
UNIVERSITÀ DI PISA

Department of Computer Science

Ph.D. Thesis

# Learning-based compressed data structures

Giorgio Vinciguerra
giorgio.vinciguerra@phd.unipi.it

*Supervisor*
Paolo Ferragina

*Internal committee*
Luca Oneto
Università di Genova

Salvatore Ruggieri
Università di Pisa

*External referees*
Stratos Idreos
Harvard University

Gonzalo Navarro
Universidad de Chile

October 31, 2021

# Abstract

This thesis revisits two fundamental problems in data structure design: predecessor search and rank/select primitives. These problems are pervasive in applications, particularly in areas such as database systems, search engines, bioinformatics, and Internet routing. We show that real data present a peculiar kind of regularity that can be explained in terms of geometric considerations. We name it "approximate linearity" and analyse its algorithmic effectiveness in a variety of possible input data distributions. We then expand the horizon of compressed data structures by presenting solutions for the problems above that discover, or "learn", in a rigorous and efficient algorithmic way, the approximate linearities present in the data. In addition, we show how to combine this new form of compressibility with the classic repetition-aware approaches thus introducing a new class of compressed indexes. We accompany our theoretical results with implementations and experiments on large amounts of data, and we show that, compared to several well-engineered known compressed indexes, our data structures provide improvements in time, in space or both (often of orders of magnitude).

# Acknowledgements

I am extremely grateful to my supervisor, Paolo Ferragina, for his continuous and inspiring guidance throughout this PhD journey. Looking back on the countless opportunities, brilliant insights, and the scientific and personal advice Paolo gave me, I can easily say that he has been the best (and principled! to use one of his favourite words) mentor I could have hoped for.

I am also profoundly grateful to my amazing co-authors Antonio Boffa, Fabrizio Lillo, and Giovanni Manzini, without whom many of the results presented here would not have been possible.

My sincere thanks to the external referees, Professors Stratos Idreos and Gonzalo Navarro, for their timely and expert feedback, and to the internal committee members, Professors Luca Oneto and Salvatore Ruggieri, for their encouraging and helpful comments over the yearly progress evaluations.

A huge thank you to my friends and colleagues at the Department for the nice and fun time I spent with them. Among others, I want to thank Benedikt, Federico, Francesco, Irene, Jacopo, Mattia, Stefano, and Veronica.

Finally, I would also like to express my gratitude to my family and friends for their constant warmth and support. *Un ringraziamento speciale a Nonna Quinta, che si è sempre assicurata, anche a distanza, che lavorassi a questa tesi a stomaco pieno.*

# Publications by the author

## Accepted


[FV20a] Paolo Ferragina and Giorgio Vinciguerra. “Learned data structures”. In: *Recent Trends in Learning From Data*. Ed. by Luca Oneto, Nicolò Navarin, Alessandro Sperduti and Davide Anguita. Springer, 2020, pp. 5–41.

[FV20b] Paolo Ferragina and Giorgio Vinciguerra. “The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds”. In: *PVLDB* 13.8 (2020), pp. 1162–1175.

[FLV20] Paolo Ferragina, Fabrizio Lillo and Giorgio Vinciguerra. “Why are learned indexes so effective?” In: *Proceedings of the 37th International Conference on Machine Learning (ICML)*. Vol. 119. 2020, pp. 3123–3132.

[FLV21] Paolo Ferragina, Fabrizio Lillo and Giorgio Vinciguerra. “On the performance of learned data structures”. In: *Theoretical Computer Science* 871 (2021), pp. 107–120.

[BFV21a] Antonio Boffa, Paolo Ferragina and Giorgio Vinciguerra. “A “learned” approach to quicken and compress rank/select dictionaries”. In: *Proceedings of the 23rd SIAM Symposium on Algorithm Engineering and Experiments (ALENEX)*. 2021, pp. 46–59.

[FMV21] Paolo Ferragina, Giovanni Manzini and Giorgio Vinciguerra. “Repetition- and linearity-aware rank/select dictionaries”. In: *Proceedings of the 32nd International Symposium on Algorithms and Computation (ISAAC)*. 2021.


## Submitted


[BFV21b] Antonio Boffa, Paolo Ferragina and Giorgio Vinciguerra. “A learned approach to design compressed rank/select data structures”. Submitted to ACM Transactions on Algorithms. 2021.

[MVF21] Giovanni Manzini, Giorgio Vinciguerra and Paolo Ferragina. “Procedimento di compressione e ricerca su un insieme di dati basato su strategie multiple”. (Italy). May 2021. Patent application filed.

# Contents

# 1 Introduction

Representing data in a computing machine and supporting efficient searches is among the oldest and most prominent class of problems in computer science, well-studied and ubiquitous in research and applications. Not surprisingly, it is often used as an introductory topic in basic algorithms courses, paving the way to the study of fundamental data structures such as arrays, lists, search trees, tries and hash tables.

These and other fundamental data structures are often sufficient to solve any given computational problem that we might encounter in practice, from a programming puzzle asked in a coding interview to a real-world application. Yet, it would be amiss to claim that an algorithm designer's toolbox is comprehensive with these design elements alone, especially considering the vast amount of facets an instance of a searching problem can exhibit (computational constraints, data volume growth over time, hardware technology, ...).

The advent of external memory devices and hierarchical memories, for example, allowed us to deal with massive amounts of data and, at the same time, it challenged the data structures design field up to one of its core assumptions, i.e. that all memory accesses are created equal.

This eventually lead to the introduction of new models of computation, such as the external-memory and the cache-oblivious model [Vit01; Fri+12]. These models enforce the design of algorithms that exploit the locality of memory accesses thereby reducing the time-consuming input/output communication (I/O) between memory levels, and they have been applied to searching as well as some other problems, such as sorting, matrix multiplications, and graph problems [Vit01].

In some situations, however, loading the hierarchical memory with more and more data is not possible either because of physical capacity constraints (think of smart and embedded devices), or because it would slow down the searches too much due to the increased I/Os needed to handle a larger problem size (think of the latency requirements a Web search engine must guarantee to its users).

In the 1990s, the field of *compact data structures* emerged (encompassing the so-called succinct, compressed or opportunistic data structures) [Nav16]. These data

structures represent data in as little space as possible while still providing efficient query operations. That is, they allow efficient searches without fully decompressing the data, as a sheer compression approach would instead require. This way, we can fit larger volumes of data in memory, where the computation is much faster.

Today, it is known how to turn almost any traditional data structure into a compact data structure that exploits some source of compressibility in the input data. In this regard, we can distinguish between a *statistical* source and a *repetitive* source. Roughly speaking, for the former, we assigning shorter descriptions to the most frequent symbols in the input data (possibly looking at the symbol's preceding context), and we relate the compact data structure size to its Shannon entropy [CT06]. For the latter, we replace repetitive subsequences in the input data with references to an existing entry in a small and properly-built dictionary, and we relate the compact data structure size to one of the multitude repetitiveness measures [Nav20].

In this thesis, we revisit some fundamental and ubiquitous data structuring problems, namely predecessor search and rank/select primitives, in light of a new kind of data regularity that we call *approximate linearity*. This regularity manifests itself when we map the input data into points in a specially-crafted Cartesian plane [Ao+11].

The first key idea underpinning our approaches to data structure design is to "learn" the distribution of the above points via error-bounded linear functions, computed via a classical result in computational geometry [ORo81]. These linear functions form a lossy and succinct learned representation of the input distribution. Intuitively, the more the input data exhibits roughly linear trends (i.e. the more the data is approximately linear), the more compressed such learned representation is.

We formalise the above intuition by studying the effectiveness and the power of approximate linearity under several theoretical and practical settings. Then, by orchestrating the learned representation with proper algorithms and data structures, we expand the horizon of compressed data structures with novel solutions that provide new and improved space-time trade-offs for the problems above. Furthermore, we also study combinations of our approaches with the ones based on statistical and repetitive sources, and we show that they complement and integrate each other nicely. Our experimental achievements corroborate these theoretical results by showing improvements in time, in space or both (often of orders of magnitude) compared to several well-engineered data structures implementations, making them promising and possibly impactful approaches also in real applications.

We conclude by discussing the plethora of research opportunities that these new learning-based approaches to data structure design open up.

## 1.1 Thesis structure

The rest of this thesis, which builds upon the author's publications listed at page vii, is structured as follows.

**Chapter 2: Basics**

We introduce and discuss the main ingredients that will be used throughout the thesis, namely the mapping of the input data to a Cartesian plane, and the concept and computation of piecewise linear $\varepsilon$-approximations. We accompany the exposition with a motivating searching problem that is pervasive in computer science.

**Chapter 3: Effectiveness of piecewise linear $\varepsilon$-approximations**

We study the power of the concept of approximate linearity by analysing the space usage of piecewise linear $\varepsilon$-approximations built on a wide range of possible input data distributions. We show that piecewise linear $\varepsilon$-approximations are indeed succinct and effective, and thus they form a strong basis for the learning-based data structure design that will be introduced in the following. This chapter is based in part on [FLV20; FLV21].

**Chapter 4: Predecessor search**

We dig into a generalisation of the searching problem introduced in Chapter 2, the so-called predecessor search problem. This problem has several critical applications, such as range queries in database systems, conjunctive queries in search engines, and the routing of Internet packets. We introduce the PGM-index, an external-memory learning-based data structure for this problem with optimal worst case bounds. We make it compressed, dynamic and adaptive not only to the approximate linearity in the data but also to a given query distribution. A wide set of experiments on large datasets show that our solution improves some efficient predecessor structure implementations in time and in space, sometimes by orders of magnitude. This chapter is based in part on [FV20b].

**Chapter 5: Rank/select dictionaries**

We focus on the problem of representing a compressed dictionary of integers while supporting rank and select operations. This well-studied problem is at the heart of virtually any compact data structure, and we revisit it by proposing a learning-based rank/select scheme, named LA-vector, that achieves competitive compressed space occupancy and efficient query operations by exploiting the approximate linearity in the data. A comparison of this approach with some other well-engineered compressed rank/select dictionaries shows new space-time trade-offs. This chapter is based in part on [BFV21a; BFV21b].

**Chapter 6: Repetition- and linearity-aware rank/select dictionaries**

We continue our study on rank/select dictionaries by showing how to combine two sources of compressibility: approximate linearity and repetitiveness in the data. We do so by building on two repetitiveness-aware methods, one more space-efficient and the other more query-efficient, namely Lempel-Ziv parsings [LZ76; ZL77; KN13] and block trees [Bel+21], respectively. We empower these two methods with the ability to exploit approximate linearities and thus introduce two solutions, the $\mathrm{LZ}_\varepsilon^\rho$ and the block-$\varepsilon$ tree. Our experimental achievements show that the combination of the two sources of compressibility is effective and achieves the best of both worlds. This chapter is based in part on [FMV21].

**Chapter 7: Conclusions**

We summarise the findings of this thesis and discuss the research opportunities that these new learning-based approaches to compressed data structure design open up.

# 2 Basics

In this chapter, we introduce the main design ingredients that will be used throughout the thesis.

As our motivating application, we consider one of the most basic and pervasive problems in computer science, the *searching problem*, formulated as follows. Suppose we are given a sorted array $A[1, n]$ of $n$ distinct keys drawn from a universe $\mathcal{U}$. Given a query element $x \in \mathcal{U}$, we are asked to return an index $i$ such that $x = A[i]$ or, if $x$ is not a key of $A$, the index $j$ after which $x$ could be inserted to keep $A$ sorted, i.e. the $j$ such that $A[j] < x < A[j + 1]$.

The elementary binary search algorithm is sufficient to solve the searching problem in $\Theta(\log n)$ comparisons, and it is also optimal if we consider a computational model that only allows comparing atomic items with a cost of 1, i.e. the *comparison model*. Yet, with the large set of operations available in today's processors, it is customary to consider less restrictive computational models.

One of the computational models that we use in this thesis is the *word RAM* [Hag98]. The word RAM model is assumed to have an infinite memory consisting of cells of $w$ bits, where $w$ is a positive integer called the *word length*. Consequently, the content of any memory cell is an integer $x$ in the range $\{0, \ldots, 2^w - 1\}$, often called a *word*. Given two words, we can execute arithmetic instructions that compute their sum, difference, product, and quotient. Moreover, we can execute Boolean instructions that compute the conjunction (AND), and disjunction (OR) of two words, and the negation (NOT) of a word. The execution of every instruction is assumed to take constant time. Then, we can measure the time complexity of an algorithm by counting how many memory accesses and instructions are executed, and we can measure the space complexity by counting how many words or bits are used by the algorithm. We observe, for when later in this chapter we show examples with non-integer data, that the above integer instructions are sufficient to implement IEEE 754 floating-point data types and operations [PH20, §3.5].

Under the word RAM and other computational models, plenty of algorithms and data structures were shown to break the $\Theta(\log n)$-barrier of comparison-based methods for the searching problem. We will review some of them in Chapter 4, in which we

dig into the closely related *predecessor search problem*. Here, instead, we want to give a different and possibly unusual perspective on the searching problem by showing how it reduces to approximating a set of points in a specially-crafted Cartesian plane [Ao+11].

## 2.1 A geometric reduction

The first ingredient to solve the searching problem consists in mapping each key $A[i]$ of the input array $A$ to a point $(A[i], i)$ in the Cartesian plane of keys-positions, for $i = 1, \ldots, n$. The second ingredient is a function $f$ that "best fits" the points in such plane, namely, a function that given a query element $x \in \mathcal{U}$ returns the appropriate position of $x$ in $A$, i.e. the answer to our searching problem.

Unsurprisingly, the best function $f$ that we can hope to find is the function $\mathsf{rank}\colon \mathcal{U} \to \{0, \ldots, n\}$ that, for any $x \in \mathcal{U}$, returns the number $\mathsf{rank}(x)$ of keys in $A$ that are less than or equal to $x$. But implementing $\mathsf{rank}$ in the word RAM in a space/time-efficient manner is simply another way of restating the searching problem. Thus, up to now, our reduction seems of no use.

The reduction becomes interesting when we allow $f$ to make errors, that is, to slightly under- or overestimate the position of $x$ in $A$. This opens up a whole range of possibilities for implementing $f$, not only using *data structure design elements* (such as nodes, pointers, hashing, samplings, etc.), but also using tools from *numerical analysis* and *machine learning* (such as linear models, polynomials, neural networks, regression trees, etc).

For example, suppose that $A = [7, 16, 17, 18, 19, 20, 29, 54, 57, 60]$, and that the function that "best fits" these points is the one computed using ordinary least squares on the dataset $\{(A[i], i)\}_{i=1,\ldots,n}$, i.e. $f(x) = 0.14x + 1.29$, as depicted in Figure 2.1. For an input query $x = 29$, we can use $f$ to solve the searching problem by computing the approximate position $p = \lfloor f(x) \rceil = \lfloor 29 \times 0.14 + 1.29 \rceil = 5$.[1] But the true position of $x$ is 7, hence, if we return 7 as the answer to the query, we make an error $err = 2$. The twist here is that, since $A$ is available, the error can always be fixed. Indeed, a simple and fast scan from $p = 5$ forward let us find the sought key $x$ in position 7, which is the answer to the query.

More in general, let us define the *maximum error* of $f$ as the value

$$err = \max_{x \in \mathcal{U}} |\lfloor f(x) \rceil - \mathsf{rank}(x)|. \tag{2.1}$$

[1] We use $\lfloor x \rceil$ to denote the nearest integer function, i.e. $\lfloor x \rceil = \lfloor x + \frac{1}{2} \rfloor$.

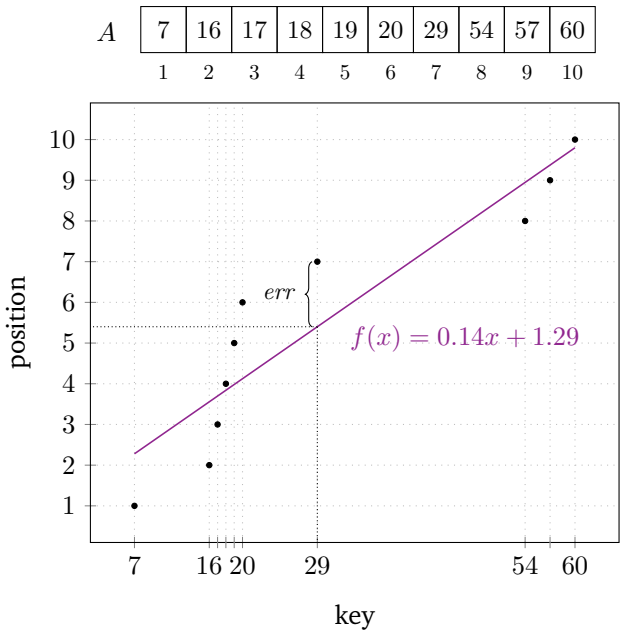



**Figure 2.1:** An array $A$ of ten keys mapped to the key-position Cartesian plane. The linear model $f$, computed using ordinary least squares on the points, estimates that $x = 29$ is in position $p = \lfloor f(x) \rceil = 5$, but the true position of $x$ is 7.

Then, our data structure for the searching problem is given by the value $err$ and the parameters of $f$, and the search algorithm amounts to compute $p = \lfloor f(x) \rceil$ and to perform a binary search on $A[p - err, p + err]$ for the sought value $x$. If $f$ takes $s_f$ space to be stored and $t_f$ time to be evaluated, then our data structure solves the searching problem in $\mathcal{O}(t_f + \log err)$ time and $s_f$ space.

Continuing with our example of a linear function $f$, we can easily verify that $err = 2$, $s_f = 2$ words, as we need to store the slope and the intercept of $f$, while $t_f = \mathcal{O}(1)$ time, as computing $f(x)$ requires one multiplication and one addition, which both take constant time in the word RAM model defined before. The search time is thus $\mathcal{O}(\log err)$, which is independent of $n$.

## 2.2 What function to use?

From the bounds above it is evident that a crucial design choice is how to implement $f$. As the data grows and exhibits irregular trends in the key-position Cartesian plane, the choice of a linear $f$ would quickly become inadequate due to the large errors that impact negatively on the $\mathcal{O}(\log err)$-term of the search time complexity. As a remedy, we could use, say, a deep neural network $f$ that was finely tuned to yield a very small $err$. But, on the other hand, we may end up paying too much

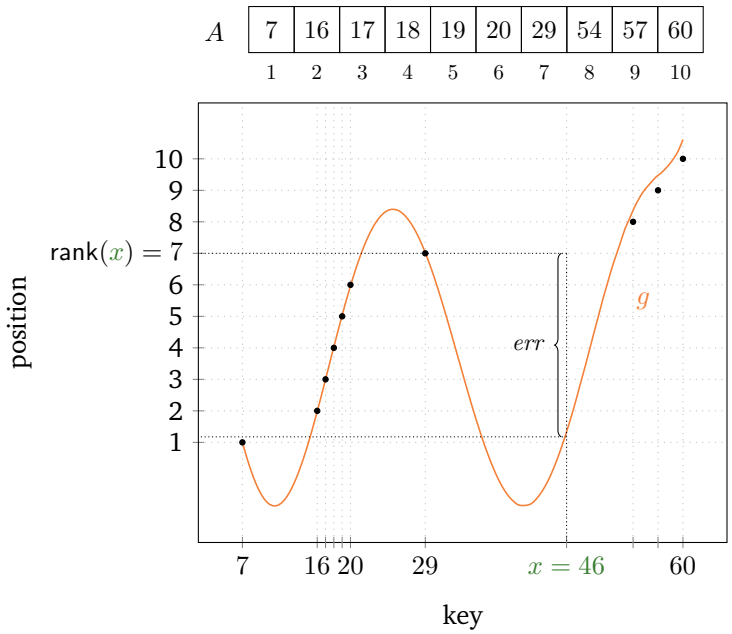



**Figure 2.2:** The same array $A$ and the corresponding key-position Cartesian plane of Figure 2.1 where instead of $f$ we use a polynomial function $g$. The function $g$ incurs in a large error when approximating the position of a query key $x$ that belongs to $\mathcal{U}$ but does not occur in $A$.

space $s_t$ for the weights of the neural network and too much time $t_f$ for the matrix multiplications needed when evaluating $f(x)$.

Another key point that we must consider when choosing $f$ is that the $\max$ in the definition of $err$ (Equation 2.1) ranges over all $x \in \mathcal{U}$ and not over all $x \in A$. This is necessary because otherwise we would not be able to provide correct answers when the query key does not occur in $A$. And in general, although for the example in the previous section the linear function $f$ was computed via ordinary least squares on $\mathcal{D} = \{(A[i], i)\}_{i=1,\ldots,n}$, i.e. without caring about the values in $\mathcal{U} \setminus A$, choosing a function giving a small error on these latter values is equally important. For instance, say that we fit a polynomial function $g$ of degree seven on $\mathcal{D}$, as shown in Figure 2.2. The error over the keys in $A$ is equal to 0, so we could be led to believe that $g$ is a good function in that it takes little space and makes no error. But in reality, the error at the value $x = 46$, which does not belong to $A$, is quite larger than what we expected, leading to increased query times.

Overall, it seems that choosing a function $f$ and then evaluating whether its accuracy is good or not via Equation 2.1 is a time-consuming, trial-and-error and very dataset-specific task. And although this task can be automated (e.g., via a grid or a random search), we should aim for a general and robust enough solution that can handle any

kind of input data, no matter its amount or the (ir)regular trends it exhibits.

For the first step toward a more general solution, we observe that it is desirable to have a way to control the error beforehand, i.e. before the choice of $f$, via a positive integer parameter $\varepsilon$. This is because the $\mathcal{O}(\log err)$-term of the search time complexity would then become $\mathcal{O}(\log \varepsilon)$, which is independent of the accuracy of $f$ over the input data. And also because, in more practical terms, modern memory technologies all access data in *blocks* (also called *pages* or *lines*) of $bw$ bits, for a certain factor $b \geq 1$ [PH20, §5], and hence it is natural to ask that our $f$ outputs predictions that require just a few fetches of blocks to be corrected, i.e. $err = \varepsilon = \Theta(b)$.

Now, within a fixed error $\varepsilon$, we ought to find a function $f$ guaranteeing such error over all the keys in $\mathcal{U}$ (recall the definition of Equation 2.1). Although nonlinear functions are appealing, especially because they form an infinite class of functions where there is certainly one fitting the input data within the $\varepsilon$ error constraint, training them is challenging by itself [GBC16, §8.2], and subtleties like computing the error efficiently and keeping it low over keys in $\mathcal{U} \setminus A$ exacerbate the problem, especially over the large universes that arise in modern applications and computers having $w = 64$ bits.

We therefore backtrack to linear functions and, to solve their shortcomings on large and potentially irregular input data, we take a turn by generalising them to what are called *piecewise linear $\varepsilon$-approximations*.

## 2.3 Piecewise linear $\varepsilon$-approximations

We now introduce the concept of piecewise linear $\varepsilon$-approximation as a sequence of segments (linear functions) partitioning $A$ and approximating the rank function, i.e. the answer to the searching problem, within a given maximum error $\varepsilon$.

**Definition 2.1.** *For a given integer parameter $\varepsilon \geq 0$ and a sorted array $A[1, n]$ of $n$ keys drawn from a universe $\mathcal{U}$, a piecewise linear $\varepsilon$-approximation for $A$ is a partition of $A$ into subarrays such that each subarray $A[i, j]$ of the partition is covered by a* segment *$f_k$ such that $|f_k(x) - \mathsf{rank}(x)| \leq \varepsilon$ for each $A[i] \leq x \leq A[j]$.*

An example of piecewise linear $\varepsilon$-approximation, with $\varepsilon = 2$, is depicted in Figure 2.3. Its first segment covers $A[1, 6]$, and its second segment covers the remaining $A[7, 10]$. The storage of this piecewise linear $\varepsilon$-approximation includes the slope and the intercept of each of the two segments, and the abscissa $A[7]$ where the second

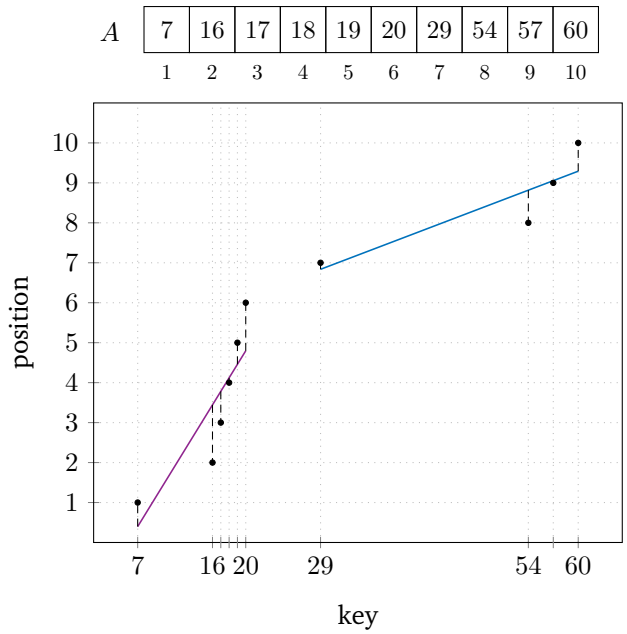



**Figure 2.3:** A piecewise linear $\varepsilon$-approximation, with $\varepsilon = 2$, composed of two segments for the same array $A$ of Figures 2.1 and 2.2.

segment starts. This way, given a search key $x$, depending on whether $x < A[7]$ or not, we retrieve the first or the second segment's parameters $\langle \alpha_j, \beta_j \rangle$, and we compute the approximate position of $x$ as $p = \lfloor f_j(x) \rfloor = \lfloor \alpha_j\, x + \beta_j \rfloor$. Finally, we run a binary search on $A[p - \varepsilon, p + \varepsilon]$ to find the answer $\mathsf{rank}(x)$ to the query.[2]

Among all the possible piecewise linear $\varepsilon$-approximations that one can construct on a given $A[1, n]$, we aim for *the most succinct one*, namely the one with the least amount of segments, or equivalently, the one that maximises the length $\ell$ of the subarray $A[i, i + \ell - 1]$ covered by a segment starting at $i$.

To this end, we recall a classical computational geometry problem that admits an optimal $\mathcal{O}(n)$-time algorithm.

**Lemma 2.1** (O'Rourke's algorithm [ORo81])**.** *Given a sequence of $n$ vertical ranges $(a_1, b_1), \ldots, (a_n, b_n)$ in the plane and corresponding abscissæ $x_1 < \cdots < x_n$, there exists an algorithm that in $\mathcal{O}(n)$ time finds all the linear functions $g(x) = \alpha x + \beta$ that pierce each vertical range, i.e. all pairs $\langle \alpha, \beta \rangle$ such that $a_i \leq g(x_i) \leq b_i$ for $i = 1, \ldots, n$.*

To construct a *minimum-sized* piecewise linear $\varepsilon$-approximation for $A$ with O'Rourke's algorithm, we simply set the input vertical ranges to $(1 + \varepsilon, 1 - \varepsilon), \ldots, (n + \varepsilon, n - \varepsilon)$

[2] As a technical note, to correctly approximate the position of the keys falling in-between a segment $j$ and the next (in the example, any key $k$ such that $A[6] = 20 < k < 29 = A[7]$), we should compute the position as $p = \min\{f_j(x), f_{j+1}(k_{j+1})\}$, where $k_{j+1}$ is the first key covered by the segment $j + 1$ (in the example, the key $k_{j+1} = A[7]$).

and the abscissæ to $A[1] < \cdots < A[n]$. The algorithm then processes each vertical range left-to-right while shrinking a convex polygon in the parameter space of slopes-intercepts. Any coordinate $(\alpha, \beta)$ inside the convex polygon represents a linear function with slope $\alpha$ and intercept $\beta$ that approximates with error $\varepsilon$ the current set of processed points of the kind $(A[i], i)$. As soon as the $i$th vertical range causes the polygon to be empty, a linear function $g$ can be chosen inside the previous polygon and written to the output, and a new polygon is started from the $i$th vertical range [ORo81].

**Theorem 2.1.** *Given a sorted array $A[1, n]$ and an integer $\varepsilon \geq 0$, there exists an algorithm that in $\mathcal{O}(n)$ time computes a piecewise linear $\varepsilon$-approximation for $A$ with the minimum number of segments.*

*Proof.* Pick a segment $g(x) = \alpha x + \beta$ covering the subarray $A[i, j]$ from the output of the procedure above, i.e. the output of O'Rourke's algorithm with input vertical ranges of the kind $(i + \varepsilon, i - \varepsilon)$ and corresponding abscissæ of the kind $x_i = A[i]$. By Lemma 2.1, we know that $g$ was chosen inside the convex polygon containing all the linear functions piercing the vertical ranges defined for $A[i, j]$, or equivalently, that $g$ is one of the functions maximising the length of the subarray $A[i, j]$ and is such that $|g(x) - \mathsf{rank}(x)| \leq \varepsilon$ for each $x = A[i], \ldots, A[j]$.

We are left with showing that there exists an $f$ (derived from $g$) such that $|f(x) - \mathsf{rank}(x)| \leq \varepsilon$ for any $A[i] \leq x < A[j]$ (the case of the values in $\mathcal{U} \setminus A$ occurring in-between two segments has already been dealt with in Footnote 2).

Without loss of generality, consider two consecutive input keys $a, b$ in $A[i, j]$ and pick an $x$ such that $a \leq x < b$. Observe that, for any such $x$, it holds $\mathsf{rank}(x) = \mathsf{rank}(a) = \mathsf{rank}(b) - 1$. Also, since the input vertical ranges are increasing, we can assume $\alpha \geq 0$, and thus that $g$ is a non-decreasing function.

We let $f(x) = \lfloor g(x) \rfloor$ and consider two cases. First, if $f(x) \geq \mathsf{rank}(x)$, then

$$\begin{aligned} |f(x) - \mathsf{rank}(x)| &= \lfloor g(x) \rfloor - \mathsf{rank}(x) \\ &\leq g(x) - \mathsf{rank}(x) \\ &< g(b) - \mathsf{rank}(x) \\ &= g(b) - \mathsf{rank}(b) + 1 \\ &\leq \varepsilon + 1. \end{aligned}$$

Therefore, we have $|f(x) - \mathsf{rank}(x)| < \varepsilon + 1$ and, since both sides of this inequality are integers, we can rewrite it as $|f(x) - \mathsf{rank}(x)| \leq \varepsilon$.

Second, if $f(x) < \mathsf{rank}(x)$, then

$$\begin{aligned} |f(x) - \mathsf{rank}(x)| &= \mathsf{rank}(x) - \lfloor g(x) \rfloor \\ &< \mathsf{rank}(x) - g(x) + 1 \\ &\leq \mathsf{rank}(x) - g(a) + 1 \\ &= \mathsf{rank}(a) - g(a) + 1 \\ &\leq \varepsilon + 1, \end{aligned}$$

and the same considerations of the previous case apply. Therefore, we conclude that, with the choice of $f(x) = \lfloor g(x) \rfloor$, it holds $|f(x) - \mathsf{rank}(x)| \leq \varepsilon$ for any $x$ such that $a \leq x < b$. □

In conclusion, we found an implementation of $f$ that is fast to construct, that takes little space (just three words per segment) and that computes an $\varepsilon$-approximation of the true position of a key *regardless* of the learnability of the pattern and trends in the input data. On the one hand, it reduces to a constant-space and -time solution for the searching problem when the data is roughly linear. On the other hand, it *adapts* to any nonlinearities by employing more than one segment, but no more than necessary thanks to its minimum-size guarantee.

## 2.4 Summary

We showed how one of the most basic problems in data structure design can be reframed as the problem of approximating, or "learning", the distribution of a set of $n$ points in a specially-crafted Cartesian plane via a function $f$. We discussed some possible implementations of $f$ along with their advantages and drawbacks. Finally, we introduced the concept of optimal piecewise linear $\varepsilon$-approximation, i.e. an implementation of $f$ via the most succinct sequence of segments (linear models) that guarantee a maximum user-given error $\varepsilon$, and we showed that it can be computed in optimal $\mathcal{O}(n)$ time thanks to a classical computational geometry algorithm.

# 3 Effectiveness of piecewise linear $\varepsilon$-approximations

Piecewise linear $\varepsilon$-approximations are the core component of some new space-efficient data structures that we will introduce in Chapters 4 and 5. To set the stage for these data structures, we devote this chapter to the study of the effectiveness of piecewise linear $\varepsilon$-approximations in terms of space usage.

Given that storing a piecewise linear $\varepsilon$-approximation amounts to store a constant amount of information for each segment — namely the slope, the intercept and possibly the abscissa/ordinate where the segment starts — we turn our attention to the equivalent task of counting the number of segments a piecewise linear $\varepsilon$-approximation is composed of. Moreover, for a fixed input array $A$ and a value $\varepsilon$, we shall focus only on *minimum-sized* piecewise linear $\varepsilon$-approximations, which, as we saw in the previous chapter, can be computed efficiently (Theorem 2.1).

In general, it is hard to predict or bound what will be the number of segments, as it heavily depends on the regularity of the input data and on the value of $\varepsilon$. For instance, consider the three situations depicted in Figure 3.1. The first plot shows an input array $A = [3, 6, 9, \ldots, 45]$ with a linear trend, which thus requires just one segment with slope $\alpha = 1/3$ and intercept $\beta = 0$. The remaining two plots show a less regular input array $A$ and the two corresponding piecewise linear $\varepsilon$-approximations for two values of $\varepsilon$, namely $\varepsilon_1 \gg \varepsilon_2$, giving two and four segments, respectively.

This notwithstanding, it is not too difficult to prove our first upper bound on the number of segments, which holds regardless of $A$ and $\varepsilon$.

**Lemma 3.1.** *For any given sorted array $A[1, n]$ and an integer $\varepsilon \geq 0$, the optimal piecewise linear $\varepsilon$-approximation for $A$ of Theorem 2.1 is composed of segments covering subarrays of length at least $2\varepsilon'$, where $\varepsilon' = \max\{\varepsilon, 1\}$. Consequently, the number of segments is at most $n/(2\varepsilon')$.*

*Proof.* We first examine the case $\varepsilon \neq 0$. For any chunk of $2\varepsilon$ consecutive keys $A[i], \ldots, A[i + 2\varepsilon - 1]$, consider the linear function $g$ with slope $\alpha = 0$ and intercept

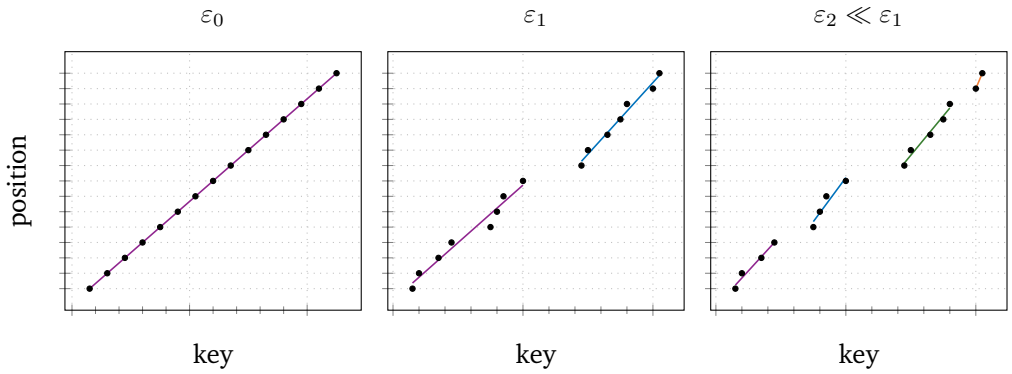



**Figure 3.1:** For the data on the left plot, only one segment is needed. On the remaining two plots, the input data is the same, but different values of $\varepsilon$ give a different number of segments in the piecewise linear $\varepsilon$-approximation.

$\beta = i + \varepsilon$. It is easy to see that $g$ is far at most $\varepsilon$ from the points $(A[i], i), \ldots,$ $(A[i + 2\varepsilon - 1], i + 2\varepsilon - 1)$ in the key-position Cartesian plane.

Then, since O'Rourke's algorithm (cf. Lemma 2.1) finds the set $\mathcal{G}$ of all functions piercing the vertical ranges of size $2\varepsilon$ derived from the keys in $A[i, i + 2\varepsilon - 1]$, it must be that $g \in \mathcal{G}$. In other words, the computation of any segment cannot stop before $2\varepsilon$ keys have been covered (except possibly due to the exhaustion of the input).

The case $\varepsilon = 0$ is trivial since any two consecutive keys admit a segment that connects them with no errors. □

The above lemma will be very useful in Chapters 4 and 5. Yet, we ought to observe that it is rather pessimistic. For instance, even for the simple case of linear trends that require just one segment like the one in the left plot of Figure 3.1, which nonetheless are not rare in real applications (e.g. an auto-incremented primary key column in a database table), Lemma 3.1 overestimates the number of segments as $n/2$. For this reason, it is customary to continue our study on the effectiveness of piecewise linear $\varepsilon$-approximations under some additional and possibly real-world assumptions on the distribution of the input data.

In the rest of this chapter, which is based on [FLV20; FLV21], we show that with some additional modelling and constraints on the input data (Section 3.1), the number of segments in a piecewise linear $\varepsilon$-approximation can be as little as $\mathcal{O}(n/\varepsilon^2)$ (Section 3.2). Then, we discuss the case in which input keys present some form of correlation (Section 3.3). Finally, we validate our claims with a large set of simulations and experiments on real and synthetic data (Section 3.4).

## 3.1 Preliminaries

We model the sorted input array $A$ as a stream of keys $x_0, x_1, \dots$ generating the gaps $g_1, g_2, \dots$ between keys, so that the $i$th input key is $x_i = \sum_{j=1}^{i} g_j$ (for convenience, we fix $x_0 = 0$). We assume that the sequence gaps $\{g_i\}_{i\in\mathbb{N}}$ is a realisation of a random process $\{G_i\}_{i\in\mathbb{N}}$, where the $G_i$s are positive, independent and identically distributed (iid) random variables with probability density function (pdf) $f_G$, mean $\mathbb{E}[G_i] = \mu$ and variance $\mathrm{Var}[G_i] = \sigma^2$. Then, we define the random variables modelling the cumulative sum as $X_i = \sum_{j=1}^{i} G_j$ (for $i = 1, 2, \dots$) and fix $X_0 = 0$.

In this setting, our problem is to find a linear model that approximates the points $(0,0), (X_1, 1), (X_2, 2), \dots$ in the Cartesian plane within a given maximum error $\varepsilon \geq 1$, measured along the $y$-axis.

Now, let us consider the two parallel lines $y = \alpha x \pm \varepsilon$, for an $\alpha$ to be chosen later, and the strip $\mathcal{S}$ of height $2\varepsilon$ between them, i.e. $\mathcal{S} = \{(x, y) \mid \alpha x - \varepsilon < y < \alpha x + \varepsilon\}$.

To mimic the behaviour of O'Rourke's algorithm (see Section 2.3), among all the possible choices of the linear model, we want the one that maximises $|\mathcal{S}|$. Hence, we are interested in the slope $\alpha$ that maximises the smallest $i$ such that the corresponding point $(X_i, i)$ is outside $\mathcal{S}$. Formally, we are interested in maximising

$$i^* = \min\{i \in \mathbb{N} \mid (X_{i^*}, i^*) \notin \mathcal{S}\}. \tag{3.1}$$

Since $i^*$ is a random variable, we will find its expected value over different realisations of the sequence $\{X_i\}_{i\in\mathbb{N}}$ as a function of $\varepsilon, \alpha, \mu, \sigma^2$. An example of a realisation is depicted in Figure 3.2a.

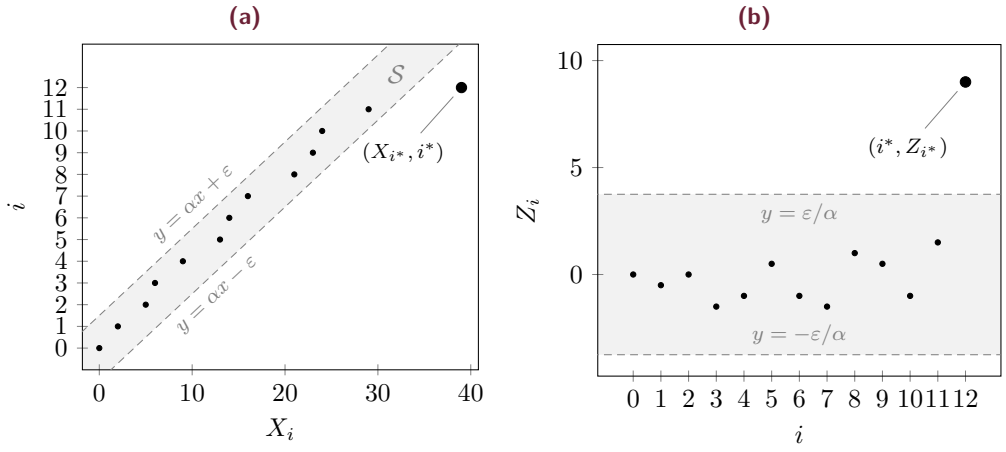



**Figure 3.2:** An example of random walk (a) and the corresponding transformed random walk (b).

## 3.2 Main results

We recall that the value of $i^*$ depends on the choice of the slope $\alpha$, and the objective of the algorithm is to maximise the expected value of $i^*$. Our main result is that, in a suitable limit, this maximum is achieved when $\alpha = 1/\mu$, and in this case the number of keys covered scales as $\Theta(\varepsilon^2)$.

More precisely, we can prove the following theorems and corollaries characterising $i^*$ on general or specific distributions of the gaps between consecutive keys in $S$ (proofs follow below all the statements).

**Theorem 3.1.** *Given any $\varepsilon \geq 1$ and a sorted set $S$ of $n$ input keys, suppose that the gaps between consecutive keys in $S$ are a realisation of a random process consisting of positive, independent and identically distributed random variables with mean $\mu$ and variance $\sigma^2$. Then, if $\varepsilon$ is sufficiently larger than $\sigma/\mu$, the expected number of keys covered by a segment with slope $\alpha = 1/\mu$ and maximum error $\varepsilon$ is*

$$\frac{\mu^2}{\sigma^2}\varepsilon^2.$$

The following theorem shows that a segment with slope $\alpha = 1/\mu$ is on average the best possible choice in terms of the number of $\varepsilon$-approximated keys.

**Theorem 3.2.** *Under the assumptions of Theorem 3.1, the largest expected number of keys covered by a segment with maximum error $\varepsilon$ is achieved with slope $1/\mu$.*

The variance of the length of the segment with slope $\alpha = 1/\mu$ can also be written in closed-form.

**Theorem 3.3.** *Under the assumptions of Theorem 3.1, the variance of the number of keys covered by a segment with slope $1/\mu$ and maximum error $\varepsilon$ is*

$$\frac{2}{3}\frac{\mu^4}{\sigma^4}\varepsilon^4.$$

By instantiating some common probability distributions in Theorem 3.1, it follows the next key corollary.

**Corollary 3.1.** *Under the assumptions of Theorem 3.1, the expected number of keys covered by a segment is:*

- $3\frac{(a+b)^2}{(b-a)^2}\varepsilon^2$ *if the gaps are iid and uniformly distributed with minimum $a$ and maximum $b$.*

- $a(a-2)\varepsilon^2$ *if the gaps are iid and Pareto (power law) distributed with minimum value* $k > 0$ *and shape parameter* $a > 2$.
- $\varepsilon^2/(e^{\sigma^2} - 1)$ *if the gaps are iid and lognormally distributed with mean* $\mu$ *and variance* $\sigma^2$.
- $\varepsilon^2$ *if the gaps are iid and exponentially distributed with rate* $\lambda > 0$.
- $k\varepsilon^2$ *if the gaps are iid and gamma distributed with shape parameter* $k > 0$ *and scale parameter* $\theta > 0$.

As the next result shows, the number of keys covered by a segment scales as $\Theta(\varepsilon^2)$ even when $S$ contains repeated keys, i.e. when some gaps are equal to zero.

**Corollary 3.2.** *Given any* $\varepsilon \geq 1$ *and a sorted set* $S$ *of input keys, suppose that the gap between any two consecutive keys in* $S$ *is zero with probability* $p$*, and that, with the remaining probability* $(1-p)$*, the gap is drawn from a distribution with mean* $\mu$ *and variance* $\sigma^2$*. Define*

$$\kappa^2 = \frac{(1-p)\mu^2}{\sigma^2 + p\mu^2}.$$

*If* $\varepsilon$ *is sufficiently larger than* $1/\kappa$*, the expected number of keys covered by a segment with slope* $\alpha = 1/(\mu(1-p))$ *and maximum error* $\varepsilon$ *is* $\kappa^2\varepsilon^2$.

Finally, we can show that the number of segments $s$ which have slope $\alpha = 1/\mu$ and guarantee a maximum error $\varepsilon$ on a stream of length $n$ is very concentrated around $\Theta(n/\varepsilon^2)$.

**Theorem 3.4.** *Under the assumptions of Theorem 3.1, the number of segments* $s$ *needed to cover a stream of length* $n$ *with error at most* $\varepsilon$ *converges almost surely to*

$$\frac{\sigma^2}{\mu^2}\,\frac{n}{\varepsilon^2},$$

*and the relative standard deviation of* $s$ *converges to zero as* $1/\sqrt{n}$ *when* $n \to \infty$.

In the following, given this last result, we will say that the number of segments $s$ is $\mathcal{O}(n/\varepsilon^2)$ "with high probability" [MR95].

The above theorems are based on the assumption that gaps are independent and identically distributed. In applications, this condition might not be true and thus it is important to assess whether our results hold, even in some asymptotic regimes, when gaps are autocorrelated. We answer this question affirmatively in Section 3.3.

## Proof of Theorem 3.1

Let us consider the Cartesian plane introduced in Section 3.1. By swapping abscissas and ordinates of the plane, the equation of the two parallel lines becomes $y = (x \pm \varepsilon)/\alpha$ ($x$ and $y$ are the new coordinates), and the sequence of points becomes $\{(i, X_i)\}_{i\in\mathbb{N}}$. This sequence describes a discrete-time random walk with iid increments $G_i = X_i - X_{i-1}$. The main idea of the proof is to determine the Mean Exit Time (MET) of the random walk out of the strip delimited by the two lines above, i.e. the mean of

$$i^* = \min\left\{ i \in \mathbb{N} \;\middle|\; X_i > \frac{i}{\alpha} + \frac{\varepsilon}{\alpha} \;\vee\; X_i < \frac{i}{\alpha} - \frac{\varepsilon}{\alpha} \right\}. \tag{3.2}$$

To simplify the analysis, we consider the following transformed random walk, where we use the equality $X_i = \sum_{j=1}^{i} G_j$ and set $W_j = G_j - 1/\alpha$:

$$Z_i = X_i - \frac{i}{\alpha} = \sum_{j=1}^{i} \left(G_j - \frac{1}{\alpha}\right) = \sum_{j=1}^{i} W_j.$$

The objective (3.2) can be thus rewritten as

$$i^* = \min\left\{ i \in \mathbb{N} \mid Z_i > \varepsilon/\alpha \vee Z_i < -\varepsilon/\alpha \right\},$$

which is the exit time of the transformed random walk $\{Z_i\}_{i\in\mathbb{N}}$ whose increments $W_j$ are iid with mean $\mathbb{E}[W_j] = \mathbb{E}[G_j - 1/\alpha] = \mu - 1/\alpha$, variance $\mathrm{Var}[W_j] = \mathrm{Var}[G_j] = \sigma^2$ and pdf $f_W(w) = f_G(w + 1/\alpha)$.

An example of this transformed random walk is depicted in Figure 3.2b above.

Let $T(z_0) = \mathbb{E}[i^* \mid Z_0 = z_0]$ be the MET if the random walk $\{Z_i\}_{i\in\mathbb{N}}$ starts from $z_0$. In our case, it starts from $z_0 = y_0 - 0/\alpha = 0$ (since $y_0 = 0$). It is well known [MMP05; Red01] that $T(z)$ satisfies the Fredholm integral equation of the second kind $T(z_0) = 1 + \int_{-\varepsilon/\alpha}^{\varepsilon/\alpha} f_W(z - z_0)\, T(z)\, dz$, which for our problem can be rewritten as

$$T(z_0) = 1 + \int_{-\varepsilon/\alpha}^{\varepsilon/\alpha} f_G\left(z - z_0 + \frac{1}{\alpha}\right) T(z)\, dz. \tag{3.3}$$

While solving exactly the integral equation (3.3) is in general impossible, it is possible to give a general limiting result when $\varepsilon$ is sufficiently large. More specifically, when $\alpha = 1/\mu$, the transformed random walk $Z_i$ has increments with zero mean and variance equal to $\sigma^2$, and the boundaries of the strip are at $\pm\varepsilon\mu$. When $\sigma \ll \varepsilon\mu$ or equivalently $\varepsilon \gg \sigma/\mu$, the Central Limit Theorem tells us that the distribution of the position of the random walker is Normal because many steps are necessary to reach

the boundary. In this case, the transformed random walk converges to a Brownian motion (or Wiener process) in continuous time [Gar85].

Now, it is well known [Gar85] that for a driftless Wiener process the MET out of an interval $[-\delta/2, \delta/2]$ is

$$T(x) = \frac{(\delta/2)^2 - x^2}{\sigma^2}, \tag{3.4}$$

where $x \in [-\delta/2, \delta/2]$ is the value of the process at the initial time. In our case, $x = 0$ and $\delta = 2\varepsilon/\alpha = 2\varepsilon\mu$, thus we finally have the statement of the theorem.

## Proof of Theorem 3.2

Using an approach similar to the one of the previous section, if $\alpha \neq 1/\mu$, the transformed random walk $Z_i = X_i - 1/\alpha = \sum_{j=1}^{i} W_j$ has increments with mean $d \equiv \mathbb{E}[W_j] = \mu - 1/\alpha$ and variance $\sigma^2$ (see the previous section). For large $\varepsilon$ the process converges to a Brownian motion with drift. The MET out of an interval $[-\delta/2, \delta/2]$ for a Brownian motion with drift coefficient $d \neq 0$ and diffusion rate $\sigma$ can be proved to be

$$T(0) = \frac{\delta}{2d}\left[\frac{e^{d\delta/\sigma^2} + e^{-d\delta/\sigma^2} - 2}{e^{d\delta/\sigma^2} - e^{-d\delta/\sigma^2}}\right]. \tag{3.5}$$

To show this, we use the known fact (see [Gar85, §5.2.7]) that the MET $T(x)$ out of an interval $[-\delta/2, \delta/2]$ of a Brownian motion with drift $d$ and diffusion rate $\sigma$ starting at position $x$ satisfies the differential equation

$$d\,\frac{dT(x)}{dx} + \frac{\sigma^2}{2}\frac{d^2T(x)}{dx^2} = -1,$$

with the boundary conditions

$$T(\delta/2) = T(-\delta/2) = 0.$$

The solution to this Cauchy problem is

$$T(x) = \frac{\delta - 2x}{2d} + \frac{\delta}{d}\left[\frac{e^{-d\delta/\sigma^2} - e^{-2dx/\sigma^2}}{e^{d\delta/\sigma^2} - e^{-d\delta/\sigma^2}}\right].$$

If the random walker starts at $x = 0$, this expression becomes $T(0)$ of Equation 3.5.

Clearly, by taking the limit $d \to 0$ (i.e. $\mu \to 1/\alpha$) in (3.5), one obtains Equation 3.4. As in the proof of Theorem 3.1, we have $\delta = 2\varepsilon/\alpha$, thus substituting it in the equation above we get

$$T(0) = \frac{\varepsilon}{\alpha d} \tanh\left(\frac{\varepsilon d}{\alpha \sigma^2}\right).$$

It is easy to see that the maximum of $T(0)$ is achieved for $d = 0$, i.e. when $\alpha = 1/\mu$, which is exactly the setting considered in Theorem 3.1.

## Proof of Corollary 3.2

Under the assumptions of the corollary, the gaps $G_j$ have mean value $\tilde{\mu} = (1-p)\mu$ and variance $\tilde{\sigma}^2 = (1-p)(\sigma^2+\mu^2) - (1-p)^2\mu^2$, thus the increments $W_j = G_j - 1/\alpha = G_j - \tilde{\mu}$ of the transformed random walk have zero mean and variance $\tilde{\sigma}^2$. Using Theorem 3.1 and Theorem 3.2, we conclude that the optimal slope is $\alpha = 1/\tilde{\mu}$ and the expected number of keys is $(\tilde{\mu}^2/\tilde{\sigma}^2)\,\varepsilon^2$, i.e. the thesis.

## Proof of Theorem 3.3

Following Gardiner [Gar85, Equation 5.2.156], the second moment $T_2(x)$ of the exit time of a Brownian motion with diffusion rate $\sigma$ starting at $x$ is the solution of the partial differential equation

$$-2T(x) = \frac{\sigma^2}{2}\,\partial_x^2\, T_2(x),$$

where $T(x)$ is the MET out of an interval $[-\delta/2, \delta/2]$ (see Equation 3.4), with boundary conditions $T_2(\pm\delta/2) = 0$.

Solving for $T_2(x)$, we get

$$T_2(x) = \frac{x^4 - 2\delta^2 x^2/3 + 5\delta^4/16}{3\sigma^4}.$$

Setting $x = 0$ and $\delta = 2\varepsilon/\alpha = 2\varepsilon\mu$, we find that the second moment of the exit time starting at $x = 0$ is

$$T_2(0) = \frac{5}{3}\frac{\mu^4}{\sigma^4}\varepsilon^4,$$

thus

$$T_2(0) - [T(0)]^2 = \frac{2}{3}\frac{\mu^4}{\sigma^4}\varepsilon^4.$$

### Proof of Theorem 3.4

Consider a process that starts a new segment $j+1$ as soon as the current one $j$ cannot cover more than $i_j^*$ keys without exceeding the error $\varepsilon$ (see Equation 3.2). We define the total number of segments $s$ on a stream of length $n$ as

$$s(n) = \sup\{k \geq 1 \mid S_k \leq n\},$$

where $S_k = i_1^* + i_2^* + \cdots + i_k^*$.

We notice that $\{s(n)\}_{n\geq 0}$ is a renewal counting process of non-negative integer random variables $i_1^*, \ldots, i_k^*$, which are independent due to the lack of memory of the random walk.

Let $\mathbb{E}[i_j^*] = 1/\lambda$ and $\mathrm{Var}[i_j^*] = \varsigma^2$. It is well known [EMK97, §2.5.2] that $\mathbb{E}[s(n)] = \lambda n + \mathcal{O}(1)$ as $n \to \infty$, $\mathrm{Var}[s(n)] = \varsigma^2\lambda^3 n + o(n)$ as $n \to \infty$, and that $s(n)/n \xrightarrow{\text{a.s.}} \lambda$. In our case (see Theorems 3.1 and 3.3), it holds

$$\frac{1}{\lambda} = \frac{\mu^2}{\sigma^2}\varepsilon^2 \quad \text{and} \quad \varsigma^2 = \frac{2}{3}\frac{\mu^4}{\sigma^4}\varepsilon^4,$$

hence $s(n)/n \xrightarrow{\text{a.s.}} \lambda = (\sigma/(\mu\,\varepsilon))^2$. Finally, the following ratio converges to zero as $n \to \infty$:

$$\frac{\sqrt{\mathrm{Var}[s(n)]}}{\mathbb{E}[s(n)]} \to \sqrt{\frac{\varsigma^2\lambda}{n}} = \sqrt{\frac{2}{3}}\frac{\mu\varepsilon}{\sigma}\frac{1}{\sqrt{n}}.$$

## 3.3 A conjecture for the case of correlated keys

In this section, we study the case in which the independence assumption of Section 3.2 is waived. Specifically, we study a random process $\{G_i\}_{i\in\mathbb{N}}$ generating gaps that consist of positive and identically distributed random variables with mean $\mathbb{E}[G_i] = \mu$, variance $\mathrm{Var}[G_i] = \sigma^2$, and covariances $C(\ell) = \mathrm{Cov}[G_i, G_{i+\ell}] = \mathbb{E}[G_i G_{i+\ell}] - \mu^2$ for any lag $\ell \geq 1$.

As usual, we define the random variables modelling the $i$th input key $X_i$ as the cumulative sum $X_i = \sum_{j=1}^{i} G_j$ (for $i = 1, 2, \ldots$) and fix $X_0 = 0$. It is easy to see

that their mean is $\mathbb{E}[X_i] = i\mu$ and their variance is

$$\begin{aligned}
\operatorname{Var}[X_i] &= \sum_{j=1}^{i}\sum_{k=1}^{i}\operatorname{Cov}[G_j, G_k] \\
&= \sum_{j=1}^{i}\operatorname{Var}[G_j] + 2\sum_{j<i}\operatorname{Cov}[G_j, G_i] \\
&= i\sigma^2 + 2\left[(i-1)C(1) + (i-2)C(2) + \cdots + C(i-1)\right] \\
&= i\sigma^2 + 2\sum_{\ell=1}^{i-1}(i-\ell)C(\ell) \\
&= i\sigma^2\left[1 + 2\sum_{\ell=1}^{i-1}\left(1 - \frac{\ell}{i}\right)\rho(\ell)\right],
\end{aligned} \tag{3.6}$$

where $\rho(\ell) \equiv C(\ell)/\sigma^2$ is the autocorrelation function.

When $i$ is much larger than the time scale $\ell_0$ after which the autocorrelation is negligible (the "memory" of the process), the $\ell/i$ term can be neglected.

Hence, as $i \gg \ell_0$ we get the approximation

$$\operatorname{Var}[X_i] \simeq i\left(\sigma^2 + 2\sum_{\ell=1}^{\ell_0} C(\ell)\right) = i\sigma^2\left(1 + 2\sum_{\ell=1}^{\ell_0}\rho(\ell)\right). \tag{3.7}$$

Thus for large $i$, the process becomes exactly diffusive (i.e. the $\operatorname{Var}[X_i]$ increases linearly with $i$) as in a random walk with iid increments and effective diffusion rate $\sigma^2(1 + 2\sum_{\ell=1}^{\ell_0}\rho(\ell))$. We therefore state the following conjecture:

**Conjecture 3.1.** *If $\varepsilon$ is sufficiently large, the random walk will make a large number $i$ of steps, and thus it will satisfy the condition of Theorem 3.1 with mean $\mathbb{E}[X_i] = i\mu$ and variance $\operatorname{Var}[X_i]$ given by Equation 3.7, giving for the expected number of keys covered by a segment with slope $\alpha = 1/\mu$ and maximum error $\varepsilon$ the value*

$$\frac{1}{1 + 2\sum_{\ell=1}^{\ell_0}\rho(\ell)}\frac{\mu^2}{\sigma^2}\varepsilon^2 \approx \frac{1}{1 + 2\sum_{\ell=1}^{\infty}\rho(\ell)}\frac{\mu^2}{\sigma^2}\varepsilon^2.$$

In the above approximation, we have extended the sum at the denominator from $\ell \le \ell_0$ to $\ell \to +\infty$, since by construction $\rho(\ell)$ is negligible (or zero) when $\ell > \ell_0$. The above formula shows that, in a random walk with correlated increments, the expected number of keys increases quadratically with $\varepsilon$ as in the random walk with iid increments. The main difference is the prefactor multiplying $\mu^2\varepsilon^2/\sigma^2$: when the increments are positively correlated ($\rho(\ell) > 0$), the prefactor is smaller

than 1, i.e. the expected number of keys is smaller than for a random walk with iid increments.

In order to dig into the significance of this observation, let us study a few specific, yet realistic, examples. In Section 3.4, we provide numerical evidence that the above conjecture describes accurately the expected number of keys in one of the examples presented below.

**Example 1** (Moving-average process)**.** Let us consider a process $\{U_i\}_{i\in\mathbb{N}}$ of positive iid variables $U_i$ having mean $\mathbb{E}[U_i] = \mu_U$ and variance $\mathrm{Var}[U_i] = \sigma_U^2$. We then assume that a gap $G_i$ is generated by a convolution of $\ell_0$ variables $U_i$ as $G_i = \sum_{k=1}^{\ell_0} \phi_i U_i$, where $\phi_i$ are positive weights, i.e. that $\{G_i\}_{i\in\mathbb{N}}$ is a *moving-average process of order* $\ell_0$.

It is immediate to show that $\mu \coloneqq \mathbb{E}[G_i] = \mu_U \sum_{k=1}^{\ell_0} \phi_k$ and $\sigma^2 \coloneqq \mathrm{Var}[G_i] = \sigma_U^2 \sum_{k=1}^{\ell_0} \phi_k^2$. Moreover, it holds

$$C(\ell) = \mathrm{Cov}[G_i, G_{i+\ell}] = \begin{cases} \sigma_U^2 \sum_{k=1}^{\ell_0-\ell} \phi_k \phi_{k+\ell} & \text{if } \ell < \ell_0 \\ 0 & \text{otherwise.} \end{cases}$$

In the special case of a flat filter, i.e. $\phi_i = 1$ for any $i$, it is easy to see that $\mu = \ell_0 \mu_U$, $\sigma^2 = \ell_0 \sigma_U^2$, and that $C(\ell) = \sigma_U^2(\ell_0 - \ell)$ if $\ell < \ell_0$ and $C(\ell) = 0$ otherwise. By plugging the last definitions of $\sigma^2$ and $C(\ell)$ into Equation 3.6, we obtain

$$\begin{aligned} \mathrm{Var}[X_i] &= i\ell_0\sigma_U^2 + 2i \sum_{\ell=1}^{\ell_0-1} \left(1 - \frac{\ell}{i}\right) \sigma_U^2(\ell_0 - \ell) \\ &\leq i\ell_0\sigma_U^2 + 2i\sigma_U^2 \sum_{\ell=1}^{\ell_0-1} (\ell_0 - \ell) \\ &= i\left(\ell_0\sigma_U^2 + 2\sigma_U^2 \frac{\ell_0(\ell_0-1)}{2}\right) \\ &= i\ell_0^2\sigma_U^2 \\ &= i\ell_0\sigma^2. \end{aligned}$$

To mimic the statement of Theorem 3.1 in the special case we just described, we conjecture that if $\varepsilon$ is sufficiently larger than $\sigma\sqrt{\ell_0}/\mu$, the expected number of keys covered by a segment with slope $1/\mu$ and maximum error $\varepsilon$ is at least

$$\frac{\mu^2}{\sigma^2\ell_0}\varepsilon^2. \tag{3.8}$$

□

**Example 2** (Autoregressive process)**.** This second example assumes that the gaps follow an AR(1) process, i.e. $G_i = \varphi\, G_{i-1} + \eta_i$, where $\varphi$ is a real parameter, and $\eta_i$ is a white noise with mean $\mu_\eta$ and variance $\sigma_\eta^2$.

It is well-known [Fel70] that when $|\varphi| < 1$

$$\mu \coloneqq \mathbb{E}[G_i] = \frac{\mu_\eta}{1-\varphi}, \quad \sigma^2 \coloneqq \mathrm{Var}[G_i] = \frac{\sigma_\eta^2}{1-\varphi^2}, \quad \text{and } C(\ell) = \sigma^2 \varphi^\ell.$$

The autocorrelation function decays exponentially to zero, thus in this case $\rho(\ell)$ is never zero, even if the time scale of the process is finite and related to $|\varphi|$. The variance of the random walk is

$$\begin{aligned}\mathrm{Var}[X_i] &= i\sigma^2 \left[1 + 2\sum_{\ell=1}^{i}\left(1 - \frac{\ell}{i}\right)\varphi^\ell\right] \\ &= i\sigma^2\left(1 + 2\frac{\varphi}{1-\varphi} - 2\frac{\varphi - \varphi^{i+1}}{(1-\varphi)^2 i}\right).\end{aligned}$$

When $i$ is very large the last (negative) term in brackets becomes negligible, and the variance of the random walk may be approximated by

$$\mathrm{Var}[X_i] \simeq i\sigma^2 \frac{1+\varphi}{1-\varphi}.$$

To mimic the statement Theorem 3.1 in the special case we just described, we conjecture that if $\varepsilon$ is sufficiently larger than $\sqrt{\frac{1+\varphi}{1-\varphi}}\frac{\sigma}{\mu}$, the expected number of keys covered by a segment with slope $1/\mu$ and maximum error $\varepsilon$ is

$$\frac{1-\varphi}{1+\varphi}\frac{\mu^2}{\sigma^2}\varepsilon^2. \tag{3.9}$$

□

## 3.4 Experiments

We start with an experiment aimed at validating our main result (Theorem 3.1).[1] We generated $10^7$ random streams of gaps for each of the following distributions: Uniform, Pareto, Lognormal, Exponential/Gamma. For each generated stream $S$, we picked an integer $\varepsilon$ in the range $[1, 2^8]$, which contains the values that are the

[1]The code to reproduce the experiments is available at `https://github.com/gvinciguerra/Learned-indexes-effectiveness`. The experiments were run on an Intel Xeon Gold 6132 CPU.

most effective in practice, as we will see in Chapter 4. Then, we ran the following piecewise linear $\varepsilon$-approximation construction algorithms with input $\varepsilon$ and $S$:

**MET.** This is the algorithm that fixes the slope of a segment to $1/\mu$ and stops when the next point of $S$ is outside the strip of width $2\varepsilon$, see Equation 3.1. This corresponds to the random process we used to prove Theorem 3.1.

**OPT.** This is O'Rourke's algorithm (see Lemma 2.1), which computes the segment (of any slope and intercept) that $\varepsilon$-approximate the longest prefix of $S$.

We analysed the length of the segments computed by the two previous algorithms, that is, the index of the first key that causes the algorithm to stop because the (vertical) distance of the point from the segment is larger than $\varepsilon$. We plot in Figure 3.3 the mean and the standard deviation of these segment lengths. The figure shows that the theoretical mean segment length computed according to Corollary 3.1 (hence the formula $(\mu^2/\sigma^2)\,\varepsilon^2$), depicted as a solid black line, accurately describes the experimented algorithm MET, depicted as red points, over all tested distributions (just observe that the solid black line overlaps the red points). Moreover, the standard deviation of the exit time, depicted as a shaded red region, follows the corresponding bound proved in Theorem 3.3 and depicted as two dashed black lines in each plot. So our theoretical analysis of Theorem 3.1 is tight.

Not surprisingly, the plots show also that OPT performs better than MET. This is because MET fixes the slope of a segment to $1/\mu$, while OPT optimally adapts to each sequence of points given in input by choosing the slope that maximises the number of points covered by a segment. Thus it is more robust to outliers and hence can find longer segments.

Overall this first experiment validates our claim that a piecewise linear $\varepsilon$-approximation use a space that decreases as fast as $\mathcal{O}(n/\varepsilon^2)$ (Theorem 3.4).

The second experiment analysed the slopes found by OPT over the sequence of points generated according to the previous experiment, and averaged over $\varepsilon$. We compared them to the fixed slope $1/\mu$ of MET. Table 3.1 clearly shows that these slopes are centred around $1/\mu$, thus confirming the result of Theorem 3.2 that $1/\mu$ is the best slope on average.

The third experiment was devoted to studying the *accuracy* of the approximation to the mean exit time provided by the formula $(\mu^2/\sigma^2)\,\varepsilon^2$ under the assumption "with $\varepsilon$ sufficiently larger than $\sigma/\mu$" present in the statement of Theorem 3.1. To this end, we properly set the distribution parameters to obtain a ratio $\sigma/\mu$ in $\{0.15, 1.5, 15\}$. We plot in Figure 3.4 the relative error between the experimented MET (i.e. the

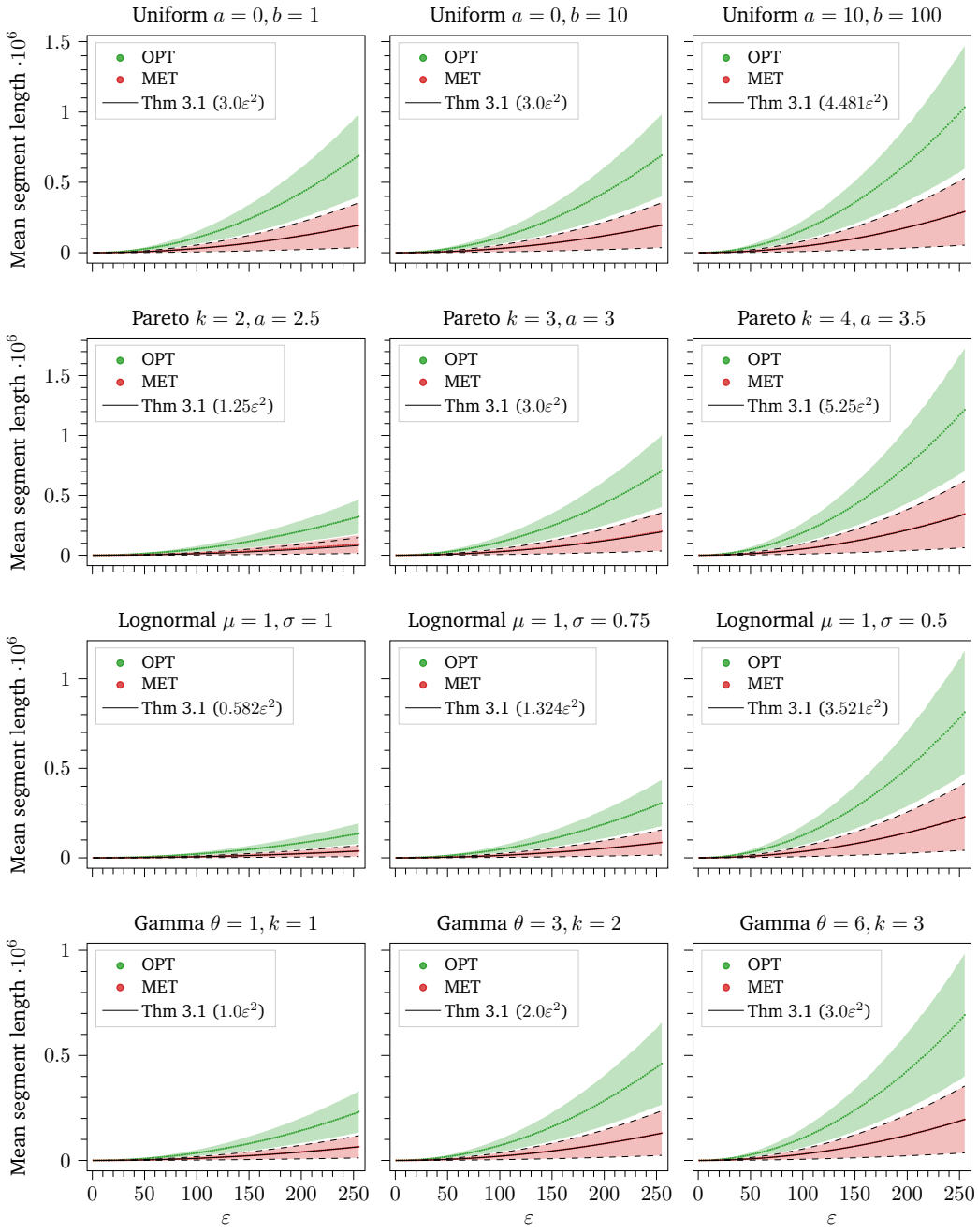



**Figure 3.3:** We consider four gap distributions — Uniform, Pareto, Lognormal, and Gamma — with various parameter settings. We plot the formula $(\mu^2/\sigma^2)\,\varepsilon^2$ given in Theorem 3.1 with a solid black line and the Mean Exit Time (MET) of the experimented random walk with red points. The figure shows that they overlap, thus the formula stated in Theorem 3.1 accurately predicts the experimented MET. The figure also shows the performance of the algorithm OPT with green points. The shaded regions represent the standard deviation. The improvement of OPT with respect to MET is evident, indicating that OPT is more robust to outliers.

| Distribution | Parameters | $1/\mu$ | Avg. slope range |
|---|---|---|---|
| Uniform | $a = 0, b = 1$ | 2 | $[2.000, 2.002]$ |
| Uniform | $a = 0, b = 10$ | 0.2 | $[0.200, 0.200]$ |
| Uniform | $a = 10, b = 100$ | 0.018 | $[0.018, 0.018]$ |
| Pareto | $k = 2, a = 2.5$ | 0.3 | $[0.300, 0.301]$ |
| Pareto | $k = 3, a = 3$ | 0.222 | $[0.222, 0.222]$ |
| Pareto | $k = 4, a = 3.5$ | 0.179 | $[0.179, 0.179]$ |
| Lognormal | $\mu = 1, \sigma = 0.5$ | 0.325 | $[0.325, 0.325]$ |
| Lognormal | $\mu = 1, \sigma = 0.75$ | 0.278 | $[0.278, 0.278]$ |
| Lognormal | $\mu = 1, \sigma = 1$ | 0.223 | $[0.223, 0.224]$ |
| Exponential | $\lambda = 1$ | 1 | $[1.000, 1.003]$ |
| Gamma | $\theta = 3, k = 2$ | 0.167 | $[0.167, 0.167]$ |
| Gamma | $\theta = 6, k = 3$ | 0.056 | $[0.056, 0.056]$ |

**Table 3.1:** The range of slopes found by OPT in the experiments of Figure 3.3. Notice that these ranges are centred and close to $1/\mu$, which is the theoretical slope that maximises the MET of the random walk depicted in Figure 3.2a.

empirical mean segment length) and the formula above, as $\varepsilon$ grows from $1$ to $2^8$. For the left plot, we notice that for all the distributions the relative error converges soon to $0$ (here, the ratio $\sigma/\mu$ is very small compared to $\varepsilon$). In the middle plot, the convergence is fast for Gamma and Lognormal distributions, but it is slower for Pareto because $a = 2.202$ generates a very fat tail that slows down the convergence of the Central Limit Theorem. This is a well-known fact [Fel70] since the third moment diverges and the region where the Gaussian approximation holds grows extremely slowly with the number of steps of the walk. This effect is even more evident in the rightmost plot where all the three distributions have very fat tails. Overall, Figure 3.4 confirms that $\varepsilon$ does not need to be "too much larger" than $\sigma/\mu$ to get convergence to the predicted mean exit time, as stated in Theorem 3.1.

The fourth experiment considered streams of increasing length $n$ (up to $10^6$) that follow the gap distributions of the first column of Figure 3.3. For each part of a stream, we computed with the MET algorithm the $s$ segments that approximate that stream with error $\varepsilon = 50$. By repeating the experiment $10^4$ times, we computed the average and the standard deviation of $s/n$. Figure 3.5 shows that for a large $n$ the distribution of $s/n$ concentrates on $\lambda = (\sigma/(\mu\,\varepsilon))^2$, with a speed that is faster for smaller $\mu\varepsilon/\sigma$, as predicted by Theorem 3.4.

The fifth experiment, reported in Figure 3.6, shows the average segment length of OPT on real-world datasets of 200 million elements from [Mar+20]. The books dataset represents book sale popularity from Amazon, while fb contains Facebook user

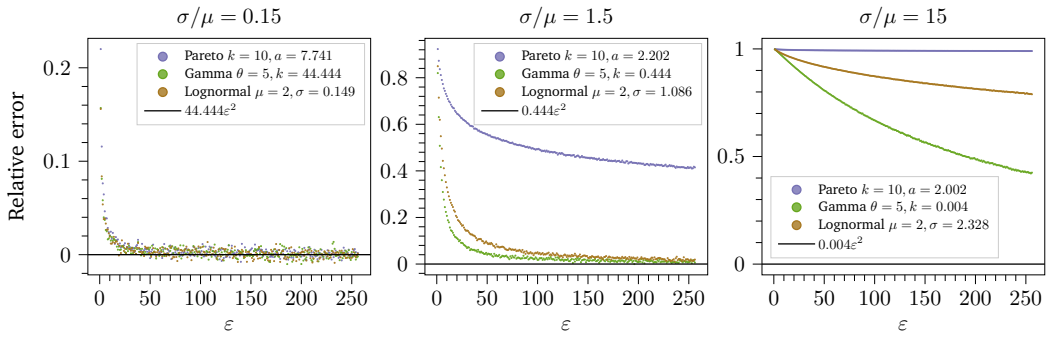



**Figure 3.4:** Three plots for three different settings of the ratio $\sigma/\mu$ for the distributions: Pareto, Gamma and Lognormal. We plot the relative error between the formula $(\mu^2/\sigma^2)\,\varepsilon^2$ of Theorem 3.1 and the experimented MET. Notice how the fat-tail of the distributions affects the accuracy of the formula with respect to MET, as commented in the text.

IDs. Even though these datasets do not satisfy the assumption of Theorem 3.1, the fitted curves show a superlinear growth in $\varepsilon$. This suggests that the $\varepsilon^{1+\mathcal{O}(1)}$ growth established in our analysis may also be valid on datasets that do not strictly follow the assumption on iid gaps.

The sixth experiment considered the random process described in Example 1 of Section 3.3, i.e. streams of gaps generated by moving-average processes of order $\ell_0$. Specifically, we computed the moving average of reals drawn from a uniform distribution (with parameters $a = 0$, $b = 1$) by using unit weights $\phi_i$ and by varying $\ell_0$ in $\{5, 50, 500\}$. For each value of $\ell_0$, we repeated the experiment $10^7$ times, each time picking an integer $\varepsilon$ in the range $[1, 2^8]$ and running OPT and MET with argument $\varepsilon$. Figure 3.7 shows that the mean segment length of the two algorithms scales quadratically in $\varepsilon$ and that the conjectured correction of the prefactor related to autocorrelation is in a very good agreement with simulations. This entails that even in the case of keys correlated at large lags (e.g. $\ell_0 = 500$) the result of Theorem 3.1 might still hold, as discussed thoroughly in Section 3.3.

The seventh and final experiment considered the random process described in Example 2 of Section 3.3, i.e. streams of gaps generated by an autoregressive process with parameter $\varphi$. We sampled the white noise terms from a uniform distribution (with parameters $a = 0$, $b = 1$) and varied $\varphi$ in $\{0.1, 0.5, 0.9\}$. For each value of $\varphi$, we repeated the experiment $10^7$ times, each time picking an integer $\varepsilon$ in the range $[1, 2^8]$ and running OPT and MET with argument $\varepsilon$. Figure 3.8 shows that the mean segment length of the two algorithms scales quadratically in $\varepsilon$ and that the conjectured correction of the prefactor related to autocorrelation is in very good agreement with simulations.

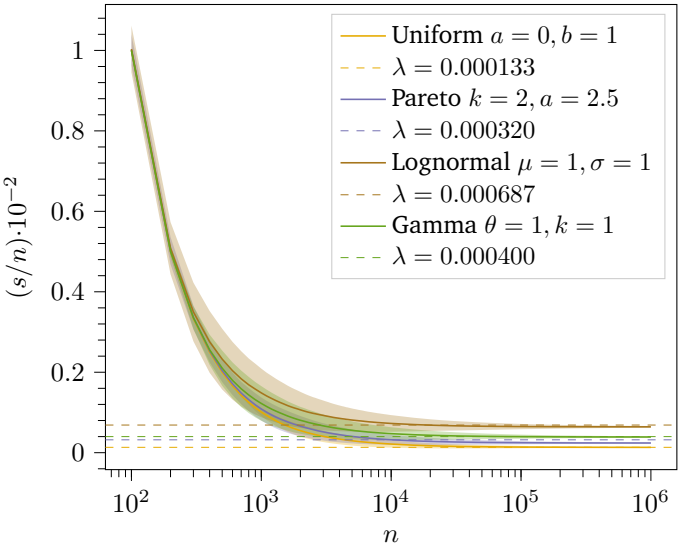



**Figure 3.5:** The solid line is the average and the shaded region is the standard deviation of $s/n$ over $10^4$ streams for four distributions, where $s$ is the number of segments computed by MET for a stream of length $n$. The dashed line depicts the limit stated in Theorem 3.4 to which the experimental values clearly converge to (quickly, at moderately small values of $n$).

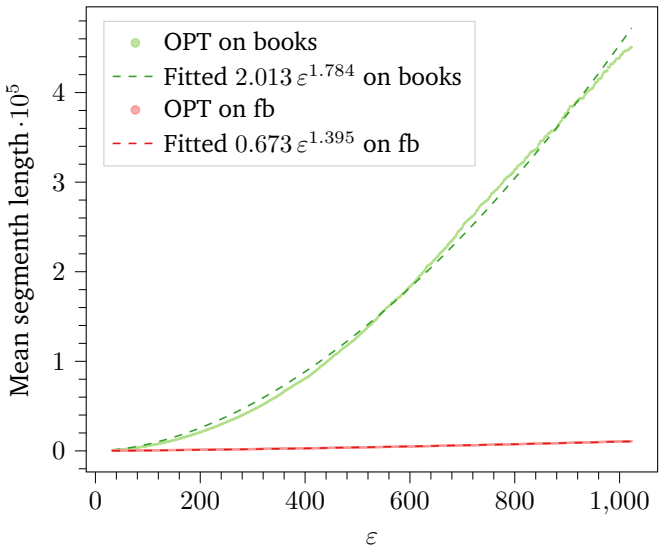



**Figure 3.6:** The average length of a segment computed by OPT on two real datasets exhibit a superlinear growth in $\varepsilon$.

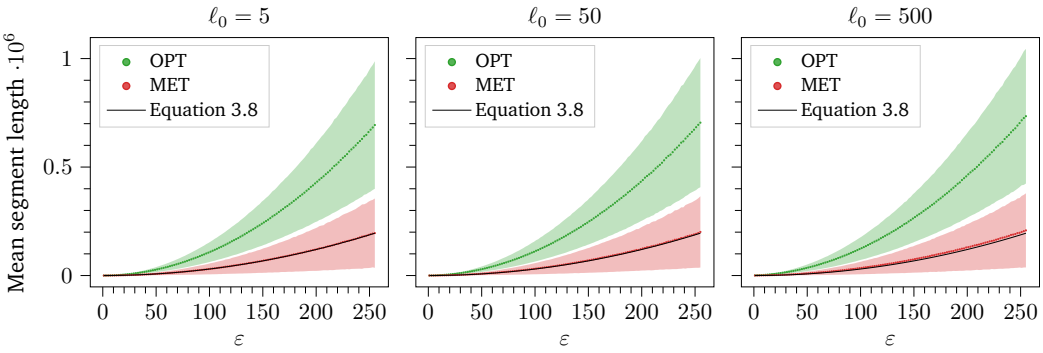



**Figure 3.7:** The mean segment length computed by OPT and MET on keys generated by three moving-average processes of order $\ell_0 = 5, 50$ and $500$, respectively. The solid black line overlaps the red dots of MET, and thus it shows that Equation 3.8 provides a good approximation for the case of correlated keys.

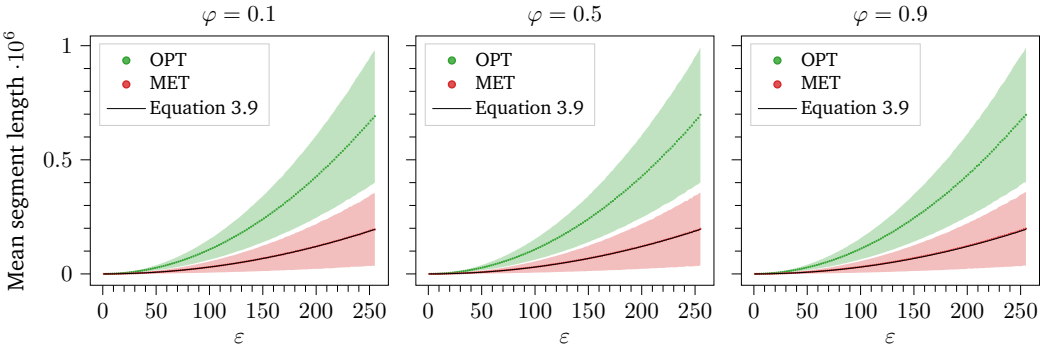



**Figure 3.8:** The mean segment length computed by OPT and MET on keys generated by three autoregressive processes with parameter $\varphi = 0.1, 0.5$ and $0.9$, respectively. The solid black line overlaps the red dots of MET, and thus it shows that Equation 3.9 provides a good approximation for the case of correlated keys.

## 3.5 Summary

We stated several results on the size of piecewise linear $\varepsilon$-approximations built on a sorted input array of size $n$. First, we proved an upper bound of $\mathcal{O}(n/\varepsilon)$ that holds for any kind of input data (Lemma 3.1). Then, we showed that if the gaps between consecutive input keys follow a distribution with finite mean and variance, then the size of a piecewise linear $\varepsilon$-approximation can be bounded by $\mathcal{O}(n/\varepsilon^2)$ (Theorem 3.4). Moreover, we argued that this last bound may hold also in the case in which the input keys are correlated. Finally, we performed an extensive set of experiments corroborating that all our theoretical achievements are highly precise.

The results in this chapter appeared in [FLV20; FLV21]. The code to reproduce the experiments is publicly available at `https://github.com/gvinciguerra/Learned-indexes-effectiveness`.

# 4 Predecessor search

We now dig into a generalisation of the *searching problem* introduced in Chapter 2, the so-called *dynamic predecessor search problem*. This problem asks to store an input set $S$ of $n$ keys drawn from an integer universe $\mathcal{U} = \{0, \ldots, u-1\}$ while supporting the following query and update operations (for $x, y \in \mathcal{U}$):

1. $predecessor(x) = \max\{a \in S \mid a \leq x\}$;
2. $range(x, y) = S \cap [x, y]$;
3. $insert(x)$ adds $x$ to $S$, i.e. $S \leftarrow S \cup \{x\}$;
4. $delete(x)$ removes $x$ from $S$, i.e. $S \leftarrow S \setminus \{x\}$.

The pervasiveness of these operations in areas such as Internet routing, databases, search engines, etc. makes predecessor search the most run algorithmic problem in the world [Păt16; Bel16; NR20].

Note that the basic *membership* or *dictionary* problem, which only aims to find whether $q \in S$, is less powerful than the predecessor search problem, as indeed we can solve it by simply checking whether $q = predecessor(q)$. Besides, the only availability of the membership operation does not allow implementing $range(x, y)$ efficiently (unless the universe range $[x, y]$ is small enough to allow an exhaustive membership test of its elements).

Existing solutions to predecessor search can be categorised according to the technique they use: *length reduction* and *cardinality reduction* [NR20]. The first technique consists in reducing the size of $\mathcal{U}$ recursively (or equivalently, the length of the searched keys), while the second one consists in reducing the size of $S$ recursively. Roughly speaking, trie-based data structures implement the former technique, while comparison-based search trees implement the latter technique.

Prototypical examples of data structures falling in the first category are van Emde Boas trees [vEB77], which support query and update operations in $\mathcal{O}(\log \log u)$ time and $\mathcal{O}(u)$ space, and Willard's y-fast tries [Wil83], which reduce the space of van Emde Boas trees to $\mathcal{O}(n)$.

Data structures in the second category range from balanced search trees [Cor+09, Ch. 13], which support query and update operations in $\mathcal{O}(\log n)$ time and $\mathcal{O}(n)$ space, to

Fredman and Willard's fusion trees, which support these operations in $\mathcal{O}(\log_w n)$ time and $\mathcal{O}(n)$ space in the $w$-bit word RAM model we defined in Chapter 2.

From a theoretical standpoint, Pătraşcu and Thorup closed the static predecessor search problem by showing matching lower and upper bounds in the word RAM model covering a full spectrum of space-time trade-offs [PT06]. Later, they also extended the results to the dynamic version when randomisation is allowed [PT14].

From a practical standpoint, modern storage systems dealing with massive data are predominantly based on B-trees and their variants [Pet18; IC20], which are cardinality reduction–based examples of data structures for external memory.

It is thus useful to introduce a model of computation that better captures the properties of these modern storage systems, the *external memory model*. This model abstracts the memory hierarchy of a computer by modelling just two levels: an internal memory of limited size $M$, and an external memory of unlimited size divided into blocks of $B$ consecutive items [Vit01]. Data is brought into internal memory and written back to external memory by transferring one block at a time, and the cost of a block transfer overshadows the cost of arithmetic and logic operations done by the machine. The efficiency of an algorithm is thus evaluated by counting the asymptotic number of transfers, or I/Os, it makes for solving a given problem. In this model, B-trees [Cor+09, Ch. 18] support query and update operations in $\mathcal{O}(\log_B n)$ I/Os, which is optimal [PT06].

Here, we build on some of the tools and results of the previous chapters and introduce novel data structures for the dynamic predecessor search problem in external memory. Compared to the data structures based on length or cardinality reduction mentioned above, and similarly to [Ao+11; Kra+18; Gal+19; Din+20] and other subsequent results surveyed in [FV20a], we exploit trends and patterns in the keys-positions Cartesian plane for the input set $S$ (see Section 2.1). Our solutions have provably efficient I/O-bounds and are shown to improve the space occupancy of B-trees from $\mathcal{O}(n/B)$ to $\mathcal{O}(n/B^2)$ under some general assumptions on the input data.

In the rest of this chapter, which is based on [FV20b], we first design a new static data structure called the PGM-index (Section 4.1) and make it dynamic (Section 4.2). Then, we introduce novel techniques for making it compressed (Section 4.3) and adaptive not only to the input data distribution but also to the query distribution (Section 4.4). We then show that our data structure can auto-tune itself efficiently to any given space or latency requirements (Section 4.5). Finally, we test the efficiency of the PGM-index on real-world datasets of up to 800M keys (Section 4.6).

## 4.1 The PGM-index

We assume that the input set $S$ is kept as a sorted array $A[1, n]$ in external memory, and we want to build an index structure that enables efficient predecessor and range searches in $A$ using as little space as possible (we defer the discussion on updates to Section 4.2).

This is common in practice, e.g. when $A$ corresponds to an immutable disk component (also referred to as run) in an LSM-based storage system [LC20], and an index must be built on it to reduce the number of disk I/Os. Here, it is essential to have a small index, especially considering that there are multiple disk components and thus multiple indexes to be kept in the faster memory levels, such as in main memory or in the CPU caches. As a matter of fact, we will see in Section 4.6.2 that just for a single array $A$, the corresponding index can take hundreds of megabytes.

The first ingredient of our index structure is the piecewise linear $\varepsilon$-approximation, which, we recall from Section 2.3, is a sequence of $m$ segments (i.e. linear models) that maps keys from $\mathcal{U}$ to their approximate positions (or rank) in the array $A$ with a maximum given error of at most $\varepsilon$. We use Theorem 2.1 to compute the piecewise linear $\varepsilon$-approximation and store the segments contiguously as triples with the first key $key$ covered by the segment, the slope $\alpha$ and the intercept $\beta$.

Since we access to $A$ via block transfers, we will focus on the case $\varepsilon = \Theta(B)$ so that a segment allows us to individuate and fetch just $\mathcal{O}(1)$ blocks of $A$ containing the answer to the predecessor search, and then we scan or binary search these blocks efficiently with no further I/Os.

It now remains for us to explain how to efficiently find a segment responsible for a query key $x$, which is the rightmost segment $s$ whose first key $s.key$ is smaller than or equal to $x$. A straightforward approach would be to build a B-tree on the segments which, in additional $\Theta(m/B)$ space on top of the $m$ segments, would allow us to find the correct segment in $\mathcal{O}(\log_B m)$ I/Os, but we will aim for something more space and time efficient than this.

Our approach, and second ingredient of our index structure, is a recursive design based entirely on segments. More precisely, we turn the piecewise linear $\varepsilon$-approximation built over $A$ into a subset of keys formed by the first key covered by each segment (i.e. an abscissa in the keys-positions Cartesian plane), and we proceed recursively by building another piecewise linear $\varepsilon$-approximation over this subset. This process continues until one single segment is obtained, which forms the root of our data structure, which we name the Piecewise Geometric Model index (PGM-index).

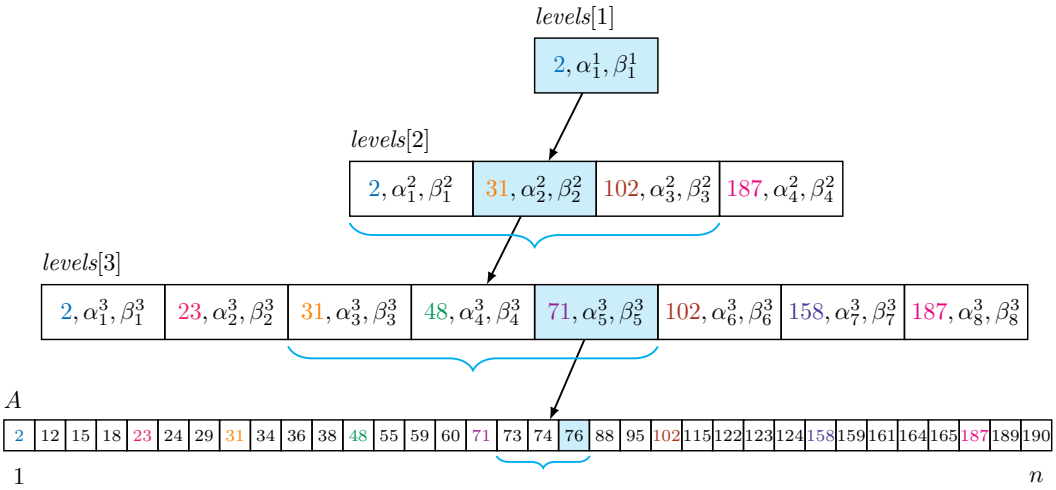



**Figure 4.1:** A PGM-index on an input array $A[1, n]$ is composed of levels of piecewise linear $\varepsilon$-approximations. Each level is thus a sequence of segments, each stored as a triple with the first key it covers, its slope $\alpha$ and intercept $\beta$.

Overall, each piecewise linear $\varepsilon$-approximation forms a level of the PGM-index, and each segment of that piecewise linear $\varepsilon$-approximation forms a sort of node of the data structure at that level, as depicted in Figure 4.1.

This construction process is formalised by Algorithm 1 and analysed here. At each iteration, we build a new level by computing at Line 5 a piecewise linear $\varepsilon$-approximation for the array $keys[1, m]$ obtained from the keys in the level below (initially $keys[1, m] = A[1, n]$). Since the time spent in each of the $l$ iterations is dominated by the execution of Line 5, which takes $\mathcal{O}(m)$ time (Theorem 2.1), and since $m$ shrinks by a factor of at least $2\varepsilon$ at each iteration (Lemma 3.1), the time complexity of Algorithm 1 is $\mathcal{O}(n + \sum_{i=1}^{l} n/(2\varepsilon)^i) = \mathcal{O}(n)$.[1]

If we map segments to nodes, then this approach constructs a sort of B-tree but with three main advantages: (i) its nodes have variable fan-out driven by the (typically large) number of keys covered by the segments associated with those nodes; (ii) the segment in a node plays the role of a constant-space and constant-time $\varepsilon$-approximate routing table for the various queries to be supported; (iii) the search in each node corrects the $\varepsilon$-approximate position returned by that routing table by a binary search (see next), and thus it has a time cost that depends logarithmically on $\varepsilon$, independently of the number of keys covered by the corresponding segment.

[1] Alternatively, we can construct the PGM-index levels simultaneously and in a single pass by keeping one "under construction segment" for each level. When O'Rourke's algorithm applied on $A$ outputs a new segment with key $k$, we insert $k$ in the segment under construction in the level above recursively. The time complexity of this alternative construction algorithm is still $\mathcal{O}(n)$, but we will use the same idea in Section 4.2 to support efficient appends (i.e. inserts of keys at the end of $A$).

**Algorithm 1** PGM-index construction.

**Input:** Input array $A[1, n]$, maximum error $\varepsilon \geq 0$
**Output:** PGM-index as an array of levels

1: $levels \leftarrow$ an empty dynamic array
2: $l \leftarrow 1$
3: $keys \leftarrow A$
4: **repeat**
5: $M \leftarrow$ build a piecewise linear $\varepsilon$-approximation on $keys$ via Theorem 2.1
6: $levels[l] \leftarrow M$
7: $l \leftarrow l + 1$
8: $m \leftarrow \text{SIZE}(M)$
9: $keys \leftarrow [M[1].key, \ldots, M[m].key]$
10: **until** $m = 1$
11: **return** $levels[1, l]$ in reverse order

**Algorithm 2** PGM-index query.

**Input:** Query key $x$, input array $A[1, n]$, PGM-index levels array $levels[1, l]$
**Output:** The predecessor of $x$ in $A$

1: $pos \leftarrow f_r(x)$, where $r = levels[1][1]$
2: **for** $i \leftarrow 2$ **to** $l$ **do**
3: $lo = \max\{pos - \varepsilon, 1\}$
4: $hi = \min\{pos + \varepsilon, \text{SIZE}(levels[i])\}$
5: $s \leftarrow$ the rightmost segment $s'$ in $levels[i][lo, hi]$ such that $s'.key \leq x$
6: $t \leftarrow$ the segment at the right of $s$
7: $pos \leftarrow \min\{f_s(x), f_t(t.key)\}$
8: $lo = \max\{pos - \varepsilon, 1\}$
9: $hi = \min\{pos + \varepsilon, n\}$
10: **return** search for $x$ in $A[lo, hi]$

We now detail how to perform a predecessor query in the PGM-index. At every level, we use the segment referring to the visited node to estimate the position of the searched key $x$ among the keys in the level below. The actual position is then found by a binary search for $x$ in a range of size $2\varepsilon + 1$ centred around the estimated position. Given that every key in the level below is the first key covered by a segment in that level, we have identified the next segment to query. As shown in Algorithm 2, this process continues up to the last level, at which point we compute an approximate position into $A$ and find the answer via a final binary search.[2]

For example, consider the PGM-index of Figure 4.1, and assume it was built with $\varepsilon = 1$. To search for the predecessor of $x = 75$, we start from the root segment

[2]Recall the observation of Footnote 2 in Chapter 2, which explains the $f_t(t.key)$ in Line 7 of Algorithm 2. Also, in the pseudocode, if there is no segment $t$ at the right of $s$, we assume that $f_t(t.key)$ returns the number of elements in the level below.

$s' = levels[1][1] = (2, \alpha_1^1, \beta_1^1)$ and compute the position $f_{s'}(x) = \lfloor \alpha_1^1 x + \beta_1^1 \rfloor = 2$ for the next level. We then search for $x$ in $levels[2][1-\varepsilon, 1+\varepsilon]$ among the keys $[2, 31, 102]$ (delimited by the cyan bracket), and we determine that the next segment responsible for $x$ is $s'' = levels[2][2] = (31, \alpha_1^2, \beta_2^2)$ because $x$ falls between 31 and 102. Then, we compute the position $f_{s''}(x) = 3$, and hence we search for $x$ in $levels[3][3-\varepsilon, 3+\varepsilon]$ among the keys $[31, 48, 71]$ (delimited by the cyan bracket). This way, we determine that $x > 71$, and hence the next segment responsible for $x$ is $s''' = levels[3][5] = (71, \alpha_5^3, \beta_5^3)$. Finally, we compute the position $f_{s'''}(x) = 17$ for the next level (i.e. the whole array $A$), and hence we search for $x$ in $A[17-\varepsilon, 17+\varepsilon]$ among the keys $[73, 74, 76]$ (delimited by the cyan bracket). Eventually, we find that $predecessor(x) = 74$.

**Theorem 4.1.** *Given a sorted array $A[1, n]$ and an integer $\varepsilon \geq 1$, the PGM-index answers predecessor queries in $\mathcal{O}((\log_\varepsilon m) \log(\varepsilon/B))$ I/Os, where $B$ is the block size of the external memory model. Range queries are answered in extra (optimal) $\mathcal{O}(K/B)$ I/Os, where $K$ is the number of keys satisfying the range query.*

*Proof.* By Lemma 3.1, each level of the PGM-index contains a number of segments that is at least $2\varepsilon$ smaller than the one in the level below. Therefore, the levels are $l = \mathcal{O}(\log_\varepsilon m)$, and the total space required by the index is $\sum_{i=0}^{l} m/(2\varepsilon)^i = \mathcal{O}(m)$. The bound on the I/O-complexity of a predecessor query follows by observing that a query performs $l$ binary searches over intervals of size at most $2\varepsilon + 1$. For a range query, the bound follows by observing that it is sufficient to scan and output $K$ keys in $A$ starting from the predecessor of the left endpoint of the range. □

An interesting setting for Theorem 4.1 is that of $\varepsilon = \Theta(B)$. In this case, the PGM-index takes $\mathcal{O}(m) = \mathcal{O}(n/B)$ space and answers predecessor queries in $\mathcal{O}(\log_B n)$ I/Os, and thus, according to the lower bound of [PT06], it solves I/O-optimally the predecessor search problem in external memory, as B-trees do.

However, compared a B-tree and its variants, the space overhead of a PGM-index does not grow as $\Theta(n/B)$, but it depends on the size $\mathcal{O}(m)$ of the piecewise linear $\varepsilon$-approximation, which in turn depends on the regularity of the input data in the keys-positions Cartesian plane. In particular, with the results obtained in Section 3.2, we can prove that the space of a PGM-index can improve the space of a B-tree from $\Theta(n/B)$ to $\mathcal{O}(m) = \mathcal{O}(n/\varepsilon^2) = \mathcal{O}(n/B^2)$, which is significant considering that $B$ in practical settings is of the order of hundreds or thousands (recall from the beginning of this section that we are not considering the space taken by $A$, which is linear in $n$ for both the B-tree and the PGM-index).

**Corollary 4.1.** *If $A$ satisfies the regularity assumptions of Theorem 3.4, the PGM-index takes $\mathcal{O}(n/B^2)$ space with high probability and answers predecessor queries in optimal $\mathcal{O}(\log_B n)$ I/Os, where $B$ is the block size of the external memory model.*

Finally, for what concerns the RAM model, a straightforward analysis reveals that predecessor queries with a PGM-index run in $\mathcal{O}(\log m + \log \varepsilon)$ time. It is possible to improve this bound by using any RAM-efficient predecessor structure over the set of the first keys covered by the segments in the last level of a PGM-index and discarding the upper levels. For example, with the structure of [BN15, Appendix A], we obtain the following result (note that other space-time trade-offs are possible [NR20]).

**Theorem 4.2.** *In the $w$-bit word RAM model, the PGM-index takes $\mathcal{O}(m)$ space and answers predecessor queries in $\mathcal{O}(\log \log_w \frac{u}{m} + \log \varepsilon)$ time.*

## 4.2 The Dynamic PGM-index

Insertions and deletions in a PGM-index are more difficult to be implemented compared to traditional indexes. First and foremost, the fact that a segment could index a variable and potentially large subset of data makes the classic split and merge algorithms on B-tree nodes inapplicable, as indeed they rely on the fact that a node contains a fixed number $\Theta(B)$ of keys. One could indeed force the segments to cover a fixed number of keys, but this would limit the indexing power of the segments at the core of the PGM-index, which could potentially cover a large number of keys, such as $\mathcal{O}(B^2)$ keys (Theorem 3.1). Existing learned indexes suggest inserting new elements in a sorted buffer for each node (model) which, from time to time, is merged with the main index, thus causing the retraining of the corresponding model [Gal+19; Kra+18]. This solution is inefficient when a model indexes many keys (and thus its retraining is slow), or when the insertions hammer a certain area of the key-space (thus causing many merges due to the rapid filling of few buffers). In this section, we propose two improved strategies for handling updates, one targeted to time series, and the other targeted to more general dynamic scenarios.

If new keys are appended to the end of the array $A$ while maintaining the sorted order, the PGM-index updates the last segment in $\mathcal{O}(1)$ time amortised [ORo81]. If the new key $x$ can be covered by this last segment while preserving the $\varepsilon$ guarantee, then the insertion process stops. Otherwise, a new segment with key $x$ is created. The insertion of $x$ is then repeated recursively in the last segment of the level above. The recursion stops when a segment at any level covers $x$ within the $\varepsilon$ guarantee, or when the root segment is reached. At that point, the root segment might need

splitting and the creation of a new root node with its corresponding segment. Since the work at each level takes constant I/Os amortised, the overall number of I/Os required by this insertion algorithm is $\mathcal{O}(\log_\varepsilon m)$ amortised.

For inserts that occur at arbitrary positions of $A$, we use instead the logarithmic method [BS80; Ove83].[3] We define a series of PGM-indexes built over sets $S_0, \ldots, S_b$ of keys which are either empty or have size $2^0, 2^1, \ldots, 2^b$, where $b = \Theta(\log n)$.

The insert of a key $x$ finds the first empty set $S_i$ and builds a new PGM-index over the merged set $S_0 \cup \cdots \cup S_{i-1} \cup \{x\}$. This union can be computed in time linear in the size of the merged set because the $S_j$s are sorted ($0 \leq j < i$). The new sorted set consists of $2^i$ keys (given that $2^i = 1 + \sum_{j=0}^{i-1} 2^j$). The new merged set is used as $S_i$, and the previous sets are emptied. If we consider one key and examine its history over $n$ insertions, we notice that it can participate in at most $b = \Theta(\log n)$ merges because each merge moves the keys to the right indexes and the full $S_{\log n}$ might include all inserted keys. Given that the merges take time linear in the number of the merged keys, we pay $\mathcal{O}(1)$ time per key at each merge, that is, $\mathcal{O}(\log n)$ time amortised per insertion overall.

The deletion of a key $x$ is handled similarly to an insert by adding a special tombstone value that signals the logical removal of $x$. As soon as the number of tombstones is sufficiently large, say $n/2$, we rebuild the whole data structure by keeping only the non-deleted elements.

A predecessor query is then implemented by combining the results of the predecessor queries in each of the $b$ PGM-indexes, thus taking e.g. $\mathcal{O}(b(\log\log_w \frac{u}{m} + \log \varepsilon))$ time with the solution of Theorem 4.2.

A range query for $[x, y]$ starts with a predecessor query for $x$ and then outputs the keys from each of the $b$ sets, which are sorted, until a key greater than $y$ is found. The cost is thus given by the cost of a predecessor query plus $\mathcal{O}(K)$ time, where $K$ is the number of keys satisfying the range query.

In the external memory model with page size $B$, instead, we follow the ideas of [AV04] and define a series of PGM-indexes with $\varepsilon = \Theta(B)$ built over sets $S_1, \ldots, S_{b'}$ of size *at most* $B^1, \ldots, B^{b'}$, where $b' = \Theta(\log_B n)$.

The insert of a key $x$ finds the first set $S_i$ such that $\sum_{j=1}^{i} |S_j| < B^i$, builds a new PGM-index over the merged set $S_1 \cup \cdots \cup S_i \cup \{x\}$ in $\mathcal{O}(|S_i|/B) = \mathcal{O}(B^{i-1})$ I/Os, and then it empties the sets $S_j$ with $j \leq i$. Since $\sum_{j=1}^{i-1} |S_j| \geq B^{i-1}$, then at least

[3] The logarithmic method underlies also the Log-Structured Merge Tree (LSM-tree) [ONe+96], which is data structure employed in several modern key-value stores [IC20; LC20; DAI18].

$B^{i-1}$ keys are moved to $S_i$, and thus we pay $\mathcal{O}(1)$ I/Os per moved key. Since an element can only move to a larger set, it participates in at most $b'$ moves during $n$ insertions, thus the insertion cost is $\mathcal{O}(b') \cdot \mathcal{O}(1) = \mathcal{O}(\log_B n)$ I/Os amortised.

A predecessor query combines the results from each of the $b'$ PGM-indexes and takes $\mathcal{O}(b' \log_\varepsilon m) = \mathcal{O}((\log_B n)(\log_\varepsilon m))$ I/Os. A range query proceeds as described before, and thus its I/O-complexity is equal to a predecessor query plus $\mathcal{O}(K/B)$ I/Os.

Summing up, we have proved the following.

**Theorem 4.3.** *Given a sorted array $A[1, n]$ and an integer $\varepsilon \geq 1$, the Dynamic PGM-index answers predecessor queries in $\mathcal{O}((\log_B n)(\log_\varepsilon m))$ I/Os, while insertions and deletions take $\mathcal{O}(\log_B n)$ I/Os amortised, where $B$ is the block size of the external memory model. Range queries are answered at the cost of a predecessor query plus extra (optimal) $\mathcal{O}(K/B)$ I/Os, where $K$ is the number of keys satisfying the range query.*

## 4.3 The Compressed PGM-index

Compressing the PGM-index boils down to providing proper lossless compressors for its building blocks, i.e. segments represented via a slope and an intercept.

For what concerns the compression of intercepts, we first recall the Elias-Fano [Eli74; Fan71] representation for compressing and random-accessing monotone integer sequences [Nav16, §4.4].

**Lemma 4.1** (Elias-Fano encoding)**.** *We can store a sequence of $n$ increasing positive integers over a universe of size $u$ in $n\lceil \log \frac{u}{n} \rceil + 2n + o(n) = n \log \frac{u}{n} + \mathcal{O}(n)$ bits and access any integer of the sequence in $\mathcal{O}(1)$ time.*

It turns out that intercepts can be made increasing by using the coordinate system of the segments, i.e. the one that for a segment $s_j = (key_j, \alpha_j, \beta_j)$ computes the position of a key $x$ as $f_{s_j}(x) = \lfloor \alpha_j(x - key_j) + \beta_j \rfloor$. Then, since the result of $f_{s_j}(x)$ is truncated to return an integer position, we represent the intercepts as integers $\lfloor \beta_j \rfloor$.[4] Finally, since these transformations form an increasing sequence of positive integers smaller than $n$, we use Lemma 4.1 to obtain the following result.

**Theorem 4.4.** *Let $m$ be the number of segments of a PGM-index indexing $n$ keys drawn from a universe $\mathcal{U}$. The intercepts of these segments can be stored using $m \log \frac{n}{m} + \mathcal{O}(m)$ bits and be randomly accessed in $\mathcal{O}(1)$ time.*

[4] Note that this transformation increases $\varepsilon$ by 1 in Algorithm 2.

The compression of slopes is more involved, and we need to design a specific novel compression technique. The starting observation is that O'Rourke's algorithm (see Section 2.3) computes not just a single segment but a whole family of $\varepsilon$-approximate segments whose slopes identify an interval of reals. Specifically, let us suppose that the slope intervals for the $m$ optimal segments are $I_1 = (a_1, b_1), \ldots, I_m = (a_m, b_m)$, hence each original slope $\alpha_j$ belongs to $I_j$ for $j = 1, \ldots, m$. The goal of our compression algorithm is to reduce the entropy of the set of these slopes by reducing their distinct number from $m$ to $t$. Given the $t$ slopes, we can store them in a table $T[0, t-1]$ and then change the encoding of each original $\alpha_j$ into the encoding of one of these $t$ slopes, say $\alpha'_j$, which is still guaranteed to belong to $I_j$ but now it can be encoded in $\lceil \log t \rceil$ bits (as a pointer to table $T$).

Let us now describe the algorithm. First, we sort lexicographically the slope intervals $I_j$s to obtain an array $I$ in which overlapping intervals are consecutive. We assume that every pair keeps as satellite information the index of the corresponding interval, namely $j$ for $(a_j, b_j)$. Then, we scan $I$ to determine the maximal prefix of intervals in $I$ that intersect each other. As an example, say the sorted slope intervals are $\{(2, 7), (3, 6), (4, 8), (7, 9), \ldots\}$. The first maximal sequence of intersecting intervals is $\{(2, 7), (3, 6), (4, 8)\}$ because these intervals intersect each other, but the fourth interval $(7, 9)$ does not intersect the second interval $(3, 6)$ and thus is not included in the maximal sequence.

Let $(l, r)$ be the intersection of all the intervals in the current maximal prefix of $I$: it is $(4, 6)$ in the running example. Then, any slope in $(l, r)$ is an $\varepsilon$-approximate slope for each of the intervals in that prefix of $I$. Therefore, we choose one real in $(l, r)$ and assign it as the slope of each of those segments in that maximal prefix. The process then continues by determining the maximal prefix of the remaining intervals, until the overall sequence $I$ is processed.

**Theorem 4.5.** *Let $m$ be the number of $\varepsilon$-approximate segments of a PGM-index indexing $n$ keys drawn from a universe $\mathcal{U}$. There exists a lossless compressor for the segments which computes the minimum number of distinct slopes $t \leq m$ while preserving the $\varepsilon$-guarantee. The algorithm takes $\mathcal{O}(m \log m)$ time and compresses the slopes into $64t + m\lceil \log t \rceil$ bits of space.*

*Proof.* The choice performed by the algorithm is to keep adding slope intervals in lexicographic order and updating the current intersection $(l, r)$ until an interval $(a_j, b_j)$ having $a_j > r$ arrives. It is easy to verify that an optimal solution has slopes within the $t$ intersection intervals found by this algorithm. The space occupancy of the $t$ distinct slopes in $T$ is, assuming double-precision floats, $64t$ bits. The new

slopes $\alpha'_j$ are still $m$ in their overall number, but each of them can be encoded as the position $0, \ldots, t-1$ into $T$ of its corresponding double-precision float. □

## 4.4 The Distribution-Aware PGM-index

The PGM-index of Theorem 4.1 implicitly assumes that the queries are uniformly distributed, but this seldom happens in practice. For example, queries in search engines are very well known to follow skewed distributions such as Zipf's law [WMB99]. In such cases, it is desirable to have an index that answers the most frequent queries faster than the rare ones, so to achieve a higher query throughput. Previous work exploited query distribution in the design of binary trees [BST85], treaps [SA96], and skip lists [BBG05], to mention a few.

In this section, we introduce a variant of the PGM-index that adapts itself not only to the distribution of the input keys but also to the distribution of the queries. This turns out to be the *first distribution-aware learned index* to date, with the additional positive feature of being very succinct in space.

Formally speaking, given a sequence $S = \{(k_i, p_i)\}_{i=1,\ldots,n}$, where $p_i$ is the probability to query the key $k_i$ (that is assumed to be known), we want to solve the distribution-aware dictionary problem, which asks for a data structure that searches for a key $k_i$ in time $\mathcal{O}(\log(1/p_i))$ so that the average query time coincides with the entropy of the query distribution $\mathcal{H} = \sum_{i=1,\ldots,n} p_i \log(1/p_i)$.

Recall from section 2.3 that O'Rourke's algorithm (Lemma 2.1) finds the set of all segments that intersect some given vertical ranges in $\mathcal{O}(n)$ time. Our key idea is to define for every key $k_i$ a vertical range of size $y_i = \min\{1/p_i, \varepsilon\}$, and then to apply Lemma 2.1 on that set of keys and vertical ranges. Clearly, for the keys whose vertical range is $\varepsilon$ we can use Lemma 2.1 and derive the same space bound of $\mathcal{O}(m)$. For the keys whose vertical range is $1/p_i < \varepsilon$, we observe that they are no more than $\varepsilon$ (in fact, the $p_i$s sum up to 1), but they are possibly spread among all positions in $A$, and thus they induce in the worst case $2\varepsilon$ extra segments. Therefore, the total space occupancy of the bottom level of the index is $\Theta(m + \varepsilon)$, where $m$ is the one defined in Theorem 4.1. Now, let us assume that the search for a key $k_i$ arrived at the last level of this Distribution-Aware PGM-index, and thus we know in which segment to search for $k_i$: the final binary search step within the $\varepsilon$-approximate range returned by that segment takes $\mathcal{O}(\log \min\{1/p_i, \varepsilon\}) = \mathcal{O}(\log(1/p_i))$ as we aimed for.

We are left with showing how to find that segment in a distribution-aware manner, namely in $\mathcal{O}(\log(1/p_i))$ time. We proceed similarly to the recursive construction

of the PGM-index, but with a careful design of the recursive step because of the probabilities (and thus the variable vertical ranges) assigned to the recursively defined set of keys.

Let us consider the segment $s_{[a,b]}$ covering the keys in $S_{[a,b]} = \{(k_a, p_a), \ldots, (k_b, p_b)\}$, denote by $q_{a,b} = \max_{i \in [a,b]} p_i$ the maximum probability of a key in $S_{[a,b]}$, and by $P_{a,b} = \sum_{i=a}^{b} p_i$ the cumulative probability of all keys in $S_{[a,b]}$ (which is indeed the probability to end up in that segment when searching for one of its keys). To move to the next upper level of the PGM-index, we create a new set of keys which includes the first key $k_a$ covered by each segment $s_{[a,b]}$ and set its associated probability to $q_{a,b}/P_{a,b}$. Then, we construct the next upper level of the Distribution-Aware PGM-index by applying Lemma 2.1 to this new set of segments. If we iterate the above analysis for this new level of "weighted" segments, we conclude that: if we know from the search executed on the levels above that $k_i \in S_{[a,b]}$, the time cost to search for $k_i$ in this level is $\mathcal{O}(\log \min\{P_{a,b}/q_{a,b}, \varepsilon\}) = \mathcal{O}(\log(P_{a,b}/p_i))$.

Let us repeat this argument for another upper level to understand the influence on the search time complexity. We denote the range of keys which include $k_i$ in the upper level with $S_{[a',b']} \supset S_{[a,b]}$, the cumulative probability with $P_{a',b'}$, and assign to the first key $k_{a'} \in S_{[a',b']}$ the probability $r/P_{a',b'}$, where $r$ is the maximum probability of the form $P_{a,b}$ of the ranges included in $[a', b']$. In other words, if $[a', b']$ is partitioned into $\{z_1, \ldots, z_c\}$, then $r = \max_{i \in [1,c)} P_{z_i, z_{i+1}}$. Reasoning as done previously, if we know from the search executed on the levels above that $k_i \in S_{[a',b']}$, the time cost to search for $k_i$ in this level is $\mathcal{O}(\log \min\{P_{a',b'}/r, \varepsilon\}) = \mathcal{O}(\log(P_{a',b'}/P_{a,b}))$ because $[a, b]$ is, by definition, one of these ranges in which $[a', b']$ is partitioned.

Repeating this design until one single segment is obtained, we get a total time cost for the search in all levels of the PGM-index equal to a sum of logarithms whose arguments "cancel out" (i.e. a telescoping sum) and get $\mathcal{O}(\log(1/p_i))$.

As far as the space bound is concerned, we recall that the number of levels in the PGM-index is $l = \mathcal{O}(\log_\varepsilon m)$, and that we have to account for the $\varepsilon$ extra segments per level returned by the algorithm of Theorem 2.1. Consequently, this distribution-aware variant of the PGM-index takes $\mathcal{O}(m + l\varepsilon)$ space, which is indeed $\mathcal{O}(m)$ because $\varepsilon$ is a constant parameter.

**Theorem 4.6.** *Given a sorted array $A[1, n]$ of keys with corresponding (known) query probabilities $p_1, \ldots, p_n$, and given an integer $\varepsilon \geq 1$, the Distribution-Aware PGM-index indexes the array $A$ in $\mathcal{O}(m)$ space and answers queries in $\mathcal{O}(\mathcal{H})$ average time, where $\mathcal{H}$ is the entropy of the query distribution, and $m$ is the size of the optimal piecewise linear $\varepsilon$-approximation built on $A$.*

## 4.5 The Multicriteria PGM-index

Tuning a data structure to match the requirements of an application is often a difficult and error-prone task for a software engineer, not to mention that these needs may change over time due to mutations in data distribution, devices, resource requirements, and so on.

The typical approach is a grid search on the various instances of the data structure to be tuned until the one that matches the application's requirements is found. However, not all data structures are flexible enough to adapt at the best to these requirements, or conversely, the search space can be so large that an optimisation process takes too much time [Idr+18; Idr+19].

In the rest of this section, we exploit the flexible design of the PGM-index to show that the tuning to any space-time requirements can be efficiently automated via an optimisation strategy that: (i) given a space constraint outputs the PGM-index that minimises its query time; or (ii) given a query-time constraint outputs the PGM-index that minimises its space footprint.

**The time-minimisation problem.** According to Theorem 4.1, the query time of a PGM-index can be described as $t(\varepsilon) = \delta\,(\log_{2\varepsilon} m)\log(2\varepsilon/B)$, where $B$ is the block size of the external memory model, $m$ is the number of segments in the last level, and $\delta$ depends on the access latency of the memory. For the space, we introduce $s_i(\varepsilon)$ to denote the minimum number of segments needed to have precision $\varepsilon$ over the keys available at level $i$ of the PGM-index and compute the overall number of segments as $s(\varepsilon) = \sum_{i=1}^{l} s_i(\varepsilon)$. By Lemma 3.1, we know that $s_l(\varepsilon) = m \leq n/(2\varepsilon)$ for any $\varepsilon \geq 1$ and that $s_{i-1}(\varepsilon) \leq s_i(\varepsilon)/(2\varepsilon)$. As a consequence, $s(\varepsilon) \leq \sum_{i=0}^{l} m/(2\varepsilon)^i = (2\varepsilon m - 1)/(2\varepsilon - 1)$.

Given a space bound $s_{max}$, the "time-minimisation problem" consists of minimising $t(\varepsilon)$ subject to $s(\varepsilon) \leq s_{max}$.[5] The main challenge here is that we do not have a closed formula for $s(\varepsilon)$ but only an upper bound. In practice, we can model $m = s_l(\varepsilon)$ with a simple power law having the form $a\varepsilon^{-b}$, whose parameters $a$ and $b$ are properly estimated on the dataset at hand. The power law covers both the pessimistic case of Lemma 3.1 and the best case in which the dataset is strictly linear.

Clearly, the space decreases with increasing $\varepsilon$, whereas the query time $t(\varepsilon)$ increases with $\varepsilon$ since the number of keys on which a binary search is executed at each level is $2\varepsilon$. Therefore, the time-minimisation problem reduces to the problem of finding the

[5] For simplicity, we assume that a disk page contains exactly $B$ keys. This assumption can be relaxed by putting the proper machine- and application-dependent constants in front of $t(\varepsilon)$ and $s(\varepsilon)$.

value of $\varepsilon$ for which $s(\varepsilon) = s_{max}$ because it is the lowest $\varepsilon$ that we can afford. Such value of $\varepsilon$ could be found by a binary search in the bounded interval $\mathcal{E} = [B/2, n/2]$, which is derived by requiring that each model errs at least a page size (i.e. $2\varepsilon \geq B$) since lower $\varepsilon$ values do not save I/Os, and by observing that one model is the minimum possible space (i.e. $2\varepsilon \leq n$, by Lemma 3.1). Furthermore, provided that our power-law approximation holds, we can speed up the search of that "optimal" $\varepsilon$ by guessing the next value of $\varepsilon$ rather than taking the midpoint of the current search interval. In fact, we can find the root of $s(\varepsilon) - s_{max}$, i.e. the value $\varepsilon_g$ for which $s(\varepsilon_g) = s_{max}$. We emphasise that such $\varepsilon_g$ may not be the solution to our problem, as it may be the case that the approximation or the fitting of $s(\varepsilon)$ by means of a power-law is not precise. Thus, more iterations of the search may be needed to find the optimal $\varepsilon$. Nevertheless, we guarantee to be always faster than a binary search by gradually switching to it. Precisely, we bias the guess $\varepsilon_g$ towards the midpoint $\varepsilon_m$ of the current search range via a simple convex combination of the two [Gra06].

**The space-minimisation problem.** Given a time bound $t_{max}$, the "space-minimisation problem" consists of minimising $s(\varepsilon)$ subject to $t(\varepsilon) \leq t_{max}$. As for the problem above, we could binary search inside the interval $\mathcal{E} = [B/2, n/2]$ for the maximum $\varepsilon$ that satisfies the time constraint. Additionally, we could speed up this process by guessing the next iterate of $\varepsilon$ via the equation $t(\varepsilon) = t_{max}$, that is, by solving for $\varepsilon$ the equation $\delta(\log_{2\varepsilon} s_l(\varepsilon)) \log(2\varepsilon/B) = t_{max}$ in which $s_l(\varepsilon)$ is replaced by the power-law approximation $a\varepsilon^{-b}$ for proper $a$ and $b$, and $\delta$ is replaced by the measured memory latency of the given machine.

Although effective, this approach raises a subtle issue, namely, the time model could not correctly estimate the actual query time because of hardware-dependent factors such as the presence of several CPU caches. To further complicate this issue, we note that both $s(\varepsilon)$ and $t(\varepsilon)$ depend on the power-law approximation.

For these reasons, instead of using the time model $t(\varepsilon)$ to steer the search, we measure and use the actual average query time $\bar{t}(\varepsilon)$ of the PGM-index over a fixed batch of random queries. Also, instead of binary searching inside the whole $\mathcal{E}$, we run an exponential search starting from the solution of the dominating term $c \log(2\varepsilon/B) = t_{max}$, i.e. the cost of searching the data. Finally, since $\bar{t}(\varepsilon)$ is subject to measurement errors (e.g. due to an unpredictable CPU scheduler), we stop the search as soon as the searched range is smaller than a given threshold.

## 4.6 Experiments

We experiment with a C++ implementation of the PGM-index, publicly available at `https://github.com/gvinciguerra/PGM-index`, on a machine with 202 GB of RAM and a 2.30 GHz Intel Xeon Gold 5118 CPU. We use these sorted datasets from [Mar+20], all containing 64-bit unsigned integer keys:

**books** 800M keys representing book popularity data from Amazon.

**osm** 800M OpenStreetMap nodes locations represented using the cell ID numbering from Google's S2 library.

**face** 200M randomly sampled Facebook user IDs.

**wiki** 200M timestamps of edits to Wikipedia pages.

### 4.6.1 Implementation notes

We store the segments of a PGM-index contiguously in level-wise order as triples with a 64-bit unsigned integer key, a floating-point slope, and a 32-bit integer intercept. For the levels above the last one (e.g. $levels[1, 2]$ in Figure 4.1), we found that setting $\varepsilon$ to a small value (e.g. $\varepsilon = 4$ in the experiments below) and using a linear search improves the query efficiency without impacting too much the space, as the number of segments contracts rapidly at each level. Moreover, we do not perform any bounds checking in Line 5 of Algorithm 2 by using a special segment with key $2^{64} - 1$ at the end of each level. Instead, at Line 10, we use a binary search.

For the construction, we parallelise Algorithm 1 by slicing the *keys* array into $p$ equal-sized subarrays, where $p$ is the number of processors, and computing a piecewise linear $\varepsilon$-approximation separately on each subarray. This way, we have an embarrassingly parallel computation at the cost of adding only at most $p-1$ segments compared to the sequential one (indeed, the sequential algorithm could have possibly computed a longer segment in correspondence of the end of a subarray).

We observe here that a PGM-index can be easily modified to index an array $A$ with repeated keys by feeding O'Rourke's algorithm (see Section 2.3 with pairs $(x, \mathsf{rank}(x))$, where $x$ ranges over all the sorted and distinct keys of $A$. However, as a technicality, at the end of a run of repeated keys with value $x$, if the next key $y$ is not equal to $x + 1$, we need to consider the additional pair $(x + 1, \mathsf{rank}(x))$ so that the output piecewise linear $\varepsilon$-approximation will map the keys in the interval $[x + 1, y)$ to their correct predecessor position $\mathsf{rank}(x)$.

For the Dynamic PGM-index with base $B$ (cf. Section 4.2), we combine the first $j$ sets into one larger set of size $B^1 + \cdots + B^j$, implemented as an array, and perform sorted insertions there. The value $j$ can be tuned so that the performance of an insertion in the sorted array is better than the performance of managing $j$ distinct sets. Moreover, we do not associate a PGM-index to the smaller sets when a simple binary search is more efficient than constructing and querying an index on that set (typically, when the corresponding array fits the processor caches).

When loading an empty Dynamic PGM-index from sorted data of size $n$, we simply copy the data to the $\lceil \log_B n \rceil$th set and leave the other sets empty. When inserting a key and combining several sets, we merge the sets pairwise starting from the smallest ones, that is, we merge $S_1$ with $S_2$ to produce $S_{1:2}$, then we merge $S_{1:2}$ with $S_3$ to produce $S_{1:3}$ and so on.

### 4.6.2 Static scenario

We start by evaluating the performance of some static indexing predecessor structures built on the keys loaded contiguously in memory and interleaved with 8-byte random payloads. We test our PGM-index, our re-implementation of a static cache-aware version of the B$^+$-tree named CSS-tree [RR99], and the RMI of [Kra+18] optimised by [Mar+20].[6] For the PGM-index, we vary $\varepsilon$ on the last level of the index as $2^3, \ldots, 2^{12}$ and fix $\varepsilon = 4$ on the upper levels. For CSS-tree, we vary the page size so that each node contains $B = 2^4, \ldots, 2^{13}$ routing keys (i.e. a leaf node locates $B$ keys as the corresponding PGM-index do, since $B = 2\varepsilon$). For RMI, we set a number of linear regression models in such a way that the final space occupancy matches the one of the PGM-index, thus yielding a head-to-head comparison.

We generate a batch of 10M predecessor queries chosen uniformly at random from the input universe[7] and measure the average query time in nanoseconds and the space occupancy of the index structure.

[6] We skip a direct evaluation of some learned indexes proposed recently, such as FITing-tree [Gal+19], which is based on a sub-optimal number of segments — on average 2.20× larger than the optimal one of PGM-index over the $\varepsilon$ values and datasets considered here — and a space-consuming B$^+$-tree over the segments. Indeed, in our previous experiments [FV20b], we showed that the recursive design of the PGM-index is superior to the ones based on binary search or multiway search trees. We also skip Radix Spline [Kip+20], which is essentially a flat and static FITing-tree variant with a lookup table implementing one step of length reduction (see the introduction of this chapter) to speed up a binary search on the segments. This approach easily degenerates to a full binary search on the segments when their keys are not uniformly scattered in $\mathcal{U}$.

[7] This contrasts with the evaluation of [Mar+20; MD21] and our previous experiments in [FV20b], which *only* tested positive lookups, i.e. equality searches of keys taken randomly from the input array $A$. Neglecting the efficiency of an index on general input queries from $\mathcal{U}$, especially if it is built by learning the keys-positions mapping, could lead to poor index choices in some applications. See the discussion in Section 2.2, and Figure 4.3 commented below.

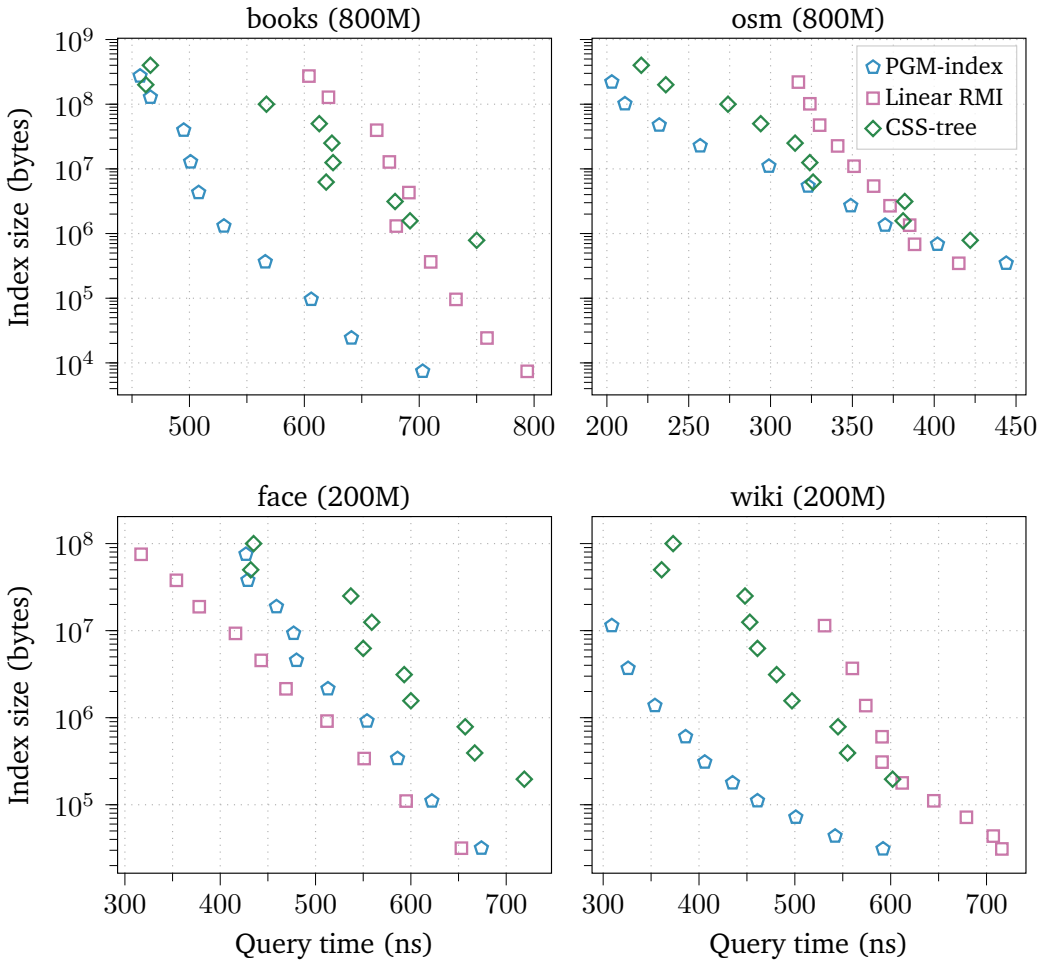



**Figure 4.2:** The space-time performance of three indexing structures — PGM-index, linear RMI, and CSS-tree — in a static scenario with no updates.

The results in Figure 4.2 show that the PGM-index is on average 10.72× as much compressed as and 8.23% faster than the corresponding CSS-tree (i.e. the same setting of page size according to the equivalence $B = 2\varepsilon$). Compared to the Linear RMI with the same space, PGM-index is on average 22.54% faster on books, osm, wiki, and it is 13.20% slower on face.

We also experiment with the long-running hyperparameter search that finds ten *Hybrid* RMIs (with linear and non-linear models) on the space-time Pareto frontier for a given machine and dataset [Mar+20]. The results in the left plot of Figure 4.3 show that these hybrid configurations improve the Pareto frontier of PGM-index.

However, even these tuned RMIs are not robust indexing choices due to the lack of efficient worst-case bounds. Indeed, as shown in the right plot of Figure 4.3, when an adversary issues queries for the universe keys where RMI has a large prediction

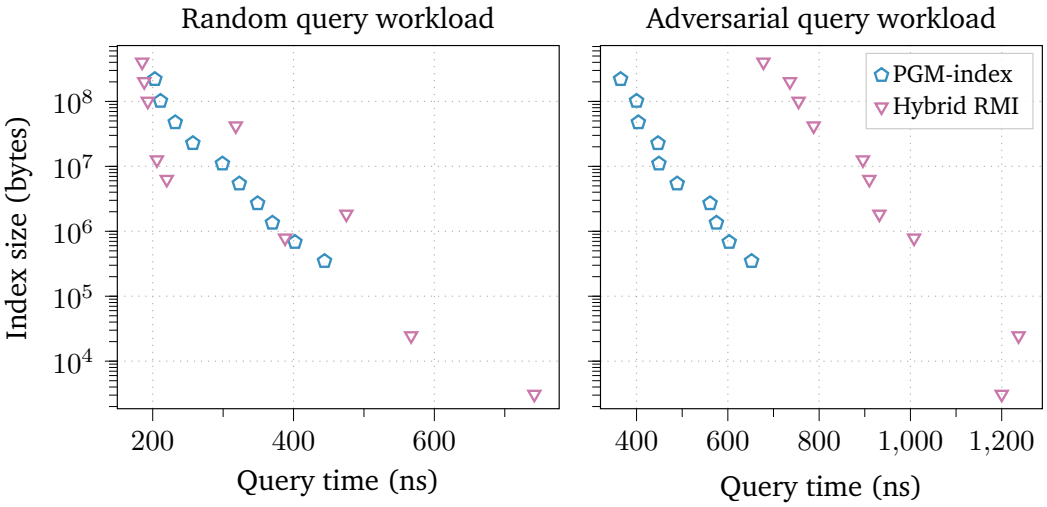



**Figure 4.3:** The performance of tuned Hybrid RMIs (with linear and nonlinear models) built on the large osm dataset (800M keys). The left plot shows a random query workload, and the right plot shows an adversarial query workload crafted for the Hybrid RMIs. The much-degraded performance of RMI shows the importance of designing index structures with worst-case bounds, as the PGM-index.

error, RMI gets much slower due to the increased number of binary search steps to find the correct position.[8] This complication, which has been overlooked by prior work [Kra+18; Din+20; Mar+20], is solved by the PGM-index by providing efficient worst-case guarantees for any input data and query keys (Theorem 4.1).

### 4.6.3 Dynamic scenario

We now experiment with the performance of some dynamic predecessor structures, namely our Dynamic PGM-index, ALEX [Din+20], the ART [LKN13] engineered by [Dau21], the B-tree engineered by [Goo17], the B$^+$-tree engineered by [Bin18], and the Y-fast trie engineered by [DFH21].

We take our two largest datasets, books and osm (800M keys), and for each of them, we generate five batches of 100M operations by extracting uniformly at random 0, 25M, 50M, 75M, 100M keys to be inserted and filling the rest of the batch with query operations (50% predecessor queries, 50% lookups). The batch is then shuffled, and the non-extracted keys ($\geq$ 700M) are used to bulk load each data structure. Again, we consider a scenario with 8-byte keys associated with 8-byte random values.

[8] See also the poisoning attacks introduced by [KRT20], whereby an attacker inserts maliciously-crafted keys to increase the prediction error of RMI of up to 3000×.

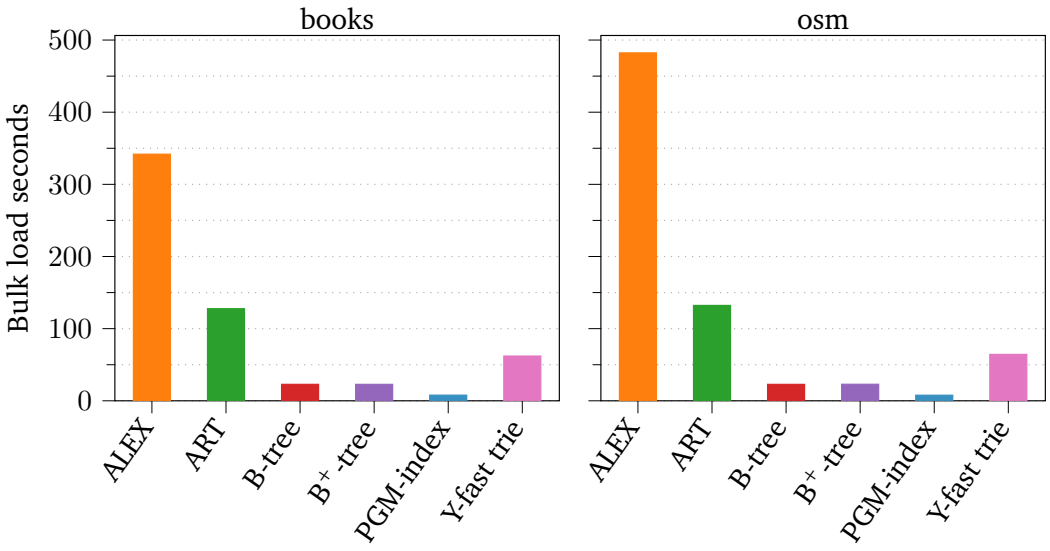



**Figure 4.4:** The time to bulk load each dynamic predecessor structure with over 700M sorted key-value pairs. These results are particularly interesting in applications where an index must be rebuilt frequently.

As some of the data structures allow configuring some kind of block-size parameter, we vary this parameter in powers-of-two as follows and show only the fastest configuration for a given batch. For the B-tree and B$^+$-tree, we vary the page size from 256 to 4096 bytes. For the PGM-index, we vary the base of the logarithmic method from 2 to 64.[9] For the engineered Y-fast trie, we vary the (sorted) bucket size from 64 to 512, as in [DFH21].

**Bulk load time.** As shown in Figure 4.4, the average time to bulk load a data structure from sorted data is low for PGM-index, B-tree and B$^+$-tree, while it is high for Y-fast trie, ART, and ALEX, whose times are 8.07×, 16.59×, and 52.60× that of PGM-index, respectively. This is because the more complex structures of Y-fast trie, ART and ALEX require more computationally expensive bulk loading algorithms with respect to the simple data copies required by PGM-index, B-tree and B$^+$-tree.

**Average latency.** We show in Figure 4.5 the average latency in nanoseconds per operation on the various query-insert batches, and we observe that:

- PGM-index is faster for insert-heavy workloads (0% to 25% queries).
- PGM-index and ART are faster for balanced workloads (50% queries).

[9]Here, we only vary the base of the logarithmic method and not $\varepsilon$ (which we fix to 16 on the last level and 4 on the upper levels) to not over-tune our data structure and keep the comparison fair.

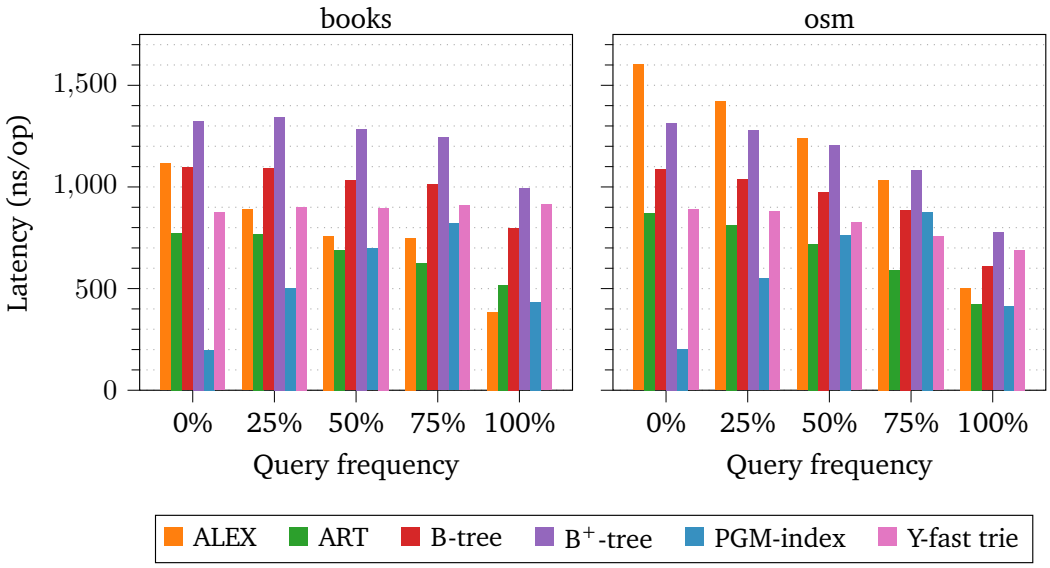



**Figure 4.5:** The average latency of some dynamic predecessor structures on various batches of 100M query-insert operations with varying query frequency.

- ART is faster for query-heavy workloads (75% queries).
- PGM-index, ART and ALEX are faster for query-only workloads (100% queries).

Therefore, despite its algorithmic simplicity, the Dynamic PGM-index here achieves efficient operations over a large spectrum of workloads. This is because of the combination of the logarithmic method — which allows fast amortised inserts of new elements — and the static PGM-indexes built on each set of the logarithmic structure — which allows solving the multiple steps of a query to this structure efficiently. Interestingly, we found that the best performance of the Dynamic PGM-index is achieved with a logarithmic method in: (i) base 2 for workloads with 0% queries; (ii) bases 4 and 8 for workloads with 25% queries; (iii) base 8 for workloads with 50% queries; (iv) base 16 for workloads with 75% queries.[10] Therefore, as the number of queries in the workload increases, the base of the logarithmic method should be set to a higher value, so that the number of sets to be searched decreases.

**Space usage.** We measure the overall space usage of each data structure after completing each batch. Note that, unlike in Section 4.6.2 where the input data is read-only and thus only the index size is considered, here the space usage includes both the space of the index structure and the size of the data. Indeed, the tested data structures use different strategies to store the data (e.g. by arranging the keys in a

[10]The base to be used in a query-only workload is not interesting since only one set in the Dynamic PGM-index is non-empty and queried, i.e. the Dynamic PGM-index is equivalent to a static one.

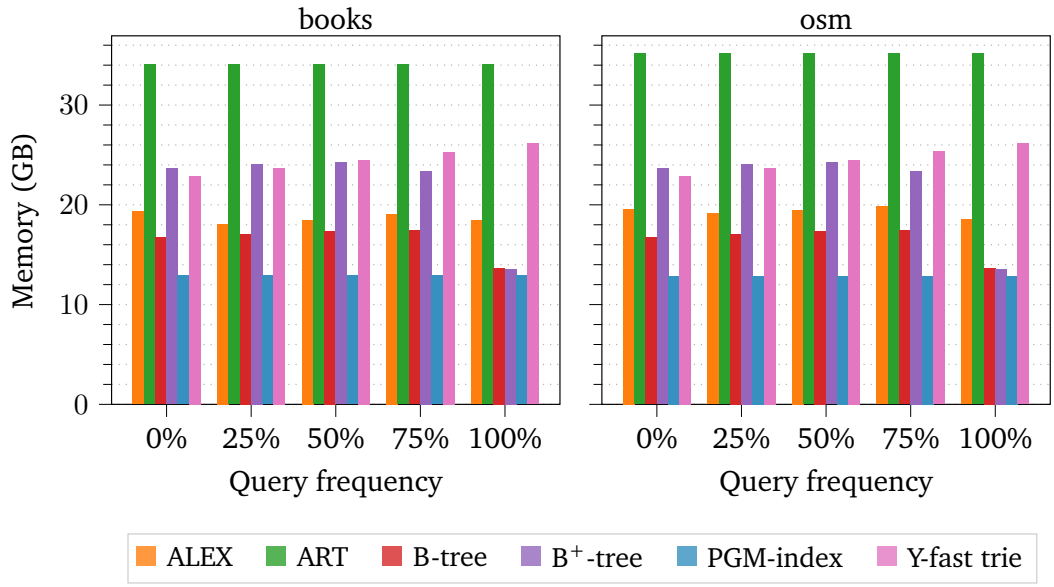



**Figure 4.6:** The overall space usage of some dynamic predecessor structures measured after running the batches of Figure 4.5.

trie-like structure like in ART, or by using half-full nodes to accommodate insertions like in B-tree) that impact on the overall space occupancy (other than on the query-insert efficiency). We show the results in Figure 4.6 and observe that PGM-index is the most space-efficient (12.90 GB on average), followed by B-tree (27.54% more than PGM-index), ALEX (49.76% more), B$^+$-tree (68.86% more), Y-fast trie (89.78% more), and ART (168.17% more). This space efficiency of PGM-index is due to the use of the logarithmic method — which saves space by not allocating half-empty nodes/slots, as instead the other data structures do — and the use of the static PGM-indexes — which constitute a very succinct indexing mechanism.

**Range queries.** After completing each query-insert batch, we present each dynamic predecessor structure with three batches containing 50M range queries, each giving result sizes of 10, 1K and 100K elements, respectively. All the data structures return results sorted by key. Figure 4.7 shows the average range query time for each result size.[11] We observe that for a small range size (10 results) all data structures take from about 2 to 4 $\mu$s and ALEX is the fastest. For a medium range size (1K results), Y-fast trie is the fastest (7.00 $\mu$s on average), followed by PGM-index (65.26% more than Y-fast trie), and ALEX (557.90% more). For large range sizes (100K results), Y-fast trie is still the fastest (609.17 $\mu$s on average), followed by PGM-index (2.95% more than Y-fast trie), and ALEX (634.88% more).

[11] We compute the average over the range query times of each dynamic predecessor structure at the state it is after running the query-insert batches of Figure 4.5. We exclude the query-only batch (100% queries) from this computation of the average.

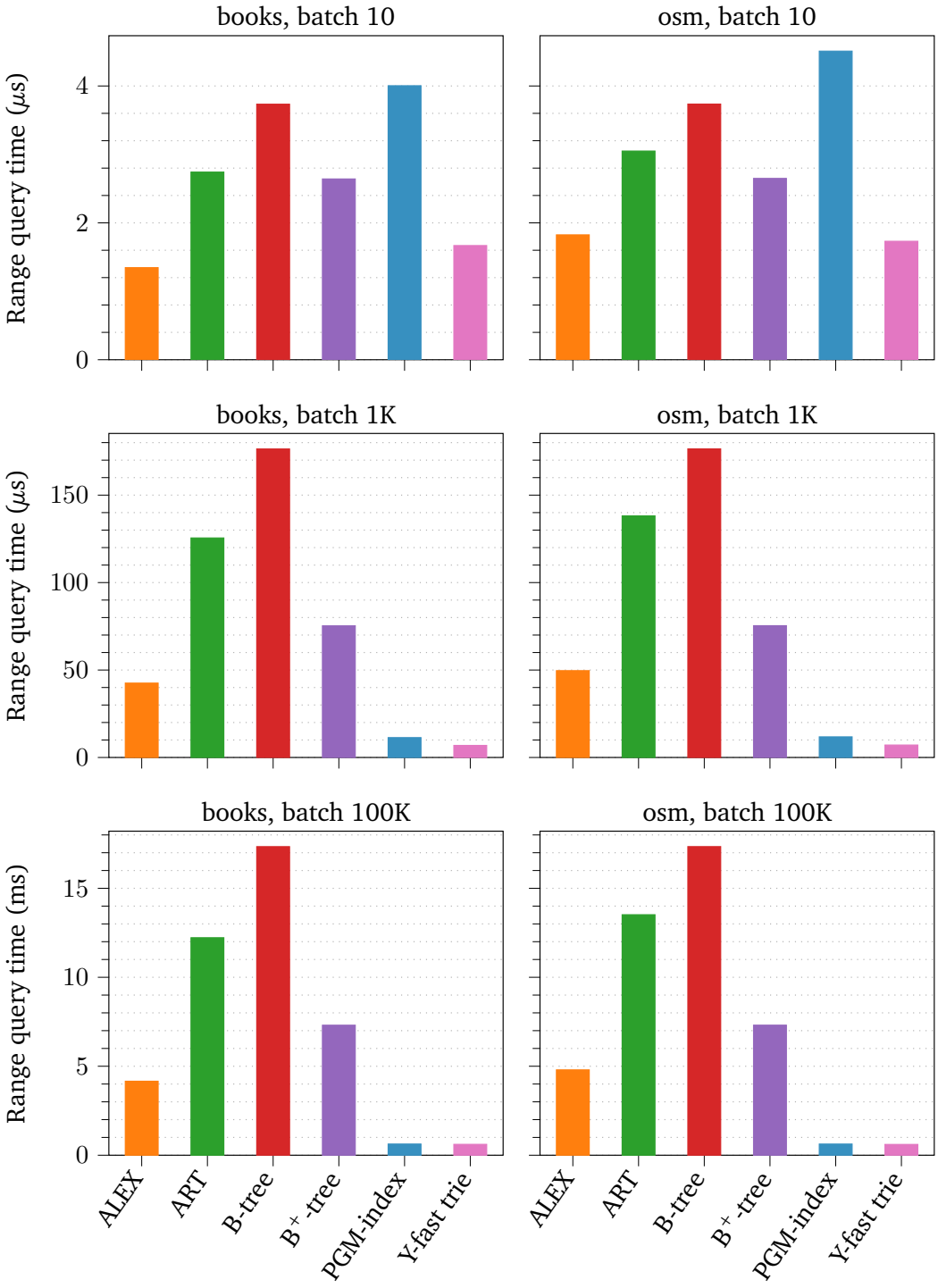



**Figure 4.7:** The range query time of some dynamic predecessor structures on two datasets (which correspond to the two columns) and three range result sizes (which correspond to the three rows).

## 4.7 The PGM-index software library

In this section, we describe the PGM-index software library, which is publicly available at `https://pgm.di.unipi.it`. The library is written in C++17, and it also provides a C interface to ease the interoperability with (and the creation of wrappers for) other programming languages.[12]

The main classes implemented in the library are `PGMIndex`, `CompressedPGMIndex`, `DynamicPGMIndex`, which correspond to the data structures previously described in Sections 4.1 to 4.3, respectively.

An example of usage of the `PGMIndex` class is given in Listing 4.1. Specifically, Lines 9 to 11 create, fill and sort a vector `data` of random integers. Line 14 constructs a PGM-index on `data` with $\varepsilon = 128$ on the last level and $\varepsilon' = 4$ on the upper levels. Lines 17 and 18 use the PGM-index to find the `range` of positions where a query key `q` can be found in `data`. Lines 19 to 21 perform the final $\mathcal{O}(\log \varepsilon)$-time binary search on $\texttt{data}[\texttt{range.lo}, \texttt{range.hi} - 1]$ and print the result.

```
1  #include <vector>
2  #include <cstdlib>
3  #include <iostream>
4  #include <algorithm>
5  #include "pgm/pgm_index.hpp"
6
7  int main() {
8      // Generate some random data
9      std::vector<int> data(1000000);
10     std::generate(data.begin(), data.end(), std::rand);
11     std::sort(data.begin(), data.end());
12
13     // Construct the PGM-index
14     pgm::PGMIndex<int, 128, 4> index(data);
15
16     // Query the PGM-index
17     auto q = 42;
18     auto range = index.search(q);
19     auto lo = data.begin() + range.lo;
20     auto hi = data.begin() + range.hi;
21     std::cout << *std::lower_bound(lo, hi, q);
22
23     return 0;
24 }
```

**Listing 4.1:** Example of usage of the `PGMIndex` class.

[12] For example, the library was used to create the PyGM package for Python 3, which provides sorted set and list containers backed by the PGM-index for efficient query operations. PyGM is available at `https://github.com/gvinciguerra/PyGM`.

Additionally, the library provides several implementation variants of `PGMIndex`, which replace the recursive structure on the last-level segments (e.g. $levels[1, 2]$ in Figure 4.1) with other techniques, as detailed below.

- The `OneLevelPGMIndex` class, which uses a binary search on the segments.
- The `BucketingPGMIndex` class, which uses a table $T$ of a user-given size $|T|$ implementing one step of length reduction to possibly reduce the number of binary search steps on the segments, i.e. a table $T$ such that the segment responsible for a query key $q$ can be found after a binary search restricted to the positions given by $T[i]$ and $T[i + 1]$ for $i = \lfloor q/(u/|T|) \rfloor$.[13]
- The `EliasFanoPGMIndex` class, which uses Lemma 4.1 to store the segments' keys in $\mathcal{O}(m \log \frac{u}{m})$ bits and finds the segment responsible for a query key in $\mathcal{O}(\min\{\log m, \log \frac{u}{m}\})$ time via the well-known search algorithm [Nav16, §4.4.2].

The library also provides the `MultidimensionalPGMIndex` class, which implements a data structure for orthogonal range queries in $k$ dimensions, that is, a data structure that stores a set $S \subseteq \mathcal{U}^k$ and answers reporting queries of the kind $S \cap Q$, where $Q = [a_1, b_1] \times \cdots \times [a_k, b_k]$ is a given $k$-dimensional query rectangle (with $0 \leq a_i \leq b_i < u$ for $i = 1, \ldots, k$). Specifically, the `MultidimensionalPGMIndex` transforms and sorts the input points using Morton codes[14] and implements the orthogonal range query using a combination of the PGM-index and the algorithm of [TH81, §4] to efficiently locate the elements in $S \cap Q$.

Finally, we mention that the library also provides:

1. an implementation of the tuner described in Section 4.5 (under the `tuner/` directory of the GitHub repository);
2. some benchmarking code (under the `benchmark/` directory of the GitHub repository) to evaluate and plot the space-time efficiency of the PGM-index variants described above on user-given data and queries;
3. a `MappedPGMIndex` class to ease the creation of a memory-mapped file backed by the PGM-index.

[13] Observe that when both the universe size $u$ and $|T|$ are powers of two, i.e. $u = 2^a$ and $|T| = 2^b$ for some $a \geq b$, the computation of $i$ reduces to $i = \lfloor q/2^{a-b} \rfloor$, which involves a simple bit shift operation.

[14] The Morton or z-order code of a $k$-dimensional point is simply the result of interleaving the binary representations of its $k$ coordinates. For example, the three-dimensional point $(4, 7, 1) = (100, 111, 001)$ is encoded as $011\ 010\ 110 = 214$.

## 4.8 Summary

We designed the PGM-index, a data structure for predecessor search that uses a recursive structure based on piecewise linear $\varepsilon$-approximations to achieve efficient space-time trade-offs, especially in the external memory model (Theorems 4.1 and 4.3). For particular kinds of input data, we showed that the PGM-index improves the space of the de facto standard solution in this scenario, the B-tree (Corollary 4.1). We also compressed the segments at the core of the PGM-index (Theorems 4.4 and 4.5) and made the query time of the PGM-index bounded by the entropy of a given query distribution (Theorem 4.6). Finally, our large set of experiments showed that: (i) in the static setting, the PGM-index provides robust query time performance compared to other learned methods thanks to its worst case bounds, and that it is 10.72× as much compressed as and 8.23% faster than static and cache-aware $B^+$-tree having the same theoretical guarantee on the query time; (ii) in the dynamic setting, the PGM-index provides the fastest performance in insert-heavy workloads while always being the most space-efficient when compared to five known predecessor structure implementations.

A part of the results in this chapter appeared in [FV20b]. The PGM-index library is publicly available at `https://pgm.di.unipi.it`.

# 5 Rank/select dictionaries

In this chapter, we consider the problem of representing, in compressed form, an ordered dictionary $A$ of $n$ elements drawn from the integer universe $[u] = \{0, \ldots, u-1\}$ while supporting the following operations:

- $\mathsf{rank}(x)$. Given $x \in [u]$, return the number of elements in $A$ that are less than or equal to $x$;
- $\mathsf{select}(i)$. Given $i \in \{1, \ldots, n\}$, return the $i$th smallest element in $A$.

This problem is intimately related to the predecessor search problem we saw in Chapter 4. Indeed, to return the predecessor of $x \in A$, it suffices to execute $y = \mathsf{select}(\mathsf{rank}(x))$. But here, compared to Chapter 4, we have an additional requirement to represent $A$ in compressed form, while still being able to access its elements via select and to perform query operations via rank.

Rank/select dictionaries are at the heart of virtually any compact data structure [Nav16], such as text indexes [FM05; GV05; NM07; MN07; Gog+19; GNP20], succinct trees and graphs [MR97; RRS07], hash tables [Bel+08], permutations [BN09], etc. Unsurprisingly, the literature is abundant in solutions, as we will review in Section 5.1. Yet, the problem of designing theoretically and practically efficient rank/select structures is anything but closed. The reason is threefold. First, there is an ever-growing list of applications of compact data structures (in bioinformatics [Fer+18; Mäk+15], information retrieval [Nav14], and databases [AKS15; Ram18], just to mention a few) each having different characteristics and requirements on the use of computational resources, such as time, space, and energy consumption. Second, the hardware is evolving [HP19], sometimes requiring new data structuring techniques to fully exploit it, e.g. larger CPU registers, new instructions, parallelism, next-generation memories such as PMem. Third, data may present different kinds of regularities, which require different techniques that exploit them to improve the space-time performance.

It is largely this last reason to motivate this chapter. That is, given the fruitful exploitation of the *approximate linearities* in the keys-position Cartesian plane for the predecessor search problem (Chapter 4), can we follow a similar approach to build novel efficient rank/select dictionaries?

In the rest of this chapter, which is based on [BFV21a; BFV21b], we first review some known rank/select dictionaries (Section 5.1). After, we introduce our novel lossless compression scheme based on piecewise linear $\varepsilon$-approximations (Section 5.2) and add proper algorithms and data structures to support fast rank and select operations on it (Section 5.3). We then design and analyse an algorithm that minimises the space occupancy of our compression scheme (Section 5.5). We also consider the combination of our approach with hybrid solutions that partition the datasets into chunks and apply the best encoding to each chunk (Section 5.6). Finally, we demonstrate with a large set of experiments that our approach provides new interesting space-time trade-offs with respect to several well-established rank/select structures implementations (Section 5.7).

Throughout this chapter, we assume the standard word RAM model of computation with word size $w = \Theta(\log u)$ and $w = \Omega(\log n)$.

## 5.1 A review of known rank/select dictionaries

Existing rank/select dictionaries differ by the way they encode $A$ and how they use redundancy to squeeze the space and still support fast operations.

In the most basic case, $A$ is represented via its characteristic bitvector $B$, namely a bitvector of length $u$ such that $B[i] = 1$ if $i \in A$, and $B[i] = 0$ otherwise, for $0 \leq i < u$. Then, $\mathsf{rank}(x)$ is the number of 1s in $B[0, x]$, and $\mathsf{select}(i)$ is the position of the $i$th 1 in $B$. One can also be interested in $\mathsf{rank}_0$ and $\mathsf{select}_0$, which look instead for the 0s in the bitvector, but it holds $\mathsf{rank}_0(i) = i - \mathsf{rank}(i)$, while $\mathsf{select}_0$ can be reduced to select via other known reductions [Ram16].

It is long known that $u + o(u)$ bits are sufficient to have constant-time rank and select [Cla96; Mun96]. Provided that we keep $B$ in plain form and look for constant-time operations, the best that we can aim for the redundancy term $o(u)$ is $\Theta(u \log \log u / \log u)$ bits [Gol07]. Later, optimal trade-offs were also given in terms of the density of 1s in $B$ [Gol+14] or for the cell-probe model [PV10; Yu19].

Practical implementations of rank/select on plain bitvectors have been extensively studied and evaluated experimentally [GP14; Gon+05; NP12; OS07; Vig08].

If $A$ is sparse, i.e. $B$ contains a few 0s or 1s, then it may be convenient to switch to compressed representations. The information-theoretic minimum space to store $A$ is $\mathcal{B} = \lceil \log \binom{u}{n} \rceil$, which may be much smaller than $u$. The value $\mathcal{B}$ is related to the (empirical) zero-order entropy of $B$, $H_0(B)$, defined as $uH_0(B) =$

$n \log \frac{u}{n} + (u-n) \log \frac{u}{u-n}$. In fact, $\mathcal{B} = uH_0(B) - \mathcal{O}(\log u)$. Here, the best upper bound on the redundancy was attained by [Păt08], whose solution takes $\mathcal{B} + u/(\frac{\log u}{t})^t + \mathcal{O}(u^{3/4} \log u)$ bits and supports both rank and select in $\mathcal{O}(t)$ time, that is, constant-time operations in $\mathcal{B} + \mathcal{O}(u/\operatorname{poly} \log u)$ bits. This essentially matches the lower bounds provided in [PV10]. A widely known solution for a sparse $A$ is the RRR encoding [RRS07], which supports constant-time rank and select in $\mathcal{B} + \mathcal{O}(u \log \log u / \log u)$ bits of space. There are also representations bounded by the $k$th order entropy of $B$, defined as $uH_k(B) = \sum_{x \in \{0,1\}^k} |B_x| H_0(B_x)$ where $B_x$ is the bitvector concatenating the bits immediately following an occurrence of $x$ in $B$. For example, the solution of [SG06] achieves constant-time operations in $uH_k(B) + \mathcal{O}(u(\log \log u + k + 1)/\log u)$ bits.

To further reduce the space, one has to give up the constant time for both operations. An example is given by the Elias-Fano representation [Eli74; Fan71], which supports select in $\mathcal{O}(1)$ time and rank in $\mathcal{O}(\log \frac{u}{n})$ time while taking $n \log \frac{u}{n} + 2n + o(n)$ bits of space [Nav16, §4.4]. Its implementations and refinements proved to be very effective in a variety of contexts [OS07; OV14; PV17; Vig08; Vig13].

Another compressed representation for $A$ is based on gap encoding. In this case, instead of $\mathcal{B}$ or the zero-order entropy, it is common to use more data-aware measures [AR19; Fos+06; Gup+07; MN07; SG06]. Consider the gaps $g_i$ between consecutive integers in $A$ taken in sorted order, i.e. $g_i = \mathsf{select}(i) - \mathsf{select}(i-1)$, and suppose we could store each $g_i$ in $\lceil \log(g_i + 1) \rceil$ bits. Then the gap measure is defined as $gap(A) = \sum_i \lceil \log(g_i + 1) \rceil$. An example of a data-aware structure whose space occupancy is bounded in terms of $gap$ is presented in [Gup+07], which takes $gap(A)(1 + o(1))$ bits while supporting select in $\mathcal{O}(\log \log n)$ time and rank in time matching the optimal predecessor search bounds [PT06; NR20]. Another example is given in [MN07] taking $gap(A) + \mathcal{O}(n) + o(u)$ bits and supporting constant-time operations. Important ingredients of these $gap$-based data-aware structures are self-delimiting codes such as Elias $\gamma$- and $\delta$-codes [WMB99].

Recent work [AR19] explored further interesting data-aware measures for bounding the space occupancy of rank/select dictionaries that take into account *runs* of consecutive integers in $A$. They introduced data structures supporting constant-time rank and select in a space bounded by these new data-aware measures. This proposal is mainly theoretical, and indeed the authors evaluated only its space occupancy. A more practical approach, described in [Arr+18; AW20], combines gap and run-length encoding by fitting as many gaps $g_i$ as possible within a single 32-bit word. This is done via a 4-bit header indicating how the remaining 28 bits must be decoded (e.g. one gap of 28 bits, two gaps of 14 bits each, etc.).

## 5.2 Compressing via linear approximations

Let us assume that $A = \{x_1, x_2, \ldots, x_n\}$ is a sorted sequence of $n$ distinct integers. Our idea is to first map $A$ to a Cartesian plane, similarly to what we have done in Section 2.1 but with the key-position axes swapped. Specifically, we map each element $x_i \in A$ to a point $(i, x_i)$ in the Cartesian plane, for $i = 1, 2, \ldots, n$.

It is easy to see that any function $f$ that passes through all the points in this plane can be thought of as an encoding of $A$ because we can recover $x_i$ by querying $f(i)$. Clearly, $f$ should be fast to be computed and occupy little space.

Here, we aim at implementing $f$ via a sequence of segments. Segments capture certain data patterns naturally. Any run of consecutive and increasing integers, for example, can be encoded by one segment with slope 1. Generalising, any run of increasing integers with a constant gap $g$ can be encoded by one segment with slope $g$. Slight deviations from these data patterns can still be captured if we allow a segment to make some "errors" in approximating $x_i$ at position $i$, provided that we fix these errors by storing some additional information.

We now adapt the Definition 2.1 of piecewise linear $\varepsilon$-approximation to the Cartesian plane with swapped axes we are considering here.[1]

**Definition 5.1.** *A piecewise linear $\varepsilon$-approximation for the integer sequence $A = \{x_1, x_2, \ldots, x_n\}$ is a partition of $A$ into subsequences such that each subsequence $x_i, x_{i+1}, \ldots, x_j$ of the partition is covered by a segment (i.e. a linear model) $f$ such that $|f(k) - x_k| \leq \varepsilon$ for each $k \in [i, j]$.*

Again, we use O'Rourke's algorithm (Lemma 2.1) to compute a piecewise linear $\varepsilon$-approximation for $A$ with the least amount of segments in $\mathcal{O}(n)$ time.

We represent the $j$th segment as the triple $s_j = (r_j, \alpha_j, \beta_j)$, where $\alpha_j$ is the slope, $\beta_j$ is the intercept, and $r_j$ is the abscissa of the point that started the segment. If $\ell$ is the number of segments forming the piecewise linear $\varepsilon$-approximation, we set $r_{\ell+1} = n$ and observe that $r_1 = 1$. The $x$-values $r_j$s partition the universe $[n]$ in $\ell$ ranges so that, for any integer $i$ between $r_j$ and $r_{j+1}$ (non-inclusive), we use the segment $s_j$ to approximate the value $x_i$ as follows:

$$f_j(i) = (i - r_j) \cdot \alpha_j + \beta_j.$$

[1] Although we reuse the name piecewise linear $\varepsilon$-approximation, the reader must not confuse the one here with the one used previously in Chapters 2 to 4. Here, the $\varepsilon$-guarantee is on the value of the keys and not on their position in the sorted order. Moreover, here we are not interested in the keys belonging to $\mathcal{U} \setminus A$.

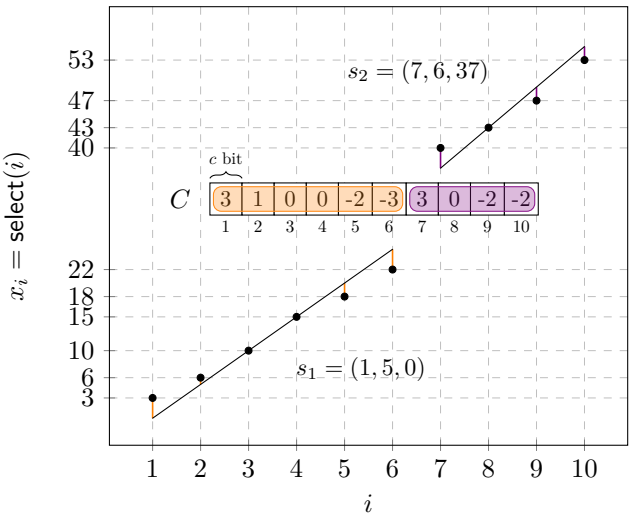



**Figure 5.1:** The LA-vector encoding of $A = \{3, 6, 10, 15, 18, 22, 40, 43, 47, 53\}$ for $c = 3$ is given by the two segments $s_1, s_2$ and the array $C$. A segment $s_j = (r_j, \alpha_j, \beta_j)$ approximates the value of an item with rank $i$ via $f_j(i) = (i - r_j) \cdot \alpha_j + \beta_j$, and $C$ corrects the approximation. For example, $x_5 = \lfloor f_1(5) \rfloor + C[5] = 20 - 2 = 18$ and $x_8 = \lfloor f_2(8) \rfloor + C[8] = 43 + 0 = 43$.

But $f_j(i)$ is an inexact approximation of $x_i$ bounded by $\varepsilon$. Then, in order to turn it into a lossless representation, we complement the values returned by $f_j$ with an array $C[1, n]$ of integers whose modulo is bounded by $\varepsilon$. Precisely, each $C[i]$ represents the small "correction value" $x_i - \lfloor f_j(i) \rfloor$, which belongs to the set $\{-\varepsilon, -\varepsilon + 1, \ldots, -1, 0, 1, \ldots, \varepsilon\}$. If we allocate $c \geq 2$ bits for each correction in $C$, then the piecewise linear $\varepsilon$-approximation is allowed to err by at most $\varepsilon = 2^{c-1} - 1$. We also consider the case $c = 0$, for which we set $\varepsilon = 0$. We ignore the case $c = 1$, because one bit is not enough to distinguish corrections in $\{-1, 0, 1\}$.

The vector $C$ completes our encoding, which we name *linear approximation vector* (LA-vector) and illustrate in Figure 5.1. Recovering the original sequence $A$ is as simple as scanning the segments $s_j$ of the piecewise linear $\varepsilon$-approximation and writing the value $\lfloor f_j(i) \rfloor + C[i] = x_i$ to the output, for $j = 1, \ldots, \ell$ and for $i = r_j, \ldots, r_{j+1} - 1$. This process, formalised in Algorithm 3, is appealing in practice because the array $C$ contains tightly-packed integers that are accessed sequentially, and the computation of $f_j$ is fast because its values are loaded into three registers when $s_j$ is first accessed. Moreover, there are no data dependencies among the iterations (as it happens for example when integers are delta-coded and a prefix sum is needed).

Recovering a single integer $x_i$ requires first the identification of the segment $s_j$ that includes the position $i$, and then the evaluation of $\lfloor f_j(i) \rfloor + C[i]$. A binary search over

**Algorithm 3** Decompression.

**Input:** Piecewise linear $\varepsilon$-approximation $\{s_1, \ldots, s_\ell\}$, corrections $C[1, n]$
**Output:** Uncompressed set $A$

```
1: out ← an empty array of size n
2: for all segments s_j = (r_j, α_j, β_j) in the PLA do
3:   for i ← r_j to r_{j+1} − 1 do
4:     out[i] ← ⌊f_j(i)⌋ + C[i], where f_j = (i − r_j) · α_j + β_j
5: return out
```

the starting positions $r_j$ of the segments in the piecewise linear $\varepsilon$-approximation would be enough and takes $\mathcal{O}(\log \ell)$ time, but we will aim for something more sophisticated in terms of algorithmic design and engineering to squeeze the most from this novel approach, as commented in the following sections.

### 5.2.1 On compression effectiveness

Two counterpoising factors influence the effectiveness of the compressed space occupancy of the LA-vector.

1. How the integers in $A$ map on the Cartesian plane, and thus how many segments they require for a lossy $\varepsilon$-approximation. The larger is $\varepsilon$, the smaller is "expected" to be the number $\ell$ of these segments.

2. The value of the parameter $c \geq 0$, which determines the space occupancy of the array $C$, having size $nc$ bits. From above, we know that $\varepsilon = \max(0, 2^{c-1} - 1)$, so the smaller is $c$, the smaller is the space occupancy of $C$, but the larger is "expected" to be the number $\ell$ of segments of the piecewise linear $\varepsilon$-approximation built for $A$.

We say "expected" because $\ell$ depends on the distribution of the points $(i, x_i)$ on the Cartesian plane. In the best scenario, the points lie on one line, so $\ell = 1$ and we can set $c = 0$. The more these points follow a linear trend, the smaller $c$ can be chosen and, in turn, the smaller is the number $\ell$ of segments approximating these points with error $\varepsilon$. Although in the worst case it holds $\ell \leq \min\{u/(2\varepsilon), n/2\}$, because of a simple adaptation of Lemma 3.1, we will show in Section 5.4 that for sequences drawn from a distribution with finite mean and variance there are tighter bounds on $\ell$. This leads us to argue that the combination of the piecewise linear $\varepsilon$-approximation and the array $C$, on which the storage scheme of the LA-vector hinges upon, is an interesting algorithmic tool to design novel compressed rank/select dictionaries.

At this point, it is useful to formally define the interplay among $A$, $c$ and $\ell$. We argue that the number $\ell$ of segments of the optimal piecewise linear $\varepsilon$-approximation (namely the one using the smallest $\ell$) can be thought of as a new compressibility measure for the information present in $A$, possibly giving some insights (such as the degree of approximate linearity of the data) that the classical entropy measures do not explicitly capture. In the following, we assume $c \leq \log u$ to avoid the case in which $nc$ exceeds the $\mathcal{O}(n \log u)$ bits needed by an explicit representation of $A$.

**Definition 5.2.** *Let $A = \{x_1, x_2, \ldots, x_n\}$ be a sorted sequence of $n$ distinct integers drawn from the universe $[u]$. Given an integer parameter $c \in \{0, \ldots, \log u\}$, we define $\ell$ as the number of segments which constitute the optimal piecewise linear $\varepsilon$-approximation of maximum error $\varepsilon = \max(0, 2^{c-1} - 1)$ computed on the set of points $\{(i, x_i) \mid i = 1, \ldots, n\}$.*

We are ready to compute the space taken by the LA-vector. As far as the representation of a segment $s_j = (r_j, \alpha_j, \beta_j)$ is concerned, we note that: (i) the value $r_j$ is an abscissa in the Cartesian plane, thus it can be represented in $\log n$ bits;[2] (ii) the slope $\alpha_j$ can be encoded as a rational number with a numerator of $\log u$ bits and a denominator of $\log n$ bits [ORo81; Xie+14]; (iii) the intercept $\beta_j$ is an ordinate in the plane, thus it can be represented in $\log u$ bits. Therefore, the overall cost of the piecewise linear $\varepsilon$-approximation is $2\ell(\log n + \log u)$ bits. Summing the $nc$ bits taken by $C$ gives our first result.

**Theorem 5.1.** *Let $A$ be a set of $n$ integers drawn from the universe $[u]$. Given integers $c$ and $\ell$ as in Definition 5.2, a plain implementation of the LA-vector takes $nc + 2\ell(\log n + \log u)$ bits of space.*

We can further improve the space taken by the segments as follows. The $r_j$s form an increasing sequence of $\ell$ positive integers bounded by $n$. The $\beta_j$s form an increasing sequence of $\ell$ positive integers bounded by $u$.[3] Using the Elias-Fano representation (Lemma 4.1), we reduce the space of the two sequences to $\ell \log \frac{n}{\ell} + \ell \log \frac{u}{\ell} + 4\ell + o(\ell) = \ell(\log \frac{un}{\ell^2} + 4 + o(1))$ bits. Then, accessing $r_j$ or $\beta_j$ amounts to call the constant-time $\mathsf{select}(j)$ on the corresponding Elias-Fano compressed sequence. Summing the $nc$ bits taken by $C$ and the $\ell(\log n + \log u)$ bits taken by the $\alpha_j$s gives our second result.

[2] For ease of exposition, we assume that logarithms hide their ceiling and thus return integers.

[3] This is because $\beta_j$ is the ordinate where $s_j$ starts, i.e. $\beta_j = f_j(r_j)$ (see Figure 5.1 and the definition of $f_j$). In the text, we referred to $\beta_j$ as the "intercept", but this is improper because $\beta_j$ is not the ordinate of the intersection between $f_j$ and the $y$-axis.

**Theorem 5.2.** *Let $A$ be a set of $n$ integers drawn from the universe $[u]$. Given integers $c$ and $\ell$ as in Definition 5.2, there exists a more compressed version of the LA-vector that takes $nc + \ell(2\log\frac{un}{\ell} + 4 + o(1))$ bits of space.*

Finally, we mention that when multiple segments share the same or similar slope, it may be beneficial to compress the $\alpha_j$s with Theorem 4.5.

### 5.2.2 Entropy-coding the corrections

In this section, we show how to further reduce the space of the LA-vector by entropy-coding the vector of corrections $C$.

Regard $C$ as a string of length $n$ from an integer alphabet $\Sigma = \{-\varepsilon, -\varepsilon+1, \ldots, \varepsilon\}$, and let $n_x$ denote the number of occurrences of a symbol $x$ in $C$. The zero-order entropy of $C$ is defined as

$$H_0(C) = \sum_{x\in\Sigma} \frac{n_x}{n} \log \frac{n}{n_x}.$$

The value $nH_0(C)$ is the output size of an ideal compressor that uses $-\log\frac{n_x}{n}$ bits for coding the symbol $x$ unambiguously [KM99; CT06]. In order to further squeeze the output size, one could take advantage not only of the frequency of symbols but also of their preceding context in $C$. Let $C_y$ be the string of length $|C_y|$ that concatenates all the single symbols following each occurrence of a context $y$ inside $C$. The $k$th order entropy of $C$ is defined as

$$H_k(C) = \frac{1}{n} \sum_{y\in\Sigma^k} |C_y|\, H_0(C_y).$$

A well-known data structure achieving zero-order entropy compression is the wavelet tree [GGV03] with the bitvectors stored in its nodes compressed using RRR [RRS07]. Considering the LA-vector of Theorem 5.2 but compressing $C$ via this approach (see also [NM07, Theorem 8]), we obtain:

**Theorem 5.3.** *Let $A$ be a set of $n$ integers drawn from the universe $[u]$. Given integers $c$ and $\ell$ as in Definition 5.2, there exists a zero-order entropy-compressed version of the LA-vector for $A$ that takes $nH_0(C) + o(nc) + \ell(2\log\frac{un}{\ell} + 4 + o(1))$ bits of space, and $\mathcal{O}(c)$ time to access a position in $C$, where $C$ is the vector of corrections.*

A well-performing high-order entropy-compressed data structure over strings drawn from an integer alphabet is the alphabet-friendly FM-index [Fer+04; FV07]. Using the alphabet-friendly FM-index to store $C$, we obtain:

**Theorem 5.4.** *Let $A$ be a set of $n$ integers drawn from the universe $[u]$. Given integers $c$ and $\ell$ as in Definition 5.2, there exists a $k$th order entropy-compressed version of the LA-vector for $A$ that takes $nH_k(C) + o(nc) + \ell(2\log\frac{un}{\ell} + 4 + o(1))$ bits of space, and $\mathcal{O}(c(\log^{1+\tau} n)/\log\log n)$ time to access a position in $C$, where $C$ is the vector of corrections, and $\tau > 0$ is an arbitrary constant.*

To get a practical sense of the real compression achieved by the above two entropy-compressed versions of the LA-vector, we compare experimentally the space taken by the uncompressed corrections (as adopted in the plain LA-vector) with the space taken by (i) a Huffman-shaped wavelet tree with RRR-compressed bitvectors on $C$ (implementing the solution in Theorem 5.3), and (ii) a compressed suffix array based on a Huffman-shaped wavelet tree with RRR-compressed bitvectors on the Burrows-Wheeler Transform of $C$ (implementing the solution in Theorem 5.4). We denote the space taken by these two choices by `wt_huff`$(C)$ and `csa_wt`$(C)$, respectively, given the name of the corresponding classes in the `sdsl` library [GP14]. For `csa_wt`$(C)$, we do not take into account the space taken by the sampled suffix array because we do not need to support the locate query, which returns the list of positions in $C$ where a given pattern string occurs. Rather, to get individual corrections from $C$, we need the sampled inverse suffix array, which indeed we store and account for in the space occupancy of `csa_wt`$(C)$.

Figure 5.2 shows the results with a value $c = 7$ on four real-world datasets, described in detail in Section 5.7. For the DNA dataset, there is no significant difference between the plain corrections and the zero-order entropy-coder `wt_huff`$(C)$. Instead, the high-order entropy-coder `csa_wt`$(C)$ is 33% smaller. For the other three datasets (5GRAM, URL, and GOV2), both `wt_huff`$(C)$ and `csa_wt`$(C)$ are up to 56% and 72% smaller than the plain corrections, respectively. This shows that there is some statistical redundancy within the array of corrections $C$ that the LA-vector could deploy to squeeze its space occupancy further.

Another important issue to investigate concerns the impact on the compression of $C$ that is induced by changing the slopes of the segments in the optimal piecewise linear $\varepsilon$-approximation computed for LA-vector. Intuitively, as depicted in Figure 5.3, different slope-intercept pairs satisfying the same $\varepsilon$-bound generate different vectors $C$ with different entropies. As a consequence, instead of picking a random slope-intercept pair within the ones that are $\varepsilon$-approximation, one can choose the slope-intercept pair minimising the entropy of $C$.

For the experiment in Figure 5.2, we adopted the strategy that always chooses the maximum slope among the $\varepsilon$-approximate segments for $A$. In Figure 5.4, we compare this strategy, which we call *mmax*, with two other strategies: (i) *mmid*,

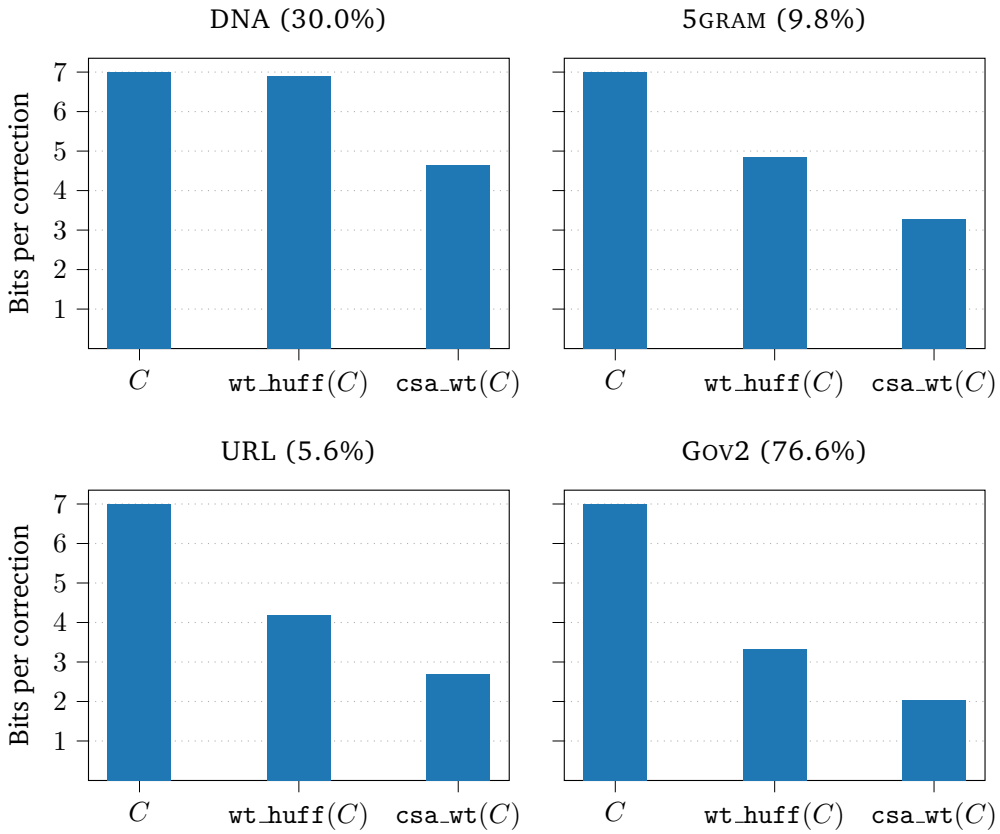



**Figure 5.2:** The space needed by the plain corrections $C$, zero-order entropy-coded corrections $\texttt{wt\_huff}(C)$, and the high-order entropy-coded corrections $\texttt{csa\_wt}(C)$ in an LA-vector with $c = 7$. Next to each dataset name, we show the density value $n/u$ as a percentage.

which chooses the average slope between the smallest and the largest feasible slopes, and (ii) *best*, a heuristic that selects nine slopes at regular intervals between the smallest and the largest feasible slopes and picks the one minimising $H_0(C)$. For the DNA and 5GRAM datasets, there is no noticeable improvement in changing the slope of the segments of the LA-vector. Instead, for URL and GOV2, changing the slope of each segment from *mmax* to *mmid* or *best* reduces $H_0(C)$. Of course, since the choice *best* targets only the zero-order entropy of the corrections, the plots show little or no reduction of $H_k(C)$.

To sum up, we can further reduce the space occupancy of the LA-vector by entropy-coding its correction vector $C$. This reduction is particularly interesting in applications in which the LA-vector is used as an archival method, that is, when efficient random access and queries are not required. In the following, we concentrate on how to support efficient select and rank queries over $A$, so we explore the variant of the LA-vector in which $C$ is kept uncompressed.

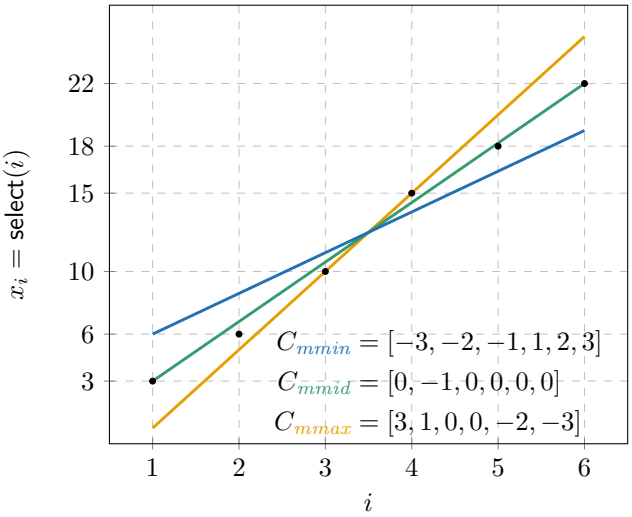



**Figure 5.3:** Three possible slopes, *mmin*, *mmid* and *mmax*, for a segment encoding the set $A = \{3, 6, 10, 15, 18, 22\}$ with $c = 3$. Each slope generates a different vector $C$ with a different entropy: $H_0(C_{mmin}) = 2.58$, $H_0(C_{mmid}) = 0.65$, and $H_0(C_{mmax}) = 2.25$.

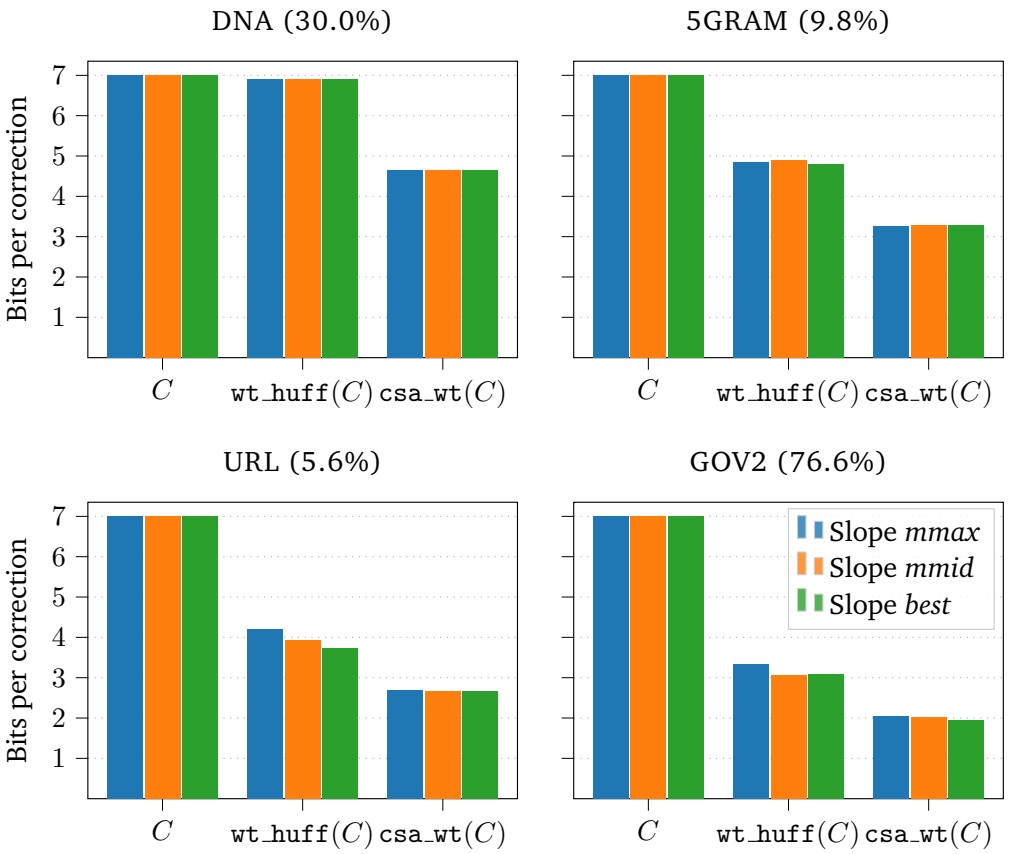



**Figure 5.4:** A different choice of the slope of the segments in an LA-vector may yield a reduced space occupancy of the entropy-coded correction vector $C$. Here we show three choices: *mmax*, *mmid* and *best* (see Section 5.2.2).

## 5.3 Supporting rank and select

To answer select$(i)$ on the LA-vector (either on the plain implementation of Theorem 5.1 or the compressed implementation of Theorem 5.2), we build a *predecessor* structure $\mathcal{D}$ on the set $R = \{r_j \mid 1 \leq j \leq \ell\}$ and proceed in three steps. First, we use $\mathcal{D}$ to retrieve the segment $s_j$ in which $i$ falls into via $j = \mathsf{pred}(i)$. Second, we compute $f_j(i)$, i.e. the approximate value of $x_i$ given by the segment $s_j$. Third, we read $C[i]$ and return the value $\lfloor f_j(i) \rfloor + C[i]$ as the answer to select$(i)$. The last two steps take $\mathcal{O}(1)$ time. Treating $\mathcal{D}$ as a black box yields the following result.

**Lemma 5.1.** *The LA-vector supports select queries in $t + \mathcal{O}(1)$ time and $b$ bits of additional space, where $t$ is the query time and $b$ is the occupied space of a predecessor structure $\mathcal{D}$ constructed on a set of $\ell$ integers over the universe $[n]$.*

If $\mathcal{D}$ is represented as the characteristic bitvector of the set $R$ augmented with a data structure supporting constant-time predecessor queries (or rank queries, as termed in the case of bitvectors [Nav16]), then we achieve constant-time select by using only $n + o(n)$ additional bits, i.e. about one bit per integer of $A$ more than what Theorem 5.1 requires. Note that this bitvector encodes $R$, so that the $\ell \log n$ bits required in Theorem 5.1 for the representation of the $r_j$s can be dropped.

**Corollary 5.1.** *Let $A$ be a set of $n$ integers drawn from the universe $[u]$. Given integers $c$ and $\ell$ as in Definition 5.2, there exists a compressed representation of the LA-vector for $A$ that takes $n(c+1+o(1)) + \ell(2\log u + \log n)$ bits of space while supporting select in $\mathcal{O}(1)$ time.*

Let us compare the space occupancy achieved by the compressed LA-vector of Corollary 5.1 to the one of Elias-Fano, namely $n(\log \frac{u}{n}+2)+o(n)$ bits (see Lemma 4.1), as both solutions support constant-time select. The inequality turns out to be

$$\ell \leq \frac{n\,(\log \frac{1}{d} + o(1))}{2\log \frac{n}{d} + \log n} = \mathcal{O}\left(\frac{n}{\log n}\right),$$

where $d = n/u$ denotes the density of 1s in $B$.

To solve rank$(x)$, it would be sufficient to perform a binary search on $[n]$ for the largest $i$ such that select$(i) \leq x$. This naïve implementation takes $\mathcal{O}(t \log n)$ time, because of the implementation of select in $\mathcal{O}(t)$ time by Theorem 5.1.

We can easily improve this solution to $\mathcal{O}(\log \ell + \log n)$ time as follows. First, we binary search on the set of $\ell$ segments to find the segment $s_j$ that contains $x$ or its predecessor. Formally, we binary search on the interval $[1, \ell]$ to find the largest $j$ such that select$(r_j) = \lfloor f_j(r_j) \rfloor + C[r_j] \leq x$. Second, we binary search on the $r_{j+1} - r_j \leq n$

**Algorithm 4** Rank implementation by Lemma 5.2.

**Input:** $x$, piecewise linear $\varepsilon$-approximation $\{s_1, s_2, \ldots, s_\ell\}$, corrections $C[1, n]$
**Output:** Returns $\mathsf{rank}(x)$

1: Find max $j \in [1, \ell]$ such that $\lfloor f_j(r_j) \rfloor + C[r_j] \leq x$ by binary search
2: $pos \leftarrow \lfloor (x - \beta_j)/\alpha_j \rfloor + r_j$
3: $err \leftarrow \lceil \varepsilon/\alpha_j \rceil$, where $\varepsilon = \max(0, 2^{c-1} - 1)$
4: $lo \leftarrow \max\{pos - err, r_j\}$
5: $hi \leftarrow \min\{pos + err, r_{j+1}\}$
6: Find max $i \in [lo, hi]$ such that $\lfloor f_j(i) \rfloor + C[i] \leq x$ by binary search
7: **return** $i$

integers compressed by segment $s_j$ to find the largest $i$ such that $\lfloor f_j(i) \rfloor + C[i] \leq x$. Finally, we return $i$ as the answer to $\mathsf{rank}(x)$.

Surprisingly, we can further speed up rank queries without adding any redundancy on top of the encoding of Theorem 5.1. The key idea is to narrow down the second binary search to a subset of the elements covered by $s_j$ (i.e. a subset of the ones in positions $[r_j, r_{j+1} - 1]$), which is determined by exploiting the fact that $s_j$ approximates all these elements by up to an additive term $\varepsilon$. Technically, we know that $|f_j(i) - x_i| \leq \varepsilon$, and we aim to find $i$ such that $x_i \leq x < x_{i+1}$. Hence we can narrow the range to those $i \in [r_j, r_{j+1} - 1]$ such that $f_j(i) - \varepsilon \leq x < f_j(i + 1) + \varepsilon$. By expanding $f_j(i) = (i - r_j) \cdot \alpha_j + \beta_j$, noting that $f$ is linear and increasing, we get all candidate $i$ as the ones satisfying

$$(i - r_j) \cdot \alpha_j + \beta_j - \varepsilon \leq x < (i + 1 - r_j) \cdot \alpha_j + \beta_j + \varepsilon.$$

By solving for $i$, we get

$$\frac{x - \beta_j}{\alpha_j} + r_j - \left( \frac{\varepsilon}{\alpha_j} + 1 \right) < i \leq \frac{x - \beta_j}{\alpha_j} + r_j + \frac{\varepsilon}{\alpha_j}.$$

Since $i$ is an integer, we can round the left and the right side of the last inequality, and then we set $pos = \lfloor (x - \beta_j)/\alpha_j \rfloor + r_j$ and $err = \lceil \varepsilon/\alpha_j \rceil$, so that the searched position $i$ falls in $[pos - err, pos + err]$.

The pseudocode of Algorithm 4 exploits these ideas to perform a binary search on the first integers compressed by the segments (Line 1), to compute the approximate rank and the corresponding approximation error (Lines 2–3), and finally to binary search on the restricted range specified above (Lines 4–6). As a final note, we observe that $\alpha_j \geq 1$ for every $j$, because $x_i \in A$ are increasing, and thus the segments have a slope of at least 1. Consequently, $\varepsilon/\alpha_j \leq \varepsilon$ and the range on which we perform

the second binary search has size $2\varepsilon < 2^c$, thus this second binary search takes $\mathcal{O}(\log \frac{\varepsilon}{\alpha_j}) = \mathcal{O}(c)$ time.

**Lemma 5.2.** *The LA-vector supports rank queries in $\mathcal{O}(\log \ell + c)$ time and in no additional space.*

Note that Lemma 5.2 applies to: (i) the plain LA-vector representation provided in Theorem 5.1 (ii) the compressed LA-vector representation provided in Theorem 5.2 (the one that compresses $\beta_j$s and $r_j$s), (iii) the representation provided in Lemma 5.1 (the one supporting select in parametric time $t$), and (iv) the representation provided in Corollary 5.1 (the one supporting select in constant time).

We can improve the bound of Lemma 5.2 by replacing the binary search at Line 1 of Algorithm 4 with the following predecessor data structure.

**Lemma 5.3** ([PT06])**.** *Given a set $Q$ of $q$ integers over a universe of size $u$, let us define $a = \log \frac{s \log u}{q}$, where $s \log u$ is the space usage in bits chosen at building time. Then, the optimal predecessor search time is*

$$\begin{aligned} PT(u, q, a) = \Theta(\min\{ & \log q / \log\log u, \\ & \log \tfrac{\log(u/q)}{a}, \\ & \log \tfrac{\log u}{a} / \log(\tfrac{a}{\log q} \cdot \log \tfrac{\log u}{a}), \\ & \log \tfrac{\log u}{a} / \log(\log \tfrac{\log u}{a} / \log \tfrac{\log q}{a})\}). \end{aligned}$$

Let $T = \{\mathsf{select}(r_j) \mid 1 \leq j \leq \ell\}$ be the subset of $A$ containing the first integer covered by each segment. We sample one element of $T$ out of $\Theta(2^c)$ and insert the samples into the predecessor data structure of Lemma 5.3 so that $s = q = \ell/2^c$ and thus $a = \log\log u$. Then, we replace Line 1 of Algorithm 4 with a predecessor search followed by an $\mathcal{O}(c)$-time binary search in-between two samples.

**Corollary 5.2.** *The LA-vector supports rank queries in $PT(u, \ell/2^c, \log\log u) + \mathcal{O}(c)$ time and $\mathcal{O}((\ell/2^c)\log u)$ bits of additional space.*

We can restrict our attention to the first two branches of the min-formula describing the PT term in Lemma 5.3, as the latter two are instead relevant for universe sizes that are super-polynomial in $q$, i.e. $\log u = \omega(\log q)$. The time complexity of rank in Corollary 5.2 then becomes $\mathcal{O}(\min\{\log_w \frac{\ell}{2^c}, \log\log \frac{u}{\ell}\} + c)$, where $w = \Omega(\log u)$ is the word size of the machine.

## 5.4 Special sequences

We now combine the results of Chapter 3 with the ones achieved by the LA-vector. Specifically, by plugging the bound on the number of segments $\ell$ of Theorem 3.4 into the constant-time select of Corollary 5.1 and the rank implementation of Corollary 5.2, we obtain the following result.

**Theorem 5.5.** *Under the assumptions of Theorem 3.4, there exists a compressed version of the LA-vector for $A$ that supports rank in PT$(u, \frac{n\sigma^2}{\mu^2 2^{3c-2}}, \log\log u) + \mathcal{O}(c)$ time and select in $\mathcal{O}(1)$ time within $n[c + 1 + o(1) + ((2 + 1/2^c)\log u + \log n)\frac{\sigma^2}{\mu^2 2^{2c-2}}]$ bits of space with high probability.*

We stress the fact that the data structure of Theorem 5.5 is deterministic. In fact, the randomness is over the gaps between consecutive integers of the input data, and the result holds for any probability distribution as long as the mean and variance are finite. Moreover, according to the experiments in Section 3.4, the hypotheses of Theorem 5.5 are very realistic in several applicative scenarios.

Having said that, we observe that the hypothesis "$\varepsilon = 2^{c-1} - 1$ is sufficiently larger than $\sigma/\mu$" implies that the ratio $\sigma/(\mu 2^{c-1})$ is much smaller than 1. Hence, it is reasonable to assume that the space bound in Theorem 5.5 is dominated by the term $n(c + 1)$ which is independent of the universe size while still ensuring constant time select and fast rank operations. If we compare the factor $c + 1$ present in the space bound of the LA-vector with the factor $\log \frac{u}{n}$ present in the space bound of Elias-Fano, we notice that the latter gets larger as the data is sparse ($n \ll u$). On the other hand, the time complexity of select is constant in both cases, whereas our rank is faster whenever $\log \frac{n\sigma}{\mu}$ is asymptotically smaller than $\log \frac{u}{n}$, which is indeed for $u = \omega(n^2\sigma/\mu)$.

In general terms, some results of the previous sections, such as Corollary 5.1 and Lemma 5.2, showed that our LA-vector is better than Elias-Fano whenever $\ell = \mathcal{O}(n/\log n)$. Since Theorem 3.4 proves that $\ell = \Theta(n/\varepsilon^2)$ for a large class of input sequences, we can derive that for such sequences our solution is better than Elias-Fano if $\varepsilon = \omega(\sqrt{\log n})$.

## 5.5 On optimal data partitioning to improve space

So far, we assumed a fixed number of bits $c \geq 0$ for each of the $n$ corrections in the LA-vector, which is equivalent to saying that the $\ell$ segments in the piecewise linear $\varepsilon$-approximation guarantee the same error $\varepsilon = \max(0, 2^{c-1} - 1)$ over all the integers

in the input set $A$. However, the input data may exhibit a variety of regularities that allow to compress it further if we use a different $c$ for different partitions of $A$. The idea of partitioning data to improve its compression has been studied in the past [BFG03; FNV11; OV14; SV10; WMB99], and it will be further developed in this section with regard to our piecewise linear approximations.

We reduce the problem of minimising the space of our rank/select dictionary to a single-source shortest path problem over a properly defined weighted Directed Acyclic Graph (DAG) $\mathcal{G}$ defined as follows. The graph has $n$ vertices, one for each element in $A$, plus one sink vertex denoting the end of the sequence. An edge $(i, j)$ of weight $w(i, j, c)$ indicates that there exists a segment compressing the integers $x_i, x_{i+1}, \ldots, x_{j-1}$ of $A$ by using $w(i, j, c) = (j - i)\, c + \kappa$ bits of space, where $c$ is the bit-size of the corrections, and $\kappa$ is the space taken by the segment representation in bits (e.g. using the plain encoding of Theorem 5.1 or the compressed encoding of Theorem 5.2). We consider all the possible values of $c$ except $c = 1$, because one bit is not enough to distinguish corrections in $\{-1, 0, 1\}$. Namely, we consider $c \in \{0, 2, 3, \ldots, c_{max}\}$, where $c_{max} = \mathcal{O}(\log u)$ is defined as the correction value that produces one single segment on $A$. Since each vertex is the source of at most $c_{max}$ edges, one for each possible value of $c$, the total number of edges in $\mathcal{G}$ is $\mathcal{O}(n\, c_{max}) = \mathcal{O}(n \log u)$. It is not difficult to prove the following:

**Fact 5.1.** *The shortest path from vertex 1 to vertex $n + 1$ in the weighted DAG $\mathcal{G}$ defined above corresponds to the piecewise linear $\varepsilon$-approximation for $A$ whose cost is the minimum among the PLAs that use a different error $\varepsilon$ on different segments.*

Fact 5.1 provides a solution to the rank/select dictionary problem which minimises the space occupancy of the approaches stated in Theorems 5.1 and 5.2.

Since $\mathcal{G}$ is a DAG, the shortest path can be computed in $\mathcal{O}(n \log u)$ time by taking the vertices in topological order and by relaxing their outgoing edges [Cor+09, §24.2]. However, one cannot approach the construction of $\mathcal{G}$ in a brute-force manner because this would take $\mathcal{O}(n^2 \log u)$ time and $\mathcal{O}(n \log u)$ space, as each of the $\mathcal{O}(n \log u)$ edges requires computing a segment in $\mathcal{O}(j - i) = \mathcal{O}(n)$ time with the algorithm of O'Rourke (Lemma 2.1).

To avoid this prohibitive cost, we propose an algorithm that computes a solution on the fly by working on a properly defined graph $\mathcal{G}'$ derived from $\mathcal{G}$, taking $\mathcal{O}(n \log u)$ time and $\mathcal{O}(n)$ space. This reduction in both time and space complexity is crucial to make the approach feasible in practice. Moreover, we will see that the obtained solution is not "too far" from the one given by the shortest path in $\mathcal{G}$.

Consider an edge $(i, j)$ of weight $w(i, j, c)$ in $\mathcal{G}$, which corresponds to a segment compressing the integers $x_i, x_{i+1}, \ldots, x_{j-1}$ of $A$ by using $w(i, j, c)$ bits of space. Clearly, the same segment compresses any subsequence $x_a, x_{a+1} \ldots, x_{b-1}$ of $x_i, \ldots, x_{j-1}$ still using $c$ bits per correction. Therefore, the edge $(i, j)$ "induces" sub-edges of the kind $(a, b)$, where $i \leq a < b \leq j$, of weight $w(a, b, c)$. We observe that the edge $(a, b)$ may not be an edge of $\mathcal{G}$ because a segment computed from position $a$ with correction size $c$ could end past $b$, thus including more integers on its right. Nonetheless this property is crucial to define our graph $\mathcal{G}'$.

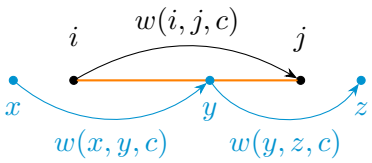
The vertices of $\mathcal{G}'$ are the same as the ones of $\mathcal{G}$. For the edges of $\mathcal{G}'$, we start from the subset of edges of $\mathcal{G}$ that correspond to the segments in the PLAs built for the input set $A$ for all the values of $c = 0, 2, 3, \ldots, c_{max}$. We call these, the *full edges* of $\mathcal{G}'$. Then, for each full edge $(i, j)$, we generate the *prefix edge* $(i, i + k)$ and the *suffix edge* $(i + k, j)$, for all $k = 1, \ldots, j - i$. This means that we are "covering" every full edge with all of its possible "splits" in two shorter edges having the same correction $c$ as $(i, j)$. The total size of $\mathcal{G}'$ is still $\mathcal{O}(n \log u)$.

We are now ready to show that the graph $\mathcal{G}'$ has a path whose weight is just an *additive* term far from the weight of the shortest path in $\mathcal{G}$. (Notice that this contrasts with the approaches that obtain a multiplicative approximation factor [FNV11; OV14].)

**Lemma 5.4.** *There exists a path in $\mathcal{G}'$ from vertex $1$ to vertex $n + 1$ whose weight is at most $\kappa\ell$ bits larger (in an additive sense) than the weight of the shortest path in $\mathcal{G}$, where $\kappa$ is the space taken by a segment in bits, and $\ell$ is the number of edges in the shortest path of $\mathcal{G}$.*

*Proof.* We show that a generic edge $(i, j)$ of weight $w(i, j, c) = (j - i)c + \kappa$ in $\mathcal{G}$ can be decomposed into at most two edges of $\mathcal{G}'$ whose total weight is at most $w(i, j, c) + \kappa$. The statement will then follow by recalling that $\ell$ is the number of edges in the shortest path of $\mathcal{G}$.

Consider the piecewise linear $\varepsilon$-approximation for $A$ with the same correction size $c$ as $(i, j)$. This piecewise linear $\varepsilon$-approximation has surely one segment that either starts from $i$ or overlaps $i$. In the former case we are done because the segment corresponds to the edge $(i, j)$, which appears in $\mathcal{G}'$ as a full edge. In the latter case, the segment corresponds to a full edge $(x, y)$ such that $x < i < y < j$, and it is followed by a segment that corresponds to a full edge $(y, z)$ such that $z > j$, as shown in the following picture. In fact, $y$ cannot occur after $j$ otherwise the segment corresponding to the edge $(i, j)$ would be longer, because the length of a segment in a piecewise linear $\varepsilon$-approximation is maximised.




Given this situation, we decompose the edge $(i, j)$ of $\mathcal{G}$ into: the suffix edge $(i, y)$ of $(x, y)$, and the prefix edge $(y, j)$ of $(y, z)$. Both edges $(i, y)$ and $(y, j)$ belong to $\mathcal{G}'$ by construction, they have correction size $c$, and their total weight is $w(i, y, c) + w(y, j, c) = (y - i)c + \kappa + (j - y)c + \kappa = (j - i)c + 2\kappa$. Since $w(i, j, c) = (j - i)c + \kappa$, the previous total weight can be rewritten as $w(i, j, c) + \kappa$, as claimed. □

We now describe an algorithm that computes the shortest path in $\mathcal{G}'$ without generating the full graph $\mathcal{G}$ but expanding $\mathcal{G}'$ incrementally so to use $\mathcal{O}(n)$ working space. The algorithm processes the vertices of $\mathcal{G}$ from left to right, while maintaining the following invariant: for $i = 1, \ldots, n + 1$, each processed vertex $i$ is covered by one segment for each correction size $c$, and all these segments form the frontier set $J$.

We begin from vertex $i = 1$ and compute the $c_{max}$ segments that start from $i$ and have any possible correction size $c = 0, 2, 3, \ldots, c_{max}$. We set $J$ as the set of these segments. As in the classic step of the shortest path algorithm for DAGs, we do a relaxation step on all the (full) edges $(i, j)$, where $j$ is the set of ending positions of the segments in $J$, that is, we test whether the shortest path to $j$ found so far can be improved by going through $i$ (initially, the shortest-path estimates are set to $\infty$ for each vertex) and update such shortest path accordingly [Cor+09, §24.2]. This completes the first iteration.

At a generic iteration $i$, we first check whether there is a segment in $J$ that ends at $i$. If so, we replace that segment with the longest segment starting at $i$ and using the same correction size, computed as usual using the algorithm of O'Rourke. Afterwards, for each full edge $(a, b)$ that corresponds to a segment in $J$, we first relax the set of prefix edges of the kind $(a, i)$, then we relax the set of suffix edges of the kind $(i, b)$. This is depicted in Figure 5.5.

**Theorem 5.6.** *There exists an algorithm that in $\mathcal{O}(n \log u)$ time and $\mathcal{O}(n)$ space outputs a path from vertex $1$ to vertex $n + 1$ whose weight is at most $\kappa\ell$ bits larger (in an additive sense) than the shortest path of $\mathcal{G}$, where $\kappa$ is the space taken by a segment in bits, and $\ell$ is the number of edges in the shortest path of $\mathcal{G}$.*

*Proof.* It is easy to see that the algorithm finds the shortest path in $\mathcal{G}'$. Indeed, it computes and relaxes: (i) the full edges of $\mathcal{G}'$ corresponding to the segments in a piecewise linear $\varepsilon$-approximation with correction size $c$ when updating the frontier

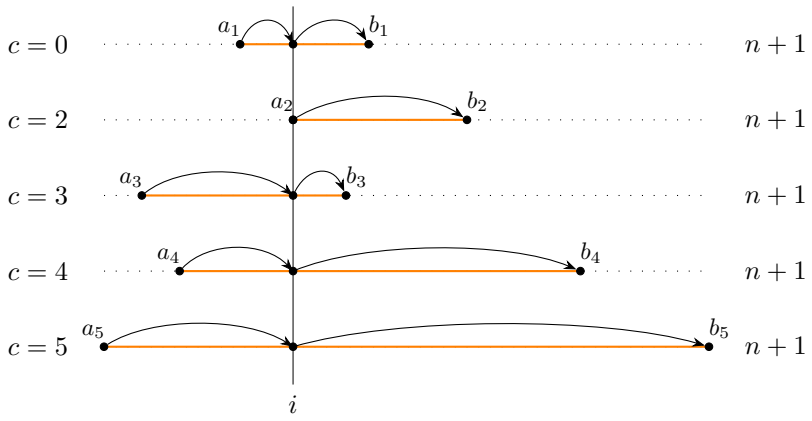



**Figure 5.5:** The algorithm of Theorem 5.6 keeps a frontier with the segments (in orange) crossing the processed vertex $i$ for each value of the correction size $c$ (here $c_{max} = 5$). For each segment with endpoints $a_c$ and $b_c$, which corresponds to a full edge $(a_c, b_c)$ with correction size $c$, the algorithm relaxes the prefix edge $(a_c, i)$ and the suffix edge $(i, b_c)$.

set $J$; and (ii) all prefix (resp. suffix) edges ending (resp. beginning) at a vertex $i$ when this vertex is processed. Therefore, the algorithm relaxes all the edges of $\mathcal{G}'$ and, according to Lemma 5.4, it finds a path whose weight is the claimed one.

As far as the space occupancy is concerned, the algorithm uses $\mathcal{O}(n + |J|) = \mathcal{O}(n + \log u) = \mathcal{O}(n)$ space at each iteration, since the size of the frontier set is $|J| = c_{max} = \mathcal{O}(\log u)$. The running time is $\mathcal{O}(|J|) = \mathcal{O}(\log u)$ per iteration, plus the cost of replacing a segment in $J$ when it ends before the processed vertex, i.e. the cost of computing a full edge. This latter cost is $\mathcal{O}(n)$ time for any given value of $c$ and over all $n$ elements (namely, it is $\mathcal{O}(1)$ amortised time per processed element [ORo81]), thus $\mathcal{O}(n \log u)$ time over all the values of $c$. □

Given Theorem 5.6, the piecewise linear $\varepsilon$-approximation computed by our algorithm can be used to design a rank/select dictionary which minimises the space occupancy of the solutions based on the approaches of Theorems 5.1 and 5.2. Section 5.7 will experiment with this approach.

## 5.6 On hybrid rank/select dictionaries

As recalled in Section 5.1, the literature offers a plethora of compressed rank/select dictionaries. Some take into account the statistical or the combinatorial properties of the input, others exploit the compressibility of clusters of consecutive integers. The compression scheme introduced in this chapter, on the other hand, exploits the "geometric properties" of the input data by accommodating their slight deviations

from linear trends with the use of small correction values. The choice of the best compression scheme in terms of space occupancy heavily depends on the characteristics of the input data, and thus it is reasonable to expect the best gains in space if we design *hybrid* solutions that combine several different approaches [AR19; OV14; SV10].

In the following, we combine the ideas of Section 5.5 with the hybrid rank/select dictionary of [OV14] and thus design an *improved* hybrid rank/select dictionary. This uses a two-level scheme in which the lower level stores $A$, properly partitioned into chunks (as detailed below), and the upper level stores, for each lower-level chunk $x_i, x_{i+1}, \ldots, x_j$, the integer $x_i = \mathsf{select}(i)$, the length $j - i + 1$, and a pointer to the encoding in the lower level. Therefore, the amount of bits stored in the upper level for each chunk is upper bounded by $F = \log u + 2 \log n$.

Following [OV14], we assign to a generic chunk $x_i, x_{i+1}, \ldots, x_j$ a cost $w(i, j)$ given by the sum of $F$ and a cost that depends on the encoding of the elements in that chunk. If $u' = x_j - x_i$ is the universe size, and $n' = j - i + 1$ is the number of elements in the chunk, then the cost of that encoding is the minimum of:

- $0$ bits, if $u' = n'$ and thus the chunk is a run (R) of consecutive integers in which rank/select can be computed in constant time from $x_i$ and $i$.
- $u' + o(u')$ bits, if we use a characteristic bitvector (BV) of size $u'$ augmented with the information to support rank and select in constant time.
- $n'(\lceil \log \frac{u'}{n'} \rceil + 2)$ bits, if we use Elias-Fano (EF), which supports select in $\mathcal{O}(1)$ time and rank in $\mathcal{O}(\log \frac{u'}{n'})$ time.
- $n'c + w + c + \log c_{max}$ bits, if there exists one single segment approximating all elements of the chunk with correction size $c$. In this case, select takes $\mathcal{O}(1)$ time and rank takes $\mathcal{O}(c)$ time.

Note that the last cost slightly differs from the one in Theorem 5.1. Similarly to Theorem 5.1, we use $n'c$ bits for the vector of corrections. Differently from Theorem 5.1, we use $c$ bits to encode the intercept instead of $\log n$ bits because the intercept value is guaranteed to be at most $\varepsilon$ far from the value $i$ (which is already stored in the upper level of the two-level structure), and thus it can be encoded by shifting $i$ by an amount stored in $c = \Theta(\log \varepsilon)$ bits. Also, we encode the slope in a word of $w$ bits. Finally, since we need to keep the value $c$ (which possibly changes for each segment), we use additional $\log c_{max}$ bits per segment, where $c_{max} = \mathcal{O}(\log u)$ is defined as in Section 5.5 as the minimum correction value that produces one single segment on $A$.

The overall cost in bits of the two-level structure corresponding to a partition $P$ of $A$ into $k$ chunks with endpoints $1 = i_0, i_1, \ldots, i_k = n$ is given by $w(P) = \sum_{h=0}^{k-1} w(i_h, i_{h+1} - 1)$. To solve the problem of finding an optimal partition $P$ that minimises $w(P)$, we slightly alter the algorithm of [FNV11; OV14] to consider also an encoding via segments. The algorithm of [FNV11; OV14] finds in $\mathcal{O}(n \log_{1+\epsilon} \frac{1}{\epsilon})$ time and $\mathcal{O}(n)$ space a partition whose cost is only $1 + \epsilon$ times larger than the optimal one, for any given $\epsilon \in (0, 1)$. This is done via a left-to-right scan of $A$, hence for $i = 1, \ldots, n$, that keeps $\mathcal{O}(\log_{1+\epsilon} \frac{1}{\epsilon})$ sliding windows that start all from $i$ and are such that the $k$th window covers a chunk $[i, j]$ such that either $w(i, j) \leq F(1 + \epsilon)^k < w(i, j + 1)$ or $j = n$.

A crucial property used in [OV14] is that computing $w(i, j)$ for the first three encoders above (namely, EF, BV, and R) takes constant time. Instead, computing whether there is a segment approximating the integers in a chunk requires $\mathcal{O}(n')$ time. Since we need to compute a segment for each value of $c \in \{0, 2, 3, \ldots, c_{max}\}$, computing the $(1 + \epsilon)$-optimal partition $P$ minimising $w(P)$ takes $\mathcal{O}(c_{max}\, n^2 \log_{1+\epsilon} \frac{1}{\epsilon})$ time and $\mathcal{O}(n)$ space in the presence of segments, where $c_{max} = \mathcal{O}(\log u)$.

In Section 5.7.4, we experiment with an approach that uses the greedy algorithm of Section 5.5 to compute a partition in $\mathcal{O}(c_{max}\, n \log_{1+\epsilon} \frac{1}{\epsilon})$ time and $\mathcal{O}(n + c_{max}) = \mathcal{O}(n)$ space. It operates by keeping a frontier of $c_{max}$ segments that overlap the corresponding window and by updating the frontier when the window moves, as we have seen in Section 5.5 (see the example in Figure 5.5).

## 5.7 Experiments

All our experiments were run on a machine with 40 GB of RAM and a 2.40 GHz Intel Xeon E5-2407v2 CPU.

### 5.7.1 Implementation notes

The implementation of our LA-vector is done in C++, and its code is available at `https://github.com/gvinciguerra/la_vector`. In the following, we will use the notation `la_vector<`$c$`>`, where $c$ is the correction size, to refer to our plain dictionary described in Sections 5.2 and 5.3, and use `la_vector_opt` to denote our space-optimised dictionary described in Section 5.5.

We store the segments triples $s_j = (r_j, \alpha_j, \beta_j)$ as an array of structures with memory-aligned fields. This allows for better locality and aligned memory accesses. Since in

practice the segments are few (see Figure 5.6) and fit the last-level cache, we avoid complex structures on top of the $r_j$s and the select$(r_j)$s (as suggested by Corollaries 5.1 and 5.2 to asymptotically speed up select and rank, respectively).

To further speed up rank and select, we introduce two small tables of size $2^{16}$ each that allow accessing in one hop a narrower range of segments to binary search on. These two tables use fixed-size cells of $\lceil \log \ell \rceil$ bits, because they index segments. Specifically, the table $T_1$ of size $2^{16}$ partitions the $n$ keys into blocks of size $d_1 = \lceil n/2^{16} \rceil$, so that $T_1[k]$ points to the segment covering the first key of the $k$th block. This way, a select$(i)$ query can be answered by binary searching the segments between positions $T_1[k]$ and $T_1[k+1]$, where $k = \lfloor i/d_1 \rfloor$ is the index of the block containing the $i$th key. Similarly, the small table $T_2$ of size $2^{16}$ partitions the universe into blocks of size $d_2 = \lceil u/2^{16} \rceil$, so that $T_2[y]$ points to the segment covering the first position of block $y$. This way, a rank$(x)$ query can be answered by binary searching the segments between positions $T_2[y]$ and $T_2[y+1]$, where $y = \lfloor x/d_2 \rfloor$ is the index of the block containing value $x$.

We introduce two other algorithm engineering tricks. The first one is to copy the first correction $C[r_j]$ into the segment $s_j$ structure. This improves the spatial locality of Line 1 in Algorithm 4, because both $C[r_j]$ and the values needed to compute $f_j(r_j)$ are stored nearby. The second trick is a two-level layout for $C$ that reduces the number of cache misses of Line 6 in Algorithm 4. Specifically, we split $C$ into an array $C_1$ storing all the corrections $C[i]$ such that $i$ is a multiple of an integer $d$, and an array $C_2$ containing the remaining corrections. Note that because of this split, we must slightly alter select$(i)$ so that it accesses $C_1[\lfloor i/d \rfloor]$ if $i \bmod d = 0$, and $C_2[i - \lfloor i/d \rfloor]$ otherwise. Then, we modify Line 6 to perform two binary searches. The first one touches only the $C[i]$s such that $i \bmod d = 0$. The second one touches the $\Theta(d)$ correction values in $C_2$ in-between two consecutive positions found by the first binary search. Experimentally, we found that the best performance is achieved when $d$ is roughly four cache lines of correction values (i.e. $d = \lceil 4 \cdot 512/c \rceil$ in our machine with 512-bit cache lines).

### 5.7.2 Baselines and datasets

We use the following rank/select dictionaries from the Succinct Data Structures Library (`sdsl`) [Gog+14]:

`sd_vector`: the Elias-Fano representation for increasing integer sequences with constant-time select [OS07].

`rrr_vector<t>`: a practical implementation of the $H_0$-compressed bitvector of Raman, Raman and Rao with $t$-bit blocks [CN08; RRS07].

`enc_vector<γ/δ, s>`: it encodes the gaps between consecutive integers via either Elias $\gamma$- or $\delta$-codes. Random access is implemented by storing, with sample rate $s$, an uncompressed integer and a bit-pointer to the beginning of the code of the following gap. We implemented `rank` via a binary search on the samples, followed by the sequential decoding and prefix sum of the gaps in-between two samples.

We also use the following rank/select dictionaries from the Data Structures for Inverted Indexes (`ds2i`) library [OV14]:

`uniform_partitioned`: it divides the input into fixed-sized chunks and encodes each chunk with Elias-Fano.

`opt_partitioned`: it divides the input into variable-sized chunks and encodes each chunk with Elias-Fano. The endpoints are computed by a dynamic programming algorithm that minimises the overall space.

In both structures above, endpoints and boundary values of the chunks are stored in a separate Elias-Fano data structure. For a fair comparison, we disallow the use of encoding schemes for chunks different from Elias-Fano, and we defer the experimentation of such hybrid rank/select dictionaries to Section 5.7.4.

To widen our experimental comparison, we also use:

`rle_vector<b>`: it implements a run-length encoding of the input bitvector by alternating the lengths of runs of `0`s and `1`s, coded in VByte (but over nibbles). To support efficient operations, two separate `sd_vectors` store, for each $b$-byte block, the position and the rank of the first `1`-bit in the block. [Mäk+10].[4]

`s18_vector<b>`: it uses gap and run-length encoding to compress the input bitvector via a sequence of 32-bit codes. To support efficient operations, it stores `rank` and `select` samples every $b$ codes [AW20].

We test lists of integers originating from different applications. We select these lists so that their density $n/u$ vary significantly, viz. up to three orders of magnitude. The universe size $u$ never exceeds $2^{32} - 1$, because the implementations in `ds2i` only support 32-bit integers. We use the following datasets, whose characteristics are summarised in Table 5.1.

[4]The implementation of `rle_vector` is available at `https://github.com/vgteam/sdsl-lite`.

**Table 5.1:** Characteristics of the datasets

| Dataset | Density $n/u$ | $n$ (M) | $u$ (M) | Size in MiB |
|---|---|---|---|---|
| GOV2 AVG +10M (53.0%) | 53.04% | 13.06 | 24.62 | 49.81 |
| GOV2 AVG 1M-10M (13.4%) | 13.37% | 3.29 | 24.62 | 12.56 |
| GOV2 AVG 100K-1M (1.3%) | 1.29% | 0.31 | 24.56 | 1.20 |
| URL (5.6%) | 5.58% | 57.97 | 1039.92 | 221.16 |
| URL (1.3%) | 1.30% | 13.55 | 1039.91 | 51.72 |
| URL (0.4%) | 0.36% | 3.73 | 1039.86 | 14.23 |
| 5GRAM (9.8%) | 9.85% | 145.39 | 1476.73 | 554.64 |
| 5GRAM (2.0%) | 1.98% | 29.19 | 1476.72 | 111.80 |
| 5GRAM (0.8%) | 0.76% | 11.21 | 1476.68 | 42.79 |
| DNA (30.0%) | 30.02% | 300.23 | 999.99 | 1145.32 |
| DNA (6.0%) | 6.00% | 60.03 | 999.99 | 229.00 |
| DNA (1.2%) | 1.20% | 12.00 | 999.99 | 45.79 |

**GOV2** is an inverted index built on a collection of about 25M `.gov` sites, in which document identifiers were assigned according to the lexicographic order of their URLs [OV14]. In Figures 5.2, 5.4 and 5.6, we use the longest inverted list which has a density of 76.6%. In Figures 5.7 and 5.8, we instead test all solutions over each list separately and average the results over lists of lengths 100K–1M, 1M–10M and $> 10$M. This grouping of lists by length induces an average density of 1.29%, 13.37% and 53.04%, respectively.

**URL** is a text file of 1.03 GB containing URLs originating from three sources, namely a human-curated web directory, global news, and journal articles' DOIs.[5] On this file, we first applied the Burrows-Wheeler Transform (BWT), as implemented by [FGM12], and then we generated three integer lists by enumerating the positions of the $i$th most frequent character in the resulting BWT. The different list sizes (and densities) were achieved by properly setting $i$, and they were 3.7M (0.36%), 13M (1.30%) and 57M (5.58%).

**5GRAM** is a text file of 1.4 GB containing 60M different five-word sequences occurring in books indexed by Google.[6] As for URL, we first applied the BWT and then generated three integer lists of sizes (densities): 11M (0.76%), 29M (1.98%) and 145M (9.85%).

**DNA** is the first GB of the human reference genome.[7] We generated an integer list

[5] Available at `https://kaggle.com/shawon10/url-classification-dataset-dmoz`, `https://doi.org/10.7910/DVN/ILAT5B`, and `https://archive.org/details/doi-urls`, respectively.
[6] Available at `https://storage.googleapis.com/books/ngrams/books/datasetsv3.html`.
[7] Available at `https://www.ncbi.nlm.nih.gov/assembly/GCF_000001405.39`.

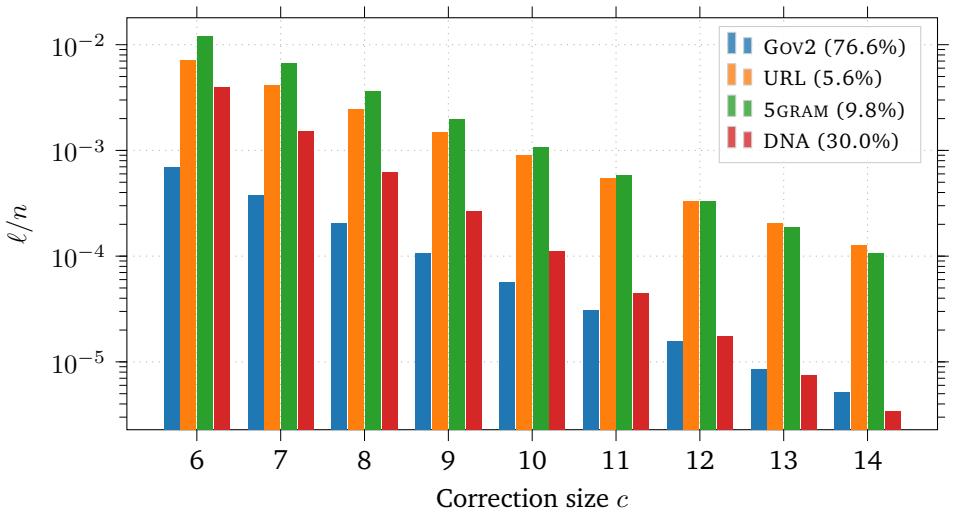



**Figure 5.6:** The ratio between the number of segments $\ell$ and the size $n$ of the largest datasets at different correction sizes $c$.

by enumerating the positions of the A nucleobase. Different densities were achieved by randomly deleting an A-occurrence with a fixed probability. The list sizes (and densities) are 12M (1.20%), 60M (6.00%) and 300M (30.02%).

As a first experiment, we show in Figure 5.6 that the number of segments $\ell$ in the piecewise linear $\varepsilon$-approximation of the various datasets is orders of magnitude smaller than the input size. These figures make our approach very promising. The following experiments will assume $c \geq 6$ for `la_vector<`$c$`>` because, on these datasets, smaller values of $c$ make $\ell$ too large, and thus the space occupied by the segments becomes significantly larger than the space taken by the correction array $C$.

### 5.7.3 Experiments on rank and select

We now experiment with the time and space performance of rank/select dictionaries by running them on each dataset (of size $n$) with a batch of $0.2n$ random queries. For clarity of the plots, we only show the implementations that use less than 16 bits per integer and whose average query time is not too high with respect to the others.

**Performance of select.** Figure 5.7 shows the results for select. We notice that our `la_vector<`$c$`>` variants consistently provide the best time performance. This comes at the cost of requiring $c$ bits per integer, plus the cost of storing the segments. For very low densities (plots in the first column) and low values of $c$, the overhead due to the segments may exceed $C$ (see e.g. 5GRAM and DNA, where the set of LA-vector configurations is U-shaped). This unlucky situation is solved by `la_vector_opt`,

which avoids the tuning of $c$ by computing the piecewise linear $\varepsilon$-approximation that minimises the overall space, possibly adopting different $c$ for different segments. Note that `la_vector_opt` is always faster than the plain Elias-Fano encoding (i.e. `sdsl::sd_vector`), except for large densities in DNA (i.e. 30%), and it is also more compressed on the GOV2, 5GRAM and URL datasets.

The other Elias-Fano encodings are generally fast as well, with `ds2i::uniform_partitioned` and `opt_partitioned` being more compressed but roughly 50 ns slower than `sdsl::sd_vector` due to the use of a two-level structure. In any case, our `la_vector_opt` and `la_vector<`$c$`>` are not dominated by these advanced Elias-Fano variants over all the datasets, except for large densities in DNA.

For what concerns `sdsl::enc_vector` and `sdsl::rrr_vector`, they are pretty slow although offering very good compression ratios. The slow performance of select in the latter is due to its implementation via a combination of a binary search on a sampled vector of ranks plus a linear search in-between two samples.

The same goes for `s18_vector`, which is very succinct but not fast, in fact, it is only on the Pareto frontier of the URL dataset.

Finally, we notice that `rle_vector` is dominated in time and space by some other data structure on all the datasets except for URL (0.4%).

**Performance of rank.** Figure 5.8 shows the results for rank. We observe that `sdsl::rrr_vector` and `sdsl::sd_vector` achieve the best time performance with LA-vector following closely, i.e. within 120 ns or less. However, at low densities (first column of Figure 5.8), `sdsl::rrr_vector` has a very poor space performance, more than 10 bits per integer.

Not surprisingly, `sdsl::enc_vector<`$\cdot, s$`>` has often the slowest rank, because it performs a binary search on a vector of $n/s$ samples, followed by the linear decoding and prefix sum of at most $s$ gaps coded with $\gamma$ or $\delta$.

`s18_vector` is very succinct but not fast, in fact, it is only on the Pareto frontier of the URL dataset, as it occurred for the select query.

`rle_vector` is dominated in time and space by some other data structure on all the datasets except for URL (0.4%), as it occurred for the select query.

Note that for GOV2, URL and 5GRAM our `la_vector_opt` falls on the Pareto frontier of Elias-Fano approaches thus offering an interesting space-time trade-off also for the rank query.

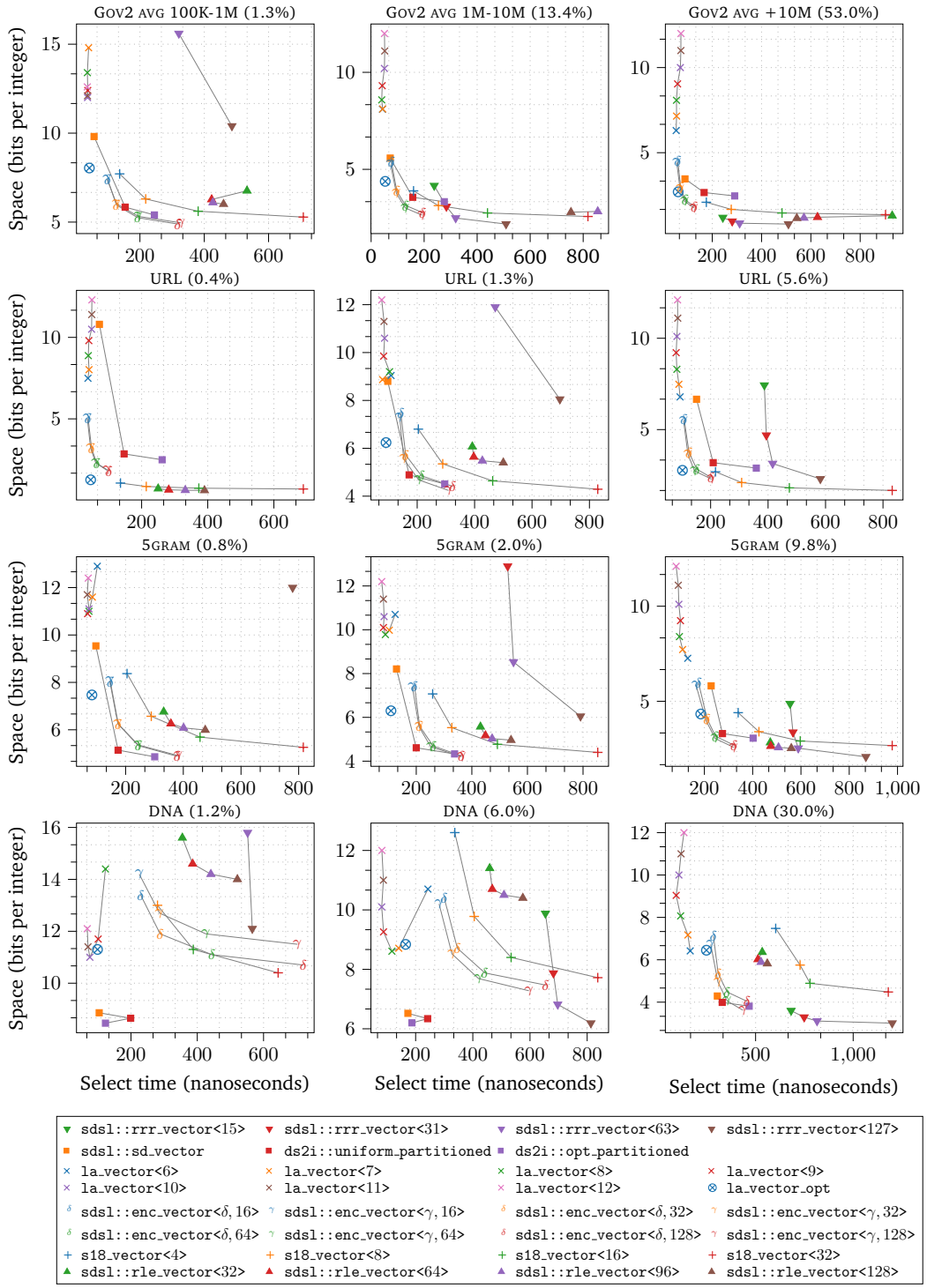



**Figure 5.7:** Space-time performance of the select query.

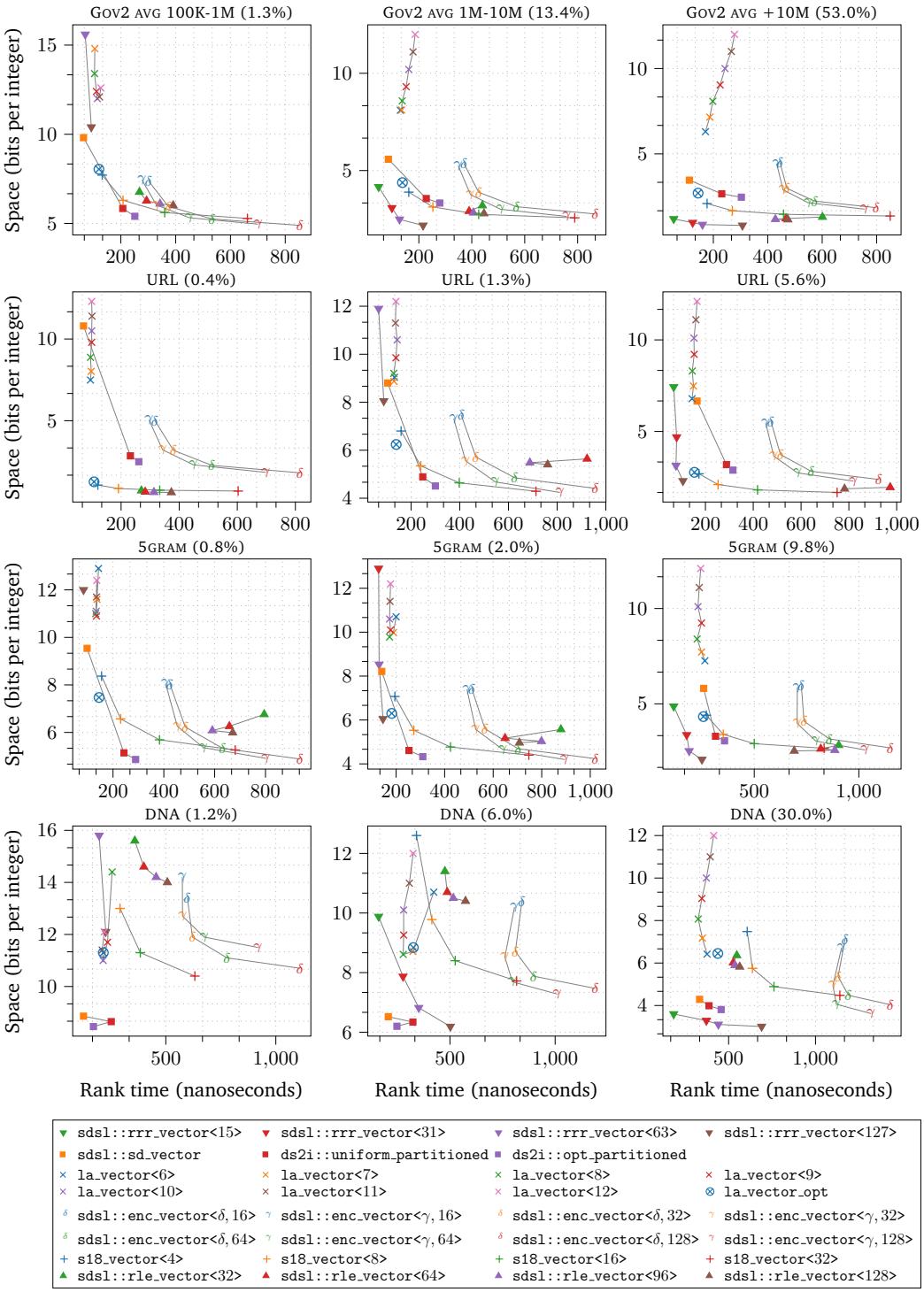



**Figure 5.8:** Space-time performance of the rank query.

**Discussion on the space-time performance.** Overall, from the experiments above, we observe that:

- `sdsl::rrr_vector` provides the fastest rank but the slowest select. Its space is competitive with other implementations only for moderate and large densities of 1s.

- The Elias-Fano approaches provide fast rank and moderately fast select in competitive space. In particular, the plain Elias-Fano (`sdsl::sd_vector`) offers fast operations but in a space competitive with other structures only on DNA; while the partitioned variants of Elias-Fano implemented in `ds2i` offer the best compression but at the cost of slower rank and select. On low densities of the DNA datasets (i.e. 6.0% and 1.2%) the implementations of `ds2i` provide the best time and space performance.

- `sdsl::enc_vector<`$\cdot, s$`>` provides a smooth space-time trade-off controlled by the $s$ parameter, but it has non-competitive rank and select operations.

- `s18_vector` is very succinct but provides generally slow rank and select operations. It is only on the Pareto frontier of the URL datasets.

- `rle_vector` is only on the Pareto frontier of URL (0.4%).

- Our `la_vector<`$c$`>` offers the fastest select, competitive rank, and a smooth space-time trade-off controlled by the $c$ parameter, where values of $c \geq 6$ were found to "balance" the cost of storing the corrections and the cost of storing the segments. Our space-optimised `la_vector_opt` in most cases (i) dominates the space-time performance of `la_vector<`$c$`>`; (ii) offers a select which is faster than all the other tested approaches; (iii) offers a rank which is on the Pareto frontier of Elias-Fano approaches.

Finally, for the construction times over the various datasets, we report that our `la_vector<`$c$`>` (we averaged over the values of $c$ used in Figures 5.7 and 5.8) builds 1.41× faster than `sdsl::enc_vector`, 2.45× faster than `sdsl::rrr_vector`, 9.74× faster than `s18_vector`, and 1.89× slower than `sdsl::sd_vector`. The space-optimised `la_vector_opt` instead builds 82.18× slower than the plain `la_vector<`$c$`>`, and 2.41× slower than the homologous space-optimised Elias-Fano (i.e. `ds2i::opt_partitioned`). Future work is needed to improve the construction performance of `la_vector_opt`.

**Table 5.2:** Number of chunks in the hybrid approach of Section 5.6 that are better compressed by a segment (plus corrections).

| Dataset | Number of chunks that use a segment | % of chunks that use a segment | % elements encoded with a segment |
|---|---|---|---|
| GOV2 AVG +10M (53.0%) | 2217 | 11.21% | 12.43% |
| GOV2 AVG 1M-10M (13.4%) | 424 | 4.57% | 9.53% |
| GOV2 AVG 100K-1M (1.3%) | 44 | 2.74% | 4.91% |
| URL (5.6%) | 30396 | 15.35% | 22.69% |
| URL (1.3%) | 7356 | 10.02% | 9.99% |
| URL (0.4%) | 2093 | 26.76% | 54.97% |
| 5GRAM (9.8%) | 31911 | 8.44% | 14.44% |
| 5GRAM (2.0%) | 5640 | 5.10% | 13.49% |
| 5GRAM (0.8%) | 1640 | 3.41% | 6.88% |
| DNA (30.0%) | 215 | 0.03% | 0.01% |
| DNA (6.0%) | 0 | 0% | 0% |
| DNA (1.2%) | 0 | 0% | 0% |

### 5.7.4 Experiments on hybrid rank/select dictionaries

We evaluate the hybrid structure of Section 5.6 that combines segments, Elias-Fano (EF), plain bitvectors (BV), and runs (R) of consecutive integers. We look in particular at how many chunks and how many integers are encoded via segments, and thus the impact of our "geometric" approach on the hybrid rank/select dictionary of [OV14].

From the results in Table 5.2, we notice that the segments are chosen as encodings of the chunks in all the datasets except for DNA (6.0%) and DNA (1.2%). The overall amount of chunks that use segments is below 16% except for URL (0.4%), where the number of chunks that use segments is very large, namely 26.76%.

As far as the percentage of integers encoded with each compression scheme is concerned, Figure 5.9 shows that segments are often selected as the best compression scheme for a substantial part of every dataset. In particular, half of the URL (0.4%) dataset is encoded with segments. Therefore, we argue that our "geometric" approach can compete with well-established succinct encoding schemes.

Looking at Figure 5.9, it is also clear that segments mainly substitute the run encoding (R). This could seem counter-intuitive since R uses 0 bits of space. But this can be explained by the fact that, for each chunk, we need to store some metadata (namely the first integer of the chunk, its length, and a pointer), and thus we can get

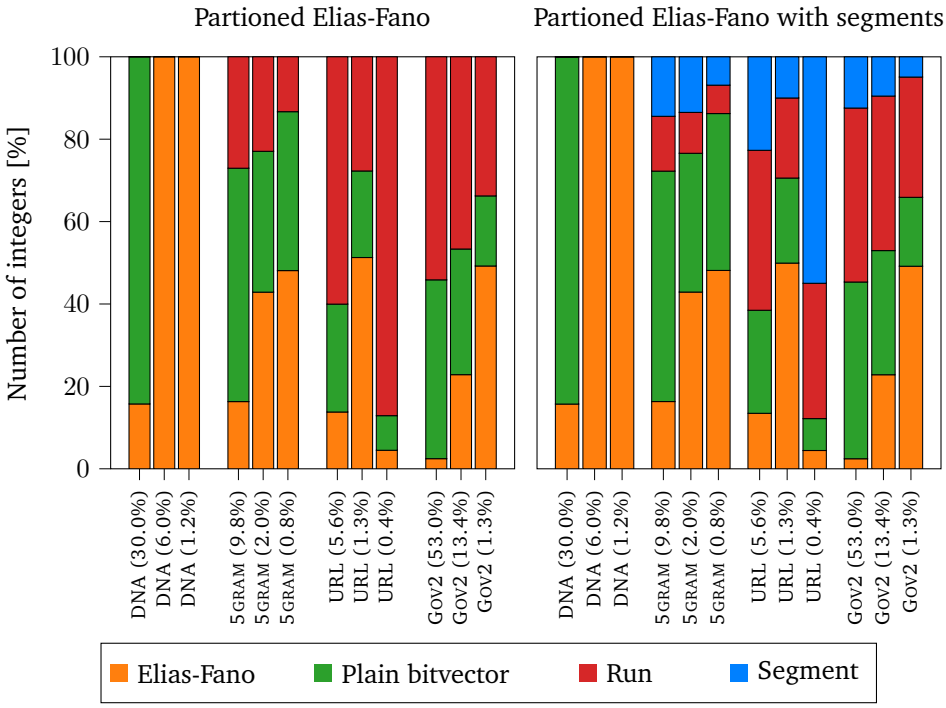



**Figure 5.9:** Percentage of integers encoded with each compression scheme

better compression by reducing the overall number of chunks, as the introduction of segments does. For example, consider a characteristic bitvector composed of $x$ equally long runs of 1 that are separated by a single 0. R would need $x$ chunks to encode that and so $x$ sets of metadata. Instead, just one segment is able to represent the $x$ runs using a few bits per integer and just 1 set of metadata. Indeed, once we introduce the segments as an encoding scheme, the total number of chunks always decreases, up to 15%. A situation similar to the previous example often happens in the BWT of high repetitive texts, and this explains the high presence of our encoding scheme in the URL and 5GRAM datasets.

Overall, the hybrid structure of Section 5.6 that combines segments, EF, BV, and R is able to use up to 1.34% less space and be just 6.5% slower on average both on rank and select than the hybrid solution without the segments. The space reduction on these datasets may not seem very impressive, but we remind the reader that the improved solution uses state-of-the-art encoders and thus it is already very squeezed. Finally, we observe that our hybrid solution turns out to be slightly slower because of the few more mathematical operations needed to work on segments in the place of R.

## 5.8 Summary

We focused on the problem of representing a compressed dictionary of integers while supporting rank and select operations, which are at the heart of virtually any compact data structure. We revisited this problem by proposing a novel learning-based compression scheme based on piecewise linear $\varepsilon$-approximations augmented with correction values. We named it LA-vector. By adding proper algorithms and data structures to this compressed representation, we showed how to support efficient rank and select operations. We also designed a piecewise linear $\varepsilon$-approximation construction algorithm that uses different $\varepsilon$ values for different chunks of the input data with the objective of minimising the overall space occupancy of the LA-vector. A comparison of LA-vector with some other well-engineered compressed rank/select dictionaries showed new space-time trade-offs.

The results in this chapter appeared in [BFV21a]. The source code of the LA-vector is publicly available at `https://github.com/gvinciguerra/la_vector`.

# 6 Repetition- and linearity-aware rank/select dictionaries

In the previous chapter, we have seen how piecewise linear $\varepsilon$-approximations can be used to capture the approximate linearity of the data and how this kind of data regularity allows designing novel succinct rank/select dictionaries. Yet, as we are about to see, piecewise linear $\varepsilon$-approximations may not consume all the possible redundancies in the data, that is, there might still be some sources of compressibility that allows squeezing further bits from the compressed representation.

One fundamental source of compressibility that piecewise linear $\varepsilon$-approximations are oblivious to is the so-called *repetitiveness* [Nav20]. To clarify, suppose the input elements of the dictionary are stored in a sorted array $A$, and consider the gap-string $S[1, n]$ defined as $S[i] = A[i] - A[i-1]$ (we let $A[0] = 0$). Say that the substring $S[i, j]$ has been encountered earlier at $S[i', i' + j - i]$ (we write $S[i, j] \equiv S[i', i' + j - i]$). Then, instead of finding a new set of segments $\varepsilon$-approximating the subarray $A[i, j]$, we can use the segments $\varepsilon$-approximating the subarray $A[i', j']$ properly shifted. Note that, even if $A[i', j']$ is covered by many segments, the same shift will transform all of them into an approximation for $A[i, j]$ (see example in Figure 6.1). Therefore, in this case, we would need to store only the *shift* and the *reference* to the segments of $A[i', j']$.

The LA-vector of Chapter 5 is unable to take advantage of such regularities. For instance, in the extreme case where $A$ consists of the concatenation of a small subarray $A'$ shifted by some amounts $\Delta_i$s for $k$ times, that is $A = A', A' + \Delta_1, A' + \Delta_2, \ldots, A' + \Delta_{k-1}$, the overall cost of representing $A$ with an LA-vector will be roughly $k + 1$ times the cost of representing $A'$.

At the opposite end, consider a binary order-$h$ De Bruijn sequence $B[1, 2^h]$, that is, a sequence in which each subsequence of length $h$ occurs exactly once. Define $A[i] = 2i + B[i]$. Then, the line with slope 2 and intercept 0 is a linear approximation of the entire array $A$ with $\varepsilon = 1$. At the same time, in the gap-string $S[i] = A[i] - A[i-1] = 2 + B[i] - B[i-1]$, we would not find repetitions longer than $h - 1$.

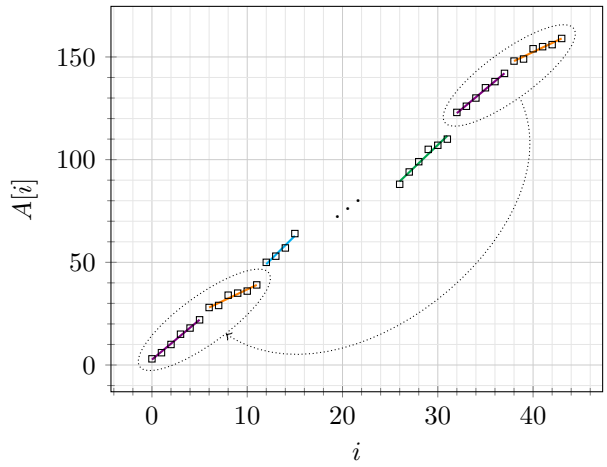



**Figure 6.1:** The points in the top-right circle follows the same "pattern" (i.e. the same distance between consecutive points) of the ones in the bottom-left circle. A piecewise linear $\varepsilon$-approximation for the top-right set can be obtained by shifting the segments for the bottom-left set.

These examples show that the approximate linearity and the repetitiveness of a string are different proxies of its compressibility and therefore it is interesting to take both of them into account, as we do in this chapter.

We do so by building on two known repetition-aware compression methods. The first method is Lempel-Ziv (LZ) parsing [LZ76; ZL77], which is one of the best-known approaches to exploit repetitiveness [Nav20]. The second method is the block tree [Bel+21], which is a recently proposed query-efficient alternative to LZ-parsing and grammar-based representations [Bel+15] suitable also for highly repetitive inputs since its space usage can be bounded in terms of the string complexity measure $\delta$, defined as $\max\{S(k)/k \mid 1 \leq k \leq n\}$, where $S(k)$ is the number of distinct length-$k$ substrings of $S$ [Ras+13; KNP20; Nav20].

In the rest of this chapter, which is based on [FMV21], we first introduce novel LZ parsings whose phrases combine backward copies with segments. The resulting representation takes a space bounded by the $k$th order entropy of the gap-string $S$,[1] and it supports rank and select in polylogarithmic time (Section 6.1). Then, we orchestrate block trees with segments and introduce another repetition- and linearity-aware data structure whose space-time performance is bounded by the string complexity measure $\delta$ (Section 6.2). Finally, we experiment with an implementation of this last result and compare it with the LA-vector and the block tree (Section 6.3).

[1] Let $\Sigma$ be the set containing the distinct gaps occurring in $S$. The zero-order entropy of $S$ is defined as $H_0(S) = \sum_{x \in \Sigma} \frac{n_x}{n} \log \frac{n}{n_x}$. The $k$th order entropy of $S$ is defined as $H_k(S) = \frac{1}{n} \sum_{y \in \Sigma^k} |S_y| H_0(S_y)$, where $S_y$ denotes the string of length $|S_y|$ consisting of the concatenation of the single gaps following each occurrence of a substring $y$ inside the gap-string $S$.

## 6.1 Two novel LZ-parsings: $\mathrm{LZ}_\varepsilon$ and $\mathrm{LZ}_\varepsilon^\rho$

Assume that $A$ contains distinct positive elements and consider the gap-string $S[1,n]$ defined as $S[i] = A[i] - A[i-1]$, where $A[0] = 0$. To make the LA-vector repetition-aware, we parse $S$ via a strategy that combines linear $\varepsilon$-approximation with LZ-end parsing [KN13].

Formally, the LZ-end parsing of a text $T[1,n]$ is a sequence $f_1, f_2, \ldots, f_z$ of phrases, such that $T = f_1 f_2 \cdots f_z$, built as follows. If $T[1,i]$ has been parsed as $f_1 f_2 \cdots f_{q-1}$, the next phrase $f_q$ is obtained by finding the longest prefix of $T[i+1,n]$ that appears also in $T[1,i]$ *ending* at a phrase boundary, i.e. the longest prefix of $T[i+1,n]$ which is a suffix of $f_1 \cdots f_r$ for some $r \leq q-1$. If $T[i+1,j]$ is the prefix with the above property, the next phrase is $f_q = T[i+1, j+1]$ (notice the addition of $T[j+1]$ to the longest copied prefix). The occurrence in $T[1,i]$ of the prefix $T[i+1,j]$ is called the *source* of the phrase $f_q$.

We generalise the phrases of the LZ-end parsing in a way that they are a "combination" of a backward copy ending at a phrase boundary (as in the classic LZ-end), computed over the gap-string $S$, plus a segment covering a subarray of $A$ with an error of at most $\varepsilon$ (unlike classic LZ-end, which instead adds a single trailing character). We call this parsing the $LZ_\varepsilon$ parsing of $S$.

Suppose that $\mathrm{LZ}_\varepsilon$ has partitioned $S[1,i]$ into $Z[1], Z[2], \ldots, Z[q-1]$. We determine the next phrase $Z[q]$ as follows (see Figure 6.2):

1. We compute the longest prefix $S[i+1,j]$ of $S[i+1,n]$ that is a suffix of the concatenation $Z[1] \cdots Z[r]$ for some $r \leq q-1$ (i.e. the source must end at a previous phrase boundary).

2. We find the longest subarray $A[j,h]$ that may be $\varepsilon$-approximated linearly, as well as the slope and intercept of such approximation. Note that using the algorithm of Lemma 2.1 the time complexity of this step is $\mathcal{O}(h-j)$, i.e. linear in the length of the processed array.

The new phrase $Z[q]$ is then the substring $S[i+1,j] \cdot S[j+1,h]$. If $h = n$, the parsing is complete. Otherwise, we continue the parsing with $i \leftarrow h+1$. As depicted in Figure 6.2, we call $S[i+1,j]$ the *head* of $Z[q]$ and $S[j+1,h]$ the *tail* of $Z[q]$. Note that the tail covers also the value $A[j]$ corresponding to the head's last position $S[j]$. When $S[i+1,j]$ is the empty string (e.g. at the beginning of the parsing), the head is empty, and thus no backward copy is executed. In the worst case, the longest subarray we can $\varepsilon$-approximate has length 2, which nonetheless guarantees that

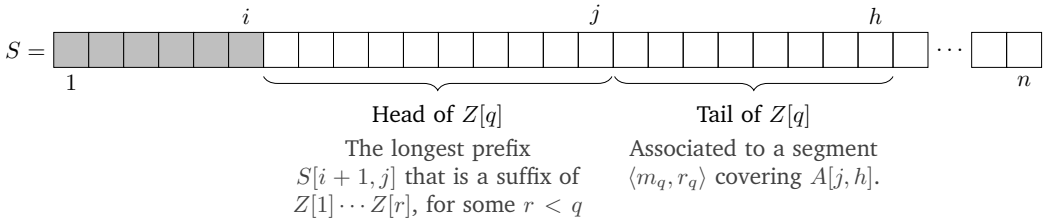



**Figure 6.2:** Computation of the next phrase $Z[q]$ in the parsing of the gap-string $S$ of the array $A$, where the prefix $S[1, i]$ has already been parsed into $Z[1], Z[2], \ldots, Z[q-1]$.

$Z[q]$ is nonempty. In practice, on the datasets of Section 5.7.2, the average segment length ranges from 76 when $\varepsilon = 31$ to 1480 when $\varepsilon = 511$.

If the complete parsing consists of $\lambda$ phrases, we store it using:

- An integer vector $\mathsf{PE}[1, \lambda]$ (Phrase Ending position) such that $h = \mathsf{PE}[q]$ is the ending position of phrase $Z[q]$, that is, $Z[q] = S[i+1, h]$, where $i = \mathsf{PE}[q-1]+1$.
- An integer vector $\mathsf{HE}[1, \lambda]$ (Head Ending position) such that $j = \mathsf{HE}[q]$ is the last position of $Z[q]$'s head. Hence, $Z[q]$'s head is $S[\mathsf{PE}[q-1]+1, \mathsf{HE}[q]]$, and $Z[q]$'s tail is $S[\mathsf{HE}[q]+1, \mathsf{PE}[q]]$.
- An integer vector $\mathsf{HS}[1, \lambda]$ (Head Source) such that $r = \mathsf{HS}[q]$ is the index of the last phrase in $Z[q]$'s source. Hence, the head of $Z[q]$ is a suffix of $Z[1] \cdots Z[r]$. If the head of $Z[q]$ is empty then $\mathsf{HS}[q] = 0$.
- A vector of pairs $\mathsf{TL}[1, \lambda]$ (Tail Line) such that $\mathsf{TL}[q] = \langle \alpha_q, \beta_q \rangle$ are the coefficients of the segment associated to the tail of $Z[q]$. By construction, such segment provides a linear $\varepsilon$-approximation for the subarray $A[\mathsf{HE}[q], \mathsf{PE}[q]]$.
- A vector of arrays $\mathsf{TC}[1, \lambda]$ (Tail Corrections) such that $\mathsf{TC}[q]$ is an array of length $\mathsf{PE}[q] - \mathsf{HE}[q] + 1$ providing the corrections for the elements in the subarray $A[\mathsf{HE}[q], \mathsf{PE}[q]]$ covered by $Z[q]$'s tail. By construction, such corrections are smaller than $\varepsilon$ in modulus.

Using the values in TL and TC we can recover the subarrays $A[j, h]$ corresponding to the phrases' tails. We show that using all the above vectors we can recover the whole array $A$.

**Lemma 6.1.** *Let $S[i+1, j]$ denote the head of phrase $Z[q]$, and let $r = \mathsf{HS}[q]$ and $e = \mathsf{PE}[r]$. Then, for $t = i+1, \ldots j$, it holds*

$$A[t] = A[t - (j - e)] + (A[j] - A[e]), \tag{6.1}$$

*where $A[j]$ (resp. $A[e]$) can be retrieved in constant time from $\mathsf{TL}[q]$ and $\mathsf{TC}[q]$ (resp. $\mathsf{TL}[r]$ and $\mathsf{TC}[r]$).*

*Proof.* By construction, $S[i+1, j]$ is identical to a suffix of $Z[1] \cdots Z[r]$. Since such a suffix ends at position $e = \mathsf{PE}[r]$, it holds $S[i+1, j] \equiv S[e-j+i+1, e]$ and

$$\begin{aligned} A[t] &= A[j] - (S[j] + S[j-1] + \cdots + S[t+1]) \\ &= (A[j] - A[e]) + A[e] - (S[e] + S[e-1] + \cdots S[t+1-(j-e)]) \\ &= (A[j] - A[e]) + A[t-(j-e)]. \end{aligned}$$

For the second part of the lemma, we notice that $A[j]$ is the first value covered by $Z[q]$'s tail, while $A[e]$ is the last value covered by $Z[r]$'s tail. □

Using the above lemma, we can show by induction that given a position $t \in [1, n]$ we can retrieve $A[t]$. The main idea is to use a binary search on $\mathsf{PE}$ to retrieve the phrase $Z[q]$ containing $t$. Then, if $t \geq \mathsf{HE}[q]$, we get $A[t]$ from $\mathsf{TL}[q]$ and $\mathsf{TC}[q]$; otherwise, we use Lemma 6.1 and get $A[t]$ by retrieving $A[t-(j-e)]$ using recursion. In the following, we will formalise this intuition in a complete algorithm, but before doing so, we need to introduce some additional notation.

Using the $\mathsf{LZ}_\varepsilon$ parsing, we partition the string $S$ into *meta-characters* as follows. The first phrase in the parsing $Z[1] = S[1, \mathsf{PE}[1]]$ is our first meta-character (note $Z[1]$ has an empty head, so $\mathsf{HE}[1] = 0$ and the pair $\langle \mathsf{TL}[1], \mathsf{TC}[1] \rangle$ encodes the subarray $A[0, \mathsf{PE}[1]]$). Now, assuming we have already parsed $Z[1] \cdots Z[q-1]$ and partitioned $S[1, \mathsf{PE}[q-1]]$ into meta-characters, we partition the next phrase $Z[q]$ into meta-characters as follows: $Z[q]$'s tail will form a meta-character by itself, while $Z[q]$'s head "inherits" the partition into meta-characters from its source. Indeed, recall that $Z[q]$'s head is a copy of a suffix of $Z[1] \cdots Z[r]$, with $r = \mathsf{HS}[q]$. Such a suffix, say $S[a, b]$, belongs to the portion of $S$ already partitioned into meta-characters. Since by construction $Z[r]$'s tail is a meta-character $X_r$, we know that $X_r$ is a suffix of $S[a, b]$. Working backwards from $X_r$ we obtain the sequence $X_0 \cdots X_r$ of meta-characters covering $S[a, b]$. Note that it is possible that $X_0$, the meta-character containing $S[a]$, starts before $S[a]$. We thus define $X'_0$ as the suffix of $X_0$ starting at $S[a]$ and define the meta-character partition of $Z[q]$'s head as $X'_0 X_1 \cdots X_r$. This process is depicted in Figure 6.3. Note that each meta-character is either the tail of some phrase or it is the suffix of a tail. We do not really compute the meta-characters but only use them in our analysis, as in the following result.

**Lemma 6.2.** *Algorithm 5 computes select$(t) = A[t]$ in $\mathcal{O}(\log \lambda + M_{\max})$ time, where $\lambda$ is the number of phrases in the $\mathsf{LZ}_\varepsilon$ parsing and $M_{\max}$ is the maximum number of meta-characters in a single phrase.*

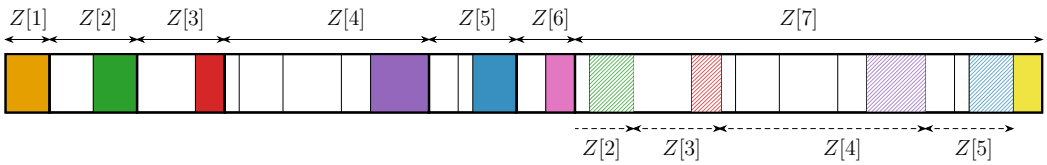



**Figure 6.3:** The $\mathrm{LZ}_\varepsilon$ parsing with the definition of meta-characters. Cells represent meta-characters, and the coloured cells are also tails. $Z[7]$'s head consists of a copy of a substring that starts inside $Z[2]$ and ends at the end of $Z[5]$ (notice the diagonal patterns in $Z[7]$'s head with the same colours of the tails in $Z[2]\cdots[5]$). Meta-characters in $Z[7]$'s head are defined from the meta-characters in the copy. Note that $Z[7]$'s first meta-character is a suffix of $Z[2]$'s first meta-character.

**Algorithm 5** Recursive select procedure.

```
1:  procedure SELECT(t)
2:      q ← the smallest i such that PE[i] ≥ t, found via a binary search on PE
3:      return SELECT-AUX(t,q)
4:  procedure SELECT-AUX(t,q)          ▷ Invariant: PE[q − 1] < t ≤ PE[q]
5:      if t > HE[q] then              ▷ If t belongs to the tail of Z[q]
6:          return A[t]                ▷ A[t] is computed from TL[q], TC[q]
7:      r ← q′ ← HS[q]                 ▷ The head of Z[q] is a suffix of Z[1]···Z[r]
8:      j ← HE[q]                      ▷ j is the last position of the head of Z[q]
9:      e ← PE[r]                      ▷ e is the last position of Z[r]
10:     Δ ← A[j] − A[e]                ▷ Δ is computed in O(1) time by Lemma 6.1
11:     t′ ← t − (j − e);              ▷ A[t] = A[t′] + Δ by Lemma 6.1
12:     while t′ > PE[q′] do           ▷ Find the phrase Z[·] containing t′
13:         q′ ← q′ − 1                ▷ Go back one word
14:     return SELECT-AUX(t′, q′) + Δ  ▷ The returned value is A[t] by Lemma 6.1
```

*Proof.* The correctness of the algorithm follows by Lemma 6.1. To prove the time bound, observe that Line 2 clearly takes $\mathcal{O}(\log\lambda)$ time. Let $\ell$ denote the number of meta-characters between the one containing position $t$ up to the end of $Z[q]$. We show by induction on $\ell$ that SELECT-AUX$(t,q)$ takes $\mathcal{O}(\ell)$ time. If $\ell = 1$, then $t$ belongs to $Z[q]$'s tail, and the value $A[t]$ is retrieved in $\mathcal{O}(1)$ time from $\mathsf{TL}[q]$ and $\mathsf{TC}[q]$.

If $\ell > 1$, the algorithm retrieves the value $A[t']$ from a previous phrase $Z[q']$, with $q' = r - k$, where $k$ is the number of times Line 13 is executed. Since $Z[q]$ meta-characters are induced by those in its source, we get that the number of meta-characters between the one containing $t'$ and the end of $Z[r]$ is $\ell - 1$, and the number of meta-characters between the one containing $t'$ and the end of $Z[q']$ is $\ell' \leq \ell - 1 - k$. By the inductive hypothesis, the call to SELECT-AUX$(t',q')$ takes $\mathcal{O}(\ell')$, and the overall cost of SELECT-AUX$(t,q)$ is $\mathcal{O}(k) + \mathcal{O}(\ell') = \mathcal{O}(\ell)$, as claimed. □

It is easy to see that for some input $t$ Algorithm 5 takes $\Theta(M_{\max})$ time. To reduce the complexity, we now show how to modify the parsing so that $M_{\max}$ is upper

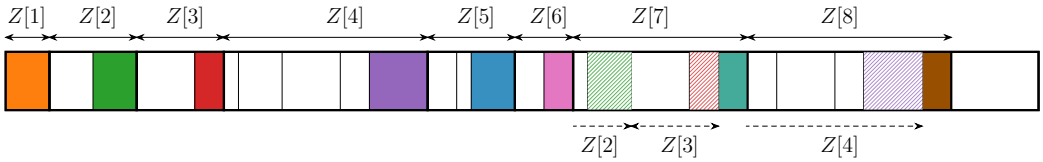



**Figure 6.4:** The $\mathrm{LZ}_\varepsilon$ parsing of the same string of Figure 6.3 with $M = 5$. The phrase $Z[7]$ from Figure 6.3 is invalid since it has 13 meta-characters. $Z[7]$ head can have at most 4 meta-characters, so we define $Z[7]$ by setting $\mathsf{HS}[7] = 3$ (Step 2b). Next, we define $Z[8]$ by setting $\mathsf{HS}[8] = 4$ (Step 2c).

bounded by a user-defined parameter $M > 1$. The resulting parsing could contain some repeated phrases, but note that Lemma 6.2 does not require the phrases to be different: repeated phrases will only affect the space usage.

To build a $\mathrm{LZ}_\varepsilon$ parsing in which each phrase contains at most $M$ meta-characters, we proceed as follows. Assuming $S[1, i]$ has already been parsed as $Z[1], \ldots, Z[q-1]$, we first compute the longest prefix $S[i+1, j]$ which is a suffix of $Z[1] \cdots Z[r]$ for some $r < q$. Let $m$ denote the number of meta-characters in $S[i+1, j]$. Then (see Figure 6.4):

1. If $m < M$, then $Z[q]$ is defined as usual with $\mathsf{HS}[q] = r$. Since $Z[q]$'s tails constitute an additional meta-character, $Z[q]$ has $m + 1 \leq M$ meta-characters, as required.

2. Otherwise, if $m \geq M$, we do the following.

   a) We scan $S[i+1, j]$ backward dropping copies of $Z[r], Z[r-1], \ldots$ until we are left with a prefix $S[i+1, k_s]$ containing less than $M$ meta-characters. By construction, $S[i+1, k_s]$ is a suffix of $Z[1] \cdots Z[s]$ for some $s < r$ and since each phrase contains at most $M$ meta-characters, $S[i+1, k_s]$ is non-empty.

   b) We define $Z[q]$ by setting $S[i+1, k_s]$ as its head, $\mathsf{HS}[q] = s$, and by defining $Z[q]$'s tail as usual.

   c) Next, we consider $Z[s+1] \equiv S[k_s, k_{s+1}]$. By construction, $Z[s+1]$ contains at most $M$ meta-characters while $S[i+1, k_{s+1}]$ contains more than $M$ meta-characters. If $Z[q]$ ends before position $k_{s+1}$ (i.e. $\mathsf{PE}[q] < k_{s+1}$), we define an additional phrase $Z[q+1]$ with heads equal to $S[\mathsf{PE}[q]+1, k_{s+1}]$, $\mathsf{HS}[q+1] = s+1$ and with a tail defined as usual. This ensures that $Z[q]$ alone or $Z[q]Z[q+1]$ contains at least $M$ meta-characters.

**Lemma 6.3.** *The $\mathrm{LZ}_\varepsilon$ parsing with limit $M$ contains at most $2n/M$ repeated phrases.*

*Proof.* In the algorithm described above, repeated phrases are created only at Steps 2b and 2c. Indeed, both $Z[q]$ defined in Step 2b and $Z[q+1]$ defined in Step 2c could be identical to a previous phrase. However, the concatenation $Z[q]Z[q+1]$ covers at least $S[i+1, k_{s+1}]$ so by construction contains *at least* $M$ meta-characters. Hence, Steps 2b and 2c can be executed at most $n/M$ times. □

In the following, let $\sigma$ denote the number of distinct gaps in $S$ (i.e., the alphabet size of $S$), for any $\rho > 0$, we denote by $\mathrm{LZ}_\varepsilon^\rho$ the parsing computed with the above algorithm with $M = \log^{1+\rho} n$. The following lemma shows that the space to represent the parsing can be bounded in terms of the $k$th order entropy of the gap-string $S$ (defined in footnote 1) plus $o(n \log \sigma)$ bits.

**Lemma 6.4.** *Let $\sigma$ denote the number of distinct gaps in $S$. The arrays* PE*,* HE*, and* HS *produced by the $\mathrm{LZ}_\varepsilon^\rho$ parsing can be stored in $nH_k(S) + \mathcal{O}(n/\log^\rho n) + o(n \log \sigma)$ bits for any positive $k = o(\log_\sigma n)$, and still support constant-time access to their elements.*

*Proof.* Let $\lambda$ denote the number of phrases in the parsing. We write $\lambda = \lambda_r + \lambda_d$, where $\lambda_r$ is the number of repeated phrases, and $\lambda_d$ is the number of distinct phrases. By Lemma 6.3 it is $\lambda_r \leq n/(2\log^{1+\rho} n)$, while for the number $\lambda_d$ of distinct phrases it is [KN13, Lemmas 3.9 and 3.10]

$$\lambda_d = \mathcal{O}\left(\frac{n}{\log_\sigma n}\right) \quad \text{and} \quad \lambda_d \log \lambda_d \leq nH_k(S) + \lambda_d \log \tfrac{n}{\lambda_d} + \mathcal{O}(\lambda_d(1 + k \log \sigma)) \quad (6.2)$$

for any constant $k \geq 0$. The vectors PE and HE contain $\lambda$ increasing values in the range $[1, n]$. We encode each of them in $\lambda \log \frac{n}{\lambda} + \mathcal{O}(\lambda)$ bits using Lemma 4.1. Since $f(x) = x \log(n/x)$ is increasing for $x \leq n/e$ and $\lambda = \mathcal{O}(n/\log_\sigma n)$, it is $\lambda \log \frac{n}{\lambda} + \mathcal{O}(\lambda) = \mathcal{O}(n(\log \sigma)(\log\log n)/\log n) = o(n \log \sigma)$.

We encode HS using $\lambda$ cells of size $\lceil \log \lambda \rceil = \log \lambda + \mathcal{O}(1)$ bits for a total of

$$\lambda_r \log(\lambda_r + \lambda_d) \;+\; \lambda_d \log(\lambda_r + \lambda_d) \;+\; \mathcal{O}(\lambda) \;\text{ bits.}$$

Since $\lambda_d = \mathcal{O}(n/\log_\sigma n)$ and $\lambda_r = \mathcal{O}(n/\log^{1+\rho} n)$, it is $\lambda_d + \lambda_r = \mathcal{O}(n/\log_\sigma n)$ and the first term is $\mathcal{O}(n/\log^\rho n)$. The second term can be bounded by noticing that, if $\lambda_d \leq \lambda_r$, the second term is smaller than the first. Otherwise, from (6.2) we have

$$\lambda_d \log(\lambda_r + \lambda_d) \;\leq\; \lambda_d \log(2\lambda_d) \;\leq\; nH_k(S) + \lambda_d \log \tfrac{n}{\lambda_d} + \mathcal{O}(\lambda_d(1 + k \log \sigma)).$$

By the same reasoning as above, we have $\lambda_d \log \frac{n}{\lambda_d} = o(n \log \sigma)$ and $\lambda_d(1 + k \log \sigma) = \mathcal{O}((nk \log \sigma)/\log_\sigma n) = o(n \log \sigma)$ for $k = o(\log_\sigma n)$. □

Combining Lemma 6.4 with 6.2 and recalling that $\log \lambda = \mathcal{O}(\log^{1+\rho} n)$, we get

**Theorem 6.1.** *Let $\sigma$ denote the number of distinct gaps in $S$. Using the $\mathrm{LZ}_\varepsilon^\rho$ parsing we can compute* select$(t)$ *in $\mathcal{O}(\log^{1+\rho} n)$ time using $nH_k(S) + \mathcal{O}(n/\log^\rho n) + o(n \log \sigma)$ bits of space plus the space used for the $\lambda$ segments (array* TL*) and for the corrections of the elements in $A$ covered by the tails in the parsing (array* TC*), for any positive $k = o(\log_\sigma n)$.*

In the proof of Lemma 6.4 one can see the interplay between the term $\mathcal{O}(n/\log^\rho n)$ coming from the repeated phrases and the term $o(n \log \sigma)$ coming from the distinct phrases in $\mathrm{LZ}_\varepsilon^\rho$. In particular, if $\sigma$ is small (i.e., there are few distinct gaps), then $o(n \log \sigma)$ becomes $\mathcal{O}(n \log \log n / \log n)$ and the space bound turns out to be $nH_k(S) + \mathcal{O}(n/\log^\rho n)$ bits. Also, note that the number of segments $\lambda$ in $\mathrm{LZ}_\varepsilon^\rho$ is always smaller than the number of segments in a plain LA-vector. Also, the total length of the $\mathrm{LZ}_\varepsilon^\rho$ tails is always smaller than $n$. Hence, our approach is no worse than the LA-vector in space.

We now show that the $\mathrm{LZ}_\varepsilon^\rho$ parsing support efficient rank queries. The starting point is the following lemma, whose proof is analogous to the one of Lemma 6.1.

**Lemma 6.5.** *Let $S[i+1, j]$ denote the head of phrase $Z[q]$, and let $r = \mathsf{HS}[q]$ and $e = \mathsf{PE}[r]$. Then, for any $v$ such that $A[i] < v \leq A[j]$, it holds* rank$(v) =$ *rank$(v - (A[j] - A[e])) + (j - e)$.*

**Theorem 6.2.** *Using the $\mathrm{LZ}_\varepsilon^\rho$ parsing we can compute* rank$(v)$ *in $\mathcal{O}(\log^{1+\rho} n + \log \varepsilon)$ time within the space stated in Theorem 6.1.*

*Proof.* We answer $\mathsf{rank}(v)$ with an algorithm similar to Algorithm 5. First, we compute the index $q$ of the phrase $Z[q]$ such that $A[\mathsf{PE}[q-1]] < v \leq A[\mathsf{PE}[q]]$ with a binary search on the values $A[\mathsf{PE}[i]]$. If the parsing has $\lambda$ phrases, this takes $\mathcal{O}(\log \lambda)$ time, since we can retrieve $A[\mathsf{PE}[i]]$ in constant time using $\mathsf{PE}[i]$, $\mathsf{TL}[i]$ and $\mathsf{TC}[i]$.

Next, we set $j = \mathsf{HE}[q]$ and compare $v$ with $A[j]$ (which again we can retrieve in constant time since it is the first value covered by $Z[q]$'s tail). If $v \geq A[j]$, we return $j$ plus the rank of $v$ in $A[j, \mathsf{PE}[q]]$, which we can compute in $\mathcal{O}(\log \varepsilon)$ time from $\mathsf{TL}[q]$ and $\mathsf{TC}[q]$ using the algorithm in Section 5.3. If $v < A[j]$, we set $e = \mathsf{PE}[HS[q]]$ and compute $\mathsf{rank}(v)$ recursively using Lemma 6.5. Before the recursive call, we need to compute the index $q'$ of the phrase such that $A[\mathsf{PE}[q'-1]] < v' \leq A[\mathsf{PE}[q']]$, for $v' = v - (A[j] - A[e])$. To this end, we execute the same while loop as the one in Lines 12–13 of Algorithm 5 with the test $t' > \mathsf{PE}[q']$ replaced by $v' > A[\mathsf{PE}[q']]$. Reasoning as in the proof of Lemma 6.2, we get that the overall time complexity is $\mathcal{O}(\log \lambda + M_{\max} + \log \varepsilon) = \mathcal{O}(\log^{1+\rho} n + \log \varepsilon)$. □

### Discussion

Overall, Theorems 6.1 and 6.2 show that, if $\sigma$ denotes the number of distinct gaps in $S$, the $\mathrm{LZ}_\varepsilon^\rho$ parsing supports rank in $\mathcal{O}(\log^{1+\rho} n + \log \varepsilon)$ time and select in $\mathcal{O}(\log^{1+\rho} n)$ time using $nH_k(S) + \mathcal{O}(n/\log^\rho n) + o(n \log \sigma)$ bits of space, for any positive $\rho$ and $k = o(\log_\sigma n)$, plus the space to store the segments and the correction values that are used to advance the parsing (like the explicit characters in traditional LZ-parsing).

On the other hand, the most common approach in the literature to design an $H_k$-compressed dictionary for a set of distinct integers $A$ over the universe $\{0, \ldots, u\}$ is to represent $A$ using the characteristic bitvector $\mathtt{bv}(A)$, which has length $u + 1$ and is such that $\mathtt{bv}(A)[i] = \mathtt{1}$ iff $i \in A$.

The best succinct data structure based on $\mathtt{bv}(A)$ is the one by [SG06] that supports constant-time rank and select in $uH_k(\mathtt{bv}(A)) + \mathcal{O}(u \log\log u / \log u)$ bits of space. This space bound cannot be compared to ours since it is given in terms of $H_k(\mathtt{bv}(A))$ instead of $H_k(S)$. To achieve space $nH_k(S)$ one can use an entropy-compressed representation of $S$ enriched with auxiliary data structures to support rank/select on $A$. For example, by sampling one value of $A$ out of $\log n$ and performing a binary search followed by a prefix sum of the gaps one can support $\mathcal{O}(\log n)$-time rank and select queries. Using the representation of [FV07], this solution uses $nH_k(S) + \mathcal{O}(n \log u / \log n) + o(n \log \sigma) = nH_k(S) + o(n \log u)$ bits of space, which is worse in space than our solution but faster in query time. Other trade-offs are possible: the crucial point however is that none of the known techniques is able to exploit simultaneously the presence of exact repetitions and approximate linearity in the input data as instead our $\mathrm{LZ}_\varepsilon^\rho$ does. In the best scenario, $\mathrm{LZ}_\varepsilon^\rho$ parsing uses segments to quickly consume any approximate linearity in $A$ thus potentially reducing significantly the number of LZ-phrases. On the other hand, if $A$ cannot be linearly approximated, segments will be short and the overall space occupancy of $\mathrm{LZ}_\varepsilon^\rho$ parsing will be $nH_k(S) + o(n \log \sigma)$ bits, i.e. no worse than a traditional LZ-parsing.

## 6.2 The block-$\varepsilon$ tree

In this section, we design a repetition aware version of the LA-vector by following an approach that focuses on query efficiency and uses space bounded in terms of the complexity measure $\delta$. We do so by building a variant of the block tree [Bel+21] on a combination of the gap-string $S$ and the piecewise linear $\varepsilon$-approximation. We name this variant block-$\varepsilon$ tree, and show that it achieves time-space bounds which

are competitive with the ones achieved by block trees and LA-vectors because it combines both forms of compressibility discussed in this paper: repetitiveness and approximate linearity.

The main idea of the block-$\varepsilon$ tree consists in first building a traditional block tree structure over the gap-string $S[1, n]$ of $A$. Recall that every node of the block tree represents a substring of $S$, and thus it implicitly represents the corresponding subarray of $A$. Then, we prune the tree by dropping the subtrees whose corresponding subarray of $A$ can be covered more succinctly by segments and corrections (i.e. whose LA-vector representation wins over the block-tree representation). Note that, compared to LA-vector, we do not encode segments and corrections corresponding to substrings of $S$ that have been encountered earlier, that is, we exploit the repetitiveness of $S$ to compress the piecewise linear $\varepsilon$-approximation at the core of LA-vector. On the other hand, compared to block trees, we drop subtrees whose substrings can be encoded more efficiently by segments and corrections, that is, we exploit the approximate linearity of subarrays of $A$. Below we detail how to orchestrate this interplay to achieve efficient queries and compressed space occupancy in the block-$\varepsilon$ tree.

For simplicity of exposition, assume that $n = \delta 2^h$ for some integer $h$, where $\delta$ is the string complexity of $S$. The block-$\varepsilon$ tree is organised into $h' \leq h$ levels. The first level (level zero) logically divides the string $S$ into $\delta$ blocks of size $s_0 = n/\delta$. In general, blocks at level $\ell$ have size $s_\ell = n/(\delta 2^\ell)$, because they are recursively halved until possibly reaching the last level $h = \log \frac{n}{\delta}$, where blocks have size $s_h = 1$.

At any level, if two blocks $S_q$ and $S_{q+1}$ are consecutive in $S$ and they form the leftmost occurrence in $S$ of their content, then we say that both $S_q$ and $S_{q+1}$ are marked. A marked block $S_q$ that is not in the last level becomes an internal node of the tree. Such an internal node has two children corresponding to the two equal-size sub-blocks in which $S_q$ is split into. On the other hand, an unmarked block $S_r$ becomes a leaf of the tree because, by construction, its content occurs earlier in $S$ and thus we can encode it by storing (i) a leftward pointer $q$ to the marked blocks $S_q, S_{q+1}$ at the same level $\ell$ containing its leftmost occurrence, taking $\log \frac{n}{s_\ell}$ bits; (ii) the offset $o$ of $S_r$ into the substring $S_q \cdot S_{q+1}$, taking $\log s_\ell$ bits. Furthermore, to recover the values of $A$ corresponding to $S_r$, we store (iii) the difference $\Delta$ between the value of $A$ corresponding to the beginning of $S_r$ and the value of $A$ at the pointed occurrence of $S_r$, taking $\log u$ bits. Overall, each unmarked block needs $\log n + \log u$ bits of space.

To describe the pruning process, we first define a cost function $c$ on the nodes of the block-$\varepsilon$ tree. For an unmarked block $S_r$, we define the cost $c(S_r) = \log n + \log u$,

which accounts for the space in bits taken by $q$, $o$ and $\Delta$. For a marked block $S_q$ at the last level $h$, we define the cost $c(S_q) = \log u$, which accounts for the space in bits taken by its single corresponding element of $A$. Instead, consider a marked block $S_q$ at level $\ell < h$ for which there exists a segment approximating with error $\varepsilon_q \leq \varepsilon$ the corresponding elements of $A$. Suppose $\varepsilon_q$ is minimal, that is, there is no $\varepsilon' < \varepsilon_q$ such that there exists a segment $\varepsilon'$-approximating those same elements of $A$. Let $\kappa$ be the space in bits taken by the parameters $\langle \alpha, \beta \rangle$ of the segment, e.g. $\kappa = 2 \log u + \log n$ if we encode $\beta$ in $\log u$ bits and $\alpha$ as a rational number with a $\log u$-bit numerator and a $\log n$-bit denominator. We assign to such $S_q$ a cost $c(S_q)$ defined recursively as

$$c(S_q) = \min \begin{cases} \kappa + s_\ell \log \varepsilon_q + \log \log u \\ 2 \log n + \sum_{S_x \in child(S_q)} c(S_x) \end{cases} \tag{6.3}$$

The first branch of Equation 6.3 accounts for an encoding of the subarray of $A$ corresponding to $S_q$ via an $\varepsilon_q$-approximate segment, the corrections of $\log \varepsilon_q$ bits for each of the $s_\ell$ elements in $S_q$, and the exponent $y$ of $\varepsilon_q = 2^y - 1$ to keep track of its value, respectively. The second branch of Equation 6.3 accounts for an encoding that recursively splits $S_q$ into two children, i.e. an encoding via two $\log n$-bit pointers plus the optimal cost of the children. Finally, if there is no linear $\varepsilon$-approximation (and thus no $\varepsilon_q$-approximation with $\varepsilon_q \leq \varepsilon$) for $S_q$, we assign to such $S_q$ the cost indicated in the second branch of Equation 6.3.

A postorder traversal of the block-$\varepsilon$ tree is sufficient to assign a cost to its nodes and possibly prune some of its subtrees. Specifically, after recursing on the two children of a marked block $S_q$ at level $\ell$, we check if the first branch of Equation 6.3 gives the minimum. In that case, we prune the subtree rooted at $S_q$ and store instead the encoding of the block via the parameters $\langle \alpha, \beta \rangle$ and the $s_\ell$ corrections in an array $C_q$. As a technical remark, this pruning requires fixing the destination of any leftward pointer that starts from an unmarked block $S_r$ and ends to a (pruned) descendant of $S_q$. For this purpose, we first make $S_r$ pointing to $S_q$. Then, since any leftward pointer points to a *pair* of marked blocks (unless the offset is zero), both or just one of them belongs to the pruned subtree. In the second case, we require an additional pointer from $S_r$ to the block that does not belong to the pruned subtree. This additional pointer does not change the asymptotic complexity of the structure.

Overall, this pruning process yields a tree with $h' \leq h$ levels. An example of block-$\varepsilon$ tree is depicted in Figure 6.5.

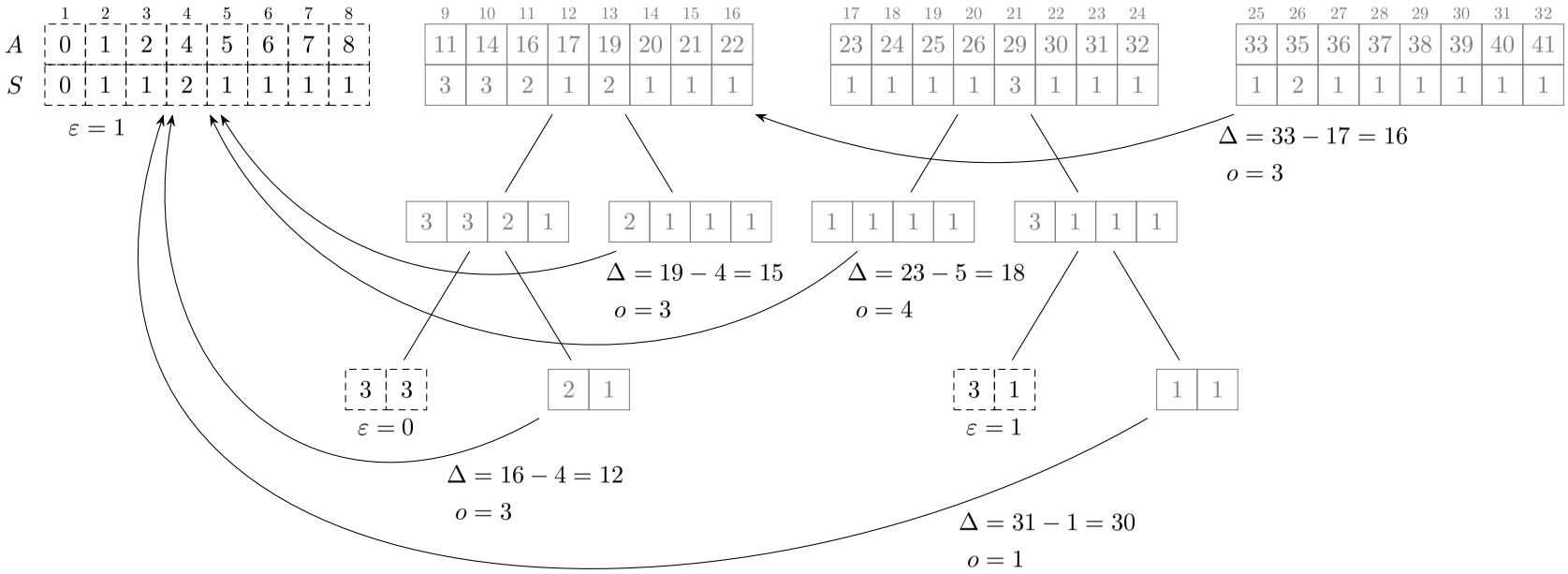



**Figure 6.5:** An example of block-$\varepsilon$ tree built on an input array $A$ with corresponding gap-string $S$. The dashed blocks represent blocks encoded with a segment (whose $\varepsilon$ value is shown below the block). A leftward pointer from a block $S_r$ to $S_q$ is annotated with the offset $o$ of the occurrence of $S_r$ into the substring $S_q \cdot S_{q+1}$ and with the difference $\Delta$ between the value of $A$ corresponding to the beginning of $S_r$ and the one at the pointed occurrence. The grey blocks are conceptual and not stored.

## Supporting rank and select

To answer $\mathsf{select}(i)$ in the block-$\varepsilon$ tree, we follow the path that starts from the first-level block in which position $i$ falls and proceeds towards a marked leaf block. We have the following cases for a visited block at level $\ell$:

- The current block is an unmarked block $S_r$ pointing to $q$ with offset $o$ and difference value $\Delta = A[b] - A[a]$, where $b$ is the position corresponding to the beginning of $S_r$, and $a$ is the position corresponding to the beginning of the copy within $S_q$. First, we jump to either $S_q$ or $S_{q+1}$ depending on whether $o < s_\ell$, where $s_\ell$ is the size of the blocks at level $\ell$. Then, we turn the $\mathsf{select}(i) = A[i]$ query to $\Delta + \mathsf{select}(a + i - b) = \Delta + A[a + i - b]$.[2] In fact, it holds

$$\begin{aligned}\Delta + A[a+i-b] &= \Delta + A[a] + S[a+1] + \cdots + S[a+i-b] \\ &= A[b] - A[a] + A[a] + S[a+1] + \cdots + S[a+i-b] \\ &= A[b] + S[a+1] + \cdots + S[a+i-b] \\ &= A[b] + S[b+1] + \cdots + S[b+i-b] \\ &= A[b+i-b] = A[i].\end{aligned}$$

- The current block is a marked internal block. We jump to its first or second child depending on whether $i \bmod s_\ell < s_\ell/2$, and we continue computing $\mathsf{select}(i)$.
- The current block is a marked leaf block $S_q$ storing the segment parameters $\langle \alpha, \beta \rangle$ and the local corrections $C_q$. We return $\alpha i + \beta + C_q[i \bmod s_\ell]$.
- The current block is a marked leaf block $S_q$ at the last level $h$, thus we return its single element.

Let us now compute the time complexity of this traversal. First observe that, if we encounter a pruned block, the traversal stops. If we encounter an unmarked block, we follow its pointer to a pruned block or to an internal node. In this latter case, the traversal proceeds top-down with a constant amount of work per level. Therefore, the time complexity of $\mathsf{select}$ is $\mathcal{O}(h')$.

For $\mathsf{rank}$ queries, we create the predecessor structure on the $\delta$ integers of $A$ corresponding to the first-level blocks, i.e. the integers $A[in/\delta]$ for $i = 1, \ldots, \delta$. We use the

[2] Special care must be taken when either or both $S_q$ and $S_{q+1}$ are pruned blocks. In this case, $S_r$ points to their ancestor, which is associated to a segment, and the argument of $\mathsf{select}$ can be recomputed accordingly.

structure of [BN15, Appendix A] giving a query time of $\mathcal{O}(\log \log_w \frac{u}{\delta})$, where $w$ is the word size, but there are many other possible trade-offs [NR20] that we skip for simplicity of exposition. Furthermore, in each marked block $S_q$ at any level $\ell$ except the first and the last ones, we store the value $y$ of $A$ corresponding to the middle of $S_q$ to descend to the correct child. The extra information does not change the asymptotic space complexity of our structure.

To answer $\mathsf{rank}(x)$, we start with a query to the predecessor structure, which indicates the first-level block in which $x$ falls into, and then we proceed towards a marked leaf block. The traversal proceeds similarly to select, so we only give a sketch. If the visited block is a marked leaf block $S_q$ at level $\ell$, we retrieve its $\langle \alpha, \beta \rangle$ parameters and the correction array $C_q$, and then we perform a binary search for $x$ on these $s_\ell$ corrections. Using the algorithm of Section 5.3, this search costs $\mathcal{O}(\log \varepsilon_q)$ time and returns the result of $\mathsf{rank}(x)$, which is the position in $A$ of (the predecessor of) the value $x$.

If the visited block is a marked internal block, we descend to its left or right child in constant time by using the aforementioned value $y$ and return the result of $\mathsf{rank}(x)$ in this block.

Finally, if the visited block is an unmarked block $S_r$, we recursively issue a rank-query on the marked block $S_q$ pointed to by $S_r$ with argument $x - \Delta$, where $\Delta$ is defined as above, and then return $b - a + \mathsf{rank}(x - \Delta)$. The shift $b - a$ takes into account the leftward jump induced by the fact that we solve the rank query not on $S_r$ but on $S_q$.[3]

Overall, the time complexity of rank is given by the sum of the costs of the initial predecessor search, the traversal of the block-$\varepsilon$ tree, and the final binary search, thus it is equal to $\mathcal{O}(\log \log_w \frac{u}{\delta} + h' + \log \varepsilon)$.

## Discussion

Overall, we have shown that the block-$\varepsilon$ tree supports rank in $\mathcal{O}(\log \log \frac{u}{\delta} + \log \frac{n}{\delta} + \log \varepsilon)$ time and select in $\mathcal{O}(\log \frac{n}{\delta})$ time using $\mathcal{O}(\delta \log \frac{n}{\delta} \log n)$ bits of space in the worst case, where $\delta$ is the string complexity of $S$.

We observe that the block-$\varepsilon$ tree achieves space-time complexities no worse than a standard block tree construction on $S$. This is due to the pruning of subtrees guided by the space-conscious cost function $c(\cdot)$ and by the resulting reduction in the number of levels, which positively impact the query time.

[3] Here, the same considerations of footnote 2 apply.

One could also consider a standard block tree construction on $\texttt{bv}(A)$ that, instead, supports rank and select in $\mathcal{O}(\log \frac{u}{\delta'})$ time using $\mathcal{O}(\delta' \log \frac{u}{\delta'} \log u)$ bits of space, where $\delta'$ is the string complexity measure on $\texttt{bv}(A)$. The time and space bounds achieved by the block tree and by our block-$\varepsilon$ tree are not comparable due to the use of $\delta'$ instead of $\delta$. Therefore, in Section 6.3, we compare an implementation of our block-$\varepsilon$ tree built on $S$ with the standard block tree built on $\texttt{bv}(A)$. Our proposal turns out to be more space-efficient for some of the experimented sparse datasets and, as far as query time is concerned, it is 2.19× faster in select, and it is either faster (1.32×) or slower (1.27×) in rank than the block tree.

Compared to LA-vector, the block-$\varepsilon$ tree can take advantage of repetitions and avoid the encoding of subarrays of $A$ corresponding to repeated substrings of $S$. Furthermore, since the block-$\varepsilon$ tree allocates the most succinct encoding for a subarray of $A$ by considering the smallest $\varepsilon_q \leq \varepsilon$ giving a linear $\varepsilon_q$-approximation, it could be regarded as the repetition-aware analogous of the space-optimised LA-vector (see Section 5.5), in which all values of $\varepsilon = 0, 2^0, 2^1, \ldots, 2^{\log u}$ are considered. The block-$\varepsilon$ tree has the advantage of potentially storing fewer corrections at the cost of storing the tree topology. Using the straightforward pointer-based encoding we discussed above, the tree topology takes $\mathcal{O}(\delta \log \frac{n}{\delta} \log n)$ bits in the worst case, but in the next section we implement a more succinct pointerless encoding. We notice, nonetheless, that the more repetitive is the string $S$, the smaller is $\delta$, thus the overhead of the tree topology gets negligible.

Finally, we mention that the block-$\varepsilon$ tree could employ other compressed rank-select dictionaries in its nodes, yielding a hybrid compression approach [OTV15] that can benefit from the orchestration of bicriteria optimisation and proper pruning of its topology to achieve the best space occupancy, given a bound on the query time, or vice versa (à la [FV20b; FGM09; OTV15]).

## 6.3 Experiments

We experiment with a C++ implementation of the block-$\varepsilon$ tree, the simplest and most practical contribution in this chapter, on a machine with 202 GB of RAM and a 2.30 GHz Intel Xeon Gold 5118 CPU. The source code is available at `https://github.com/gvinciguerra/BlockEpsilonTree`.

### 6.3.1 Implementation notes

In our implementation of the block-$\varepsilon$ tree, instead of starting from a pre-determined number of blocks, we follow [Bel+21] and construct a full block tree to then remove the top levels that do not contain any unmarked block.[4] We use a pointerless representation of the tree topology via a plain bitvector for each level indicating with a `0` which block in the level is unmarked (hence, has a leftward copy) or pruned by a segment, and with a `1` which block is marked but not pruned by a segment (hence, it is an internal node). We use $\mathsf{rank}_1$ on these bitvectors to traverse the tree downwards. If we reach a marked or pruned node, we use $\mathsf{rank}_0$ on the bitvector to access two separate packed arrays[5] storing the pointers and the $\Delta$-values, respectively, associated with each unmarked or pruned block. We store the segment blocks as an array of structures, with each structure storing the slope $\alpha$, the intercept $\beta$, the exponent $y$ of the error $\varepsilon = 2^y - 1$ of the segment, and a pointer to the correction packed array $C_q$. For the visit of the block-$\varepsilon$ tree that assigns costs to its nodes, we consider the values of $y$ ranging from 0 to 15, and we slightly change the constants of the cost function $c$ defined in Section 6.2 to reflect our pointerless implementation. Marked leaf blocks containing less than a number $b$ of elements are not split further, and they are concatenated left-to-right and encoded with Lemma 4.1. Intuitively, since these blocks cannot be replaced by leftward pointers or pruned by segments they lack both repetitiveness and approximate linearity, hence a compression via Lemma 4.1 (or any other method) is likely to be more appropriate. The samples at each level needed to support $\mathsf{rank}$ on $A$ are stored in a packed array. For the predecessor query on the first-level samples, we use a binary search.

### 6.3.2 Baselines and datasets

We compare our block-$\varepsilon$ tree with the block tree of [Bel+21], built on the characteristic bitvector $\mathtt{bv}(A)$, and with the space-optimised LA-vector of Section 5.5 (referred to as `la_vector_opt` in Section 5.7). All these implementations are written in C++ and build on the `sdsl` library [Gog+14]. For both the block tree and the block-$\varepsilon$ tree, we use a branching factor of two and vary the length $b$ of the last-level blocks as $b \in \{2^3, 2^4, \ldots, 2^9\}$. We do not show the full space-time trade-off of each structure but report only the most space-efficient configurations. A comparison with other rank/select dictionaries is beyond the scope of this chapter, and it was already investigated in Section 5.7.

[4] We experimented with the theoretical proposal of starting with $\delta$ blocks. Although this makes the query time faster, it worsens the compression (up to 2.7×) as it misses the copies longer than $n/\delta$.

[5] By packed array, we mean an array with fixed-length entries sized to contain the largest element.

**Table 6.1:** The performance of LA-vector, the block tree over the characteristic bitvector $\texttt{bv}(A)$ and the block-$\varepsilon$ tree over twelve datasets of different size $n$ and universe size $u$. The select and rank columns show the average query time of the operations in nanoseconds. The space of each structure is shown in Bits Per Integer (BPI). For the block tree and the block-$\varepsilon$ tree, the value $b$ denotes the length of the last-level block that gave the most space-efficient configuration.

| Dataset | | | LA-vector | | | Block tree on $\texttt{bv}(A)$ | | | | | Block-$\varepsilon$ tree on $S$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Name ($n/u$) | $n/10^6$ | $u/10^6$ | select | rank | BPI | $b$ | select | rank | BPI | Depth | $b$ | select | rank | BPI | Depth (Avg) |
| GOV2 (76.6%) | 18.85 | 24.62 | 69 | 130 | 1.85 | 64 | 668 | 519 | **0.69** | 12 | 16 | 451 | 825 | 1.89 | 14 (9.98) |
| GOV2 (40.6%) | 9.85 | 24.62 | 60 | 129 | 3.48 | 128 | 686 | 531 | **1.56** | 11 | 256 | 367 | 638 | 3.26 | 10 (8.73) |
| GOV2 (4.1%) | 1.00 | 24.62 | 33 | 96 | 3.01 | 32 | 645 | 573 | 4.62 | 13 | 128 | 407 | 465 | **2.92** | 10 (9.73) |
| URL (5.6%) | 57.98 | 1039.92 | 124 | 144 | 2.83 | 32 | 1017 | 733 | **2.58** | 18 | 16 | 762 | 909 | 3.41 | 16 (12.94) |
| URL (1.3%) | 13.56 | 1039.91 | 98 | 123 | **6.34** | 32 | 987 | 753 | 8.57 | 18 | 32 | 463 | 664 | 7.32 | 10 (8.39) |
| URL (0.4%) | 3.73 | 1039.86 | 34 | 87 | **1.28** | 32 | 831 | 783 | 1.84 | 19 | 16 | 400 | 553 | 1.51 | 11 (7.92) |
| 5GRAM (9.8%) | 145.40 | 1476.73 | 171 | 249 | 4.40 | 32 | 1176 | 876 | **3.64** | 18 | 32 | 621 | 999 | 5.01 | 12 (10.27) |
| 5GRAM (2.0%) | 29.20 | 1476.73 | 132 | 177 | **6.37** | 32 | 1143 | 863 | 8.80 | 18 | 64 | 483 | 733 | 6.96 | 9 (7.81) |
| 5GRAM (0.8%) | 11.22 | 1476.69 | 95 | 125 | **7.56** | 32 | 1017 | 826 | 11.25 | 19 | 64 | 421 | 592 | 8.34 | 9 (7.61) |
| DNA (49.0%) | 490.10 | 1000.00 | 250 | 446 | 5.27 | 512 | 1158 | 922 | **2.09** | 14 | 512 | 535 | 1070 | 3.65 | 3 (2.98) |
| DNA (29.5%) | 294.68 | 1000.00 | 218 | 416 | 6.20 | 512 | 1227 | 989 | **3.46** | 14 | 512 | 368 | 718 | 4.57 | 2 (1.96) |
| DNA (19.6%) | 195.42 | 1000.00 | 195 | 384 | 6.69 | 512 | 1206 | 972 | 5.21 | 14 | 512 | 335 | 654 | **5.01** | 2 (1.94) |

As datasets, we use (i) three postings lists with different densities $n/u$ from the GOV2 inverted index (see Section 5.7.2); (ii) six integers lists obtained by enumerating the positions of the first, second and third most frequent character in each of the Burrows-Wheeler transform of two text files: URL and 5GRAM (as in Section 5.7.2); (iii) three integers lists obtained by enumerating, respectively, the positions of both `Ts` and `Gs`, of `Ts`, and of `Gs` in the Burrows-Wheeler transform of the first gigabyte of the human reference genome `GRCh38.p13`.

### 6.3.3 Results

For each tested structure, query operation, and dataset, we generate a batch of $10^5$ random queries and measure the average query time in nanoseconds and the space occupancy of the structure in bits per integer

Table 6.1 shows the results. First and foremost, we note that LA-vector is 10.51× faster in select and 4.69× faster in rank than the block tree on average, while for space there is no clear winner over all the datasets. This comparison, which was not known in the literature, illustrates that the combination of approximate linearity and repetitiveness is interesting not only from a theoretical point of view, as commented in the introduction, but also from a practical point of view.

Let us now compare the performance of our block-$\varepsilon$ tree against the block tree and the LA-vector. The block-$\varepsilon$ tree is 2.19× faster in select than the block tree, and it is either faster (1.32×) or slower (1.27×) in rank. With respect to LA-vector, the block-$\varepsilon$ tree is always slower. But, for what concerns the space, the block-$\varepsilon$ tree

improves both the LA-vector and the block tree in the sparsest GOV2 and DNA, and in the vast majority of the remaining datasets it is the second-best structure for space occupancy (except for the densest GOV2, URL and 5GRAM). This shows that space-wise, the block-$\varepsilon$ tree can be a robust data structure in that it often achieves a good compromise by exploiting both kinds of regularities: repetitiveness (block trees) and approximate linearity (LA-vectors).

For future work, we believe the block-$\varepsilon$ tree can be improved along at least two avenues. First, the block-$\varepsilon$ tree at a certain level is constrained to use fixed-length blocks (and thus segments), whilst the LA-vector minimises its space occupancy using segments whose start/end positions do not have to coincide with a subdivision in blocks. Removing this limitation, inherited from the block tree, would help to better capture approximate linearity and improve the space occupancy of the block-$\varepsilon$ tree. Second, the block-$\varepsilon$ tree captures the repetitiveness of the gap string $S$, while for the densest datasets of Table 6.1 it appears worthwhile to consider the repetitiveness in $\mathtt{bv}(A)$, as done by the block tree. Therefore, adapting our pruning strategy to $\mathtt{bv}(A)$ is likely to improve the space occupancy in these densest datasets (though, the space-time bounds will then depend on $u$ instead of $n$).

## 6.4 Summary

We introduced compressed rank/select dictionaries that exploit two sources of compressibility arising in real data: repetitiveness and approximate linearity. Our first result, the $\mathrm{LZ}_\varepsilon^\rho$ parsing, combines backward copies with linear $\varepsilon$-approximation thus supporting efficient queries within a space complexity bounded by the $k$th order entropy of the gaps in the input data. Our second result, the block-$\varepsilon$ tree, adapts smoothly to both sources of compressibility and offers improved query times compared to $\mathrm{LZ}_\varepsilon^\rho$. We experimented with an implementation of the block-$\varepsilon$ tree showing that it effectively exploits both repetitiveness and approximate linearity.

The results in this chapter appeared in [FMV21]. The source code of the block-$\varepsilon$ tree is publicly available at `https://github.com/gvinciguerra/BlockEpsilonTree`.

# 7 Conclusions

We conclude the thesis with a summary of the main contributions and a discussion of the open research problems.

**Chapter 3**

We studied the effectiveness of the *approximate linearity* concept, which underlies the data structures presented in the following chapters, by stating some bound on the size of the optimal piecewise linear $\varepsilon$-approximation built on a sorted input array of size $n$. Our main result is an upper bound of $\mathcal{O}(n/\varepsilon^2)$, which holds under some general assumptions on the distribution of the input data.

The results in this chapter appeared on [FLV20; FLV21], and the code to reproduce the experiments was made available at `https://github.com/gvinciguerra/Learned-indexes-effectiveness`.

The first open question regards the assumption "$\varepsilon$ is sufficiently larger than $\sigma/\mu$" of Theorem 3.3. It is natural to ask whether this condition can be waived, thus making the theorem stronger, and whether we can bound the error made by the approximation for finite and not too large values of $\varepsilon\mu/\sigma$. A second question asks to provide a formal analysis of the distribution of the segment lengths found in the optimal piecewise linear $\varepsilon$-approximation of O'Rourke [ORo81]. We know that their number grows on average as $\Omega((\mu\varepsilon/\sigma)^2)$, but how much are they longer than what this $\Omega$-bound asymptotically states? A final intriguing research question concerns the study of (piecewise) *nonlinear* $\varepsilon$-approximations, which are likely to decrease the number of "pieces" with respect to segments, at the expense of more parameters to store for a single piece. Do these nonlinear $\varepsilon$-approximations improve the $\mathcal{O}(n/\varepsilon^2)$ space bound of the linear ones?

**Chapter 4**

We designed the PGM-index, a data structure for the predecessor search problem with provably efficient I/O-bounds. In the static case, it guarantees the same optimal I/O-complexity of a B-tree while improving its space from $\mathcal{O}(n/B)$ to $\mathcal{O}(n/B^2)$, which is significant, since $B$ in practice is of the order of hundreds or thousands. We made the PGM-index compressed and query distribution aware, and we showed

experimentally that it significantly improves the space-time performance of both static and dynamic predecessor structures.

The PGM-index first appeared on [FV20b]. The corresponding software library was released at `https://pgm.di.unipi.it`, and since then it has been improved with new features, such as the capability of answering orthogonal range queries on multidimensional data. The library has become popular on GitHub (with over 550 stars) and has also appeared on several social networks and websites, such as on Hacker News. Contributors helped in debugging, implementing new functionality (e.g. approximate $k$-nearest neighbour search), and asking for new features (e.g. the integration into database management systems and porting into other programming languages).

For future work, we believe that there is room for further improvements in the performance of the Dynamic PGM-index, especially over query-heavy workloads. We also mention the design and implementation of concurrency control techniques, which would be a key step for the integration of the PGM-index into real database management systems. Finally, it is worth investigating an extension of the PGM-index in the context of variable-length string keys.

**Chapter 5**

We shone a new light on the core component of any compact data structure, namely compressed rank/select dictionaries, by proposing the LA-vector, a novel learning-based compression scheme based on piecewise linear $\varepsilon$-approximations augmented with correction values. We showed how to support efficient query operations, and we designed a piecewise linear $\varepsilon$-approximation construction algorithm that uses different $\varepsilon$ values for different chunks of the input data with the objective of minimising the overall space occupancy of the LA-vector. A comparison of LA-vector with some other well-engineered compressed rank/select dictionaries showed new space-time trade-offs.

The LA-vector appeared in [BFV21a], and its source code was made available at `https://github.com/gvinciguerra/la_vector`.

For future work, we argue that the space of LA-vector can be further improved by computing segments in such a way that the statistical redundancy of the correction values in $C$ is increased. This could be possibly achieved by jointly optimising the space occupied by the segments and the space occupied by the compressed $C$, playing on both the segments' lengths and their correction values. We also mention the use of vectorised instructions to achieve faster query times [LB15]. Finally,

we suggest an in-depth study, design and experimentation of hybrid rank/select structures, possibly integrating *nonlinear* $\varepsilon$-approximations in the LA-vector.

**Chapter 6**

We showed that the results obtained in the previous chapter, which exploit the approximate linearity in the data, can be combined with the ones that exploit the repetitiveness in the data, such as the classical Lempel-Ziv parsings [LZ76; ZL77; KN13] and the recently-proposed block trees [Bel+21]. Our first result, the $\mathrm{LZ}_\varepsilon^\rho$, is a rank/select data structure that combines backward copies with linear $\varepsilon$-approximations. It supports efficient queries within a space complexity bounded by the $k$th order entropy of the gaps in the input data. Our second result, the block-$\varepsilon$ tree, is a rank/select data structure that carefully deploys leftward copies of block trees and linear $\varepsilon$-approximations to minimise the space occupancy of the compressed representation. Our experimental achievements show that the combination of the two sources of compressibility is effective and achieves the best of both worlds.

The results in this chapter appeared in [FMV21]. The source code of the block-$\varepsilon$ tree is publicly available at `https://github.com/gvinciguerra/BlockEpsilonTree`.

For future work, we plan to implement and experiment with the $\mathrm{LZ}_\varepsilon^\rho$ parsing, possibly adapting the efficient parsing algorithms of [KK17]. It is also worth investigating the construction of the $\mathrm{LZ}_\varepsilon^\rho$ phrases and the block-$\varepsilon$ tree inside a bicriteria framework, which seeks to optimise the query time and space usage under some given constraints. Finally, a study of the relation between the known repetitiveness measures [Nav20] and the number of segments in a piecewise linear $\varepsilon$-approximation would shed some more light on their differences and interplay.

# Bibliography


[AKS15] Rachit Agarwal, Anurag Khandelwal and Ion Stoica. “Succinct: enabling queries on compressed data”. In: *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2015 (cit. on p. 59).

[Ao+11] Naiyong Ao, Fan Zhang, Di Wu, Douglas S. Stones, Gang Wang, Xiaoguang Liu, Jing Liu and Sheng Lin. “Efficient parallel lists intersection and index compression algorithms using graphics processing units”. In: *PVLDB* 4.8 (2011), pp. 470–481 (cit. on pp. 2, 6, 34).

[AV04] Lars Arge and Jan Vahrenhold. “I/O-efficient dynamic planar point location”. In: *Computational Geometry* 29.2 (2004), pp. 147–162 (cit. on p. 40).

[Arr+18] Diego Arroyuelo, Mauricio Oyarzún, Senen Gonzalez and Victor Sepulveda. “Hybrid compression of inverted lists for reordered document collections”. In: *Information Processing & Management* 54 (May 2018) (cit. on p. 61).

[AR19] Diego Arroyuelo and Rajeev Raman. “Adaptive succinctness”. In: *Proceedings of the 26th International Symposium on String Processing and Information Retrieval (SPIRE)*. 2019 (cit. on pp. 61, 78).

[AW20] Diego Arroyuelo and Manuel Weitzman. “A hybrid compressed data structure supporting rank and select on bit sequences”. In: *Proceedings of the 39th International Conference of the Chilean Computer Science Society (SCCC)*. 2020 (cit. on pp. 61, 81).

[BBG05] Amitabha Bagchi, Adam L. Buchsbaum and Michael T. Goodrich. “Biased skip lists”. In: *Algorithmica* 42.1 (2005), pp. 31–48 (cit. on p. 43).

[BN09] Jeremy Barbay and Gonzalo Navarro. “Compressed representations of permutations, and applications”. In: *Proceedings of the 26th International Symposium on Theoretical Aspects of Computer Science (STACS)*. 2009 (cit. on p. 59).

[Bel16] Djamal Belazzougui. “Predecessor search, string algorithms and data structures”. In: *Encyclopedia of Algorithms*. Ed. by Ming-Yang Kao. 2016, pp. 1605–1611 (cit. on p. 33).

[Bel+08] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh and Sebastiano Vigna. “Theory and practice of monotone minimal perfect hashing”. In: *ACM Journal of Experimental Algorithmics* 16 (2008) (cit. on p. 59).

[Bel+21] Djamal Belazzougui, Manuel Cáceres, Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Gonzalo Navarro, Alberto Ordóñez, Simon J. Puglisi and Yasuo Tabei. “Block trees”. In: *Journal of Computer and System Sciences* 117 (2021), pp. 1–22 (cit. on pp. 4, 92, 100, 107, 113).

[Bel+15] Djamal Belazzougui, Patrick Hagge Cording, Simon J. Puglisi and Yasuo Tabei. "Access, rank, and select in grammar-compressed strings". In: *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA)*. 2015, pp. 142–154 (cit. on p. 92).

[BN15] Djamal Belazzougui and Gonzalo Navarro. "Optimal lower and upper bounds for representing sequences". In: *ACM Transactions on Algorithms* 11.4 (2015) (cit. on pp. 39, 105).

[BST85] Samuel W. Bent, Daniel Dominic Sleator and Robert Endre Tarjan. "Biased search trees". In: *SIAM Journal on Computing* 14.3 (1985), pp. 545–568 (cit. on p. 43).

[BS80] Jon Louis Bentley and James B. Saxe. "Decomposable searching problems I. Static-to-dynamic transformation". In: *Journal of Algorithms* 1.4 (1980), pp. 301–358 (cit. on p. 40).

[Bin18] Timo Bingmann. *TLX: collection of sophisticated C++ data structures, algorithms, and miscellaneous helpers*. `https://panthema.net/tlx`, retrieved on July 1, 2021. 2018 (cit. on p. 50).

[BFV21a] Antonio Boffa, Paolo Ferragina and Giorgio Vinciguerra. "A "learned" approach to quicken and compress rank/select dictionaries". In: *Proceedings of the 23rd SIAM Symposium on Algorithm Engineering and Experiments (ALENEX)*. 2021, pp. 46–59 (cit. on pp. 3, 60, 90, 112).

[BFV21b] Antonio Boffa, Paolo Ferragina and Giorgio Vinciguerra. "A learned approach to design compressed rank/select data structures". Submitted to ACM Transactions on Algorithms. 2021 (cit. on pp. 3, 60).

[BFG03] Adam L. Buchsbaum, Glenn S. Fowler and Raffaele Giancarlo. "Improving table compression with combinatorial optimization". In: *Journal of the ACM* 50.6 (Nov. 2003), pp. 825–851 (cit. on p. 74).

[Cla96] David Clark. "Compact PAT trees". PhD thesis. University of Waterloo, Canada, 1996 (cit. on p. 60).

[CN08] Francisco Claude and Gonzalo Navarro. "Practical rank/select queries over arbitrary sequences". In: *Proceedings of the 15th International Symposium on String Processing and Information Retrieval (SPIRE)*. 2008 (cit. on p. 81).

[Cor+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms, Third Edition*. 3rd ed. The MIT Press, 2009 (cit. on pp. 33, 34, 74, 76).

[CT06] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. 2nd ed. Wiley, 2006 (cit. on pp. 2, 66).

[Dau21] Justinas V. Daugmaudis. *`art_map: std::map/std::set` implementation using the Adaptive Radix Tree*. `https://github.com/justinasvd/art_map`, retrieved on July 1, 2021. 2021 (cit. on p. 50).

[DAI18] Niv Dayan, Manos Athanassoulis and Stratos Idreos. “Optimal Bloom filters and adaptive merging for LSM-trees”. In: *ACM Transactions on Database Systems* 43.4 (Dec. 2018), 16:1–16:48 (cit. on p. 40).

[Din+20] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet and Tim Kraska. “ALEX: an updatable adaptive learned index”. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2020, pp. 969–984 (cit. on pp. 34, 50).

[DFH21] Patrick Dinklage, Johannes Fischer and Alexander Herlez. “Engineering predecessor data structures for dynamic integer sets”. In: *Proceedings of the 19th International Symposium on Experimental Algorithms (SEA)*. 2021, 7:1–7:19 (cit. on pp. 50, 51).

[Eli74] Peter Elias. “Efficient storage and retrieval by content and address of static files”. In: *Journal of the ACM* 21.2 (1974), pp. 246–260 (cit. on pp. 41, 61).

[EMK97] Paul Embrechts, Thomas Mikosch and Claudia Klüppelberg. *Modelling Extremal Events: For Insurance and Finance*. Springer-Verlag, 1997 (cit. on p. 21).

[Fan71] Robert Mario Fano. *On the number of bits required to implement an associative memory. Memo 61*. Massachusetts Institute of Technology, Project MAC, 1971 (cit. on pp. 41, 61).

[Fel70] William Feller. *An introduction to probability theory and its applications*. Vol. 2. John Wiley & Sons, 1970 (cit. on pp. 24, 27).

[FGM12] Paolo Ferragina, Travis Gagie and Giovanni Manzini. “Lightweight data indexing and compression in external memory”. In: *Algorithmica* 63.3 (2012), pp. 707–730 (cit. on p. 82).

[FGM09] Paolo Ferragina, Raffaele Giancarlo and Giovanni Manzini. “The myriad virtues of wavelet trees”. In: *Information and Computation* 207.8 (2009), pp. 849–866 (cit. on p. 106).

[Fer+18] Paolo Ferragina, Stefan Kurtz, Stefano Lonardi and Giovanni Manzini. “Computational biology”. In: *Handbook of Data Structures and Applications*. Ed. by Dinesh P. Mehta and Sartaj Sahni. 2nd ed. CRC Press, 2018. Chap. 59 (cit. on p. 59).

[FLV21] Paolo Ferragina, Fabrizio Lillo and Giorgio Vinciguerra. “On the performance of learned data structures”. In: *Theoretical Computer Science* 871 (2021), pp. 107–120 (cit. on pp. 3, 14, 31, 111).

[FLV20] Paolo Ferragina, Fabrizio Lillo and Giorgio Vinciguerra. “Why are learned indexes so effective?” In: *Proceedings of the 37th International Conference on Machine Learning (ICML)*. vol. 119. 2020, pp. 3123–3132 (cit. on pp. 3, 14, 31, 111).

[FM05] Paolo Ferragina and Giovanni Manzini. “Indexing compressed text”. In: *Journal of the ACM* 52.4 (2005), pp. 552–581 (cit. on p. 59).

[Fer+04] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen and Gonzalo Navarro. "An alphabet-friendly FM-index". In: *Proceedings of the 11th International Conference on String Processing and Information Retrieval (SPIRE)*. 2004, pp. 150–160 (cit. on p. 66).

[FMV21] Paolo Ferragina, Giovanni Manzini and Giorgio Vinciguerra. "Repetition- and linearity-aware rank/select dictionaries". In: *Proceedings of the 32nd International Symposium on Algorithms and Computation (ISAAC)*. 2021 (cit. on pp. 4, 92, 109, 113).

[FNV11] Paolo Ferragina, Igor Nitto and Rossano Venturini. "On optimally partitioning a text to improve its compression". In: *Algorithmica* 61.1 (2011), pp. 51–74 (cit. on pp. 74, 75, 79).

[FV07] Paolo Ferragina and Rossano Venturini. "A simple storage scheme for strings achieving entropy bounds". In: *Theoretical Computer Science* 372.1 (2007), pp. 115–121 (cit. on pp. 66, 100).

[FV20a] Paolo Ferragina and Giorgio Vinciguerra. "Learned data structures". In: *Recent Trends in Learning From Data*. Ed. by Luca Oneto, Nicolò Navarin, Alessandro Sperduti and Davide Anguita. Springer, 2020, pp. 5–41 (cit. on p. 34).

[FV20b] Paolo Ferragina and Giorgio Vinciguerra. "The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds". In: *PVLDB* 13.8 (2020), pp. 1162–1175 (cit. on pp. 3, 34, 48, 57, 106, 112).

[Fos+06] Luca Foschini, Roberto Grossi, Ankur Gupta and Jeffrey Scott Vitter. "When indexing equals compression: experiments with compressing suffix arrays and applications". In: *ACM Transactions on Algorithms* 2.4 (Oct. 2006), pp. 611–639 (cit. on p. 61).

[Fri+12] Matteo Frigo, Charles E. Leiserson, Harald Prokop and Sridhar Ramachandran. "Cache-oblivious algorithms". In: *ACM Transactions on Algorithms* 8.1 (Jan. 2012) (cit. on p. 1).

[GNP20] Travis Gagie, Gonzalo Navarro and Nicola Prezza. "Fully functional suffix trees and optimal text searching in BWT-runs bounded space". In: *Journal of the ACM* 67.1 (2020) (cit. on p. 59).

[Gal+19] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca and Tim Kraska. "FITing-tree: a data-aware index structure". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2019, pp. 1189–1206 (cit. on pp. 34, 39, 48).

[Gar85] Crispin W. Gardiner. *Handbook of stochastic methods for physics, chemistry and the natural sciences*. 2nd ed. Springer-Verlag, 1985 (cit. on pp. 19, 20).

[Gog+14] Simon Gog, Timo Beller, Alistair Moffat and Matthias Petri. "From theory to practice: plug and play with succinct data structures". In: *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA)*. 2014, pp. 326–337 (cit. on pp. 80, 107).

[Gog+19] Simon Gog, Juha Kärkkäinen, Dominik Kempa, Matthias Petri and Simon J. Puglisi. “Fixed block compression boosting in FM-indexes: theory and practice”. In: *Algorithmica* 81.4 (2019), pp. 1370–1391 (cit. on p. 59).

[GP14] Simon Gog and Matthias Petri. “Optimized succinct data structures for massive data”. In: *Software: Practice and Experience* 44.11 (2014), pp. 1287–1314 (cit. on pp. 60, 67).

[Gol07] Alexander Golynski. “Optimal lower bounds for rank and select indexes”. In: *Theoretical Computer Science* 387.3 (2007), pp. 348–359 (cit. on p. 60).

[Gol+14] Alexander Golynski, Alessio Orlandi, Rajeev Raman and S. Srinivasa Rao. “Optimal indexes for sparse bit vectors”. In: *Algorithmica* 69.4 (2014), pp. 906–924 (cit. on p. 60).

[Gon+05] Rodrigo González, Szymon Grabowski, Veli Mäkinen and Gonzalo Navarro. “Practical implementation of rank and select queries”. In: *Poster Proceedings of the 4th Workshop on Efficient and Experimental Algorithms (WEA)*. 2005, pp. 27–38 (cit. on p. 60).

[GBC16] Ian Goodfellow, Yoshua Bengio and Aaron Courville. *Deep Learning*. MIT Press, 2016 (cit. on p. 9).

[Goo17] Google. *Abseil Common Libraries (C++)*. `https://github.com/abseil/abseil-cpp`, retrieved on June 10, 2021. 2017 (cit. on p. 50).

[Gra06] Goetz Graefe. “B-tree indexes, interpolation search, and skew”. In: *Proceedings of the 2nd International Workshop on Data Management on New Hardware (DaMoN)*. 2006 (cit. on p. 46).

[GGV03] Roberto Grossi, Ankur Gupta and Jeffrey Scott Vitter. “High-order entropy-compressed text indexes”. In: *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2003, pp. 841–850 (cit. on p. 66).

[GV05] Roberto Grossi and Jeffrey Scott Vitter. “Compressed suffix arrays and suffix trees with applications to text indexing and string matching”. In: *SIAM Journal on Computing* 35.2 (2005), pp. 378–407 (cit. on p. 59).

[Gup+07] Ankur Gupta, Wing-Kai Hon, Rahul Shah and Jeffrey Scott Vitter. “Compressed data structures: dictionaries and data-aware measures”. In: *Theoretical Computer Science* 387.3 (2007), pp. 313–331 (cit. on p. 61).

[Hag98] Torben Hagerup. “Sorting and searching on the word RAM”. in: *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*. 1998, pp. 366–398 (cit. on p. 5).

[HP19] John L. Hennessy and David A. Patterson. “A new golden age for computer architecture”. In: *Communications of the ACM* 62.2 (2019), pp. 48–60 (cit. on p. 59).

[IC20] Stratos Idreos and Mark Callaghan. “Key-value storage engines”. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2020, pp. 2667–2672 (cit. on pp. 34, 40).

[Idr+19] Stratos Idreos, Kostas Zoumpatianos, Subarna Chatterjee, Wilson Qin, Abdul Wasay, Brian Hentschel, Mike Kester, Niv Dayan, Demi Guo, Minseo Kang and Yiyou Sun. "Learning data structure alchemy". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 42.2 (2019), pp. 46–57 (cit. on p. 45).

[Idr+18] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester and Demi Guo. "The Data Calculator: data structure design and cost synthesis from first principles and learned cost models". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2018, pp. 535–550 (cit. on p. 45).

[KK17] Dominik Kempa and Dmitry Kosolobov. "LZ-End parsing in linear time". In: *Proceedings of the 25th Annual European Symposium on Algorithms (ESA 2017)*. 2017, 53:1–53:14 (cit. on p. 113).

[Kip+20] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska and Thomas Neumann. "RadixSpline: a single-pass learned index". In: *Proceedings of the International Workshop on Exploiting Artificial Intelligence Techniques for Data Management at SIGMOD*. 2020 (cit. on p. 48).

[KNP20] Tomasz Kociumaka, Gonzalo Navarro and Nicola Prezza. "Towards a definitive measure of repetitiveness". In: *Proceedings of the 14th Latin American Symposium on Theoretical Informatics (LATIN)*. 2020 (cit. on p. 92).

[KRT20] Evgenios M. Kornaropoulos, Silei Ren and Roberto Tamassia. *The Price of tailoring the index to your data: poisoning attacks on learned index structures*. 2020. arXiv: `2008.00297 [cs.CR]` (cit. on p. 50).

[KM99] Sambasiva Rao Kosaraju and Giovanni Manzini. "Compression of low entropy strings with Lempel-Ziv algorithms". In: *SIAM Journal on Computing* 29.3 (1999), pp. 893–911 (cit. on p. 66).

[Kra+18] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean and Neoklis Polyzotis. "The case for learned index structures". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2018, pp. 489–504 (cit. on pp. 34, 39, 48, 50).

[KN13] Sebastian Kreft and Gonzalo Navarro. "On compressing and indexing repetitive sequences". In: *Theoretical Computer Science* 483 (2013), pp. 115–133 (cit. on pp. 4, 93, 98, 113).

[LKN13] Viktor Leis, Alfons Kemper and Thomas Neumann. "The adaptive radix tree: ARTful indexing for main-memory databases". In: *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE)*. 2013, pp. 38–49 (cit. on p. 50).

[LB15] Daniel Lemire and Leonid Boytsov. "Decoding billions of integers per second through vectorization". In: *Software: Practice and Experience* 45.1 (2015), pp. 1–29 (cit. on p. 112).

[LZ76] Abraham Lempel and Jacob Ziv. "On the complexity of finite sequences". In: *IEEE Transactions on Information Theory* 22.1 (1976), pp. 75–81 (cit. on pp. 4, 92, 113).

[LC20] Chen Luo and Michael J. Carey. “LSM-based storage techniques: a survey”. In: *The VLDB Journal* 29.1 (2020), pp. 393–418 (cit. on pp. 35, 40).

[Mäk+15] Veli Mäkinen, Djamal Belazzougui, Fabio Cunial and Alexandru I. Tomescu. *Genome-Scale Algorithm Design*. Cambridge University Press, 2015 (cit. on p. 59).

[MN07] Veli Mäkinen and Gonzalo Navarro. “Rank and select revisited and extended”. In: *Theoretical Computer Science* 387.3 (2007), pp. 332–347 (cit. on pp. 59, 61).

[Mäk+10] Veli Mäkinen, Gonzalo Navarro, Jouni Sirén and Niko Välimäki. “Storage and retrieval of highly repetitive sequence collections”. In: *Journal of Computational Biology* 17.3 (2010), pp. 281–308 (cit. on p. 81).

[MD21] Marcel Maltry and Jens Dittrich. *A critical analysis of recursive model indexes*. 2021. arXiv: `2106.16166 [cs.DB]` (cit. on p. 48).

[MVF21] Giovanni Manzini, Giorgio Vinciguerra and Paolo Ferragina. “Procedimento di compressione e ricerca su un insieme di dati basato su strategie multiple”. (Italy). May 2021. Patent application filed.

[Mar+20] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann and Tim Kraska. “Benchmarking learned indexes”. In: *PVLDB* 14.1 (Sept. 2020), pp. 1–13 (cit. on pp. 27, 47–50).

[MMP05] Jaume Masoliver, Miquel Montero and Josep Perelló. “Extreme times in financial markets”. In: *Physical Review E* 71 (5 2005), p. 056130 (cit. on p. 18).

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge University Press, 1995 (cit. on p. 17).

[Mun96] J. Ian Munro. “Tables”. In: *Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. 1996 (cit. on p. 60).

[MR97] J. Ian Munro and Venkatesh Raman. “Succinct representation of balanced parentheses, static trees and planar graphs”. In: *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS)*. 1997 (cit. on p. 59).

[Nav16] Gonzalo Navarro. *Compact data structures: a practical approach*. Cambridge University Press, 2016 (cit. on pp. 1, 41, 56, 59, 61, 70).

[Nav20] Gonzalo Navarro. “Indexing highly repetitive string collections, part I: repetitiveness measures”. In: *ACM Computing Surveys* 54.2 (2020) (cit. on pp. 2, 91, 92, 113).

[Nav14] Gonzalo Navarro. “Spaces, trees, and colors: the algorithmic landscape of document retrieval on sequences”. In: *ACM Computing Surveys* 46.4 (2014) (cit. on p. 59).

[NM07] Gonzalo Navarro and Veli Mäkinen. “Compressed full-text indexes”. In: *ACM Computing Surveys* 39.1 (2007) (cit. on pp. 59, 66).

[NP12] Gonzalo Navarro and Eliana Providel. “Fast, small, simple rank/select on bitmaps”. In: *Proceedings of the 11th International Symposium on Experimental Algorithms (SEA)*. 2012, pp. 295–306 (cit. on p. 60).

[NR20] Gonzalo Navarro and Javiel Rojas-Ledesma. “Predecessor search”. In: *ACM Computing Surveys* 53.5 (2020) (cit. on pp. 33, 39, 61, 105).

[ONe+96] Patrick O’Neil, Edward Cheng, Dieter Gawlick and Elizabeth O’Neil. “The log-structured merge-tree (LSM-tree)”. In: *Acta Informatica* 33.4 (1996), pp. 351–385 (cit. on p. 40).

[ORo81] Joseph O’Rourke. “An on-line algorithm for fitting straight lines between data ranges”. In: *Communications of the ACM* 24.9 (1981), pp. 574–578 (cit. on pp. 2, 10, 11, 39, 65, 77, 111).

[OS07] Daisuke Okanohara and Kunihiko Sadakane. “Practical entropy-compressed rank/select dictionary”. In: *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2007 (cit. on pp. 60, 61, 80).

[OTV15] Giuseppe Ottaviano, Nicola Tonellotto and Rossano Venturini. “Optimal space-time tradeoffs for inverted indexes”. In: *Proceedings of the 8th ACM International Conference on Web Search and Data Mining (WSDM)*. 2015 (cit. on p. 106).

[OV14] Giuseppe Ottaviano and Rossano Venturini. “Partitioned Elias-Fano indexes”. In: *Proceedings of the 37th International ACM Conference on Research & Development in Information Retrieval (SIGIR)*. 2014, pp. 273–282 (cit. on pp. 61, 74, 75, 78, 79, 81, 82, 88).

[Ove83] Mark H. Overmars. *The Design of Dynamic Data Structures*. Vol. 156. Lecture Notes in Computer Science. Springer, 1983 (cit. on p. 40).

[Păt16] Mihai Pătraşcu. “Predecessor search”. In: *Encyclopedia of Algorithms*. Ed. by Ming-Yang Kao. Springer, 2016, pp. 1601–1605 (cit. on p. 33).

[Păt08] Mihai Pătraşcu. “Succincter”. In: *Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 2008 (cit. on p. 61).

[PT14] Mihai Pătraşcu and Mikkel Thorup. “Dynamic integer sets with optimal rank, select, and predecessor search”. In: *Proceedings of the 55th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*. 2014, pp. 166–175 (cit. on p. 34).

[PT06] Mihai Pătraşcu and Mikkel Thorup. “Time-space trade-offs for predecessor search”. In: *Proceedings of the 38th Annual ACM Symposium on Theory of Computing (STOC)*. 2006 (cit. on pp. 34, 38, 61, 72).

[PV10] Mihai Pătraşcu and Emanuele Viola. “Cell-probe lower bounds for succinct partial sums”. In: *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2010 (cit. on pp. 60, 61).

[PH20] David A. Patterson and John L. Hennessy. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Morgan Kaufmann, 2020 (cit. on pp. 5, 9).

[Pet18] Alex Petrov. “Algorithms behind modern storage systems”. In: *Communications of the ACM* 61.8 (2018), pp. 38–44 (cit. on p. 34).

[PV17] Giulio Ermanno Pibiri and Rossano Venturini. “Clustered Elias-Fano indexes”. In: *ACM Transactions on Information Systems* 36.1 (Apr. 2017) (cit. on p. 61).

[Ram18] Rajeev Raman. “In-memory representations of databases via succinct data structures”. In: *Proceedings of the 37th ACM Symposium on Principles of Database Systems (PODS)*. 2018, pp. 323–324 (cit. on p. 59).

[Ram16] Rajeev Raman. “Rank and select operations on bit strings”. In: *Encyclopedia of Algorithms*. Ed. by Ming-Yang Kao. 2nd ed. Springer, 2016, pp. 1772–1775 (cit. on p. 60).

[RRS07] Rajeev Raman, Venkatesh Raman and Srinivasa Rao Satti. “Succinct indexable dictionaries with applications to encoding $k$-ary trees, prefix sums and multisets”. In: *ACM Transactions on Algorithms* 3.4 (2007) (cit. on pp. 59, 61, 66, 81).

[RR99] Jun Rao and Kenneth A. Ross. “Cache conscious indexing for decision-support in main memory”. In: *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*. 1999, pp. 78–89 (cit. on p. 48).

[Ras+13] Sofya Raskhodnikova, Dana Ron, Ronitt Rubinfeld and Adam Smith. “Sublinear algorithms for approximating string compressibility”. In: *Algorithmica* 65.3 (2013), pp. 685–709 (cit. on p. 92).

[Red01] Sidney Redner. *A guide to first-passage processes*. Cambridge University Press, 2001 (cit. on p. 18).

[SG06] Kunihiko Sadakane and Roberto Grossi. “Squeezing succinct data structures into entropy bounds”. In: *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2006, pp. 1230–1239 (cit. on pp. 61, 100).

[SA96] Raimund Seidel and Cecilia R. Aragon. “Randomized search trees”. In: *Algorithmica* 16.4/5 (1996), pp. 464–497 (cit. on p. 43).

[SV10] Fabrizio Silvestri and Rossano Venturini. “VSEncoding: efficient coding and fast decoding of integer lists via dynamic programming”. In: *Proceedings of the 19th ACM International Conference on Information and Knowledge Management (CIKM)*. 2010, pp. 1219–1228 (cit. on pp. 74, 78).

[TH81] Hermann Tropf and Helmut Herzog. “Multidimensional range search in dynamically balanced trees”. In: *Angewandte Informatik (Applied Informatics)* 23.2 (1981), pp. 71–77 (cit. on p. 56).

[vEB77] Peter van Emde Boas. “Preserving order in a forest in less than logarithmic time and linear space”. In: *Information Processing Letters* 6.3 (1977), pp. 80–82 (cit. on p. 33).

[Vig08] Sebastiano Vigna. “Broadword implementation of rank/select queries”. In: *Proceedings of the 7th International Workshop on Experimental Algorithms (WEA)*. 2008, pp. 154–168 (cit. on pp. 60, 61).

[Vig13] Sebastiano Vigna. “Quasi-succinct indices”. In: *Proceedings of the 6th ACM International Conference on Web Search and Data Mining (WSDM)*. 2013, pp. 83–92 (cit. on p. 61).

[Vit01] Jeffrey Scott Vitter. “External memory algorithms and data structures: dealing with massive data”. In: *ACM Computing Surveys* 33.2 (2001), pp. 209–271 (cit. on pp. 1, 34).

[Wil83] Dan E. Willard. “Log-logarithmic worst-case range queries are possible in space $\Theta(N)$”. In: *Information Processing Letters* 17.2 (1983), pp. 81–84 (cit. on p. 33).

[WMB99] Ian H. Witten, Alistair Moffat and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. 2nd ed. Morgan Kaufmann Publishers Inc., 1999 (cit. on pp. 43, 61, 74).

[Xie+14] Qing Xie, Chaoyi Pang, Xiaofang Zhou, Xiangliang Zhang and Ke Deng. “Maximum error-bounded piecewise linear representation for online stream approximation”. In: *The VLDB Journal* 23.6 (2014), pp. 915–937 (cit. on p. 65).

[Yu19] Huacheng Yu. “Optimal succinct rank data structure via approximate nonnegative tensor decomposition”. In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC)*. 2019, pp. 955–966 (cit. on p. 60).

[ZL77] Jacob Ziv and Abraham Lempel. “A universal algorithm for sequential data compression”. In: *IEEE Transactions on Information Theory* 23.3 (1977), pp. 337–343 (cit. on pp. 4, 92, 113).