequential decoding (refer to Figure 29), Opt-VByte is almost as fast as the fastest VByte representations but also takes almost one third of the space; in turn, the DINT Time-Opt configuration is smaller than Opt-VByte (by 30% on Gov2, but by 15% on ClueWeb09) and decodes only slightly slower. Opt-PFOR, Simple16, QMX and VByte are all very similar in speed but are outperformed by the aforementioned methods because they take more space and are slower.

Regarding selective queries (refer to Figure 30): DINT, PEF and Opt-VByte are the ones performing best. Again, Opt-PFOR, Simple16 and QMX are outperformed by these methods because are larger and slower. In this case, also the VByte family does not pay off since their speed is not better (or only marginally better) than the leading methods and their space much larger, by $2 \div 3$ times. Also notice that the CPEF mechanism is dominated by both DINT configurations that are even smaller and faster. For this reason, DINT dominates over BIC and ANS too because it comes very close to the space of BIC (only 3% away on Gov2 and 12% away on ClueWeb09) but is much faster.

Concerning non-selective queries (refer to Figure 31) : DINT dominates here too since it is as fast as the fastest SIMD-ized decoders in the VByte family and almost as small as the smallest BIC, CPEF and ANS (that are all better than PEF regarding the space). PEF, Opt-PFOR, Simple16 and Opt-VByte are all similar in speed, with PEF being the smallest among them. The QMX mechanism is comparable in speed with DINT Time-Opt, but its space usage is much larger.

In conclusion, Opt-PFOR, Simple16 and QMX are always outperformed by other mechanisms. The same conclusion applies for SIMD-ized decoders in the VByte family that can still be the fastest but actually not as fast as one could expect compared to other methods that take 1/3 of their space and do *not* use SIMD instructions. PEF and Opt-VByte are similar, with the former being smaller but the latter being faster. DINT Time-Opt uses the same space of PEF (actually, slightly less) and it is faster in performing sequential decoding by $3\times$ and on non-selective queries by 30%. Only on selective queries it is slower, by only 22% on average. The other space-efficient methods BIC, CPEF and ANS are also dominated by the DINT technique that is much faster and not significantly larger.

For these reasons, DINT, PEF and Opt-VByte embody the best time/space trade-offs. In particular, as apparent from the trade-off plots, the DINT method almost bridges the gap between space effectiveness and efficiency, by combining a small memory footprint with the highest speed in the literature.

8 Compressed Indexes for N-Gram Strings

In Section 2.4, we discussed the importance of n-gram strings in several crucial applications in Information Retrieval and Machine Learning and noted that at the core of all such applications lies an *efficient* data structure mapping n-grams to their associated satellite data, e.g., a frequency count representing the number of occurrences of the n-gram or probability/backoff weights for word-predicting computations [76, 122]. The efficiency of the data structure should come both in time *and* space, because modern string search and machine translation systems make very frequent queries over databases containing several billion n-grams that often do not fit in internal memory [90, 43]. To reduce the memory-access rate and, *hence*, speed up the execution of the retrieval algorithms, the design of an efficient *compressed* representation of the data structure appears as mandatory. While several solutions have been proposed for the indexing and retrieval of n-grams, either based on tries or hashing (see the background Section 2.4.2), 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 [76]. In fact, space reductions are usually achieved by either: lossy quantization of satellite values, or by randomized approaches with false positive allowed [154]. The most space-efficient and lossless proposals still employ binary search over the compressed representation to lookup for a n-gram: this results in a severe inefficiency during query processing because of the lack of a compression strategy with a fast random access operation [122]. To support random access, current methods leverage on *block-wise compression* with expensive decompression of a block every time an element of the block has to be retrieved. Finally, hashing schemes based on open addressing with linear probing result extremely large for static corpora as long as the tables are allocated with $30-50\%$ extra space to allow fast random access [76, 122].

Since a solution that is compact, fast and lossless at the same time is still missing, the first aim of this chapter is that of addressing the aforementioned inefficiencies by introducing compressed data structures that, despite their small memory footprint, support efficient random access to the satellite n-gram values. This is the problem of *indexing* n-gram datasets that we introduced in Section 2.4.

Given a n-gram string, the compressed data structure should allow fast random access to the corresponding associated value, being either a frequency count (integer) or a probability (floating point) by means of the operation Lookup.

Our contributions. We list here the main contributions of this chapter.

(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 (see Section 2.2.6) as to efficiently support random access operations and successor queries over the compressed sequence. We refer to such data structure as the *Elias-Fano trie*.

We also 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, i.e., its preceding k words, is encoded as an integer whose value is proportional to the number of words that follow such context.

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

8.1 Related work

Section 2.4 offers a general overview about N-gram strings, their applications and the two basic indexing techniques: tries and hash tables. Here, we describe how these two fundamental data structures have been adopted in the literature.

State-of-the-art. Pauls and Klein [122] proposed trie-based data structures in which the nodes are represented via sorted arrays or with hash tables with linear probing. The trie sorted arrays are compressed using a variable-length block encoding: a configurable radix $r = 2^k$ is chosen and the number of digits d to represents a number in base r is written in unary. The representation then terminates with the d digits, each of which requires exactly k bits. To preserve the property of looking up a record by binary search, each sorted array is divided into blocks of 128 bytes. The encoding is used to compress words, pointers and the positions that frequency counts take in a unique-value array that collect all distinct counts. The hash-based variant is likely to be faster than the sorted array variant, but requires extra table allocation space to avoid excessive collisions.

Heafield [76] 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 $O(\log \log m)$ with high probability by using *interpolation* search [50]. Records in each sorted arrays are minimally sized at the bit level, improving the memory consumption over [122]. Pointers are compressed using the integer compressor devised in [137]. Values can also be quantized using the *binning* method [59] 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 [154] use Bloom filters [21] with lossy quantization of frequency counts to achieve small memory footprint. In particular, the raw frequency count f_g of gram g is quantized using a logarithmic codebook, i.e., $\widetilde{f_g} = 1 + \log_b f_g$. The scale is determined by the base b of the logarithm: in the implementation b is set to $2^{1/v}$, where v is the quantization range used by the model, e.g., $v = 8$. Given the quantized count $\widetilde{f_g}$ of gram g, a Bloom filter is trained by entering composite events into the filter, represented by g with an appended integer value j, which is incremented from 1 to $\widetilde{f_g}$. Then at query time, to retrieve $\widetilde{f_g}$, the filter is queried with a 1 appended to g. This event is hashed using the k hash functions of the filter: if all of them test positive, then the count is incremented and the process repeated. The procedure terminates as soon as any of the k hash functions hits a 0 and the previous count is reported. This procedure avoids a space requirement for the counts proportional to the number of grams in the corpus because only the codebook needs to be stored. The one-sided error of the filter and the training scheme ensure that the actual quantized count cannot be larger than the reported value. As the counts are quantized using a logarithmic-scaled codebook, the count will be incremented only a small number of times. The quantized logarithmic count is finally converted back to a linear count.

The use of the succinct encoding Level-Order Unary-Degree Sequence [87], in short LOUDS, is advocated in [164] to implicitly represent the trie nodes. In particular, the pointers for a trie of m nodes are encoded using a bitvector of $2m + 1$

bits. Bit-level searches on such bitvector allow forward/backward navigation of the trie structure. Words and frequency counts are compressed using Variable-Byte encoding [155, 143], with an additional bitvector used to indicate the boundaries of such byte sequences as to support random access to each element. The paper also discusses the use of block-wise compression (basically gzip on blocks of 8 KB) though it is not used in the implementation for time efficiency reasons. Shareghi et al. [146, 147] also consider the usage of succinct data structures to represent *suffix trees* that can be used to compute Kneser-Ney probabilities on-the-fly. Experimental results indicate that the method is practical for large-scale language modeling although significantly slower to query than leading toolkits for language modeling [76].

Because of the importance of strings as one of the most common computerized kind of information, the problem of representing trie-based storage for string dictionaries is among one of the most studied in computer science, with many and different solutions available [78, 117, 39]. It goes without saying that, given the properties that n-gram datasets exhibit, generic trie implementations are *not* suitable for their efficient handling. However, comparing with the performance of such implementations gives useful insights about the performance gap with respect to a general solution. We mention Marisa [172] as the best and practical general-purpose trie implementation. The core idea is to use Patricia tries [116] 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.

8.2 Elias-Fano tries

In this subsection we present a compressed trie data structure, based on the Elias-Fano representation of monotone integer sequences for its efficient random access and search operations. We point the reader to Section 2.2.6 for the description of this elegant integer encoding. As we will see, the constant-time random access of Elias-Fano makes it the right choice for the encoding of the sorted-array trie levels, given that we fundamentally need to randomly access the sub-array pointed to by a pair of pointers. Such pair is retrieved in constant time too. Now every access performed by binary search takes $O(1)$ *without* requiring any block decompression, differently from currently employed strategies [122].

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

8.2.1 Data structure

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

As it is standard, a unique integer ID is assigned to each distinct word to form the vocabulary of the indexed n-gram corpus. Such vocabulary is implemented using a hash data structure that stores for each uni-gram its ID in order to retrieve it when needed in $O(1)$. If we sort the n-grams following the token-ID order, we have that all the successors of gram $w_1^{n-1} = w_1 \cdots w_{n-1}$, i.e., all grams whose prefix is w_1^{n-1}, form a strictly increasing integer sequence. For example, suppose we have the uni-grams[1] {A, B, C, D}, which are assigned IDs $\{0, 1, 2, 3\}$ respectively. Now consider the bi-grams {AA, AC, BB, BC, BD, CA, CD, DB, DD} sorted by IDs. The sequence of the successors of A, referred to as the *range* of A, is ⟨A, C⟩, i.e., $\langle 0, 2 \rangle$; the sequence of the successors of B, is ⟨B, C, D⟩, i.e., $\langle 1, 2, 3 \rangle$ and so on. Figure 32 shows a graphical representation of what we described. Concatenating the ranges, we obtain the integer sequence $\langle 0, 2 | 1, 2, 3 | 0, 3 | 1, 3 \rangle$ that we can see in Figure 32b (the vertical bars, depicted in dark blue in Figure 32b, are not really 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). The ID assigned to a uni-gram is also used as the position at which we read the uni-gram pointer in the uni-grams pointer sequence.

Therefore, apart from uni-grams that are stored in a hash table, each level of the trie is composed by *two* integer sequences: one for the representation of the gram-IDs, the other for the pointers.

We have therefore reduced the problem of representing a trie to the problem of compressing (a few) integer sequences. Among the many integer compressors available in the literature (see Section 2.2), we choose Elias-Fano, along with its partitioned variant, PEF [120] (Sections 2.2.6 and 2.2.7 respectively).

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 32b. 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

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

prefix sum (initially equal to 0). Then, our example sequence $\langle 0, 2|1, 2, 3|0, 3|1, 3\rangle$ becomes $\langle 0, 2|3, 4, 5|5, 8|9, 11\rangle$.

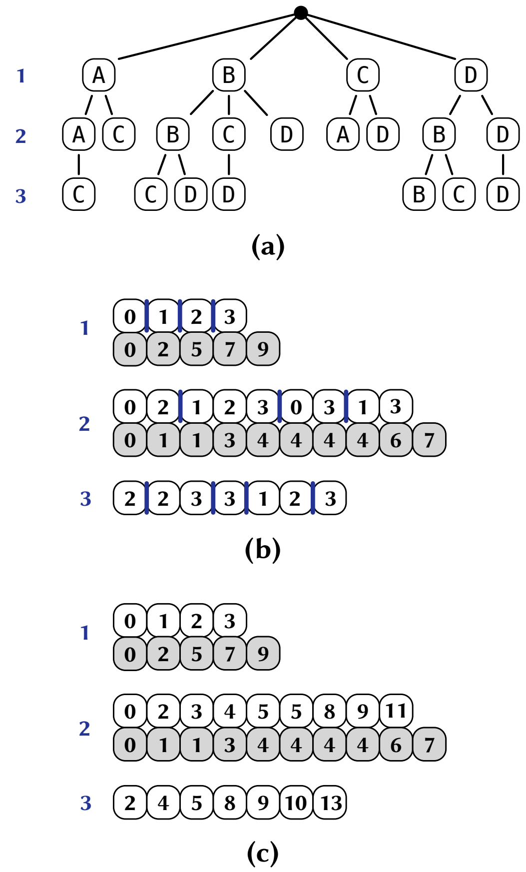


Figure 32: In **(a)** we show an example of a trie of order 3, representing the set of grams {A, AA, AAC, AC, B, BB, BBC, BBD, BC, BCD, BD, C, CA, CD, D, DB, DBB, DBC, DDD}. In **(b)** we see the sorted-array representation of the trie, where each vocabulary token is assigned a distinct integer ID. Lastly, in **(c)**, we show the final representation of the trie where each sorted array has been transformed in a monotone sequence by computing the prefix sums of the ranges marked with the dark blue bars in **(b)**. The shaded arrays represent the pointers.

The last prefix sum is initially 0, therefore the range of A remains the same, i.e., $\langle 0, 2\rangle$. Now the last prefix sum is 2, so we sum 2 to the values in the range of B, yielding $\langle 3, 4, 5\rangle$. Now the last prefix sum is 5, so 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 32c.

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

Frequency counts. To represent the frequency counts, we use the *unique-value array* technique, i.e., each count is represented by its *index* in an array $C[n]$, $1 \le n \le N$, that collects all the *distinct* frequency counts for the n-grams. The *unique-value array* 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 of them appear only a few times. As we can better see in Table 28, the number of distinct counts is very small compared to the number of n-grams themselves, so the space for the arrays $C[n]$, $1 \leq n \leq N$, is negligible.

Now, each level of the trie, besides the sequences of gram-IDs and pointers, has also to store the sequence made by all the frequency-count indexes. Unfortunately, this sequence of indexes is not monotone, yet it follows the aforementioned highly repetitive distribution. Therefore, 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 with just a few bits. More specifically, starting from $k = 1$, we first assign all the 2^k codewords of length k before increasing k by 1 and repeating the process until all 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 [67], to the i-th 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. From codeword c and its length ℓ_c in bits, we can retrieve i by taking the inverse of the previous formula, i.e., $i = c - 2 + 2^{\ell_c}$. Besides the bitvector for the codewords themselves, we also need to know where each codeword begins and ends. We can use another bitvector for this purpose, say L, that stores a 1 for the starting position of every codeword. A small additional data structure built on L allows efficient computation of 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 i-th codeword. Its length is easily computed by scanning L upward from position b until we hit the next 1, say in position e. Finally, $\ell_c = e - b$ and $c = B[b, e - 1]$.

In conclusion, a trie is 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 supported, i.e., how to retrieve the frequency count given an n-gram w_1^n. The corresponding pseudo code is illustrated in Figure 33. We first perform n vocabulary lookups to map the gram tokens into its constituent IDs. We write these IDs into an array $ids[1, n]$ (lines 2-4 in Figure 33a). This preliminary query-mapping step takes $\Theta(n)$. Now, the search procedure has to locate $ids[i]$ in the i-th level of the trie (lines 3-6 in Figure 33b), as follows.

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

(a)

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

(b)

Figure 33: 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.

If $n = 1$, then our search terminates: at the position $p = ids[1]$ we read the rank $i = levels[1].indexes[p]$ 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 Elias-Fano-encoded sequence is performed in constant time. Now p is updated to be the position in $levels[2].ids$ at which $ids[2]$ 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.

8.2.2 Context-based identifier remapping

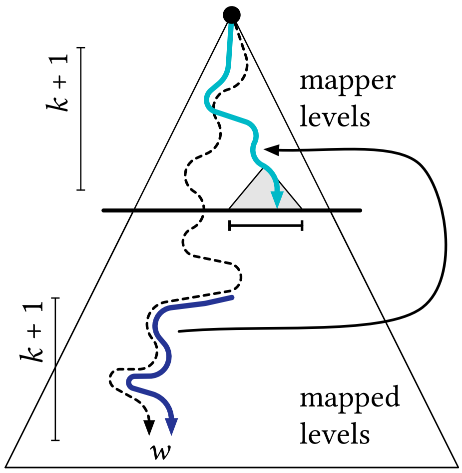
In this subsection we describe a novel technique that lowers the space occupancy of the gram-ID sequences that constitute, as we have seen, the main component of the trie data structure.

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

Figure 34a 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 to the light green path in order to search w along its sibling IDs that are the ones occurring after the gram w_1^k (the small dark gray triangle in the figure). We stress that this projection is *always possible*, i.e., we are guaranteed to find any sub-path of length $k+1$ in the first $k+1$ levels of the trie, because of the sliding-window extraction process described in Section 2.4.1. Figure 34a also highlights that using a context of length k will partition the levels of the trie into two categories: the so-called *mapper* levels and the *mapped* levels. The first $k+1$ levels of trie act, in fact, as a mapper structure whose role is to map any word ID through searches; all the other $N-k-1$ levels are the ones formed by the remapped IDs.

The salient feature of our strategy is that it takes full advantage of the n-gram model represented by the trie structure itself in that it does *not* need any redundancy to perform the mapping of IDs, because these are mapped by means of searches in the first $k+1$ levels of the trie. The strategy also allows a great deal of flexibility in that we can choose the length k of the context. In general, with an n-gram dataset of order $N \geq 2$, we can choose between $N-2$ distinct context lengths k, i.e., $1 \leq k \leq N-2$. Clearly, the greater the context length we use, the smaller the remapped IDs will be but the 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 motivate *why* the introduced remapping strategy offers a valuable contribution to the overall space reduction of the trie data structure, throughout some didactic and real examples. As we will see in the experimental Section 8.4, the dataset vocabulary can contain several million tokens, whereas



(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)

Figure 34: In **(a)**, we depict the action performed by the context-based identifier remapping strategy. The last word ID w of any sub-path of length $k+1$, e.g., the dark blue one, is replaced with the position it takes within its sibling IDs. These sibling IDs are found at the end (spanned by the gray triangle) of the search of w along the *same* path, e.g., the light green one, in the first $k+1$ levels of the trie. In **(b)**, we show the effect of the context-based remapping on the average gap (ratio between universe and size) of the gram-ID sequences of the datasets used in the experiments, with context length $k = 0, 1, 2$.

the number of words that naturally occur after another is typically very small. Even in the case of stopwords, such as “the” or “are”, the number of words that can follow is far less than the whole number of distinct words for *any* n-gram dataset. This ultimately means that the remapped integers forming the gram-ID

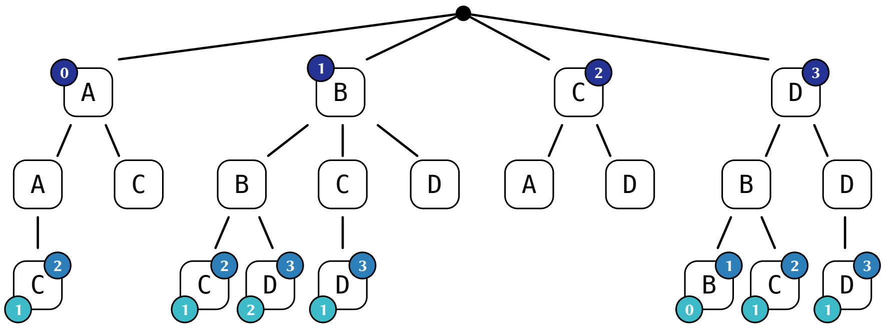


Figure 35: Example of a trie of order 3, representing the set of grams {A, AA, AAC, AC, B, BB, BBC, BBD, BC, BCD, BD, CA, CD, DB, DBB, DBC, DDD}. Vocabulary IDs are represented in darkest blue while level-3 IDs in light blue. The light green IDs are derived by applying a context-based remapping with context length 1.

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 [115, 120]. In our case the gram-ID sequences are encoded with Elias-Fano: from Section 8.2.1, we know that Elias-Fano spends $\lceil \log \frac{u}{m} \rceil + 2$ bits per integer, thus a number of bits proportional to the average gap u/m between its values. The remapping strategy reduces the universe u of representation, thus lowering the average gap and space of the sequence.

This effect is illustrated by the numbers in Figure 34b that shows how the average gap of the gram-ID sequences of the datasets we used in the experiments (see also Table 28) is affected by the context-based remapping. As uni-grams and bi-grams constitute the mapper levels, these are kept unmapped: we show the statistics for the mapped levels, i.e., the third, fourth and fifth, of a trie of order 5 built from the n-grams of the datasets. For each dataset we did the experiment for context lengths 0, 1 and 2. As we can see by considering Europarl, our technique with a context of length 1 achieves an average reduction of 7.2 times (up to 11.3 on tri-grams). With a context of length 2, instead, we obtain an average reduction of 43.4 times (up to 58 on 4-grams). Very similar considerations and numbers hold for the YahooV2 dataset as well. The reduction on the GoogleV2 dataset is less dramatic instead, being on average of 3 times with a context of length 1 and of 16.75 times with a context of length 2.

Example. To better understand how the remapping algorithm works, we consider now a small didactic example. We continue with the example from Section 8.2.1 and represented in Figure 35. The blue IDs are the vocabulary IDs and the red ones are the last token IDs of the tri-grams as assigned by the vocabulary. We now

```
lookup(w_1^n, k)
    ids[1, n] = [0, 0]
    remapped_ids[1, n] = [0, 0]
    for i = 1; i ≤ n; i = i + 1
        id = vocab.lookup(w_i)
        ids[i] = remapped_ids[i] = id
    for i = k + 1; i ≤ n; i = i + 1
        remapped_ids[i] = search(ids, i − k, i, true)
    p = search(remapped_ids, 1, n, false)
    i = levels[n].indexes[p]
    return C[n][i]
```

Figure 36: The **lookup** function with context-based remapping of order k.

explain how the remapped IDs, represented in 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, i.e., C (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 35 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, i.e., w_1^4, we can choose to map w_4 as the position 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).

Lookup. The described remapping strategy comes with an overhead at query time because the lookup algorithm illustrated in Figure 33 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 35, before searching the mapped ID 1 of D for the tri-gram BCD, we have to map the vocabulary ID of D, i.e., 3, to 1. For this task, we search 3 within the successors of C. As 3 is found in position 1, we now know that we have to search for 1 within the successors of BC.

On the one hand, the context-based remapping will assign smaller IDs as the length of the context rises, on the other hand it will also spend more time at query processing. Therefore, we have a space/time trade-off that we explore with an extensive experimental analysis in Section 8.4. The pseudo code for the Lookup operations with context-based remapping is illustrated in Figure 36. Note that, in comparison with the pseudo code in Figure 33a, the remapping technique uses an array to store the re-mapped IDs (line 3) and an additional `for` loop (lines 7-8).

8.3 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 [76, 122]: 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 [17] and available at `https://github.com/ot/emphf`. This implementation requires 2.61 bits per key on average.

At the hash location for an n-gram we store: its 8-byte hash key as to have a false positive probability of 2^{-64} (4-byte hash keys are supported as well) and the position of the frequency count in the unique-value array $C[n]$ which keeps all distinct frequency counts for order n. As already motivated, these unique-value arrays, one for each different order of n, are negligibly small compared to the number of n-grams themselves and act as a direct map from the position of the count to its value. Although these unique values could be sorted and compressed, we do not perform any space optimization as these are too few to yield any improvement but we store them uncompressed and byte-aligned, in order to favour lookup time. 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 $levels[n]$, then we access the count index i stored at position p and finally retrieve the count value $C[n][i]$.

8.4 Experiments

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

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

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
gzip	6.98		6.45		6.20	

Table 28: Number of n-grams and distinct frequency counts for the datasets used in the experiments. We also report the average bytes per gram achieved by gzip as a useful baseline for comparison.

- Europarl consists in all unpruned n-grams extracted from the English Europarl parallel corpus [95], available at: `http://www.statmt.org/europarl`.
- YahooV2 is a collection of English n-grams with minimum frequency count equal to 2, extracted from a corpus of 14.6 million documents crawled from more than 12,000 sites during 2006 [4]. The dataset is available at: `http://webscope.sandbox.yahoo.com/catalog.php?datatype=l`.
- GoogleV2 is the latest English version of Web1T [24], 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 28 shows the basic statistics of the datasets. We choose these datasets in order to test our data structures on different corpora sizes: starting from the left of Table 28 each dataset has roughly 10 times the number of n-grams of the previous one.

Compared indexes. We compare the performance of our data structures against the following software packages that use the approaches introduced in Section 8.1.

- BerkeleyLM implements two trie data structures based on sorted arrays and hash tables to represent the nodes of the trie [122]. The code is written in Java and available at: `https://github.com/adampauls/berkeleylm`.

- Expgram makes use of the LOUDS succinct encoding [87] to implicitly represent the trie structure, while the frequency counts are compressed using VByte encoding [164]. The code is written in C++ and available at: `https://github.com/tarowatanabe/expgram`.

- KenLM implements a trie with interpolation search and a hashing with linear probing [76]. The code is written in C++ and available at: `http://kheafield.com/code/kenlm`.

- Marisa is a general-purposes string dictionary implementation in which Patricia tries are recursively used to represent the nodes of a Patricia trie [172]. The code is written in C++ and available at: `https://github.com/s-yata/marisa-trie`.

- RandLM employs Bloom filters with lossy quantization of frequency counts to attain to low memory footprint [154]. The code is written in C++ and available at: `https://sourceforge.net/projects/randlm`.

Experimental setting and methodology. All experiments have been performed on a machine with 16 Intel Xeon E5-2630 v3 cores (32 threads) clocked at 2.4 Ghz, with 193 GB of RAM, running Linux 3.13.0, 64 bits. Our implementation is in standard C++11 and compiled with gcc 5.4.1 with the highest optimization settings. Template specialization has been preferred over inheritance to avoid the virtual method call overhead, which can be disruptive for the very fine-grained operations we consider. Except for the instructions to count the number of bits set in a word (popcount), and to find the position of the least significant bit (number of trailing zeroes), no special processor feature was used. In particular, we did not add any SIMD instruction to our code.

The data structures were saved to disk after construction, and loaded into main memory to be queried. For the scanning of input files we used the posix_madvice system, called with the parameter POSIX_MADV_SEQUENTIAL to instruct the kernel to optimize the sequential access to the mapped memory region.

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

Source code. `https://github.com/jermp/tongrams`

8.4.1 Elias-Fano tries

In this subsection we test the efficiency of our trie data structure. As already done for the description in Section 8.2.1, we dedicate one paragraph to the validation of each of the main building components of the trie, as well as to the introduced performance optimizations.

		bytes/gram		µs/query	
	EF	1.97		1.28	
	PEF	1.87	(−5%)	1.35	(+6%)
$k = 1$	EF	1.67	(−15%)	1.58	(+24%)
$k = 1$	PEF	1.53	(−22%)	1.61	(+26%)
$k = 2$	EF	1.46	(−26%)	1.60	(+25%)
$k = 2$	PEF	1.28	(−35%)	1.64	(+28%)

(a) Europarl

		bytes/gram		µs/query	
	EF	2.17		1.60	
	PEF	1.91	(−12%)	1.73	(+8%)
$k = 1$	EF	1.89	(−13%)	2.05	(+28%)
$k = 1$	PEF	1.63	(−25%)	2.16	(+35%)
$k = 2$	EF	1.68	(−22%)	2.08	(+30%)
$k = 2$	PEF	1.38	(−36%)	2.15	(+35%)

(b) YahooV2

		bytes/gram		µs/query	
	EF	2.13		2.09	
	PEF	1.52	(−29%)	1.91	(−9%)
$k = 1$	EF	1.91	(−10%)	3.03	(+45%)
$k = 1$	PEF	1.31	(−39%)	2.30	(+10%)

(c) GoogleV2

Table 29: Average bytes per gram (bytes/gram) and average Lookup time per query in microseconds (µs/query).

Gram-ID sequences. Table 29 shows the average number of bytes per gram including the cost of pointers, and lookup speed per query. The first two rows refers to the trie data structure described in Section 8.2.1, when the sorted arrays are encoded with Elias-Fano (EF) and partitioned Elias-Fano (PEF) [120]. Subsequent rows indicate the space gains obtained by applying the context-based remapping strategy using EF and PEF for contexts of lengths respectively 1 and 2. For GoogleV2 we use a context of length 1, as the tri-grams alone roughly constitute 66% of the whole the dataset, thus it would make little sense to optimize only the space of 4- and 5-grams that take 27% of the dataset.

As expected, partitioning the gram sequences using PEF yields a better space occupancy. Though the paper by Ottaviano and Venturini [120] describes a dynamic programming algorithm that finds the partitioning able of minimizing the space occupancy of a monotone sequence, we instead adopt a *uniform* partitioning strategy. Partitioning the sequence uniformly has several advantages over variable-length partitions for our setting. As we have seen

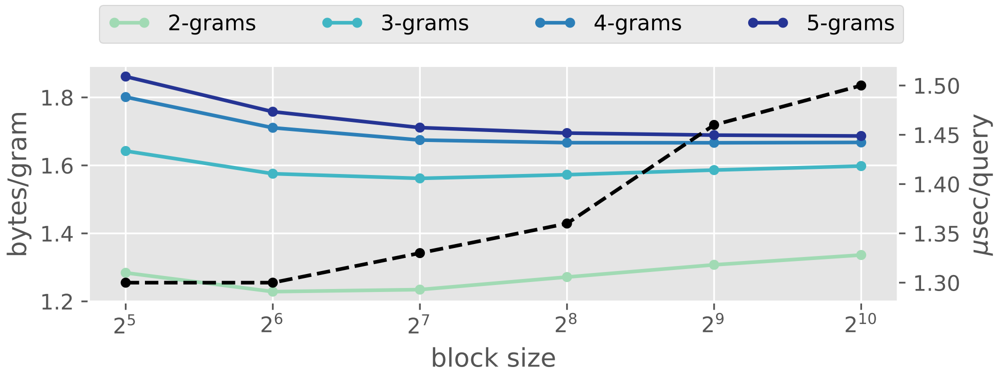


Figure 37: Bytes per gram (left vertical axis) and μs per query (right vertical axis, black dashed line) by varying block size in PEF uniform on the gram-ID sequences of Europarl.

in Section 8.2.1, trie searches are carried out by performing a preliminary random access to the endpoints of the range pointed to by a pointer pair. Then a search in the range follows to determine the position of the gram-ID.

Partitioning the sequence by variable-length blocks introduces an additional search over the sequence of partition endpoints to determine the proper block in which the search must continue. While this preliminary search only introduces a minor overhead in query processing for inverted index queries [120] (as it has to be performed once and successive accesses are only directed to forward positions of the sequence), it is instead the major bottleneck when random access operations are very frequent as in our case. By resorting on uniform partitions, we eliminate this first search and the cost of representation for the variable-length sizes. To speed up queries even further, we also keep the upper bounds of the blocks bit-packed in the minimum number of bits.

As the problem of deciding the optimal block size is posed, Figure 37 shows the space/time trade-off obtained by varying the block size on the gram-ID sequences. The plots for YahooV2 and GoogleV2 datasets exhibit the same shape, therefore we report the one for Europarl. The dashed black line illustrates how the average Lookup time varies when *all* the gram-ID sequences are partitioned using the same block size. The figure suggests to use partitions of 64 integers for bi-gram sequences, and of 128 for all other orders, i.e., for $N \geq 3$, given that the space usage remains low without increasing much the query processing speed. With this choice of block sizes, the loss in space with respect to PEF 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 37. However, excessively reducing the block size may ruin the

	Europarl		YahooV2		GoogleV2	
Variable-len. codewords	0.36		0.47		1.46	
Prefix sums + EF	0.35	(−2%)	0.62	(+32%)	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%)	—	

Table 30: Average bytes per count for different techniques.

advantage in space reduction. Therefore it is convenient to use small block sizes for the most traversed sequences, e.g., the bi-gram sequences, that indeed must be searched several times during the query-mapping phase when the context-based remapping is adopted. In conclusion, as we can see by the second row of Table 29, there is *no* practical difference between the query processing speed of EF and PEF: this latter sequence organization brings a negligible overhead in query processing speed (less than 8% on Europarl and YahooV2), while maintaining a noticeable space reduction (up to 29% on GoogleV2).

Context-based identifier remapping. Concerning the efficacy of the context-based remapping, we have that remapping the gram IDs with a context of length $k = 1$ is already able of reducing the space of the sequences by ≈13% on average when sequences are encoded with Elias-Fano, with respect to the EF cost. If we consider a context of length $k = 2$ we *double* the gain, allowing for more than 28% of space reduction *without* affecting the lookup time with respect to the case $k = 1$. As a first conclusion, when space efficiency is the main concern, it is always convenient to apply the remapping strategy with a context of length 2. The gain of the strategy is even more evident with PEF: this is no surprise as the encoder can better exploit the reduced IDs by encoding all the integers belonging to a block with a universe relative to the block and not to the whole sequence. This results in a space reduction of more than 36% on average and up to 39% on GoogleV2.

Regarding the query processing speed, as explained in Section 8.2.2, the remapping strategy comes with a penalty at query time as we have to map an ID before it can be searched in the proper gram sequence. On average, by looking at Table 29, we found that 30% more time is spent with respect to the Elias-Fano baseline. Notice that PEF does *not* introduce any time degradation with respect to EF with context-based remapping: it is actually faster on GoogleV2.

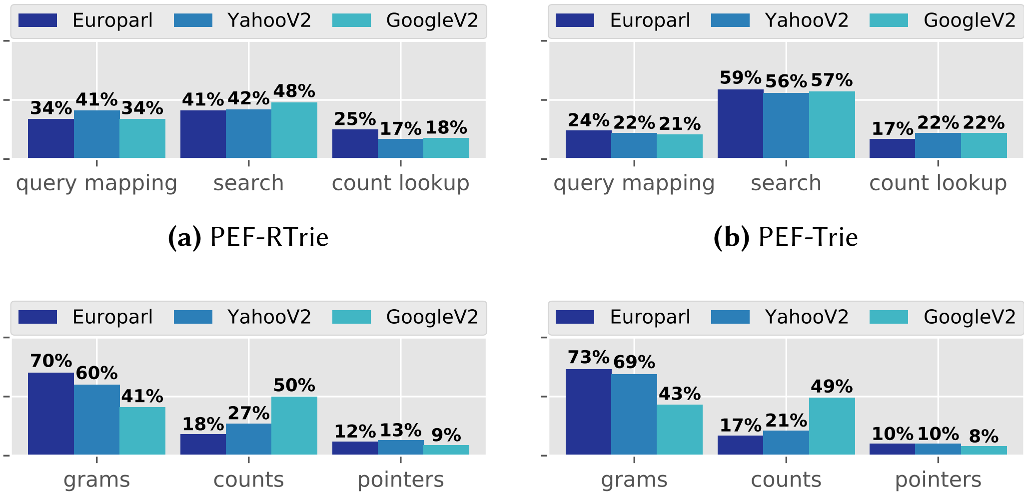
Frequency counts. For the representation of frequency counts we compare three different encoding schemes: the first one refers to the strategy described in Section 8.2.1 that assigns variable-length codewords to the ranks of the counts and keeps track of codewords length using a binary vector (Variable-len. codewords); the other two schemes transform the sequence of count ranks into a non-decreasing sequence by taking its prefix sums and then applies EF or PEF (Prefix sums + EF/PEF).

Table 30 shows the average number of bytes per count for these different strategies. The reported space also includes the space for the storage of the arrays containing the distinct counts for each order of N. As already pointed out, these take a negligible amount of space because the distribution of frequency counts is highly repetitive (see Table 28). The percentages of Prefix sums + EF/PEF are done with respect to the first row of the table, i.e., Variable-len. codewords.

The time for retrieving a count was pretty much the same for all the three techniques. Prefix-summing the sequence and apply EF does not bring any advantage over the codeword assignment technique because its space is practically the same on Europarl but it is actually larger on both YahooV2 (by up to 32%) and GoogleV2. These two reasons together place the codeword assignment technique in net advantage over EF. PEF, instead, offers a better space occupancy of more than 16% on Europarl and 10% on GoogleV2. Therefore, in the following we assume this representation for frequency counts, except for YahooV2, where we adopt Variable-len. codewords.

We also report the space occupancy for the counts representation of BerkeleyLM and Expgram which, differently from all other competitors, can also be used to index frequency counts. BerkeleyLM COMPRESSED variant uses the Variable-len. block-coding mechanism explained in Section 8.1 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 28). Expgram, instead, does not store count ranks but directly compress the counts themselves using Variable-Byte encoding (VByte) with an additional binary vector as to be able of randomly accessing the counts sequence. The available RAM of our test machine (193 GBs) was not sufficient to successfully build Expgram on GoogleV2. The same holds for KenLM and Marisa, as we are going to see next. Therefore, we report its space for Europarl and YahooV2.

We first observe that rank-encoding schemes are far more advantageous than compressing the counts themselves, as done by Expgram. Moreover, none of the these techniques beats the three ones we previously introduced, except for the BerkeleyLM COMPRESSED variant which is ≈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.



(a) PEF-RTrie **(b)** PEF-Trie

(c) PEF-RTrie **(d)** PEF-Trie

Figure 38: 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.

Time and space breakdowns. Before concluding the subsection, we use the analysis to fix two different trie data structures that respectively privilege space efficiency and query time: we call them PEF-RTrie (the R stands for *remapped*) and PEF-Trie. For the PEF-RTrie variant we use PEF for representing the gram-ID sequences; Prefix sums + PEF for the counts on Europarl and GoogleV2 but Variable-len. codewords for YahooV2. We also use the maximum applicable context length for the context-based remapping technique, i.e., 2 for Europarl and YahooV2; 1 for GoogleV2. For the PEF-Trie variant we choose a data structure using PEF for representing gram-ID sequences and Variable-len. codewords for the counts, *without* remapping.

The corresponding size breakdowns are shown in Figures 38c and 38d respectively. The sequences for pointers take very little space for both data structures (approximately 10.3%), while most of the difference lies, not surprisingly, in the space of the gram-ID sequences (roughly 70% for Europarl and YahooV2; 40% for GoogleV2). The timing breakdowns in Figures 38a and 38b clearly highlight, instead, how the context-based remapping technique *rises* the time we spend in the query-mapping phase, during which the IDs are mapped to their reduced IDs. In such case, the two phases of query mapping and search are almost the same, while in the PEF-Trie the search phase dominates.

8.4.2 Hashing

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

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

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 31). In particular, all the time is spent in computing the hash function itself and access the relative table location: the final count lookup is completely negligible.

8.4.3 Overall comparison

In this subsection we compare the performance of our selected trie-based solutions, i.e., the PEF-RTrie and PEF-Trie, as well as our minimal perfect hash approach against the competitors introduced at the beginning of this subsection. The results of the comparison are shown in Table 31, 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 only store probabilities and backoffs, whereas BerkeleyLM can be used to store either counts or probabilities. For those competitors storing frequency counts, we already discussed their count representation in Section 8.4.1. Expgram, KenLM and Marisa require too much memory for the building of their data structures on GoogleV2, therefore we mark as empty their entry in the table for this dataset.

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

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

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

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

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

RandLM is designed for small memory footprint and returns approximated frequency counts when queried. We build its data structures using the default setting recommended in the documentation: 8 bits for frequency count quantization and 8 bits per value as to yield a false positive rate of $\frac{1}{256}$. While being from 2.3 to 5 times slower than our exact and lossless approach, it is quite compact because the quantized frequency counts are recomputed on the fly using the procedure described in Section 8.1. Therefore, while its space occupancy results even larger with respect to our grams representation by 61%, it is still no better than the whole space of our PEF-RTrie data structure. With respect to the whole 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).

	Europarl		YahooV2		GoogleV2	
	bytes/gram	μs/query	bytes/gram	μs/query	bytes/gram	μs/query
PEF-Trie	1.87	1.35	1.91	1.73	1.52	1.91
PEF-RTrie	1.28	1.64	1.38	2.15	1.31	2.30
BerkeleyLM C.	1.70 (−9%) (+33%)	2.83 (+109%) (+73%)	1.69 (−11%) (+22%)	3.48 (+102%) (+62%)	1.45 (−5%) (+11%)	4.13 (+117%) (+80%)
BerkeleyLM H.3	6.70 (+259%) (+423%)	0.97 (−28%) (−41%)	7.82 (+310%) (+465%)	1.13 (−34%) (−47%)	9.24 (+508%) (+608%)	2.18 (+14%) (−5%)
BerkeleyLM H.50	7.96 (+326%) (+521%)	0.97 (−28%) (−41%)	9.37 (+391%) (+578%)	0.96 (−44%) (−55%)	—	—
Expgram	2.06 (+10%) (+61%)	2.80 (+107%) (+71%)	2.24 (+17%) (+62%)	9.23 (+435%) (+329%)	—	—
KenLM T.	2.99 (+60%) (+134%)	1.28 (−5%) (−22%)	3.44 (+80%) (+149%)	1.94 (+12%) (−10%)	—	—
Marisa	3.61 (+93%) (+182%)	2.06 (+52%) (+26%)	3.81 (+100%) (+175%)	3.24 (+88%) (+51%)	—	—
RandLM	1.81 (−3%) (+41%)	4.39 (+224%) (+168%)	2.02 (+6%) (+46%)	5.08 (+194%) (+136%)	2.60 (+71%) (+99%)	9.25 (+385%) (+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%)	—	—

Table 31: Average bytes per gram (bytes/gram) and average Lookup time per query in microseconds per query (μs/query). For our data structures, i.e., PEF-Trie and PEF-RTrie, the bytes/gram cost also includes the space of representation for the pointer sequences.

The last two rows of Table 31 regard the performance of our MPH table with respect to KenLM PROBING. As similarly done for BerkeleyLM H., we also test the PROBING data structure with 3% (P.3) and 50% (P.50) extra space allocation factor for the tables. While being larger as expected, the KenLM implementation makes use of expensive hash key recombinations that yields a slower random access capability with respect to our minimal perfect hashing approach.

We finally compare the *total* space occupancy, as given by the sum of the space of gram-ID sequences, frequency counts and pointers, of our trie data structures against the gzip baseline reported in Table 28. The total average bytes per represented n-gram for PEF-Trie are 2.17, 2.38 and 2.82 on the three datasets Europarl, YahooV2 and GoogleV2 respectively. Table 28 shows that gzip takes, instead, 6.98, 6.45 and 6.2 bytes per gram. This means that our PEF-Trie is 3.2×, 2.7× and 2.2× smaller than gzip and it does also support efficient search of individual n-grams. Finally, our PEF-RTrie is 4.4×, 3.5×, 2.4× smaller.

Perplexity benchmark. Besides the efficient indexing of frequency counts, our data structures can also be used to map n-grams to language model probabilities and backoffs. As done by KenLM, we also use the *binning* method [59] to quantize probabilities and backoffs, but allowing any quantization bits ranging from 2 to 32. Uni-grams values are stored unquantized to favor query speed: as vocabulary size is typically very small compared to the number of total n-grams, this has a minimal impact on the space of the data structure. Our trie implementation is *reversed* as to permit a more efficient computation of sentence-level probabilities, with a *stateful* scoring function that carries its state on from a query to the next, as similarly done by KenLM and BerkeleyLM.

For the perplexity benchmark we used the standard query dataset publicly available at `http://www.statmt.org/lm-benchmark`, that contains 306,688 sentences, for a total of 7,790,011 tokens [33]. We used the utilities of Expgram to build modified Kneser-Ney [35, 36] 5-gram language models from the counts of Europarl and YahooV2 that have an OOV (out of vocabulary) rate of, respectively, 16% and 1.82% on the test query file. As Expgram only builds quantized models using 8 quantization bits for both probabilities and backoffs, we also use this number of quantization bits for our tries and KenLM trie. For all data structures, BerkeleyLM truncates the mantissa of floating-point values to 24 bits and then stores indices to distinct probabilities and backoffs. RandLM was build, as already said, with the default parameters recommended in the documentation.

Table 32 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

	Europarl				YahooV2			
	bytes/gram		μs/query		bytes/gram		μs/query	
PEF-Trie	3.48		0.25		3.64		0.38	
PEF-RTrie	2.91		0.28		3.06		0.43	
BerkeleyLM C.	6.50	(+87%)	1.19	(+372%)	6.39	(+76%)	1.08	(+187%)
		(+123%)		(+322%)		(+109%)		(+152%)
BerkeleyLM H.3	9.36	(+169%)	0.84	(+234%)	8.75	(+140%)	0.74	(+96%)
		(+222%)		(+199%)		(+186%)		(+72%)
BerkeleyLM 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%)
		(+57%)		(−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%)

Table 32: Perplexity benchmark results reporting average number of bytes per gram (bytes/gram) and microseconds per query (μs/query) using modified Kneser-Ney 5-gram language models built from Europarl and YahooV2 counts.

all other competitors in both space and time, including the BerkeleyLM H.3 variant. In particular, notice that we are also smaller than RandLM which is randomized and, therefore, less accurate. The query time of BerkeleyLM H.50 is smaller on YahooV2; however, it also uses from 3 up to 4 times the space of our tries.

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

9 Language Models Estimation

The problem we tackle in this chapter is the one of computing the probability distribution of the n-grams extracted from large textual collections. We refer to this problem as the one of *estimation*, that is the second problem that we introduced in Section 2.4. This problem is clearly related to the one of *indexing* n-gram strings that we study in Chapter 8 because, after estimation, the computed language model needs to be queried efficiently.

In other words, we would like to create and efficient, compressed, index that maps the n-grams of a large text to its probability of occurrence in the text. The way such probability is computed depends on the chosen model. This is an old problem and has received a lot of attention: not surprisingly, several models have been proposed in the literature, such as Laplace, Good-Turing, Katz, Jelinek-Mercer, Witten-Bell and Kneser-Ney (see the background Section 2.4.1).

Among the many, *Kneser-Ney* language models [94] and, in particular, their *modified* version introduced by Chen and Goodman [36], have gained popularity thanks to their relatively low-perplexity performance. This makes modified Kneser-Ney the de-facto choice for language model toolkits. The following software libraries, widely used in both academia and industry (e.g., Google [23, 34] and Facebook [37]), all support modified Kneser-Ney smoothing: KenLM [76], BerkeleyLM [122], RandLM [154], Expgram [164], MSRLM [119], SRILM [152], IRSTLM [60] and the recent approach based on suffix trees by Shareghi et al. [146, 147]. For such reasons, Kneser-Ney is the model we consider in this work too and that we review in Section 9.1.

The current limitation of the mentioned software libraries is that estimation of such models occurs in internal memory and, as a result, these approaches are not able to scale to the dimensions we consider in this work. An exception is represented by the work of Heafield, Pouzyrevsky, Clark, and Koehn [77] (KenLM) that contributed an estimation algorithm involving three steps of sorting in external memory. Their solution embodies the current state-of-art solution to the problem: the algorithm takes, on average, as low as 20% of the CPU and 10% of the RAM of the cited toolkits [77]. Therefore, the focus of this chapter is on improving upon the I/O efficiency of such an approach.

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

(a)

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

(b)

Figure 39: A block of 12 5-grams sorted in *suffix* order **(a)** and sorted in *context* order **(b)**.

Our contributions. We list here the main contributions of this chapter.

(1) We present a faster estimation algorithm that requires only one step of sorting in external memory, as opposed to the state-of-the-art approach [77] that requires three steps of sorting. The result is achieved by the careful exploitation of the properties of the extracted N-gram strings. Thanks to such properties, we show how it is possible to perform the whole estimation on the *context*-sorted strings and, yet, be able to efficiently lay out a reverse trie data structure, indexing such strings in *suffix* order.

We provide an efficient implementation of the described algorithm targeting billions of N-grams in external memory and show that saving two steps of sorting in external memory yields a solution that is 2.87× faster on average than the fastest algorithm proposed in the literature.

(2) We introduce many optimizations to further enhance the running time of our proposal, such as: asynchronous CPU and I/O threads, parallel LSD radix sort, block-wise compression and multi-threading.

(3) With an extensive experimental analysis conducted over standard text datasets, we study the behavior of our solution at each step of estimation; quantify the impact of the introduced optimizations and consider the comparison against the state-of-the-art. The devised optimizations further improve the running time by 1.6× on average, making our optimized algorithm 4.5× faster than the state-of-the-art.

9.1 Related Work

Basics and assumptions. Since the sorted orders defined over a set of n-grams are central to the description of the algorithms we are going to consider, we now define them. Consider a set of n-grams. The set is put into sorted order by sorting the n-grams on their words, as considered in a *specific* order.

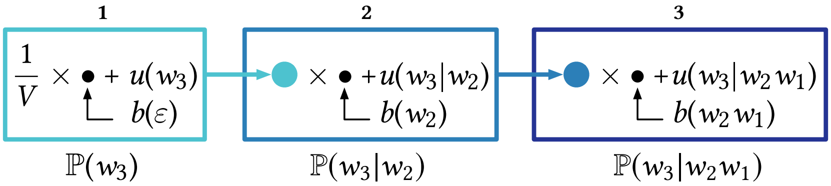


Figure 40: The Kneser-Ney interpolated probabilities for a 3-gram, calculated in a bottom-up fashion, from left (1-gram) to right (3-gram).

If this specific order is $N, N-1, \ldots, 1$, i.e., we sort n-grams from the last word up to the first, then the block is *suffix*-sorted: last word is primary (Figure 39a). If the considered order is $N-1, N-2, \ldots, 1, N$, then the block is *context*-sorted: penultimate word is primary (Figure 39b).

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

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

(2) the estimate is performed *without* pruning [77], thus the minimum occurrence count for an n-gram is 1;

(3) the compressed index built over the n-gram strings (e.g., the trie presented in Section 8.2) *must* reside in internal memory to allow fast query processing (perplexity and machine translation) [76, 122].

Modified Kneser-Ney smoothing. We detail here the *modified* version of Kneser-Ney smoothing [94] that was introduced by Chen and Goodman [35] and shown to have superior performance with respect to regular Kneser-Ney in terms of perplexity score. Under this model, the conditional probability $\mathbb{P}(w_n|w_1^{n-1})$ is computed recursively according to the following equation

$$\mathbb{P}(w_n|w_1^{n-1}) = u(w_n|w_1^{n-1}) + b(w_1^{n-1}) \times \mathbb{P}(w_n|w_2^{n-1}) \tag{10}$$

where 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

$$u(w_n|w_1^{n-1}) = \frac{a(w_1^n) - D_n(a(w_1^n))}{\sum_x a(w_1^{n-1}x)} \tag{11}$$

$$b(w_1^{n-1}) = \frac{\sum_{k=1}^{2} D_n(k) \times N_k(w_1^{n-1}\bullet) + D_n(3) \times N_{3+}(w_1^{n-1}\bullet)}{\sum_x a(w_1^{n-1}x)} \tag{12}$$

Also, refer to Figure 40 for an example of a 3-gram interpolated probability.

We now explain the quantities used in the above formulas. The quantity $a(w_1^n)$ is called the *modified count* for the n-gram w_1^n and it is equal to: the raw occurrence count $c(w_1^n)$ of w_1^n in the text if $n = N$; $|\{x : xw_1^n\}|$, that is, informally, 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; $N_{3+}(w_1^{n-1}\bullet)$ is equal, instead, to $|\{x : a(w_1^{n-1}x) \geq 3\}|$, i.e., the number of n-grams having modified count greater than or equal to 3.

Recursion terminates when uni-grams 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. Notice that $b(\varepsilon)/V$ is a constant quantity that depends on the textual collection used for estimating the language model. Finally, following [35, 36], closed-form discounts $D_n(k)$ are computed according to

$$D_n(k) = \begin{cases} 0, & k = 0 \\ k - (k+1) \times \frac{t_{n,1} t_{n,k+1}}{(t_{n,1} + 2t_{n,2}) t_{n,k}}, & k = 1, 2, 3 \\ D_n(3), & \text{otherwise} \end{cases} \tag{13}$$

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

State-of-the-art. The use of the Map+Reduce paradigm for the problem has been advocated in [23]. As reported in the paper, estimation involved hundreds of machines for a few days. Our work does not consider distributed computations, rather it shows how to let the estimation process scale well on the cores of a single target machine. Nguyen et al. [119] (MSRLM) also considered estimation on a single machine, using a parallel merge sort implementation. However, part of the estimation process is delayed until query-time: while this allows to save some resources during estimation, it also imposes a significant burden during the most efficiency-demanding use of language models which is query processing [36, 76]. We, instead, prefer to follow the approach of [77] that performs all steps of estimation as to permit the building of an efficient, static, compressed index over the computed model.

The works done by Stolcke [152] (SRILM), Federico et al. [60] (IRSTLM), Pauls and Klein [122] (BerkeleyLM) and Watanabe et al. [164] (Expgram) build Kneser-Ney language models in internal memory without resorting on sophisticated software optimizations and data compression techniques: as a result, are not able to scale to the dimensions we consider in this work.

Heafield et al. [77] (KenLM) contributed an estimation algorithm involving three steps of sorting in external memory. Their solution, referred to as the 3-Sort algorithm in the following, significantly outperforms the approaches that we have mentioned above, making it the state-of-art solution to the problem.

Shareghi et al. [146] resort on compressed suffix trees to compute on-the-fly the Kneser-Ney probabilities. The experimental analysis reported in the paper compares against SRILM and shows that such approach is comparable in building time with SRILM indexes but several orders of magnitude (e.g., 1000×) slower to query. In [147] the same authors improved over their previous work [146] by pre-computing some modified counts to speed up the on-the-fly calculation of the Kneser-Ney probabilities. Although pre-computing allows for significant improvement at query time (by up to 2500× faster than the previous solution) at the price of a larger index construction time (70% more time), the resulting language model is still 5× slower than KenLM.

For the reasons discussed in this paragraph, we aim at improving upon the I/O efficiency of the 3-Sort approach of KenLM that we describe in details in Section 9.2.

9.2 The 3-Sort algorithm

In this section we review the algorithm devised by Heafield et al. [77] since our work aims at improving its I/O efficiency. As already states, the algorithm is the fastest implementation of modified Kneser-Ney smoothing up to date, as it takes 25.4% and 7.7% of, respectively, CPU time and RAM of SRILM; 16.4% and 16.6% of CPU and RAM of IRSTLM [77].

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

9.2.1 Counting

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

9.2.2 Adjusting

All blocks sorted in suffix order are merged together in 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, as we have seen in Section 9.1, 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 Formula 13.

9.2.3 Normalization

This step computes normalized probabilities and backoffs according to, respectively, Formula 11 and 12. For such purpose, the blocks B_n, $1 < n \leq N$, produced during the previous Adjusting 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 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 records, enclosed in the rectangles in Figure 40, is one needed to perform interpolation. The computed backoffs are saved twice on disk, also as bare values without keys, one file per order $1 \leq n < N$ to facilitate the next step of interpolation and joining.

9.2.4 Interpolation and joining

The last streaming step performs interpolation of all orders to compute the final Kneser-Ney probability as in Equation 10. 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 40 offers a pictorial representation of this bottom-up process for a 3-gram. Note that the backoffs for the contexts that are needed for interpolation were saved in-line with the string w_1^n during the previous step. Also note that since normalization streamed through the blocks sorted in context order, the backoffs were saved to disk in suffix order.

Therefore, during this step the two quantities $\mathbb{P}(w_n|w_1^{n-1})$ and $b(w_1^n)$ are joined together, for $1 \leq n < N$ (N-grams do not have backoff).

9.3 Improved construction: the 1-Sort algorithm

In this section we introduce our main result that is an estimation algorithm for unpruned, modified, Kneser-Ney language models which substantially improve upon the I/O efficiency of 3-Sort by requiring *only one* sorting in external memory.

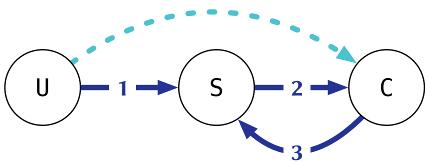


Figure 41: Sorting passes performed between N-grams: unsorted (U), suffix-sorted (S) and context-sorted (C). Solid arrows describe the path followed by the 3-Sort algorithm; the dashed arrow the one followed by the 1-Sort algorithm.

In fact, from the description in Section 9.2 we observe that the running time of 3-Sort is dominated by the cost of sorting in external memory, which is paid *three times* in total: (1) from extraction order (unsorted) to suffix order, (2) from suffix order to context order and then (3) from context order to, again, suffix order. This round-trip is the performance bottleneck of 3-Sort and it is graphically represented in Figure 41. 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 positively to such question.

As a general overview, the 1-Sort algorithm we are going to describe performs three steps:

(1) Counting N-grams (Section 9.3.1).

(2) Adjusting counts (Section 9.3.2).

(3) In a single, last, pass: normalization and interpolation (Section 9.3.3), joining and index construction (Section 9.3.4).

In what follows we detail the steps performed by the algorithm in comparison with 3-Sort and, thus, show how to save two steps of sorting.

9.3.1 Counting

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

If a cell of the set is not empty and contains the identifier $k \geq 0$, our probe consists in comparing the extracted N-gram string with the $4N$ bytes stored in the block starting from position $k \times (4N + 8)$. If the comparison yields equality, then we increment the corresponding counter, otherwise we advance to the next probe position. If any probed cell is found to be empty, then we write there the next available identifier (equal to the number of distinct seen N-grams) and append a new record to the in-memory block. As soon as we completely fill the block, we use a parallel thread to sort and write it to disk, thus hash deduplication of the text and I/O happen simultaneously.

The key difference of this step with respect to the one of 3-Sort, lies in the fact that we sort the blocks in *context* order instead of suffix order. The reason for this choice will become clear as we proceed in the description of the subsequent steps.

9.3.2 Adjusting

All blocks written to disk by the Counting step are merged together 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 $k = 1, \ldots, 4$ (Formula 13). Because smoothing statistics and, thus, discount coefficients, depend on the modified counts of the n-grams, the key ingredient we develop in this subsection is a linear-time algorithm that computes the modified counts of all n-grams for $1 \leq n < N$ *by scanning the context-sorted block* B_N.

In particular, the merged records are accumulated in a block $block[1, m]$, in memory, of m records. When the block fills up, we run the algorithm over the block B. We repeat the process until the whole input B_N is processed completely. At the end of the process, we use Formula 13 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 in-

	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

Figure 42: 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.

troduce 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* (Section 9.2). Because the next step of estimation is normalization and it requires context order, computing discount coefficients *directly* over the strings sorted in context order has the benefit of *avoiding to sort from suffix to context*. We are, therefore, eliminating the sorting step 2 of Figure 41.

Exploiting the completeness of N-gram strings. First of all, observe that since estimation is done without pruning by assumption *and* N-grams are extracted using a window of size N that slides by one word at a time, *the strings in B_N cover the input text completely*. This means that *all* the substrings of length $1 \leq n < N$ of each N-gram occur as substrings of some other N-gram in B_N. Refer to Figure 42 and consider the first 5-gram ABAAC in the context-sorted block at the bottom of the picture. For example, we know that its sub-string BAA must appear at positions 1, 2 and 3 of some other N-grams (the ones in position 7, 1 and 2, respectively). In particular we know that its *prefix* of length 4, i.e., ABAA will be matched at position 2 in some other N-gram (in this case, the second N-gram XABAA). We will return to this point later on, in Section 9.3.4.

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

(1) A sorted scan of the n-grams can be performed by just scanning B_N, without the need of replicating on disk all other n-gram strings, for $1 \leq n < N$, as done by 3-Sort during the Adjusting step.

(2) Let C_{n-1} be the context of length $n-1$ of an n-gram w_1^n. The number of distinct left extensions, i.e., the distinct words appearing to the left of w_1^n,

```
compute_left_extensions(block, m)
    p_1^N = block[1]                                  ▷ previous record
    for i = 1; i ≤ m; i = i + 1
        w_1^N = block[i]
        right = w_N
        for n = 1; n < N; n = n + 1
            if n != 1 and w_{N-n}^{N-1} != p_{N-n}^{N-1}
                ++ranges[n]
                for k = 1; k ≤ 4; k = k + 1
                    T[n][k] += R[n][k]
                    R[n][k] = 0
            left = w_{N-n-1}
            update(n, left, right)
        p_1^N = w_1^N
        k = c(w_1^N)
        if k ≤ 4
            ++T[N][k]
```

Figure 43: The algorithm for computing the left extensions in context order.

can be computed by *scanning the N-grams whose context of length $n-1$ is equal to* C_{n-1}[1].

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. We now introduce the linear-time algorithm for computing the 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 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, i.e., $a(C_{n-1}w_N)$. Since B_N is sorted in context order, the entries PC_{n-1} are consecutive

[1] 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 42. We could scan the N-grams in position 7 and 8 to compute the distinct left extensions (words in blue) of the bi-gram AC, instead of the ones in position 1, 2, 3 and 4. The problem with this approach is that we would not be able to compute the wanted quantity for $(N-1)$-grams because, obviously, a context of length $N-1$ can not be extended to the left. Moreover, consider the first 5-gram ABAAC. Since interpolation produces the probabilities for all its suffixes, i.e., for C, AC, AAC and BAAC, we need the modified counts for *these* suffixes and *not* for its contexts A, AA, BAA and ABAA that we could have computed with the other approach.

```
update(n, left, right)
    s = statistics[n][right]
    k = s.count
    ℓ = s.left
    if n != 1
        if not_seen(n, right)
            k = 0
            ℓ = -1                                  ▷ invalid word ID
    if ℓ != left
        ℓ = left
        k = k + 1
        if k == 1
            ++R[n][1]
        else
            if 1 < k ≤ 5
                ++R[n][k]
                --R[n][k − 1]
```

Figure 44: The **update** algorithm at the core of the Adjusting step.

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}, and thus we need to only scan 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 44, 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 do nothing. This update step takes $O(1)$ worst-case and it is coded in the **update** function shown in Figure 44. We are sure to count correctly the number of left extensions because left words are seen in sorted order.

Figure 42 shows an example for the bi-gram AC. In this case we have C_{n-1} = A, thus we need to scan all the (consecutive) N-grams having an A as a context of length 1. These N-grams are the ones spanned by the light green rectangle in Figure 42. In this example, AC can be extended to the left with words A and B, as depicted in blue in the picture, thus $a(\text{AC}) = 2$. 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 the Figure 49. The trie stores the strings in suffix order. In other terms, the node spelling out the bi-gram CA will store two pointers: one for A and one for B. Again, we will return to this point when we will discuss how to lay out efficiently the reverse trie, in Section 9.3.4.

At the end of the scan of all entries with the same context C_{n-1}, it is therefore guaranteed that the table contains the modified counts for all the n-grams $C_{n-1}x$.

When the context C_{n-1} changes (line 7 in the pseudo code of Figure 43), then we would need a fast way of zeroing all counts in the table. Instead, we do not re-initialize the table explicitly which would cost $\Theta(V)$ time, but we associate each context an increasing identifier, as follows.

We store an identifier for each distinct word w_N, called *range* in the function **not_seen** of Figure 45, in the table *statistics*, that represents the identifier of the range in which the word w_N was last seen.

```
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
```

Figure 45: The **not_seen** function that checks whether the *right* word was not seen in the current *range*.

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 (lines 6-8 in the pseudo code of Figure 44 and Figure 45).

Before concluding, there are two corner cases that we must mention for completeness: the one of N-grams and the one of 1-grams. The former because N-grams do not have modified counts, rather their counts are equal to the raw frequency counts written in the block B_N (lines 15-17 in Figure 43). The latter because their context is empty and we do not have to re-initialize their counts in the table when we switch range (`if` at line 5 in the pseudo code in Figure 44).

Collecting smoothing statistics. We are left to 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 of size $\Theta(V)$ used by the algorithm whenever we change context and update the counters accordingly. This approach is clearly infeasible in terms of running time. 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 44), 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 44). Whenever we change context, the local counts accumulated in R are first combined with the global ones in another array T and, then, re-initialized (lines 9-11 in the pseudo code in Figure 43). Also this re-combining step takes constant time.

Finally, from the computed smoothing statistics we can calculate the discount coefficients D_n using Formula 13. These are kept in an array $D[1, k]$, one for each order $1 \leq n \leq N$ and $k = 1, 2, 3$.

9.3.3 Normalization and interpolation

In the previous subsection we have shown how we can compute the modified counts over the block B_N sorted in context order. Thanks to this tool, we can therefore calculate pseudo probabilities and backoff values using Formula 11 and 12 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 46. In order to interpolate all 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 47 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 in three loops. The loop in the lines 4-6 calculates the numerator of the backoff for the context. The one in the lines 8-11 calculates the denominator for normalized probabilities and backoffs. Finally, the one in the lines 13-20 calculates the interpolated probabilities. The case for the N-grams in line 22 is not shown here because, as observed in the previous subsection, it is identical to the general case for $n < N$ with the only difference that the N-grams' counts are not modified but are the occurrence counts as seen in the text. We will return to this case in the next subsection.

Finally, for ease of presentation, the line 17 assumes that the uni-grams' probabilities are stored in the array *probabilities*[1]. Actually, a uni-gram probability

```
last(block, m)
    iterators[1, N] = [0, 0]
    while iterators[N] < m
        for n = 2; n ≤ N; n = n + 1
            i = iterators[n]
            p_1^N = block[i]
            size = 0
            while i < m
                w_1^N = block[i]
                if w_{N-n}^{N-1} == p_{N-n}^{N-1}
                    size = size + 1
                    left = w_{N-n-1}
                    right = w_N
                    update(n, left, right)
                else
                    break
                p_1^N = w_1^N
                i = i + 1
            write(n, size)
```

Figure 46: The main loop of the **last** step of estimation.

$\mathbb{P}(w_n)$ can be computed in $O(1)$ when needed as illustrated in the pseudo code in Figure 48.

In conclusion, normalization and interpolation are carried on as explained for the 3-Sort algorithm described in Section 9.1, but *without* requiring two separate sorting passes over the N-gram strings.

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

Figure 48: 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.

Another crucial difference is that the two phases are performed during the same scan of only one block, i.e., B_N, and we do not need to jointly iterate through N distinct files, one for each value of n, as done by 3-Sort. The net result is that we *avoid to sort from context to suffix* in order to perform interpolation, thus eliminating the sorting step 3 of Figure 41. Summing up, given that we have formerly shown how to save the sorting from suffix to context

```
write(n, size)
    i = iterators[n], j = offsets[n]
    b = 0, d = 0
    for k = 1; k ≤ 3; k = k + 1
        b += R[n][k] × D[n][k]
        R[n][k] = 0
    if n < N
        for ℓ = i − size; ℓ < i; ℓ = ℓ + 1
            w_1^N = block[ℓ]
            if not_seen(n, w_N)
                d += statistics[n][w_N].count
        b = b/d
        for ℓ = i − size; ℓ < i; ℓ = ℓ + 1
            w_1^N = block[ℓ]
            k = statistics[n][w_N].count
            u = (k − D[n][k])/d
            p = u + b × probabilities[n − 1][j]
            probabilities[n].add(p)
            j = j + 1
            lines 1-4 of Figure 50a
    else
        lines 1-13 of Figure 50b
    offsets[n + 1] = 0
```

Figure 47: The **write** function that performs normalization and interpolation.

too (Section 9.3.2), we have completely eliminated the round-trip of 3-Sort mentioned at the beginning of Section 9.3.

9.3.4 Joining and indexing

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

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 Chapter 8. In particular, it is the *reverse* trie variant, such as the one depicted in Figure 49, because it optimizes the left-to-right pattern of lookups

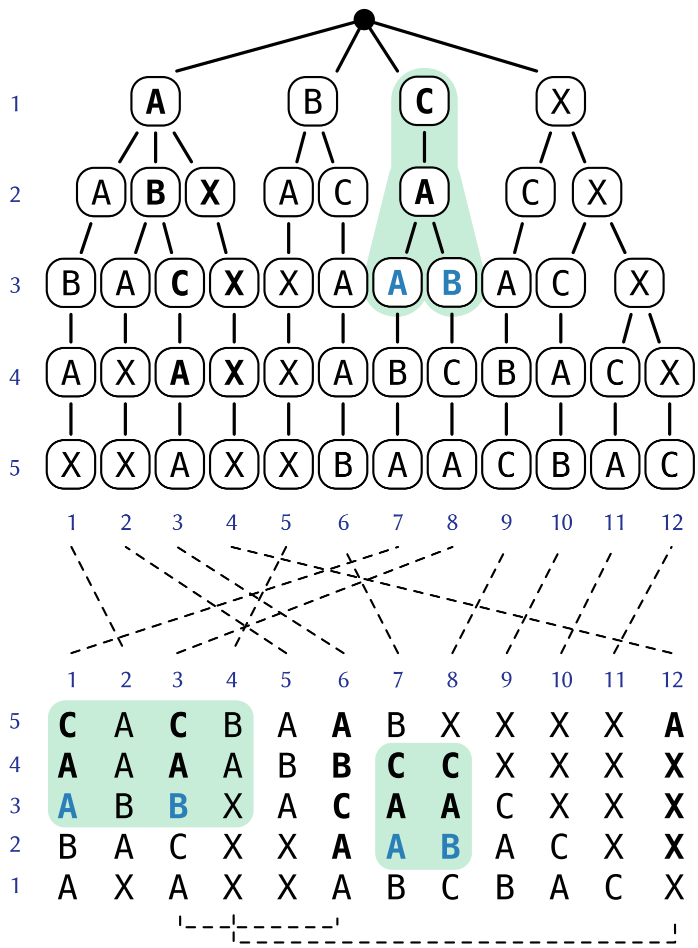


Figure 49: The 5-gram block sorted in context order of Figure 42 in relation with its reverse trie representation. The bottom level of the trie, i.e., [X, X, A, X, X, B, A, A, C, B, A, C] is obtained by permuting the *first* words of the strings in the context-sorted block, i.e., [A, X, A, X, X, A, B, C, B, A, C, X], according to the lexicographic position of their *last* words, i.e., [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.

performed by perplexity scoring [122, 76] (see also the perplexity benchmark in Section 8.4.3).

For this purpose, we exploit the property already mentioned in Section 9.3.2, that is: *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 the example in Figure 49 for $N = 5$, 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 N-gram strings in the context-sorted block. For example, the prefix of length 4 of ACBAC, i.e., ACBA, is found in the 6-th string; the one of XXXAB in the 12-th string instead (follow the dashed lines at the bottom of Figure 49). Notice that we *always* find the match at position 2, thus the first 4 levels of the trie store such 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 can be, therefore, 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 N-gram ABAAC. Since interpolation produces the probabilities for all the suffixes, i.e., for C, AC, AAC and BAAC, *we compute the backoffs for their contexts*, i.e., $b(\varepsilon)$, b(A), b(AA) and b(BAA) respectively, which appear in sorted order in the block. Refer to Figure 40 too 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. In fact, notice that: regarding the first problem, we can not 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 rather 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 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 49 (page 166) 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 such string will terminate with A (first word) in the bottom level of the trie. The position at which we have to place this first word in the bottom level is the lexicographic position of the last word, i.e., the C. Since the lexicographic position of the C is 7 within all 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 correct position, we use a *count-indexing technique*. For each vocabulary word, we maintain the number of times it appears *as last word* of an N-gram in a direct-address table of size $\Theta(V)$.

Prefix-summing such counts (and shifting them 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 such position, we write the integer w_N in $O(1)$ and increment the position in the table by one. Notice that this is the same procedure used by counting sort, thus the correctness of the approach follows automatically (see Section 8.2 of [42]). It only requires V integer counters, that we store in an array $positions[1, N]$.

Let us consider a complete example. Refer to Figure 49 (page 166) and the pseudo code in Figure 50b. For the uni-grams A, B, C and X, we count how many times they appear as last words of the N-grams and we obtain the following counts $[4, 2, 2, 4]$, because A and X appear 4 times each, while B and C appear twice each. Now we prefix sums such counts[2], obtaining $[5, 7, 9, 13]$, and we shift them one position to the right, obtaining the following initial $positions[5][4] = [1, 5, 7, 9]$.

Consider the first 5-gram in the context-sorted block, i.e., 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. Indeed, the 7-th string in the reverse trie of Figure 49 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, i.e., XABAA. The position associated to A is 1, so we have to write the first word X at position 1. We repeat the process for all the N-grams in the context-sorted block: following the dashed lines of Figure 49, 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 50b and it represents the case for $n = N$ in the **write** pseudo code in Figure 47 (line 22).

```
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)

Figure 50: 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 47.

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 only appears in 3 distinct contexts, i.e., 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

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

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

Table 33: Number of n-grams for the datasets used in the experiments.

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 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 50a and it completes the **write** function coded in Figure 47 (line 20).

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).

9.4 Experiments

The experiments we now show have the purpose of first characterizing the running time of our solution, i.e., the 1-Sort algorithm, of introducing optimizations and of finally considering 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 [33] and 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 web pages crawled between January and February 2009, available at: http://www.lemurproject.org/clueweb09.

From each dataset we removed all non-ASCII characters and markup tags. We use the (standard) value of $N = 5$ in every experiment. The datasets are of increasing size, reported as the number of n-grams in Table 33: 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 3.6 GHz, with 64 GB of RAM DDR3, running Linux 4.4.0, 64 bits. RAM is clocked at 2.133 GHz. The machine is equipped with a mechanical disk of 3 TB WDC WD30EFRX-68E, with standard page size of 4 KB.

We implemented the 1-Sort algorithm in standard C++14. As our competitor, we use the C++ implementation of 3-Sort as provided by the authors of [77] and available at http://kheafield.com/code/kenlm. We refer to this implementation as KenLM, which is the lead toolkit for language modeling [76]. As a matter of fact, KenLM provides the fastest estimation algorithm, significantly outperforming the previous approaches [77] as reported in Section 9.1. This is also confirmed by other recent experiments, showing KenLM to be up to 10× faster to build for the typical values of $n \le 5$ than approaches based on compressed suffix trees [147].

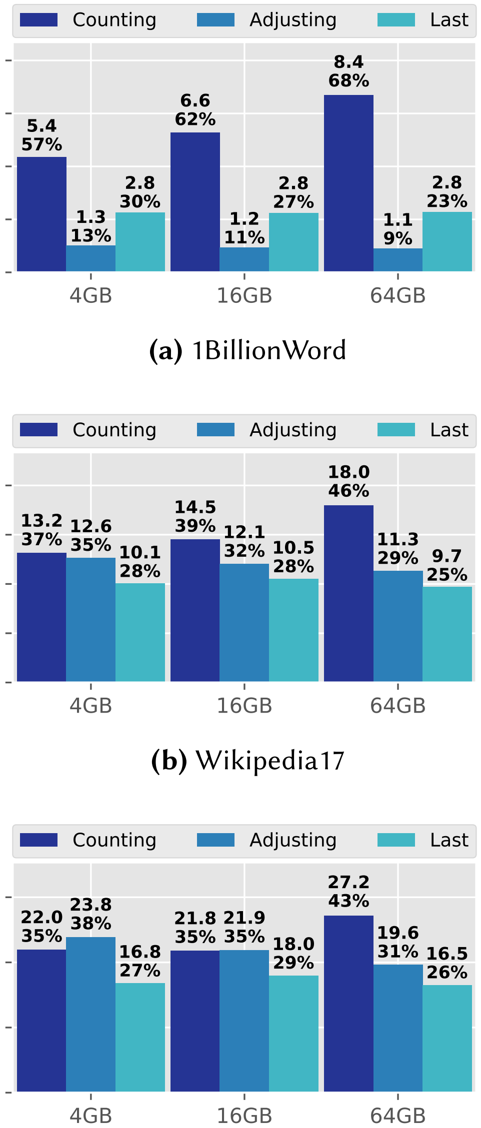
Both implementations were compiled with gcc 5.4.0, using the highest optimization setting, i.e., with compilation flags `-O3` and `-march=native`.

Source code. https://github.com/jermp/tongrams

9.4.1 Preliminary analysis

As a first set of experiments we show how the running time of our algorithm depends on the amount of provided internal memory. We inspect CPU and I/O activity to explain the observed runtime.

Varying the amount of internal memory. As a first experiment, we show the running time of our algorithm at each step of estimation, by varying the allowed amount of internal memory. We show the results using three values: 4 GB, 16 GB and the maximum available RAM, 64 GB. This experiment aims at showing which steps are the most expensive and fix the amount of internal memory that we will use for the subsequent analysis.



(a) 1BillionWord

(b) Wikipedia17

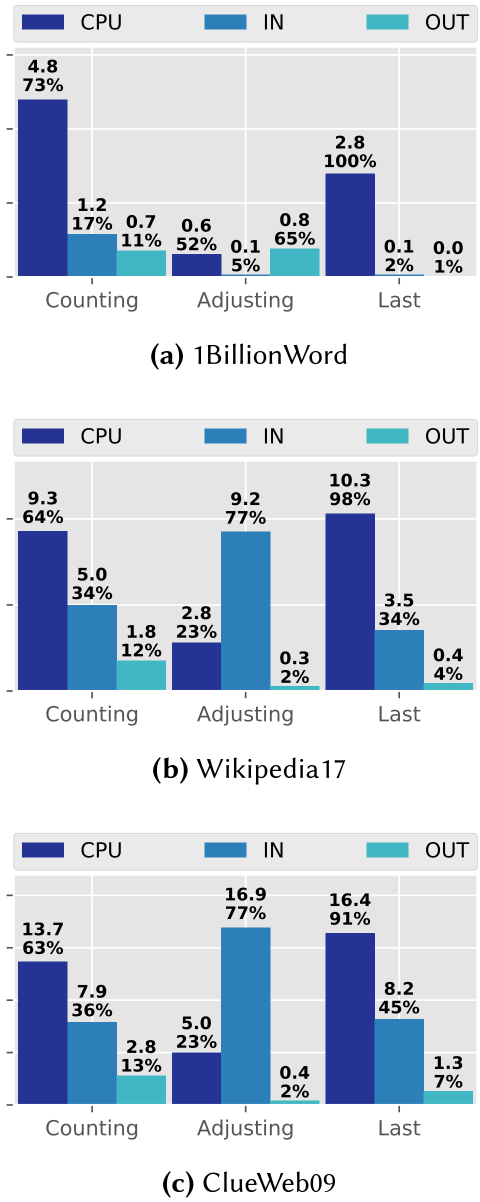
(c) ClueWeb09

Figure 51: Time in minutes spent at each step of estimation by using different amounts of internal memory.

The plots in Figure 51 illustrate the results. Above each bar, we report two numbers: the first indicating the number of minutes spent during the step, the second indicating the percentage with respect to the total running time of the algorithm. This grand total measures the time of the whole estimation process, i.e., the time it takes from the scanning of the input text to the flushing on disk of the compressed index built over the extracted strings. Some considerations are in order.

First of all, we can observe that, not surprisingly, the size of the language model has a significant impact not only on the total running time but also on which step becomes the most expensive. In fact, while on the 1BillionWord dataset the Counting and the Last steps contribute for more than 80% of the total running time and the Adjusting step has a quite low impact, the trend changes significantly on the larger datasets. In fact, on Wikipedia17 and ClueWeb09 the total running time is almost evenly distributed across the three steps. Notice 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 together during the Adjusting step. On the smaller dataset 1BillionWord, we have relatively few blocks to merge, thus Adjusting is performed quickly. Clearly, using more internal memory helps in lowering the number of blocks to merge and, thus, reducing the time for Adjusting.



(a) 1BillionWord

(b) Wikipedia17

(c) ClueWeb09

Figure 52: Time in minutes spent by CPU computation and I/O activity at each step of estimation.

We also observe that the step of Counting and the Last one do not vary much when more memory is available. Concerning the Counting step, more memory is not useful to lower the running time because using larger hash sets also means sorting larger blocks of N-grams. Indeed, observe that the total running time of Counting (slightly) increases by increasing the amount of memory.

However, as we have discussed above, using more memory for sorting implies fewer of blocks to merge, thus internal memory size has an impact only on the Adjusting step. For the open-address hash set implementation that we use in the Counting step, we experimented with linear probing, quadratic probing and double hashing. No significant difference among the three strategies was observed, thus we prefer linear probing for its better locality of accesses. Concerning the last step, we need to scan the merged N-gram file once. We use a standard buffered-scan approach using blocks of 64 MB by default. Using larger blocks does not impact the running time.

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

Inspecting CPU and I/O activity. It is now interesting to quantify the impact that CPU and I/O operations have on the total running time of each step. Under a different perspective, this analysis is also useful to understand and *how* disk usage is impacted by the size of the language model. The plots in Figure 52 illustrate such impact, i.e., the time spent by CPU and I/O at each step by using the amount of RAM that we fixed before (16 GB).

Dealing with external memory poses the challenge of trying to avoid CPU idle time by overlapping CPU computation with I/O activity. For such reason, we use asynchronous threads to handle input/output operations, so that while the CPU is performing internal processing, data is read or written to disk simultaneously [52]. This is a feature of particular importance for on-disk programs such as the ones we are considering, given the huge discrepancy in speed of modern processors and (mechanical) disks. Clearly, a perfect overlapping between CPU and I/O time would mean to only pay the maximum of the two. Consequently, the sum of three percentages for CPU, IN and OUT time for a given step in Figure 52, may exceed 100% because these are handled by different threads. Let us now consider each step in order.

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

The total running time of the Adjusting step is, instead, dominated by the cost of reading the blocks from the disk. This is no surprise given that multiple input streams are contending the disk for input operations, thus incurring in more disk seeks [163]. 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 9.3.2 is negligible compared to the overall running time of the step and contributes to a small percentage of the CPU: it is just 0.42, 1.2 and 1.8 minutes on 1BillionWord, Wikipedia17 and ClueWeb09 respectively. The remaining part of the CPU is spent by iterating through the fetched block of N-grams and comparing records during the merging process.

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

9.4.2 Optimizing our solution

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

Counting: implementing a parallel radix sort. In order to lower the total running time of the Counting step, it is important to guarantee a good overlap between input scanning and sorting in order to only pay the maximum of the two latencies and not the sum of the two. For this reason, we use LSD (least-significant-digit) *radix sort* [42], instead of the general-purpose std::sort. This sorting algorithm is the right choice in our setting because each N-gram is a (short) string of exactly N word identifiers, thus N passes of counting sort, i.e., one for each word index j, $j = N-1, 0, 1, \ldots, N-2$, are sufficient (and necessary) to sort a block in context order. The time complexity to sort a block of m N-grams is $\Theta((m+V)\times N)$, which is $\Theta(m \times N)$ given that $V = O(m)$.

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

Thanks to this strategy and by using all the available cores on our test machine ($K = 4$), the time for the Counting step improves substantially[3] 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 running time should be devoted in reducing the loading time from disk, because lowering the CPU cost will result in a negligible improvement. For this reason we compress the N-gram blocks created during the Counting step. Compressing the blocks has the potential of reducing the time spent in reading from disk because

[3] 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 8 GB, the gnu::parallel_sort takes 30 seconds while our parallel LSD radix sort takes 16.4 seconds.

	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×)

(a) Wikipedia17

	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×)

(b) ClueWeb09

Table 34: The effect of compressing blocks during the Adjusting step, on Wikipedia17 and ClueWeb09 datasets. The table reports: the time in minutes spent by computation (CPU), reading from disk (IN) and globally (total) and the average bytes per gram achieved by the different implementations.

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* [168] representation of an N-gram block, as follows. We fix a window size in bytes (64 MB by default, in our implementation) and compress as many records $\langle w_1^N, c(w_1^N)\rangle$ as possible, i.e., 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 words identifier we have to copy from the previous record (in context order) and s is the remaining part of the string. The first record of each window is written uncompressed.

We can use the minimum number of bits or bytes to represent each word identifier and frequency count. We refer to such strategies as, respectively, FC bit-aligned and FC byte-aligned, whose impact is evaluated in Table 34. As we can see from the data reported in the table, the bit-aligned version offers a 3× space reduction: from 28 bytes per record of the uncompressed version, we pass to an average of 9 bytes per record on Wikipedia17 and to 9.75 bytes per record on ClueWeb09. As a net result, the Adjusting step on Wikipedia17 and ClueWeb09 runs 2× and 1.5× faster. Indeed, we can observe that the input time decreases significantly: it is almost 100× smaller on Wikipedia17 and more than 3× smaller on ClueWeb09.

However, notice that the CPU time rises as well, roughly 2×, due to decoding from a compressed stream: we trade CPU time for less reading from disk. The byte-aligned version, FC byte-aligned, avoids the many bit-level instructions to decode a record. Not surprisingly, we can see that this strategy is actually faster than the bit-aligned version by 25% on average, while only allowing a slightly worse compression (2.5× on average compared to 3×). In conclusion, compressing the N-gram blocks with byte-aligned front-coding yields an improvement of 2.4× and 1.9× on Wikipedia17 and ClueWeb09 datasets, respectively. Therefore, for the rest of the experiments we use the FC byte-aligned representation of the blocks. On the smaller dataset 1BillionWord, however, compressing the blocks does not yield an appreciable improvement since input time from disk takes a negligible fraction of the total running time of the step (see Figure 52a).

Last: processing N-gram blocks in parallel. As discussed in Section 9.4.1, the last step of estimation is CPU-bound. Thus, we can use multi-threading to speed up the execution of the step. If K is the chosen parallelism degree, we use 1 reader thread to load the next $K-1$ blocks from the merged N-gram file and $K-1$ worker threads to process these blocks in parallel. While each worker thread independently executes the algorithm described in Section 9.3.4 on its own block, the reader thread asynchronously loads the next $K-1$ blocks in memory. The main challenge of this approach lies in computing the partition of each level of the trie that has to be written by a worker thread. For such purpose, we adopted a similar partitioning strategy to the one described in the previous subsection: in a first phase, each worker thread computes the number of distinct n-grams in its own block; in a second phase these counts are combined to obtain the offsets of the global partition of the trie. Although the first phase is performed in parallel, it has an impact on the achieved scalability.

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

9.4.3 Overall comparison

In this subsection we compare the performance of our solution, featuring all the optimizations that we have discussed before, against the state-of-the-art implementation of 3-Sort that is KenLM. The first comparison plots we show are illustrated in Figure 53. The plots strictly confirm the thesis of this chapter. The round-trip performed by 3-Sort, i.e., the sorting from suffix to context and then back from context to suffix (see Figure 41), results in a severe penalty on the total running time of the estimation process: our improved 1-Sort algorithm exploits the properties of the extracted N-gram strings in order to *completely avoid* the round-trip.

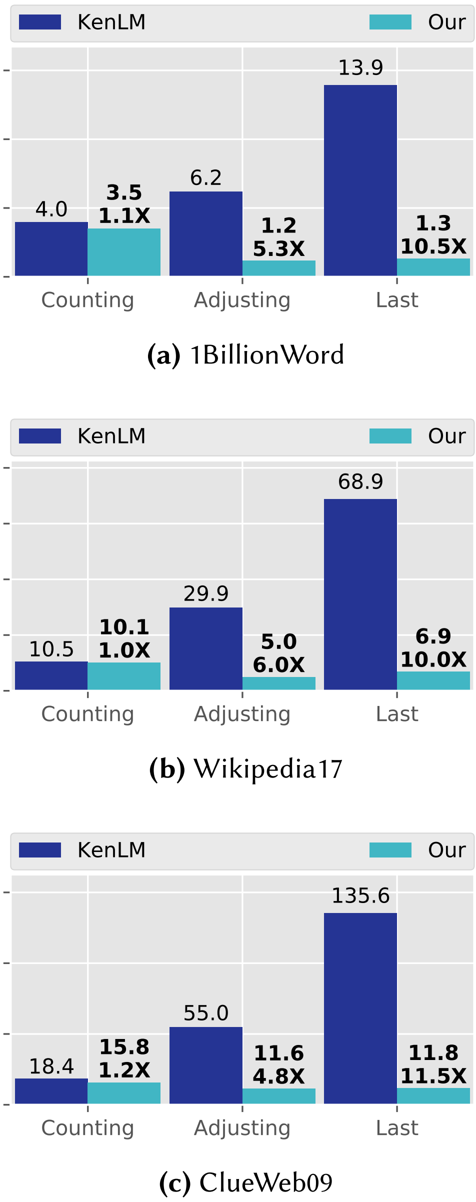


Figure 53: Time in minutes spent by KenLM and our algorithm at each step of estimation.

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 explained in Section 9.3.1, the first step of Counting is performed similarly by the two algorithms and this is the reason why the corresponding running times are comparable.

In fact, both algorithms use a separate thread to sort the previously-formed block in parallel and flushing it to disk while input scanning takes place at the same time. Both implementations also use open-addressing with linear probing. The key difference lies in the fact that we sort in context order, whereas KenLM adopts suffix order. Another crucial difference is that our solution compresses the blocks to reduce the merging time in the next step, that KenLM does not do.

During the Adjusting step, our approach computes the modified counts in context order as described in Section 9.3.2 on every output block formed during the merging process. KenLM does the same but over suffix-sorted blocks, thus it has to write back to disk *each n-gram, for* $1 < n \leq N$, along with its own modified count, in context order.

Since our approach re-computes the modified counts during the process of normalization itself, we only need to handle the N-grams and merge their blocks. Instead, KenLM has to finally merge the blocks or all n-grams written to disk.

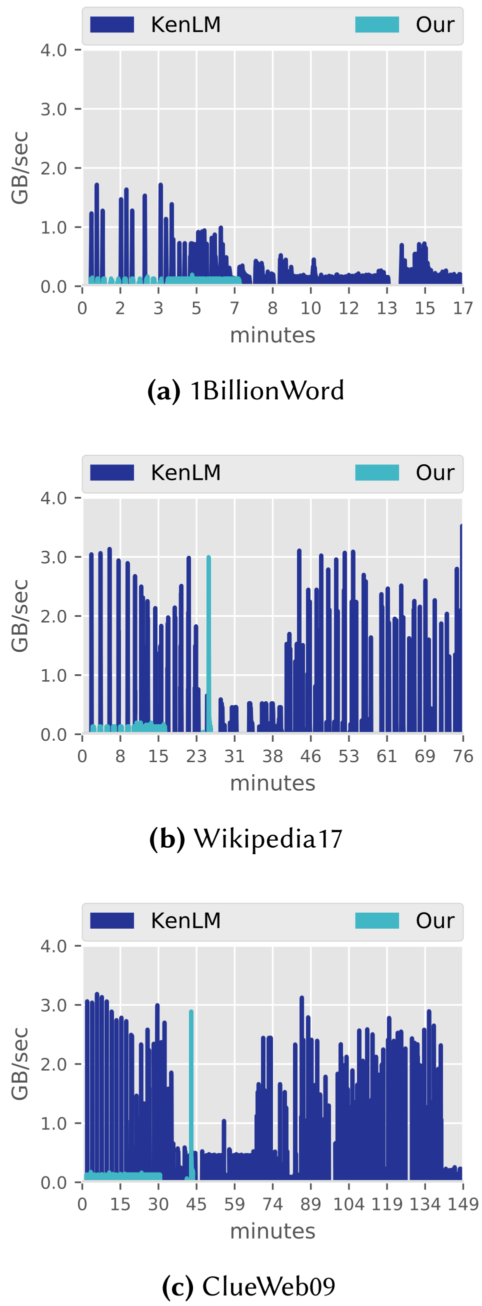


Figure 54: Gigabytes per second written on disk by KenLM and our algorithm.

Although it exploits multiple threads (one for each order), the additional writes to disk and sorting operations cause KenLM be on average 5.3× slower during this step than our approach.

During the last step, normalization, interpolation and indexing are performed (Section 9.3.3 and 9.3.4). Again, we can observe an average speed-up of 10.6×. Since our algorithm builds a compressed reverse trie index during the same step, we also sum to the time of KenLM the time it takes to build the same data structure, because the current implementation does not build the index during the same pass (although the possibility is advocated in the paper [77]). To ensure fairness, the indexing time for KenLM is measured by excluding the time to write and parse the intermediate (ARPA) file on disk: it is anyway a significant amount of the total running time of KenLM, equal to 7, 31 and 61 minutes for, respectively, 1BillionWord, Wikipedia17 and ClueWeb09. Apart from indexing, the rest of the time is spent in sorting again from context to suffix order, as needed for interpolation.

Both normalization and interpolation phases of KenLM exploits multi-threading, by using separate threads for each value of n. In particular, two threads are used to compute the denominators and numerators of the quantities in Formula 11 and 12. Again, recall that we only need to tackle N-grams because we consider the other n-gram strings implicitly, thus our implementation uses multiple threads for in-memory processing and a thread to asynchronously feed the CPU with input.

Output volume. Another way of visualizing the comparison between our solution and KenLM is to measure the number of bytes read/written per second from/to disk by the two algorithms. Figure 54 shows the number of GB written per second on disk for each dataset. We collect the statistic using the Linux utility pidstat with time interval of 1 second and matching the name of the executed task. The volume for our construction also includes the one spent when flushing the compressed index to disk, whereas the volume for KenLM does not because the current implementation builds the index with a separate program.

The plots strictly match the results shown in Figure 53, i.e., not surprisingly the improvement in running time is directly proportional to the quantity of data written to disk. In fact, the area below the curve of our algorithm is ≈6× less than the one of KenLM (20.4 GB vs. 124.5 GB) on 1BillionWord; ≈5× less on Wikipedia17 (63.6 GB vs. 310.7 GB) and ≈5.8× less on ClueWeb09 (88 GB vs. 514 GB).

10 Future Research Directions

In this concluding chapter we discuss some challenging open problems that could be excellent candidates for investigation during the upcoming years.

Dynamic inverted indexes. We described the use of inverted indexes in Section 2.3 and presented many different techniques to effectively represent the inverted lists in Section 2.2 and in chapters 4, 5 and 6. We have also mentioned that all those results regard the use of a *static* inverted index: from a static textual collection, the data structure is built and no updates are possible. This could be a great limitation for interesting real-world applications that actually need to reflect changes in the indexing structure as users add and delete some contents. For example, we mentioned Firefly [82], the Dropbox full-text search engine. Other practical examples include Facebook, e.g., users constantly add new post and/or delete past ones, and Twitter, e.g., tweets must be indexed at exceptionally high rates. Therefore, an interesting open problem regards the engineering of *dynamic inverted indexes*.

A classical solution to this problem resorts on using two distinct indexes (many variations of this approach could be possible): one index is static and usually very big, the other is small and should reflect dynamic updates very fast. When the small index reaches a predefined capacity, it is merged with the big one into a new static index. The small index is dumped and the process repeated. This solution is simple, it has an amortized running time, and may require additional resources during the merging phase given that updates may occur before the merging phase completes. Is it possible to directly update the inverted lists without inducing a whole index reconstruction and, therefore, eliminating the need for two indexes? As already noted, dynamic updates can be a source of performance degradation for compressed data structures.

However, in Chapter 3 we presented a dynamic data structure which exhibits an optimal running time for such dynamic operations in almost optimal compressed space. Therefore, we believe that a practical implementation of the ideas described in that chapter may provide a new solution to the problem.

Compressed B-trees. The B-tree data structure has been one of the most successful data structures since its invention with applications to databases and filesystems (see Chapter 18 of [42]). Indeed the best known data structure which has been commercialized for databases is the *fractal tree index* [18], a variation of a classic B-tree. A fractal tree index matches the performance of a B-tree for search and sequential scan but allows faster insertion and deletions. The space usage is the same as the one for a B-tree. Unlike a B-tree, a Fractal Tree Index maintains for each node a buffer which records insertions and/or deletions to direct to its children. The role of such buffers is the one of amortizing the I/O cost over many updates. By exploiting such buffers, the insertion/deletion time is improved by a factor proportional to the length of employed buffers, i.e., B^{ε}, where B is the tree branching factor and $\varepsilon > 0$. Notice that buffering updates to a dynamic data structure may work well in cases when the data structure is compressed, since expensive rebuildings can be amortized over time while using only little extra space. In fact, the sorted keys of a B-tree are excellent to be compressed, e.g., with Elias-Fano because it can allow fast searches. Compressing the keys at each node of a B-tree will effectively enlarge by a lot the quantity of data we can index. The crucial algorithmic challenge for this problem is to design *fast* split and merge operations on blocks of compressed keys.

Fast successor queries for IP-lookup. As observed by Pătraşcu and Thorup [134], the most important application of predecessor search is IP-lookup. This is the algorithm that Internet routers use to choose the subnetwork to which packets have to be forwarded to and, thus, the most run algorithm in the world. Time-efficiency for this algorithm is crucial. Nonetheless, compressing the forwarding tables so that a better cache exploitation is achieved has a great impact on time too. Unibit and Patricia tries (and their many variations) are the typical data structures used for this problem [48, 124, 153] as they support longest prefix match in time proportional to the length of the searched binary string (IP address, in this case). However, if a routing table contains n sorted IP (binary) strings, we know that can jump directly to the position of the first IP address having the same $\lceil \log n \rceil$ bits in $O(1)$, resorting on the powerful search capabilities of Elias-Fano (see Section 2.2.6). Therefore, the applicability of this encoder to forwarding tables is a profitable research problem.

Floating point compression. In this thesis we concentrated on compressing integer sequences and strings, and also discussed about their applicability to several practical scenarios. The integer data type is, however, not the only one relevant to real-world applications. In recent years there has been an explosion of works on Machine Learning. Most of these works have to train a neural network, an object whose input and output is a vector of floating-point numbers. Scientific, financial computing and 3D graphics in video games are good examples of applications that manage huge quantities of floating-point numbers. However, while a plethora of different results have been proposed for integer compression, little attention has been devoted to floating-point compression. To the best of our knowledge, the best result is zfp[1], a C/C++ library primarily written by Lindstrom [106]. To achieve the highest compression ratio, the library resorts on *lossy* compression. A nice research direction would be proposing a lossless coder for floating-point arrays.

For example, consider word2vec [111], a technology that produces word embeddings, i.e., a correspondence between a word and a floating-point vector. The length of the produced vector is user-defined. The key characteristics of word embeddings is that words that are similar (or semantically related) have also similar vectors. In other words, if plotted in a 2/3-dimensional space the points are very close to each other. In order to compress a large collection of word embeddings, we can therefore cluster similar vectors together and encode them with respect to the representative (e.g., a centroid) of each cluster, as similarly proposed in Chapter 4.

Practical implementation of compressed dynamic integer dictionaries. The dynamic data structure presented in Chapter 3 combines the space efficiency of Elias-Fano in representing integer sequences with the optimal running time for dynamic operations, e.g., adding and/or removing an integer. We have already argued that a practical implementation of this data structure could be of great impact for dynamic inverted indexes.

We also point out that in recent years dynamic succinct data structures have received attentions, with efforts spent in making entropy-compressed dynamic bit vectors [41] and dynamic partial sums data structures in compressed space [133]. Note that these are both useful building-blocks of the dynamic data structure designed in Chapter 3 of this thesis, thus a competitive implementation and comparison with these approaches could be a good starting point for a new experimental work.

Compression of spatial data. Geo-tagged data is now ubiquitous in several commercial applications, like Facebook, Twitter, GoogleMaps, Uber and the successful video games run by Niantic. The most primitive form of data manipulated by the aforementioned application is a latitude-longitude pair representing a point on the Earth surface. As mentioned in Chapter 1, such points can be represented in a 2-dimensional space, i.e., a sequence, by using a space filling Hilbert curve, as done by S2 Geometry [85]. Uber, instead, has developed an indexing technique based on hexagon cells and called H3 [86].

Some preliminary experiments performed on real datasets of points reveal that, not surprisingly, such sequences of points exhibit a clustering effect similar to the one exhibited by the inverted lists. This means that a great deal of compression is already achievable by adopting the classical techniques reviewed in Section 2.2.

However, the current storage engine of both S2 Geometry and H3 do not use any sophisticated compression technique (or at least, we suspect to). Therefore, a

[1] https://github.com/LLNL/zfp

good research direction could focus on developing an ad-hoc coder for sequences of coordinates.

Bibliography

[1] Protocol Buffers - Google's data interchange format, https://github.com/google/protobuf. Accessed on 15-04-2018. → 75

[2] RediSearch, https://github.com/RedisLabsModules/RediSearch/blob/master/docs/DESIGN.md. Accessed on 15-04-2018. → 75

[3] UpscaleDB, https://upscaledb.com/about01.html#compression. Accessed on 15-04-2018. → 75

[4] 2006. Yahoo! N-Grams, version 2.0, http://webscope.sandbox.yahoo.com/catalog.php?datatype=l. → 138

[5] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, Samuel Madden, and others. 2013. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases* 5, 3 (2013), 197–280. → 75

[6] Charu Aggarwal and Chandan Reddy. 2013. *Data Clustering: Algorithms and Applications* (1st ed.). Chapman and Hall/CRC. → 55 and 61

[7] Miklós Ajtai. 1988. A lower bound for finding predecessors in Yao's cell probe model. *Combinatorica* 8, 3 (1988), 235–247. → 36

[8] Arne Andersson and Mikkel Thorup. 2007. Dynamic ordered sets with exponential search trees. *Journal of the ACM (JACM)* 54, 3 (2007), 13. → 38

[9] Vo Ngoc Anh and Alistair Moffat. 2005. Inverted Index Compression Using Word-Aligned Binary Codes. *Information Retrieval Journal (IRJ)* 8, 1 (2005), 151–166. → 17

[10] Vo Ngoc Anh and Alistair Moffat. 2010. Index compression using 64-bit words. *Software: Practice and Experience (SPE)* 40, 2 (2010), 131–147. → 17

[11] A. Apostolico and S. Lonardi. 2000. Off-line compression by greedy textual substitution. *Proc. IEEE* 88, 11 (Nov. 2000), 1733–1744. → 102

[12] David Arthur and Sergei Vassilvitskii. 2007. K-means++: the advantages of careful seeding. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1027–1035. → 55, 56, and 62

[13] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. 2012. Scalable K-means++. In *Proceedings of the Very Large Database Endowment (PVLDB)*. 622–633. ⟶ 56

[14] Ziv Bar-Yossef and Naama Kraus. 2011. Context-sensitive query auto-completion. In *International Conference on World Wide Web (WWW)*. 107–116. ⟶ 31

[15] Paul Beame and Faith E. Fich. 1999. Optimal Bounds for the Predecessor Problem. In *Proceedings of the Thirty-First Annual Symposium on Theory of Computing (STOC)*. 295–304. ⟶ 37 and 38

[16] Paul Beame and Faith E. Fich. 2002. Optimal Bounds for the Predecessor Problem and Related Problems. *Journal of Computer and System Sciences (JCSS)* 65, 1 (2002), 38–72. ⟶ 37 and 38

[17] Djamal Belazzougui, Paolo Boldi, Giuseppe Ottaviano, Rossano Venturini, and Sebastiano Vigna. 2014. Cache-oblivious peeling of random hypergraphs. In *International Data Compression Conference (DCC)*. 352–361. ⟶ 137

[18] Michael A Bender, Martin Farach-Colton, Jeremy T Fineman, Yonatan R Fogel, Bradley C Kuszmaul, and Jelani Nelson. 2007. Cache-oblivious streaming B-trees. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. ACM, 81–92. ⟶ 181

[19] Bishwaranjan Bhattacharjee, Lipyeow Lim, Timothy Malkemus, George Mihaila, Kenneth Ross, Sherman Lau, Cathy McArthur, Zoltan Toth, and Reza Sherkat. 2009. Efficient index compression in DB2 LUW. *Proceedings of the Very Large Database Endowment (PVLDB)* 2, 2 (2009), 1462–1473. ⟶ 75

[20] Philip Bille, Patrick Hagge Cording, Inge Li Gørtz, Frederik Rye Skjoldjensen, Hjalte Wedel Vildhøj, and Søren Vind. 2018. Dynamic Relative Compression, Dynamic Partial Sums, and Substring Concatenation. *Algorithmica* 80, 11 (2018), 3207–3224. ⟶ 38 and 46

[21] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM (CACM)*. 422–426. ⟶ 127

[22] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis. 1994. Linear approximation of shortest superstrings. *Journal of the ACM (JACM)* 41, 4 (1994), 630–647. ⟶ 104

[23] Thorsten Brants, Ashok C Popat, Peng Xu, Franz J Och, and Jeffrey Dean. 2007. Large language models in machine translation. In *International Conference*

Empirical Methods on Natural Language Processing (EMNLP). 858–867. → 151 and 154

[24] 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*. → 138

[25] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Y. Zien. 2003. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the 12th ACM International Conference on Information and Knowledge Management (CIKM)*. 426–434. → 28 and 51

[26] Andrei Z Broder, Nadav Eiron, Marcus Fontoura, Michael Herscovici, Ronny Lempel, John McPherson, Runping Qi, and Eugene Shekita. 2006. Indexing shared content in information retrieval systems. In *International Conference on Extending Database Technology*. Springer, 313–330. → 54

[27] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. 2002. Holistic twig joins: optimal XML pattern matching. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, 310–321. → 27

[28] A.L. Buchsbaum, G.S. Fowler, and R. Giancarlo. 2003. Improving table compression with combinatorial optimization. *J. ACM* 50, 6 (2003), 825–851. → 78

[29] Michael Busch, Krishna Gade, Brian Larson, Patrick Lok, Samuel Luckenbill, and Jimmy Lin. 2012. Earlybird: Real-time search at twitter. In *Proceedings of the 28th International Conference on Data Engineering (ICDE)*. IEEE, 1360–1369. → 41 and 63

[30] Stefan Büttcher and Charles Clarke. 2007. Index compression is good, especially for random access. In *Proceedings of the 16th ACM International Conference on Information and Knowledge Management (CIKM)*. 761–770. → 51

[31] Stefan Büttcher, Charles Clarke, and Gordon Cormack. 2010. *Information retrieval: implementing and evaluating search engines*. MIT Press. → 26, 27, and 62

[32] Surajit Chaudhuri, Kenneth Church, Arnd Christian König, and Liying Sui. 2007. Heavy-Tailed Distributions and Multi-Keyword Queries. In *Proceedings of International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*. 663–670. → 54

[33] 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. → 148 and 170

[34] Ciprian Chelba and Johan Schalkwyk. 2013. Empirical Exploration of Language Modeling for the google.com Query Stream as Applied to Mobile Voice Search. In *Mobile Speech and Advanced Natural Language Solutions (MSANLS)*. 197–229. → 151

[35] Stanley Chen and Joshua Goodman. 1996. An empirical study of smoothing techniques for language modeling. In *Association for Computational Linguistics (ACL)*. 310–318. → 148, 153, and 154

[36] Stanley Chen and Joshua Goodman. 1999. An empirical study of smoothing techniques for language modeling. In *Computer Speech and Language (CSL)*, Vol. 13. 359–394. → 32, 148, 151, and 154

[37] Wenlin Chen, David Grangier, and Michael Auli. 2015. Strategies for Training Large Vocabulary Neural Language Models. In *Preprint arXiv:1512.04906*. 13. → 151

[38] David Clark. 1996. *Compact Pat Trees*. Ph.D. Dissertation. University of Waterloo. → 20

[39] David R. Clark and J. Ian Munro. 1996. Efficient suffix trees on secondary storage. In *Symposium on Discrete Algorithms (SODA)*. 383–391. → 128

[40] F. Claude, A. Fariña, and G. Navarro. 2009. Re-Pair compression of inverted lists. *CoRR* abs/0911.3318 (2009), 19. `http://arxiv.org/abs/0911.3318` → 102

[41] Joshimar Cordova and Gonzalo Navarro. 2016. Practical Dynamic Entropy-Compressed Bitvectors with Applications. In *International Symposium on Experimental Algorithms (SEA)*. 105–117. → 183

[42] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms - 3rd ed.* MIT Press. → 10, 35, 168, 175, and 181

[43] Bruce Croft, Donald Metzler, and Trevor Strohman. 2009. *Search Engines: Information Retrieval in Practice* (1st ed.). Addison-Wesley Publishing Company. → 31 and 125

[44] Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, Soren Lassen, Philip Pronin, Sriram Sankar, Guanghao Shen, Gintaras Woss, Chao Yang, and Ning Zhang. 2013. Unicorn: A System for Searching the Social Graph. In *Proceedings of the Very Large Database Endowment (PVLDB)*, Vol. 6. 1150–1161. → 27

[45] Jeffrey Dean. 2009. Challenges in building large-scale information retrieval systems: invited talk. In *Proceedings of the 2nd International Conference on Web Search and Data Mining (WSDM)*. → 16 and 75

[46] Jeffrey Dean. 2018. Latency Numbers Every Programmer Should Know, https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html. → 1

[47] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2011. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 25–36. → 27

[48] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. 1997. *Small forwarding tables for fast routing lookups*. Vol. 27. ACM. → 182

[49] Renaud Delbru, Stéphane Campinas, and Giovanni Tummarello. 2012. Searching web data: An entity retrieval and high-performance indexing model. *Journal of Web Semantics* 10 (2012), 33–58. → 17

[50] Erik D. Demaine, Thouis Jones, and Mihai Pătraşcu. 2004. Interpolation search for non-independent data. In *Symposium on Discrete Algorithms (SODA)*. 529–530. → 127

[51] Erik D. Demaine and Mihai Pătraşcu. 2004. Tight bounds for the partial-sums problem. In *Proceedings of the 15-th Annual Symposium on Discrete Algorithms (SODA)*. 20–29. → 38

[52] Roman Dementiev, Lutz Kettner, and Peter Sanders. 2008. STXXL: standard template library for XXL data sets. In *Software, Practice and Experience (SPE)*, Vol. 38. 589–637. → 174

[53] Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. 2016. Compressing Graphs and Indexes with Recursive Graph Bisection. In *Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (KDD)*. 1535–1544. → 30 and 68

[54] Ulrich Drepper. 2007. What every programmer should know about memory. *Red Hat, Inc* 11 (2007), 114. → 1

[55] Jarek Duda. 2013. Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding. *arXiv preprint arXiv:1311.2540* (2013), 24. → 25

[56] Peter Elias. 1974. Efficient Storage and Retrieval by Content and Address of Static Files. *Journal of the ACM (JACM)* 21, 2 (1974), 246–260. → 18, 19, 21, 35, and 42

[57] Peter Elias. 1975. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory (IT)* 21, 2 (1975), 194–203. → 14

[58] Robert Mario Fano. 1971. On the number of bits required to implement an associative memory. *Memorandum 61, Computer Structures Group, MIT* (1971). → 18 and 35

[59] Marcello Federico and Nicola Bertoldi. 2006. How many bits are needed to store probabilities for phrase-based translation?. In *Workshop on Statistical Machine Translation (WMT)*. 94–101. → 127 and 148

[60] Marcello Federico, Nicola Bertoldi, and Mauro Cettolo. 2008. IRSTLM: an open source toolkit for handling large scale language models. In *INTERSPEECH*. 1618–1621. → 151 and 154

[61] Paolo Ferragina, Igor Nitto, and Rossano Venturini. 2011. On optimally partitioning a text to improve its compression. *Algorithmica* 61, 1 (2011), 51–74. → 23, 78, and 79

[62] Agner Fog. 2014. *Optimizing software in C++: an optimization guide from Windows, Linux and Mac platforms*. Technical University of Denmark. → 33

[63] Edward Fredkin. 1960. Trie memory. In *Communications of the ACM (CACM)*. 490–499. → 33

[64] Michael L Fredman, János Komlós, and Endre Szemerédi. 1984. Storing a sparse table with O(1) worst-case access time. *Journal of the ACM (JACM)* 31, 3 (1984), 538–544. → 36

[65] Michael L. Fredman and Michael E. Saks. 1989. The Cell Probe Complexity of Dynamic Data Structures. In *Proceedings of the 21-st Annual Symposium on Theory of Computing (STOC)*. 345–354. → 5, 35, 36, 37, 38, and 43

[66] Michael L. Fredman and Dan E. Willard. 1993. Surpassing the Information Theoretic Bound with Fusion Trees. *Journal of Computer and System Sciences (JCSS)* 47, 3 (1993), 424–436. → 10, 35, 36, 37, and 38

[67] Kimmo Fredriksson and Fedor Nikitin. 2007. Simple compression code supporting random access and fast string matching. In *Workshop on Experimental Algorithms (WEA)*. 203–216. ⟶ 131

[68] J. Gallant, D. Maier, and J. A. Storer. 1980. On finding minimal length superstrings. *Journal of Computer and System Sciences (JCSS)* 20, 1 (1980), 50–58. ⟶ 104

[69] Michael Garey and David Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company. ⟶ 60

[70] Simon Gog and Matthias Petri. 2014. Optimized succinct data structures for massive data. *Software: Practice and Experience (SPE)* 44, 11 (2014), 1287–1314. ⟶ 9 and 13

[71] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing Relations and Indexes. In *Proceedings of the 14th International Conference on Data Engineering (ICDE)*. 370–379. ⟶ 17

[72] Solomon Golomb. 1966. Run-length encodings. *IEEE Transactions on Information Theory* 12, 3 (1966), 399–401. ⟶ 15

[73] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. 2005. Practical implementation of rank and select queries. In *WEA*. 27–38. ⟶ 13

[74] Roberto Grossi, Rajeev Raman, S. Srinivasa Rao, and Rossano Venturini. 2013. Dynamic Compressed Strings with Random Access. In *Proceedings of 40-th International Colloquium on Automata, Languages, and Programming (ICALP)*. 504–515. ⟶ 39

[75] Ankur Gupta, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. 2007. Compressed data structures: Dictionaries and data-aware measures. *Theoretical Computer Science (TCS)* 387, 3 (2007), 313–331. ⟶ 35

[76] Kenneth Heafield. 2011. KenLM: Faster and smaller language model queries. In *Workshop on Statistical Machine Translation (WMT)*. 187–197. ⟶ 31, 125, 127, 128, 137, 139, 151, 153, 154, 167, and 171

[77] 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. ⟶ 151, 152, 153, 154, 155, 171, and 179

[78] Steffen Heinz, Justin Zobel, and Hugh E. Williams. 2002. Burst tries: a fast, efficient data structure for string keys. In *Transactions on Information Systems (TOIS)*. 192–223. → 128

[79] C. Hoobin, S. J. Puglisi, and J. Zobel. 2011. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *Proceedings of the Very Large Database Endowment (PVLDB)* 5, 3 (2011), 265–273. → 97

[80] Vagelis Hristidis, Yannis Papakonstantinou, and Luis Gravano. 2003. Efficient IR-Style Keyword Search over Relational Databases. In *Proceedings of the Very Large Database Endowment (PVLDB)*. 850–861. → 27

[81] Samuel Huston, Alistair Moffat, and W. Bruce Croft. 2011. Efficient indexing of repeated n-grams. In *International Conference on Web Search and Data Mining (WSDM)*. 127–136. → 31

[82] Dropbox Inc. Firefly - Dropbox Techblog, `https://blogs.dropbox.com/tech/2016/09/improving-the-performance-of-full-text-search/`. Accessed on 15-04-2018. → 3, 75, and 181

[83] Dropbox Inc. Lepton, `https://blogs.dropbox.com/tech/2016/07/lepton/`. Accessed on 10-09-2018. → 3

[84] Facebook Inc. 2014. RocksDB Cuckoo Hashing Table Format, `http://rocksdb.org/blog/2014/09/12/cuckoo.html`. → 3

[85] Google Inc. 2015. S2 Geometry, `http://s2geometry.io/`. → 4 and 183

[86] Uber Inc. 2018. H3, `https://uber.github.io/h3/`. → 183

[87] Guy Jacobson. 1989. Space-efficient Static Trees and Graphs. In *Foundations of Computer Science (FOCS)*. 549–554. → 13, 127, and 139

[88] Guy Jacobson. 1989. *Succinct Static Data Structures.* Ph.D. Dissertation. Carnegie Mellon University. → 13

[89] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. 2012. CRAM: Compressed random access memory. In *Proceedings of 39-th International Colloquium on Automata, Languages, and Programming (ICALP)*. 510–521. → 38, 46, and 48

[90] Dan Jurafsky and James H. Martin. 2014. *Speech and language processing.* Pearson. → 4, 31, and 125

[91] Tomasz Jurkiewicz and Kurt Mehlhorn. 2014. On a Model of Virtual Address Translation. *ACM Journal of Experimental Algorithmics (JEA)* 19, 1 (2014). → 9

[92] H. Kaplan and N. Shafrir. 2005. The greedy algorithm for shortest superstrings. *Information Processing Letters (IPL)* 93, 1 (2005), 13–17. → 104

[93] Brian W. Kernighan. 1988. *The C Programming Language* (2nd ed.). Prentice Hall Professional Technical Reference. → 10

[94] Reinhard Kneser and Hermann Ney. 1995. Improved backing-off for m-gram language modeling. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Vol. 1. 181–184. → 151 and 153

[95] Philipp Koehn. 2005. Europarl: A parallel corpus for statistical machine translation, `http://www.statmt.org/europarl`. In *MT summit*. 79–86. → 138

[96] Grzegorz Kondrak. 2005. N-gram similarity and distance. In *International symposium on string processing and information retrieval (SPIRE)*. Springer, 115–126. → 31

[97] Karen Kukich. 1992. Techniques for automatically correcting words in text. In *ACM Computing Surveys (CSUR)*. 377–439. → 31

[98] Hoang Thanh Lam, Raffaele Perego, Nguyen Thoi Minh Quan, and Fabrizio Silvestri. 2009. Entry Pairing in Inverted File. In *Proceedings of the 10th International Conference Web Information Systems Engineering (WISE)*. 511–522. → 54

[99] N. Jesper Larsson and Alistair Moffat. 1999. Offline Dictionary-Based Compression. In *Data Compression Conference (DCC)*. 296–305. → 102

[100] Daniel Lemire and Leonid Boytsov. 2013. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45, 1 (2013), 1–29. → 17 and 18

[101] Daniel Lemire, Nathan Kurz, and Christoph Rupp. 2018. Stream-VByte: faster byte-oriented integer compression. *Inform. Process. Lett.* 130 (2018), 1–6. → 16 and 17

[102] Ted G. Lewis and Curtis R. Cook. 1988. Hashing for dynamic and static internal tables. In *Computer*. 45–56. → 33

[103] K.-H. Li. 1994. Reservoir-sampling algorithms of time complexity $O(n(1 + \log(N/n)))$. *ACM Transactions on Mathematical Software (TOMS)* 20, 4 (1994), 481–493. → 100

[104] K. Liao, M. Petri, A. Moffat, and A. Wirth. 2016. Effective construction of Relative Lempel-Ziv dictionaries. In *International Conference on World Wide Web*. 807–816. → 97

[105] Yuri Lin, Jean-Baptiste Michel, Erez Lieberman Aiden, Jon Orwant, Will Brockman, and Slav Petrov. 2012. Syntactic annotations for the google books ngram corpus. In *Association for Computational Linguistics (ACL)*. 169–174. → 32

[106] Peter Lindstrom. 2014. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 2674–2683. → 183

[107] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE Transactions on Information Theory (IT)* 28 (1982), 129–137. → 55

[108] Veli Mäkinen and Gonzalo Navarro. 2007. Rank and select revisited and extended. *Theoretical Computer Science (TCS)* 387, 3 (2007), 332–347. → 20 and 35

[109] Christopher Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press. → 26, 27, and 62

[110] M. Martinez, M. Haurilet, R. Stiefelhagen, and J. Serra-Sagristà. 2017. Marlin: A high throughput variable-to-fixed codec using plurally parsable dictionaries. In *Data Compression Conference (DCC)*. 161–170. → 96

[111] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013), 12. → 183

[112] Bhaskar Mitra and Nick Craswell. 2015. Query auto-completion for rare prefixes. In *International Conference on Information and Knowledge Management (CIKM)*. 1755–1758. → 31

[113] Bhaskar Mitra, Milad Shokouhi, Filip Radlinski, and Katja Hofmann. 2014. On user interactions with query auto-completion. In *International Conference on Research and Development in Information Retrieval (SIGIR)*. 1055–1058. → 31

[114] Alistair Moffat and Matthias Petri. 2017. ANS-Based Index Compression. In *International ACM Conference on Information and Knowledge Management (CIKM)*. 677–686. → 106

[115] Alistair Moffat and Lang Stuiver. 2000. Binary Interpolative Coding for Effective Index Compression. *Information Retrieval Journal* 3, 1 (2000), 25–47. → 24, 75, and 135

[116] Donald R. Morrison. 1968. PATRICIA: practical algorithm to retrieve information coded in alphanumeric. In *Journal of the ACM (JACM)*. 514–534. → 128

[117] Gonzalo Navarro, Ricardo A. Baeza-Yates, Erkki Sutinen, and Jorma Tarhio. 2001. Indexing methods for approximate string matching. In *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*. 19–27. → 128

[118] Gonzalo Navarro and Yakov Nekrich. 2013. Optimal Dynamic Sequence Representations. In *Proceedings of the Twenty-Fourth Annual Symposium on Discrete Algorithms (SODA)*. 865–876. → 39 and 44

[119] Patrick Nguyen, Jianfeng Gao, and Milind Mahajan. 2007. MSRLM: a scalable language modeling toolkit. In *Microsoft Research MSR-TR-2007-144.2007*. 19. → 151 and 154

[120] Giuseppe Ottaviano and Rossano Venturini. 2014. Partitioned Elias-Fano Indexes. In *Proceedings of the 37th International Conference on Research and Development in Information Retrieval (SIGIR)*. 273–282. → 22, 23, 24, 53, 64, 75, 78, 79, 91, 114, 129, 135, 140, and 141

[121] Athanasios Papoulis. 1991. *Probability, Random Variables, and Stochastic Processes (3. ed.)*. McGraw-Hill. → 13

[122] Adam Pauls and Dan Klein. 2011. Faster and Smaller N-gram Language Models. In *Association for Computational Linguistics (ACL)*. 258–267. → 31, 125, 126, 127, 128, 137, 138, 151, 153, 154, and 167

[123] Dan Pelleg and Andrew Moore. 2000. X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In *Proceedings of the 17th International Conference on Machine Learning (ICML)*. 727–734. → 55 and 56

[124] Marco Pellegrini and Giordano Fusco. 2004. Efficient IP table lookup via adaptive stratified trees with selective reconstructions. In *European Symposium on Algorithms*. Springer, 772–783. → 182

[125] Giulio Ermanno Pibiri, Matthias Petri, and Alistair Moffat. 2019. Fast Dictionary-Based Compression for Inverted Indexes. In *International ACM Conference on Web Search and Data Mining (WSDM)*. 9. → 6

[126] Giulio Ermanno Pibiri and Rossano Venturini. 2017. Clustered Elias-Fano indexes. *ACM Transactions on Information Systems (TOIS)* 36, 1, Article 2 (2017), 33 pages. → 5

[127] Giulio Ermanno Pibiri and Rossano Venturini. 2017. Dynamic Elias-Fano Representation. In *Annual International Symposium on Combinatorial Pattern Matching (CPM)*. 30:1–30:14. → 5

[128] Giulio Ermanno Pibiri and Rossano Venturini. 2017. Efficient Data Structures for Massive N-Gram Datasets. In *International ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 615–624. → 7

[129] Giulio Ermanno Pibiri and Rossano Venturini. 2018. Inverted Index Compression. *Encyclopedia of Big Data Technologies* (2018), 1–8. → 14 and 75

[130] Giulio Ermanno Pibiri and Rossano Venturini. 2018. Variable-Byte Encoding is Now Space-Efficient Too. *ArXiv e-prints* (2018), 14. arXiv:cs.IR/1804.10949 → 6

[131] Giulio Ermanno Pibiri and Rossano Venturini. 2019. Handling Massive N-Gram Datasets Efficienctly. *ACM Transactions on Information Systems (TOIS). To appear.* (2019), 42. → 8

[132] Jeff Plaisance, Nathan Kurz, and Daniel Lemire. 2015. Vectorized VByte Decoding. In *International Symposium on Web Algorithms (iSWAG)*. 7. → 16 and 90

[133] Nicola Prezza. 2017. A Framework of Dynamic Data Structures for String Processing. In *International Symposium on Experimental Algorithms (SEA)*. 11:1–11:15. → 183

[134] Mihai Pătraşcu and Mikkel Thorup. 2006. Time-space trade-offs for predecessor search. In *Proceedings of the 38-th Annual Symposium on Theory of Computing (STOC)*. 232–240. → 5, 36, 37, 38, 43, and 182

[135] Mihai Pătraşcu and Mikkel Thorup. 2007. Randomization does not help searching predecessors. In *Proceedings of the 18-th Annual Symposium on Discrete Algorithms (SODA)*. 555–564. → 37

[136] Mihai Pătraşcu and Mikkel Thorup. 2014. Dynamic Integer Sets with Optimal Rank, Select, and Predecessor Search. In *Proceedings of the 55-th Annual Symposium on Foundations of Computer Science (FOCS)*. 166–175. → 5, 35, 36, and 38

[137] Bhiksha Raj and Ed Whittaker. 2003. Lossless Compression of Language Model Structure and Word Identifiers. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. 388–391. → 127

[138] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. 2001. Succinct Dynamic Data Structures. In *Proceedings of the 7-th International Workshop on Algorithms and Data Structures (WADS)*. 426–437. → 38, 46, 48, and 49

[139] Vijayshankar Raman, Lin Qiao, Wei Han, Inderpal Narang, Ying-Lin Chen, Kou-Horng Yang, and Fen-Ling Ling. 2007. Lazy, adaptive rid-list intersection, and its application to index anding. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, 773–784. → 27

[140] Robert Rice and J. Plaunt. 1971. Adaptive Variable-Length Coding for Efficient Compression of Spacecraft Television Data. *IEEE Transactions on Communications* 16, 9 (1971), 889–897. → 15

[141] Stephen Robertson and Sparck Jones. 1976. Relevance weighting of search terms. *Journal of the American Society for Information Science* 27, 3 (1976), 129–146. → 28

[142] Kunihiko Sadakane and Roberto Grossi. 2006. Squeezing succinct data structures into entropy bounds. In *Proceedings of the 17-th Annual Symposium on Discrete Algorithms (SODA)*. 1230–1239. → 35

[143] David Salomon. 2007. *Variable-length Codes for Data Compression*. Springer. → 128

[144] Jangwon Seo and W. Bruce Croft. 2008. Local text reuse detection. In *International Conference on Research and Development in Information Retrieval (SIGIR)*. 571–578. → 31

[145] Claude Elwood Shannon. 2001. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review* 5, 1 (2001), 3–55. → 13

[146] Ehsan Shareghi, Matthias Petri, Gholamreza Haffari, and Trevor Cohn. 2015. Compact, efficient and unlimited capacity: Language modeling with compressed suffix trees. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2409–2418. → 128, 151, and 155

[147] Ehsan Shareghi, Matthias Petri, Gholamreza Haffari, and Trevor Cohn. 2016. Fast, Small and Exact: Infinite-order Language Modelling with Compressed Suffix Trees. *Transactions of the Association of Computational Linguistics (TACL)* 4, 1 (2016), 477–490. → 128, 151, 155, and 171

[148] Fabrizio Silvestri. 2007. Sorting Out the Document Identifier Assignment Problem. In *Proceedings of the 29th European Conference on IR Research (ECIR)*. 101–112. → 7 and 30

[149] Fabrizio Silvestri and Rossano Venturini. 2010. VSEncoding: Efficient Coding and Fast Decoding of Integer Lists via Dynamic Programming. In *Proceedings of the 19th International Conference on Information and Knowledge Management (CIKM)*. 1219–1228. → 17, 25, and 78

[150] Michael Steinbach, George Karypis, and Vipin Kumar. 2000. A comparison of document clustering techniques. In *6th Annual Conference on Knowledge Discovery and Data Mining (KDD), Workshop on Text Mining*. 109–111. → 55, 56, and 61

[151] Alexander Stepanov, Anil Gangolli, Daniel Rose, Ryan Ernst, and Paramjit Oberoi. 2011. SIMD-based decoding of posting lists. In *Proceedings of the 20th International Conference on Information and Knowledge Management (CIKM)*. 317–326. → 16 and 75

[152] Andreas Stolcke. 2002. SRILM - an extensible language modeling toolkit. In *International Conference on Spoken Language Processing (ICSLP)*. 901–904. → 151 and 154

[153] Muhammad Tahir and Shakil Ahmed. 2015. Tree-Combined Trie: A Compressed Data Structure for Fast IP Address Lookup. *International Journal of Advanced Computer Science & Applications* 1, 6 (2015), 168–175. → 182

[154] David Talbot and Miles Osborne. 2007. Randomised language modelling for statistical machine translation. In *Association for Computational Linguistics (ACL)*. 512–519. → 125, 127, 139, and 151

[155] Larry H Thiel and HS Heaps. 1972. Program design for retrospective searches on large data bases. *Information Storage and Retrieval (ISR)* 8, 1 (1972), 1–20. → 15, 75, and 128

[156] Andrew Trotman. 2014. Compression, SIMD, and postings lists. In *Proceedings of the 2014 Australasian Document Computing Symposium*. ACM, 50. → 17

[157] Peter van Emde Boas. 1975. Preserving Order in a Forest in less than Logarithmic Time. In *Proceedings of the 16-th Annual Symposium on Foundations of Computer Science (FOCS)*. 75–84. → 35 and 37

[158] Peter van Emde Boas. 1977. Preserving Order in a Forest in Less Than Logarithmic Time and Linear Space. *Information Processing Letters (IPL)* 6, 3 (1977), 80–82. → 35 and 37

[159] Peter van Emde Boas, Robert Kaas, and Erik Zijlstra. 1977. Design and Implementation of an Efficient Priority Queue. *Mathematical Systems Theory (MST)* 10 (1977), 99–127. → 35 and 37

[160] Sebastiano Vigna. 2008. Broadword implementation of rank/select queries. In *Workshop on Experimental Algorithms (WEA)*. 154–168. → 13

[161] Sebastiano Vigna. 2013. Quasi-succinct indices. In *Proceedings of the 6th ACM International Conference on Web Search and Data Mining (WSDM)*. 83–92. → 20, 22, and 66

[162] J. S. Vitter. 1985. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)* 11, 1 (1985), 37–57. → 100

[163] Jeffrey Scott Vitter. 1998. External memory algorithms. In *European Symposium on Algorithms (ESA)*. 1–25. → 174

[164] Taro Watanabe, Hajime Tsukada, and Hideki Isozaki. 2009. A succinct n-gram language model. In *International Joint Conference on Natural Language Processing (IJCNLP)*. 341–344. → 127, 139, 151, and 154

[165] Dan E. Willard. 1983. Log-Logarithmic Worst-Case Range Queries are Possible in Space $\Theta(N)$. *Information Processing Letters (IPL)* 17, 2 (1983), 81–84. → 35 and 37

[166] Hugh E Williams and Justin Zobel. 1999. Compressing integers for fast file access. *Comput. J.* 42, 3 (1999), 193–201. → 75

[167] Niklaus Wirth. 1995. A plea for lean software. *Computer* (1995), 64–68. → 1

[168] Ian Witten, Alistair Moffat, and Timothy Bell. 1999. *Managing gigabytes: compressing and indexing documents and images* (2nd ed.). Morgan Kaufmann. → 25 and 176

[169] Rui Xu and Donald Wunsch. 2005. Survey of Clustering Algorithms. *IEEE Transactions on Neural Networks (NN)* 16, 3 (2005), 645–678. → 55 and 61

[170] Hao Yan, Shuai Ding, and Torsten Suel. 2009. Inverted index compression and query processing with optimized document ordering. In *International Conference on World Wide Web (WWW)*. 401–410. → 18, 25, and 55

[171] Andrew Chi-Chih Yao. 1981. Should Tables Be Sorted? *Journal of the ACM (JACM)* 28, 3 (1981), 615–628. → 37 and 38

[172] Susumu Yata. 2011. Prefix/Patricia trie dictionary compression by nesting Prefix/Patricia tries. In *International Conference on Natural Language Processing (NLP)*. → 128 and 139

[173] J. Zhang, X. Long, and T. Suel. 2008. Performance of compressed inverted list caching in search engines. In *International Conference on World Wide Web (WWW)*. 387–396. → 17

[174] Zhaohua Zhang, Jiancong Tong, Haibing Huang, Jin Liang, Tianlong Li, Rebecca J. Stones, Gang Wang, and Xiaoguang Liu. 2016. Leveraging Context-Free Grammar for Efficient Inverted Index Compression. In *Proceedings of the 39th International Conference on Research and Development in Information Retrieval (SIGIR)*. 275–284. → 55 and 97

[175] Justin Zobel and Alistair Moffat. 2006. Inverted files for text search engines. *ACM Computing Surveys (CSUR)* 38, 2 (2006), 1–56. → 26 and 27

[176] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*. 59–70. → 18