UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

COMPACT DATA STRUCTURES FOR INFORMATION RETRIEVAL ON NATURAL LANGUAGE

TESIS PARA OPTAR AL GRADO DE DOCTOR EN CIENCIAS,
MENCIÓN COMPUTACIÓN

ROBERTO DANIEL KONOW KRAUSE

PROFESOR GUÍA:
GONZALO NAVARRO BADINO

MIEMBROS DE LA COMISIÓN:
JORGE PÉREZ ROJAS
BÁRBARA POBLETE LABRA
ALISTAIR MOFFAT

Este trabajo ha sido parcialmente financiado por Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F, Fondecyt Grant 1-140796, Basal Center for Biotechnology and Bioengineering (CeBiB) y Beca de Doctorado Nacional Conicyt

SANTIAGO DE CHILE
2016

Resumen

El principal objetivo de los sistemas de recuperación de información (SRI) es encontrar, lo más rápido posible, la mejor respuesta para una consulta de un usuario. Esta no es una tarea simple: la cantidad de información que los SRI manejan es típicamente demasiado grande como para permitir búsquedas secuenciales, por lo que es necesario la construcción de índices. Sin embargo, la memoria es un recurso limitado, por lo que estos deben ser eficientes en espacio y al mismo tiempo rápidos para lidiar con las demandas de eficiencia y calidad. La tarea de diseñar e implementar un índice que otorgue un buen compromiso en velocidad y espacio es desafiante tanto del punto de vista teórico como práctico. En esta tesis nos enfocamos en el uso, diseño e implementación de estructuras de datos compactas para crear nuevos índices que sean más rápidos y consuman menos espacio, pensando en ser utilizados en SRI sobre lenguaje natural.

Nuestra primera contribución es una nueva estructura de datos que compite con el índice invertido, que es la estructura clásica usada en SRIs por más de 40 años. Nuestra nueva estructura, llamada *Treaps Invertidos*, requiere espacio similar a las mejores alternativas en el estado del arte, pero es un orden de magnitud más rápido en varias consultas de interés, especialmente cuando se recuperan unos pocos cientos de documentos. Además presentamos una versión incremental que permite actualizar el índice a medida que se van agregando nuevos documentos a la colección. También presentamos la implementación de una idea teórica introducida por Navarro y Puglisi, llamada *Dual-Sorted*, implementando operaciones complejas en estructuras de datos compactas.

En un caso más general, los SRI permiten indexar y buscar en colecciones formadas por secuencias de símbolos, no solamente palabras. En este escenario, Navarro y Nekrich presentaron una solución que es óptima en tiempo, que requiere de espacio lineal y es capaz de recuperar los mejores k documentos de una colección. Sin embargo, esta solución teórica requiere más de 80 veces el tamaño de la colección, haciéndola poco atractiva en la práctica. En esta tesis implementamos un índice que sigue las ideas de la solución óptima. Diseñamos e implementamos nuevas estructuras de datos compactas y las ensamblamos para construir un índice que es órdenes de magnitud más rápido que las alternativas existentes y es competitivo en términos de espacio. Además, mostramos que nuestra implementación puede ser adaptada fácilmente para soportar colecciones de texto que contengan lenguaje natural, en cuyo caso el índice es más poderoso que los índices invertidos para contestar consultas de frases.

Finalmente, mostramos cómo las estructuras de datos, algoritmos y técnicas desarrolladas en esta tesis pueden ser extendidas a otros escenarios que son importantes para los SRI. En este sentido, presentamos una técnica que realiza agregación de información de forma eficiente en grillas bidimensionales, una representación eficiente de registros de accesos a sitios web que permite realizar operaciones necesarias para minería de datos, y un nuevo índice que mejora las herramientas existentes para representar colecciones de trazas de paquetes de red.

Abstract

The ultimate goal of modern Information Retrieval (IR) systems is to retrieve, as quickly as possible, the best possible answer given a user's query. This is not a simple task: the amount of data that IR systems handle is typically too large to admit sequential query processing, so IR systems need to build indexes to speed up queries. However, memory resources are limited, so indexes need to be space-efficient and, at the same time, fast enough to cope with current efficiency and quality standards. The task of designing and implementing an index that provides good trade-offs in terms of space and time is challenging from both a theoretical and a practical point of view. In this thesis we focus on the use, design and implementation of compact data structures to assemble new faster and smaller indexes for information retrieval on natural language.

We start by introducing a new data structure that competes with the inverted index, the structure that has been used for 40 years to implement IR system. We call this new index *Inverted Treaps* and show that it uses similar space as state-of-the-art alternatives, but it is up to an order of magnitude faster at queries, particularly when less than a few hundred documents are returned. In addition, we introduce an incremental version that supports online indexing of new documents as they are added to the collection. We also present the implementation of a theoretical idea introduced by Navarro and Puglisi, dubbed *Dual-Sorted*, by engineering complex query processing algorithms on compact data structures.

In a more general scenario, IR systems are required to handle collections containing arbitrary sequences of symbols instead of just words. On general string collections Navarro and Nekrich introduced a time-optimal and linear-space solution for retrieving the top-k best documents from a string collection given a pattern. However, in practice, the theoretical solution requires more than 80 times the size of the collection, making it unpractical. In this thesis, we implement an index following the ideas of the optimal solution. We design and engineer new compact data structures and assemble them to construct a compact index that is orders of magnitude faster than state-of-the-art alternatives and is competitive in terms of space. We also show that our implementation can be easily adapted to support text collections of natural language, in which case the index is much more powerful than inverted indexes to handle phrase queries.

Finally, we show how the data structures, algorithms and techniques developed in this thesis can be extended to other scenarios that are important for IR. We present a technique to perform efficient aggregated queries on two-dimensional grids. We also present a space-efficient representation of web access logs, which supports fast operations required by web usage mining processes. Finally, we introduce a new space-efficient index that performs fast query processing on network packet traces and outperforms similar existing networking tools.

Dedicated to my grandmother
Hayra Díaz (1917-2016)

Acknowledgements

Every PhD student usually thanks his advisor for his academic talent, patience and knowledge. What most students *don't do*, is to thank their advisors for what they did *besides* the academic aspects. I would like thank Gonzalo for everything else that is not reflected in this thesis: his friendship, the humongous amount of time he spent listening to my personal matters and giving me advices, for the amazing and crazy black-sheep email conversations at 4 AM, for letting me to try getting him drunk in so many occasions, but utterly failed, and of course, for the best barbecues ever. I had a blast doing the work presented in this document and none of that fun would have been possible without him. I will be always grateful for these amazing years.

I am very grateful to have such amazing friends like Luciano Ahumada and Francisco Claude. It would have been impossible for me to even consider doing a PhD without the help and support from Luciano, who has helped me in every possible way. Francisco, his friendship, talent and patience over the last 13 years never end to amaze me, thank you both for all the support and help you gave during this process and others.

To Nelson Baloian, for teaching me what research meant, for being one of the first ones in believing in me, for his friendship and advices over the years, and of course, gambate kudazai.

I would like to thank the committee, Jorge Peréz, Bárbara Poblete and Alistair Moffat for their time and dedication in reviewing my thesis. I would also like to thank Simon Gog, Diego Seco, Guillermo de Bernardo and Yung-Hwan Kim for their awesome work and help in various papers where we worked together.

Big thanks to my friends: Jorge Lanzarotti, Cristián Valdés, Cristián Tala, Marcelo Frías, Philippe Pavez, Fernanda Palacios, Christopher Espinoza, Giorgio Botto, Pablo Cristi, Waldemar Vildoso, Benjamín Sepúlveda, Paulina Jara, Rodrigo Mellado, Ximena Geoffroy, Jonathan Frez, Felipe Stein, Cristián Vera, Domenico Tavano, Valentina Concha, Támara Avil'es, Carolina Claude, Ingrid Faust, Francisco Claude (father), OMA, Carolina Bascuñan, Luis Loyola, Leandro Llanza, Rodrigo Canóvas, Alberto "Negro" Ordoñez, Susana Ladra and Simon Gog, for all the fun, alcohol and good times we shared together :).

To the most important people in my life, my family: Werner Konow for all his support and teaching me what the word effort really means, I would not ever achieve anything without his example of hard work and patience. Sylvia Krause for providing me all her love and teaching me the desire of learning every day something new about life with her example of patience and kindness. Andrés Konow for teaching me the passion of Computer Science, and for always believing in me. Carolina Konow for such great advices, friendship and help with everything I can imagine. Fabiola Konow for teaching me the luck of being healthy, to appreciate simple things of life and special thanks for providing me with rock solid patience.

And finally, and most importantly, to my wife, Nadia Eger, who had to struggle with me at every step in this crazy, yet incredibly fun process. Without her it would have been impossible for me to be alive after all the nights without sleep while working on this. You deserve credit for every single page of this document. Thanks for being with me in the most darkest and hardest times. I (still) cannot believe how lucky I am to be able to share my life with such a beautiful person like you. Thank you for loving me even after everything this thesis might have caused <3.

Contents

List of Tables

List of Figures

List of Algorithms

Part I

What, Why and How

Chapter 1

Introduction

We are facing an information overload problem [6]. The huge amount of information generated by users on the Internet is making the task of storing, searching and retrieving relevant information a big challenge for computer scientists and engineers. Modern Information Retrieval (IR) systems [40, 50, 17] have to deal with this humongous amount of data while serving millions of queries per day on billions of Web pages. The most prominent challenges when building IR systems are (1) quality, (2) time, and (3) space. The quality problem boils down to defining a suitable score scheme that returns the best possible answer to a user's query. Score functions range from very simple (such as "term frequency", the number of times the query appears in the document) to very sophisticated ones. In many cases, a simple score function is used to filter a few hundred candidate documents and more sophisticated rankings are then computed on those [40, 105]. This is because of the second concern, time. Text collections are usually too large to admit a sequential query processing. Indexes are data structures built on the text to speed up queries, and this is connected with the third challenge: space. Designing and implementing an index that provides a good trade-off in terms of space and time is a challenging problem from both a theoretical and a practical point of view.

Compact Data Structures is a Computer Science research field motivated by the need to handle increasing amounts of data in main memory while taking advantage of the large performance gap between the different levels of the memory hierarchy. It has achieved important theoretical and practical breakthroughs, such as representing trees of n nodes using just $2n + o(n)$ bits while supporting powerful constant-time navigation [14], representing texts within their high-order entropy space while supporting fast extraction of arbitrary segments and pattern searches [118, 66] or representing Web graphs within about 1 bit per edge [29], just to name a few. An area that has surprisingly remained outside of this recent virtuous circle of achievements is IR. IR is precisely where most of the early research on compressed text and index representations focused. The combination of text and index compression is a classical topic treated in IR textbooks over the past decade [17, 166]. It might be argued that IR has not been included in recent research because space efficiency is a closed problem in this area, but this is not the case.

The *inverted index* is the central data structure in every IR system. No matter if the index is stored on disk or in main memory, reducing its size is crucial. On disk, it reduces transfer time. In main memory, it increases the size of the collections that can be managed within a given RAM budget, or alternatively reduces the number of servers that must be allocated in a cluster to hold the index, the energy they consume, and the amount of communication in the network. Compression of inverted indexes is possibly the oldest and most successful application of compressed data structures (e.g., see Witten et al. [166]).

Inverted indexes have been designed for so-called Natural Language scenarios, that is, where queryable terms are predefined and are not too numerous as compared to the size of the collection. For example, they work very well on collections where the documents can be easily tokenized into "words", and queries are also formed by words. This scenario is common for documents written in most Western languages. However, inverted indexes are not suitable for documents where words are difficult to segment, such as documents written in East Asian languages, or in agglutinating languages such as Finnish and German, where one needs to search for particles inside the words. There are also sequence collections with no concept of words such as software source repositories, MIDI streams, DNA and protein sequences. We will refer to these sequences as *general sequences*. These scenarios are more complicated to handle, since virtually any text substring is a potential query.

IR systems have to provide very precise results in response to user queries, often identifying a few relevant documents among increasingly larger collections. These requirements can be addressed via a two-stage ranking process [163, 40]. In the first stage, a fast and simple filtration procedure extracts a subset of a few hundred or thousand candidates from the possibly billions of documents forming the collection. In the second stage, more complex learned ranking algorithms are applied to the reduced candidate set to obtain a handful of high-quality results. In this thesis, we focus on improving the efficiency of the first stage, freeing more resources for the second stage and thus increasing the overall performance. *Top-k query processing* is the key operation for solving the first stage. Given a query, the idea is to retrieve the k most "relevant" documents associated to that query for some definition of relevance.

In the case of solving top-k query processing on natural language collections, the inverted index requires extra data structures and complex algorithms in order to avoid having to perform a full evaluation of all documents. Authors in the past [152, 12, 59, 131, 13] proposed different algorithms, index organizations, and compression techniques to reduce the amount of work that a top-k query requires, and hopefully, reduce the size of the index at the same time. Despite the existing findings, we show that with a combination of recent well-engineered compact data structures and new algorithms, there is still room for improvement.

In order to handle the case of indexing a general sequence, traditional solutions build a *suffix tree* [164] or a *suffix array* [107], which are both indexes that can count and list all the individual occurrences of a string in the collection, in optimal or near-optimal time. Still this functionality is not sufficient to solve top-k query processing efficiently. The problem of finding top-k documents containing a substring, even with a simple relevance measure like term frequency, is challenging. Current state of the art solutions are complex to implement and their solutions demand 3 to 5 times the size of the data, making them impractical for large-scale scenarios.

In this thesis we present new compact indexes for IR on natural language, and also show extensions of these techniques to other relevant IR scenarios. In the first part we provide a comprehensive study of the basic concepts, compact data structures, and techniques that are employed.

In the second part, we present two new representations of the inverted index and new query algorithms for solving the top-k problem. The first technique is dubbed *Dual-Sorted* and is based on the work of Navarro and Puglisi [122]. We present this data structure and the implementation details for making it space-efficient and competitive in terms of query processing times. The second technique is dubbed *inverted treaps*, which is based on the *treap* data structure for representing the key elements of an inverted index and allows performing efficient top-k query processing algorithms. We also present an incremental version of this data structure, that can be applied in more complex scenarios. We performed comprehensive experiments and compared these new approaches to existing state-of-the-art techniques.

In the third part, we show a new compact index for solving top-k document retrieval problem on general sequences, based on an optimal theoretical solution presented by Navarro and Nekrich [119]. In this part we also present two techniques that are fundamental for an efficient implementation of the theoretical solution, for solving top-k two-sided queries on two-dimensional grids. For solving these queries, we introduce a new data structure, named the K^2-treap. At the end of this part, we present an experimental study of this index for general sequences and also for natural language. Our index is considerably faster than other approaches that are within the same space requirements.

In the fourth part we show how these techniques can be extended to other scenarios that are relevant for IR systems such as web access logs, indexing network packet traces and for solving aggregated two-dimensional queries on OLAP databases.

Finally, in the fifth part, we conclude with the main contributions of this thesis and discuss the future work.

Figure 1.1 shows the outline of this thesis, separated by parts and their main contributions.

1.1 Publications derived from the Thesis

The main contributions of this thesis have appeared in the following articles. We categorize the conference articles according to the CORE conference ranking (`http://core.edu.au/index.php/conference-rankings`).

Journals

1. Practical Compact Indexes for Top-k Document Retrieval. Simon Gog, Roberto Konow and Gonzalo Navarro. *ACM Journal of Experimental Algorithms (JEA)*. Submitted 2016. The main results of this article are discussed in Part III.
2. Inverted Treaps. Roberto Konow, Gonzalo Navarro, Charles L.A Clarke and Alex

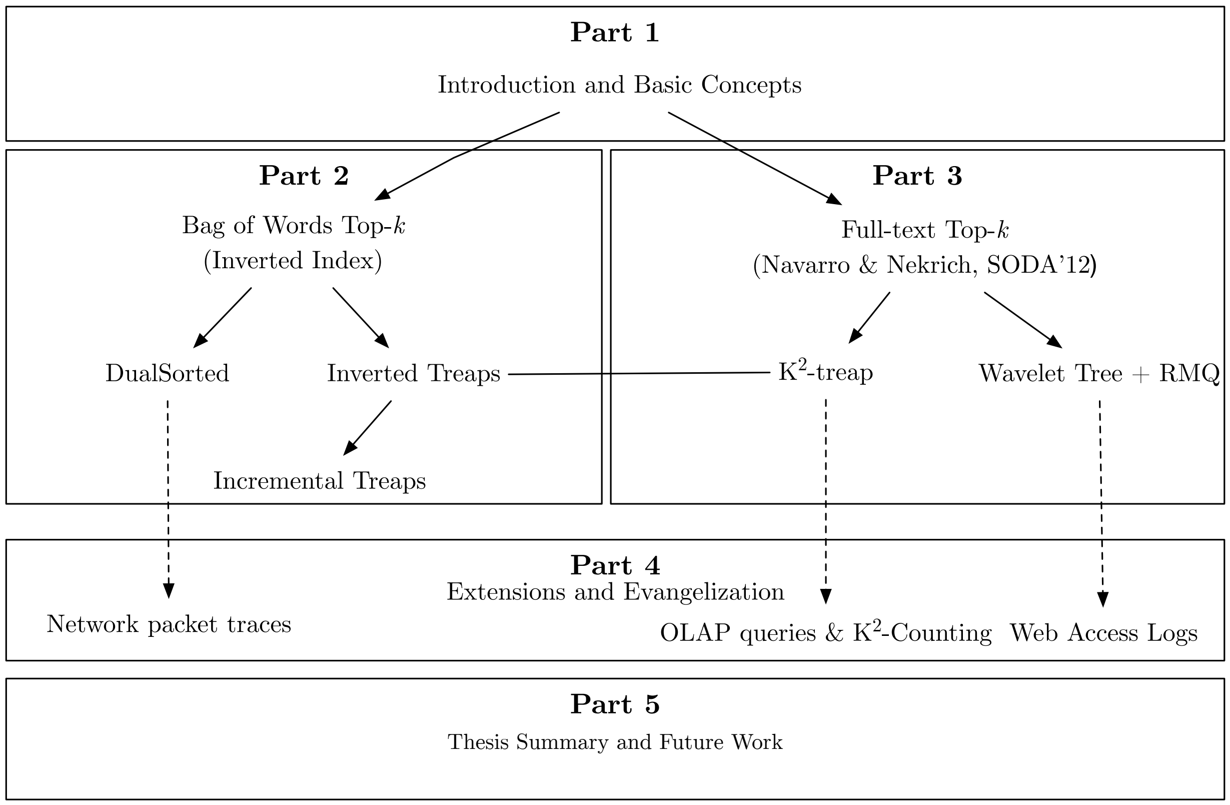


Figure 1.1: Outline of this thesis. Dotted lines represent extensions and applications derived from the main data structures and algorithms developed in the thesis.

López-Ortíz. *ACM Transactions on Information Systems (TOIS)*. Submitted 2016. Part II discusses the main results of this article.

3. Aggregated 2D Range Queries on Clustered Points [33]. Nieves R. Brisaboa, Guillermo de Bernardo, Roberto Konow, Gonzalo Navarro and Diego Seco. *Information Systems*, pages 34-49, 2016. Chapter 10 and Chapter 7 describes the main results of this article.
4. PcapWT: An Efficient Packet Extraction Tool for Large Volume Network Traces [98]. Young-Hwan Kim, Roberto Konow, Diego Dujovne, Thierry Turletti, Walid Dabbous, and Gonzalo Navarro. *Computer Networks*, 79:91-102, 2015. Chapter 12 describes the results of this article.

A* Conferences

5. Faster and Smaller Inverted Indices with Treaps [101] . Roberto Konow, Gonzalo Navarro, Charles Clarke, and Alejandro López-Ortíz. *Proceedings of the 36th ACM Special Interest Group in Information Retrieval (SIGIR) Conference*, pages 193-202, 2013. Part II discusses the main results of this article.
6. Faster Compact Top-k Document Retrieval [100]. Roberto Konow and Gonzalo Navarro. *Proceedings of the 23rd Data Compression Conference* (DCC), pages 351-360, 2013. Part III discusses the main results of this article.

B Conferences

7. K^2-Treaps: Range Top-k Queries in Compact Space [32]. Nieves Brisaboa, Guillermo de Bernardo, Roberto Konow, and Gonzalo Navarro. *Proceedings of the 21st String Processing and Information Retrieval conference* (SPIRE), pages 215-226. LNCS 8799, 2014. Chapter 10 describes the results of this article.
8. Efficient Representation of Web Access Logs [45]. Francisco Claude, Roberto Konow, and Gonzalo Navarro. *Proceedings of the 21st String Processing and Information Retrieval conference* (SPIRE), pages 65-76. LNCS 8799, 2014. Chapter 11 describes the results of this article.
9. Dual-Sorted Inverted Lists in Practice[99]. Roberto Konow and Gonzalo Navarro. *Proceedings of 19th the String Processing and Information Retrieval conference* (SPIRE), pages 295-306. LNCS 7608, 2012. Section 3.3 and Chapter 5 describes the main results of this article.

1.2 Awards

1. Capocelli Prize, Data Compression Conference 2013 for the presentation of the work *Faster Compact Top-k Document Retrieval.*

1.3 Research Stays and Internships

1. Research stay at the David R. Cheriton School of Computer Science,University of Waterloo. Advisors: Charles L.A Clarke and Alex López-Ortíz. June 2012 to December 2012.
2. Internship at eBay inc. Search Backend, Query Service Stack Group. Mentor: Raffi Tutundjian. Manager: Anand Lakshminath. June 2014 to September 2014.

Chapter 2

Basic Concepts

In this chapter, we define and explain the main concepts, data structures, and algorithms that are employed in the development of this thesis.

2.1 Computation model

All data structures used in this thesis work in the word-RAM model. In this model, the memory is divided into cells storing words of $w = \Theta(\log n)$ bits, where n is the size of the data. Any cell can be accessed and modified in constant time and is able to represent any integer in the set $\{0, \ldots, 2^w - 1\}$. In addition, other operations such as arithmetic operations, comparisons, and bitwise Boolean operations (and, or, exclusive-or, <<, >>), are also considered to require constant time.

2.2 Sequences, Documents and Collections

We define a sequence $\mathcal{S}$ of length n that contains symbols over an alphabet Σ of size σ. We define a collection of D documents $\mathcal{D} = \{d_1, d_2, \ldots, d_D\}$ containing symbols from an alphabet Σ of size σ, and $|d_i|$ denotes the number of symbols of document d_i. The total length of the collection is $\sum_{i=1}^{D} |d_i| = n$ and we call $\mathcal{C}$ the concatenation of its documents, where at the end of each d_i a special symbol $\#$ is used to mark the end of that document and the symbol $\$$ marks the end of the collection. It holds $\$ < \# < c$ for all $c \in \Sigma$. Figure 2.1 shows an example of a collection containing 3 documents with a total length of $n = 15$.

Another scenario is when a collection of documents contains text of *Natural Language*. In this case, the alphabet Σ can be divided into two disjoint sets: separators (i.e., space, tab) that separate words, and symbols that belong to the words. The set of distinct words form what is called the *vocabulary* of the collection. Natural language collections follow two empirical laws that are well accepted in IR [17]: Heaps' Law and Zipf's law. Heaps' law

$$\begin{array}{ccccccccccccccccc} i & = & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \end{array}$$

$$C = \underbrace{\mathbf{A}\ \mathbf{T}\ \mathbf{A}\ \#}_{d_1}\ \underbrace{\mathbf{T}\ \mathbf{A}\ \mathbf{A}\ \mathbf{A}\ \#}_{d_2}\ \underbrace{\mathbf{T}\ \mathbf{A}\ \mathbf{T}\ \mathbf{A}\ \#}_{d_3}\ \mathbf{\$}$$

Figure 2.1: Concatenation $\mathcal{C}$ of our 3-document example collection $\mathcal{D}$.

states that the vocabulary of a text of n words is of size $V = Kn^{\beta} = O(n^{\beta})$, where K and β depends on the text and $0 < \beta < 1$. On the other hand, Zipf's law attempts to capture the distribution of the frequencies, that is, the number of occurrences of the words in the text. It states that the i−th most frequent word in the collection has frequency f_1/i^{α}, where f_1 is the frequency of the most frequent word and α is a text dependent parameter. In other words, these two laws establish that the number of distinct terms (the size of the vocabulary) is much smaller than the collection, and that natural language documents contain few words that occur frequently and many words that rarely occur in the text.

2.3 Empirical Entropy

A widely used measure of the compressibility of a fixed sequence is the empirical entropy [137]. For a sequence $\mathcal{S}$ of length n containing symbols from an alphabet Σ of size σ, the zero-order entropy is defined as follows [1]

$$H_0(\mathcal{S}) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c} \tag{2.1}$$

where n_c is the number of occurrences of symbol c in $\mathcal{S}$. $H_0(\mathcal{S})$ gives the average number of bits needed to represent a symbol in S if the same symbol always receives the same code.

If we take into account the *context* in which the symbol appears, we can achieve better compression ratios by extending the definition to the kth order empirical entropy:

$$H_k(\mathcal{S}) = \sum_{c \in \Sigma^k} \frac{|\mathcal{S}^c|}{n} H_0(\mathcal{S}^c). \tag{2.2}$$

where $\mathcal{S}^c$ is the sequence of symbols preceded by the context c in $\mathcal{S}$. It can be proved that $0 \leq H_k(\mathcal{S}) \leq H_{k-1}(\mathcal{S}) \leq \cdots \leq H_0(\mathcal{S}) \leq \log \sigma$.

2.4 Huffman Codes

Given a sequence $\mathcal{S}[1, n] = x_1, x_2, \ldots, x_n$ containing n elements from an alphabet of size σ, we can encode each value of the sequence using $\lfloor \log_2 \sigma \rfloor + 1$ bits [2]. However, this representation

[1]We will always use $\log x$ to denote $\log_2 x$, unless otherwise stated.

[2]Assuming it is possible to map any symbol to an integer in $[1..\sigma]$.

does not take into account that some elements may appear more frequently than others and thus we can represent the sequence in a more efficient way. One way to reduce the space is to use *variable-length* codes, that is, employ codes that have different amounts of bits for different symbols. One family of variable-length codes are the ones that take advantage of statistical properties of the sequences and are referred to as statistical codes. The idea is to assign smaller codes to the symbols that occur more frequently and larger codes for the infrequent symbols. However, the task of assigning codes is not trivial: on the one hand, they have to be as small as possible, on the other hand, they have to be *prefix-free*. Prefix-free codes are uniquely decodable codes, since there is no code that is a prefix of another one.

Huffman codes [89] are prefix-free codes and belong to the family of statistical encoders. They represent a sequence $\mathcal{S}[1,n]$ using less than $H_0(\mathcal{S})+1$ bits per symbol, which is optimal (among methods that encode symbols separately). The algorithm to generate the codes builds what is known as a Huffman tree and is used to assign different codes to the symbols. The procedure to generate such codes and building the tree is as follows: first, one node for each distinct symbol is created; these are regarded as leaf nodes and store the symbol and its frequency of occurrences in $\mathcal{S}$. The two least frequent nodes, x and y, are then taken out of the set of nodes and merged together into a new internal node v, which is assigned as the parent of x and y. The frequency of v is the sum of the frequencies of its children. This procedure is repeated as long as two or more nodes remain without merging. The process finishes when a unique node remains, which is the root of the Huffman tree. Every left branch is labeled as 0 and the right one is labeled as 1. The code for each symbol is generated as the concatenation of the *labels* of the traversal path from the root to its leaf. The Huffman tree can be constructed in $O(\sigma \log \sigma)$ time if the symbols are not sorted according to their frequencies and in $O(\sigma)$ time if the symbols are sorted. The encoding procedure has to store the codeword lengths. An example of the Huffman tree and the generated codes are shown in Figure 2.2.

2.5 Integer Encoding Mechanisms

Given a sequence of positive integers $\mathcal{S}[1,n] = x_1, \ldots, x_n$, and letting M be the maximum integer in that sequence, it is possible to be represented using $\lceil \log_2 M \rceil$ bits per element and have access to any position in constant time. One problem with this representation is that it is not space-efficient if many integers are much smaller than others, so it is possible to represent the sequence more efficiently. For instance, we can use statistical compression such as Huffman codes. However, in some applications smaller values are more frequent, so in this case one can directly encode the numbers with codes that give shorter codes to smaller numbers. The advantage with this idea is that it is not necessary to store a mapping of the symbols to their codes, which can be prohibitive when the set of distinct numbers is too large.

In this section we briefly describe the most used and popular encoding mechanisms for integer sequences. We assume that the integer to be encoded is positive ($x > 0$) and we define BINARY(x, ℓ) as a function that returns the binary representation of a positive integer x using ℓ bits by adding leading zeroes as needed. We will refer to $|x|$ as the number of bits

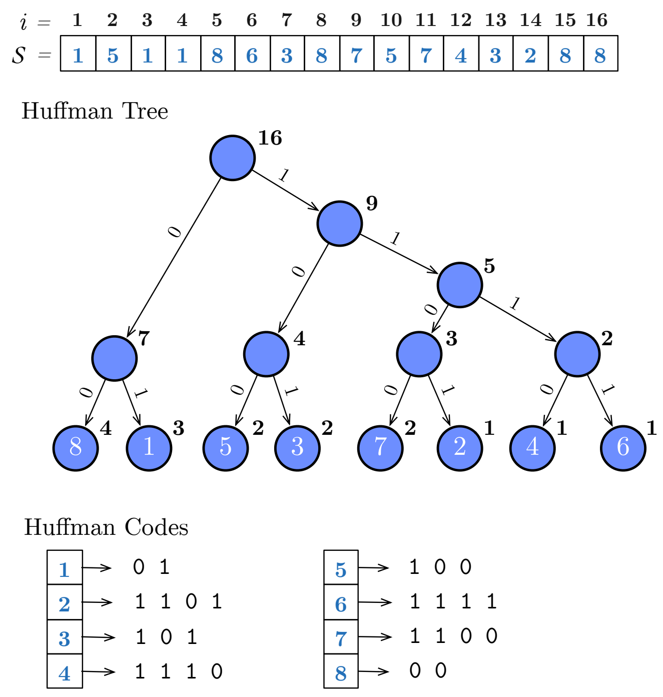


Figure 2.2: Top: A sequence over $\Sigma = [1, 8]$. The Huffman tree and the resulting Huffman codes are shown below. Numbers next to the tree nodes represent the accumulated frequency of that node.

that are needed to represent x in binary.

2.5.1 Bit-aligned Codes

Bit-aligned codes are used in the case of representing sequences containing small integers. We describe here the most representative codes of this family of encodings. We show an example of the encodings in Table 2.1.

Unary Codes [143]. An integer $x \geq 1$ is coded as a binary sequence with $x - 1$ one bits followed by a zero bit, $\textsc{unary}(x) = 1^{x-1}0$. For example, the integer 4 is represented as $\textsc{unary}(4) = 1110$. This encoding mechanism is usually only suitable for very small x values.

Gamma Codes (γ) [143]. Gamma codes represent $|x|$ using unary code and then concatenate it with the binary representation of $x - 2^{\lfloor \log x \rfloor}$ or, in other words, x without its most significant bit. It is defined as $\gamma(x) = \textsc{unary}(|x|)\textsc{binary}(x - 2^{\lfloor \log x \rfloor}, |x| - 1)$. For example, consider the number $x = 7$, and the binary length of the binary code is $|x| = 1 + \lfloor \log 7 \rfloor = 3$ (110 in unary) and $7 - 2^2 = 3$ (11 in binary), the concatenation is the binary sequence $\gamma(7) = 11011$. The amount of bits required for coding any x using γ codes is $1 + 2\lfloor \log x \rfloor$ bits.

Delta Codes [143]. The idea is to represent the length of the code using γ encoding and then concatenate it with the binary representation of the symbol x without its most significant bit. That is $\delta(x) = \gamma(|x|)\textsc{binary}(x - 2^{\lfloor \log x \rfloor}, |x| - 1)$. For $x = 7$, the length of the binary code is $|x| = 3$, so it is represented as 101 using γ encoding. The final representation is 10111. Encoding a number using Delta codes uses $1 + 2\lfloor \log \log(x + 1)\rfloor + \lfloor \log x \rfloor$ bits and it may be of interest when x is too large to be efficiently represented by γ codes.

Rice Codes [138, 143]. Rice Codes receive two values as inputs: the symbol x and a parameter b. The encoding is performed by representing $q = \lfloor (x - 1)/2^b \rfloor$ in unary and then representing $r = n - q2^b - 1$ in binary using b bits. The final code is the concatenation of r and q as $\textsc{rice}(x) = \textsc{unary}(\lfloor (x-1)/2^b \rfloor)\textsc{binary}(n - q2^b - 1, b)$. Consider the same example as before with $x = 7$ and choosing $b = 2$. $\textsc{unary}(\lfloor 6/4 \rfloor) = 10$ and $\textsc{binary}((7 - 4 - 1), b) = 10$, therefore $\textsc{rice}(7) = 1010$. Rice codes require $\lfloor (x - 1)/2^b \rfloor + b + 1$ bits in total.

Integer	**Unary**	γ	δ	**Rice**($b = 2$)
1	0	0	0	000
2	10	100	1000	001
3	110	101	1001	010
4	1110	11000	10100	011
5	11110	11001	10101	1000
6	111110	11010	10110	1001
7	1111110	11011	10111	1010
8	11111110	1110000	11000000	1011
9	111111110	1110001	11000001	11000
10	1111111110	1110010	11000010	11001

Table 2.1: Encoding positive integers using different bit-aligned encoding techniques.

2.5.2 Byte-aligned Codes

Bit-aligned codes can be inefficient at decoding since they require several bitwise operations which are not always efficiently implemented. Therefore, byte-aligned [147] or word-aligned codes [168, 11] are preferred when the speed is the main concern. Examples of this family of techniques are *Variable Byte* (VByte), Simple9, and PForDelta. We briefly describe the most representative methods of this family below:

Simple-9 [11]. Simple-9 is a 32-bit word-aligned encoding scheme. This mechanism packs as many numbers as possible into a 32-bit word by assigning an equal number of bits to each symbol. A selector value (b) has to be chosen and is encoded using 4 bits in a 32 bit word. The rest of the 28 bits from the 32-bit word will store as many integers as possible using a fixed amount of bits that depends on the selector value. We show in Table 2.2 the number of integers in a 32-bit word for different selector b values, the bits per integer (bpi) that are used and amount of bits that are not used. Other versions [13] extend this approach to make use of 64-bit words, having 60 bits instead of 28 bits available for data.

Selector b	**Ints**	**bpi**	**wasted bits**
0	1	28	0
1	2	14	0
2	3	9	1
3	4	7	0
4	5	5	3
5	7	4	0
6	9	3	1
7	14	2	0
8	28	1	0

Table 2.2: Different configurations for Simple9 depending on the selector value b

Variable Byte [165]. Variable Byte (vByte) is designed to represent large integers. The idea is to split the integer x_i into $\lceil |x|/7 \rceil$ byte chunks using 7 bits as payload and the most significant bit is used as a "continuation" bit. The continuation bit is set to 0 for the last byte of x_i and to 1 if another byte needs to be read. In practice, vByte provides a good trade-off between space and time, and is also considered simple to implement.

Varint [56]. Varint is Google's version of vByte that improves the decoding performance by reducing the branch miss-prediction caused by the continuation bit. The main difference is the use of two bits containing the exact amount of bytes that are required to be read at the beginning of the encoding chunks. Therefore, except for the first chunk, one can use the complete word for the other chunks. In practice, Varint decoding speed can be up to four times faster than vByte.

PForDelta [172]. PForDelta provides a very simple and fast decoding, which is usually preferable for many applications. This mechanism analyzes a fixed-size block of integers and chooses a sufficient number of bits to represent most of them. For example, suppose that in a block of 128 integer numbers, 90% of them are smaller than 32. We can then choose b, the number of bits per integer as 5, since most of the numbers will fit in 5 bits. Now we allocate 128×5 bits, plus some extra space for the 10% that will not fit into 5 bits. In practice, this scheme is one of the fastest for decoding and achieves good compression ratios.

SIMD based. A recent trend exploits the advantages of SIMD operations available on modern CPUs. For example, Varint-G8IU [151] tries to encode as many integers as possible into 8 consecutive bytes preceded by a one-byte descriptor, and uses SIMD instructions to encode/decode. Another example is the SIMD-BP128 [151] encoding mechanism, which can be seen as an adaptation of Simple9 using 128-bit SIMD words.

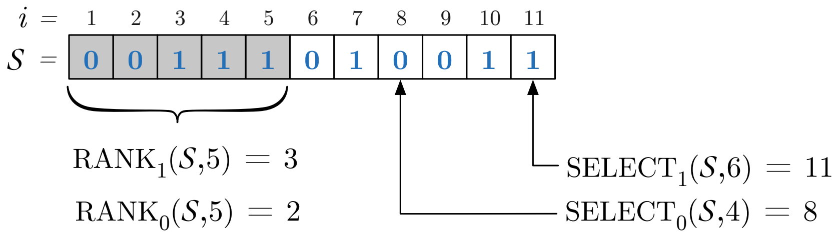
QMX. Recently, Trotman [157] introduced a new mechanism to encode integers, called *QMX*. This encoding scheme combines word-aligned, SIMD and run-length techniques, resulting in a highly space effective and extremely quick decoding scheme.

2.6 Rank, Select and Access

Three basic operations that are widely used in succinct data structures are RANK, SELECT and ACCESS in sequences. Given a sequence $\mathcal{S}[1,n]$ containing symbols from an alphabet Σ of size σ, these operations are defined as follows.

- RANK$_b(\mathcal{S},x)$ returns the number of elements equal to b up to position x.
- SELECT$_b(\mathcal{S},x)$ returns the position of the x-th occurrence of b in B.
- ACCESS$(\mathcal{S},x)$ returns the symbol at position x of the sequence. This operation will also be denoted as $\mathcal{S}[x]$.

Figure 2.3 shows an example of RANK and SELECT operations over a binary sequence $\mathcal{S}$.



i = 1 2 3 4 5 6 7 8 9 10 11
$\mathcal{S}$ = 0 0 1 1 1 0 1 0 0 1 1
RANK$_1$($\mathcal{S}$,5) = 3
RANK$_0$($\mathcal{S}$,5) = 2
SELECT$_1$($\mathcal{S}$,6) = 11
SELECT$_0$($\mathcal{S}$,4) = 8

Figure 2.3: Example of RANK and SELECT operations on a binary sequence $\mathcal{S}[1,11]$.

2.6.1 Binary Sequences

We describe how RANK and SELECT operations can be efficiently performed in the special case of sequences $B[1,n]$ containing binary symbols ($\sigma = 2$).

Uncompressed solution We describe the classic representation introduced by Munro [113] and Clark [44] that requires $o(n)$ bits of extra space to the n bits required for the sequence. This solution is able to answer RANK, SELECT and ACCESS in constant time. In order to answer RANK we start by dividing the sequence B in super-blocks of length $s = \frac{\log^2 n}{2}$. We store in an array $S[1,\frac{n}{s}]$ the corresponding answers for each RANK up to the end of each super-block. There are n/s values to be stored in array S, and none of them can be bigger than n, so the space required for this array is $\frac{n}{s}\log n = \frac{2n}{\log n} = o(n)$ bits. We proceed with another subdivision of B into smaller blocks of length $b = \frac{\log n}{2}$ bits. Each super-block of size s has s/b blocks. We store into an array $R[1,n/b]$ the corresponding RANK results relative to each super block. Each value of R therefore requires at most $\log s$ bits. Summing up, we have $\frac{n}{b}\log s = \frac{2n\log(\log^2 n)/2)}{\log^2 n} \leq \frac{4n\log\log n}{\log n} = o(n)$ bits. Finally, a table $T[1,2^b][1,b]$ is constructed containing the pre-computed results of RANK for any position of any segment of b bits. There are 2^b possible sequences of bits of size b, and for each one the maximum value is b, requiring in total $2^b \cdot b\log b \leq \frac{\sqrt{n}}{2}\log n\log\log n = o(n)$ bits for storing table T. To answer RANK$_1(B,i)$ we proceed as follows: we determine the relative positions of i in the arrays S, R and table

T, such that i is decomposed as $i = j \cdot s + k \cdot b + r$. To obtain the final result we continue by summing $S[j] + R[j(\frac{s}{b}) + k] + T[B[j \cdot s + k \cdot b, j \cdot s + (k+1) \cdot b]][r]$. Note that we can answer $\text{RANK}_0(B, i)$ by performing $i - \text{RANK}_1(B, i)$.

In order to support SELECT in constant-time a similar procedure is performed: instead of dividing the sequence into blocks and super-blocks of fixed size, we divide the sequence according to the number of bits that are set. This way, we have blocks of variable length but every block will contain a fixed amount of 1's inside them. As before, a similar table T with precomputed values is constructed and stored.

An implementation of this solution was introduced by González et al. [80] supporting RANK and ACCESS in constant time and SELECT in $O(\log n)$ time [3]. Experimental results show that this implementation requires only 5% extra space of the original sequence to support these operations. We will refer to this implementation as RG. In practice, operation $\text{SELECT}_b(B, x)$ is answered by performing a binary search on the values from the arrays $S[1, \frac{n}{s}]$ and $R[1, n/b]$ and then sequentially scanning until the the position where the number of ones equals x.

Compressed solution. A compressed representation of the bitmap B that is able to answer RANK and SELECT operations was presented by Raman et al. [135, 136]. Their solution reduces the space required to represent the bitmap to $nH_0(B) + o(n)$ bits.

The idea is to divide the sequence into blocks of size $b = \frac{\log n}{2}$. A block will be assigned to a class c_i if it contains exactly c_i 1s. Since each block has b bits, we have at most $b+1$ classes. However, each class c_i has at most $\binom{b}{c_i}$ different shuffles of its bits. We can represent any block B_i as a pair (c_i, o_i), where c_i indicates the class and o_i is the *offset* within that class. For example, the class c_2 for blocks of length $b = 3$ containing 2 ones is $c_2 = \{011, 101, 110\}$ and their offsets are $o_2 = \{0, 1, 2\}$. The binary representation of each class c_i requires $\lceil \log(b+1) \rceil$ bits. We will store in a sequence $C[1, \lceil n/b \rceil]$ the class of each block, requiring a total of $\lceil n/b \rceil \lceil \log(b+1) \rceil = (n/b) \log b + O(n/b)$ bits. The offsets o_i are stored separately in a sequence $O[1, n/b]$, and each o_i component requires $\lceil \log \binom{b}{c_i} \rceil$ bits, so the total space in bits for representing sequence $O[1, n/b]$ is:

$$\sum_{i=1}^{\lceil n/b \rceil} \left\lceil \log \binom{b}{c_i} \right\rceil \leq \sum_{i=1}^{\lceil n/b \rceil} \log \binom{b}{c_i} + \lceil n/b \rceil = \log \prod_{i=1}^{\lceil n/b \rceil} \binom{b}{c_i} + \lceil n/b \rceil \leq$$

$$\log \binom{n}{m} + \lceil n/b \rceil \leq nH_0(B) + \lceil n/b \rceil$$

where m is the number of 1's in the sequence [128].

In order to support RANK and SELECT operations we still need to add additional arrays storing relative results, pointers and precomputed tables, similarly to the uncompressed solution. Raman et al. [136] solution obtains a final space bound of $nH_0(B) + o(n)$. We will refer to practical implementations of this approach as RRR. In practice, the cost of handling

[3]Please note, that even though SELECT can be solved in $O(1)$, most implementations use a binary search to solve *select*, thus making the implementation of SELECT$O(\log n)$ time

a reduced-space bitmap is that queries are up to two times slower than the uncompressed alternative (`RG`).

Other solutions. Several other practical implementations have been proposed. Okahonara and Sadakane [126] presented a data structure called `SDArray` that requires $nH_0(B) + 2m + o(m)$ bits for representing a bitmap with m 1s. It is able to support SELECT in constant time, and RANK and ACCESS in time $O(\log(n/m))$. Vigna [160] introduced an implementation that uses SIMD words to improve the space and speed of RANK and ACCESS operations. Navarro and Providel [134, 121] presented a new representation that improved the space/time tradeoffs for SELECT operations. Recently, Kärkkäinen et al. [97] introduced the *hybrid bit vector* which divides the bitmap into blocks and then chooses the best encoding of each block separately from the techniques previously described. Most of the techniques previously described were implemented and improved to support large bit sequences by Gog et al. [77].

2.6.2 General Sequences

There are different approaches for solving RANK, SELECT and ACCESS for the case of general sequences (i.e., $\sigma > 2$). A naive solution consists in creating a bitmap $B_i[1, n]$ for each distinct symbol $i \in \Sigma$, by setting $B_i[x] = 1$ iff $\mathcal{S}[x] = i$. The operations RANK and SELECT can then be solved by querying the corresponding bitmap B_i. However, this solution requires $n\sigma + o(n\sigma)$ bits if an uncompressed solution is used (i.e., `RG`). Another drawback of this solution is that one has to traverse σ bitmaps in order to answer ACCESS in the worst case, making it unpractical for sequences having large alphabets. We can improve the size by using the `SDArray`, where the space becomes $nH_0(\mathcal{S}) + O(n)$ bits, however method ACCESS is still inefficient.

The most popular data structure employed to solve RANK, SELECT and ACCESS operations on general sequences is the *wavelet tree* [81, 117] . We describe in detail the properties and virtues of this structure in Section 2.7. Other very competitive alternatives are `GMR` [79] and Alphabet Partitioning (`AP`) [19] solutions. We briefly describe these solutions below, however we do not explain how the inner mechanisms work.

GMR. Golynski et al. [79] presented a data structure that uses $n \log \sigma + o(n \log \sigma)$ bits and answers SELECT in constant time. This data structure requires $O(\log \log \sigma)$ time to answer ACCESS and RANK operations. `GMR` is of interest when the speed of the operations is the main concern and for big alphabets, since the size of the final structure is not compressed.

Alphabet Partitioning. In the case where the alphabet of the sequence is big and the size is the main concern, the technique called *alphabet partitioning* is preferred. It supports RANK, SELECT, and ACCESS operations in $O(\log \log \sigma)$ time and represents the sequence $\mathcal{S}$ in $nH_0(\mathcal{S}) + o(nH_0(\mathcal{S}) + n)$ bits.

2.7 Wavelet Tree

In this section we describe the main properties and operations of the wavelet tree [81, 117]. We start by describing the composition of the data structure and then explain how the wavelet tree answers RANK, SELECT and ACCESS operations. We then describe other applications of the wavelet trees when representing grids.

2.7.1 Data Structure

Let $\mathcal{S}[1,n]$ be a sequence of symbols from a finite alphabet Σ of size σ. For simplicity we will assume that $\Sigma = [1..\sigma]$. The wavelet tree over an alphabet $[a..b] \subseteq [1..\sigma]$ is a balanced binary tree that stores a bit sequence in every node except the leaves. It has $b-a+1$ leaves and if $a \neq b$ it has at least an internal node, v_{root} that represents $\mathcal{S}[1,n]$. The root of the tree is represented by a bit sequence, $B_{v_{root}}[1,n]$, and is defined as follows: if $\mathcal{S}[i] \leq (a+b)/2$ then $B_{v_{root}}[i]$ is marked with a '0', and if $\mathcal{S}[i] > (a+b)/2$ then $B_{v_{root}}[i]$ is marked with a '1'. We define $\mathcal{S}_{left}[1, n_{left}]$ as the subsequence of symbols from $\mathcal{S}[1,n]$ that were marked with a '0' in $B_{v_{root}}$, that is, all symbols c in $\mathcal{S}[1,n]$ such that $c \leq (a+b)/2$. Let $\mathcal{S}_{right}[1, n_{right}]$ be the sequence of symbols that were marked with a '1' in $B_{v_{root}}$, that is, all symbols c such that $c > (a+b)/2$. The left child of v_{root} is going to be a wavelet tree of a sequence $\mathcal{S}_{left}[1, n_{left}]$ over an alphabet $[a..\lfloor(a+b)/2\rfloor]$ and the right child of v_{root} is a wavelet tree of the sequence $\mathcal{S}_{right}[1, n_{right}]$ over an alphabet $[1+\lfloor(a+b)/2\rfloor..b]$. This decomposition is done recursively until the alphabet of the subsequence is a single symbol. The tree has σ leaves and each level of the tree requires n bits. The height of the tree is $\lceil\log\sigma\rceil$, thus we need $n\lceil\log\sigma\rceil$ bits to represent the tree.

Each bit sequence at every level must be capable of answering ACCESS, RANK and SELECT queries, thus we can use the `RG` or `RRR` representation to represent the bit sequences. If `RG` is employed, the wavelet tree requires $n\log\sigma + o(n\log\sigma)$ bits. On the other hand, if we use `RRR` to represent the bit sequences at each level, the wavelet tree uses $nH_0(\mathcal{S}) + o(n\log\sigma)$ bits of space. If we want to compress the data structure even more, we can change the shape of the tree to the shape of the Huffman tree of the symbols frequencies appearing in $\mathcal{S}$. Figure 2.4 shows an example of a regular wavelet tree and a Huffman shaped wavelet tree is shown on Figure 2.5.

2.7.2 Rank, Select and Access

The regular wavelet tree answers RANK, SELECT and ACCESS in $O(\log\sigma)$ time, that is to say, the execution time depends only on the size of the alphabet, not on the length of the sequence. If a Huffman shaped wavelet tree is employed, the average time required of these operations is $O(1+H_0(\mathcal{S}))$.

We define LABELS(v) as a method that returns the set of symbols that belong to the subrange $[a..b] \subseteq [1..\sigma]$ from the sequence represented by node v and we will refer to v_l or v_r

as the left or right child of node v respectively.

We will explain how to perform ACCESS operation through an example based on the wavelet tree from Figure 2.4. Let us assume that we want to perform $\textsc{access}(\mathcal{S}, 6)$, that is, return the symbol located at position 6. We start at position 6 of the root bit sequence ($B_{v_{root}}$) and ask if the corresponding bit is marked as '1' or '0'. If the bit is '0', we go to the left branch, if not, to the right. In this case $B_{v_{root}}[6] = 1$, so we go to the right branch of the tree. Now we have to map the original position ($x = 6$) to the corresponding position at the right branch. In other words, we want to know how many $1's$ were at the root bit sequence up to position 6. We can easily do this in constant time by performing the operation supported by `RG` and `RRR` bit sequences: $\textsc{rank}_1(B_{v_{root}}, 6)$, which returns the value 3. Using this value we can now go to the right node of the root, v, and execute the same procedure. The value at $B_{v_{root}}[3]$ is 0 so we know we have to go to the left and count how many values to the left exist in B_v by executing $\textsc{rank}_0(B_v, 3) = 2$. This process is done recursively until we reach a leaf. Algorithm 1 shows this procedure as $\textsc{acc}(v, x)$. Note that we do not need to store the label at the leaves; they are deduced as we successively restrict the subrange $[a..b]$ as we descend down the tree.

Algorithm 1: Basic wavelet tree algorithms: On the wavelet tree of sequence $\mathcal{S}$, $\textsc{acc}(v_{root}, x)$ returns $S[x]$; $\textsc{rnk}(v_{root}, j, b)$ returns $\textsc{rank}_b(\mathcal{S}, x)$; and $\textsc{sel}(v_{root}, x, b)$ returns $\textsc{select}_b(\mathcal{S}, x)$.

```
ACC(v, x)
  if v is a leaf then
    return LABELS(v)
  if B_v[x] = 0 then
    x ← RANK_0(v, x)
    return ACC(v_l, x)
  else
    x ← RANK_1(v, x)
    return ACC(v_r, x)
```

```
RNK(v, x, b)
  if v is a leaf then
    return x
  if b ∈ LABELS (v_l) then
    x ← RANK_0(B_v, x)
    return RNK(v_l, x, b)
  else
    x ← RANK_1(B_v, x)
    return RNK(v_r, x, b)
```

```
SEL(v, x, b)
  if v is a leaf then
    return x
  if b ∈ LABELS (v_l) then
    x ← SEL(v_l, x, b)
    return SELECT_0(B_v, x)
  else
    x ← SEL(v_r, x, b)
    return SELECT_1(B_v, x)
```

A similar procedure is performed for $\textsc{rank}_b(\mathcal{S}, x)$. We start at the root of the tree and check if the symbol b belongs to the left or right child of the root. If the symbol belongs to the left subtree, we transform the position x to a position in the the left subtree by performing $x = \textsc{rank}_0\,(B_{v_{root}}, x)$ or $x = \textsc{rank}_1\,(B_{v_{root}}, x)$ if the symbol belongs to the right subtree. This is done recursively until we reach a leaf, where the final answer is x. Algorithm 1 shows this procedure as $\textsc{rnk}(v, x, b)$.

The case of $\textsc{select}_b(\mathcal{S}, x)$ is different: instead of starting at the root, we start at position x of the leaf that represents symbol b in the wavelet tree. If the leaf is the left child of a node v, then the x-th occurrence of symbol b in v is at position $x = \textsc{select}_0(B_v, x)$, and if the leaf is the right child of v the x-th occurrence of b in v is located at the position $x = \textsc{select}_1(B_v, x)$. We continue this procedure until we reach the root bit sequence of the wavelet tree $B_{v_{root}}$ and return position x. Algorithm 1 shows this procedure as $\textsc{sel}(v, x, b)$.

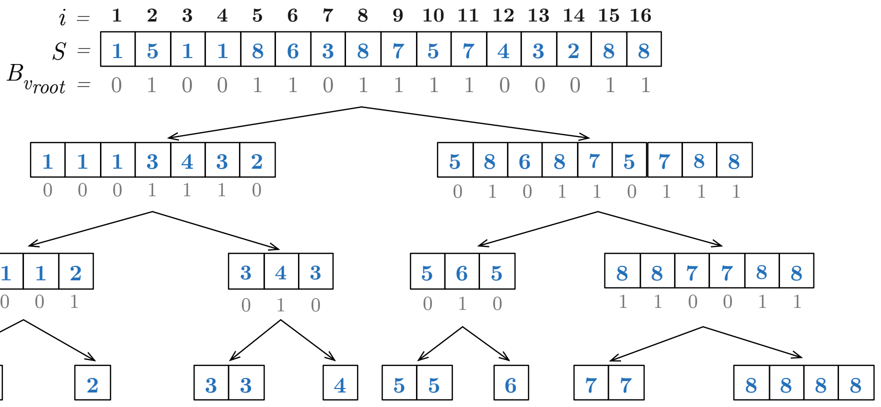


Figure 2.4: Wavelet tree of the sequence $\mathcal{S} = 1511863875743288$. The bitmaps are represented below the sequences in grey. For example, B_{root} represents the bitmap corresponding to the root of the wavelet tree. Note that none of the values shown in the cells need to be stored.

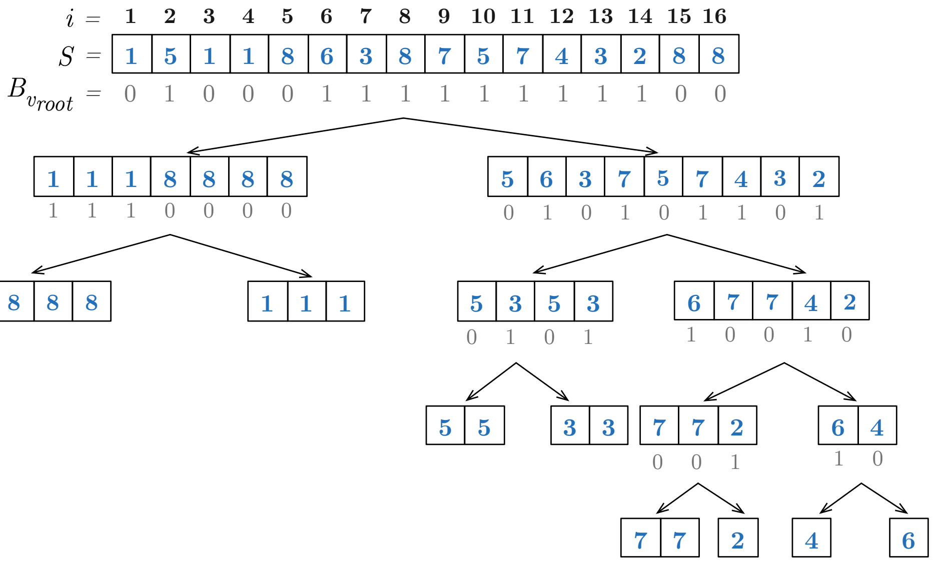


Figure 2.5: Huffman shaped wavelet tree of an example sequence $\mathcal{S} = 1511863875743288$.

2.7.3 Wavelet Trees and Grids

An $n \times m$ grid with n points, exactly one per column (i.e., x values are unique), can be represented using a wavelet tree [106]. In this case, this is a perfect balanced binary tree of height $\lceil \log m \rceil$ where each node corresponds to a contiguous range of values $y \in [1, m]$ and represents the points falling in that y-range, sorted by increasing x-coordinate. The root represents $[1, m]$ and the two children of each node split its y-range by half, until the leaves represent a single y-coordinate. Each internal node stores a bitmap, which tells whether each point corresponds to its left or right child. Adding RANK and SELECT support on the bitmaps, the wavelet tree uses $n \log m + o(n \log m)$ bits. Any point can be tracked up (to find its x-coordinate) or down (to find its y-coordinate) in $O(\log m)$ time as well. This case can be interpreted as representing the grid of a sequence $\mathcal{S}[1, n]$ and each two dimensional point (x, y) is regarded as $(i, \mathcal{S}[i])$.

When the grids contain more than one point per column, an additional bitmap B is used to map from the original domain to a new domain that satisfies this constraint. This bitmap stores, for each column, a bit 1 followed by as many zeros as the number of points in such a column. Then, a range of columns $[c_{sp}, c_{ep}]$ in the original domain can be mapped to a new range $[\text{RANK}_0(B, \text{SELECT}_1(B, c_{sp}) + 1), \text{RANK}_0(B, \text{SELECT}_1(B, c_{ep} + 1))]$. If the grid is very sparse and/or the distribution of the data points is very skewed, this bitmap can be compressible using RRR (recall Section 2.6.1). Figure 2.6 shows the wavelet tree and the grid interpretation of an example sequence.

The idea of representing grids using a similar structure to the wavelet tree was originally introduced by Chazzelle [42], where the wavelet tree can be seen as a slight generalization of a rather old (1988) data structure that is heavily used in computational geometry. The use of a wavelet tree to represent grids was later rediscovered and extended [106, 120, 117].

2.7.4 Two-Dimensional Range Queries

The wavelet tree can also report, in $O(occ \log(m/occ))$ time, the occ points lying inside a rectangle $[x_1, x_2] \times [y_1, y_2]$ as follows: Start at the root node v_{root} with the interval $[x_1, x_2]$ and project those values towards the left and right nodes (v_l and v_r respectively). The range of the projection to the left is $[\text{RANK}_0(B_l, x_1 - 1) + 1)$,$\text{RANK}_0(B_l, x_2)]$, and similarly to the right with RANK_1 and B_r. This is continued until reaching at most $y_2 - y_1 + 1$ wavelet tree nodes that cover $[y_1, y_2]$. Note that, if at some point a range becomes empty we can discard that projection and if we enter a node whose symbols are disjoint from the range $[y_1, y_2]$, we can also discard that traversal. As soon as we reach a leaf we can report the symbol (i.e y-coordinate) including the number of occurrences in the range, $x_2 - x_1 + 1$. Algorithm 2 shows this procedure as RANGE_REPORT. Algorithm 2 can also be extended to support multiple ranges at the same time. At the bottom of Figure 2.6 we show in grey the range corresponding to $[10, 14] \times [3, 7]$. The wavelet tree shows, also in grey background, the values that belong to this range. This operation takes $O(\log \sigma + m \log \frac{y_2 - y_1 + 1}{m})$ time, where m is the number of reported points.

The wavelet tree also supports other operations that are useful for information retrieval scenarios [117, 75, 74]. The most relevant ones related to this thesis are briefly described below:

RANGE_QUANTILE(x_1, x_2, k)**:** Returns the kth smallest value in $\mathcal{S}[x_1, x_2]$ in time $O(\log m)$.

RANGE_NEXT_VALUE(x_1, x_2, k)**:** Returns the smallest $\mathcal{S}[v] \geq k$ value such that $x_1 \leq v \leq x_2$ in $O(\log m)$ time.

RANGE_COUNT(x_1, x_2, y_1, y_2)**:** Returns the number of points that lie in a query rectangle $[x_1, x_2] \times [y_1, y_2]$ in $O(\log m)$ time.

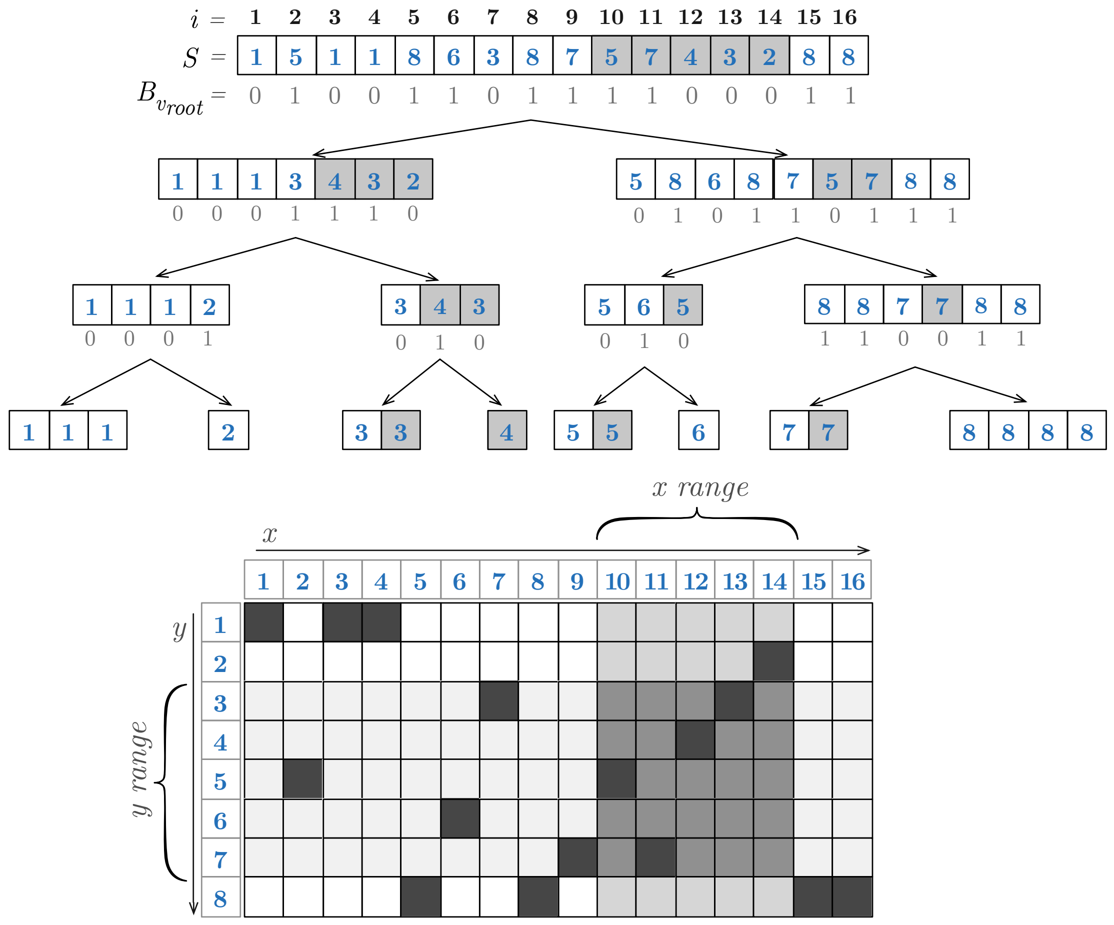


Figure 2.6: Top: The wavelet tree for the sequence $\mathcal{S} = 1511863875743288$. Bottom: The grid representation of the sequence. The $x-$ and $y-$ranges are shown in light grey, while their intersection is shown in darker grey. The highlighted values on the wavelet tree correspond to the ones that belong to the range $[10, 14] \times [3, 7]$.

2.8 Directly Addressable Codes

Directly Addressable Codes (DAC) [34] is a variable-length encoding that is inspired by vByte. Given a chunk length b and a sequence of positive integers $\mathcal{S}[1, n] = x_1, x_2, \ldots, x_n$, DAC divides each integer x_i into $\lceil |x_i|/b \rceil$ chunks. Similar to vByte, DAC employs a 'continuation' bit, but instead of concatenating the encoding of x_{i+1} after that of x_i, it builds a multi-layer data structure. Let M be the maximum element in the sequence, a DAC encoding will contain $\ell = \lceil \lfloor \log(M) \rfloor + (1/b) \rceil$ layers consisting of two parts: (1) the lowest b bits of each chunk, which are stored contiguously in an array A_k with $1 \leq k \leq \ell$, using b bits per element, and (2) the 'continuation' bits, which are concatenated into a bitmap B_k. Any integer x_i will require exactly $\lceil (|x_i|)/b \rceil$ layers. For example, say we want to encode the integer $x = 5$ and we set the chunk length $b = 2$. The lowest b bits of 5 are 01 and we append them to the first layer of array A_1. Since we still require more bits to represent the number, we need to set the continuation bit to 1, and this is appended to the bit sequence B, again, at first layer. The next b lowest bits for representing x are now located in the next layer A_2, and since we

Algorithm 2: Method RANGE_REPORT$(B_v, x_{start}, x_{end}, y_1, y_2)$. Reports all the symbols that are located in the range $[x_{start}, x_{end}] \times [y_1, y_2]$ and the frequency of occurrences within the range using a wavelet tree.

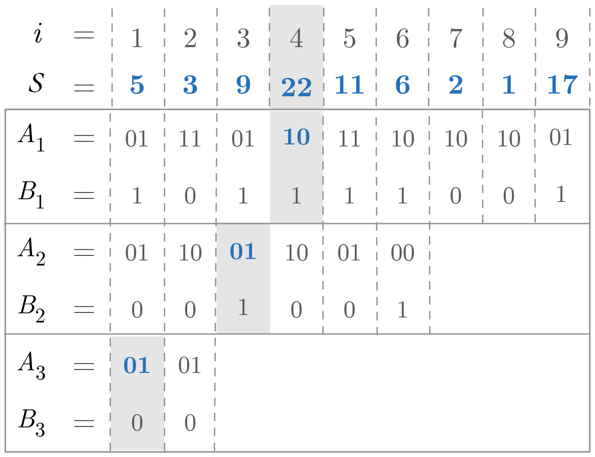
```
RANGE_REPORT$(v, x_{start}, x_{end}, y_1, y_2)$
  if $x_{start} > x_{end} \vee$ LABELS $(v) \cap [y_1, y_2] = \emptyset$ then
    return
  else if $v$ is a leaf then
    report (LABELS $(v)$ , $x_{end} - x_{start} + 1$)
  else
    $x_{start\_left} \leftarrow$ RANK$_0(B_v, x_{start} - 1) + 1$
    $x_{end\_left} \leftarrow$ RANK$_0(B_v, x_{end})$
    $x_{start\_right} \leftarrow x_{start} - x_{start\_left}$
    $x_{end\_right} \leftarrow x_{end} - x_{end\_left}$
    RANGE_REPORT$(v_l, x_{start\_left}, x_{end\_left}, y_1, y_2)$
    RANGE_REPORT$(v_r, x_{start\_right}, x_{end\_right}, y_1, y_2)$
```

do not require more bits to represent x we set the continuation bit to 0. Figure 2.7 shows this example in $\mathcal{S}[1]$. DAC codes allows direct access to the sequence in time $O(\ell/b)$ and the space overhead for storing a number of ℓ bits is $\lceil \ell/b \rceil + b$ bits. A further improvement of DAC encoding is to choose different chunk lengths for each layer. The authors [34] have presented a dynamic programming algorithm that computes the optimal chunk length and the optimal number of layers to achieve the most compact representation of the sequence $\mathcal{S}$ using DAC.

We show an example of the encoding of integers using DAC in Figure 2.7. To retrieve the number located at a position i in $\mathcal{S}$ we start by reading $A_1[i]$ and $B_1[i]$. If $B_1[i] = 0$, then we are done with the decoding and the number $\mathcal{S}[i]$ corresponds to $A_1[i]$. On the other hand, if $B_1[i] \neq 0$ it means that $\mathcal{S}[i]$ continues in the next layer. To compute the corresponding position in the next layer we perform $i' = \text{RANK}_1(B_1, i)$ and concatenate the value of $A_2[i']$ with $A_1[i]$. Again, we need to check if $B_2[i']$ is marked or not, and continue if necessary. We show an example of this procedure in Figure 2.7. The highlighted parts correspond to the traversal when obtaining the value located at $\mathcal{S}[4]$. The final result is the concatenation of these segments: $\mathcal{S}[4] = A_3[1]A_2[3]A_1[4] = 010110 = 22$.

2.9 K^2-tree

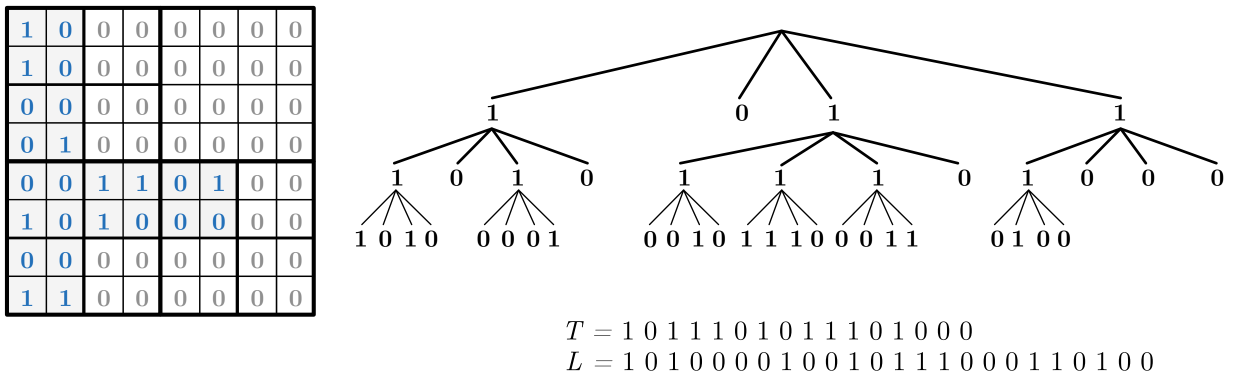
The K^2-tree [35] is a data structure to compactly represent sparse binary matrices (which can also be regarded as point grids). The K^2-tree subdivides the matrix into K^2 submatrices of equal size. The submatrices are considered in row major order, top to bottom. Each one is represented with a bit, set to 1 if the submatrix contains at least one non-zero cell. Each node whose bit is marked is recursively decomposed, subdividing its submatrix into K^2



i =	1	2	3	4	5	6	7	8	9
$\mathcal{S}$ =	5	3	9	22	11	6	2	1	17
A_1 =	01	11	01	10	11	10	10	10	01
B_1 =	1	0	1	1	1	1	0	0	1
A_2 =	01	10	01	10	01	00			
B_2 =	0	0	1	0	0	1			
A_3 =	01	01							
B_3 =	0	0							

Figure 2.7: Example of a DAC structure for the sequence $\mathcal{S} = 5, 3, 9, 22, 11, 6, 2, 1, 17$ and $b = 2$. The highlighted segments represent the DAC representation for $\mathcal{S}[4] = 22$.

children, and so on. The subdivision ends when a fully-zero submatrix is found or when we reach the individual cells. The K^2-tree is represented with two bitmaps: one that describes the topology of the tree (bitmap T) and is built by traversing the matrix tree level-wise, appending the K^2 bits of each node, excluding the parents of leaves. The other bitmap, L, that represents the the leaves of tree, stores all the cell values. In practice, this compact data structure has shown to be efficient at representing web graphs and grids and is able to answer range queries efficiently. It has no good worst-case time guarantees, yet in practice query times are competitive. In a worst-case scenario, if a $n \times n$ matrix contains t points, the K^2-tree needs $K^2\, t \log_{K^2} \frac{n^2}{t}(1 + o(1))$ bits. This can be reduced to $t \log \frac{n^2 \cdot K^2}{t}(1 + o(1))$ bits if the bitmaps are compressed. This is similar to the wavelet tree space, but in practice K^2-trees use much less space when the points are clustered [35]. Figure 2.8 shows an example of a K^2-tree.



1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	1	1	0	1	0	0
1	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0

$T = 1\ 0\ 1\ 1\ 1\ 0\ 1\ 0\ 1\ 1\ 1\ 0\ 1\ 0\ 0\ 0$
$L = 1\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 0\ 1\ 1\ 1\ 0\ 0\ 0\ 1\ 1\ 0\ 1\ 0\ 0$

Figure 2.8: On the left, a sparse binary matrix. On the right, its corresponding K^2-tree with $K = 2$. The topology bitmap (T) and leaves bitmap (L) are shown below the tree.

The procedure to traverse the tree is the following: any 1 located at position p in T represents an internal node in the tree, and its K^2 children start right after position $sp =$

$\text{RANK}_1(T, p) \cdot K^2$ as all internal nodes are marked with a 1 in T and they have exactly K^2 children. If the starting position of the children of a node is greater than the length of bitmap T, then its children are leaves, and their positions have to be mapped to the bitmap L as $sp - |T|$.

The K^2-tree can also answer range queries with multi-branch top-down traversal of the tree, following only the branches that overlap the query range. This is illustrated in Algorithm 3 (adapted from Algorithm 1 in [35]), which, to solve the query $Q = [x_1, x_2] \times [y_1, y_2]$, is invoked as $\text{RANGE}(n, x_1, x_2, y_1, y_2, 0, 0, -1)$. While this algorithm has no good worst-case time guarantees, in practice times are competitive.

Algorithm 3: $\text{K}^2\_\text{RANGE}(n, x_1, x_2, y_1, y_2, d_p, d_q, p)$ reports all points in the area $[x_1, x_2] \times [y_1, y_2]$ using a K^2-tree.

```
K^2_RANGE($n, x_1, x_2, y_1, y_2, d_p, d_q, p$)
   if $p \geq |T|$ then
      if $L[p - |T|] = 1$ then
         report $\langle d_p, d_q \rangle$
   else
      if $p = -1$ or $T[p] = 1$ then
         $y \leftarrow \text{RANK}_1\ (T, p) \cdot K^2$
         for each $i \in [\lfloor x_1/(n/k) \rfloor .. \lfloor x_2/(n/k) \rfloor]$ do
            if $i = \lfloor x_1/(n/k) \rfloor$ then
               $x'_1 \leftarrow x_1\ mod\ (n/k)$
            else
               $x'_1 \leftarrow 0$
            if $i = \lfloor x_2/(n/k) \rfloor$ then
               $x'_2 \leftarrow x_2\ mod\ (n/k)$
            else
               $x'_2 \leftarrow (n/k) - 1$
            for each $j \in [\lfloor y_1/(n/k) \rfloor .. \lfloor y_2/(n/k) \rfloor$ do
               if $j = \lfloor y_1/(n/k) \rfloor$ then
                  $y'_1 \leftarrow y_1\ mod\ (n/k)$
               else
                  $y'_1 \leftarrow 0$
               if $j = \lfloor y_2/(n/k) \rfloor$ then
                  $y'_1 \leftarrow y_2\ mod\ (n/k)$
               else
                  $y'_2 \leftarrow (n/k) - 1$
               K^2_RANGE($n/k, x'_1, x'_2, y'_1, y'_2, d_p + (n/k) \cdot i, d_q + (n/k) \cdot j, y + k \cdot i + j$)
            end for
         end for
```

2.10 Compact Trees

There are $\Theta(4^n/n^{3/2})$ general trees of n nodes [114], and thus one needs $\log(4^n/n^{3/2}) = 2n - \Theta(\log n)$ bits to represent any such tree. There are various compact tree representations using $2n + o(n)$ bits that can in addition carry out many tree operations efficiently, including retrieving the first child, next sibling, computing postorder of a node, lowest common ancestor, and so on. We describe three compact tree representations: Balanced Parenthesis (`BP`), Level-Ordered Unary Degree Sequence (`LOUDS`) and Depth-First Unary Degree Sequence (`DFUDS`):

BP. Balanced Parentheses were introduced by Jacobson [93] and later improved to allow constant time operations [114]. They represent a compact tree built from a depth-first preorder traversal of the tree. The idea is to write an *opening* parenthesis when arriving at a node for the first time and a *closing* parenthesis after traversing the subtree of the node, therefore each node will generate two parentheses. The parentheses sequence is represented using a bitmap by assigning a '1' to the opening parenthesis and a '0' to the closing one. The topology of the tree requires $2n$ bits. Table 2.3 lists the main operations that `BP` is capable of solving in constant time when it is augmented with extra data structures [141]. In practice [14], this representation requires $2.37n$ bits since the bitmap needs to be pre-processed with rank/select operations and additional data structures are required. We refer the reader to the work of Arroyuelo et al. [14] for implementation details.

LOUDS. Level Ordered Unary Degree Sequence [93] compact tree representation is a simpler, yet efficient mechanism to represent ordinal trees. We start with an empty bitmap T and traverse the tree in a level-order fashion starting from the root. As we visit a node v with $d \geq 0$ children we append 1^d0 to T. We need to augment T with RANK and SELECT operations to support some key operations such as PARENT and CHILD in $O(1)$ time. However, `LOUDS` constant-time operations are more limited than the ones supported by `BP`. This representation does not support complex operations such as SUBTREE_SIZE or LOWEST_COMMON_ANCESTOR. In practice, we can use a bitmap with rank/select support using 5% of extra space such as `RG`, requiring $2.10n$ bits. In Table 2.3, we mark the operations that are supported in constant time by the `LOUDS` representation.

DFUDS. Depth-First Unary Degree Sequence [23] is built over the same traversal as balanced parentheses, but when arriving at a node v, it appends d '1's and one '0', being d the number of children of v. This representation allows us to perform almost the same operations as `BP`, except for retrieving the distance from a node to the root. An advantage is that the implementation of the operation to retrieve the i-th child is simpler than in `BP`.

We show an example of these schemes in Figure 2.9. The left part shows an example tree with labeled nodes. The right part shows the corresponding compact representations of the topology and where each node is located.

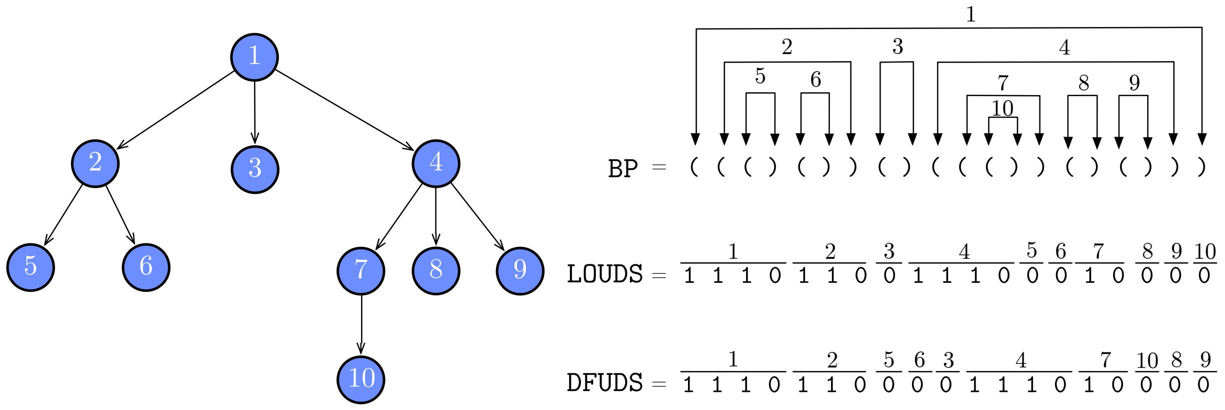

BP = ((( ) ( )) ( ) ((( )) ( ) ( )))
LOUDS = 1 1 1 0 1 1 0 0 1 1 1 0 0 0 1 0 0 0 0
DFUDS = 1 1 1 0 1 1 0 0 0 0 1 1 1 0 1 0 0 0 0

Figure 2.9: Example of compact representation of a tree using `BP`, `LOUDS` and `DFUDS`

Operation	**Description**	`BP`	`LOUDS`	`DFUDS`
ROOT()	root of the tree	✓	✓	✓
FIRST_CHILD(v)	first child of node v	✓	✓	✓
LAST_CHILD(v)	last child of node v	✓	✓	✓
CHILD(v, i)	ith child of node v	✓	✓	✓
CHILDREN(v)	number of children of node v	✓	✓	✓
NEXT_SIBLING(v)	next sibling of node v	✓	✓	✓
PARENT(v)	ith child of node v	✓	✓	✓
IS_LEAF(v)	whether v is a leaf	✓	✓	✓
DEPTH(v)	depth of node v	✓	✗	✗
HEIGHT(v)	distance from root	✓	✗	✗
LCA(u, v)	lowest common ancestor of u and v	✓	✗	✓
PREORDER_RANK(v)	pre order rank of node v	✓	✗	✓
PREORDER_SELECT(v)	pre order select of node v	✓	✗	✓
SUBTREE_SIZE(v)	number of nodes in subtree node v	✓	✗	✓

Table 2.3: List of compact tree operations for different representations. The ✗ mark shows the operations are not solved in constant time. Note that for implementing these operations in constant time we also require constant time implementations of RANK and SELECT operations.

2.11 Differentially Encoded Search Trees

Differentially Encoded Search Trees (DEST) were introduced by Claude el al. [47]. They provide a compact representation of a an integer-valued binary search trees. Every node stores the difference with its parent and, by knowing if the node is a left or right child, one can determine whether the difference is positive or negative. The difference values can be afterwards encoded using any integer encoding mechanism such as δ, γ, Rice, (see Section 2.5) or DAC (see Section 2.8), leading to a smaller representation. DEST allows access to any position i in a sorted sequence in $O(h)$ time, where h is the tree height. Figure 2.10 shows an example of a DEST tree for the sorted sequence $\mathcal{S} = 11, 13, 14, 15, 16, 19, 20$.

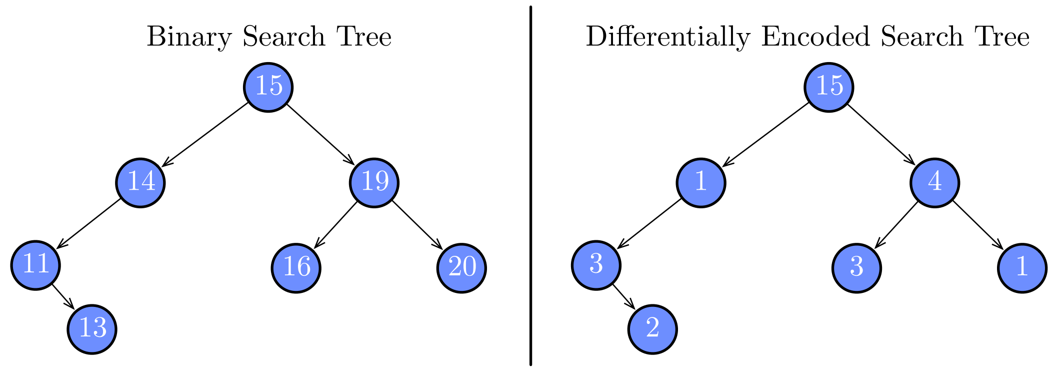


Figure 2.10: Left: the original binary search tree. Right: the differentially encoded search tree.

2.12 Range Maximum Queries

A *Range Maximum Query* (RMQ) over a sequence of integers $\mathcal{S}$ is defined as $\textsc{rmq}_{\mathcal{S}}(i,j) = \text{argmax}_{i \le k \le j} \mathcal{S}[k]$. In other words, $\textsc{rmq}_{\mathcal{S}}(i,j)$ retrieves the position within the range $[i,j]$ in $\mathcal{S}$ where the maximum element is located. Fischer et al. [69] showed that this operation can be solved in constant time and using $2n + o(n)$ bits of space without accessing the sequence $\mathcal{S}$. This data structure can be configured to answer range minimum queries as well.

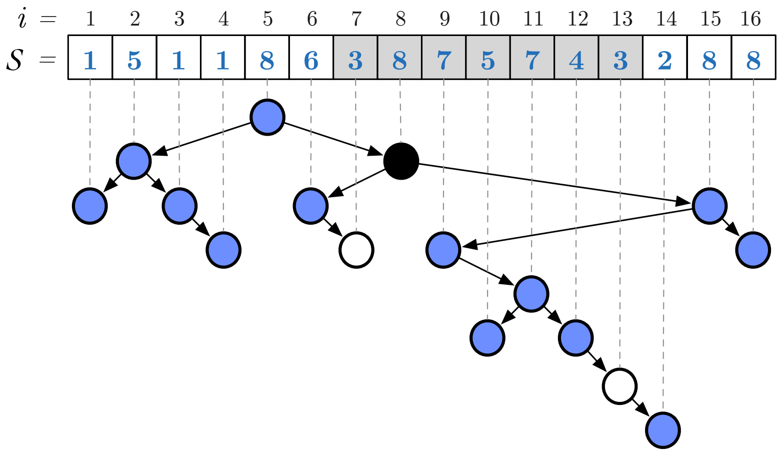
The main idea is to construct a *Cartesian tree* over the sequence of values $\mathcal{S}$ and then perform a tree isomorphism. A Cartesian tree is a binary tree defined as follows: the root of the tree is the leftmost maximum position p in $\mathcal{S}[1,n]$, the left child of the root is the Cartesian tree constructed over $\mathcal{S}[1,p-1]$, and its right child is the Cartesian tree over $\mathcal{S}[p+1,n]$. This recursion is done until the range of the sequence becomes empty. Note that the Cartesian tree of $\mathcal{S}[p,p]$ is a single node. We show an example of the Cartesian tree in Figure 2.11.

The most important property for solving $\textsc{rmq}_{\mathcal{S}}(i,j)$ is that the position of the maximum element in $\mathcal{S}[i,j]$ is the inorder position of lowest common ancestor of v_i and v_j where v_i and v_j are the Cartesian tree nodes with inorder positions i and j respectively. Thus, the formula to compute $\textsc{rmq}_{\mathcal{S}}(i,j)$ is:

$$\textsc{rmq}_{\mathcal{S}}(i,j) = \textsc{inorder\_rank}(\textsc{lca}(\textsc{inorder\_select}(i), \textsc{inorder\_select}(j))). \quad (2.3)$$

We show an example of a Cartesian tree and this procedure in Figure 2.11. Nodes v_i and v_j are marked in white background, while the LCA(v_i, v_j) is marked in black.

Since the operations INORDER_RANK and INORDER_SELECT are not simple to implement it is preferable to use the methods PREORDER_RANK and PREORDER_SELECT over a general tree that is isomorphic to the binary tree [65]. This general tree is constructed as follows: a new root node is added and its children are all the nodes in the rightmost path of the binary tree. This procedure is done recursively considering all these nodes as new roots of the same transformation of their left child's subtree. Using the general tree, the inorder index of a node can be easily mapped to a preorder index using constant time operations implemented using BP. In practice, the tree using the BP compact representation requires $2.3n$ bits.



i = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
$\mathcal{S}$ = 1 5 1 1 8 6 3 8 7 5 7 4 3 2 8 8

Figure 2.11: A Cartesian tree built over the sequence $\mathcal{S}$. The values in the range $[7, 13]$ are highlighted in grey background. The nodes in the Cartesian tree, corresponding to the start position and end position are marked with white background. The lowest common ancestor (LCA) of those nodes points to the maximum value on that range and is highlighted in black.

2.13 Treaps and Priority Search Trees

A *treap* [148] is a binary search tree with nodes having two attributes: *key* and *priority*. The treap maintains the binary search tree invariants for the keys and the heap invariants for the priorities, that is, the key of a node is larger than those in its left subtree and smaller than those in its right subtree, whereas its priority is not smaller than those in its subtree. The treap does not guarantee logarithmic height, except on expectation if priorities are independent of keys [108]. By assigning a random priority to each element to be inserted into a binary search tree, it is possible to use treaps to create randomly-built binary search trees, which tend to be balanced. This is the most common use of treaps in the literature. Nevertheless, a treap can also be regarded as the Cartesian tree [162] of the sequence of priorities once the values are sorted by keys.

The *priority search tree* [109] is somewhat similar, but it is balanced. In this case, the root is not the node with highest priority in its subtree, but that element is stored in addition to the root element at the node. The element stored separately is also removed from the subtree. Priority search trees can be used to solve 3-sided range queries on t-point grids, returning k points in time $O(k + \log t)$. This has been used to add rank query capabilities to several index data structures such as suffix trees and range trees [91].

2.14 Text Indexes

The main text indexes employed in this thesis can be divided into two types: the first type corresponds to the indexes that are designed for the scenarios where any substring is a possible query. In this family of indexes we have the generalized suffix tree, suffix array and

Keys =	**4**	**9**	**12**	**14**	**15**	**16**	**22**	**27**	**30**	**32**	**35**	**37**	**39**	**42**	**44**
Priorities =	**6**	**2**	**14**	**1**	**1**	**1**	**2**	**1**	**24**	**4**	**6**	**1**	**2**	**1**	**3**

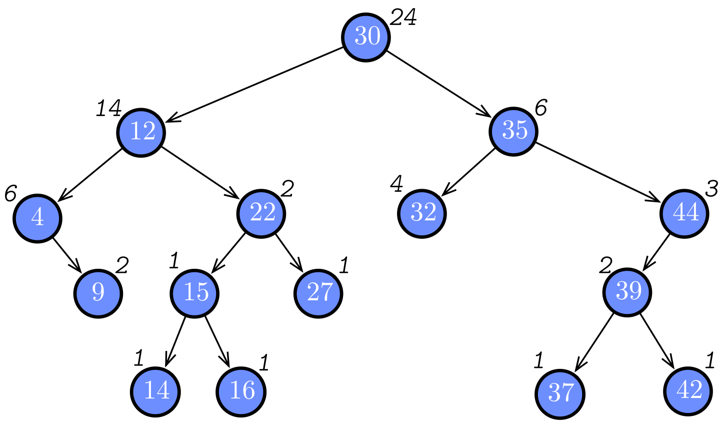


Figure 2.12: Example of a treap over the sequence of keys and priorities (shown at the top of the figure). The numbers inside the nodes correspond to the keys and the numbers outside the nodes represent the priority.

the compressed versions of them, which are regarded as self-indexes. The second type is the inverted index, which is usually employed when the queries are formed by words or terms. In this section, we describe the main properties and operations of these indexes.

2.14.1 Generalized Suffix Tree and Suffix Array

Let $\mathcal{C} = d_1 d_2 d_3 \ldots d_D\$$ be the text obtained by concatenating all the documents in $\mathcal{D}$ (recall Section 2.2). Each substring $\mathcal{C}[i, n]$, with $i \in [1, n]$, is called a *suffix* of $\mathcal{C}$. The *generalized suffix tree* (GST) of $\mathcal{C}$ is a path-compressed trie (i.e., unary paths are collapsed) in which all the suffixes of $\mathcal{C}$ are inserted. Internal nodes correspond to repeated strings of $\mathcal{C}$ and the leaves correspond to suffixes. For internal nodes v, $path(v)$ is the concatenation of the edge labels from the root to v. The suffix tree can list the occ occurrences of a pattern P of length m in optimal $O(m + occ)$ in time, by traversing from the root to the *locus* of P, i.e., the highest node v such that P is a prefix of $path(v)$. Then all the occurrences of P correspond to the leaves of the subtree rooted at v. We will call sp and ep the left-most leaf of node v and ep the right-most. Thus, v is the lowest common ancestor of the sp-th and the ep-th leaves. The suffix tree has $O(n)$ nodes and it uses $O(n)$ words of space. Figure 2.13 shows an example of a generalized suffix tree built over the collection of Figure 2.1. We define $tf_{v,d}$ as the number of leaves associated with document d that descend from suffix tree node v (i.e., the number of times the string label of v appears in document d). In practice, a pointer based implementation of a suffix tree can take up to 20 times the size of the collection, making this indexing data structure impractical for scenarios where large collections need to be handled.

The *suffix array* [107] is a common full-text index that allows us to efficiently retrieve, for an arbitrary pattern, the number of its occurrences and their positions in a given text. The

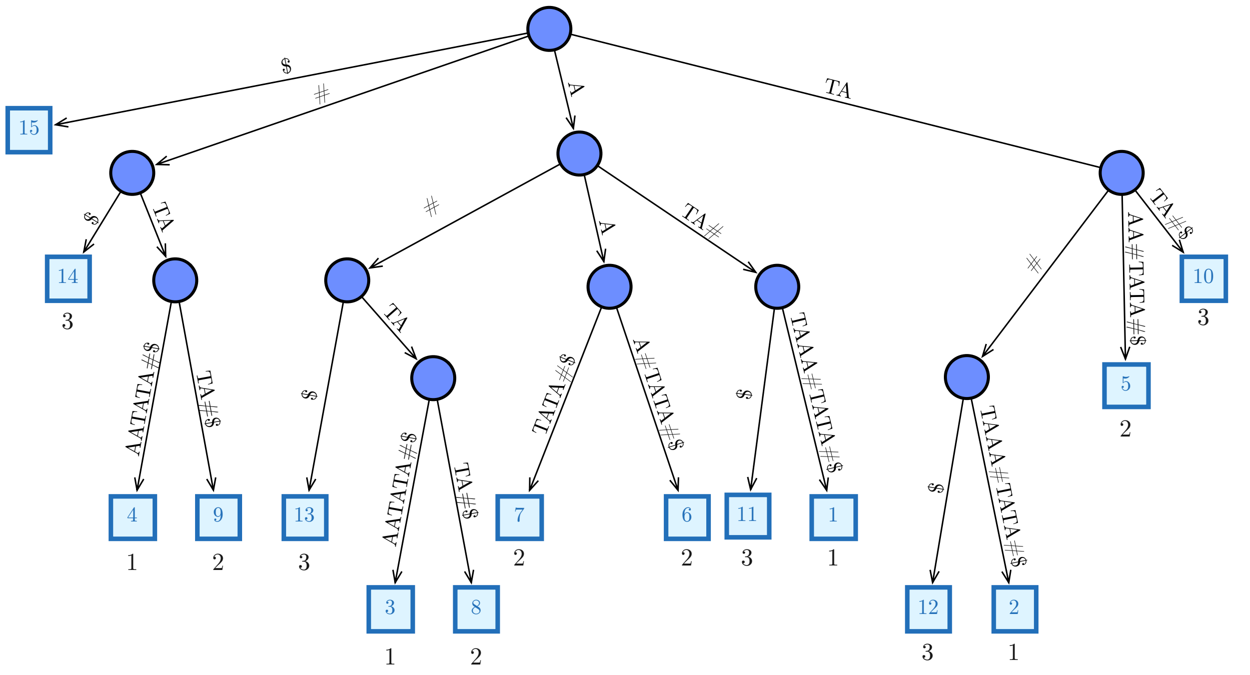


Figure 2.13: Generalized suffix tree for the example sequence $\mathcal{C}$ = ATA#TAAA#TATA#\$. The number inside the leaves points to the positions in $\mathcal{C}$ where the suffix represented by that leaf appears. The document numbers are written below the leaves.

suffix array SA$[1, n]$ contains pointers to every suffix of a sequence $\mathcal{C}$, lexicographically sorted. For a position $i \in [1, n]$, SA$[i]$ points to the suffix $\mathcal{C}[\text{SA}[i], n] = t_{\text{SA}[i]} t_{\text{SA}[i]+1} \ldots t_n$, where t_j is the symbol located at position j, and it holds that the suffix $\mathcal{C}[\text{SA}[i], n]$ is lexicographically smaller than $\mathcal{C}[\text{SA}[i+1], n]$.

To find the occurrences of an arbitrary pattern P of length m, two binary searches are performed to find the maximal interval $[sp, ep]$ such that for every position in SA$[sp \leq i \leq ep]$ the pattern P is a prefix of $\mathcal{C}[\text{SA}[i], n]$, that is, P occurs at the positions SA$[i]$ in $\mathcal{C}$. It takes $O(m \log n)$ time to find the interval. Figure 2.14 shows an example of a suffix array. In practice, a simple implementation of the suffix array requires 4 to 8 times the size of the collection. This reduces the size of the suffix tree, yet it is still prohibitive for handling very large collections.

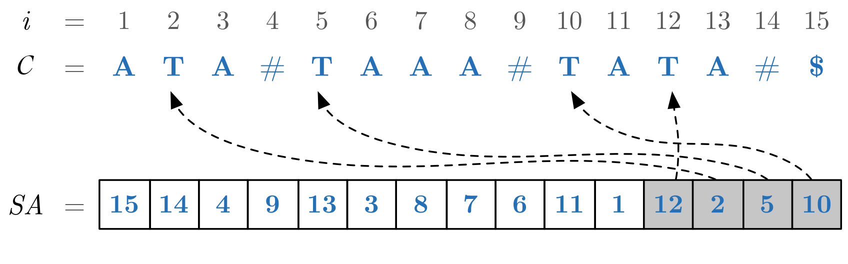

i = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
C = A T A # T A A A # T A T A # $
SA = 15 14 4 9 13 3 8 7 6 11 1 12 2 5 10

Figure 2.14: Suffix array (SA) for the example sequence $\mathcal{C}$ = ATA#TAAA#TATA#\$. The positions SA$[12, 15] = \langle 12, 2, 5, 10 \rangle$ are those where the string "TA" appears.

2.14.2 Self Indexes

The *Compressed Suffix Array* (CSA) [118] can represent the text $\mathcal{C}$ *and* its suffix array SA within essentially $nH_k(\mathcal{C}) \leq n \log \sigma$ bits. This representation allows us to perform the following operations:

- $\textsc{Search}(P)$: determine the interval $[sp, ep]$ corresponding to a pattern P.
- $\textsc{Count}(P)$: return the number of occurrences of P (i.e., $ep - sp + 1$).
- $\textsc{Locate}(i)$: compute SA$[i]$ for any i.
- $\textsc{Extract}(l, r)$: rebuild any $\mathcal{C}[l, r]$ of the original text.

Since the CSA is able to extract the original text from the index, it is considered as a replacement of the original text, and thus called a *self-index*. A class of CSAs [81, 139, 82], can compute SA$[i]$ in $O(\log^{\epsilon} n)$ for any constant $\epsilon > 0$ and is able to search for the pattern and count its occurrences in time $O(m \log n)$. Another class, called *FM-Index* [67, 68] computes SA$[i]$ in $O(\log^{1+\epsilon} n)$ time (in practice $O(\log n)$), and searches for P in time $O(m \frac{\log \sigma}{\log \log n})$ (in practice, $O(m \log \sigma)$). We refer the reader to the survey of Navarro et al. [118] and Ferragina et al. [66] for a deeper analysis and evaluation of these families of solutions.

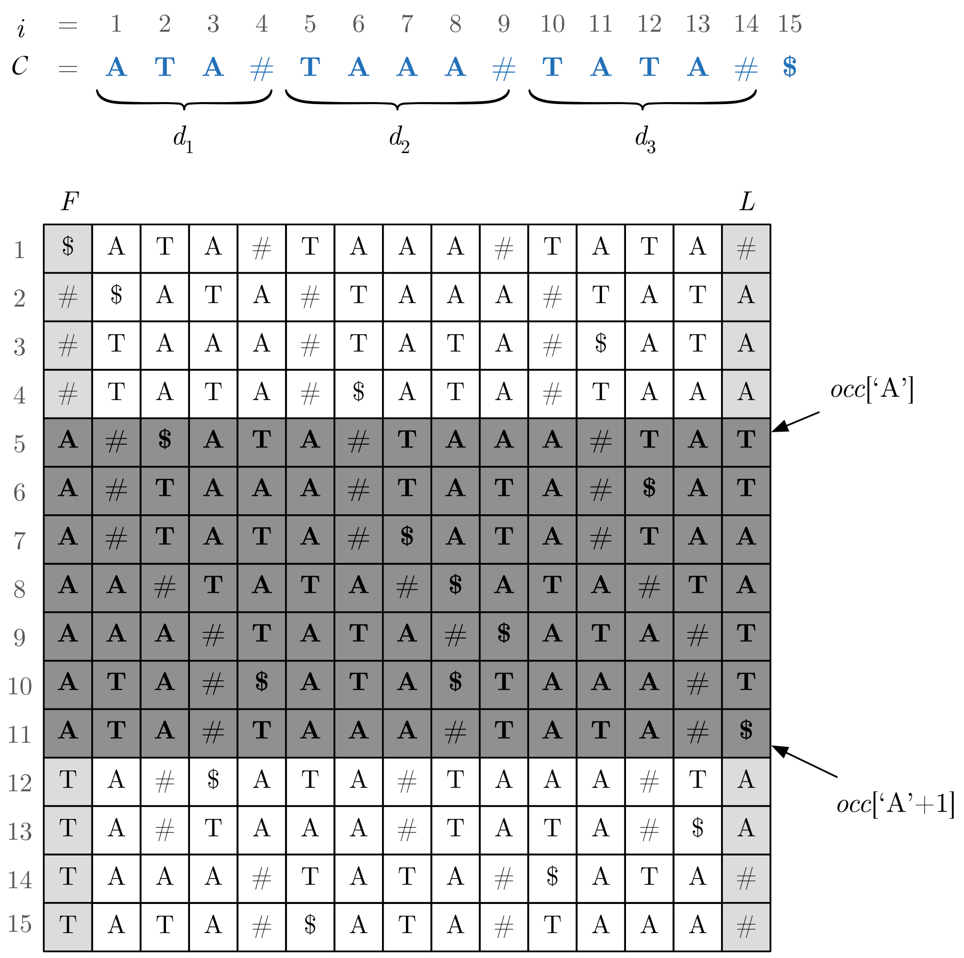
We proceed to briefly explain how the *FM-Index* family of indexes works, and we will refer to them as *Succinct Suffix Arrays* (`SSA`). The `SSA` [118] is a compressed index that builds on the *Burrows-Wheeler transform* (BWT) of a text [38]. The main idea is to represent the BWT of a collection of texts in compressed space and use it to support pattern matching operations. We start by describing the BWT and the LF operation, one of the main blocks for the operations supported by the index. Then we describe how to solve the three main operations: count, locate and extract.

Given a text $\mathcal{C}$, the BWT is a permutation of the symbols in $\mathcal{C}$ that is reversible. A simple way to describe the transformation is to imagine the $n \times n$ matrix of the n cyclic rotations of the text $\mathcal{C}$, sort the rows lexicographically, and then keep the last column of the matrix.

An example of this is shown in Figure 2.15. We refer to the first column as F and the last one as L or $BWT(T)$.

Local compressors tend to handle the BWT of a text much better than the text itself, since the transformation tends to cluster together occurrences of the same symbol. This is not surprising, since the symbols are actually arranged according to their context (symbols appearing after it). This way, the BWT is able to represent the text achieving kth order entropy (recall Section 2.3). An interesting operation that allows us to support the ones we are interested in is known as LF-mapping. LF(i) retrieves the symbol before L$[i]$ in $\mathcal{C}$ by retrieving the position where L$[i]$ occurs in F (hence the name LF). The LF operation can be computed as LF$(i) = \textsc{rank}_c(\mathrm{L}, i) + occ[c]$, where $occ[c]$ is a fixed array storing the number of occurrences in T of symbols lexicographically smaller than the symbol c. By representing L with a wavelet tree, the LF operation takes $O(\log \sigma)$ time, the same as accessing any position in L.

The *backward search* operation returns the range $[sp, ep]$ where all occurrences of a given


i = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
C = A T A # T A A A # T A T A # $
d_1
d_2
d_3
occ[‘A’]
occ[‘A’+1]

	F														L
1	$	A	T	A	#	T	A	A	A	#	T	A	T	A	#
2	#	$	A	T	A	#	T	A	A	A	#	T	A	T	A
3	#	T	A	A	A	#	T	A	T	A	#	$	A	T	A
4	#	T	A	T	A	#	$	A	T	A	#	T	A	A	A
5	**A**	**#**	**$**	**A**	**T**	**A**	**#**	**T**	**A**	**A**	**A**	**#**	**T**	**A**	**T**
6	**A**	**#**	**T**	**A**	**A**	**A**	**#**	**T**	**A**	**T**	**A**	**#**	**$**	**A**	**T**
7	**A**	**#**	**T**	**A**	**T**	**A**	**#**	**$**	**A**	**T**	**A**	**#**	**T**	**A**	**A**
8	**A**	**A**	**#**	**T**	**A**	**T**	**A**	**#**	**$**	**A**	**T**	**A**	**#**	**T**	**A**
9	**A**	**A**	**A**	**#**	**T**	**A**	**T**	**A**	**#**	**$**	**A**	**T**	**A**	**#**	**T**
10	**A**	**T**	**A**	**#**	**$**	**A**	**T**	**A**	**$**	**T**	**A**	**A**	**A**	**#**	**T**
11	**A**	**T**	**A**	**#**	**T**	**A**	**A**	**A**	**#**	**T**	**A**	**T**	**A**	**#**	**$**
12	T	A	#	$	A	T	A	#	T	A	A	A	#	T	A
13	T	A	#	T	A	A	A	#	T	A	T	A	#	$	A
14	T	A	A	A	#	T	A	T	A	#	$	A	T	A	#
15	T	A	T	A	#	$	A	T	A	#	T	A	A	A	#

Figure 2.15: Matrix of all the cyclic rotations of “ATA#TAAA#TATA#$”. The last column corresponds to the BWT (also called L or $BWT(T)$). We call the first column F.

pattern P of length m are located within the text in $O(m \log \sigma)$ time. It is called backward search since its procedure seeks the pattern in reverse order. Backward search can also compute COUNT(P) by just performing $ep - sp + 1$. In order to retrieve the original text from a range $[sp, ep]$ we must perform an EXTRACT operation, which takes $(s_a + ep - sp + 1)(\log \sigma)$ time, where s_a is the *sampling factor* of the `SSA` index, which is a parameter that produces different time/space trade offs. The space of the `SSA` grows depending on this sampling factor, so the extra space is $O(n \log n / s_a)$ bits. This index is able to perform these operations and replace the collection, requiring $nH_k(\mathcal{C}) + o(n \log \sigma)$ bits of space, by choosing a sampling factor of $s_a = \frac{\log^{1+\epsilon} n}{\log \sigma}$.

To illustrate this process we first show how to find the range $[sp, ep]$ in the SA array for a two-letter pattern “TA” using the example collection shown in Figure 2.15. To find “TA” we start with the last symbol ‘A’. Every occurrence of ‘A’ in L is a potential candidate, and we want to find those whose LF lead to a ‘T’, since those will correspond to occurrences of “TA”. The process starts by considering all elements as candidates, and moving to a range where we know that all the symbols are followed by ‘A’. This range corresponds to $[sp = occ[\text{‘A’}] = 5, ep = occ[\text{‘A’+1}]$, marked in Figure 2.15. Within this range we want to retrieve the elements whose value is ‘T’ and move to their corresponding place in F. We can compute this range as $[occ[\text{‘T’}]+\text{RANK}_\text{T}(L, sp = 5), occ[\text{‘T’}]+\text{RANK}_\text{T}(\text{L}, ep = 11)]$, obtaining the ‘T’ in position 12 in F, where we can see the pattern “TA” at the beginning of that cyclic rotation. The complete procedure is described in Algorithm 4.

Algorithm 4: Method BACKWARD_SEARCH(P, m, occ, n). Returns the range $[sp, ep]$ in SA where pattern P of length m occurs.

```
BACKWARD_SEARCH($P, m, occ, n$)
  $sp \leftarrow 1$, $ep \leftarrow n$
  for $i \leftarrow m$ to 1 do
    $c \leftarrow P[i]$
    $sp \leftarrow occ[c]+\text{RANK}_c(L, sp-1)+1$
    $ep \leftarrow occ[c]+\text{RANK}_c(L, ep)$
    $i \leftarrow i-1$
    if $ep < sp$ then
      return $\emptyset$
  end for
    return $[sp, ep]$
```

Algorithm 5: Method LOCATE(i) on the `SSA`, returns SA$[i]$.

```
LOCATE($i$)
  $i' \leftarrow i, t \leftarrow 0$
  while SA[$i'$] is not explicitly stored do
    $i' \leftarrow LF[i']$
    $t \leftarrow t+1$
  end while
  return SA[$i'$] + $t$
```

In order to retrieve a certain position in the suffix array SA$[i]$ using the `SSA`, we need to store absolute samples (s_a) at regular intervals to permit the efficient decoding of any (i). The idea is to perform the LF-mapping operation until we find a sampled value of SA$[i']$, and then report SA$[i']+t$ where t is the number of backward steps used to find such i'. Algorithm 5 shows this procedure.

To perform the EXTRACT(l, r) operation, we use the same sampling mechanism used for LOCATE. In this case we store the positions i such that SA$[i]$ is a multiple of s_a, now in the text order. The idea is to find the first sample that follows the area of interest, that is $\lceil (r+1)/s_a \rceil$. From here we get the desired text backwards following the same procedure as described before. EXTRACT requires at most $s_a + r - l + 1$ applications of the LF-mapping.

2.14.3 Inverted Indexes

The inverted index is an old and simple, yet efficient, data structure that is at the heart of every modern information retrieval system, and plays a central role in any book on the topic [166, 40, 17, 50]. Given a collection of text containing a set of D documents $\mathcal{D} = \{d_1, d_2, \ldots, d_D\}$. A document d_i can be regarded as a sequence of terms or words and the number of words in the document is denoted by $|d_i|$. The total length of the collection is then $\sum_{i=1}^{D} |d_i| = n$. Each document has a unique *document identifier* (*docid*) $\in [1, D]$. The set of distinct terms in the collection is called the *vocabulary*, which is comparatively small in most cases [84], more precisely of size $O(n^\beta)$, for some constant $0 < \beta < 1$ that depends on the text type. The inverted index can be seen as an array of *lists* or *postings*, where each entry of the array corresponds to a different term or word in the vocabulary of the collection, and the lists contain one element per distinct document where the term appears. For each term, the index stores in the list the *document identifier* (*docid*), the *weight* of the term in the document and, if needed, the positions where the term occurs in the document. The weight of the term in the document is a utility function that represents the importance of that word inside the document. The main components of the inverted index are then defined as the *vocabulary* and the *inverted lists*:

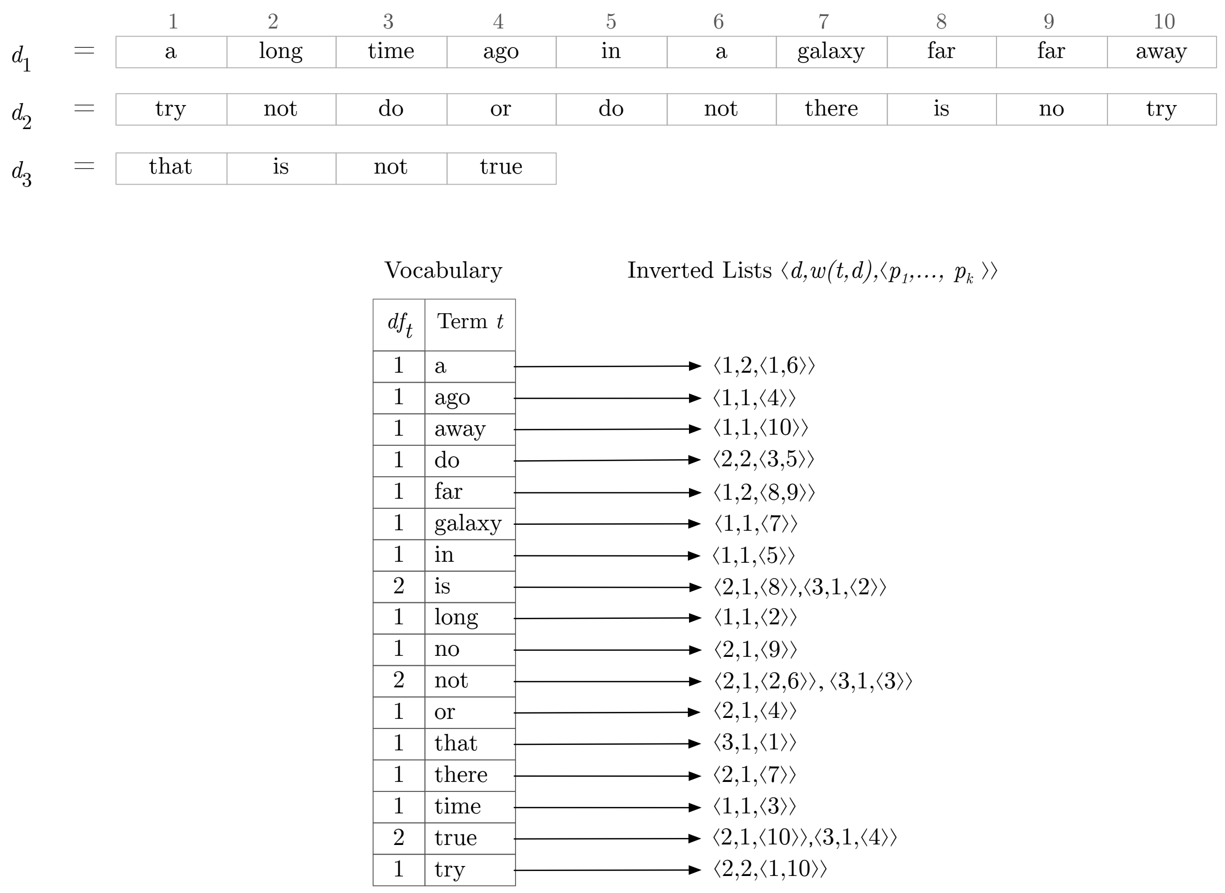
- **The Vocabulary.** The vocabulary stores all distinct terms contained in the collection of documents $\mathcal{D}$. This is commonly implemented with a dictionary data structure such as hash tables or tries. In practice, the vocabulary stores two elements associated with each term: an integer value df_t, that is, the amount of documents that contain the term t, and a pointer to the start of its corresponding *inverted list*.
- **Inverted Lists.** A *non-positional inverted list* stores a list of elements containing pairs $\langle d, w(t,d)\rangle$, where d is the document identifier (*docid*) and $w(t,d)$ is a relevance measure of the term t in document d. A *positional inverted list* contains a list of triples $\langle d, w(t,d), \langle p_1, p_2, \cdots, p_k\rangle\rangle$, where the third component is a vector of positions where the occurrences term t are located in the document d. An inverted index that makes use of positional inverted lists is also commonly referred to as *full-text* inverted index.

Figure 2.16 shows a example of a positional inverted index for a collection consisting of three documents. The inverted index from the example shows a *docid-sorted* organization, where each posting list is in increasing docid order. The posting lists could also follow a *weight-sorted* organization

2.15 Muthukrishnan's Algorithm

The inverted index is not suitable for the case where the query is any possible text substring and we want to retrieve the document identifiers that contain a string pattern P. This is a common scenario for text collections written in East Asian languages such as Chinese or Korean, where words are difficult to segment, or in agglutinating languages such as Finnish and German, where one needs to search for particles inside words.

Muthukrishnan [115] introduced the first solution for retrieving the documents containing



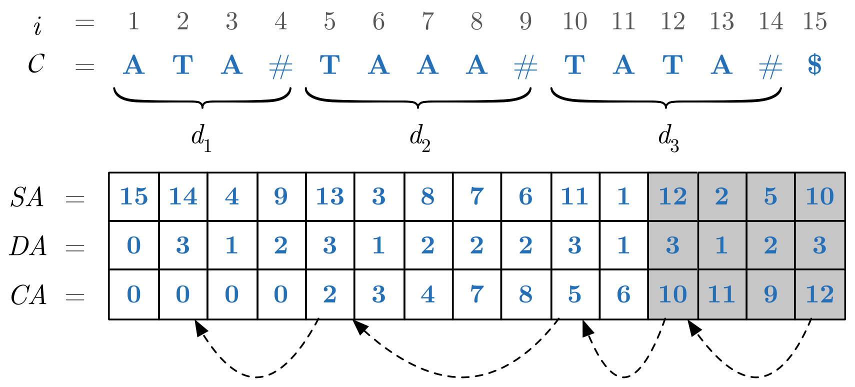
		1	2	3	4	5	6	7	8	9	10
d_1	=	a	long	time	ago	in	a	galaxy	far	far	away
d_2	=	try	not	do	or	do	not	there	is	no	try
d_3	=	that	is	not	true						

Vocabulary — Inverted Lists $\langle d, w(t,d), \langle p_1, \ldots, p_k \rangle\rangle$

df_t	Term t	
1	a	⟨1,2,⟨1,6⟩⟩
1	ago	⟨1,1,⟨4⟩⟩
1	away	⟨1,1,⟨10⟩⟩
1	do	⟨2,2,⟨3,5⟩⟩
1	far	⟨1,2,⟨8,9⟩⟩
1	galaxy	⟨1,1,⟨7⟩⟩
1	in	⟨1,1,⟨5⟩⟩
2	is	⟨2,1,⟨8⟩⟩,⟨3,1,⟨2⟩⟩
1	long	⟨1,1,⟨2⟩⟩
1	no	⟨2,1,⟨9⟩⟩
2	not	⟨2,1,⟨2,6⟩⟩, ⟨3,1,⟨3⟩⟩
1	or	⟨2,1,⟨4⟩⟩
1	that	⟨3,1,⟨1⟩⟩
1	there	⟨2,1,⟨7⟩⟩
1	time	⟨1,1,⟨3⟩⟩
2	true	⟨2,1,⟨10⟩⟩,⟨3,1,⟨4⟩⟩
1	try	⟨2,2,⟨1,10⟩⟩

Figure 2.16: An example positional inverted index built over the collection of documents at the top. The vocabulary and the posting lists built over the collection are shown on the bottom part.

a pattern P in a collection of documents, in optimal time and linear space. The idea is based on suffix arrays and introduces a new data structure called *document array* DA$[1, n]$. The algorithm starts by constructing the suffix array SA$[1, n]$ from the concatenation $\mathcal{C}$ of a collection of documents $\mathcal{D}$, and for every position pointed from SA$[i]$ it stores in DA$[i]$ the *docid* where the suffix came from. The document array requires $n \log D$ bits. In order to list all the distinct documents from the range SA$[sp, ep]$ of the occurrences of P (recall Section 2.14.1), Muthukrishnan lists all the distinct docid values in DA$[sp, ep]$. The idea is to use another array CA$[1, n]$ where CA$[i] = \max\{j < i,\ \text{DA}[j] = \text{DA}[i]\} \cup \{0\}$, which is preprocessed for range minimum queries (recall Section 2.12). Each value CA$[m] < sp$ for $sp \leq m \leq ep$ is a distinct value DA$[m]$ in DA$[sp, ep]$. A range minimum query in CA$[sp, ep]$ gives one such value m, and we continue recursively on DA$[sp, m-1]$ and DA$[m+1, ep]$ until the minimum is $\geq sp$. Figure 2.17 shows the corresponding document array (DA) and array CA for the example collection $\mathcal{C}$. Algorithm 6 shows the procedure to list all the distinct elements of DA$[sp, ep]$.



i = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
C = A T A # T A A A # T A T A # $
d_1
d_2
d_3
SA = 15 14 4 9 13 3 8 7 6 11 1 12 2 5 10
DA = 0 3 1 2 3 1 2 2 2 3 1 3 1 2 3
CA = 0 0 0 0 2 3 4 7 8 5 6 10 11 9 12

Figure 2.17: Document array (DA) and array CA example for the sequence shown in Figure 2.1. The positions SA[12, 15] = $\langle 12, 2, 5, 10\rangle$ are those where the string "TA" appears. The dotted arrows represent a list of pointers to the next smaller position in CA where d_3 occurs.

Algorithm 6: Algorithm for retrieving distinct documents from DA. Receives as parameter the start position *sp*, end position *ep*, and a copy of the original starting position *gsp*. Returns a set containing the distinct documents in the range of $DA[sp, ep]$.

```
      DOCUMENT_LISTING($sp, ep, gsp$)
01      if $ep < sp$ then
02          return $\emptyset$
03      $p \leftarrow$ RMQ($sp, ep$)
04      if CA[$p$] < $gsp$ then
05          $doc$ = DA[$p$]
07          return $\{doc\}$ ∪ DOCUMENT_LISTING($sp, p-1, gsp$)
08                          ∪ DOCUMENT_LISTING($p+1, ep, gsp$)
09      return $\emptyset$
```

2.16 Information Retrieval Concepts

In this section we describe three main concepts related to IR that are of interest for this thesis.

2.16.1 Scoring

The popular *document vector* model is considered, in the IR domain [17, 40, 50], as a pragmatic tool for document retrieval. In this model, each document in a corpus is represented by a vector, where components are associated with specific dimensions representing a property or attribute. Usually, a numeric value is assigned to the attribute, for instance, when the attribute indicates the number of times a word appears in the document. In other cases, a vector entry may be a bit value where the value 1 indicates that a property is present in the document, or 0 otherwise. A document vector model represents any document d_i as a vector $\vec{d_i} = (w(t_1, d_i), ..., w(t_k, d_i))$ where t_j is a term (usually a word) and the function $w(t_j, d_i)$ is the weight of the term t_j in the document d_i. The weight of the term is a utility function that represents the importance of that word inside the document. Another model is the *bag-of-words*. In this model, a collection of documents is represented as a set (*bag*) of the words contained in the collection, not taking into consideration the word ordering, grammar or other properties from the text.

Most IR systems that are based on these models define the score of a document d for a query Q as

$$score(Q, d) = \sum_{t \in Q} w(t, d), \tag{2.4}$$

where the query Q is a set of terms and thus considered as a "bag of words".

For example, in the well-known tf-idf scoring scheme, the weight is computed as $w(t, d) = tf_{t,d} \cdot idf_t$. Here, $tf_{t,d}$ is the *term frequency* of t in d, that is, the number of times t occurs in d. The second term is $idf_t = \log \frac{D}{df_t}$, where df_t is the *document frequency*, that is, the number of documents where the term t appears, and D is the total number of documents. Since idf_t (or df_t) depends only on t, an efficient way to store $w(t, d)$ in an inverted index is to store idf_t or df_t together with each distinct vocabulary term, and store the values $tf_{t,d}$ in the posting list of term t, together with each docid d. Recent state- of-the-art information retrieval systems employ more complex ranking formulas such as the Okapi BM25 [95]:

$$score(Q, d) = \sum_{t \in Q} \frac{tf_{t,d} \cdot (k_1 + 1)}{tf_{t,d} + k_1 \cdot (1 - b + b \cdot |d|/avg\_dlength)} \cdot idf_t, \tag{2.5}$$

where k_1 and b are tuning parameters and $avg\_dlength$ is the average document length in the collection $\mathcal{D}$.

Computing the ranking score can be a major bottleneck of the system's query processing routines. One idea to improve speed is to pre-compute the scores and store them as floating-point numbers in 24 to 32 bits. This alternative is usually not possible, since it increases the size of the index significantly. If we want to speed up the score calculation process and also maintain a reasonable size of the index we can *discretize* the range of possible scores into a predefined set of buckets. Anh et al. [12] proposed a uniform quantization method that is an index-wide linear scaling of the term weights, in other words, the idea is to precompute and store the *impact score* for the term weight as $I(d, t)$ using the following formula:

$$I(t, d) = \left\lfloor \frac{w(t, d) - min(w(t, d))}{max(w(t, d)) - min(w(t, d))} \times 2^b \right\rfloor, \tag{2.6}$$

where b is the number of bits that we are willing to reserve for each score contribution. Anh et al. [9] showed that by setting $b = 8$, quantization methods achieves effectiveness that is indistinguishable from using exact term weights, this has been also been further explored in the work of Anh and Moffat [10, 12].

2.16.2 Top-k Query Processing and Two-stage Ranking Process

In the case of natural language collections, we define a query Q as a set of q words or terms $t \in Q$. In the case of general sequences, we define the query as a pattern P, which is a sequence of symbols from an alphabet Σ of size σ and its length is $|P| = m$.

The problem with retrieving a fixed amount of k documents with the highest "score" to a query Q (natural language case) or a pattern P (general sequence case) for some definition of score, is known as the top-k document retrieval problem. This problem is of interest for the two-stage ranking process that is performed in modern Web search engines and other IR systems. The two-stage ranking process is a technique that copes with the huge amounts of data and enables IR systems to deliver precise results in response to the user queries [163, 40]. In the first stage, a fast and simple filtration procedure extracts a subset of a few hundred or thousand of candidates from the possibly billions of documents forming the collection. In the second stage, more complex learned ranking algorithms are applied to the reduced candidate set in order to obtain a handful of high-quality results. In general, improving the efficiency of the first stage frees more resources for the second stage, and this way increases the overall performance. In contexts where traditional ranking methods are sufficient, the goal of the first stage is to directly convey a few top-quality results to the final user.

In distributed main-memory systems, documents are usually distributed across indexes, and each index contributes with a few results to the final top-k list. Therefore, it is most interesting that individual indexes solve top-k queries efficiently for k values in the range 10–100 [40].

Part II

Inverted Indexes

Chapter 3

Query Processing and Compression of Posting Lists

In this chapter we survey the main techniques to perform top-k query processing on inverted indexes. As previously discussed in Section 2.16.2, complex IR systems requirements are handled via a two-stage ranking process [163, 40]. The first stage aims to return a set of the highest ranked documents containing either all the query terms (a *ranked intersection*) or some of the most important query terms (a *ranked union*). In most cases, ranked intersections are solved via a Boolean intersection followed by the computation of scores for the resulting documents. Ranked unions are generally solved only in approximate form, avoiding a costly Boolean union. In order to avoid a complete traversal of all lists, there are query processing techniques that make use of *early termination* mechanisms. Different organizations of the posting lists have been proposed and it has been shown that they can assist early termination mechanisms. Moreover, there are different ways to traverse the posting lists to improve query processing. We cover these concepts in this chapter.

Another important issue is compression. The inverted index can be stored on disk or in main memory, and in both cases reducing its size is crucial. On disk, it reduces transfer time when reading the lists of the query terms. In main memory, it increases the size of the collections that can be managed within a given RAM budget, or alternatively reduces the number of servers that must be allocated in a cluster to hold the index, the energy they consume, and the amount of communication. In this chapter, we also cover compression techniques that are typically employed to reduce the size of the index

3.1 Query Processing

Query processing involves a number of phases such as query parsing, query rewriting and the computation of complex machine-learned ranking functions that may include hundreds of features from the documents. At the same time, IR systems need to serve as many queries as possible, needing to execute huge amounts of queries per second. To do this efficiently, search engines commonly separate the ranking process into two or more phases. In the first phase, a simple and fast ranking function such as BM25 is used to get the top 100 or 1000 documents. Then in the second and following phases, increasingly more complicated ranking functions with more features are applied to documents that pass through the earlier phases. Thus, the later phases only examine a fairly small number of candidates, therefore a significant amount of computation is spent in the first phase.

3.1.1 Early Termination

When processing simple ranked queries on large document collections, the main performance challenge is caused by the lengths of the inverted lists and the large number of candidates that pass the Boolean filter. Early termination is a set of techniques that address this problem. We say that a query processing algorithm is exhaustive if it fully evaluates all the documents that satisfy the Boolean filter condition. Any other algorithm uses early termination (ET). In general, early termination techniques can be categorized by the use of one or more of the following approaches:

1. Stopping early. Each inverted list is arranged from the most to the least promising posting and traversal is stopped once enough good results are found.
2. Skipping. Inverted lists are sorted by docids, so promising documents are spread out over the lists. The idea is to skip over uninteresting parts of a list using additional information.
3. Partial scoring. Documents are partially or approximately scored.

3.1.2 Index Organization

Early termination techniques may require inverted index structures to be arranged in specific ways. In particular, some techniques assume a specific organization such that the most promising documents appear early in the inverted lists, which can be done by either reordering the postings in a list, or partitioning the index into several layers. In general, most query processing techniques use one of the following index organizations [59]:

- **Document-Sorted Indexes:** The standard approach for exhaustive query processing, where postings in each inverted list are sorted by docid.
- **Frequency-Sorted Indexes:** Postings in each list are sorted by their frequency.
- **Impact-Sorted Indexes:** Postings in each list are sorted by decreasing impact (or term score), that is, their contribution to the document score.

3.1.3 Query Processing Strategies

There are different query evaluation strategies that exhibit affinities to different index organizations. They are classified in three different categories: Document-at-a-time (DAAT), Term-at-a-time (TAAT) and Score-at-a-time (SAAT) [159].

Document-at-a-time. DAAT processing is more popular for Boolean intersections and unions. Given a query containing q terms, all lists are processed in parallel, looking for the same document in all of them. Posting lists must be sorted by increasing docid, and we keep a pointer to the current position in each of the q lists. Once a document is processed, the pointers move forward. Much research has been carried out on Boolean intersections [57, 18, 145, 51, 20]. While DAAT processing is always used to intersect two lists, experimental results suggest that the most efficient way to handle more lists is to intersect the two shortest ones, then the result with the third list, and so on [20].

Term-at-a-time. TAAT processes one posting list after the other. The lists are considered from shortest to longest, starting with the first one as a candidate answer set, and refining it as we consider the next lists. TAAT is especially popular for processing ranked unions [131, 12, 152, 111], as the successive lists have decreasing idf_t value and thus a decreasing impact on the result, not only for the tf-idf scoring, but also for BM25 and other scorings. Thus, heuristic thresholds can be used to obtain an approximate ranked union efficiently by pruning the processing of lists earlier, or avoiding complete posting lists, as we reach less relevant documents and our candidate set becomes stronger [131, 12]. A more sophisticated approach based on similar ideas can be used to guarantee that the answer is exact [152]. TAAT approaches usually make use of an *accumulator* data structure that holds the intermediate scores for candidate documents.

Score-at-a-time. SAAT mechanism can be seen as a hybrid between DAAT and TAAT in which multiple index lists are open. This strategy opens all index pointers simultaneously, and start reading from the posting list that has the score with the largest *impact* [12]. In Impact-ordered indexes the actual score contributions of each term are precomputed and quantized into what are known as *impact scores* [9, 10, 12]. The idea is that a complete impact block is processed at each step, and the results are obtained using a set of accumulator variables.

3.2 Algorithms for Top-k Ranked Retrieval

Boolean query processing is probably the simplest form of resolving queries. A query with the form "*force* **AND** *awakens*" is dubbed *intersection* or *conjunctive query*, while the query "*episode* **OR** *viii*" is referred to as *union* or *disjunctive query.* In its basic form, union queries are solved by linearly traversing all posting lists from all terms involved in the corresponding query and marking all documents that have those terms. On the other hand, intersection queries have been a field of study for decades [20, 52, 96]. It can be proved that the optimal intersection algorithm for two sets of length m and n with $m \leq n$ is $O(m \log \frac{n}{m})$. The most popular alexgorithm for solving intersections is dubbed *Set Versus Set*, or also *Small Versus*

Small, which is shown on algorithm 7 (adapted from Barbay et al. [20]). The operation of searching for a candidate element e in a range (line 8, in the Algorithm) is commonly solved by a combination of regular sampling and linear, exponential, or binary search. We refer the reader to the work of Barbay et al. [20], Culpepper et al.[52] and Kane et al.[96] for an experimental comparison of intersection algorithms.

Algorithm 7: SVS(set, q). Receives q sets of integers and returns the intersection.

```
      SVS($set, q$)
01      sort the sets by size ($|set[1]| \leq |set[2]| \leq \ldots |set[q]|$)
02      the smallest set ($set[1]$) is the candidate answer
03      **for each** $s \in set$ **do**
03        $R[s] \leftarrow 0$
04      **end for**
05      **for each** $s \in set[2..q]$ **do**
06        **for each** $e \in set[1]$ **do**
07          search $e$ in set $s$ in the range $R[s]$ to $|s|$
08          **if** $e$ not found **then**
09            remove $e$ from $set[1]$
10          **end if**
10          update $R[s]$ to the rank of $e$ in $s$
13          **end for**
13        **end for**
13      **return** $set[1]$
```

Many ranked intersection strategies employ a full Boolean intersection followed by a post processing step for ranking. However, recent work has shown that it is possible to do better [59]. The advantage of DAAT processing is that once we have processed a document, we have complete information about its score, and thus we can maintain a current set of top-k candidates whose final scores are known. This set can be used to establish a threshold on the scores other documents need to surpass to become relevant for the current query. Thus the emphasis on ranked DAAT is not on terminating early but on skipping documents. This same idea has been successfully used to solve exact (not approximate) ranked unions before. [36, 59].

The following two approaches have recently displayed the best performance for exact ranked intersections and unions.

3.2.1 Weak-And (WAND)

WAND stands for *Weak And* or *Weighted AND*, which is a Boolean predicate that was introduced by Broder et al. [36]. WAND takes as arguments a list of boolean variables $x_1, x_2, \ldots, x_k$ a list of positive weights associated with each boolean variable $w_1, w_2, \ldots, w_k$ and a threshold parameter θ. Based on their definition [36] WAND$(x_1, w_1, \ldots, x_k, w_k, \theta)$ is true if and only if:

$$\sum_{1 \le i \le k} x_i w_i \ge \theta, \tag{3.1}$$

where x_i is a boolean variable containing a 1 or 0. Depending on the value of the parameter θ WAND can be used to implement boolean conjunctive (AND) operator and disjunctive (OR) operator. For example, the conjunctive case is the following:

$$\text{AND}(x_1, x_2, \ldots, x_k) = \text{WAND}(x_1, 1, x_2, 1, \ldots, x_k, 1, k), \tag{3.2}$$

while the disjunctive case is:

$$\text{OR}(x_1, x_2, \ldots, x_k) = \text{WAND}(x_1, 1, x_2, 1, \ldots, x_k, 1, 1). \tag{3.3}$$

Note that these cases are for strict boolean operations, since the weight of each boolean variable is always 1. If we want to extend the WAND operator to solve ranked queries, we can do so by assigning a weight associated to each boolean variable.

Any term t in an inverted index is associated to an upper bound U_t as its maximal contribution to any document. We denote this upper bound as $U_t = max(w(t, d_1), \ldots, w(d_{|d|}))$. The idea is now to evaluate each document d with the WAND operator as

$$\text{WAND}(x_1, U_1, x_2, U_2, \ldots, x_q, U_q, \theta),$$

where x_i indicates if the query term i is contained in document d and the threshold θ is the minimum score among the top-k results found so far.

Using this setup, we can employ WAND query processing algorithm to perform ranked intersections and unions following a DAAT strategy. The algorithm consists in three phases: pivot selection, alignment check, and evaluation. Each list contains a pointer to the current docid, starting from the smallest one. The lists are sorted by smallest docid of these pointers and each list is augment by storing the highest score of the list (U_t). The procedure starts by selecting a pivot term. This is done by summing the maximum score of the participating lists of the query until the the sum is larger or equal than a threshold θ. The term where this happens is selected as the pivot. The key observation is that there is no list pointer to a docid smaller than the pivot's current docid that could make it into the top-k results, unless it was already considered earlier. Next, WAND will try to align all posting lists to the pivot's current docid by moving the pointers in these lists forward. If this step succeeds (the docid was found in the other lists), then this docid is evaluated and possibly inserted in the top-k heap. For solving conjunctive queries, all terms should contain the corresponding docid, if not, we proceed by moving the pointer in the pivot list forward and start a new iteration. In the case of disjunctive queries, we do not need all terms to contain the current pivot's docid. This procedure is shown in Algorithm 8 (adapted from the work of Petri et al. [132]). The first phase of the algorithm, pivot selection, is shown in lines 7–19. The alignment check phase is shown in Algorithm 9 in lines 11–13. The evaluation phase is shown in lines 4–9 in Algorithm 9. Algorithm 10 shows the procedure to insert evaluated documents into a min-priority queue. A posting list of a term t is represented in the following algorithms as p_t. In the case of WAND, p_t has the following operations or attributes: *max_score* is an attribute that stores the maximum score in the list, method BEGIN() obtains the first posting for term t, method SEEK_DOCUMENT(DOC) moves the pointer of the current $\langle docid, weight \rangle$ to the first value in p_t that is equal or greater than *doc*. Finally, method NEXT() moves the pointer of posting list p_t to the next $\langle docid, weight \rangle$.

Algorithm 8: WAND(q, k, p). Receives a query containing q posting lists, the number of documents to be returned k and the posting lists p.

```
WAND($q, k, p$)
  $c \leftarrow$ set of struct $\langle docid, weight \rangle$
  for $t \in 1 \ldots q$ do
    $U[t] \leftarrow p_t.max\_score$ // max_score of posting list of term $t$
    $c[t] \leftarrow p_t$.BEGIN() // first posting in posting list of term $t$
  end for
  $\theta \leftarrow -\infty$ // current threshold
  $result \leftarrow \emptyset$ // min priority queue of pairs $\langle docid, weight \rangle$ sorted by weight
  while set of candidates $c$ has valid postings do
    sort candidates so that $c[1].docid \leq c[2].docid \leq \ldots c[q].docid$.
    re-arrange $U$ according to $c$.
    $score \leftarrow 0$
    $pivot \leftarrow 1$
    while $pivot \leq q$ do
      $tmp\_score \leftarrow score + U[pivot]$
      if $tmp\_score > \theta$ then // we found a candidate pivot
        EVALUATE($c, pivot, result, p$)
        break
      end if
      $score \leftarrow tmp\_score$
      $pivot \leftarrow pivot + 1$
    end while
    EVALUATE($c, pivot, result, p$)
  end while
  return $result$
```

3.2.2 Block-Max WAND

BLOCK-MAX-WAND [59] is an improvement of the WAND query processing mechanism. It follows a similar procedure, but instead of using the maximum lists weights, it also cuts the lists into blocks and stores the maximum weight for each block. We describe this procedure in Algorithm 11. The idea of cutting the lists into blocks is to enable skipping of whole blocks whose maximum possible contribution is very low, by comparing its maximum weight with the threshold θ given by the current candidate set.

Algorithm 9: EVALUATE$(c, pivot, result, p)$. Receives the candidate set c, the *pivot* number that corresponds to the position of the pivot posting list in c, a result set containing pairs $\langle docid, weight\rangle$ and the postings lists p.

```
EVALUATE($c, pivot, result, p$)
  if $c[0] = c[pivot]$ then
    $score \leftarrow 0$
    $t \leftarrow 1$
    while $t \leq q \wedge c[t].docid = c[pivot].docid$ do
      $score \leftarrow score + c[t].weight$
      $c[t] \leftarrow p_t$.NEXT() // move pointer of posting $p_t$ to the next $\langle docid, weight\rangle$
      $t \leftarrow t + 1$
    end while
    TRY_REPORT($result, score, c[pivot].docid, \theta$)
  else
    for $t \in 1 \ldots pivot$ do
      $c[t] \leftarrow p_t$.SEEK_DOCUMENT($c[pivot].docid$)
    end for
  end if
```

Algorithm 10: TRY_REPORT$(result, score, docid, \theta)$. Receives a *result* min-priority queue of size k that contains element of pairs $\langle docid, weight\rangle$, the score that we try to add to *result*, the document *docid* and the threshold θ.

```
TRY_REPORT($result, score, docid, \theta$)
  if $score > \theta$ then
    PUSH($result, \langle docid, score\rangle$)
    if $|result| > k$ then
      POP($result$)
      $\theta \leftarrow$ TOP($result$).score
    end if
  end if
```

The basic concept is as follows: Suppose the next document of interest, d, belongs to blocks $b_1, \ldots, b_q$ in the q lists. Compute an upper bound to $score(Q, d)$ (lines 13–27 of Algorithm 11) using the block maxima instead of the weights $w(t, d)$. If this upper bound does not surpass the kth best score known up to now, no document inside the current blocks can make it to the top-k list. So we can safely skip some blocks (lines 29–32). BLOCK-MAX-WAND obtains considerable performance gains over the previous techniques for exact ranked unions [36, 152]. Just as WAND, BLOCK-MAX-WAND can be used for ranked intersections too: once we know a block contain a candidate document d, we check for that document in all posting lists from the query; if the document is contained in all posting lists, we evaluate it and insert it into the top-k results if possible. Recently, Petri et al. [132] presented a complete evaluation of

WAND and BLOCK-MAX-WAND methods, supporting other more complex scoring formulas.

For implementing BLOCK-MAX-WAND, the posting lists p_t requires extra methods and attributes in addition to the ones required by WAND: method SEEK_BLOCK(DOC) returns a pointer to the block whose smallest *docid* is greater or equal to *doc*, the attribute *cur_block.mac_weight* contains the maximum weight of the current block being examined and *cur_block.maxdoc* is an attribute that contains the maximum *docid* that lies in the current block.

3.2.3 Persin et al.'s Algorithm

Persin et al.'s algorithm [131] is one of the most famous TAAT query processing algorithms. The idea is to solve bag-of-word queries without scanning all of the lists. The algorithm requires the lists of each term to be sorted by decreasing weight. While the algorithm is described for the *tf-idf* model, it can be easily adapted to many variants, with lists sorted by impact [12].

The first step of the algorithm creates an accumulator acc_d for each document d in the dataset (in practice, one can dynamically add a new accumulator when a candidate document is found). The second step will store into the corresponding accumulators acc_d the weights of the documents of the shortest among the lists of the query terms, that is, the one with the highest idf_t. The third step processes the rest of the lists in increasing length order, where the weight of each document is accumulated in its corresponding acc_d. In order to avoid processing the complete lists, they enforce a minimum threshold such that if the $w(t,d)$ values fall below it, the list is abandoned. Since the longer lists have a lower idf_t multiplying the term frequencies, it turns out that a lower proportion of the longer lists is traversed. They also apply a stricter threshold that limits the insertion of new documents as candidates. These thresholds provide a time/quality tradeoff.

3.2.4 Other Approaches

Buckley and Lewit [37] introduced one of the earliest query optimization techniques. The idea follows a TAAT strategy, from the least frequent term to the most frequent one. During the query evaluation, it is possible to show that is not necessary to evaluate any more terms, since it is impossible for the document at rank $k+1$ to surpass the score of the document at rank k. This technique has the advantage that it is not required to load from disk to main memory some of the posting lists involved in the query.

Moffat and Zobel [111] proposed another technique that also follows a TAAT traversal, and introduced two heuristics: Quit and Continue. The idea behind the Quit heuristic is to dynamically add accumulators for the documents until the amount of them meets some fixed value. The documents are ranked by the partial scores from the accumulators and are reported to the user. On the other hand, the Continue heuristic follows a similar procedure. It also uses a fixed number of accumulators, however, when the threshold is

Algorithm 11: BLOCK-MAX-WAND(Q, k, p). Receives a query containing q postings lists, the number of documents to be returned k and the posting lists p.

```
BLOCK-MAX-WAND$(Q, k, p)$
  compute candidate set $c$ as described in lines $1-4$ in Algorithm 8.
  compute pivot as described in lines $7-19$ in Algorithm 8.
  $score\_limit \leftarrow 0$
  $\theta \leftarrow -\infty$ // current threshold
  $result \leftarrow \emptyset$ // min priority queue of pairs $\langle docid, weight \rangle$ sorted by weight
  while set of candidates $c$ has valid postings do
    for $t \in 1 \ldots pivot$ do
      $c[t] \leftarrow p_t$.SEEK_BLOCK$(c[pivot].docid)$
      $score\_limit \leftarrow score\_limit + p_t.cur\_block.max\_weight$
      $t \leftarrow t + 1$
    end for
    $t \leftarrow pivot + 1$
    while $t \leq q \wedge c[t].docid = c[pivot].docid$ do
      $score\_limit \leftarrow score\_limit + p_t.cur\_block.max\_weight$
      $t \leftarrow t + 1$
    end while
    $pivot \leftarrow t - 1$
    if $score\_limit > \theta$ then
      $docid = c[pivot].docid$
      for $t \in 1 \ldots pivot$ do
        $c[t] \leftarrow p_t$.SEEK_DOCUMENT$(c[pivot].docid)$
        if $c[t] = c[pivot]$ then
          $score \leftarrow score + c[t].weight$
          $c[t] \leftarrow p_t$.NEXT$()$
        end if
      end for
      TRY_REPORT$(result, score, docid, \theta)$
    else
      $min\_next\_doc \leftarrow 1 + min\{p_t.cur\_block.maxdoc \mid 1 \leq t \leq pivot\}$
      for $t \in 1 \ldots pivot$ do
        $c[t] \leftarrow p_t$.SEEK_DOCUMENT$(min\_next\_doc)$
      end for
    end if
  compute $pivot$ as described in lines $7-19$ in Algorithm 8.
  end while
  return $result$
```

reached it continues the evaluation, but only considering the documents that are already represented by the accumulators.

Strohman and Croft [152] use an impact-sorted index with a specific ranking function proposed by Anh and Moffat [12], yet it can be extended to a ranking function such as BM25. This approach uses a variation of TAAT query processing where it first accesses the higher layers of the lists, named segments, and then moves to the lower layers with smaller impact scores. The algorithm stops when it is guaranteed to have a set containing the top-k results. More recently Lin and Trotman [104] introduced an *anytime ranking index* that is based on impact-sorted organization of the posting lists and SAAT traversal.

See the work of Fontoura et al. [70] for a comprehensive evaluation and comparison of these techniques.

3.3 Dual-Sorted Inverted Lists

Navarro and Puglisi [122] presented a data structure based on wavelet trees (recall Section 2.7) to represent the posting lists of an inverted index. The index is dubbed *Dual-Sorted*, since it is possible to traverse the posting lists as if they were sorted by docid or by weight.

For each term $t \in V$ they sort the posting list of each term in *descending* weight order, and they store the document ids in a list $L_t[1, df_t]$. Let $N = \sum_{t \in V} df_t$, they propose to concatenate all lists L_t into a unique List $L[1, N]$ and store for each term t the starting position, sp_t, of a list L_t within L in an array $ptrs[1, V]$. The sequence L is then represented using a wavelet tree.

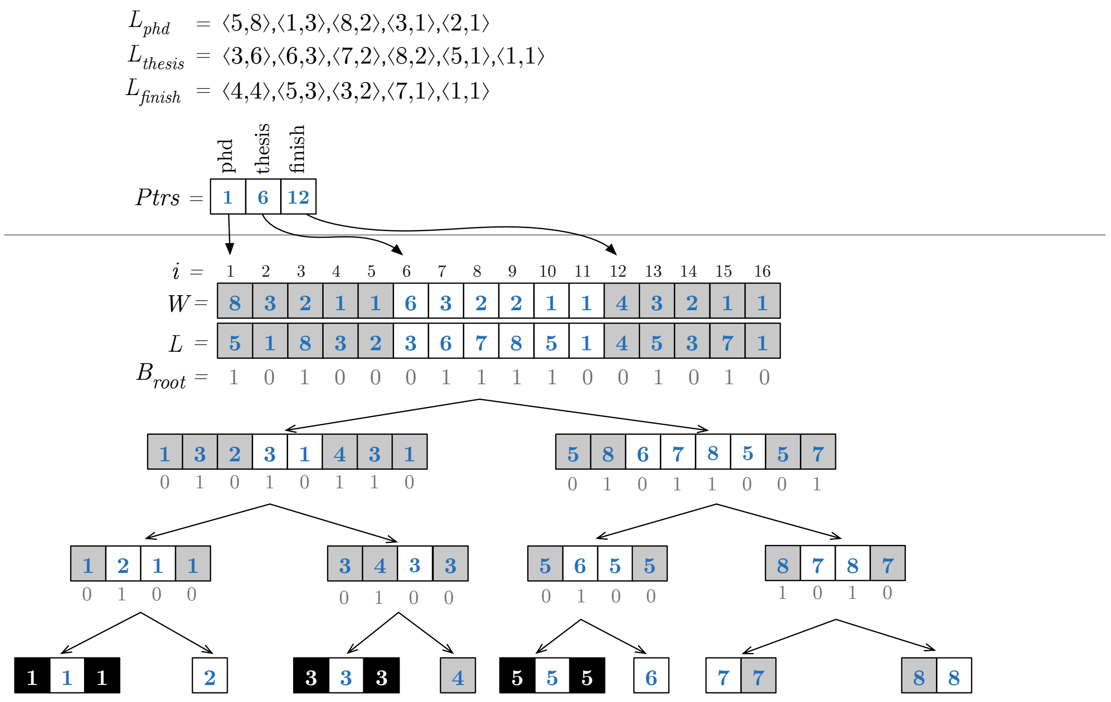
The weights $W[1, N]$ are stored in a differential and run-length compressed form in a separate sequence W'. In their original work, they represent the weights as the frequency $tf_{t,d}$, by marking the different v_t values of each weight list W_t in a bit sequence $T_t[1, m_t]$, where $m_t = max_d tf_{t,d}$, and the points in $L_t[1, df_t]$ where the value $tf_{t,d}$ changes in a bit sequence $R_t[1, df_t]$. They preprocess T_t and R_t to answer RANK and SELECT queries, so that obtaining any value $W_t[i]$ is possible by performing $W_t[i] = \text{SELECT}_1(T_t, v_t - \text{RANK}_1(R_t, i) + 1)$. Navarro and Puglisi represent the $ptrs[1, V]$ array as a bit sequence $VP[1, N]$ also preprocessed for RANK and SELECT operations. This way, the starting point of any list L_t in L can be obtained by performing $\text{SELECT}_1(VP, t)$, and $\text{RANK}_1(VP, i)$ can be used to know to which posting list position i belongs to in L.

The space required to represent $L[1, N]$ using a wavelet tree is $NH_0(L) + o(N \log D)$ bits using RRR (see Section 2.6.1) as the implementation for the bit sequences.

Figure 3.1 shows a simplified version of the data structure representing three posting lists. In this example we show the explicit values of $W[1, N]$.

3.3.1 Basic operations

As previously mentioned, each list $L_t[1, df_t]$ contains the document ids sorted by decreasing weight. We denote $P_t[1, df_t]$ the list sorted by document ids. We will describe in the following sections how to carry out different operations on P_t, using the wavelet tree representation L,



L_{phd} = ⟨5,8⟩,⟨1,3⟩,⟨8,2⟩,⟨3,1⟩,⟨2,1⟩
L_{thesis} = ⟨3,6⟩,⟨6,3⟩,⟨7,2⟩,⟨8,2⟩,⟨5,1⟩,⟨1,1⟩
L_{finish} = ⟨4,4⟩,⟨5,3⟩,⟨3,2⟩,⟨7,1⟩,⟨1,1⟩
phd
thesis
finish
Ptrs = 1 6 12
i = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
W = 8 3 2 1 1 6 3 2 2 1 1 4 3 2 1 1
L = 5 1 8 3 2 3 6 7 8 5 1 4 5 3 7 1
B_{root} = 1 0 1 0 0 0 1 1 1 1 0 0 1 0 1 0

Figure 3.1: Example of Dual-Sorted inverted list for three posting lists. The top part of the figure shows the posting lists values as pairs $\langle docid, weight\rangle$ for three terms. Array $Ptrs$ shows where each posting list is represented at the root of the wavelet tree. Sequence W stores the concatenation of the weights in descending order, and sequence L is the concatenation of the docids. The positions with grey background represent the intersection traversal of the lists L_{phd} and L_{finish}. At the leaves, we show in black background the final intersection.

thus we do not require P_t to be represented separately.

Direct Retrieval

In order to obtain any value $P_t[i]$ (that is, as if the lists were sorted by document id) we can use the RANGE_QUANTILE operation from the wavelet tree (see Section 2.7). The idea is to find the i-th smallest value in $L[sp_t, sp_{t+1}-1]$. The algorithm for a general range $L[l,r]$, is as follows [122]: Starting from the root v of the wavelet tree, we count the numbers of ones in the bit sequence B_v as $n_1 = \text{RANK}_1(B_v, r) - \text{RANK}_1(B_v, l-1)$ and count the number of zeros as $n_0 = (r-l+1)-n_1$. The main idea is to know how many values belong to the smaller half of the document identifiers in $L[l,r]$. With n_0 we know that if $i \le n_0$ there are at least i values that belong to the smaller half of docids, thus we continue recursively on the left child of v setting $[l,r]$ to $[l = \text{RANK}_0(B_v, l-1)+1, r = \text{RANK}_0(B_v, r)]$. In the other case, $i > n_0$, we proceed to the right child of v with the range $[l = \text{RANK}_1(B_v, l-1)+1, r = \text{RANK}_1(B_v, r)]$ and set $i = i - n_0$. The symbol located at the leaf is the answer.

Retrieving a Range

It is also possible to extract a range $P_t[i, i']$, in docid order, in time $O((i'-i+1)(1+\log \frac{D}{i'-i+1}))$. The procedure is similar to the one previously described: we go to the same branch when both i and i' tell us to do it, and we split the interval into two separate ranges when they do not. The worst case is when we arrive at $i' - i + 1$ leaves of the wavelet tree, but part of the work of arriving at these leave is shared.

3.3.2 Complex Operations

Intersection

Given a candidate document d, an important operation for intersecting posting lists is to find the first position j such that $P_t[j] \geq d$. This is commonly solved with a combination of regular samplings and linear, binary or exponential search (recall Section 3.2). In order to solve this operation using the wavelet tree, we start at the root node v, the root bit sequence B_v and the interval $L[l, r]$ that represents the docids of term t, with $l = st_t$ and $r = st_{t+1} - 1$. In the case that the number d belongs to the first half, we descend to the left, otherwise we descend to the right. In both cases we need to update the corresponding ranges l and r as in the procedure to obtain any value $P_t[i]$. If the interval $[l, r]$ becomes empty, this means that there is no value d in the subtree and we return without a value. On the other hand, if we arrive at a leaf with $l = r$, the leaf represents the document that we are searching for and we return this value. If at some point during the recursion, we return no result, we have to look for the first result to the right, searching for the smallest document $d' > d$. If the recursion was to the right child, or the left but there was no 1 marked in $B_v[l, r]$, we know that there is no document d' bigger than d. Otherwise, we enter to the right child looking for the smallest value. From there we follow the leftmost path if there is some 0 in $B_v[l, r]$, otherwise we descend to the right. This way, in at most two root-to-leaf traversals we find the first $d' \geq d$ value in P_t. To obtain j, the position of d' in P_t, we must perform the sum of all n_0 values from all the nodes that were traversed to reach d' where we have gone to the right. This operation can be carried out in $O(\log D)$ time, which is not far from the time required to perform a binary search on P_t. In the case that $P_t[j] = d$, we must look for the first value $P_t[j'] \geq d'$, that can be done in amortized time proportional to $O(\log(d' - d + 1)$.

The wavelet tree is capable of performing the Boolean intersection of multiple lists efficiently by traversing all list at the same time. We refer to this procedure as RANGE_INTERSECTION and is described in Algorithm 12. Given a set of k x-ranges $[x_1^s, x_1^e]$, $[x_2^s, x_2^e]$...$[x_k^s, x_k^e]$, this operation returns the y-coordinates that are common to all the x-ranges. The idea is to start from the k ranges at the root of the wavelet tree, and as before, project those ranges to the leaves. We map the ranges to the left child and right child of the root using RANK$_0$ and RANK$_1$. We continue recursively on the branches where all of the k sub-intervals are not empty. If we are able to reach a leaf, then its corresponding symbol is in the intersection. In the worst-case, this operation takes $O(\sum_{i=1}^{k} x_i^e - x_i^s + 1)$, however there are more detailed time complexity measures for this operation, see for example the work of Gagie et al. [74, Theorem 8] for a more detailed analysis.

Using this operation, this data structure also allows us to carry out another intersection algorithm: Let us say that we want to intersect all values that are shared by P_t and P'_t, we know that the ranges in the wavelet tree for these postings lists are $[st_t, st_{t+1} - 1]$ and $[st_{t'}, st_{t'+1} - 1]$. Now we can apply Algorithm 12. Recall that this algorithm allows us to intersect k ranges, so we can perform the intersection of k posting lists at the same time.

Algorithm 12: Method RANGE_INTERSECT$(v, [x^s_1, \ldots, x^s_k], [x^e_1, \ldots, x^e_k])$. Reports all the symbols that are common in the k x-ranges $[x^s_1, x^e_1], [x^s_2, x^e_2]...[x^s_k, x^e_k]$. We denote x^s_i the starting position of the i-th ($1 \leq i \leq k$) range and x^e_i its ending position.

```
RANGE_INTERSECT($v, [x^s_1, \ldots, x^s_k], [x^e_1, \ldots, x^e_k]$)
  if $x^s_1 > x^e_1 \vee \ldots x^s_k > x^e_k$ then
    return
  if $v$ is a leaf then
    report LABELS ($v$)
    return
  $r_{start\_left}, r_{end\_left}, r_{start\_right}, r_{end\_left} \leftarrow \emptyset$
  for each $x_{start} \in [x^s_1, \ldots, x^s_k], x_{end} \in [x^e_1, \ldots, x^e_k]$ do
      $r_{start\_left} \leftarrow r_{start\_left} \cup \{\text{RANK}_0(B_v, x_{start} - 1) + 1)\}$
      $r_{end\_left} \leftarrow r_{end\_left} \cup \{\text{RANK}_0(B_v, x_{end})\}$
      $r_{start\_right} \leftarrow r_{start\_right} \cup \{x_{start} - x_{start\_left}\}$
      $r_{end\_right} \leftarrow r_{end\_right} \cup \{x_{end} - x_{end\_left}\}$
  end for
  RANGE_INTERSECT($v_l, r_{start\_left}, r_{end\_left}$)
  RANGE_INTERSECT($v_r, r_{start\_right}, r_{end\_right}$)
```

Union

We can use a similar procedure as the one described in Algorithm 12 for the union. Recall from the algorithm that, for reporting a document, all ranges have to reach a leaf. In the case of the Boolean union, we can relax this condition and allow that some ranges could become empty, and we only report the ones that reached the leaves. The cost of this procedure is $O(M(1 + \log \frac{D}{m}))$, where m is the size of the output and M is the sum, over the returned documents, of the number of intervals where they appeared.

Ranked Unions

It is possible to support ranked union operations on this data structure by implementing Persin et al.'s algorithm [131]. We use the operation described to extract a range $L[i, j]$ in order to obtain the whole shortest list, and to extract a prefix of the next lists. The extension of the prefix to be extracted (according to the threshold on $w(t, d)$ given by the algorithm) is computed by an exponential search on W. Note that the range operation obtains the documents of the lists sorted by docid, which makes it convenient to merge into our set of accumulators acc_d if they are also sorted by docid. Note that W stores *tf* values; these are multiplied by idf_t before accumulating them.

Ranked Intersections

Ranked intersections can be handled by first performing the complete Boolean intersection and then proceeding to score documents. First, we find the q intervals $[s_t, e_t]$ of the query words using the pointers from the vocabulary to the inverted lists L_t, and sort them by increasing lengths. Then we use Algorithm 12 for tracking the ranges within the wavelet tree. We track all the q ranges simultaneously, stopping as soon as any of those becomes empty. Every time we reach a leaf, it corresponds to a document d that participates in the intersection. Their term frequencies are available by first obtaining the original position in sequence L. This can be done by performing SELECT operation using the docid as the symbol and the leaf position as the number of occurrences. This way we can compute the document score, by traversing the tree upwards and obtaining the weight using sequence W. As we obtain each document that belongs to the intersection, we retain the k highest scored documents in a min priority queue containing pairs $\langle docid, weight \rangle$ prioritized by weight.

3.4 Compression of Posting Lists

Compression of the postings lists is essential for efficient retrieval, on disk and in main memory [166]. The main idea to achieve compression is to differentially encode the docids, whereas the weights are harder to compress. A list of docids $\langle d_1, d_2, d_3, \ldots d_n \rangle$ is represented as a sequence of d-gaps $\langle d_1, d_2 - d_1, d_3 - d_2, \ldots, d_n - d_{n-1} \rangle$, and variable-length encoding is used to encode the differences (see Section 2.5) Extracting a single list or merging lists is done optimally by traversing lists from the beginning, but query processing schemes can be done much faster if random access to the sequences is possible. This can be obtained by cutting the lists into *blocks* that are differentially encoded, while storing in a separate sequence the absolute values of the block headers and pointers to the encoded blocks [51, 145].The result is a two-level structure: the first contains the sampled values and the second stores the encoded sequence itself. For example, Culpepper and Moffat [51] extract a sample every $p' = p \log n$ values from the compressed list, where p is a space/time tradeoff parameter. Direct access requires a binary search on the samples list plus the decompression of a within-samples block. Sanders and Transier [145], instead, sample regularly the domain values: all the values differing only in their $p = \log(B/n)$ lowest bits (for a parameter B, typically 8), form a block. The advantages are that binary searches on the top structure, and storing absolute values in it, are not necessary. A disadvantage is that the blocks are of varying length and more values might have to be scanned on average for a given number of samples.

Since bit-aligned codes can be inefficient at decoding as they require several bitwise operations, byte-aligned [147] or word-aligned codes [168, 11] are preferred when the speed is the main concern. Examples of this family of techniques are *Variable Byte* (VByte), Simple9 and

Simple16 (recall Section 2.5). Another approach is to encode blocks of integers at the same time, this way trying to improve both compression and decoding speed. The most popular block-based encoding mechanism is known as PForDelta. Any other integer compression mechanism can be employed to compress the posting lists, providing different space/time tradeoffs. See the work of Trotman [157] for a comprehensive study and comparison of these techniques.

Vigna [161] explored an *Elias-Fano* representation of monotone sequences to represent the docids of the posting lists. Different from the traditional approach, it is not necessary to represent the list as a sequence of d-gaps. Vigna showed how to perform random access directly over the compressed sequence and implement efficient operators that are useful for skipping when performing intersections. The usage of this compression scheme showed that it is possible to achieve competitive space and time when compared to state-of-the-art methods such as γ-δ-Rice and even PForDelta. Ottaviano and Venturini [127] extended this idea by performing an $(1 + \epsilon)$-optimal partitioning of the list into chunks, for any $\epsilon > 0$, and then encoding both the chunks and their endpoints with Elias-Fano. Their results show that the partitioned approach offers significantly better compression and similar performance in terms of query time efficiency.

When the lists are sorted by decreasing weight (for approximate ranked unions), the differential compression of docids is not possible, in principle. Instead, term weights can be stored differentially. When storing *tf* values, one can take advantage of the fact that long runs of equal *tf* values (typically low ones) are frequent, and thus not only perform run-length encoding of them, but also sort the corresponding docids increasingly, so as to encode them differentially [16, 171].

Chapter 4

Inverted Treaps

In this chapter we introduce a new compressed representation for the lists of the inverted index, which performs ranked intersections and (exact) ranked unions directly. Our representation is based on the *treap* data structure (See Section 2.13) , a binary tree that simultaneously represents a left-to-right and a top-to-bottom ordering. We use the left-to-right ordering for document identifiers (which supports fast Boolean operations) and the top-to-bottom ordering for term weights (which supports the thresholding of results simultaneously with the intersection process). Using this data structure, we can obtain the top-k results for a ranked intersection/union without having to produce the full Boolean result first.

We explore different alternatives to engineer the new list representation using state-of-the-art compact data structures to represent the treap topology. The classical differential representation of docids becomes less efficient in the treap, but in exchange the treap representation allows us to differentially encode *both* docids and weights, which compensates for loss. We also present novel algorithms for processing the queries on the treap structure, and compare their performance against well-known approaches such as WAND [36], Block-Max [59], and Dual-Sorted [99]. Our experiments under the classical tf-idf scoring scheme show that the space usage of our treap-based inverted index representation is competitive with the state-of-the-art compressed representations: our faster variant is around 25% larger than WAND, as large as Block-Max, and 15% smaller than Dual-Sorted. In terms of time, the fastest inverted treap variant outperforms previous techniques in many cases: on ranked one-word queries it is 20 times faster than Dual-Sorted and 25–200 times faster than Block-Max; WAND is even slower. On ranked unions it is from 10 times (for $k = 10$) to 3 times (for $k = 1000$) faster than Block-Max, and similarly from 45–50 times to 6–7 times faster than WAND and Dual-Sorted. It is always the fastest index for one-word and two-word queries, which are the most popular. On ranked intersections, our fastest treap alternative is twice as fast as Block-Max for $k = 10$, converging to similar times for $k = 1000$. In the same range of k, it goes from twice as fast to 40% faster than WAND, and from 80% faster to 10% slower than Dual-Sorted. Our inverted treap is always the fastest index up to $k = 100$. We also experimented under a quantized BM25 scoring scheme, where the fastest inverted treap uses 35%–40% more space than Block-Max or Dual-Sorted. It is still slightly better than all the alternatives on ranked unions. For ranked intersections, it is from twice as fast (for $k = 10$) to 15% slower (for $k = 1000$) on ranked intersections, still being the fastest for $k = 100$. On one-word queries, the inverted treap is 20 times faster than Dual-Sorted and 40–300 times faster than Block-Max.

Those ranges of k values make this result very relevant both for a first stage retrieving a few hundreds or thousands of documents, or for directly conveying a handful of final results to the user. We also show how to support incremental updates on the treap, making this representation a useful alternative for scenarios where new documents must be available immediately. The overhead of allowing for incremental updates compared to our static alternatives, in terms of space, construction and query times, ranges from 50% to 100%.

The first version of this work appeared in the proceedings of the 36th ACM Special Interest Group in Information Retrieval (SIGIR) Conference held in Dublin, Ireland on August 2013. An extended version has been submitted for revision to the ACM Transactions on Information Systems in March 2016. This work has been developed in collaboration with Charles L.A Clarke and Alejandro López-Ortíz from the University of Waterloo, Canada.

4.1 Inverted Index Representation

We consider the posting list of each term as a sequence sorted by docids (which act as keys), each with its own term frequency (which act as priorities). Term impacts, or any other term weights, may also be used as priorities. We then use a treap to represent this sequence. Therefore the treap will be binary searchable by docid, whereas it will satisfy a heap ordering on the frequencies. This means, in particular, that if a given treap node has a frequency below a desired threshold, then all the docids below it in the treap can be discarded as well.

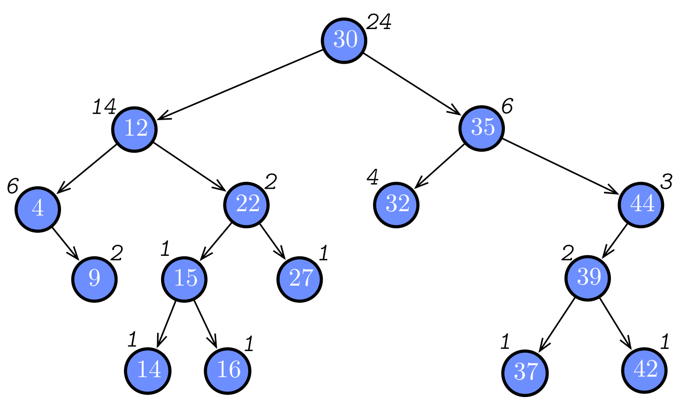
Figure 4.1 illustrates a treap representation of a posting list. This treap will be used as a running example. Ignore for now the differential arrays on the bottom.

4.1.1 Construction

A treap on a list $\langle(d_1, w_1), \ldots, (d_n, w_n)\rangle$ of documents and weights can be built in $O(n)$ time in a left-to-right traversal [27, 22, 69]. Initially, the treap is just a root node (d_1, w_1). Now, assume we have processed $(d_1, w_1), \ldots, (d_{i-1}, w_{i-1})$ and the rightmost path of the treap, root to leaf, is $v'_1, \ldots, v'_\ell$, each v'_j representing the posting (d'_j, w'_j). Then we traverse the path from v'_ℓ to v'_1, until finding the first node v'_j with $w'_j \geq w_i$ (assume the treap is the right child of a fake root with weight $w'_0 = +\infty$, to avoid special cases). Then, (d_i, w_i) is set as the right child of v'_j and the former right child of v'_j becomes the left child of (d_i, w_i), which becomes the lowest node in the rightmost path.

Since for every step in this traversal the rightmost path decreases in length, and it cannot increase in length by more than 1 per posting, the total number of steps is $O(n)$. Under reasonable assumptions (i.e., the weight and the docid are statistically independent and the input list is already sorted by *docid*) the height of a treap is $O(\log n)$ [108], and so is the length of its rightmost path. Therefore, the maximum time per insertion is $O(\log n)$ expected

docids =	4	9	12	14	15	16	22	27	30	32	35	37	39	42	44
freqs =	6	2	14	1	1	1	2	1	24	4	6	1	2	1	3



diff *docids* =	8	5	18	1	7	1	10	7	30	3	5	2	5	3	9
diff *freqs* =	8	4	10	0	1	0	12	1	24	2	18	1	1	1	3

Figure 4.1: An example posting list with docids and frequencies and the corresponding treap representation in our scheme. Note that docids (inside the nodes) are sorted inorder and frequencies (outside the nodes) are sorted top to bottom. The differentially encoded docids and frequencies are shown below the tree.

(but $O(1)$ worst-case when amortized over all the insertions on the treap).

4.1.2 Compact Treap Representation

To represent this treap compactly we must encode the tree topology, the docids, and the term frequencies. We discuss only the docids and frequencies in this subsection. Our plan is not to access the posting lists in sequential form as in classical schemes, thus a differential encoding for each docid with respect to the previous one is not directly applicable. Instead, we make use of the invariants of the treap data structure.

Let $id(v)$ be the docid of a treap node v, and $f(v)$ its frequency. We represent $id(v)$ and $f(v)$ for the root in plain form, and then represent those of its left and right children recursively. For each node v that is the left child of its parent u, we represent $id(u) - id(v)$ instead of $id(v)$. If, on the other hand, v is the right child of its parent u, we represent $id(v) - id(u)$ [47]. In both cases, we represent $f(u) - f(v)$ instead of $f(v)$. Those numbers get smaller as we move downwards in the treap.

The sequence of differentially encoded $id(v)$ and $f(v)$ values is represented according to an inorder traversal of the treap, as show on the bottom of Figure 4.1. As we move down the treap, we can easily maintain the correct $id(v)$ and $f(v)$ values for any node arrived at, and use it to compute the values of the children as we descend.

To do this we need to randomly access a differential value in the sequence, given a node. We store those values in an array indexed by node inorders and use the Direct Addressable Codes (DACs) described in Section 2.8 to directly access the values while taking advantage of their small size.

4.1.3 Representing the Treap Topology

We have shown that if we use a treap to represent posting lists we can differentially encode both docids and frequencies, however, we still need to represent the topology. A pointer-based representation of a treap topology of n nodes requires $O(n \log n)$ bits, which is impractical for large-scale data. Still, space is not the only concern: we need a compact topology representation that supports fast navigation in order to implement the complex algorithms deriving from ranked intersections and unions. In this subsection we introduce three representations designed to be space-efficient and to provide fast navigation over the topology.

Compact Treap using BP

This representation uses the balanced parentheses (BP) described in Section 2.10. However, this representation is designed for general ordinal trees, not for binary trees. For example, if a node has only one child, general trees cannot distinguish between it being the "left" or the "right" child.

A well-known isomorphism [114] allows us represent a binary tree of n nodes using a general tree of $n + 1$ nodes: First, a fake root node v_{root} for the general tree is created. The children of v_{root} are the nodes in the rightmost path of the treap, from the root to the leaf. Then each of those nodes is converted recursively. The general tree is then represented as a BP sequence $\mathcal{S}_{\text{BP}}[1, 2n + 2]$. With this transformation, the original treap root is the first child of v_{root}. The left child of a treap node v is its first child in the general tree and the right child of v is its next sibling in the general tree. Moreover, the inorder in the original treap, which we use to access docids and frequencies, corresponds to the preorder in the general tree, which is easy to compute with parentheses. Figure 4.2 shows the transformed treap for our running example.

Each treap node i is identified with its corresponding opening parenthesis. Thus the root of the treap is node 2. The inorder of a node i is simply $\text{RANK}_1(\mathcal{S}_{\text{BP}}, i) - 1$, the number of opening parentheses up to i excluding the fake root. The left child of i in the treap, or its first child in the general tree, is simply $\text{FIRST\_CHILD}(i) = i + 1$. If, however, $\mathcal{S}_{\text{BP}}[i + 1] = 0$, this means that i has no first child. For the right child of node i in the treap, or its next sibling in the general tree, we use $\text{NEXT\_SIBLING}(i) = \text{CLOSE}(i) + 1$, where CLOSE returns the closing parenthesis that matches a given opening parenthesis. If $\mathcal{S}_{\text{BP}}$ has a 0 in the resulting position, this means that i has no right child.

Therefore, in addition to RANK, we need to implement operation CLOSE in constant time. This is achieved with a few additional data structures of size $o(n)$, as described in Section 2.10.

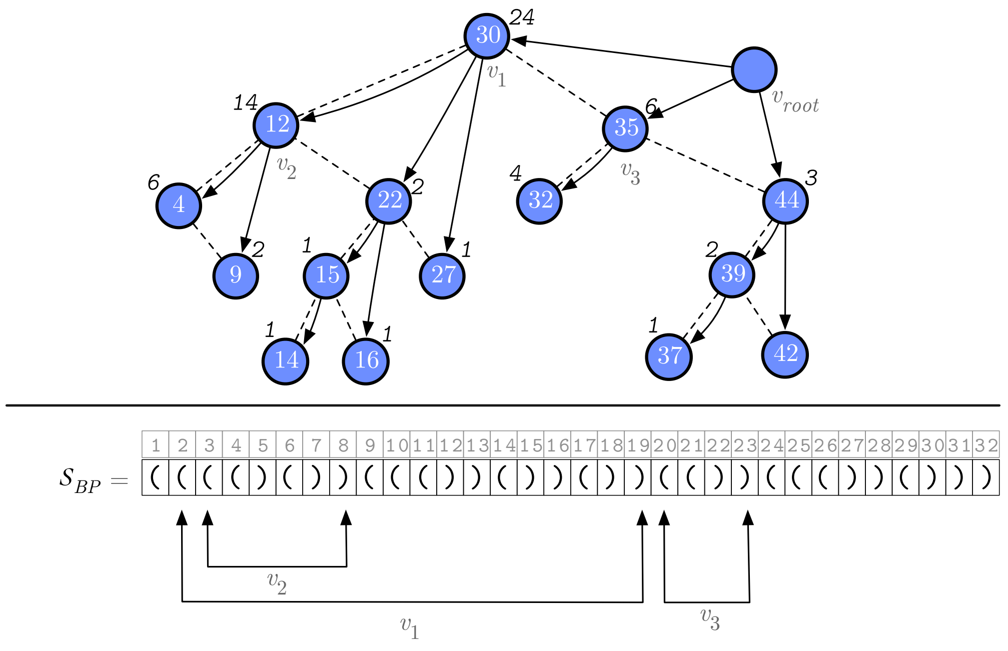


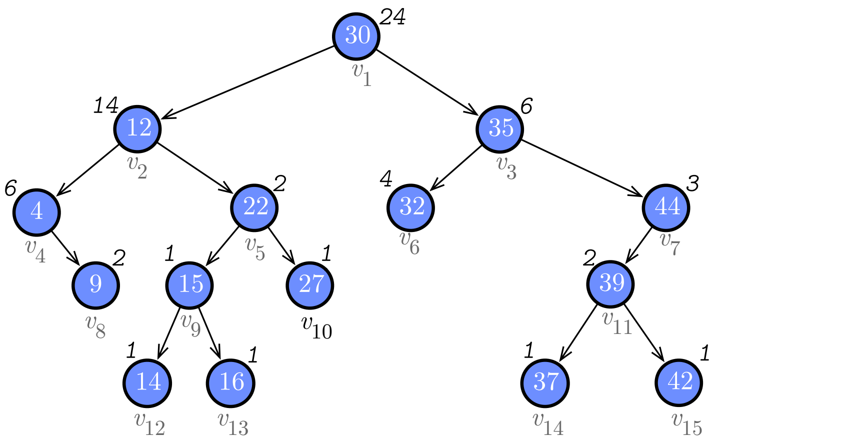
Figure 4.2: The original binary tree edges (dashed) are replaced by a general tree, whose topology is represented with parentheses. The opening and closing parentheses of nodes v_1, v_2 and v_3 in $\mathcal{S}_{\text{BP}}$ are shown on the bottom part.

As an example, we demonstrate this procedure using the treap from Figure 4.2. We see that v_1 (the treap's original root) starts at position $i = 2$ in $\mathcal{S}_{\text{BP}}$. If we want to retrieve the index of its left child, which is node v_2, we perform FIRST_CHILD(2). The procedure checks that $S_{\text{BP}}[3]$ is an opening parenthesis, so we can return 3 as the answer for the starting position of the left node of v_1. On the other hand, if want to obtain the position where the right child of v_1 begins, which is v_3, we perform $\text{CLOSE}(2) + 1$ and obtain position 20, which contains a 1 and thus corresponds to the starting position of v_3.

Compact treap using LOUDS

The LOUDS representation (recall Section 2.10) can be adapted to support binary trees efficiently: for every node in a level-order traversal we append two bits to the bit sequence, setting the first bit to '1' iff the node contains a left child and setting the second to '1' iff the node contains a right child. For example, a leaf node will be represented as 00, while a binary tree node containing both children is represented as 11. The concatenation of these bits builds a bit sequence $\mathcal{S}_{\text{LOUDS}}[1, 2n]$. Since every node adds two bits to this sequence, we can use the levelwise order i of a node as its identifier, knowing that its two bits are at $\mathcal{S}_{\text{LOUDS}}[2i-1, 2i]$.

The LOUDS representation is simpler than BP, as it can be navigated downwards using only RANK operations. In addition, it does not require the tree isomorphism. Given the sequence $\mathcal{S}_{\text{LOUDS}}$ representing the treap topology as a binary tree and a node i, we navigate



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
LOUDS Topology ($\mathcal{S}_{LOUDS}$) =	1	1	1	1	1	1	0	1	1	1	0	0	1	0	0	0	1	1	0	0	1	1	0	0	0	0	0	0	0	0
Node =	v_1		v_2		v_3		v_4		v_5		v_6		v_7		v_8		v_9		v_{10}		v_{11}		v_{12}		v_{13}		v_{14}		v_{15}	

Figure 4.3: The LOUDS representation of an example treap. $\mathcal{S}_{\text{LOUDS}}$ denotes the bit sequence that describes the topology.

the tree as follows: if i has a left child (i.e., if $\mathcal{S}_{\text{LOUDS}}[2i-1] = 1$) then the child is the node (with levelwise order) $\text{RANK}_1(\mathcal{S}_{\text{LOUDS}}, 2i-1)+1$. Analogously, the right child exists if $\mathcal{S}_{\text{LOUDS}}[2i] = 1$, and it is $\text{RANK}_1(\mathcal{S}_{\text{LOUDS}}, 2i)+1$.

Figure 4.3 shows an example of a binary tree using a LOUDS representation. We demonstrate how to navigate the tree with an example: say that we are at node v_4 (meaning its LOUDS identifier is 4), which is represented by the bits located at positions 7 and 8. We know that v_4 has no left child because the first bit is 0, but it has a right child since the second bit is a 1. The right child of v_4 is the node with identifier (or levelwise order) $\text{RANK}_1(\mathcal{S}_{\text{LOUDS}}, 2\cdot 4)+1 = 8$, which we draw as v_8. The two bits of this node are at positions 15 and 16. Since both bits are set to 0, this node is a leaf.

Compared to the BP representation, the LOUDS-based solution requires less space in practice (2.10 bits per node instead of 2.37) and simpler operations. On the other hand, the BP representation has more locality of reference when traversing subtrees.

For this representation, we store the sequences of differentially encoded docids and frequencies (sequences "diff *docids*" and "diff *freqs*" of Figure 4.1) following the level-order traversal of the binary tree. This ordering is shown as "Node" at the bottom part of Figure 4.3.

Compact treap using Heaps

Even if the topology representations using LOUDS or BP support constant-time tree navigation, in practice they are 10 to 100 times slower than a direct access to a memory address (i.e., accessing any position in an array), as shown in the preliminary version of this work

[101]. In order to avoid these costly operations, we designed a new compact binary tree representation that is inspired by binary heaps. The main idea is to take advantage of the fact that a *complete* binary tree of n nodes does not require any extra information to represent its topology, since the values can be represented using just an array of size n. In order to navigate the tree, we can use traditional binary heap operations: the left child of node i is located at position $2i$ and the right child at position $2i+1$. However, a treap posting list representation will rarely be a complete binary tree. Therefore, we take the maximal top part of the tree that is complete and represent it as a heap. We then recursively represent the subtrees that sprout from the bottom of the complete part. The motivation is to avoid the use of RANK or more complex operations every time we need to navigate down the tree.

We start at the root of the treap and traverse it in levelwise order, looking for the first node that does not contain a left or a right child. Say that this happens at level ℓ of the tree, so we know that all the nodes up to level ℓ have both left and right children. In other words, the subtree formed by all the nodes starting from the root up to level ℓ forms a complete tree $\mathcal{T}_1$ that contains $2^\ell - 1$ nodes. Figure 4.4 shows an example, where the first node that does not have a left or right child (v_4) is located at level $\ell = 3$, therefore $|\mathcal{T}_1| = 2^3 - 1 = 7$. We then append the differential values of the nodes (docid and frequency) of the complete subtree to the sequences "diff *docids*" and "diff *freqs*" (see Figure 4.1), in levelwise order.

For each of the $2^{\ell-1}$ leaves of $\mathcal{T}_1$ we write in a bit sequence $\mathcal{S}_{\text{HEAP}}$ two bits, indicating if the corresponding leaf contains a left or a right child, similarly to LOUDS.

We continue this procedure by considering each node located at level $\ell+1$ as a new root, for further trees $\mathcal{T}_j$. The trees yet to process are appended to a queue, so the trees $\mathcal{T}_j$ are also deployed levelwise. Note that any of these new roots (including the original root) could lack a left or a right node, in which case we will have a complete subtree of only one node.

We also need a sequence P where $P[j]$ is the starting point of $\mathcal{T}_j$ in the sequences of docids and frequencies. This also serves to compute $|\mathcal{T}_j| = P[j+1] - P[j]$. Since there may be a considerable number of small complete trees, P may require up to $n \log n$ bits. To reduce its size, and considering that $P[j+1] - P[j]$ is of the form $2^\ell - 1$, we store another array instead, P', that stores the number of levels in the corresponding complete binary tree, $P'[0] = 0$ and $P'[j+1] = \ell - 1 = \log_2(|\mathcal{T}_j| + 1) - 1$. This reduces the space to at most $n \log \log n$ bits, and the starting position of the sequences of $\mathcal{T}_j$ can be obtained as $P[j] = \sum_{i=0}^{j-1}(2^{P'[i]+1} - 1)$. To compute this sum faster, we divide P' into blocks of fixed size b and store in a separate sequence the sums up to the beginning of each block. This way, we limit to b the number of elements that are summed up. A similar trick, computing $-1 + \sum_{i=0}^{j-1} 2^{P'[i]+1} = P[j] + j - 1$, gives the starting position of $\mathcal{T}_j$ in the bitvector $\mathcal{S}_{\text{HEAP}}$.

Figure 4.4 shows our representation. The grey triangles represent the complete trees $\mathcal{T}_1$ to $\mathcal{T}_5$. On the bottom of the figure we show the extra structures discussed.

In order to navigate the tree we proceed as follows: we represent a node v as a pair $\langle j, pos \rangle$, so that v is the node at levelwise-order position *pos* inside the subtree $\mathcal{T}_j$. To move to the left child, we just set $pos' = 2 \cdot pos$, and to move to the right child we set $pos' = 2 \cdot pos + 1$. If $pos' \leq |\mathcal{T}_j|$ we are within the same complete subtree $\mathcal{T}_j$, so we are done. On the other hand, if $pos' > |\mathcal{T}_j|$, we know two things: first, node v is a leaf within its complete subtree $\mathcal{T}_j$, and second, we need to move to another complete subtree. Before moving to another subtree we first need to check if the leaf node has the desired (left or right) child. Thus we map the position of the leaf within its subtree, *pos*, to the sequence $\mathcal{S}_{\text{HEAP}}$. This can be done with $pos\_map = P[j] + j - 1 + 2 \cdot (pos - 1 - \lfloor |\mathcal{T}_j|/2 \rfloor) = P[j] + j - 2 + 2 \cdot pos - |\mathcal{T}_j|$, adding 1 if we descend to the right child. Now, we check in $\mathcal{S}_{\text{HEAP}}[pos\_map]$ if the corresponding bit is set. In the case the leaf node in the subtree $\mathcal{T}_j$ has the desired left or right child, we calculate the new node subtree index with $j' = \text{RANK}_1(\mathcal{S}_{\text{HEAP}}, pos\_norm) + 1$, and set $pos' = 1$.

We demonstrate this process with an example based on Figure 4.4: Let us begin at node v_7, which is represented by the pair $\langle 1, 7 \rangle$ and let us say that we want to move to the left. We set $pos' = 2 \cdot pos = 14$. Since $|\mathcal{T}_1| = 7 < 14$, we realize that we are located at a leaf node. Thus we map *pos* to the sequence $\mathcal{S}_{\text{HEAP}}$ with $pos\_map = 1 + 1 - 2 + 2 \cdot 7 - 7 = 7$. Note that the 7-th bit in $\mathcal{S}_{\text{HEAP}}$ tells if v_7 has a left child or not. Since $\mathcal{S}_{\text{HEAP}}[7] = 1$ we proceed to figure out which tree we must go to. This is computed with $\text{RANK}_1(\mathcal{S}_{\text{HEAP}}, 7) + 1 = 5$. Our new node is then represented as the pair $\langle 5, 1 \rangle$.

For this representation, we maintain the sequences of docids and frequencies following the level-order traversal of the nodes within each complete subtree. This traversal is denoted "Node" in the bottom part of Figure 4.4).

The idea of separating the treap into complete trees is inspired by the level-compressed tries of [8]. Under reasonable models for tries they show that the expected number of complete subtrees traversed in a root-to-leaf traversal is $O(\log \log n)$ and even $O(\log^* n)$. While we are not aware of an analogous result for random binary trees, it is reasonable to expect that similar results hold. Note that this is the number of RANK operations needed in a traversal, instead of the $O(\log n)$ that we can expect using BP or LOUDS.

4.1.4 Practical Improvements

The scheme detailed above would not be so successful without three important improvements. First, because many posting lists are very short, it turns out to be more efficient to store two single DAC sequences, with all the differential docids and all the differential frequencies for all the lists together, even if using individual DACs would have allowed us to optimize their space for each sequence separately. The overhead of storing the chunk lengths and other administrative data outweighs the benefits for short sequences.

A second improvement is to break ties in frequencies so as to make the treap as balanced as possible, by choosing the root as the maximum that is closest to the center of each interval (in every subtree). This improves the binary searches for docids and the tree traversal for the HEAP representation. While it is still possible to build the treap in linear time with this restriction, a simple brute-force approach to find the centered maximum performs better in most practical cases.

The third, and more important, improvement is to omit from the treap representation all the elements of the lists where the frequency is below some threshold f_0. According to Zipf's law [169, 50, 40, 17], a large number of elements will have low frequencies, and thus using a

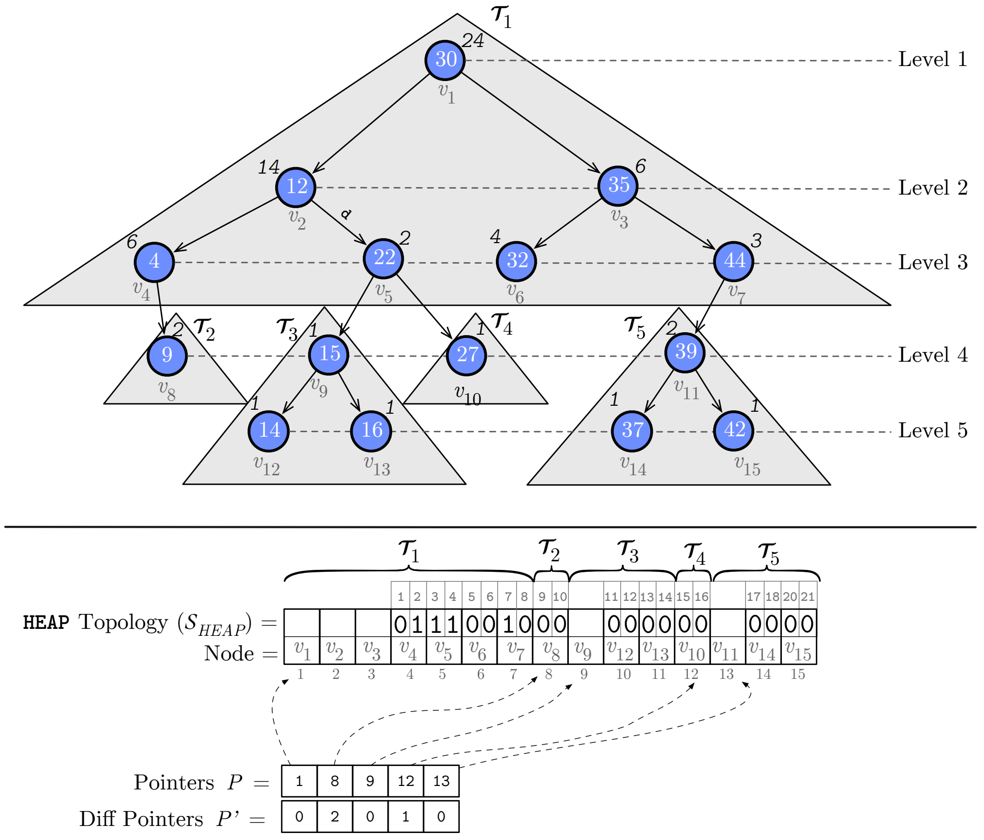


Figure 4.4: An example HEAP treap topology representation. At the top, we draw each complete tree with a grey background. The levels of the tree are shown on the right. At the bottom, we show the resulting $\mathcal{S}_{\text{HEAP}}$ sequence (the holes are not represented) and mark the area of each complete treap $\mathcal{T}_j$. The starting positions of these areas correspond to the sequences of docids and frequencies, as they consider the holes and thus account for the internal nodes as well (see array Node). Those starting positions are written below, in array P. Note that the corresponding starting position in $\mathcal{S}_{\text{HEAP}}$ is simply $P[j] + j - 1$. Instead of P, we store the logarithms of the sizes, in P'.

separate posting list for each frequency below f_0 will save us from storing those frequencies wherever those elements would have appeared in the treap. Further, the docids of each list can be differentially encoded in classical sequential form, which is more efficient than in treap order.

It turns out that many terms do not have to store a treap at all, as they never occur more than f_0 times in any document. We represent the gap-encoded lists using PforDelta and take an absolute sample every 128 values (which form a block). Samples are stored separately and explicitly in an array, with pointers to the block [51]. Searches in these lists will ask for consecutively larger values, so we remember the last element found and exponentially search for the next query starting from there. Figure 4.5 illustrates the separation of low-frequency elements from our example treap.

A neat feature of these lists is that often we will not need to access them at all during

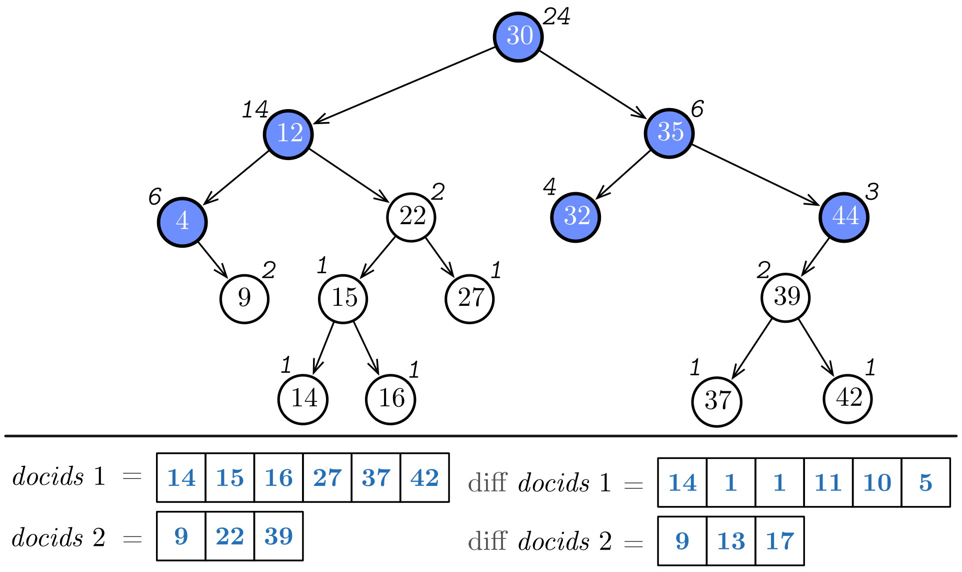


Figure 4.5: Separating frequencies below $f_0 = 2$ in our example treap. The nodes that are removed from the treap are on white background. For the documents with frequencies 1 and 2, we show the absolute docids on the left and their differential version on the right.

queries, since ranked queries aim at the highest frequencies.

4.2 Query Processing

In this section we describe the procedure to perform efficient top-k query processing using the inverted treaps.

4.2.1 General Procedure

Let Q be a query composed of q terms $t \in Q$. To obtain the top-k documents from the intersection or union of q posting lists we proceed in DAAT fashion: We traverse the q posting lists in synchronization, identifying the documents that appear in all or some of them, and accumulating their weights $w(t,d)$ into a final $score(Q,d) = \sum_t w(t,d) = \sum_t tf_{t,d} \cdot idf_t$. Those documents are inserted in a min-priority queue limited to k elements, where the priority is the score. Each time we insert a new element and the queue size reaches $k+1$, we remove the minimum. At the end of the process, the priority queue contains the top-k results. Furthermore, at any stage of the process, if the queue has reached size k, then its minimum score L is a lower bound to the scores we are interested in for the rest of the documents.

4.2.2 Intersections

Let d be the smallest docid not yet considered (initially $d = 1$). Every treap t involved in the query Q maintains a stack of nodes (initially holding just a sentinel value element u_t with $id(u_t) = +\infty$ and $f(u_t) = +\infty$), and a cursor v_t (initially the treap root). The stack will contain the nodes in the path from the root to v_t where we descend by the left child. We will always call u_t the top of the stack, thus u_t is an ancestor of v_t and it holds $id(u_t) > id(v_t)$.

We advance in all the treaps simultaneously towards a node v with docid $id(v) = d$, while skipping nodes using the current lower bound L. In all the treaps t we maintain the invariant that, if v is in the treap, it must appear in the subtree rooted at v_t. In particular, this implies $d < id(u_t)$.

Because of the decreasing frequency property of treaps, if d is in a node v within the subtree rooted at v_t, then $f(v) \leq f(v_t)$. Therefore, we can compute an *upper bound* U to the score of document d by using values $f(v_t)$ instead of $f(v)$, for example $U = \sum_{t \in Q} f(v_t) \cdot idf_t$ for a tf-idf scoring[1]. If this upper bound is $U \leq L$, then there is a valid top-k answer where d does not participate, so we can discard d. Further, no node that is below all the current v_t nodes can qualify. Therefore, we can safely compute a new target $d \leftarrow \min_t(id(u_t))$. Each time the value of d changes (it always increases), we must update the stack of all the treaps t to restore the invariants: While $id(u_t) \leq d$, we assign $v_t \leftarrow u_t$ and remove u_t from the stack. We then resume the global intersection process with this new target d. The upper bound U is recomputed incrementally each time any v_t value changes (U may increase or decrease).

When $U > L$, it is still feasible to find d with sufficiently high score. In this case we have to advance towards the node containing d in some treap. We obtained the best results by choosing the treap t of the shortest list. We must choose a treap where we have not yet reached d; if we have reached d in all the treaps then we can output d as an element of the intersection, with a known score (the current U value is the actual score of d), insert it in the priority queue of top-k results as explained (which may increase the lower bound L), and resume the global intersection process with $d \leftarrow d + 1$ (we must update stacks, as d has changed).

In order to move towards $d \neq id(v_t)$ in a treap t, we proceed as follows. If $d < id(v_t)$, we move to the left child of v_t, l_t, push v_t in the stack, and make $v_t \leftarrow l_t$. Instead, if $d > id(v_t)$, we move to the right child of v_t, r_t, and make $v_t \leftarrow r_t$. We then recompute U with the new v_t value.

If we have to move to the left and there is no left child of v_t, then d does not belong to the intersection. We stay at node v_t and redefine a new target $d \leftarrow id(v_t)$. If we have to move to the right and there is no right child of v_t, then again d is not in the intersection. We make $v_t \leftarrow u_t$, remove u_t from the stack, and redefine $d \leftarrow id(u_t)$. In both cases we adjust the stacks of the other treaps to the new value of d, as before, and resume the intersection process.

[1]Replacing $f(v)$ by $f(v_t)$ will yield an upper bound whenever the scoring function is monotonic with the frequencies. This is a reasonable assumption and holds for most weighting formulas, including tf-idf and BM25.

Algorithm 13 gives pseudocode for the intersection. The auxiliary methods REPORT, CHANGEV and CHANGED are shown in Algorithms 14,15 and 16 respectively.

Algorithm 13: Method TREAP_INTERSECT(Q, k). Performs a ranked intersection from query terms Q and returns the top-k documents and scores.

```
   TREAP_INTERSECT$(Q, k)$
01   $results \leftarrow \emptyset$ // priority queue of pairs $\langle key, priority \rangle$
02   for $t \in Q$ do
03     $stack_t \leftarrow \langle \perp \rangle$ // stack of treap $t$, $id(\perp) = f(\perp) = +\infty$
04     $v_t \leftarrow$ root of treap $t$
05   end for
06   $d \leftarrow 1, L \leftarrow -\infty$
07   while $d < +\infty$ do
08     while $U \leq L$ do
09       CHANGED$(\min_{t \in Q} id($TOP$(stack_t)), stack, U, d)$
10     end while
11     if $\forall t \in Q,\ d = id(v_t)$
12       REPORT$(d, U, k, results)$
13       CHANGED$(d + 1, stack, U, d)$
14     else
15       $t \leftarrow$ treap of shortest list such that $d \neq id(v_t)$
16       if $d < id(v_t)$ then
17         $l_t \leftarrow$ left child of $v_t$
18         if $l_t$ is not null then
19           PUSH$(stack_t, v_t)$, CHANGEV$(t, l_t, U)$
20         else
21           CHANGED$(id(v_t), stack, U, d)$
22         end if
23       else
24         $r_t \leftarrow$ right child of $v_t$
25         if $r_t$ is not null
26           CHANGEV$(t, r_t, U)$
27         else
28           CHANGEV$(t,$POP$(stack_t), U)$
29           CHANGED$(id(v_t), stack, U, d)$
30         end if
31       end if
32     end if
33   end while
34   return $results$
```

Algorithm 14: Method REPORT$(d, s, k, results)$. Maintains the top-k candidates in the min-priority queue $results$, d is the document to be inserted, s is the corresponding score, k is the top-k variable, and $results$ is the min priority queue.

```
REPORT$(d, s, k, results)$
  $results \leftarrow results \cup (d, s)$
  if $|results| > k$ do
    remove minimum from $results$, $L \leftarrow$ minimum priority in $results$
  end if
```

Algorithm 15: Method CHANGEV(t, v, U). Changes the contribution in U according to the current node in v in t.

```
CHANGEV$(t, v)$
  remove contribution of $f(v_t)$ from $U$, e.g. $U - f(v_t) \cdot idf_t$
  $v_t \leftarrow v$
  add contribution of $f(v_t)$ to $U$, e.g. $U + f(v_t) \cdot idf_t$
```

Algorithm 16: Method CHANGED$(newd, stack, U)$. Changes the current document candidate to be evaluated.

```
CHANGED$(newd, stack, d)$
  $d \leftarrow newd$
  for $t \in Q$ do
    $v \leftarrow v_t$
    while $d \geq id($TOP$(stack_t))$ do
      $v \leftarrow$TOP$(stack_t)$
      POP$(stack_t)$
    end while
    CHANGEV$(t, v, U)$
  end for
```

Handling low-frequency lists

We have not yet considered the lists of documents with frequencies up to f_0, which are stored separately, one per frequency, outside the treap. While a general solution is feasible (but complicated), we describe a simple strategy for the case $f_0 = 1$, which is the case we implemented.

Recall that we store the posting lists in gap-encoded blocks. Together with the treap cursor, we will maintain a *list cursor*, which points inside some block that has been previously decompressed. Each time there is no left or right child in the treap, we must search the list for potential elements omitted in the treap. More precisely, we look for elements in the range $[d, id(v_t) - 1]$ if we cannot go left, or in the range $[d, id(u_t) - 1]$ if we cannot go right. Those elements must be processed as if they belonged to the treap before proceeding in the actual treap. Finding this new range $[l, r]$ in the list may imply seeking and decompressing a new block.

The cleanest way to process range $[l, r]$ is to search as if it formed a subtree fully skewed to the right, descending from v_t. If we descended to the left of v_t towards the range, we push v_t into the stack. Since all the elements in the list have the same frequency, when we are required to advance towards (a new) d we simply scan the interval until reaching or exceeding d, and the docid found acts as our new $id(v_t)$ value. When the interval $[l, r]$ is exhausted, we return to the treap. Note that the interval $[l, r]$ may span several physical list blocks, which may be subsequently decompressed.

4.2.3 Unions

The algorithm for ranked unions requires a few changes on the algorithm for intersections. First, in the two lines that call CHANGED($id(v_t)$), we do not change the d for all the treaps when the current treap does not find it. Rather, we keep values $nextd_t$ where each treap stores the minimum $d' \geq d$ it contains, thus those lines are changed by $nextd_t \leftarrow id(v_t)$. Second, we will choose the treap t to advance only among those where $id(v_t) \neq d$ and $nextd_t = d$, as if $nextd_t > d$ we cannot find d in treap t. Third, when all the treaps t where $id(v_t) \neq d$ satisfy $nextd_t > d$, we have found exactly the treaps where d appears. We add up $score(Q, d)$ over those treaps where $id(v_t) = d$, report d, and advance to $d + 1$. If, however, this happens but no treap t satisfies $id(v_t) = d$, we know that d is not in the union and we can advance d with CHANGED($\min_{t \in Q} nextd_t$). Finally, CHANGED($newd$) should not only update d but also update, for all the treaps t, $nextd_t$ to $\max(nextd_t, newd)$.

Algorithm 17 gives the detailed pseudocode. A special case for the method CHANGED has to be implemented for this case, which is depicted in Algorithm 18.

4.2.4 Supporting Different Score Schemes

Arguably the simplest scoring scheme is to use the sum of term frequencies $tf_{t,d}$ of the words involved in a bag-of-words union or intersection query. This case is easy to implement using inverted treaps, since the topology is constructed employing the term frequency as the priority and the term frequencies are represented differentially. A trivial extension is tf-idf scoring: every time we need to calculate U, we multiply the term frequency by the corresponding idf_t, as shown in Algorithm 13 and Algorithm 17. However, in order to support more complex scoring schemes, such as BM25, additional information is required (i.e., document length) and the resulting relative order of documents inside a list may be different from tf. In these cases, creating the treap topology based on the the term frequency $tf_{t,d}$ is not useful. Moreover, if we actually use the exact score, we would require float or double precision numbers, thus

Algorithm 17: Method TREAP_UNION(Q, k). Performs a ranked intersection from query terms Q and returns the top-k documents and scores.

```
TREAP_UNION$(Q, k)$
  $results \leftarrow \emptyset$ // priority queue of pairs $\langle key, priority \rangle$
  for $t \in Q$ do
    $stack_t \leftarrow \langle \perp \rangle$ // stack of treap $t$, $id(\perp) = f(\perp) = +\infty$
    $v_t \leftarrow$ root of treap $t$
    $nextd_t \leftarrow 1$ // next possible value in treap $t$
  end for
  $d \leftarrow 1$, $L \leftarrow -\infty$
  while $d < +\infty$ do
    while $U \leq L$ do
      CHANGED_UNION$(\min_{t \in Q} id(\text{TOP}(stack_t)), stack, U, d)$
    end while
    if $\forall t \in Q,\ d = id(v_t) \vee nextd_t > d$ then
      if $\exists t \in Q,\ d = id(v_t)$ then
        REPORT$(d, U, k, results)$
        CHANGED_UNION$(d + 1, stack, U, d)$
      else
        CHANGED_UNION$(min_{t \in Q} nextd_t, stack, U, d)$
      end if
    else
      $t \leftarrow$ treap of shortest list such that $d \neq id(v_t)$
      if $d < id(v_t)$ then
        $l_t \leftarrow$ left child of $v_t$
        if $l_t$ is not null then
          PUSH$(stack_t, v_t)$, CHANGEV$(t, l_t, U)$
        else
          $nextd_t \leftarrow id(v_t)$
        end if
      else
        $r_t \leftarrow$ right child of $v_t$
        if $r_t$ is not null
          CHANGEV$(t, r_t), U$
        else
          CHANGEV$(t, \text{POP}(stack_t), U)$
          $nextd_t \leftarrow id(v_t)$
        end if
      end if
    end if
  end while
  return $results$
```

Algorithm 18: Method CHANGED_UNION($newd, stack, U$). Changes the current document candidate to be evaluated in the case of top-k unions using treaps.

```
CHANGED($newd, stack, d$)
  $d \leftarrow newd$
  for $t \in Q$ do
    $nextd_t \leftarrow \max(nextd_t, newd)$
    $v \leftarrow v_t$
      while $d \geq id(\text{TOP}(stack_t))$ do
        $v \leftarrow \text{TOP}(stack_t)$
        POP($stack_t$)
      end while
    CHANGEV($t, v, U$)
  end for
```

increasing the size of the index.

An alternative to cope with BM25 is to compute each score at construction time, and build the topology according to the computed score, but still store the term frequency at each node. The query processing algorithm is still valid since we are able to compute the complete score at query time. However, the treap cannot encode term frequencies differentially anymore, since it is possible that a term frequency stored in a node's child is greater than the one of the node itself. If we represent the absolute frequencies, the resulting inverted treap approach is not competitive in terms of space.

In this work we use another approach to this problem. We employ impact-scoring (Section 2.16.1) instead of term frequencies. This enables the inverted treaps to support any type of scoring scheme. The procedure is to construct the treap topology using the pre-calculated impacts, and store them as if they were the term frequencies in the nodes. This allows for differential encoding of impacts, and we can use the same query algorithms without any change.

4.3 Incremental Treaps

So far, we described a *static* representation of posting list using treaps. In this section we show how to extend the inverted treap representation to support incremental updates, that is, to allow the addition of new documents to the collection while the index is loaded in memory.

Incremental in-memory inverted indexes have been developed to cope with the efficiency challenges in Twitter [39] and for indexing microblogs [167]. In these two cases, the more recent posts are generally more relevant, thus appending them at the end of the inverted lists, just as in the indexes designed for Boolean intersections, allows for efficient query processing.

In main memory, the problem of maintaining such an inverted index up to date is simpler, because the price for non-contiguous storage of the inverted lists is not so high. In a thorough recent study, Asadi and Lin [15] show that the difference in query performance between lists cut into many short isolated blocks versus fully contiguous lists is only 10%–20% for Boolean intersections and 3%–6% for ranked intersections.

However, there are cases where we require immediate updating of the index but have no preference for the most recent posts. Obvious examples are online stores like Ebay or Amazon, where new products must be immediately available but they are not necessarily better than previous ones. In those cases, we are interested in ranked retrieval using traditional relevance measures, which are mostly independent of the insertion time.

While this form of dynamization is simple for the WAND and Block-Max formats [15], it is much more challenging for the treaps, because postings are not physically stored in increasing document identifier order, and therefore one cannot simply append the inserted postings at the end of the inverted lists.

4.3.1 Supporting Insertions

Our solution is inspired by the linear-time algorithms for building treaps offline (recall Section 4.1.1). We maintain the rightmost path of the tree in uncompressed form, and their left subtrees are organized into progressively larger compressed structures. This allows for smooth insertion times without large sudden reconstructions, reasonable compression performance and search times.

The main idea is to maintain a treap for each inverted list, as in the static case. However, this treap is only gradually converted into a compressed static structure, and never completely. Some nodes are represented with classical pointers (we call those *free* nodes), whereas some subtrees are represented in the form of static treaps.

The rightmost path is always composed of free nodes. Some nodes descending from the left children of those nodes may also be free, but not many. Each free node v stores the number $\mathcal{F}(v)$ of free nodes in its subtree; these always form a connected subtree rooted at the node. We use a blocking parameter b, so that when a left child v of a rightmost path node has b free nodes or more, all those free nodes are converted into a static treap.

Precisely, when a rightmost node v'_j is converted into the left child of a new incoming node v_i, we check if $\mathcal{F}(v'_j) \geq b$ (since v'_j belonged to the rightmost path, the limit to free nodes did not apply to it, so if v'_j has ℓ rightmost descendants, it could have up to $f(v'_j) = b\ell$ free nodes descending from it). If $\mathcal{F}(v'_j) \geq b$, all those $\mathcal{F}(v'_j)$ free nodes are converted into a static treap and we set $\mathcal{F}(v_i) \leftarrow 1$ for the new node. Otherwise, we set $\mathcal{F}(v_i) \leftarrow \mathcal{F}(v'_j) + 1$.

Hence, the maximum time per insertion is $O(b\ell)$, which is $O(b \log n)$ in expectation. This, however, does not add up to more than $O(n)$, since static treaps are built in linear time and each node becomes part of a static tree only once.

The static treaps we create are not completely identical to the static ones of Section 4.1. In this case, some free nodes may be parents of static treaps. Therefore, the process results in a tree of static treaps (with some free nodes in the top part). To accommodate this extension, the static structure is expanded with a bitvector B that has one bit per leaf of the static treap, indicating whether the leaf is actually a leaf or it contains a pointer to another static treap. Those pointers are packed in an array inside the structure, so that the ith 1 in B corresponds to the ith pointer in the array. Its position is computed with $\textsc{rank}_1(B, i)$.

4.3.2 Gradual Growth

Note that there will be $O(b \log n)$ free nodes *per inverted list* in expectation. Therefore, b must be reasonably small to avoid their pointers blow up the space (in practice, b should not exceed a few thousands). On the other hand, a small b implies that the static treaps may contain as little as b nodes. Thus b should be large enough for the static structures to use little space (otherwise, the constant number of integers and pointers they use may be significant). It may be impossible to satisfy both requirements simultaneously.

To cope with this problem, we enforce a gradual increase of the treap sizes. Static treaps will be classified in *generations*. A static treap T is of generation $g(T) = 1$ when it is first created. No treap can have descendants of lower generations. Each static treap T stores its generation number $g(T)$ and the number $d(T)$ of descendant treaps of its same generation, including itself. It is easy to compute $d(T) = 1 + \sum_{T', g(T')=g(T)} d(T')$ for a newly created static treap T that points to several other existing treaps T'. Given a parameter c, we establish that, when a new static treap T is created and $d(T) \geq c$, that is, its subtree has c or more treaps of its generation, then all those are collected and recompressed into a larger static treap S, which now belongs to generation $g(S) = g(T) + 1$. The same formula above is used to compute $d(S)$ for the new treap.

This technique creates larger and larger static treaps towards the bottom and left part of the tree. Now a node can be reprocessed $\log_c(n/b)$ times along its life, to make it part of larger and larger static treaps; therefore the total construction time becomes $O(n \log n)$ (albeit with a very low constant). Parameter c should be a small constant.

Figure 4.6 shows a normal left-to-right construction process (as described in Section 4.1.1), but it also illustrates how the incremental version is built. We have enclosed in gray sets the nodes that are grouped into static treaps, for $b = c = 2$. In particular, observe the situation in the rightmost cell of the second row. A static treap of generation 1 is created for the nodes with docid 13, 22, and 27. But now this node is of generation 1 and its subtree has $d = 3 \geq c$ treaps of its same generation. Thus, a new static treap, of generation 2, is created with all those generation-1 descendants. This is shown in the leftmost cell of the third row.

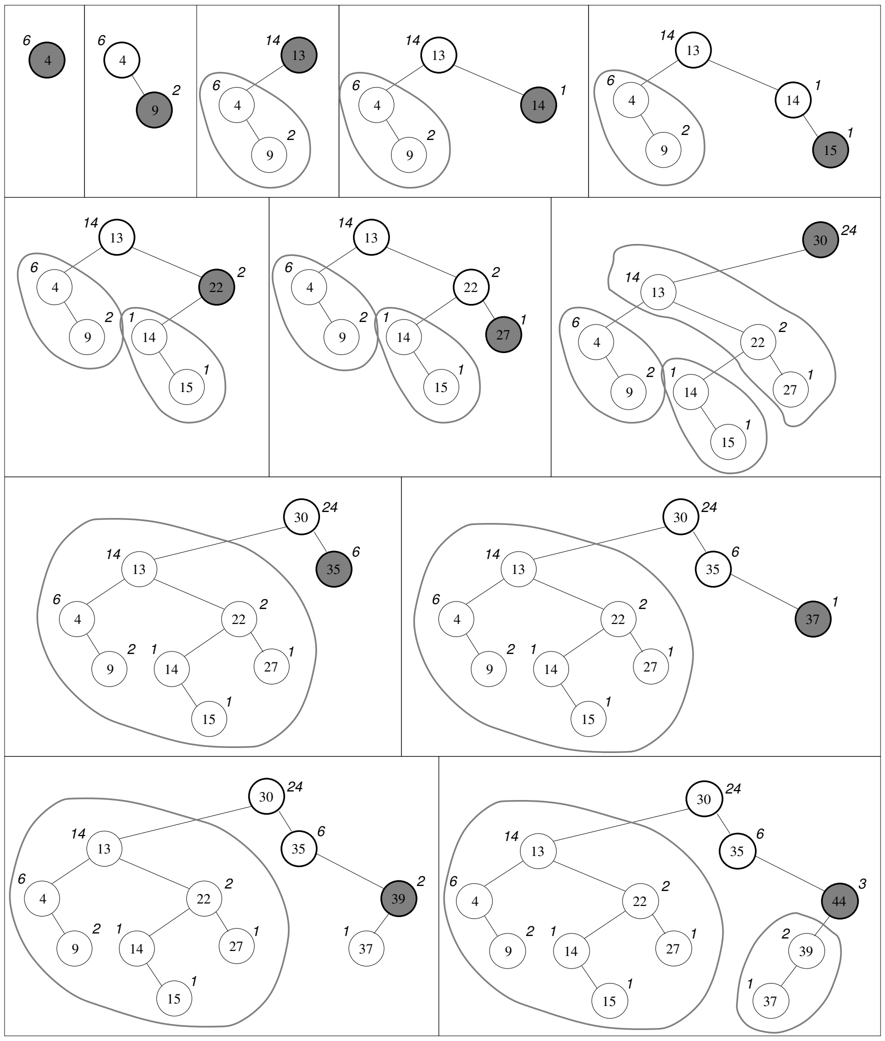


Figure 4.6: The left-to-right construction of an example treap. We show in bold the rightmost path and shade the node that is added in each iteration. Sets of nodes with gray borders indicate static treaps that are built.

Chapter 5

Experimental Evaluation

In this chapter we describe the experimental setup, in terms of the collections used and the environment employed for the experiments. We also explain the engineering details required to implement the indexes and the baselines, and discuss the space/time results obtained.

5.1 Datasets and Test Environment

We use the TREC GOV2 collection, parsed using the Indri search engine[1], and Porter's stemming algorithm. The collection contains about 25.2 million documents and about 39.8 million terms in the vocabulary. The inverted lists contain about 4.9 billion postings in total. After pre-processing and filtering, a plain representation of the GOV2 collection requires about 72GB. The average posting list length is 76 and the average document contains 932 words.

We also performed experiments using other collections, such as the English Wikipedia dump, containing about 5 million documents and 6 million terms. An average Wikipedia article contains 352 words and the average number of elements in a posting list is 132. We also performed experiments on the Weblogs collection[2], containing about 50 million documents and requiring 120GB of space. For queries, we used the 50,000 TREC2005 and TREC2006 Efficiency Queries dataset with distinct numbers of terms, from $q = 2$ to 5. We omitted those not appearing in our collection, thus we actually have 48,583 queries.

In this work we only show the results obtained using the TREC GOV2 collection, since the results over the others do not change significantly.

[1]`http://www.lemurproject.org/indri/`
[2]`http://www.icwsm.org/data/`

5.1.1 Baselines and Setup

We compare our results with five baselines: (1) Elias Fano WAND implementation [127], (2) Block-compressed WAND Implementation [36], (3) Block-Max [59], (4) Dual-Sorted [100] and (5) ATIRE [158].

For Elias Fano WAND, we use the implementation [3] provided by the authors [127]; we denote this implementation EF in the charts. For PforDelta WAND, we use the implementation obtained from the `SURF` framework[4], we denote this implementation WAND in the charts. For Block-Max, we adapted the implementation of [132] by extending it to support ranked intersections and including it into the `SURF` framework. For both WAND and Block-Max we use the optimal PForDelta encoding for the docids and Simple9 encoding for the frequencies, as these gave the smallest indexes. The encoding implementations were obtained from the FastPFor library[5]. In both cases the posting lists were encoded using blocks of 128 values. In the case of Block-Max, we also store the maximum value for every block. We denote Block-Max as BMAX in the charts.

We use ATIRE [6] as our baseline for impact-sorted indexes. ATIRE is an open-source search engine that supports different early termination algorithms based on impact or frequency sorted posting lists. We constructed both, frequency-sorted and quantized BM25 ($q = 8$) impact-sorted indexes for our experiments. All baselines were modified when needed, to support both quantized impact BM25 scores (Section 2.16.1) and tf-idf scoring. Our experiments were run on a dedicated server with 16 processors Intel Xeon E5-2609 at 2.4GHz, with 256 GB of RAM and 10 MB of cache. The operating system is Linux with kernel 3.11.0-15 64 bits. We used GNU `g++` compiler version 4.8.1 with full optimizations (`-O3`) flags.

5.1.2 Dual-Sorted Implementation

In the case of Dual-Sorted (see Section 3.3), we make the following practical considerations to implement this data structure. The wavelet tree is represented using a pointerless version [46] because D is considerably large compared to n. We use the `RRR` (recall Section 2.6.1) bit sequence representation for the wavelet tree, in order to save space.

We do not use the complex mapping of the bit sequence s_t; this is replaced by an array using $V \log N$ bits, we call it $Ptrs$ (see Figure 3.1) from the terms t to the starting positions of the lists L_t in L. The weights within the range of each list L_t, which are decreasing, are represented in differential form using Rice codes (recall Section 2.5). Absolute values are sampled every K positions, where K is a space/time tradeoff parameter. In fact, as there are many runs of equal weights, we store only the nonzero differences, and store a bitmap $W'[1, n]$ indicating which differences are nonzero. So we actually access position $W[\textsc{rank}_1(W', i)]$ instead of $W[i]$. We use the `RG` representation that uses 5% extra space on top of the bit

[3]`http://github.com/ot/partitioned_elias_fano`

[4]`http://github.com/simongog/surf`

[5]`https://github.com/lemire/FastPFor`

[6]`http://atire.org/index.php?title=Main_Page`

sequence W' to support RANK (recall Section 2.6).

All data structure employed in the implementation of this data structure were obtained from the `LIBCDS`[7] library.

5.1.3 Inverted Treaps Implementation

We implemented our indexes based on the `sdsl-lite` library [77]. The document ids are stored in a `dac_vector` of fixed width 6, which gave the best results at parameter tuning time. The weights are stored in a `dac_vector` of fixed width 2.

The f_0 list are represented using PForDelta using a similar implementation to the ones used in WAND and Block-Max. We do not use inverted treaps to represent every posting list, but only those containing at least 1024 elements. The other posting lists are represented using WAND. At query time, if necessary, they are fully decompressed and handled by maintaining a pointer to the current docid being evaluated.

The BP topology is implemented using the `bp_tree` class, while the LOUDS topology is implemented using the `bit_vector` class enhanced with rank operations, for which we use the alternative dubbed `rank_support_v5`, which requires 5% extra space. The HEAP implementation uses an integer vector `int_vector` requiring $\lceil \log(X+1) \rceil$ bits, where X is the maximum element in the array P'. The implementation of the bit sequence for the topology is the same one as the one employed in LOUDS.

We perform all the experiments for tf-idf score and for quantized BM25 impact scoring, as described in Section 4.2.4, using 8 bits for each impact.

5.2 Results

5.2.1 Index Size

We start by showing the size required of each index in Figure 5.1, separated by the scoring scheme used. The left part of the figure shows the case of tf-idf scoring, where EF is clearly the smallest alternative. The second alternative, WAND is about 10% bigger than the EF index.

On the other hand, Block-Max requires more space than any of the treap alternatives, LOUDS being about 10% smaller than Block-Max and HEAP almost equal. Dual-Sorted, on the other hand, is the most space-consuming alternative. On the right side, which shows BM25 scoring, all the inverted treap alternatives require more space than the baselines, climbing from about 13% of the text space under tf-idf scoring to up to 18%. This is mainly because, when using the quantized BM25 score, the number of posting lists elements having

[7] `http://www.github.com/fclaude/libcds/`

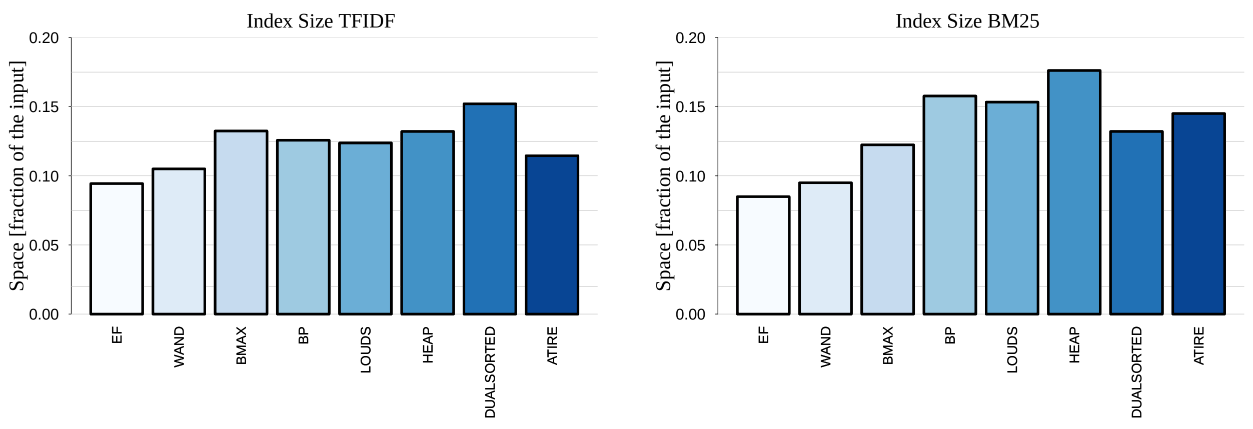


Figure 5.1: Total sizes of the indexes depending on the scoring scheme.

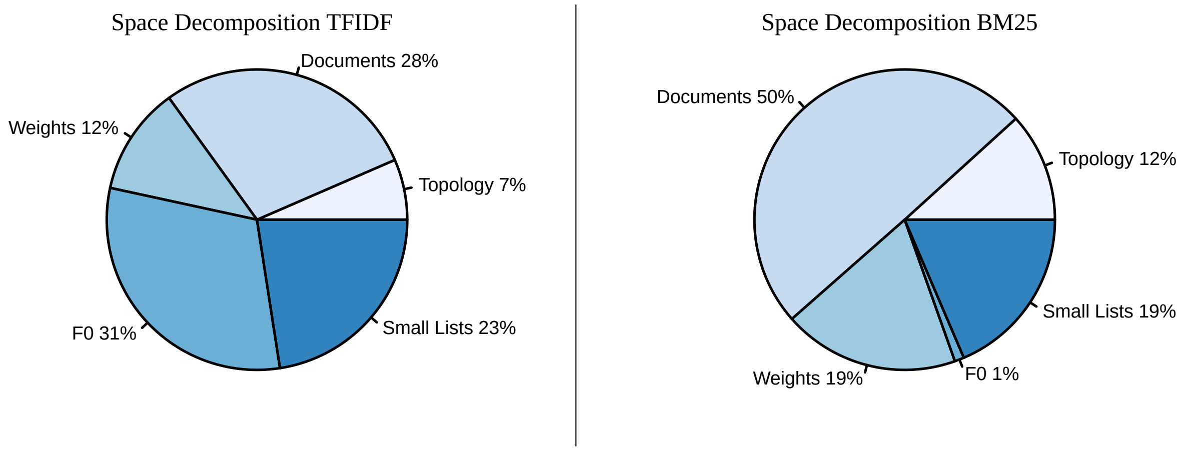


Figure 5.2: Total sizes of the indexes depending on the scoring scheme.

score 1, which is efficiently represented using the f_0 lists, is considerably reduced: for the tf-idf score scheme there are about 3 billion posting list elements with frequency 1, but this decreases to 400 million under BM25.

Figure 5.2 shows the space breakdown of the inverted treap components, using tf-idf scoring on the left and BM25 on the right. This figure is based on our smallest case, which is the LOUDS alternative. The component *Small Lists* represents all the posting lists that have less than 1024 elements and are represented using WAND. For the tf-idf case, we see that the topology requires 7% of the total index size, and the biggest component is the f_0 lists. However, in the BM25 case, the f_0 lists use a negligible amount of space, and the document ids is the heaviest component.

The only component that changes between our three alternatives is how we represent the treap topology. Figure 5.3 shows the difference in space requirements, depending on the scoring scheme. We see that LOUDS is always the smallest alternative, and HEAP is the biggest, requiring about twice the size of LOUDS.

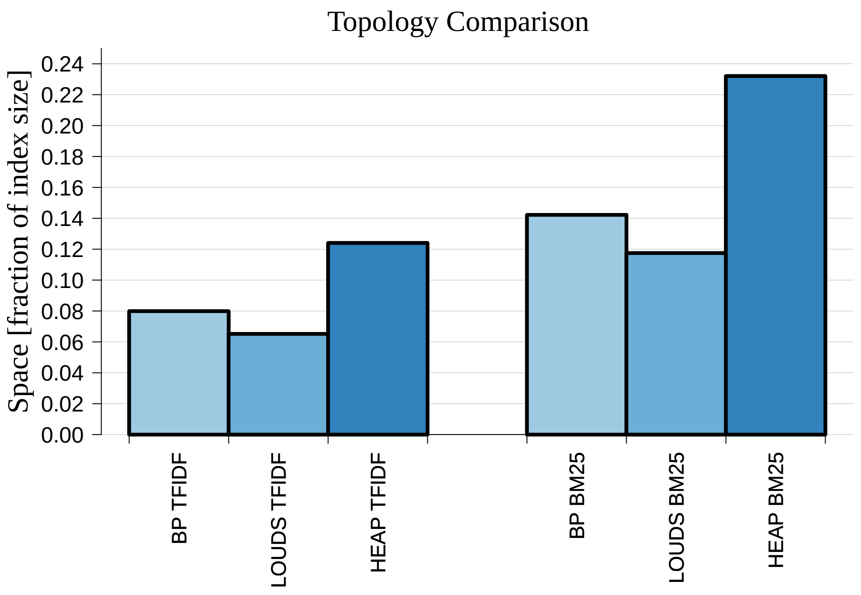


Figure 5.3: Sizes of the topology components depending on the scoring scheme.

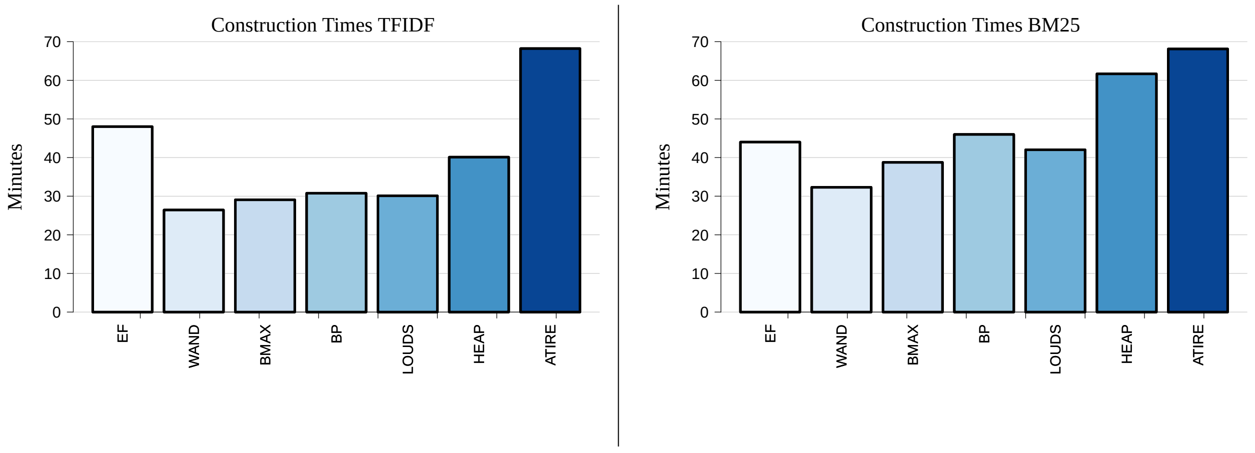


Figure 5.4: Construction time of the indexes depending on the scoring scheme, in minutes.

5.2.2 Construction Time

Figure 5.4 shows the time required to build each index. We see that the construction times of the inverted treap alternatives are not so distant from those of the baseline inverted index representations. This holds except for the HEAP alternative, which is about twice as slow to build than the fastest baseline. It is interesting to note that the Elias-Fano WAND approach is 1.4 to 1.8 times slower than the block-compressed WAND implementation. ATIRE was the slowest alternative in both cases. We do not include the time required to build the Dual-Sorted index, since the construction is not optimized and was above 200 minutes.

5.2.3 Ranked Union Query Processing

We describe the time results for the processing of ranked union queries. We first discuss the results globally and then consider how they evolve as a function of k or the number of words in the query.

Global analysis Figure 5.5 (left) shows the average time per query, for distinct values of k, using the tf-idf scoring scheme. These times average all the queries of all the lengths (2 to 5 terms) together.

The results show that EF, WAND and Dual-Sorted are not competitive for these queries, as they are sharply outperformed by Block-Max. In turn, all our inverted treap alternatives outperform Block-Max by a wide margin. The differences become less drastic as we increase k, but still for $k = 1000$ the HEAP alternative is more than 3 times faster than Block-Max. Our LOUDS alternative is always slower than HEAPS, and BP is slower than LOUDS. Still, BP is almost twice as fast as Block-Max even for $k = 1000$. We discarded results from ATIRE in most of the following figures since it required more than 500 milliseconds on average.

Figure 5.5 (right) shows the distribution of the results using the BM25 quantized score scheme. The differences are much smaller in this case, and in particular our LOUDS and BP variant are the slowest. Our HEAP alternative, instead, is still the fastest or on par with the fastest.

The worse performance of our variants under BM25 owes to the fact that most of the lists are stored as treaps, whereas under tf-idf many of them are stored as f_0 lists. The union algorithm performs a significant amount of sequential traversal, and the simple f_0 lists are faster at this than the treaps. Instead, the considerable improvement obtained by EF and WAND owes to the narrower universe of impacts, which increases the chances that the lower bound θ is not reached along the process and enables more frequent skipping (recall Section 3.2.1). Up to a lesser, extent, Block-Max also improves thanks to more frequent skipping, as on a narrower universe it is also less likely to outperform the current kth highest score. Finally, the significant improvement of Dual-Sorted owes to the fact that it implements the method of [131] (which does not give exact results, so the comparison is not totally fair) and this method is also favoured by the BM25 quantized scores: it reaches sooner a stable situation where the upcoming scores are no better than those already obtained. The optimal-partitioned Elias-Fano implementation was consistently slightly slower than the block-compressed WAND implementation, so we will only consider the latter alternative for the rest of comparisons in this work.

We show more detailed results grouped by percentiles in Table 5.2.3. In the case of tf-idf, the best alternative by far is the LOUDS treap for all percentiles. For the quantized BM25 case, BMAX, BP, LOUDS and HEAP have very similar results, HEAP being the best alternative. In general terms, the table shows that our improved time results are consistently better, and are not blurred by a high variance.

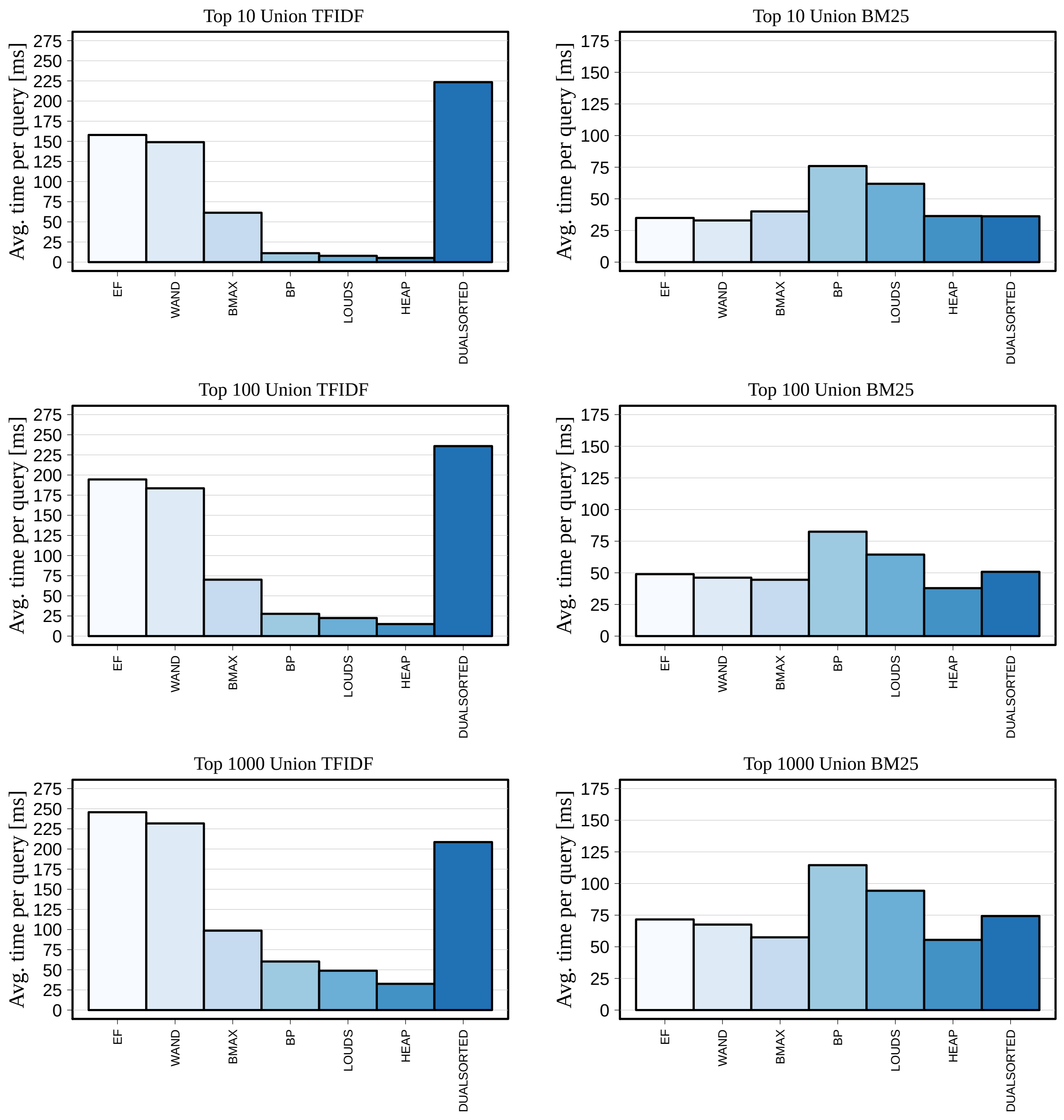


Figure 5.5: Ranked union times for distinct k values, in milliseconds. We show the case of tf-idf scoring on the left and quantized BM25 on the right.

Analysis as a function of k and separated by query length Figure 5.6 separates the times according to the number of words in the query, and shows how times evolve with k, using tf-idf scoring. Note that the times are independent of k for some techniques, or grow very slowly with k in the others.

For 2 query terms, all the inverted treap alternatives are an order of magnitude faster than WAND and Dual-Sorted. For small k values, Block-Max is about twice as slow as the fastest treap alternative, HEAP. For large k values, instead, HEAP is about 5 times faster than Block-Max. The LOUDS and BP alternatives are also faster or on par with Block-Max.

	Union TFIDF					Union BM25				
Index/Percentile	**50%**	**80%**	**90%**	**95%**	**99%**	**50%**	**80%**	**90%**	**95%**	**99%**
WAND	177	462	701	787	918	30	77	116	141	168
BMAX	28	139	195	246	332	20	74	122	169	222
BP	10	27	47	73	94	60	80	156	229	294
LOUDS	7	21	39	61	80	40	62	103	123	231
HEAP	40	177	289	457	617	20	49	76	96	156
DualSorted	151	416	736	866	1101	25	70	121	155	202

Table 5.1: Ranked union results grouped by percentiles. The numbers indicate the maximum time reached by $X\%$ of the queries.

As the number of query terms increases, however, Block-Max starts to outperform the slower inverted treap alternatives. Still, HEAP is always faster than Block-Max and 3–4 times faster than WAND and Dual-Sorted.

Figure 5.7 shows the same experiment on the quantized BM25 scoring scheme. In this case, the treap alternatives are competitive only on queries of 2 and 3 words. In particular, our fastest approach, HEAP, is twice as fast as Block-Max for 2 words, but up to 50% slower on the longest queries. Still, we note that most real queries are short. For example, the average query length has been measured in 2.4 words [150], and in our dataset of real queries, 44% of the multi-word queries have indeed 2 words.

Analysis as a function of query length and separated by k value Figure 5.8 shows the ranked union times as a function of the query length, for distinct k values. As mentioned before, in the case of tf-idf (left side of the figure), our fastest approach HEAP is consistently faster than all the other alternatives, for all query lengths and up to $k = 1000$. In the case of BM25 (right side of the figure), our HEAP alternative is competitive when 2 or 3 query terms are involved. In general, the costs grow linearly with the number of query terms, but the growth rate of WAND and Dual-Sorted is higher on tf-idf and lower on BM25.

5.2.4 Ranked Intersection Query Processing

We proceed to describe the time results for processing ranked intersection queries. As before, we first discuss the results globally and then consider how they evolve as a function of k or the number of words in the query. We do not include the results from ATIRE since it does not support a native mechanism to perform top-k ranked intersection.

Global analysis Figure 5.9 (left) shows the average time per query, for distinct values of k, using the tf-idf scoring scheme. These times average all the queries of all the lengths (2 to 5 terms) together.

As expected, Block-Max always outperforms WAND by a significant margin. Among our alternatives, as before, HEAP is always faster than LOUDS and this is faster than BP. Our

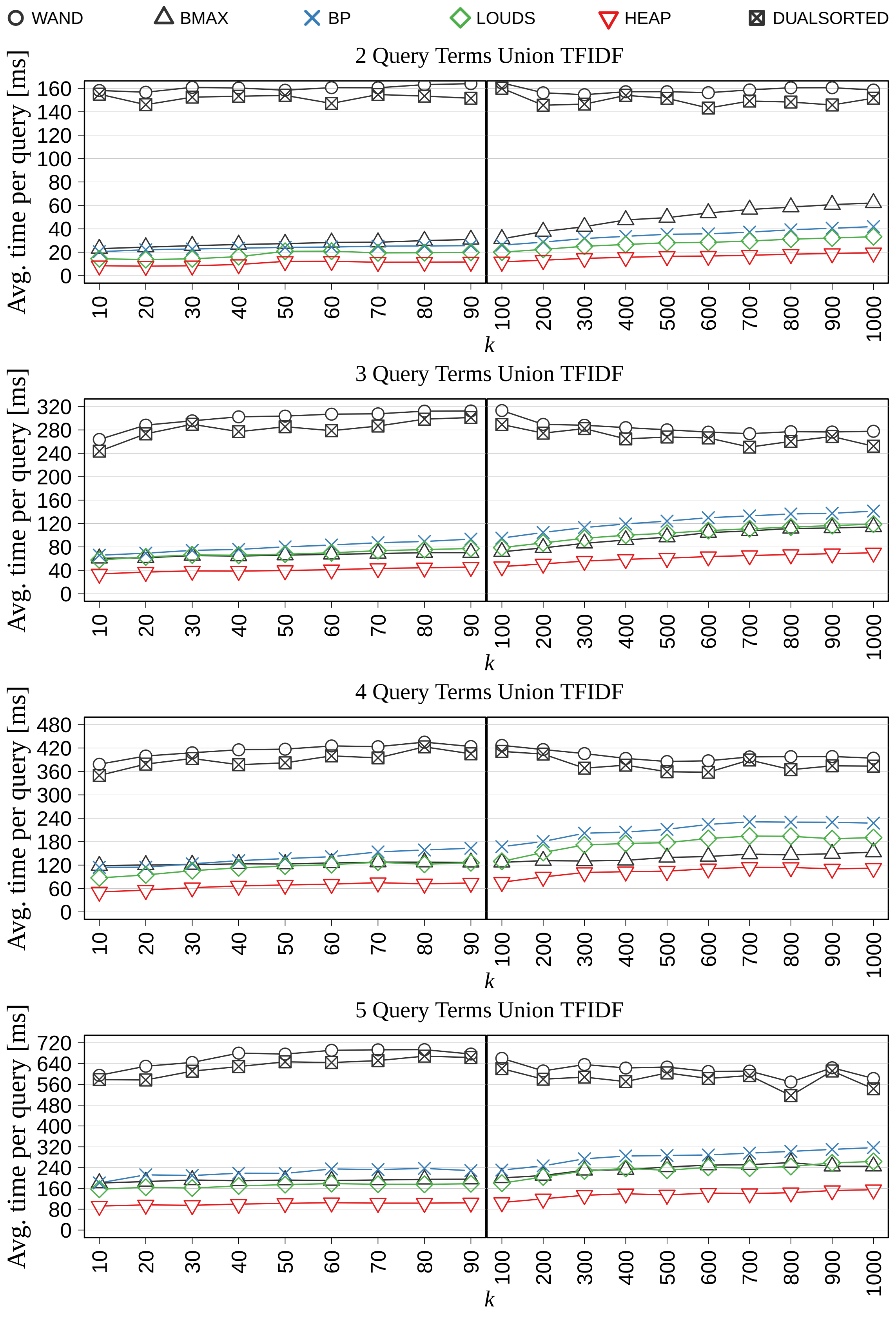


Figure 5.6: Ranked union times as a function of k, grouped by number of terms per query, using tf-idf.

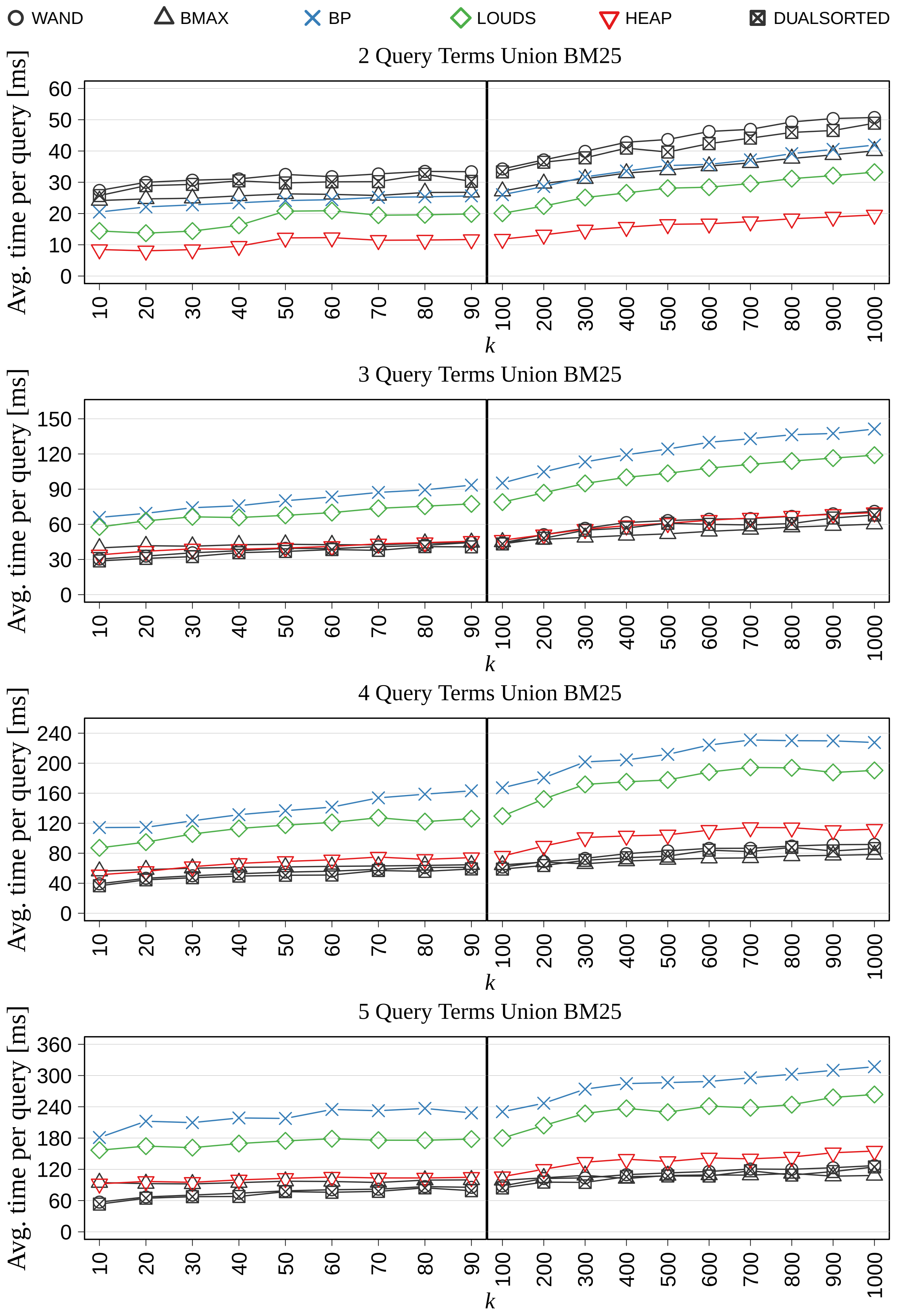


Figure 5.7: Ranked union times as a function of k, grouped by number of terms per query, using BM25.

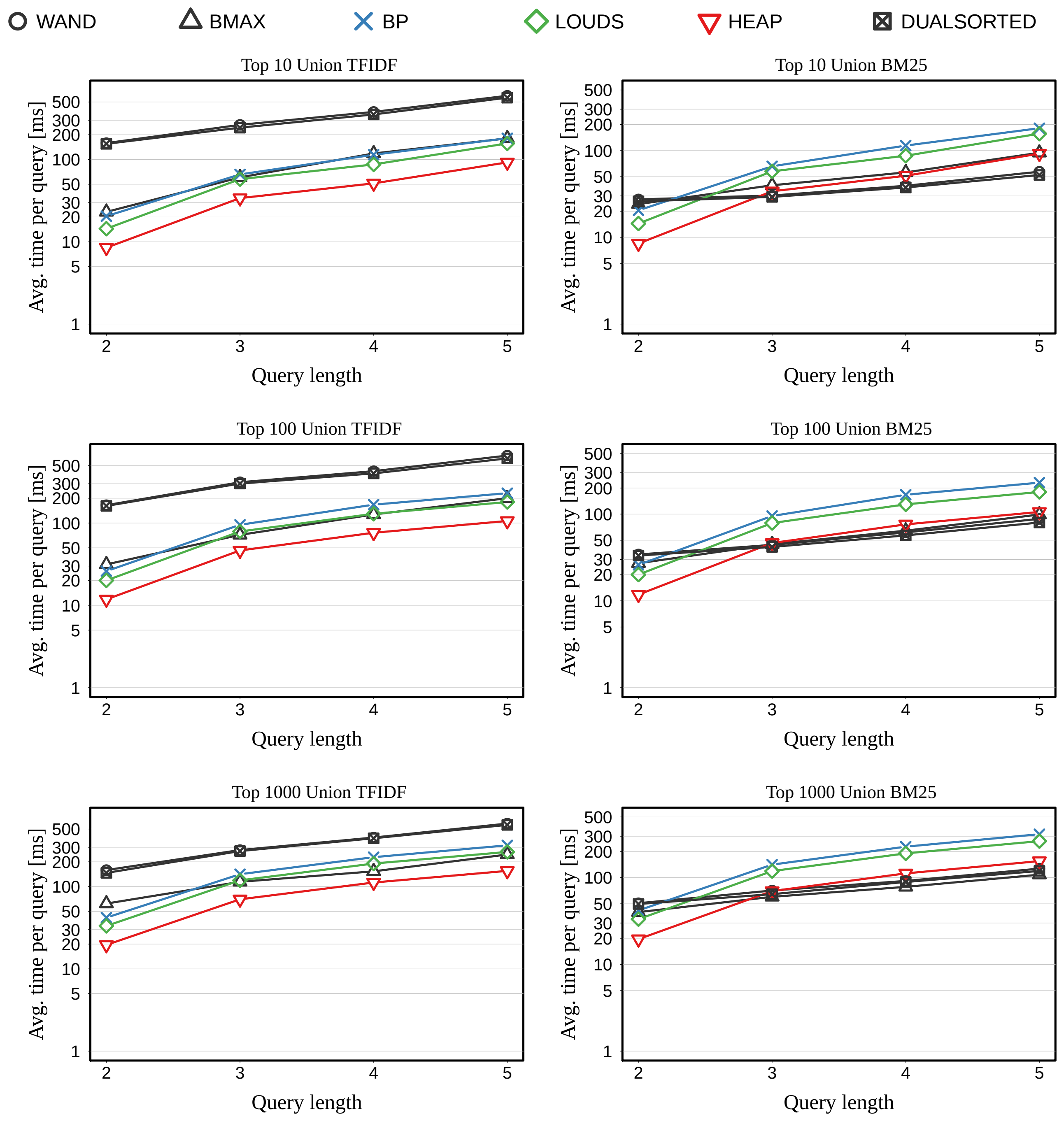


Figure 5.8: Ranked union times for distinct k values as a function of the query query length. We show the case of tf-idf scoring on the left and quantized BM25 on the right.

results are better for small k, where HEAP outperforms all the other indexes by a factor of 2 or more. The difference narrows down for larger k, but still for $k = 1000$ we have that HEAP is faster than Block-Max. Dual-Sorted performs a Boolean intersection and then computes the score of all the qualifying documents. The experiment shows that, for $k = 1000$, this becomes (slightly) better than the more sophisticated alternatives that try to filter the documents on the fly. The optimal-partitioned Elias-Fano implementation was consistently slightly slower than the block-compressed WAND implementation.

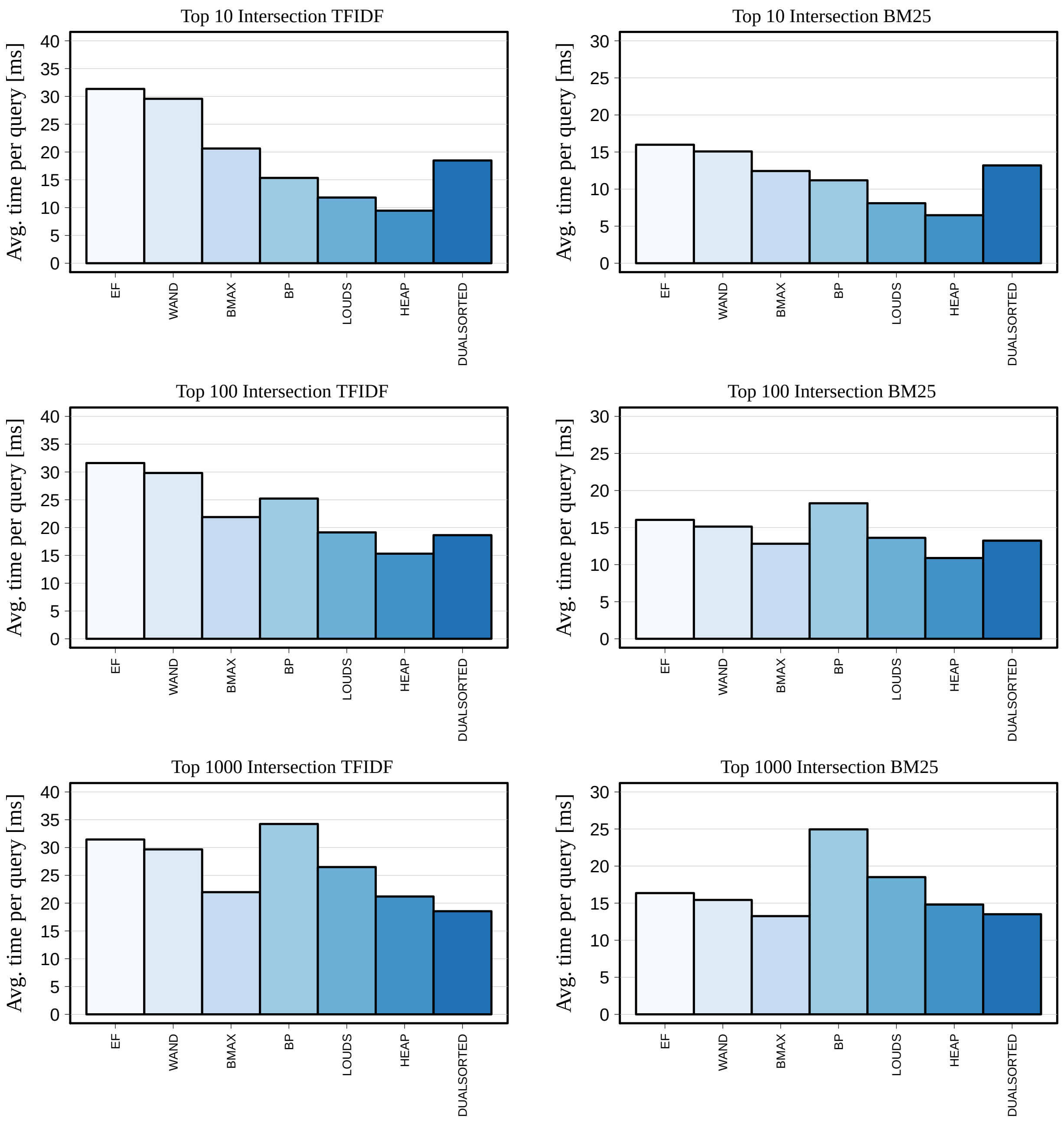


Figure 5.9: Ranked intersection times for distinct k values, in milliseconds. We show the case of tf-idf scoring on the left and quantized BM25 on the right.

Figure 5.9 (right) shows the distribution of the results using the BM25 quantized score scheme. Unlike in the case of unions, the powerful filtration enabled by the treaps outweighs its slowness compared to traversing an f_0 list. As a result, the inverted treaps are faster on BM25 than on tf-idf scores. The methods WAND and Block-Max also improve thanks to the quantized scores. Dual-Sorted also improves: even if it always performs the same Boolean intersection and then computes the scores of the surviving candidates, this computation is faster because it uses the stored quantized scores, whereas for tf-idf it must multiply each stored tf by the idf associated with the query term. The comparisons between all the alternatives stay, overall, similar as in the case of tf-idf scoring.

	Intersection TFIDF					**Intersection BM25**				
Index/Percentile	**50%**	**80%**	**90%**	**95%**	**99%**	**50%**	**80%**	**90%**	**95%**	**99%**
WAND	4	43	93	178	451	9	35	89	128	223
BMAX	29	146	209	260	321	3	29	62	93	156
BP	17	42	58	68	77	18	43	67	85	101
LOUDS	13	22	34	47	60	12	29	44	56	67
HEAP	3	12	18	23	34	9	21	32	42	51
DualSorted	24	132	230	287	386	9	33	84	122	212

Table 5.2: Ranked intersection results grouped by percentiles. The numbers indicate the maximum time reached by $X\%$ of the queries.

We show detailed results grouped by percentiles in Table 5.2 for all alternatives that were considered. In both cases, tf-idf and BM25, the best alternative is the treap using the HEAP topology. In general terms, the table shows that our improved time results are consistently better, and are not blurred by a high variance.

Analysis as a function of k and separated by query length Figure 5.10 shows how times evolve with k, using tf-idf scoring. As in the ranked unions, some techniques are independent of k and others (in this case, the inverted treaps with 2 or 3 words) grow slowly with k.

For 2 query terms, the HEAP alternative is the fastest up to $k = 300$, from where Block-Max takes over. For 3 and 4 words, HEAP is either the fastest choice or very close to it, and for 5-word queries it takes over again. As mentioned by [59], the performance of Block-Max is considerably affected by the number of terms participating in the query. For 4 query terms, it is one of the slowest alternatives, together with the BP inverted treap variant. For 5 query terms, it is definitely the slowest. This explains the poor performance of Block-Max compared to WAND when considering all the queries together.

Figure 5.11 shows the results for ranked intersections on the quantized BM25 scoring scheme. While the general picture is similar to the case of tf-idf, HEAP is overcomed more frequently. For few-word queries it is outperformed sooner by Block-Max, for $k = 200$ on 2 words and for $k = 80$ on 3 words. On 4-word queries, it is almost always outperformed by a small margin. It is again the fastest alternative on 5 words, but by a smaller margin than for tf-idf. Block-Max is also heavily affected as the number of words increases.

Analysis as a function of query length and separated by k value Figure 5.12 shows the ranked intersection times as a function of the query length, for distinct k values. On the left side of the figure we show the results of the tf-idf scoring scheme. We can see more clearly how Block-Max is heavily affected by the query length. The others stay unaltered or fluctuate as a function of the number of words. This is because, as this number increases, more lists have to be handled, but it is also more likely to filter out portions of the lists. The interaction of the two effects produces increments and decrements in the query times. Recall that WAND and Dual-Sorted perform a Boolean intersection followed by the evaluation of

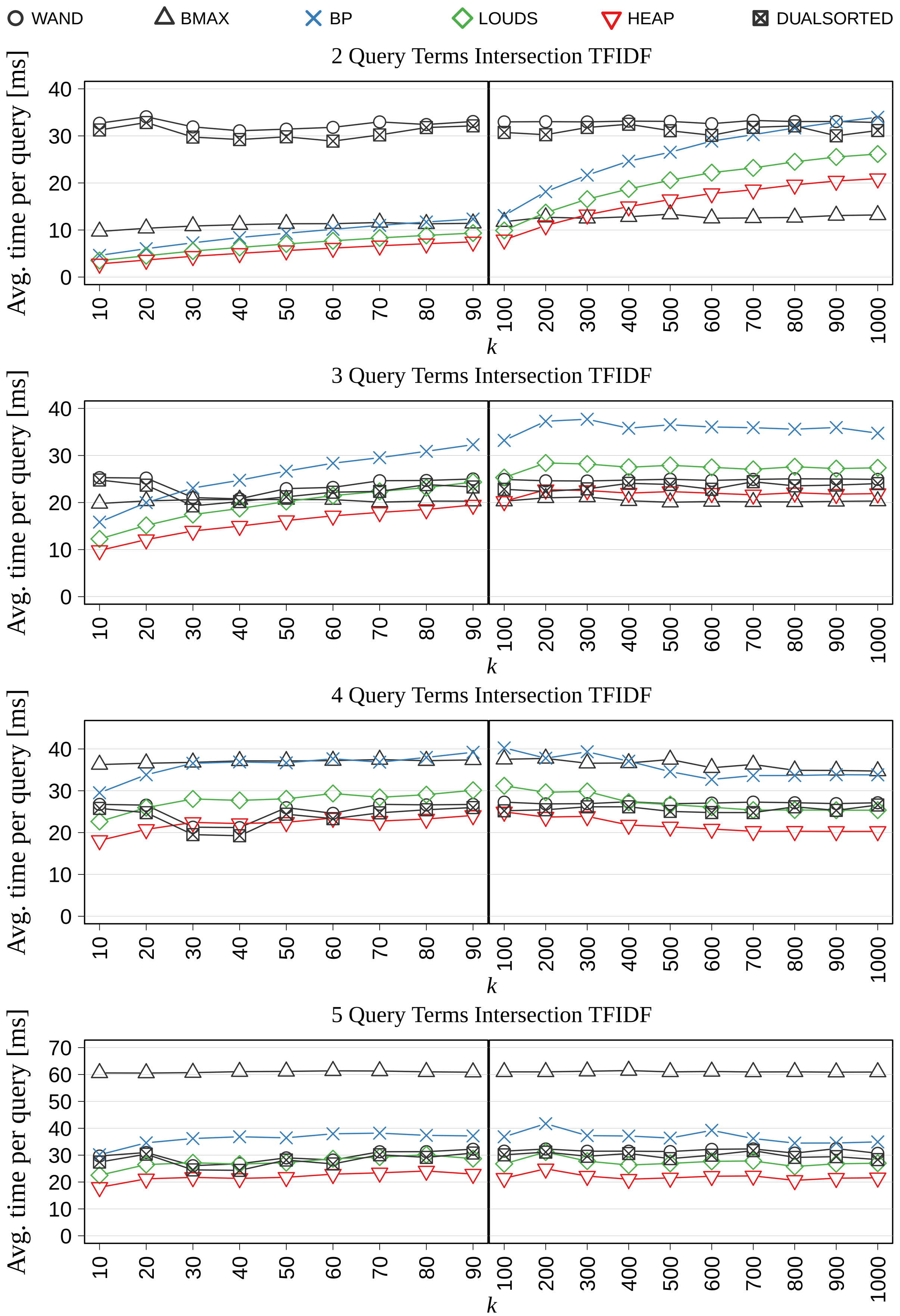


Figure 5.10: Ranked intersection times as a function of k, grouped by number of terms per query, using tf-idf.

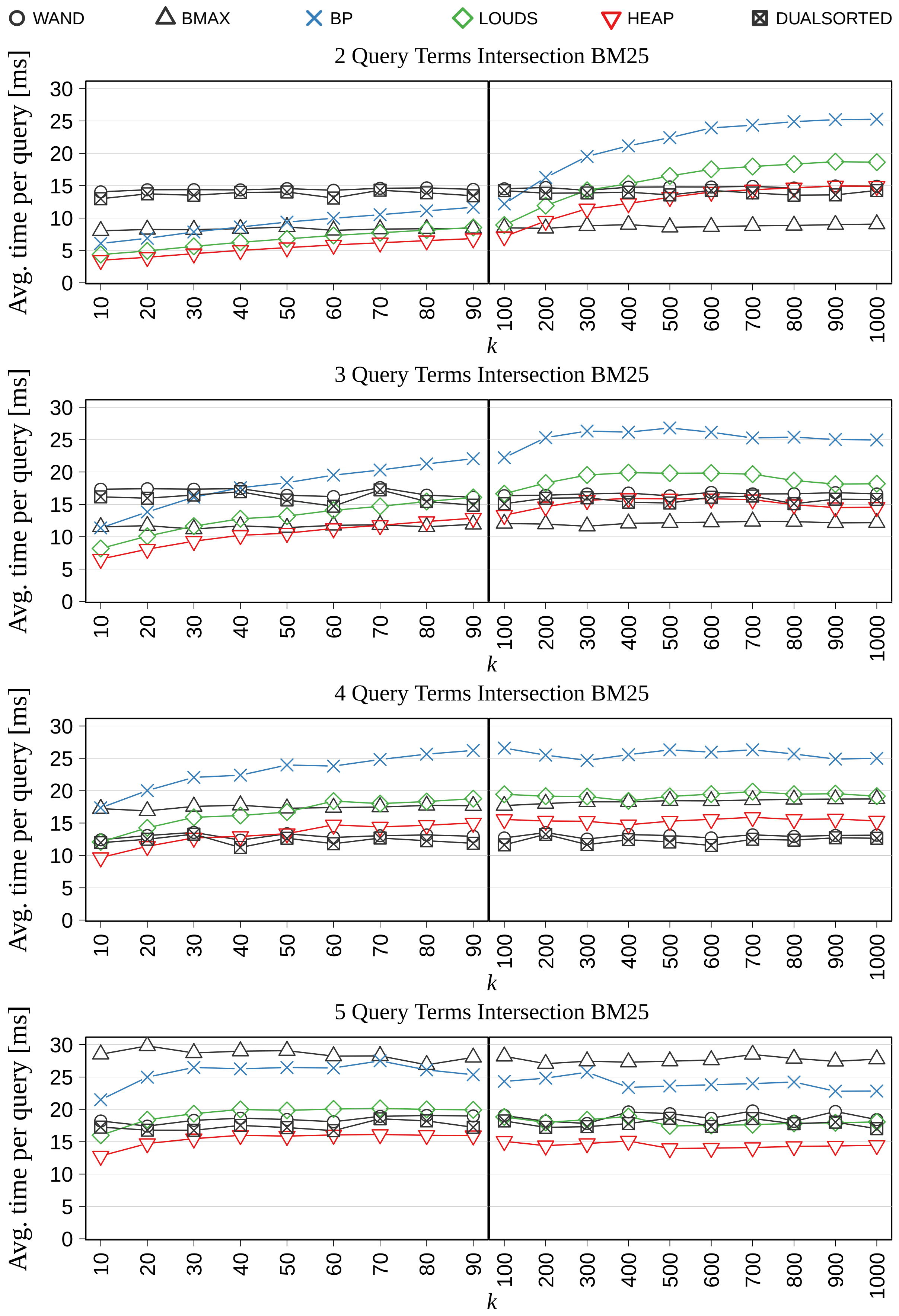


Figure 5.11: Ranked intersection times as a function of k, grouped by number of terms per query, using BM25.

all the resulting scores, so their behaviour is very similar. Note that our HEAP alternative is generally the best on tf-idf, whereas on BM25 it is the best for $k = 10$ and in some cases for larger k.

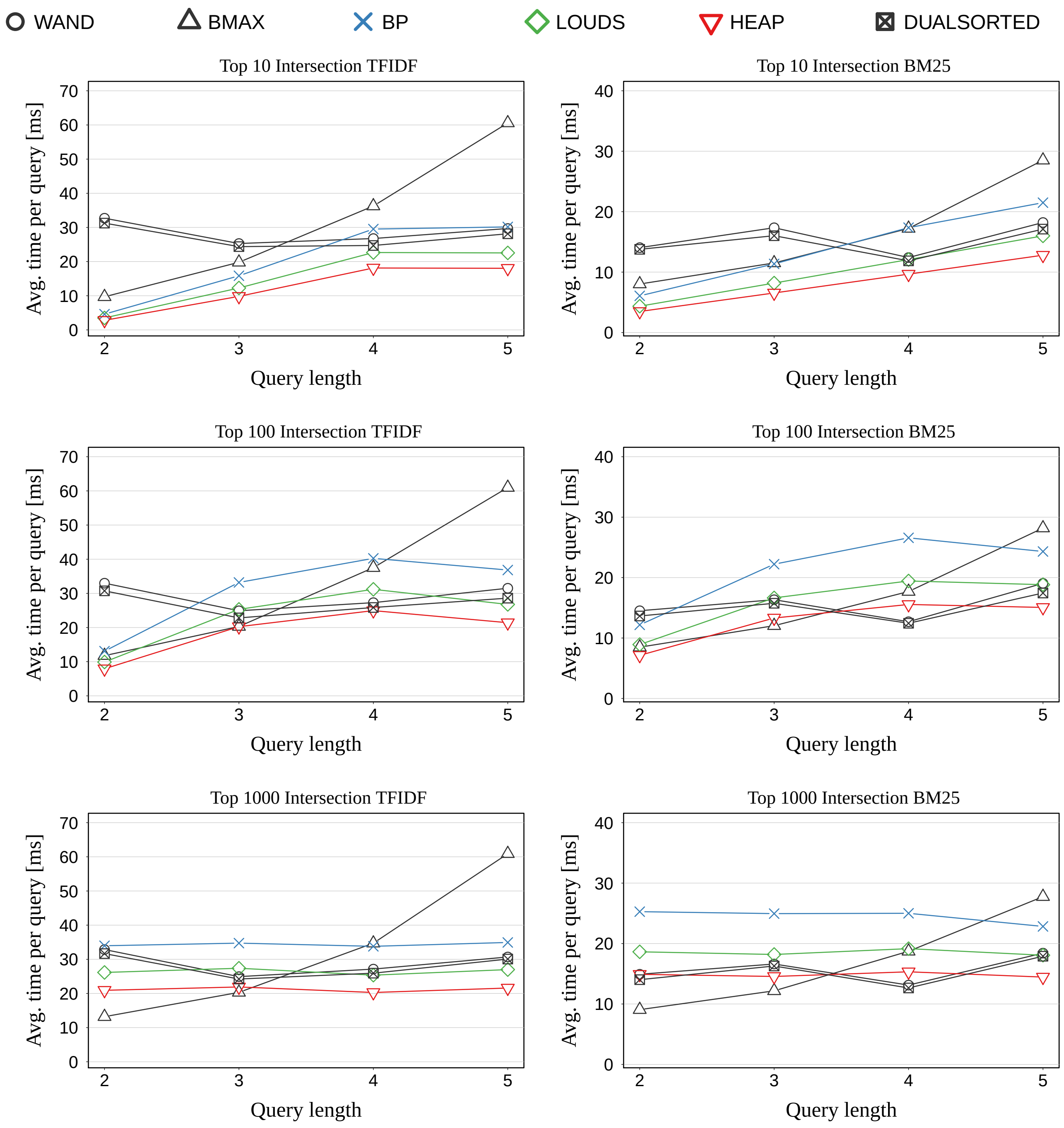


Figure 5.12: Ranked intersection times for distinct k values as a function of the query length. We show the case of tf-idf scoring on the left and quantized BM25 on the right.

5.2.5 One-word Queries

We have not yet considered the simplest one-word queries, which account for a significant percentage of typical queries (almost 24% in our query set). For these queries, we must obtain the k highest-ranked documents from a single inverted list. In the case of WAND, this requires traversing the whole list and retaining the k highest scores. Block-Max speeds this up by skipping blocks where the maximum score is not higher than the kth score we already know. Dual-Sorted, instead, simply requires to extract the first k elements from the list of the query term, as its lists are sorted by decreasing frequency. Therefore the Dual-Sorted time is bounded by $O(k \log D)$, as it is implemented on a wavelet tree. This is the best scenario for ATIRE, since the posting lists are sorted by either frequencies or quantized scores, so returning the k best documents is done simply by traversing the first k postings.

For our inverted treaps, we use a simplification of the procedures for ranked unions and intersections. We insert the root of the treap in a heap that sorts by decreasing score. We then iteratively extract the top of the heap, report its document, and insert its two children. Therefore we require $O(k)$ operations on the treap and the heap, leading to total time $O(k \log k)$.

Figure 5.13 shows the time performance. The time differences are so significant that we have used logarithmic scale. Our fastest variant, HEAP, requires from 5–10 microseconds per query with $k = 10$ to 100–200 with $k = 1000$, whereas our slower variant, BP, requires 10–20 to 200–500 microseconds. Dual-Sorted, instead, goes from 100–200 to 2000–5000 microseconds, that is, around 20 times slower than HEAP. The slowest technique in the plots is Block-Max, which requires 1–5 milliseconds per query, that is, 25–200 times slower than HEAPS. This is because its time is not bounded in terms of k. Still, the time of Block-Max increases with k because its ability to filter blocks decreases as k grows. On the other hand, the times of the other indexes grows more or less linearly with k after a query initialization time: A rough fitting gives, for HEAP, $0.15k + 6.5$ microseconds on tf-idf and $0.10k + 5$ on BM25; for BP it gives $0.5k + 10$ on tf-idf and $0.3k + 5$ on BM25; and for DualSorted it climbs to $4k + 100$ on tf-idf and $2k + 80$ on BM25. ATIRE is clearly the fastest alternative for single term queries, as it takes $0.5k$ microseconds on both cases. We have not included the times of WAND as they are much higher, 15000 microseconds almost independently of k (note that WAND needs to decompress the whole list, independently of k, and then find the k largest scores).

5.2.6 Incremental Treaps

Figure 5.14 shows the time required to insert increasing prefixes of the GOV2 collection on our incremental inverted treaps. For these experiments, we use the block parameter $b = 1024$ and recompress any subtree that has generation $c = 16$ into a new larger static treap (recall Section 4.3.1). These values were chosen by parameter tuning experiments. As a baseline, we consider the static inverted treap construction, for LOUDS and HEAP, on the same prefixes. The figure shows that the time for incremental treap construction grows slightly superlinearly, as expected from its $O(n \log n)$ time complexity. The static construction,

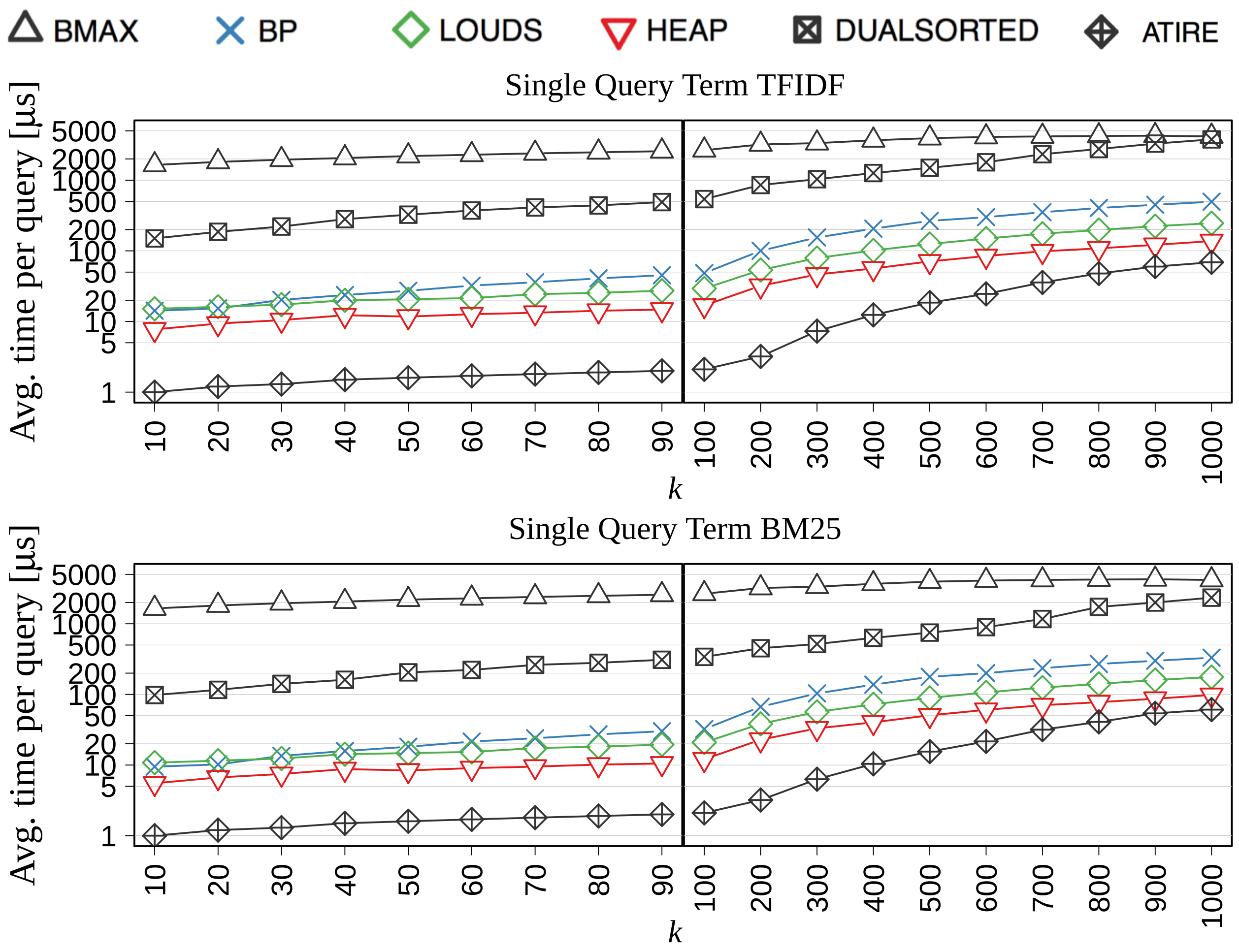


Figure 5.13: One-word query times as a function of k, using tf-idf (top) and BM25 (bottom) scoring schemes. Note the logscale.

instead, displays linear-time performance. Still, after inserting 24 million documents, the incremental construction is only twice as slow as the fastest static construction (LOUDS) and 40%–60% slower than the static construction giving the best query times (HEAPS).

In terms of memory usage the incremental treap requires an additional bitvector, causing an increase in the overall size of about 7%. However, the size occupied by the *free* nodes is considerably larger, using about 40% more space. This is because the free nodes are not compressed in any way, that is, we are using 64-bit pointers, and 32-bit integers for the the docid and the weight. In addition, we need to keep track of counters for the block parameter b and the generations for parameter c. For these variables we use integers of 16 and 8 bits, respectively. In total, the incremental variant is about 50% larger than the static LOUDS variant.

The incremental treap is also about 45% slower than the LOUDS implementation in all cases of ranked unions and intersections, and more than twice as slow as HEAPS. We included the times of BP, which show that the incremental treap is still slightly slower than it. There are two main reasons for such a degradation in the performance: First, the free nodes are

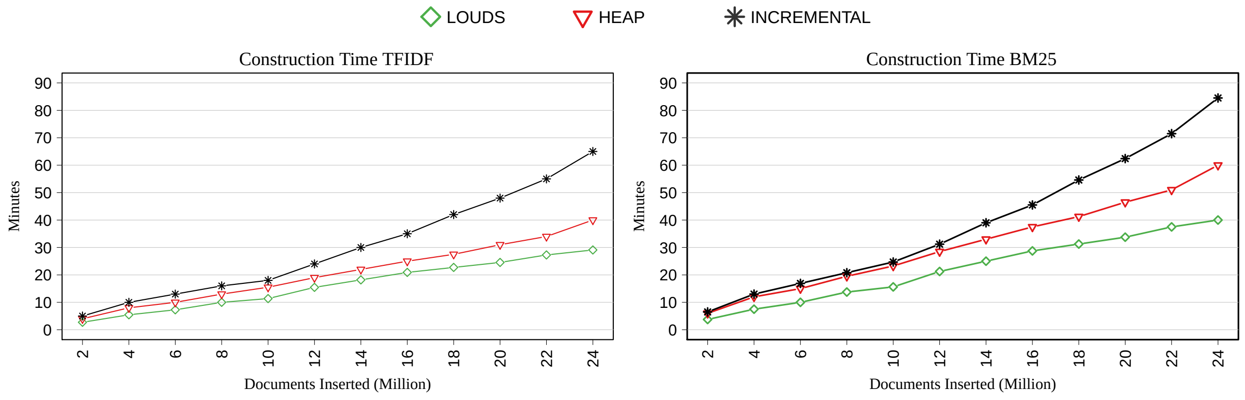


Figure 5.14: Incremental versus static construction of the inverted treaps, for tf-idf (left) and BM25 (right) scoring schemes.

not located in contiguous memory, leading to cache misses. Cache misses are also caused because each static tree has its own `dac_vector` and topology bitmap, isolated from those of other static treaps. Second, the incremental treap requires an additional RANK_1 operation each time we move from a free node to a static tree, or from a static tree to another static tree.

Overall, compared with LOUDS, dynamism costs us about 50% overhead in both space and query time performance, and building from scratch by successive insertions requires twice the time of a static construction. Compared with HEAPS, dynamism poses a 50% overhead in both space and construction time, and it requires twice the time at queries. Of course, reconstruction from scratch is not an alternative when insertions are mixed with queries.

5.3 Discussion

We have introduced a new inverted index representation based on the treap data structure. Treaps turn out to be an elegant and flexible tool to represent simultaneously the docid and the weight ordering of a posting list. We use them to design efficient ranked union and intersection algorithms that simultaneously filter out documents by docid and frequency. The treap also allows us to represent both docids and frequencies in differential form, to improve the compression of the posting lists. Our experiments under the tf-idf scoring scheme show that inverted treaps use about the same space as competing alternatives like Block-Max and Dual-Sorted, but they are significantly faster: from 20 times faster on one-word queries to 3–10 times faster on ranked unions and 1–2 times faster on ranked intersections. On a quantized BM25 score, inverted treaps use about 40% more space than the best alternatives, but they are still 20 times faster on one-word queries, slightly faster on unions, and up to 2 times faster on intersections. Inverted treaps are generally the fastest alternative for $k \leq 100$, and on one- and two-word queries, which are the most popular ones. In addition, we have shown that treaps can handle insertions of new documents, with a 50%–100% degradation in space, construction and query time performance.

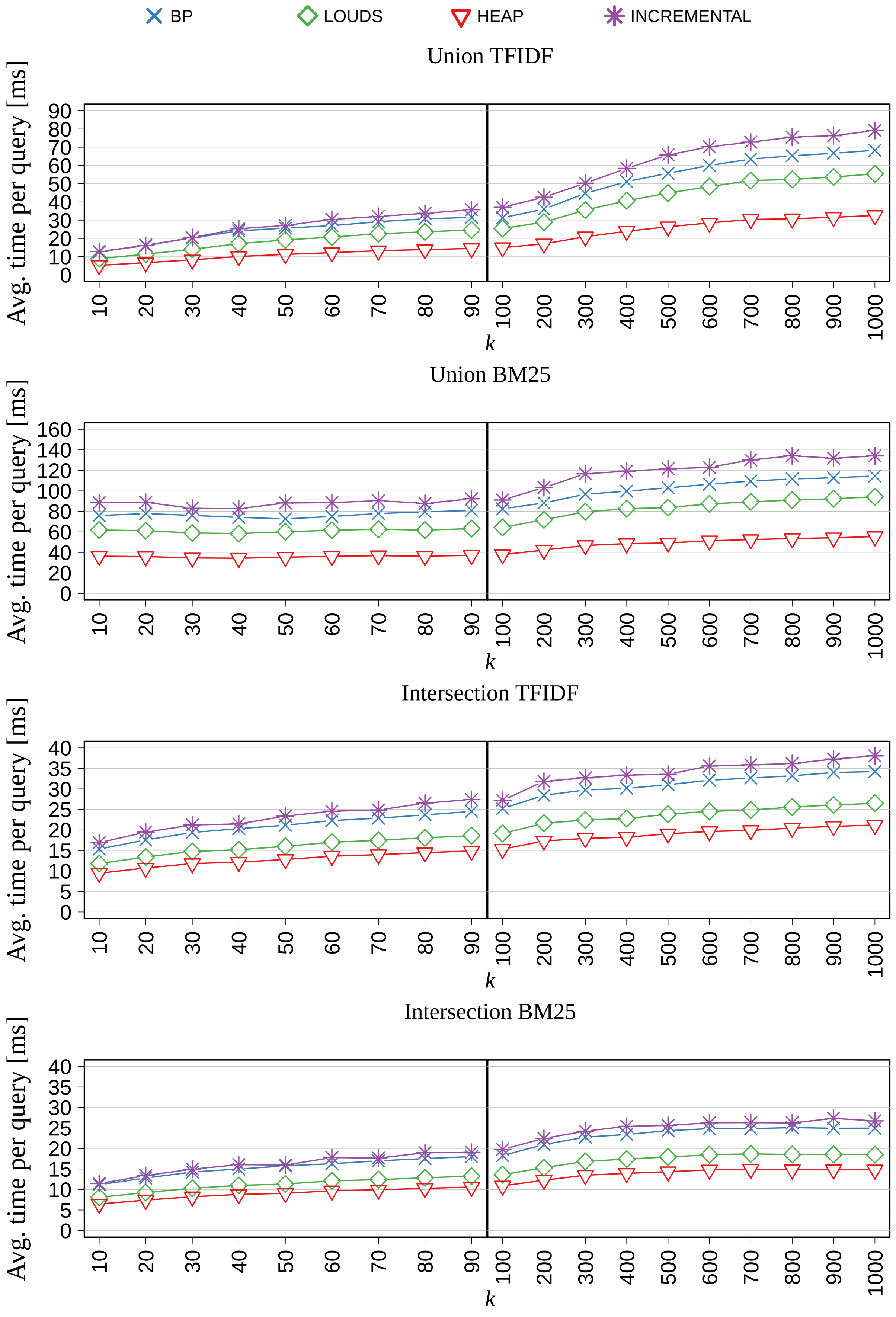


Figure 5.15: Ranked union and intersection times as a function of k, using tf-idf and BM25, for our static and incremental variants.

A future research line is to study the effect of reassigning docids. Some results [59] show that reassignment can significantly improve both space and processing time. How much would treaps improve with such schemes? Can we optimize the reassignment for a treap layout?

An important part of our gain owed to separating lists with frequency $f_0 = 1$ (this is the main explanation why our scheme performs better on tf-idf than on quantized BM25). How to efficiently separate lists with higher frequencies or impacts is a challenge, and it can lead to important gains. It is also interesting to explore how this idea could impact on schemes like Block-Max and Dual-Sorted.

Finally, we have used DAAT processing on our inverted treaps. Such an approach penalizes long queries, as already noted before in Block-Max [59]. We believe the time would become almost nonincreasing with the query length if we used treaps under a TAAT scheme, where the longer lists were processed after determining good lower bounds with the shorter lists. This constitutes another interesting line of future work.

Part III

Practical Top-k on General Sequences

Chapter 6

Top-k Document Retrieval on General Sequences

In this chapter we focus on indexing a collection of generic *strings* to support top-k queries on it. The top-k document retrieval problem can then be defined as follows: a collection $\mathcal{D}$ of D documents, consisting of sequences of total length n over an alphabet of size σ, is preprocessed so that given a query string P of length m, the system retrieves k documents with the highest "score" to P, for some definition of score.

The basic solutions to this general problem build a *suffix tree* [164] or a *suffix array* [107], which are indexes that can count and list all the individual occurrences of P in the collection, in optimal or near-optimal time. Still this functionality is not sufficient to solve top-k document retrieval efficiently. The problem of finding top-k documents containing the pattern as a substring, even with a simple relevance measure like term frequency, is challenging. Hon et al. [87] presented the first efficient solution, achieving $O(m + \log n \log\log n + k)$ time, yet with super-linear space usage, $O(n \log^2 n)$ bits. Then Hon et al. [88] improved the solution to $O(m+k \log k)$ time and linear space, $O(n \log n)$ bits. The problem was then essentially closed by Navarro and Nekrich [119], who achieved optimal $O(m+k)$ time using $O(n(\log\sigma+\log D))$ bits.

In practice, however, those solutions are far from satisfactory. Implemented directly from the theory, the constants involved in the optimal (and previous) solutions are not small, especially in space: The index can be as large as 80 times the size of the collection, making it unfeasible in practice. Just a part of the solution, the suffix tree, can be 20 times larger than the text, and the suffix array, 4 times. There has also been a line of research aiming at optimal space [116]. While they achieved important theoretical results, a verbatim implementation is likely to be equally unsatisfactory.

The most practical implementations [53, 124, 77] have followed an intermediate path in terms of space, using 2–5 times the text size and answering queries within milliseconds. Still, they are toy implementations, limited to relatively small collections of a few hundred megabytes. They are also restricted to sequences with over small alphabets (usually $\sigma = 255$ distinct symbols), which prevents using them on sequences of words, for example.

In this work, we show that a carefully engineered implementation of the optimal-time proposal [119] is competitive in space with current implementations and performs orders of magnitude faster. Our new top-k index implementation uses 2.5–3.0 times the text size and answers queries within microseconds, in collections of hundreds of gigabytes and over alphabets of up to millions of symbols. Our experimental comparison shows that our index is orders of magnitude faster than previous heuristics [53], naive solutions [77], and compressed solutions [124], whereas only the latter uses less space than ours. We also present results on collections of 500 GB, where the previous structures cannot be built.

Our ability to handle large alphabets allows us apply our index on collections of natural language text, which are regarded as sequences of words (not characters), so that our index offers a functionality similar to an inverted index. In this case our index takes up about the same space of the tokenized collection (i.e., one integer per word, and still including the storage of the collection itself).

An initial version of this work was published in the Data Compression Conference (*DCC*), 2013, held in Salt Lake City, USA. An extended version of this work was submitted for revision to the ACM *Journal on Experimental Algorithms (JEA)* on January 2016. The extended version was developed in collaboration with Simon Gog from Karlsruhe Institute of Technology (KIT), Germany. Simon Gog provided a faster and smaller re-implementation of the work published in *DCC* and included the implementation from *ALENEX*'15 [78]. He also implemented part of the new indexes presented here.

The first version of this work, published in the *DCC* conference had an error in the measurements of the size of the index [1].

6.1 Linear Space Solutions

This section describes in detail the top-k framework of Hon et al. [88] and the subsequent optimal-time solution of Navarro and Nekrich [119], which is the one we implement in this work.

6.1.1 Hon et al. solution

Let $\mathcal{T}$ be the suffix tree of the concatenation $\mathcal{C}$ of a collection of documents $d_1, d_1, \ldots, d_D$. This tree contains the nodes corresponding to all the suffix trees $\mathcal{T}_i$ of the documents d_i: for each node $u \in \mathcal{T}_i$, there is a node $v \in \mathcal{T}$ such that $path(v) = path(u)$. We will say that $v = map(u, i)$. Also, let $parent(u)$ be the parent of u and $depth(u)$ be its depth. The idea of Hon et al. [88] is to store $\mathcal{T}$ with additional information about the trees $\mathcal{T}_i$. For each $v = map(u, i)$, they store a pointer $ptr(v, i) = v' = map(parent(u), i)$, noting where the parent of u maps in $\mathcal{T}$. These pointers are stored at the target nodes v' in a so-called F-list. Together with the pointers $ptr(v, i)$ they also store a weight $w(v, i)$, which is the relevance of

[1] `http://dcc.uchile.cl/~gnavarro/fixes/dcc13.1.html`

$path(u)$ in d_i. This relevance can be any function that depends on the set of starting positions of $path(u)$ in d_i. In this Part of the thesis, we will focus on a simple one: the number of leaves of u in $\mathcal{T}_i$, that is, $\mathrm{tf}(P, d)$. Let v be the locus in $\mathcal{T}$ of a pattern P. They proved that, for each distinct document d_i where P appears, there is exactly one pointer $ptr(v'', i) = v'$ going from a descendant v'' of v (v itself included) to a (strict) ancestor v' of v, and $w(v'', i)$ is the relevance of P in d_i. Obtaining the top-k documents using this structure boils down to identifying all the pointers in the F-lists of the ancestors of v that come from the subtree of v, and then selecting k corresponding documents with the highest scores. This task can be carried out in $O(m + k \log k)$ time and the index requires linear space.

Figure 6.1 shows this structure on our example collection $\mathcal{C}$. The top left part of the figure shows the the suffix tree $\mathcal{T}_3$ corresponding to document d_3. The bottom part shows the generalized suffix tree $\mathcal{T}$ of the collection $\mathcal{C}$. The numbers inside the nodes correspond to the preorder number and the dark nodes are those mapped from $\mathcal{T}_3$ to $\mathcal{T}$. The dotted lines correspond to the pointers $ptr(v, i) = map(parent(u), i)$.

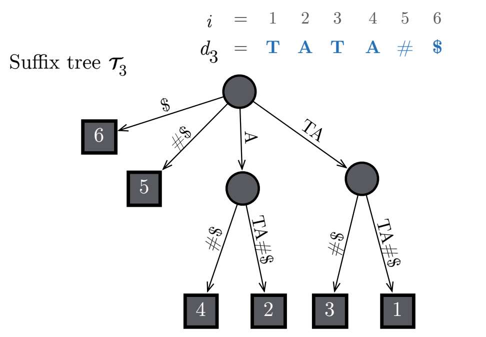
We store the documents and weights as pairs $(d_i, w(v, i))$ and append them to the F-lists, which are associated with the node where the pointer arrives. These lists are shown on the top right part of the figure.

6.1.2 Optimal solution

Navarro and Nekrich [119] presented an optimal $O(m + k)$ time solution that requires linear space, $O(n \log n)$ bits, for solving the top-k document retrieval problem. They represent the structure of Hon et al. [88] as a grid of size $O(n) \times O(n)$ with labeled weighted points. Each pointer $v' = ptr(v, i)$ is represented as a point in the grid, whose x-coordinate is the preorder number of the source v, and whose y-coordinate is the depth of the target v'. Then, the pointers going from the subtree of v to an ancestor of v correspond to the points whose x-coordinate is the preorder range of the subtree of v, and whose y-coordinates are smaller than the depth of v. By giving weights $w(v, i)$ to the points, the problem boils down to retrieving the k heaviest points in that 3-sided query range.

The construction procedure is the following: they traverse $\mathcal{T}$ in preorder, where they add for each node $v \in \mathcal{T}$ and pointer $ptr(v, i) = v'$, a new rightmost x-coordinate with only one point, with a y-coordinate value set to $depth(v')$, weight equal to $w(v, i)$ and label equal to i. To solve a query, they find the locus v of P, determine the range $[x_1, x_2]$ of all the x-coordinates filled by v or its descendants, find the k heaviest points in $[x_1, x_2] \times [0, depth(v)-1]$, and report their labels. A linear-space representation allows them to carry out this task in optimal time. Figure 6.2 shows an example of this procedure: the top part

shows the generalized suffix tree $\mathcal{T}$. The grey area corresponds to the locus of the pattern "TA" and the corresponding subtree. The bottom part of the figure shows a two-dimensional grid that maps the content of the F-lists. The grey area shows the mapping from the query pattern to a range $[x_1, x_2]$ in this grid corresponding to the subtree of the locus. The dark grey area is the final range $[x_1, x_2] \times [0, depth(v) - 1]$ from where the procedure finally selects the top-k highest-weighted points.



Node	F-Lists (d,w)
v_1	(3,1)(1,1)(2,1)(3,2)(1,2)(2,3)(3,2)(1,1)(2,1)
v_4	(3,1)(1,1)(2,1)(2,2)(3,1)(1,1)
v_7	(2,1)(2,1)
v_9	(3,1)(1,1)(2,1)(3,1)

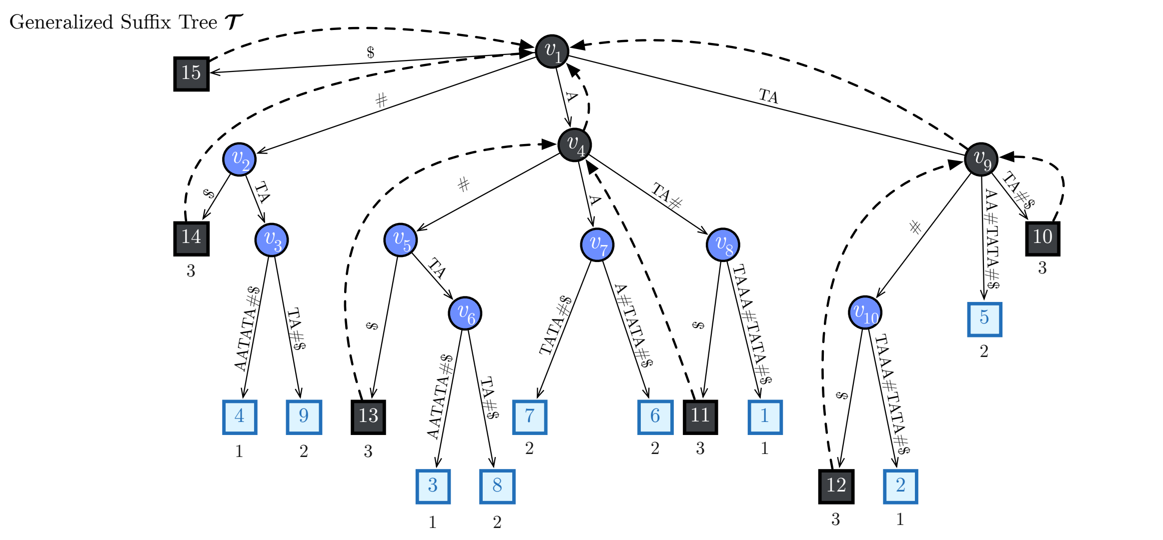


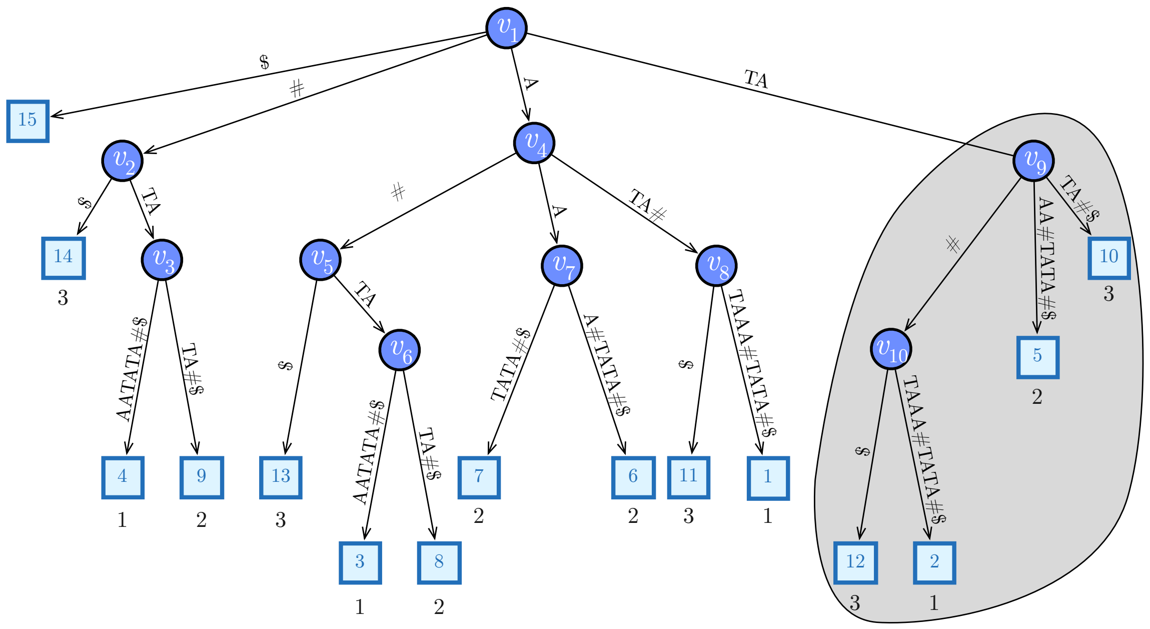
Figure 6.1: Top left: The suffix tree $\mathcal{T}_3$ of document d_3. Bottom: The generalized suffix tree of $\mathcal{T}$ with preorder naming of the inner nodes and the corresponding document numbers below the leaves. The dark nodes represent the nodes mapped from $\mathcal{T}_3$ to $\mathcal{T}$. The dotted lines corresponds to the pointers $ptr(v,i) = map(parent(u),i)$. Top right: the nonempty F-lists generated containing elements as pairs $(d_i, w(v,i))$.

6.2 Practical Implementations

The practical solutions build on the CSA of $\mathcal{C}$ and its document array DA. Once we establish that the interval of P is SA$[sp, ep]$, the query is solved by finding the k values that appear most often in DA$[sp, ep]$. We describe here the most relevant practical implementations for top-k document retrieval.

6.2.1 Hon et al.

Apart from their linear-space index, Hon et al. [88] proposed a succinct index based on sampling suffix tree nodes and storing top-k answers on those. The idea is to store the top-k answers for the lowest suffix tree nodes whose leaves contain any range SA$[i \cdot g, j \cdot g]$ for a sampling parameter value g. This can be done by marking every gth leaf in the suffix tree



Grid Representation

Node Depth																					
0	3,1	1,1	2,1	1,2	2,3	3,2									**3,2**	**1,1**	**2,1**				
1							3,1	1,1	2,1	2,2			3,1	1,1				3,1	1,1	2,1	3,1
2											2,1	2,1									
	1	2	3	4	5	6	7	8	9	10	11	12	14	15	16	17	18	19	20	21	22

Figure 6.2: Top: Suffix tree of $\mathcal{C}$ with preorder naming of the inner nodes. The subtree of v_9, the locus of pattern "TA", is highlighted in grey. Bottom: Navarro and Nekrich [119] geometric representation of the F-lists of [88]. Each cell contains a pair $(d_i, w(v,i))$. The grey area in the grid represents all the pairs that are generated from pointers that belong to the tree rooted at v_9. The darkest grey area represents the final range $[x_1, x_2] \times [0, depth(v) - 1]$.

and then performing a lowest common ancestor query for every consecutive pair of marked leaves. The tree of marked nodes is called τ_k and has $O(n/g)$ nodes. For every marked suffix tree node v, they store the k pairs $(d, tf_{v,d})$ with the highest $tf_{v,d}$. This way, one can retrieve the top-k documents by locating the interval SA$[sp, ep]$ for the pattern P and then finding the highest preprocessed suffix tree node whose interval SA$[sp', ep']$ is contained in SA$[sp, ep]$. To correct the answer one has to add the extra occurrences that appear in SA$[sp, sp'-1]$ and SA$[ep'+1, ep]$. For each such extra occurrence SA$[i]$, one has to (1) obtain the corresponding document number, (2) compute the number of occurrences of P within that document and (3) include that result in the top-k calculation. It is guaranteed that $sp' - sp < g$ and $ep - ep'g$, so the cost of correcting the answer is bounded. Hon et al. [88] build this data structure using a CSA for $\mathcal{C}$ and adds one CSA for each document d_i, plus some additional structures for handling the precomputed results.

6.2.2 GREEDY

Culpepper et al. [53] studied various heuristics to solve top-k queries on top of a CSA and a wavelet tree of the document array DA. They introduced two solutions: *Quantile probing* and *Greedy traversal*, both of them based on properties obtained by representing the document array using a wavelet tree.

- *Quantile probing* is based on the observation that in a sorted array A, if there is an item d with frequency larger than f, then there exists at least one j such that $A[j \cdot f] = d$. Culpepper et al. use range quantile queries on the wavelet tree (recall Section 2.7) to retrieve the $(j \cdot f)$th document in the sorted DA$[sp, ep]$ without actually sorting it. The algorithm makes successive passes using this observation, refining the results at each step while storing the candidates in a min-heap of size k.
- *Greedy traversal.* This technique relies on the fact if P appears many times in a document d, then the $[sp, ep]$ interval on the path to that document leaf in the wavelet tree should also be long. The wavelet tree is traversed the in a greedy fashion, that is, selecting the longest $[sp, ep]$ interval first. Then the algorithm will first reach the leaf with the highest term frequency value. The next document reported will be the second-highest score and so on. This procedure continues until k documents are reported. We show this procedure in Algorithm 19.

In practice, the greedy variant outperforms quantile probing.

Culpepper et al. [92] adapted the scheme to large natural language text collections (where each word is taken as an atomic symbol), showing that it was competitive with inverted indexes for some queries (see previous work on this line by Patil et al.[129]). The solution, however, resorts to approximations when handling ranked Boolean queries.

6.2.3 NPV

Apart from their linear-space index, Hon et al.[88] proposed a succinct index based on sampling suffix tree nodes and storing top-k answers on those. When the locus of P reaches a non-sampled node, they choose the highest sampled node below the locus and *correct* its precomputed answer by considering the additional leaves of the locus. The sampling ensures that this work is bounded.

Navarro et al. [124] implemented a practical version of the proposal of Hon et al. [88] and also combined it with the greedy algorithm (see Algorithm 19) of Culpepper et al. [53] to speed up the correction process. They also studied compressed representations of the wavelet tree of DA, by using grammar compression [102] on its bitmaps. They carried out a thorough experimental study of these approaches and previous ones, establishing one of the first baselines for future comparisons of top-k indexes.

Algorithm 19: WT_GREEDY$(v, x_{start}, x_{end}, k, result = \emptyset)$ returns the top-k symbols in $result$ with the highest amount of occurrences.

```
WT_GREEDY$(v, x_{start}, x_{end}, k, result)$
  $pq \leftarrow \emptyset$ // priority queue of struct $\langle length, v, x_{start}, x_{end}\rangle$ prioritized by length
  PUSH$(pq, \langle(x_{end} - x_{start} + 1), v, x_{start}, x_{end}\rangle)$
  **while** TOP$(pq) \neq \emptyset$ **do**
    $t \leftarrow$TOP$(pq)$
    **if** $t.v$ is a leaf **then**
      $result \cup$ LABELS$(t.v)$
    **if** $|result| = k$ **then**
      **return** $result$
    $x_{start} \leftarrow t.x_{start}$
    $x_{end} \leftarrow t.x_{end}$
    $x_{start\_left} \leftarrow$ RANK$_0(B_{t.v}, x_{start} - 1) + 1$
    $x_{end\_left} \leftarrow$ RANK$_0(B_{t.v}, x_{end})$
    $x_{start\_right} \leftarrow x_{start} - x_{start\_left}$
    $x_{end\_right} \leftarrow x_{end} - x_{end\_left}$
    **if** $x_{end\_left} - x_{start\_left} \neq 0$ **then**
      PUSH$(pq, \langle x_{end\_left} - x_{start\_left} + 1, v_l, x_{start\_left}, x_{end\_left}\rangle)$
    **if** $x_{end\_right} - x_{start\_right} \neq 0$ **then**
      PUSH$(pq, \langle x_{end\_right} - x_{start\_right} + 1, v_r, x_{start\_right}, x_{end\_right}\rangle)$
  **end while**
```

6.2.4 SORT

The first successful attempt to engineer an index handling hundred-gigabyte collections, having millions of documents and containing large alphabets, was presented by Gog et al. [77]. They build a framework for experimentation with succinct data structures, where they reimplement the greedy approach of Culpepper et al. [53] in this scenario.

Gog et al. [77] also introduce a basic solution to the top-k problem, called SORT. This is based on simply collecting all the values in DA$[sp, ep]$, sorting them by document identifier, computing term frequencies, and choosing the k largest ones. This is a good baseline to evaluate if more sophisticated ideas are worthy.

6.2.5 Patil et al.

One of the first implementations of practical top-k document retrieval was introduced by Patil et al. [129]. In their work, they store for some nodes the complete inverted lists containing the document identifiers and the frequency of the string represented by the node. These inverted lists are then sorted by pre-order rank of the strings in the GST and stored contiguously in an array. For a given pattern P they find the locus in the GST and then map the preorder rank of the locus and its rightmost leaf. This creates a range in the array of inverted lists, which is found by performing a binary search and then they are able to find the top-k documents, using RMQ queries over the frequencies of the inverted lists that lie within that range. In practice, this solution requires 5 to 19 times the size of the collection, thus making it unpractical for most scenarios. For this reason, we do not compare our work to this solution.

6.2.6 Other aproaches

Belazzougui et al. [21] presented a different approach based on monotone minimal perfect hash functions (mmphf). Instead of using individual CSAs or the document array DA to compute the frequencies, they use a weaker data structure that requires less space. The idea is to enrich the τ_k trees of Hon et al.[87], so that they have enough information to answer the top-k queries with the help of mmphfs built on the occurrences of each value d in DA.

Ferrada et al. [64] introduced an index that solves *approximate* top-k document retrieval based on the LZ78 [170] compression technique. In practice, the approximation is orders of magnitude faster and requires less space than previous approaches. They show that it is possible to obtain good-quality results for moderately large text collections.

Navarro et al. [123] carried out a thorough experimental study of the performance of most of the existing solutions to top-k document retrieval and evaluate them in the scenario where the collections are highly repetitive. They also designed new alternatives specifically tailored for repetitive collections, which are very compactable. The technique is inspired in the GST sampling technique [88] and factors out repetitions in the top-k lists. They show that use grammar compression techniques such as Re-Pair [102] provide the best trade off in terms of space and time for this scenario.

6.2.7 Summary

Current implementations need about 2–5 times the size of the collection and answer queries in tens of milliseconds. The simple implementation of the solution of Hon et al. requires 2–5 times the size of the collection and answers queries in about 10–100 milliseconds. The greedy approach introduced by Culpepper [53] improves the space to 2–4 times the size of the collection and reduces the time to 1–20 milliseconds. The best results obtained by Navarro et al. [124] use 2–3 times the size of the collection and answer queries in about 1–10 milliseconds. These implementations are limited to relatively small collections of a few hundred megabytes. Moreover, these are limited to sequences containing a small alphabet size ($\sigma \leq 255$). Table 6.1 shows a summary of these results, compared to our achievements in this work.

Index	Max. σ	Max. $\lvert\mathcal{C}\rvert$ in MB	Size (times $\lvert\mathcal{C}\rvert$)	Time in μs
Hon et al. implementation [124]	$2^8 - 1$	137	$2 - 5$	$10^4 - 10^5$
GREEDY [53]	$2^8 - 1$	100	$2 - 4$	$10^3 - 10^4$
NPV [124]	$2^8 - 1$	137	$2 - 3$	$10^3 - 10^4$
Patil et al. [129]	$2^8 - 1$	100	$5 - 19$	$10^2 - 10^3$
SORT [77]	$2^{64} - 1$	72,000	$2 - 3$	$10^4 - 10^6$
Ours	$2^{64} - 1$	72,000	$2 - 3$	$10^1 - 10^3$

Table 6.1: Comparison of practical results.

Chapter 7

Top-k on Grids

Top-k queries on multidimensional weighted point sets ask for the k heaviest points in a range. These type of queries arises most prominently in data mining and OLAP processing (e.g., find the sellers with most sales in a time period) and in GIS applications (e.g., find the cheapest hotels in a city area), but also in less obvious document retrieval applications [119]. In the example of sales, one coordinate is the seller id, which is arranged hierarchically to allow queries for sellers, stores, areas, cities, states, etc., and the other is time (in periods of hours, days, weeks, etc.). Weights are the number of sales made by a seller during a time slice. Thus the query asks for the k heaviest points in some range $Q = [x_1, x_2] \times [y_1, y_2]$ of the grid.

The optimal solution proposed by Navarro and Nekrich (recall Section 6.1) considers the problem of top-k document retrieval on general sequences as a geometrical problem: locate the top-k heaviest points from a two-dimensional query on a grid. In order to develop practical implementations of this theoretical work, we need to address the geometrical problem using data structures that take into consideration both space and time constraints.

In this chapter we present two data structures to solve this geometrical problem, and they are a fundamental component that is used for our practical solutions that are detailed in this part. The first data structure is based on the work of Navarro et al. [120] and makes use of wavelet trees and RMQ (Section 2.7 and Section 2.12). The second data structure, K^2-treap, is a new data structure based on the K^2-tree (see Section 2.9) and the Treap (see Section 2.13) that was developed in collaboration with Nieves R. Brisaboa and Guillermo de Bernardo [32].

7.1 Wavelet Trees and RMQ

Recall from Section 2.7 that the wavelet trees can be used to represent a two-dimensional grid $[1, n] \times [1, m]$ and that it can be regarded as sequence $\mathcal{S}[1, n]$ since it has only one point per column. Now, let us enhance this grid by adding another dimension to each point: a weight. Thus, the sequence now contains pairs as $\mathcal{S}[i] = \langle s_i, w_i \rangle$. The operation that is of interest to the development of this thesis is to retrieve the top-k maximum points according to their weights, that are located within a rectangle $Q = [x_1, x_2] \times [y_1, y_2]$.

Navarro et al. [120] introduced a solution for two-dimensional range top-k queries based on wavelet trees. The idea is to enhance the bitmaps of each node of the wavelet tree as follows: Let $p_1, \ldots, p_r$ be the points represented at a node v, and $w(p)$ be the weight of point p. Then a range maximum query (RMQ) data structure built on $w(p_1), \ldots, w(p_r)$ is stored together with the bitmap. Recall from Section 2.12 that such structure requires $2r + o(r)$ bits and finds the maximum weight in any range $[w(p_i), \ldots, w(p_j)]$ in constant time and without accessing the weights themselves. Therefore, the total space becomes $3n \log m + o(n \log m)$ bits.

To solve top-k queries on a grid range $Q = [x_1, x_2] \times [y_1, y_2]$, we first traverse the wavelet tree to identify the $O(\log m)$ bitmap intervals where the points in Q lie. The heaviest point in Q in each bitmap interval is obtained with an RMQ, but we need to obtain the actual priorities in order to find the heaviest among the $O(\log m)$ candidates. The priorities are stored sorted by the x- or y-coordinate, so we obtain each one in $O(\log m)$ time by tracking the point with maximum weight at each interval. Thus a top-1 query is solved in $O(\log^2 m)$ time. For a top-k query, we must maintain a priority queue of the candidate intervals and, each time the next heaviest element is found, we remove it from its interval and reinsert it in the queue with the two resulting subintervals. The total query time is $O((k+\log m)\log(km))$.

It is possible to reduce the time to $O((k + \log m)\log^{\epsilon} m)$ for any constant $\epsilon > 0$ [119], but the space usage is much higher, even if linear. A better result, not using wavelet trees, was introduced by Prosenjit et al. [112], who presented a succinct data structure matching the optimal $O(\log n / \log\log n)$ time using just $n + o(n)$ integers. This is for n points on an $n \times n$ grid.

We demonstrate this procedure with an example in Figure 7.1. At the top of the tree, we show the weights of each point in a sequence $W[1, n]$. At each level of the wavelet tree we show how the weight sequence is permuted according to the values of the grid points; they are displayed with grey borders and grey fonts. Let us say that we want to retrieve the top-1 value from the range $[10, 14] \times [3, 7]$. We start at the root of the wavelet tree and project the values to its left and right child. For all nodes, we do not store the permutation of the weights, instead, we store a RMQ data structure of the permutations. Since the left node of the root does not belong completely to the query range, we continue with the traversal until we find a node that is fully contained. We perform an RMQ query over projected ranges of the suitable nodes. For example, we perform a RMQ query over the range $[2, 3]$ on the node containing the values 343 on the left side of the tree, and obtain the first position of the range that contains the maximum value, as shown on the Figure. On the other hand, we perform a RMQ query over the range $[6, 7]$ for the right child of the root, since it is fully contained in the query range and obtain the second position, which corresponds to the maximum weight value of the range. We show in black background, the RMQ query results over the ranges. Note that, since we do not store explicitly the weights at these nodes, we need to traverse down the tree until the last level where we actually store the permutation of $W[1, n]$ according to the leaves in an explicit way. The final step is to obtain the corresponding weight and compare it to the other branches that might have reached a leaf. In this case, the final result is the point located at $(11, 7)$ with weight 9.

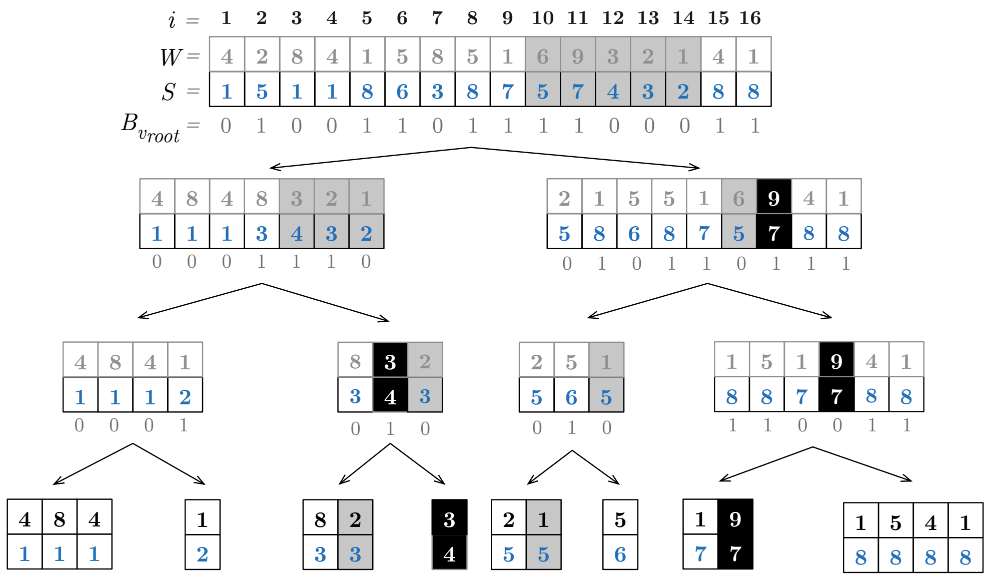
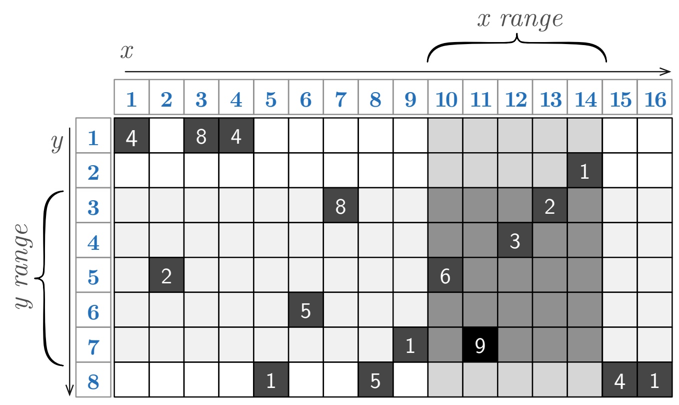



Figure 7.1: Wavelet tree and RMQ combination for top-k two-dimensional queries. The weights associated to each point are shown in sequence W. At each level of the wavelet tree, we show the permutation of the weights with a grey border and grey font. Note that these values are not stored, instead we store a RMQ data structure for those sequences. We show in black background the maximum elements inserted in a priority queue for a top-1 traversal. The final result is the point located at $(11, 7)$ with weight 9.

7.2 K^2-treap

Recall from Section 2.9, that K^2-trees can be used to perform range queries on grids. In this section we present a new data structure inspired by the concept of treaps (see Section 2.13). We propose a data structure that merges the concept of the K^2-tree and the treap and we name this data structure K^2-treap. We further improve this technique to solve general aggregated queries on two dimensional grids. We present these improvements in Chapter

10. In this section we describe the particular case, that is, a data structure that obtains the maximum points within a two-dimensional range. This work was done in collaboration with Nieves Brisaboa and Guillermo de Bernardo from Univerisidad de A Coruña [32].

Consider a matrix $M[n \times n]$ where each cell can either be empty or contain a weight in the range $[0, d-1]$. We consider a quadtree-like recursive partition of M into K^2 submatrices, the same performed in the K^2-tree with binary matrices. We build a conceptual K^2-ary tree similar to the K^2-tree, as follows: the root of the tree will store the coordinates of the cell with the maximum weight of the matrix, and the corresponding weight. Then the cell just added to the tree is marked as *empty*,deleting it from the matrix. If many cells share the maximum weight, we pick anyone of them. Then, the matrix is conceptually decomposed into K^2 equal-sized submatrices, and we add K^2 children nodes to the root of the tree, each representing one of the submatrices. We repeat the assignment process recursively for each child, assigning to each of them the coordinates and value of the heaviest cell in the corresponding submatrix and removing the chosen points. The procedure continues recursively for each branch until we find a completely empty submatrix (either because the matrix did not contain any weights in the region or because the cells with weights have been "emptied" during the construction process) or we reach the cells of the original matrix.

Figure. 7.2 shows an example of K^2-treap construction, for $K = 2$. On the top of the image we show the state of the matrix at each level of decomposition. $M0$ represents the original matrix, where the maximum value is highlighted. The coordinates and value of this cell are stored in the root of the tree. In the next level of decomposition (matrix $M1$) we find the maximum values in each quadrant (notice that the cell assigned to the root has already been removed from the matrix) and assign them to the children of the root node. The process continues recursively, subdividing each matrix into K^2 submatrices. The cells chosen as local maxima are highlighted in the matrices corresponding to each level of decomposition, except in the last level where all the cells are local maxima. Empty submatrices are marked in the tree with the symbol "-".

7.2.1 Local Maximum Coordinates

The data structure is represented in three parts: The location of local maxima, the weights of the local maxima, and the tree topology.

The conceptual K^2-treap is traversed level-wise, reading the sequence of cell coordinates from left to right in each level. The sequence of coordinates at each level ℓ is stored in a different sequence $coord[\ell]$. The coordinates at each level ℓ of the tree are transformed into an offset in the corresponding submatrix, transforming each c_i into $c_i \bmod (n/K^\ell)$ using $\lceil \log(n) - \ell \log K \rceil$ bits. For example, in Figure. 7.3 (top) the coordinates of node $N1$ have been transformed from the global value $(4, 4)$ to a local offset $(0, 0)$. In the bottom of Fig. 7.3 we highlight the coordinates of nodes $N0$, $N1$ and $N2$ in the corresponding *coord* arrays. In the last level all nodes represent single cells, so there is no *coord* array in this level. With this representation, the worst-case space for storing t points is $\sum_{\ell=0}^{\log_{K^2}(t)} 2K^{2\ell} \log \frac{n}{K^\ell} = t \log \frac{n^2}{t}(1 + O(1/K^2))$, that is, the same as if we stored the points using the K^2-tree.

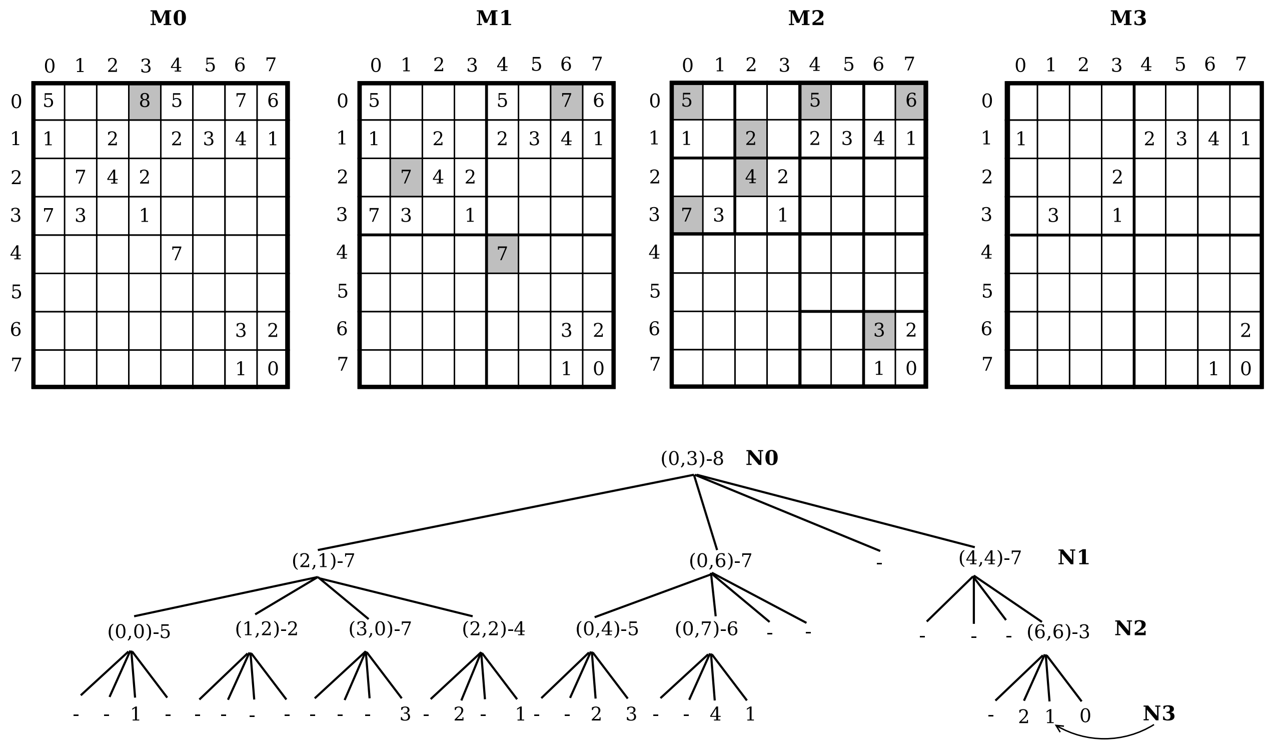


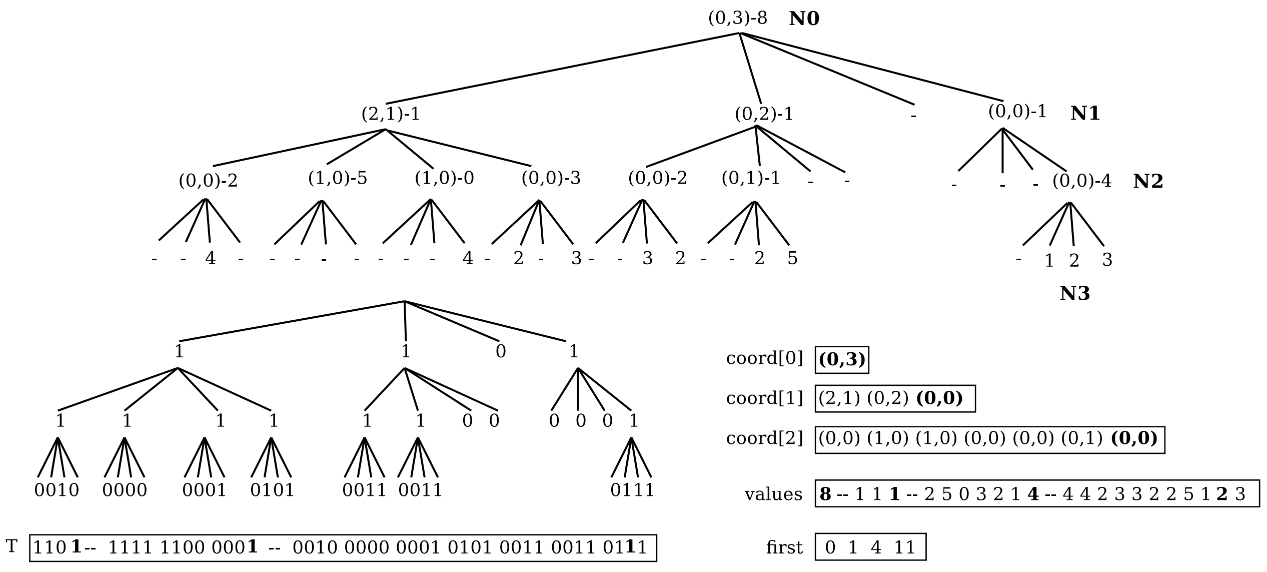
Figure 7.2: Example of K^2-treap construction from the matrix in Fig. 10.1

7.2.2 Local Maximum Values

The maximum value in each node is encoded differentially with respect to the maximum of its parent node [48]. The result of the differential encoding is a new sequence of non-negative values, smaller than the original. Now the K^2-treap is traversed level-wise and the complete sequence of values is stored in a single sequence named *values*. To exploit the small values while allowing efficient direct access to the array, we represent *values* with Direct Access Codes [34]. Following the example in Fig. 7.3, the value of node $N1$ has been transformed from 7 to $8 - 7 = 1$. In the bottom of the figure the complete sequence *values* is depicted. We also store a small array $first[0, \log_{K^2} n]$ that stores the offset in *values* where each level starts.

7.2.3 Tree Structure

We separate the structure of the tree from the values stored in the nodes. The tree structure of the K^2-treap is stored in a K^2-tree. Fig. 7.3 shows the K^2-tree representation of the example tree, where only cells with value are labeled with a 1. We will consider a K^2-tree stored in a single bitmap T with *rank* support, that contains the sequence of bits from all the levels of the tree. Our representation differs from a classic K^2-tree (that uses two bitmaps T and L and only adds rank support to T) because we will need to perform rank operations also in the last level of the tree. The other difference is that points stored separately are removed from the grid, thus in a worst-case space analysis it turns out that the space used to represent those explicit coordinates is subtracted from the space the K^2-tree would use,



(0,3)-8 N0
(2,1)-1
(0,2)-1
(0,0)-1 N1
(0,0)-2
(1,0)-5
(1,0)-0
(0,0)-3
(0,0)-2
(0,1)-1
(0,0)-4 N2
N3
coord[0] (0,3)
coord[1] (2,1) (0,2) (0,0)
coord[2] (0,0) (1,0) (1,0) (0,0) (0,0) (0,1) (0,0)
values 8 -- 1 1 1 -- 2 5 0 3 2 1 4 -- 4 4 2 3 3 2 2 5 1 2 3
first 0 1 4 11
T 1101-- 1111 1100 0001 -- 0010 0000 0001 0101 0011 0011 0111

Figure 7.3: Storage of the conceptual tree in our data structures

therefore storing those explicit coordinates is free in the worst case.

7.2.4 Top-k Query Processing

The process to answer top-k queries starts at the root of the tree. Given a range $Q = [x_1, x_2] \times [y_1, y_2]$, the process initializes an empty max-priority queue and inserts the root of the K^2-tree. The priority queue stores, in general, K^2-tree nodes sorted by their associated maximum weight. Now, we iteratively extract the first priority queue element (the first time this is the root). If the coordinates of its maximum element fall inside Q, we output it as the next answer. In either case, we insert all the children of the extracted node whose submatrix intersects with Q, and iterate. The process finishes when k results have been found or when the priority queue becomes empty (in which case there are less than k elements in Q).

7.2.5 Other Supported Queries

The K^2-treap can also answer basic range queries (i.e., report all the points that fall in Q). This is similar to the procedure on a K^2-tree, where the submatrices that intersect Q are explored in a depth-first manner. The only difference is that we must also check whether the explicit points associated to the nodes fall within Q, and in that case report those as well. Finally, we can also answer *interval queries*, which ask for all the points in Q whose weight is in a range $[w_1, w_2]$. To do this, we traverse the tree as in a top-k range query, but we only output weights whose value is in $[w_1, w_2]$. Moreover, we discard submatrices whose maximum weight is below w_2.

Chapter 8

Practical Implementations

In this chapter, we present our practical implementations of the solution proposed by Navarro and Nekrich (recall Section 6.1) for solving, in optimal time, top-k queries on general sequences. As discussed before, the main constraints of the implementation of this solution is space, so we make use of the wavelet tree combined with RMQ solution (see Section 7.1) and the K^2-treap (see Section 7.2). We show how we engineered different alternatives, in order to obtain indexes that are practical in terms of size and faster than other solutions from the state of the art.

8.1 A Basic Compact Implementation

In this section we describe a basic compact implementation of the optimal-time solution [119], which will be labeled WRMQ.

8.1.1 Searching for Patterns

In order to search for patterns in the collection, we use a CSA, which computes the interval $[sp, ep]$ corresponding to P in time $O(t_{search(m)})$. We also add a compact representation of the topology of the suffix tree, seeing it as an ordinal tree. Note that this is just the topology, not a full suffix tree, so we need to search using the CSA. We will use the suffix tree topology to obtain the locus v of P from sp and ep, and then to map it to the grid.

8.1.2 Mapping to the Grid

The grid G is of width $\sum_i |\mathcal{T}_i| \leq 2n$, as we add one coordinate per node in the suffix tree of each document. To save space, we will consider a *virtual* grid just as defined, but will store a narrower *physical* grid. In the physical grid, the entries corresponding to leaves of $\mathcal{T}$ (which contain exactly one pointer $ptr(v, i)$) will not be represented. Thus the physical grid is of width at most n. This *frequency thresholding* is a key idea, as it halves the space of most structures in our index.

A bitmap $B[1, 2n]$ will be used to map between the suffix tree and the grid. Bitmap B will mark starting positions of nodes of $\mathcal{T}$ in the physical grid: along the grid construction process, each time we arrive at an internal node v we add a 1 to B, and each time we add a new x-coordinate to the grid (due to a pointer $ptr(v, i)$) we add a 0 to B. This structure will be called MAP.

8.1.3 Finding Isolated Documents

Due to frequency thresholding, the information on the grid may be insufficient to answer top-k queries when the result includes documents where P appears only once. For this purpose, we store the RMQ structure on top of the array CA of Muthukrishnan [115] (recall Section 2.15) (the array itself is not stored). This structure, RMQC, allows us list k distinct documents in any interval SA$[sp, ep]$.

To determine the identity of a document found at position i of CA, we will use the CSA to compute $p = \mathrm{SA}[i]$ and then determine to which document position $\mathcal{C}[p]$ belongs. For this sake we store a sparse bitmap DOCBITMAP, that marks beginnings of documents in $\mathcal{C}$. The document is then $rank_1(\mathrm{DOCBITMAP}, p)$.

8.1.4 Representing the Grid

In the grid there is exactly one point per x-coordinate. We represent with a wavelet tree [81] the sequence of corresponding y-coordinates. Each node v of the wavelet tree represents a subsequence of the original sequence of y-coordinates. We consider the (virtual) sequence of the weights associated to the points represented by v, $W(v)$, and build an RMQ data structure for each node $W(v)$, as explained, to support top-k queries on the grid.

8.1.5 Representing Labels and Weights

The labels of the points, that is, the document identifiers, are represented directly as a sequence of at most $n \log D$ bits, aligned to the bottom of the wavelet tree. Given any point to report, we descend to the leaf and retrieve the aligned document identifier.

The weights are stored similarly, but using direct access codes, to take advantage of the fact that most weights (term frequencies) are small.

8.1.6 Summing Up

The WRMQ index consists of eight components: the compressed suffix array (CSA), the suffix tree topology (TREE), the suffix tree to grid mapping (MAP), the wavelet tree over the grid G (WT) including the RMQ structures at each node, the document ids associated with the grid elements (DOC), in the same order of leaves of the wavelet tree, the weights associated to the documents (FREQ), the RMQ structure to retrieve documents occurring once over the CA array (RMQC), and the bitmap marking document borders (DOCBITMAP).

If the height of the suffix tree is $O(\log n)$, as is the case in many reasonable distributions [155], then WT requires $3n \log\log n + O(n)$ bits. The other main components are $|\mathrm{CSA}| \leq n \log \sigma$ bits, the array DOC, which uses $n \log D$ bits, and the array FREQ, which in the worst case needs $n \log n$ bits but uses much less in practice. The other elements add up to $O(n)$ bits, with a constant factor of about 8–10.

8.1.7 Answering Queries

The first step to answer a query is to use the CSA to determine the range $[sp, ep]$. To find the locus v of P in the topology of the suffix tree, we compute l and r, the *sp*th and *ep*th leaves of the tree, respectively, using $l = \textsc{leaf\_select}(sp)$ and $r = \textsc{leaf\_select}(ep)$; then we have $v = \textsc{lca}(l, r)$.

To determine the horizontal extent $[x_1, x_2]$ of the grid that corresponds to the locus node v, we first compute $p_1 = \textsc{preorder}(v)$ and $p_2 = p_1 + \textsc{subtree\_size}(v)$. This gives the preorder range $[p_1, p_2)$ including leaves. Now $l_1 = \textsc{leaf\_rank}(p_1)$ and $l_2 = \textsc{leaf\_rank}(p_2 - 1)$ give the number of leaves up to those preorders. Then, since we have omitted the leaves in the physical grid, we have $x_1 = \textsc{select}_1(B, p_1 - l_1) - (p_1 - l_1) + 1$ and $x_2 = \textsc{select}_1(B, p_2 - l_2) - (p_2 - l_2)$. The limits in the y axis are just $[0, \textsc{depth}(v) - 1]$. Thus the grid area to query is determined with a constant amount of operations on bitmaps and trees.

Once the range $[x_1, x_2] \times [y_1, y_2]$ to query is determined, we proceed to the grid. We determine the wavelet tree nodes that cover the interval $[y_1, y_2]$, and map the interval $[x_1, x_2]$ to all of them.

We now use the top-k algorithm for wavelet trees we described. Let $v_1, v_2, \ldots, v_s$ be the wavelet tree nodes that cover $[y_1, y_2]$ and let $[x_1^i, x_2^i]$ be the interval $[x_1, x_2]$ mapped to v_i. For each of them we compute $\textsc{rmq}_{W(v_i)}(x_1^i, x_2^i)$ to find the position x_i with the largest weight among the points in v_i, and find out that weight and the corresponding document, w_i and d_i. We set up a max-priority queue that will hold at most k elements (elements smaller than the kth are discarded by the queue). We initially insert the tuples $(v_i, x_1^i, x_2^i, x_i, w_i, d_i)$, being w_i the sort key. Now we iteratively extract the tuple with the largest weight, say $(v_j, x_1^j, x_2^j, x_j, w_j, d_j)$. We report the document d_j with weight w_j, and create two new ranges in v_j: $[x_1^j, x_j - 1]$ and $[x_j + 1, x_2^j]$. We compute their RMQ, find the corresponding documents and weights, and reinsert them in the queue. After k steps, we have reported the top-k documents. We will refer to this procedure as TOPK_GRID_QUERY.

Algorithm 20: Top-k algorithm using WRMQ index

Input: Query pattern P, amount k of documents to retrieve
Output: top-k documents

```
TOPK($P, k$)
   $result \leftarrow \emptyset$
   $[sp, ep] \leftarrow$LOCATE($P$)
   $l \leftarrow$ LEAF_SELECT($sp$)
   $r \leftarrow$ LEAF_SELECT($ep$)
   $v \leftarrow$ LCA($l, r$)
   $p_1 \leftarrow$ PREORDER_RANK($v$)
   $p_2 \leftarrow p+$SUBTREE_SIZE($v$)
   $l_1 \leftarrow$LEAF_RANK($p_1$)
   $l_2 \leftarrow$LEAF_RANK($p_2 - 1$)
   $x_1 \leftarrow$ SELECT$_1$($B, p_1 - l_1$) $+ 1$
   $x_2 \leftarrow$ SELECT$_1$($B, p_2 - l_2$) $- (p_2 - l_2)$
   $depth \leftarrow$ DEPTH($v$) $- 1$
   $result \leftarrow$ TOPK_GRID_QUERY($x_1, x_2, 0, depth, k$)
   if $|result| < k$ then
      $result \leftarrow$ DOCUMENT_LISTING($sp, ep, result, k, sp$)
   return $result$
```

If we implement the bitmap, tree, and RMQ operations in constant time, and assuming again that the suffix tree is of height $O(\log n)$, the total time of this process is $O(t_{search(m)} + (k+\log\log n)(\log\log n+\log k))$. This is because there are $O(\log\log n)$ wavelet tree nodes covering $[y_1, y_2]$, and for each of them we descend to the leaf to find the weight and the document identifier (again in $O(\log\log n)$ time) and insert them in the priority queue ($O(\log k)$ time). Then we repeat k times the process of extracting a result from the queue, and generating two new weights to insert in it.

We remind that we have not stored the leaves in the grid. Therefore, if the procedure above yields less than k results, we must complete it with documents where the pattern appears only once. Here we use Algorithm 6 to complete our current result d to make it of size k by adding new documents of DA$[sp, ep]$. Here we use RMQC and DOCBITMAP. We might have to extract up to k distinct documents with this technique to complete the answer (since we may obtain again those we already have from the grid). Each step requires $O(t_{locate})$ time to obtain the document identifier, so this may add up to time $O(k\, t_{locate})$ to the complexity.

Algorithm 20 shows the complete procedure.

8.2 An Improved Index

In this section we present the two main ideas that lead to a conceptually simpler and, as we will see later, faster and smaller representation than the basic one (WRMQ).

8.2.1 Mapping the Suffix Tree to the Grid

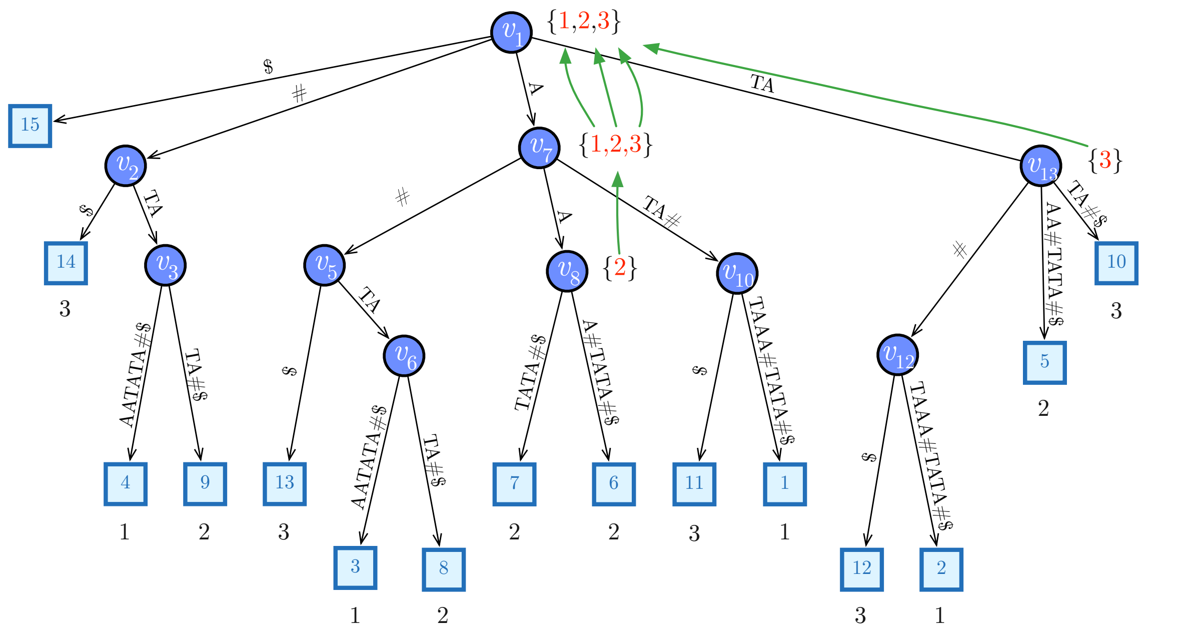
We reorganize the grid so that a more space-efficient mapping from pattern P to a 3-sided range query in the grid is possible. The basic solution uses $2t$ bits to represent the topology of the suffix tree (of t nodes, where $n \leq t \leq 2n$), plus up to $2n$ bits for B, for a total of up to $6n$ bits (plus the sublinear part). The improved representation will consist of a single bitvector H, of length $2(n-D)$, and the mapping from the suffix tree to the grid will be simpler.

We first explain how we list the parent pointers $ptr(v,i)$ leaving the suffix tree nodes v in the grid, and then show how we can efficiently map to the grid. We identify each internal node with the position in SA of the rightmost leaf of its leftmost subtree. Fig. 8.1 shows this naming for the internal nodes: The root node is named v_1 because the position of its first child (which is also a leaf) is at position 1, the rightmost leaf of the leftmost child of node v_7 is at position 7, and so on. Note that the names are between 1 and n and they are unique (although not all names must exist, e.g., there is no node v_4 in our suffix tree). The bitvector H is generated by first writing n 1s and then inserting a 0 right before the j-th 1 per pointer $ptr(v_j,i)$ leaving node v_j.

The 0s in H then correspond to the x-domain in the grid, thus we do not need to represent it explicitly. On the other hand the y-coordinate (string depth of target node), weight and document id of the pointers, are stored associated to the corresponding 0 in H (in arrays y, FREQ, and DOC, respectively, see the right of Fig. 8.1). Now assume the CSA search yields the leaf interval $[sp, ep]$ for P. These are the positions of the leftmost and rightmost leaves that descend from the locus node v of P (although we will not compute v). Then, we note that a node v_j lies in the subtree of v (including v) if and only if $j \in [sp, ep-1]$.

Lemma 8.2.1 A node v_j, if it exists, lies in the subtree of v (including v) if and only if $j \in [sp, ep-1]$.

Proof. If v_j is in the subtree of v, then its range of descendant leaves is included in that of v, $[sp, ep]$. Since there are no unary nodes, the rightmost child v_r of v_j has at least one descendant leaf, and thus the leaves descending from the leftmost child v_l of v_j are within $[sp, ep-1]$. In particular, the rightmost leaf descending from v_l, j, also belongs to $[sp, ep-1]$. Conversely, if $j \in [sp, ep-1]$ and v_j exists, then v_j is an ancestor of leaf j and so is v, thus they are the same or one descends from the other. However, if v descended from v_j, then j could only be ep (if p was the leftmost child v_l of v_j or belonged to the rightmost path descending from v_l), or it would be outside $[sp, ep]$. □



	1	2	3	4	5	6				7		8	9	10	11	12		13	14	15
$H =$	1	1	1	1	1	1	0	0	0	1	0	1	1	1	1	1	0	1	1	1
$DOC =$							1	2	3		2						3			
$x =$							1	2	3		4						5			
$y =$							0	0	0		1						0			
$w =$							2	3	2		2						2			

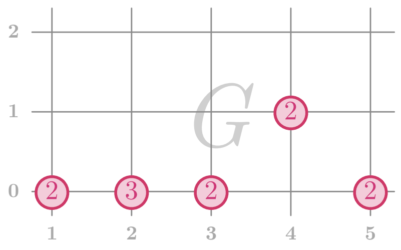


Figure 8.1: Top: Suffix tree of Hon et al.'s solution with a canonical naming of the inner nodes. Each node v has attached its F-list, which excludes pointers from leaves. Bottom left: Bitvector H is built by concatenating the unary encoding of the number of pointers in the F-list of each node. Bottom right: Resulting grid.

Therefore, all the pointers leaving from nodes in the subtree of v are stored contiguously in the grid, and can be obtained by finding the 0s that are between the $(sp-1)$th and the $(ep-1)$th 1 in H (as the 0s are placed before their corresponding 1). Note that no LCA operation on any suffix tree topology is necessary, only operation $\text{SELECT}_1(H,p)$.

Note that, since we do not represent the suffix tree topology, we cannot compute the depth of the locus node v, $depth(v)$. Instead of node depths, we will store the string depths of the nodes, $|path(v)|$, as the y-coordinates. Thus the query can use simply $|P|-1$ as the y-coordinate limit of the 3-sided query. That is, after applying the mapping using H, we find the k heaviest points in the range $[sp', ep'] \times [0, |P|-1]$ of G. The top-k algorithm using this mapping is considerably simplified, as can be seen in Algorithm 21.

Algorithm 21: Top-k algorithm using H-mapping index

Input: Query pattern P, amount k of documents to be retrieved

Output: top-k documents

```
00  TOPK_HMAP($P, k$)
01     $result \leftarrow \emptyset$
02     $[sp, ep] \leftarrow$LOCATE($P$)
03     $[sp', ep'] \leftarrow [1, 0]$
04     if $sp > 1$ then
05        $sp' \leftarrow$SELECT$_1(H, sp - 1) - (sp - 1)$
06     if $ep > 1$
05        $ep' \leftarrow$SELECT$_1(H, ep - 1) - ep$
06     $result \leftarrow$ TOPK_GRID_QUERY($sp', ep', 0, |P| - 1, k$)
07     if $|result| < k$ then
08        $result \leftarrow$ DOCUMENT_LISTING($sp, ep, result, k, sp$)
09     return $result$
```

8.2.2 Smaller Grid Representations

In the basic implementation (WRMQ), the grid G is represented with a combination of a wavelet tree and range maximum query (RMQ) data structures, that ensure query time $O((k + \log\log n)(\log\log n + \log k))$ if the suffix tree is of height $O(\log n)$. The price of this guaranteed worst-case time is a heavy representation of the grid, which triples the space of the basic wavelet tree of the y-coordinates vector. In addition, it has to store the vector FREQ in absolute form. We consider two ways of reducing this space.

K^2**Treap**^{H} We can use for G a K^2-treap, which as explained is a representation that compresses the coordinates and FREQ. Even if the grid is not square as we described it, we complete it to a square grid (the added areas having no points). The price of this compression is that there are no good worst-case search time guarantees. However, the experiments will show that this grid representation uses less space and yields a query time comparable with that of the WRMQ implementation.

W1RMQ^H We can use the wavelet tree but store only one RMQ structure in it, aligned to the level of the leaves (that is, it is the RMQ of FREQ). Therefore, instead of tripling the wavelet tree space, we only add $2n$ bits. The price is that, instead of solving the top-k query on the $O(\log r)$ maximal nodes that cover $[y_1, y_2]$, we must project them to the leaves before starting the process of filling the priority queue. Therefore, the $\log r$ factor in the cost may rise to r. In exchange, since the nodes are already leaves, finding the document identifiers and weights costs $O(1)$. The oveall impact is not that high if the strings $path(v)$ are of length $O(\log n)$ as before, since then $r = O(\log n)$ and the cost of this part becomes $O((k + \log n)\log k)$.

8.2.3 Efficient Construction for Large Collections

A bottleneck not addressed by the WRMQ implementation is the efficient construction of the index[1]. Both time-and space- efficiency have to be considered. We will concentrate here on the construction of H, the grid G, and its K^2-treap representation, suffix tree (CST) of $\mathcal{C}$ and a wavelet tree over the document array, WTD, of $n \log D$ bits. We perform a depth-first-search traversal on the CST and calculate, for each node v_j, the list of its document id marks, by intersecting the document array ranges of v's children. This can be done using the intersection on wavelet trees [74]. Since we can calculate the nodes in the order of their names, we can write H and the document ids (DOC) directly to disk. In a second traversal we calculate the pointers. For each document i we use a stack to maintain the string depth of the last occurrence of i in the tree. For a node v marked with i we push the string depth of v at the first visit and pop it after all the subtree of v is traversed. Note that this time we can read H and DOC from disk (in streamed mode) and avoid the intersection. In the same traversal we can calculate the weight array FREQ by performing counting queries on WTD. Again, arrays y and FREQ can be streamed to disk.

Finally, the K^2-treap is constructed in-place by a top-down level by level process. Let the input be stored as a sequence of triples (x, y, w) and let 1 be the root level and b be the bottom level. First, we determine the heaviest element by a linear scan, stream its weight out to disk and mark the element as deleted. We then partition the elements of the root level into K^2 ranges, such that all elements in range r $(0 \leq r \leq K^2)$ have the property that $x/K^{b-\ell} \bmod K = r \bmod K$ and $y/K^{b-\ell} \bmod K = r/K$. For each non-empty (resp. empty) range r we add a 1 (resp. 0) to the bitvector T, which is streamed to disk. On the next level $\ell - 1$ we can detect nodes by checking the partitioning condition of the last level, find and mark the maximum weighted entry and apply the partitioning in the node. The time complexity of the process is $O(K^2 \log_K n)$ and does not use extra space.

8.2.4 Summing Up

We introduce new indexes, K^2Treap^{H}and W1RMQ^H, that simplify WRMQ by converting its MAP and TREE components into a single bitmap H. In addition, they reduce the space of the grid representation, by trading it for weaker time guarantees. The experiments will show that this does not translate into markedly reduced performance.

[1]For example, the liner-time LCA-based index construction method takes 1.5 hours for an 80MB Wikipedia collection, where their peak main memory usage is 12.25 GB.

Chapter 9

Experimental Evaluation

In this chapter we describe the experimental setup in terms of the collections where the index were evaluated, query generation and environment employed. We also explain the engineering details required to implement the indexes.

9.1 Datasets and Test Environment

We will split the collections into two categories depending on the alphabet type: character alphabet and word alphabet. For the character alphabet, we have four "small" collections, that have been used as baselines in previous work [124], and a "big" collection containing natural language text. We describe and label the collections as follows:

- KGS^c. Consists of a collection of 18,839 sgf-formmated Go game records from year 2009 (`http://www.u-go.net/gamerecords`), containing 52,721,176 characters.
- PROTEINS^c. A collection of 143,244 sequences of Human and Mouse proteins (`http://www.ebi.ac.uk/swissport`), containing 59,103,058 symbols.
- ENWIKI-SML^c. A sample of a Wikipedia dump, consisting of 4,390 English articles, containing 68,210,334 symbols.
- DNA^c. A sequence of 10,000 highly repetitive (0.05% difference between documents) synthetic DNA sequences with 100,030,016 bases in total.
- ENWIKI-BIG^c. A bigger sample of English Wikipedia articles, consisting of 8.5GB of natural text and 3.8 million documents.

In the case of word alphabets we use three collections of documents containing natural language. We preprocessed these collections using the Indri search engine (`http://www.lemurproject.org/indri/`) for generating a sequence of stemmed words and excluding all html tags.

- ENWIKI-SML^w. A word parsing of ENWIKI-SML^c, containing 281,577 distinct words.
- ENWIKI-BIG^w. A collection containing 8,289,354 words obtained from regarding the

text as a sequence of words from ENWIKI-BIG^c.

- GOV2^w. Probably the most frequently used natural text collection for comparing efficiency of IR systems. The parsed collection consists of more than 72GB of data, around 40 million terms and more than 25 million documents.

Table 9.1 summarizes properties of the collections that are used. For queries, we selected 40,000 random substrings of length m obtained from the top-k documents for $m = 5$. We increase k exponentially from 1 to 256.

For all experimental comparisons, the relevance measure used is the term frequency and the query is a single pattern string. All experiments were run on a server equipped with an Intel(R) Xeon(R) E5-4640 CPU clocking at 2.40GHz. All experiments use a single core and at most 150GB of the installed 512GB of RAM. All programs were compiled with optimizations using g++ version 4.9.0.

9.2 Implementations

In this section we describe the implementation details for each engineered index. Our implementation and benchmarks are publicly available at `https://github.com/rkonow/surf/tree/wt_topk`.

9.2.1 Baselines

As baselines we use the GREEDY solution [53] and the SORT approach [77], which is included in the Succinct Data Structure Library (SDSL, https://github.com/simongog/sdsl-lite). Baseline SORT uses the CSA and document array DA. It first gets the range $[sp, ep]$ of all occurrences of P from the CSA. Then it copies all documents from DA$[sp, ep]$ and extracts the top-k most relevant documents by sorting and accumulating the occurrences. Baseline GREEDY also gets the range $[sp, ep]$ from the CSA, but then does a greedy traversal of the wavelet tree over DA to get the top-k documents. As an additional baseline we also ran the experiments using the original source code of Navarro et al. [124] (NPV). This implementation consists of more than 24 alternative configurations, including grammar compressed wavelet trees using different bitmap representations and improvements to the GREEDY algorithm. For the experimental time-related results, we will always show the points that correspond to the pareto-optimal border in terms of space and time considering all possible configurations. We refer to this index as as NPV^{opt}. We will not compare to the work of Patil et al. [129] since the space requirements of this solution is considerably bigger (see Table 6.1) than the ones that are being evaluated in this work.

Collection	n	D	n/D	σ	$\lvert\mathcal{C}\rvert$ in MB
character alphabet					
KGS^c	52,721,176	18,839	2798	75	51
PROTEINS^c	57,144,040	143,244	412	40	56
ENWIKI-SML^c	68,210,334	4,390	15,538	206	65
DNA^c	100,020,016	10,000	1002	4	97
ENWIKI-BIG^c	8,945,231,276	3,903,703	2,291	211	8,535
word alphabet					
ENWIKI-SML^w	12,741,343	4,390	2,902	281,577	29
ENWIKI-BIG^w	1,690,724,944	3,903,703	433	8,289,354	4,646
GOV2^w	23,468,782,575	25,205,179	931	39,177,922	72,740

Table 9.1: Collection statistics: n is the number of characters or words, D the number of documents, n/D the average document length, σ the alphabet size, $\mathcal{C}$ and total size in MB assuming that the character based collections use one byte per symbol and the word based ones use $\lceil \log \sigma \rceil$ bits per symbol.

9.2.2 Implementation of WRMQ

The implementation of WRMQ requires the assembly of a complex set of compact data structures: CSA, wavelet tree (WT), RMQ, rank/select-capable bitmaps, direct access codes, compact tree topologies and an efficient integer array representation. In practice, for the CSA, we use an off-the-shelf SSA from *PizzaChili* site, (`http://pizzachili.dcc.uchile.cl`), and add a rank/select capable bitmap, `BitSequenceRG`, implemented in the LIBCDS library (`https://github.com/fclaude/libcds`), that requires 5% of extra space. Bitmap DOCBITMAP is used to mark where each distinct document starts in $\mathcal{C}$. This way, the document corresponding to SA$[i]$ is obtained by performing $\text{RANK}_1(\text{DOCBITMAP}, \text{SA}[i])$. We also use the 5% space overhead bitmaps for the MAP bitmap $B[1, 2n]$. We use the fully-functional compact tree representation [14] requiring 2.3 bits per element to perform LCA and PREORDER_SELECT over the suffix tree topology. For LEAF_RANK and LEAF_SELECT we use another bitmap $L[1, 2n]$ where the leaves are marked, and we use 5% space overhead to implement $rank_1$ and $select_1$ (this could be slightly improved but makes little difference). We employ the original optimal implementation of direct access codes for representing the weights. For the wavelet tree, we use the no-pointers version from LIBCDS (`WaveletTreeNoPtrs`) using `BitSequenceRG` to represent the bitmaps. Recall that the wavelet tree has to be enhanced to support range maximum queries (RMQ) at each of the weight sequences $W(v_i)$. We implemented the RMQ structure that requires 2.3 bits per element [69]. For the document identifiers we use an integer array that requires $\lceil \log(D + 1) \rceil$ bits per element and access any position directly. All of these implementations and data structures were limited to handle $2^{32} - 1$ memory addresses, thus this index is not capable of handling big datasets such as ENWIKI-BIG^c.

9.2.3 Implementation of K^2Treap^H

The set of data structures required for implementing K^2Treap^H is smaller: We need a CSA, RMQ, rank/select capable bitmaps, direct access codes, a K^2-treap, and an efficient integer array representation. We based our implementation on structures from SDSL and will use the SDSL class names in the following. For character based indexes, we use a CSA based on a wavelet tree (`csa_wt`) which is parametrized by a Huffman-shaped wavelet tree (`wt_huff`) which uses a compressed bitmap (`rrr_vector`). For word based indexes, we opted for a CSA based on the Ψ function (`csa_sada`) which is compressed with Elias-δ codes (`coder::elias_delta`). The latter provided a better time-space trade-off for large alphabet pattern matching. For the mapping we store the bitmap H using a compressed bitmap `rrr_vector`. The $select_1$ performance of `rrr_vector` is slow compared to a plain bitmap representation but negligible in our case, where only two $select_1$ queries are done per top-k query (recall Algorithm 21). The RMQC structure is realized by a MinMax-tree implementation `rmq_succinct_sct` which uses about 2.3 bits per element. The document ids, DOC, are stored in an integer vector (`int_vector`) using $\lceil \log(D+1) \rceil$ bits per element. For this work we have implemented the K^2-treap structure and added it to SDSL (available now as `k2_treap`). The K^2-treap implementation is generic (the value K, the bitmap representation of the K^2-ary tree, and the representation of the vector of relative weights, can be parametrized) and complements the description of Brisaboa et al.[32] with an efficient in-place construction described in Section 8.2.3. For the relative weights we use direct access codes (`dac_vector`) with a fixed width of 4.

9.2.4 Implementation of W1RMQ^H

The only difference between K^2Treap^H and W1RMQ^H is the grid implementation. The W1RMQ^H solution uses a wavelet tree and an RMQ data structure to represent the grid. We opted for a `wt_int` parametrized with `rrr_vector`. The compressed bitvector decreased the size considerably since the grid contains many small y-values. The absolute weights are stored in a `dac_vector` of fixed width 4, which gave the most practical result when performing parameter tuning experiments. The RMQ structure was again realized by `rmq_succinct_sct` using 2.3 bits per element.

9.3 Results

In this section we study the practical properties of the different index implementations. We first analyze the space consumption of the structures; both total space and the space of substructures – like the grid – are considered. Then the query time is analyzed. Again, we consider total query time and also study the time for the different phases of the query process. To get a precise picture about the performance we vary various parameters: the number of results k, the pattern length m, and collections. The latter are from different application domains, cover different scales of magnitude – from 50 MB to 72 GB –, and vary in alphabets size from 4 to almost 40 million. Note that some non-SDSL based baselines did not support alphabets larger than 256 and collection sizes above 2 GB and were therefore only evaluated in some setups.

9.3.1 Space

We start by decomposing the space required of our first index, WRMQ. Fig. 9.1 shows the space required for each of its components in terms of fraction of the input. For most collections the space requirements are quite similar, except for the KGS^c collection. In general, WRMQ requires 2–4 times the size of the input. If we analyze the space of the components, the most resource-consuming piece is the wavelet tree with multiple RMQ structures, which requires up to the size of the collection. The CSA requires 0.6–0.7 times the collection size. Recall that for this implementation we need to store the topology of the suffix tree (TREE) plus two bitmaps to perform the mapping of the suffix tree nodes to the grid (MAP). These data structures add up to 0.6 times the size of the input. Storing the frequencies (FREQ) using direct access codes use 0.2–0.3 times the size of the input, which is about the same size of the tree topology. The space requirements to represent WRMQ is probably unpractical for real scenarios. Furthermore, the implementation of this index is not able to handle collections that are bigger than 200MB nor collections that are parsed as words.

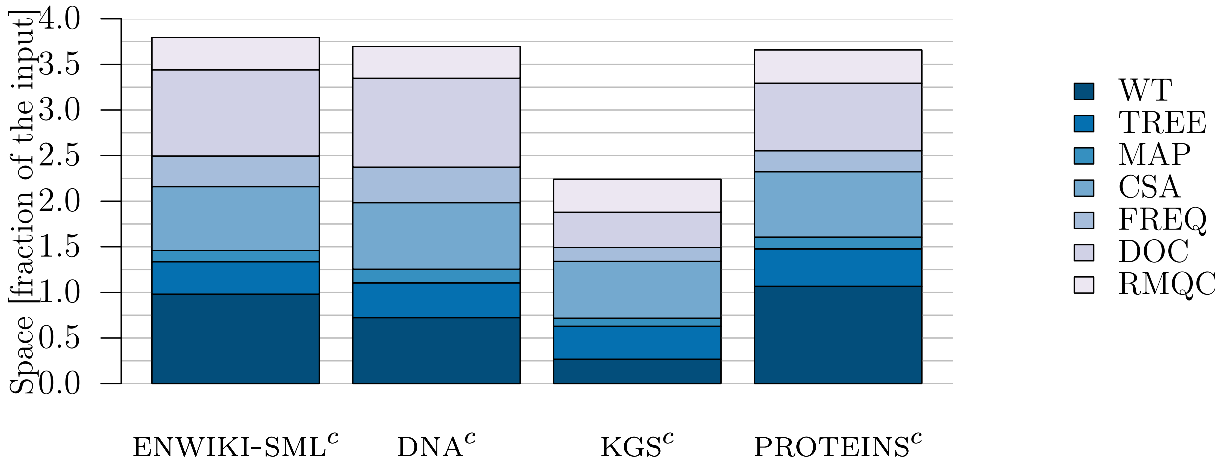


Figure 9.1: Space usage decomposition as fraction of the input, for each structure employed in WRMQ.

Fig. 9.2 shows the decomposition of the index K^2Treap^H, which employs the H bitmap to map the suffix tree nodes to the grid. In addition, this index uses the K^2-treap to represent the grid G. The efficient representation of G is about 30% smaller than WRMQ. We exemplify the space reduction of K^2Treap^H compared to WRMQ in the case of ENWIKI-SML^c. While the grid mapping takes 46.5 MB (8.6+12.8+25.1 for bitvector B, L, and the tree topology) in WRMQ, K^2Treap^H just takes 6.8 MB for bitvector H (13.0 MB in uncompressed form). The grid representation as K^2-treap takes 57.7 MB ($21.7 + 24.5 + 11.5$ for weights, y-values, and topology) compared to 77.7 MB (23.3 MB for weights and 55.4 MB for the RMQ-enhanced wavelet tree) in WRMQ. These space savings result in index sizes between 1.5–3.0 times the original collection size for character alphabet collections, see Fig. 9.2 top. For word-parsed collections, IDX_GN takes space close to the original input, for example, for the 71.0 GB word parsing of GOV2^w, the index size is 64.6 GB. Recall that this space includes the CSA component, which can recover any portion of the text collection, and thus the collection does not need to be separately stored.

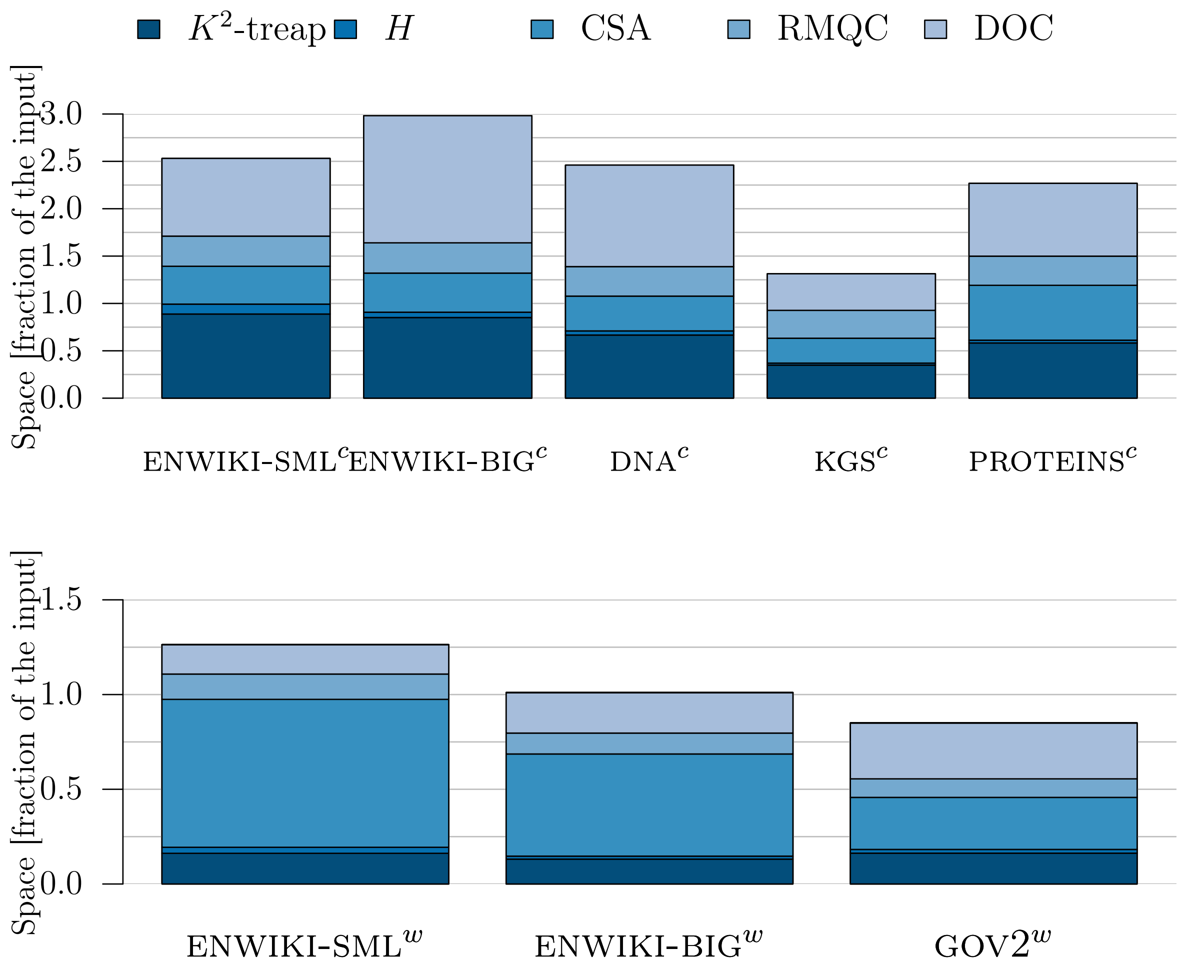
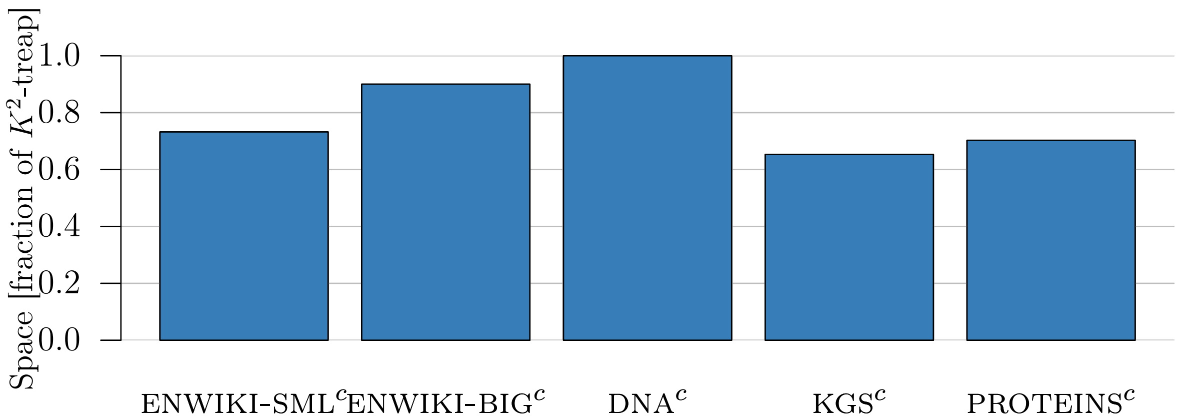


Figure 9.2: Space usage decomposition as fraction of the input, for each structure employed in K^2Treap^H. Top: results for character alphabet collections. Bottom: results for word alphabet collections.

The difference between K^2Treap^H and W1RMQ^H is the grid representation, so we compare the space required by the bit-compressed wavelet tree with a single RMQ structure to the space required of the K^2-treap (see Fig. 9.3). In this case, the y-axis represents the fraction of the space required for the wavelet tree-based grid to the space required to represent the grid using the K^2-treap. We observe that, in most of the character based collections, except for the case of DNA^c, the space savings of the wavelet tree are 30% to 40%. A similar result can be seen in the case of word collections.

Fig. 9.4 shows the space comparison in terms of fraction of the input of all of the introduced indexes and the baselines for the character alphabet collections. ENWIKI-BIG^c is not shown due to the implementations constraints of some of the indexes. As previously mentioned, NPV^{opt} chooses the best possible result obtained from all the alternative config-



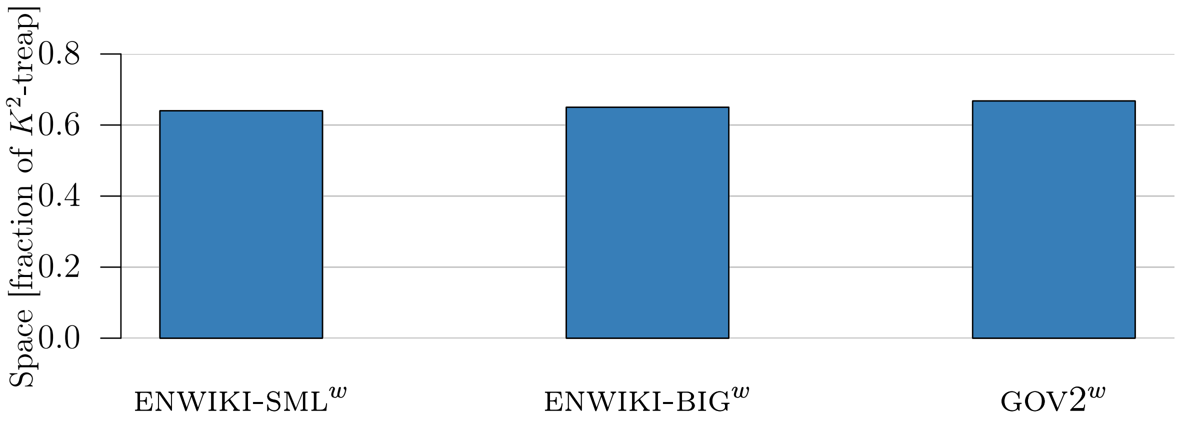


Figure 9.3: Comparison of the space required to represent the grid G using a wavelet tree plus a single level of RMQ (W1RMQ^H) in terms of the space required if the K^2-treap is used.

urations of the implementations presented by Navarro et al. [124]. In this case, we show the variants that yield the best compression for each collection (NPV^{min}) and the ones yielding the highest space usage (NPV^{max}). We also show the space requirements of the other baselines, GREEDY and SORT. For almost all cases WRMQ is the most space-demanding index, even when compared to the uncompressed version NPV^{max}. Only for the KGS^c collection is WRMQ smaller than NPV^{max}. W1RMQ^H and K^2Treap^H require about 20% more space than other variants for ENWIKI-SML^c and DNA^c. However, the space of these indexes for KGS^c is about half of GREEDY and SORT, and they are smaller than the most compressed solution, NPV^{opt}.

9.3.2 Retrieval Speed Varying k

The time of a top-k query consists of the time to match the pattern in the CSA, the time to map $[sp, ep]$ to the grid, the time to report the top-k documents using the K^2-treap or wavelet tree and, if less than k documents are found, the time to report frequency-one documents (which are not stored in the grid) using RMQC and, again, the CSA. The top-k retrieval

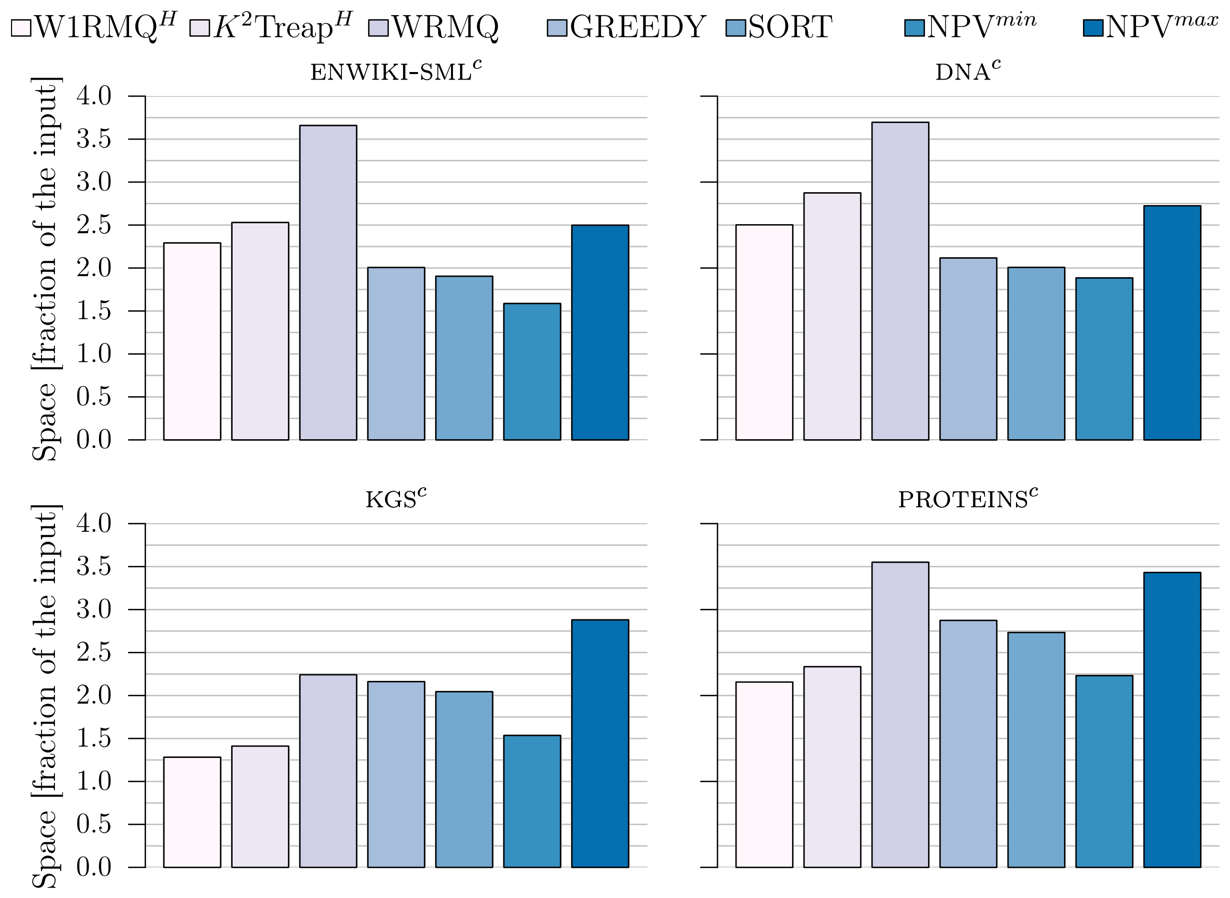


Figure 9.4: Space as fraction of the input for small character alphabet collections. NPV^{min} is the smallest possible result obtained from Navarro et al. [124], while NPV^{max} corresponds to the uncompressed variant.

time for a pattern P depends on multiple factors: First, the length m of P. For an FM-index based CSA (resp. Ψ-function based) it takes $O(m \log \sigma)$ (resp. $O(n \log n)$) steps. Second, the time for the mapping from the lexicographic range into the grid's x-range. This method, described in Algorithm 21, requires two $select_1$ operations, whose time is negligible compared to the first step. The last steps depend on the output size of the query, k, compared to the number of occurrences of the pattern. Documents in which the pattern occurs more than once are calculated via the corresponding top-k grid query (either K^2-treap for K^2Treap^H or wavelet tree for WRMQ and W1RMQ^H), while documents in which the pattern just occurs once are retrieved via the document listing algorithm that uses the RMQC structure on CA, a locate operation on the CSA, and a $rank_1$ on DOCBITMAP. We examine the cost of the different phases in Fig.9.5 for K^2Treap^H.

As expected, the pattern matching with the CSA is independent of k and takes about 5–10μs. The time to retrieve the first document out of the K^2-treap is relatively expensive. For most of the collections it is about 40–70μs and is dominated by the cost of the priority-queue based search down the K^2-treap until a first (heaviest) element within the query range

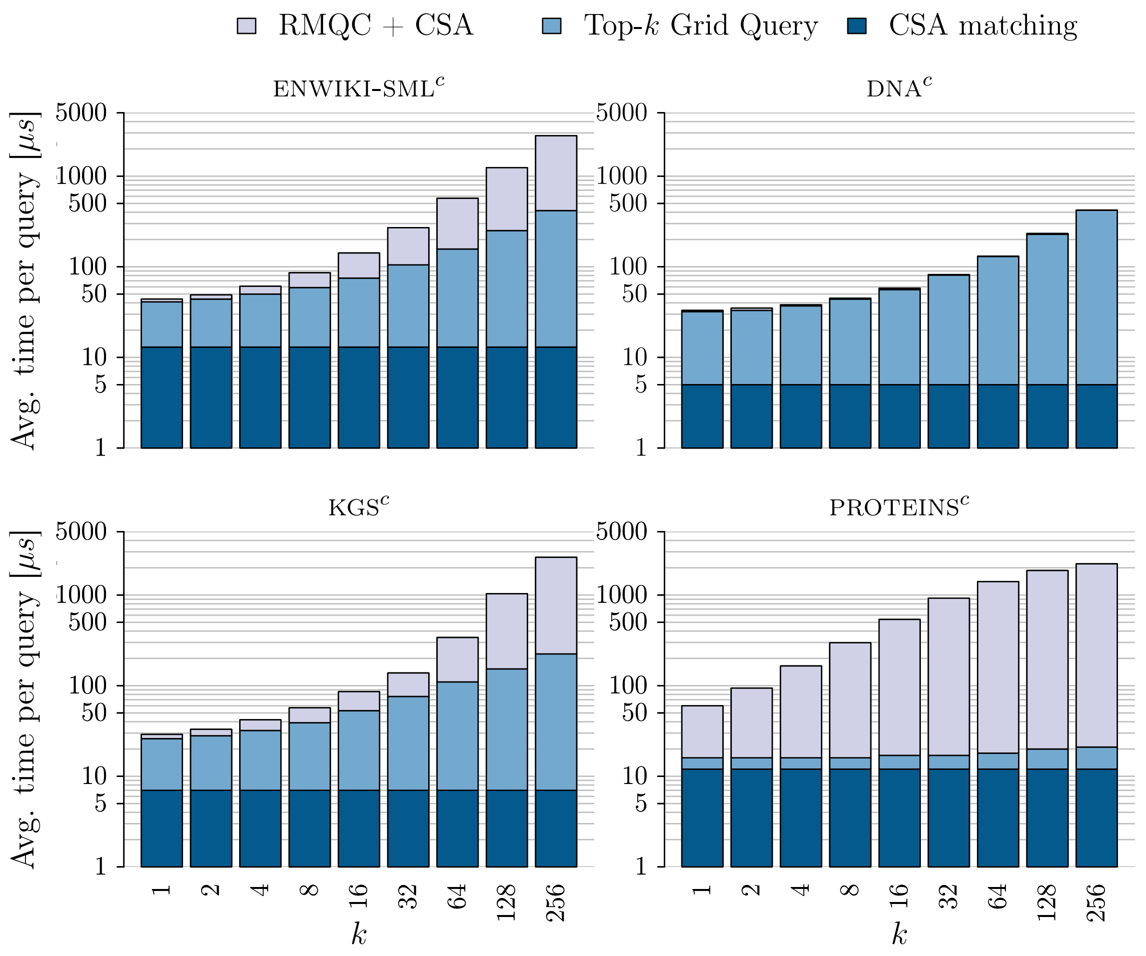


Figure 9.5: Detailed breakdown of average query time for index $K^2\text{Treap}^H$ which uses a CSA pattern search, a K^2-treap for the top-k grid query, and the document listing algorithm that requires RMQ queries plus CSA accesses (time fractions plotted from bottom to top in log-scale). We use $m = 5$.

is found.

The subsequent documents are cheaper to report. The time spent in the K^2-treap to report 16 documents is about twice the time to report a single document except for the case of PROTEINS^c. The average time per document retrieved via the K^2-treap is typically about 3–5μs for $k \geq 64$. Essentially, for each such document, a constant number of RMQ and extraction of a suffix array cell from the CSA are performed. The cost of an RMQ is typically below 2μs [76, Sec. 6.2], while the CSA access accounts for the remaining 100–300μs. The CSA access time is linearly dependent on a space/time tradeoff parameter s, which is set to $s = 32$. Note that top-k queries are meaningful when documents have different weights, and thus large k values that retrieve many documents with frequency 1 are not really interesting. An interesting case arises for the DNA^c collection, where the amount of time spent to retrieve documents with frequency 1 is negligible. Recall that DNA^c contains synthetic highly repetitive sequences, and a limited alphabet size $\sigma = 4$, therefore it is quite improbable to generate a query-string of length $m = 5$ that has a single occurrence.

Fig. 9.6 shows the results for our biggest character alphabet dataset, ENWIKI-BIG^c. On the left part of the figure we show the time required to perform each part of the query process as in Fig. 9.5. On the right, we show average time per document retrieved and perform a breakdown depending on the mechanism that was employed. For values of $k \leq 8$, the weighted average amount of time spent to retrieve each document is always greater than $10\mu s$. As the value of k increases, the average time to report a document decreases. For $k = 256$ the average time required is less than $5\mu s$. Note that for all k values (for $k \geq 4$, since for smaller k values, it was not necessary to execute the single-occurrence procedure) the time for the RMQC+CSA accesses to report a document is about 60–90μs.

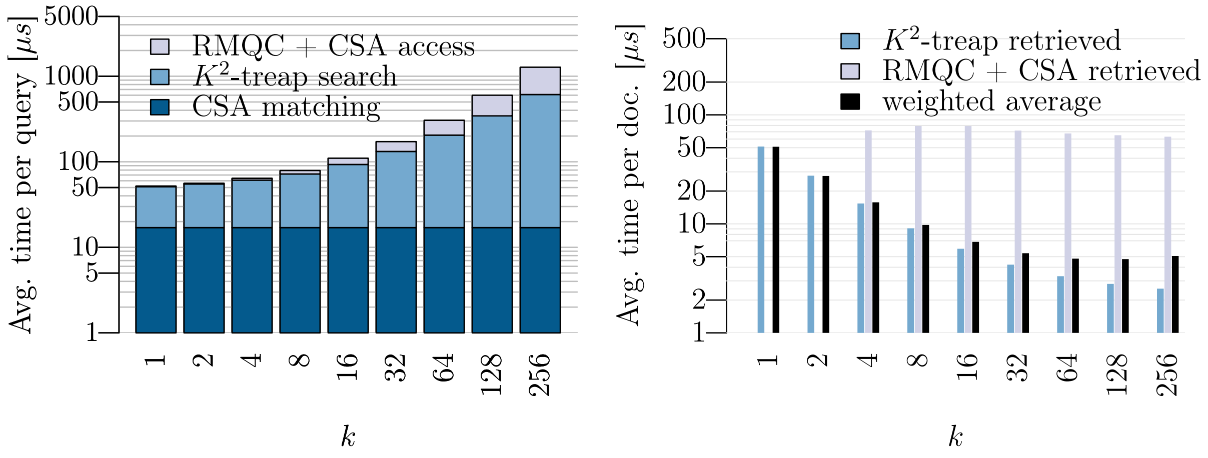


Figure 9.6: Query times for IDX_GN on ENWIKI-BIG^c, with $m = 5$. Left: Query time depending on k with a detailed breakdown of the three query phases. Right: Average time per document (mixed), per K^2-treap retrieved document and RMQC+CSA retrieved document. CSA matching time is included in all cases.

We now examine the performance of the K^2-treap grid representation compared to the use of the wavelet tree alternative. We use the implementation of W1RMQ^H using compressed bitmap representations and only one RMQ level, since the results obtained with WRMQ were almost identical, except for PROTEINS^c where WRMQ is 3 times *slower* due to the many RMQs performed. Fig. 9.7 shows the comparison of the average time required to complete a top-k query, for varying k, using the K^2-treap or the wavelet tree. For most cases, the wavelet tree-based approach is faster than the K^2-treap, except in the DNA^c collection, where for $k \geq 8$ the K^2-treap is faster. The most significant difference can be seen on ENWIKI-SML^c, where on average the wavelet tree is two times faster. Fig. 9.8 shows the times on our biggest character alphabet collection (ENWIKI-BIG^c): the wavelet tree is about 50% slower for $k \leq 64$. This difference gets smaller for larger k values, and for $k = 256$ the wavelet tree is already 10% faster than the K^2-treap. This is quite different from the results obtained on the small collection of the same type (ENWIKI-SML^c), where the wavelet tree was up to 5 times faster for $k = 256$ and dominated the average time per query for all the k values.

We compare the overall performance of our indexes against the baselines in Fig. 9.9. Recall that index NPV^{opt} represents the best achieved result from all alternatives in terms of space/time. We show the results using the smaller character alphabet collection since there are implementation constraints for NPV^{opt} and WRMQ. Except for the PROTEINS^c

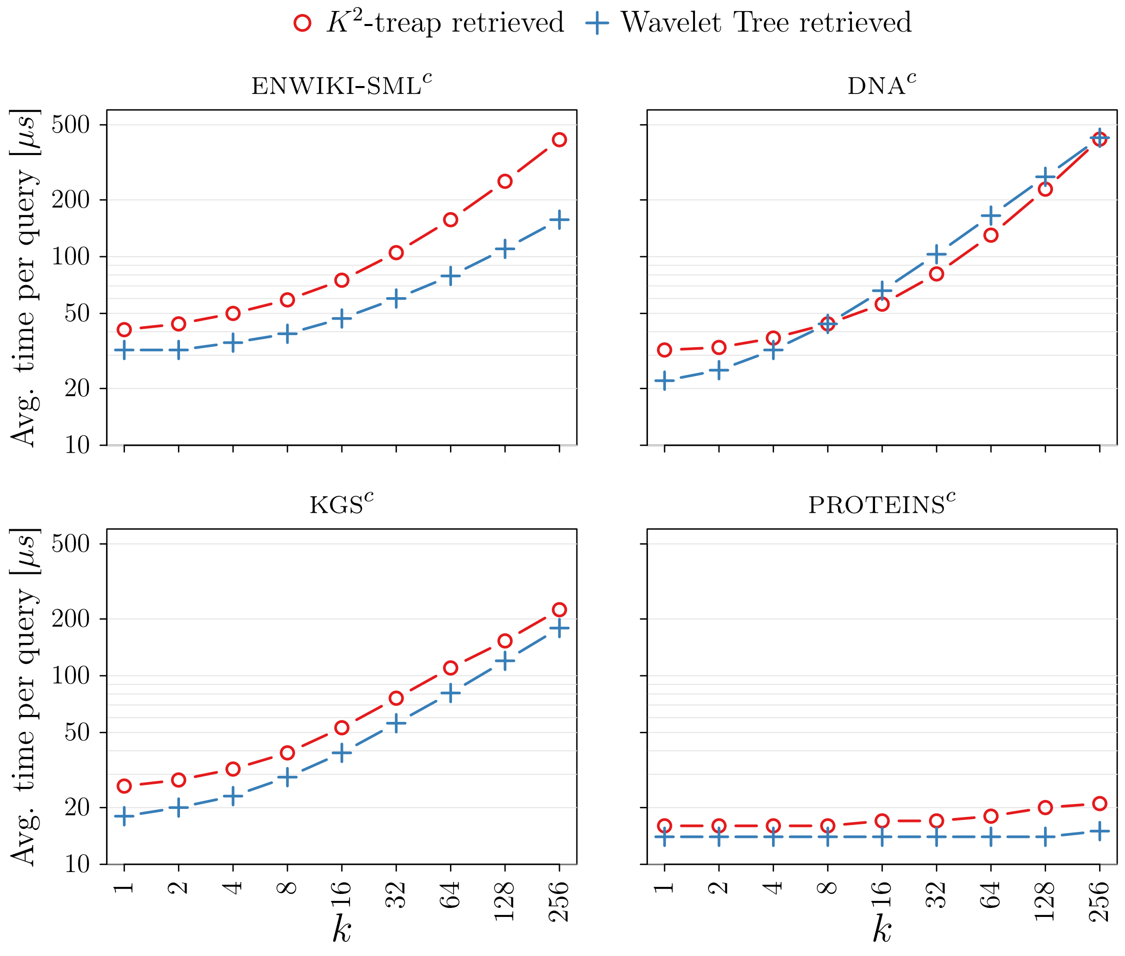


Figure 9.7: Comparison of the average time per query required to retrieve the top-k results using a K^2-treap or the wavelet tree for representing G.

collection, all of our our indexes require less than $200\mu s$ for $k \leq 16$. We start by analyzing the performance for the English Wikipedia collection (ENWIKI-SML^c). Our indexes are faster than all other approaches for $k \leq 32$. For larger k values, the best combination of NPV^{opt} is faster than our approaches by up to a factor of 3 ($k = 256$). In the case of the DNA^c collection, all of our indexes are up to 12 times faster than the fastest alternative (NPV^{opt}). As mentioned before, the performance of WRMQ is similar to that of W1RMQ^H and K^2Treap^H. The extra RMQ operations performed at each wavelet tree level adds a constant time factor to the query time, as can be clearly observed from the results on DNA^c and PROTEINS^c. In the case of KGS^c, the difference between our indexes and the other approaches is considerably bigger: our approaches are 21 times faster than the closest alternative (NPV^{opt}) for $k = 2$, and even for larger k values all of our approaches are up to 3 times faster than the fastest baseline. Finally, in the case of PROTEINS^c, our results show that there is no alternative that is faster than the most basic approach, which is sorting (SORT). In this case, our indexes are slower than SORT for all k values, and also than NPV^{opt} for $k \geq 4$. The constant factor added to the query times for the extra RMQ operations of WRMQ is more evident for this collection, making this alternative up to 10 times slower than W1RMQ^H and K^2Treap^H. Note that in this collection most patterns occur once in each document, thus top-k queries are equivalent

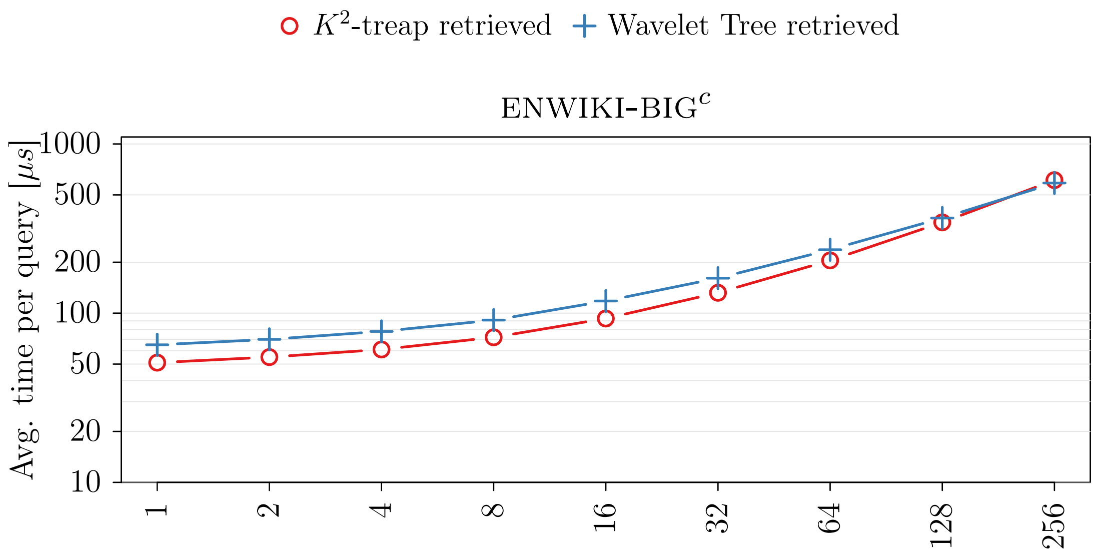


Figure 9.8: .
Comparison of the average time per query required to retrieve the top-k results using a K^2-treap or the wavelet tree for representing G for ENWIKI-BIG^c.

to plain document listing queries.

Fig. 9.10 shows the results for the bigger character based collection (ENWIKI-BIG^c). Note that for this experiment we are not able to construct the index using NPV^{opt} and WRMQ since their implementations are not able to handle more than $2^{32} - 1$ memory addresses. For all distinct k values, we are up to 100 times faster than the fastest baseline (GREEDY) and even 1,000 times faster than the simple SORT method. The reason for this is that GREEDY, as well as SORT, have to handle larger $[sp, ep]$-intervals for this bigger collection, while our index is less dependent on this factor. As mentioned before, it turns out that for the case of the collection ENWIKI-BIG^c W1RMQ^H is slower than K^2Treap^H for values of $k \leq 128$. Therefore, the difference between heuristics and indexes with stronger guarantees shows up as the data sizes increase.

9.3.3 Varying Pattern Length

We proceed to analyze the effect of changing the query pattern length m. As before, we selected 40,000 random substrings of lengths m ranging from 3 to 10 and obtained the top-k documents for fixed $k = 10$. Fig. 9.11 shows the results, in terms of average time per query for different query pattern lengths. For the collection ENWIKI-SML^c, our index query time ranges in 80–110μs. Note that for query patterns longer than 8 symbols, SORT is the fastest index. Since the ranges in the wavelet tree for longer patterns are smaller, GREEDY starts to be competitive for $m \geq 7$. In general, the query times are quite similar for all of our indexes on this collection. A similar scenario arises on the DNA^c collection: our indexes range in 80–180μs, and for longer patterns SORT is the best alternative. Note that NPV^{opt} is 10

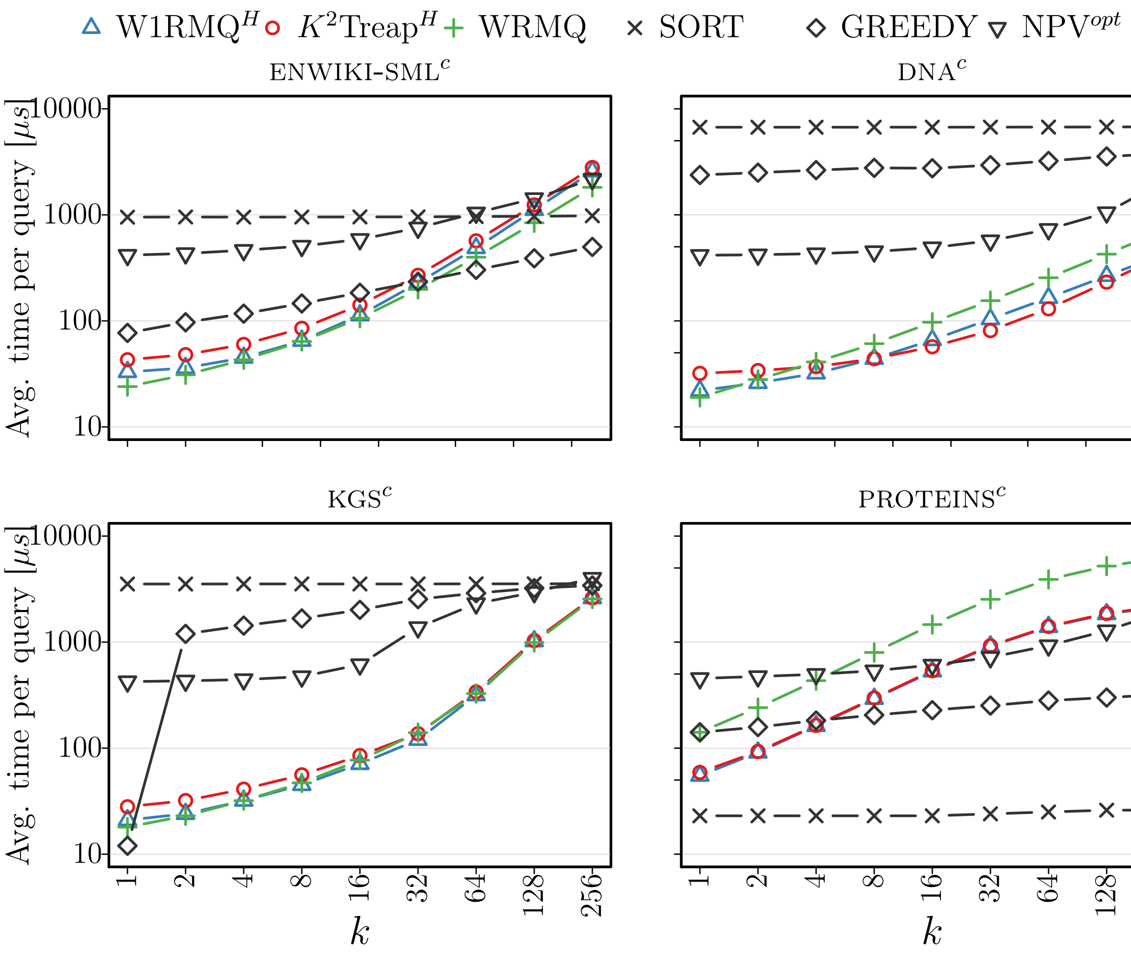


Figure 9.9: Average time per query, in microseconds, for different k values and fixed pattern length $m = 5$, evaluated on small character alphabet collections.

times slower for most m values. In the case of KGS^c collection, our indexes are the best for the whole range of pattern lengths and we still are one order of magnitude faster than GREEDY and NPV^{opt}. A special case arises for the PROTEINS^c collection. Our indexes are generally slower than the baselines, especially WRMQ, which is up to 5 times slower than W1RMQ^H and K^2Treap^H. Still, they are faster than NPV^{opt}. In this case, the best alternative is the simplest solution: the index SORT, for patterns having more than 4 symbols.

We show the results of our biggest character alphabet dataset, ENWIKI-BIG^c in Fig. 9.12. Recall that for this case, we are not able to compare with WRMQ and NPV^{opt} due to the implementation constraints already described before. Compared to SORT and GREEDY our indexes are up to 100 times faster for small pattern lengths and up to 8 times faster for the longest case ($m = 10$).

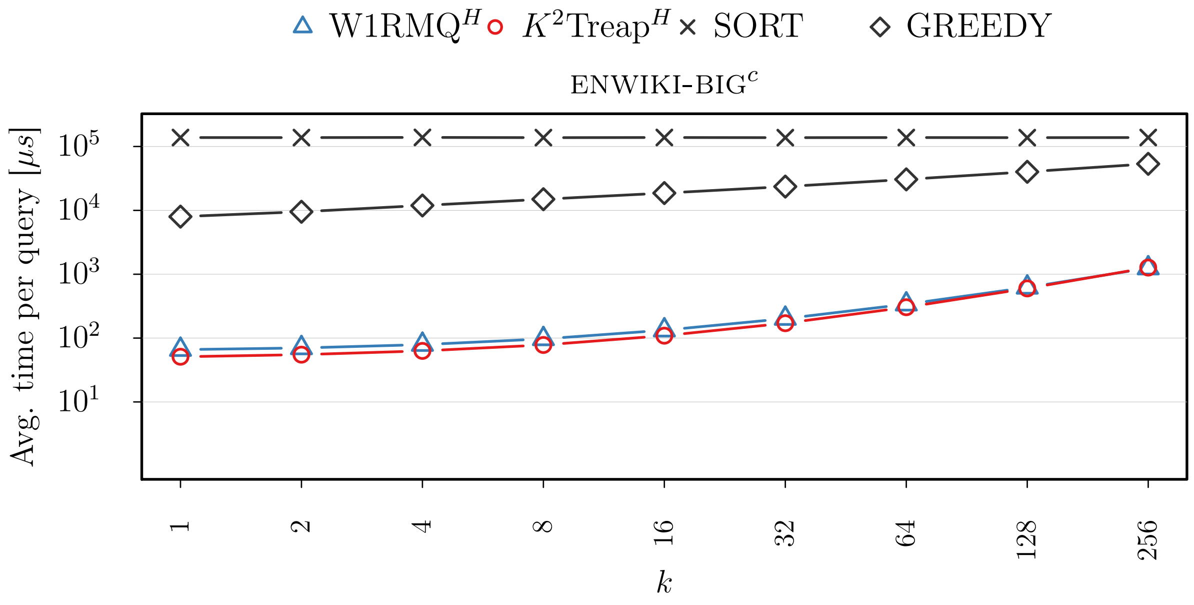


Figure 9.10: Average time per query in microseconds for varying k values and fixed pattern length $m = 5$ using big character alphabet collections.

9.3.4 Word Alphabet Collections

One of the most important feature of our best implementations (K^2Treap^H and W1RMQ^H) is that they are able to index collections with large alphabets, thus allowing the mapping of words to integer symbols. We measured the performance of our indexes and other baseline implementations on the word alphabet collections (ENWIKI-SML^w, ENWIKI-BIG^w, and GOV2^w). We used the same experimental setup, generating 40,000 single-word ($m = 1$) queries chosen at random from each collection $\mathcal{C}$ and increasing k exponentially from 1 to 256.

We start by analyzing the results with the same breakdown of the top-k procedure as done before for the character alphabet case. We show the details of the average query time required for the index K^2Treap^H on word alphabet collections in Fig.9.13. The pattern matching using the CSA takes less than $5\mu s$ for all cases. These results are considerably faster than those obtained on character alphabet collections. The main reason for this difference is that the CSA searches for a shorter pattern ($m = 1$ words), even if on a larger alphabet. In general the top-k grid query takes a great portion of the total time, of about 20–30μs for retrieving a single document. Interestingly, for ENWIKI-SML^w, the portion of total time spent performing the single-occurrence procedure (RMQC + CSA accesses) is much bigger than for the larger collections. This is expected, since ENWIKI-SML^w contains a small amount of document (4,390) when compared to ENWIKI-BIG^w (3,903,703) and GOV2^w (25,205,179), and thus in the latter it is more likely to find k documents where P appears more than once. In general, K^2Treap^H is up to three times faster in these types of collections than in the smaller, character alphabet ones.

We show the average time required to retrieve a document depending on whether it was retrieved using the K^2-treap or the single-occurrence procedure. We show the results obtained for the GOV2^w collection in Fig. 9.14. As before, retrieving a single document using either

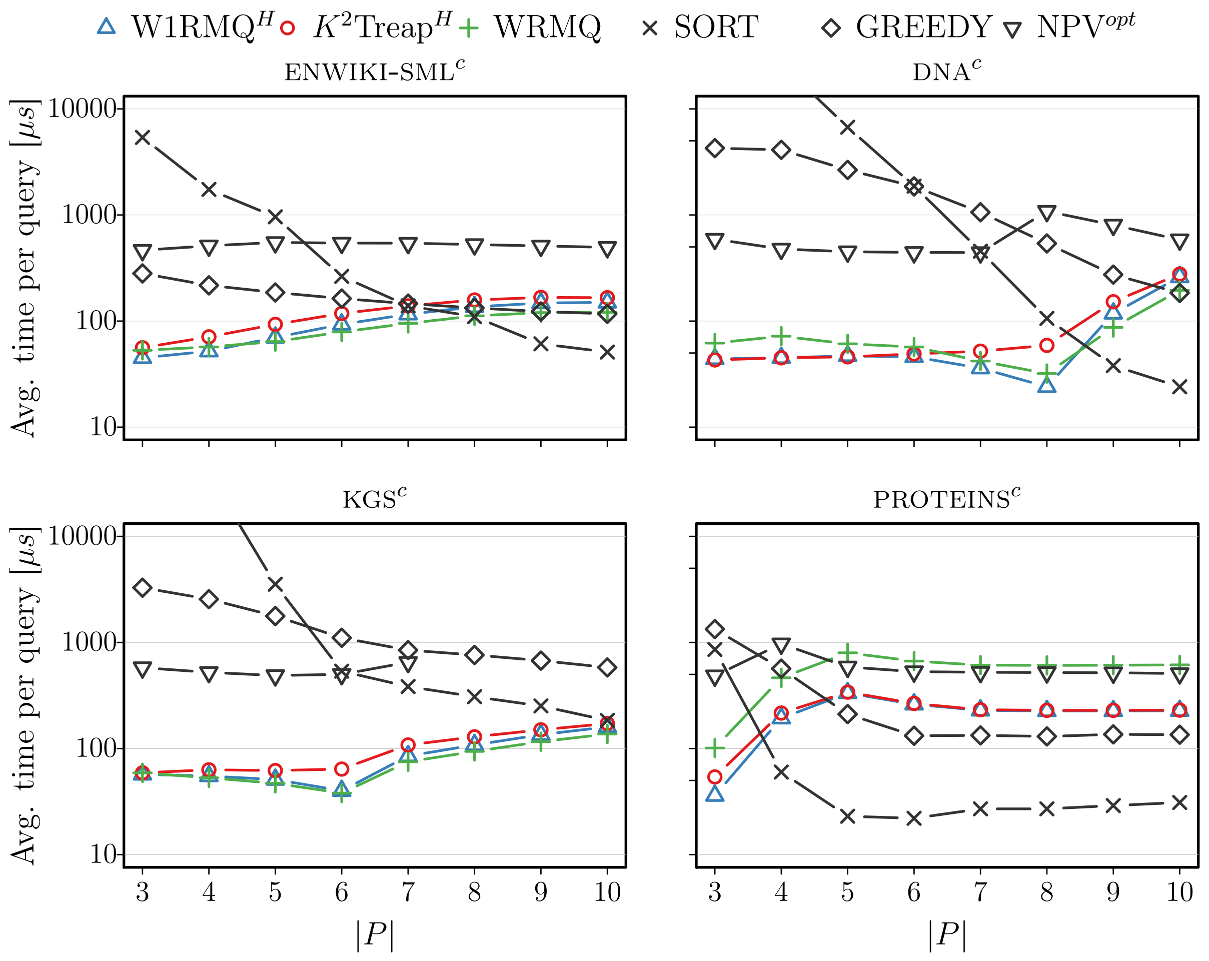


Figure 9.11: Average time per query in microseconds for different pattern lengths and fixed $k = 10$ value.

approach is costly: about $20\mu s$ with the K^2-treap and $100\mu s$ with RMQC + CSA accesses. The time per document decreases significantly as k increases. For $k = 256$, the average time to retrieve a single document is below $2\mu s$ for the K^2-treap and $31\mu s$ for the single-occurrence procedure. As before, this is because the K^2-treap spends some time until extracting the first result, and the next ones come faster. Instead, the document listing method has a more constant-time behaviour: After $k \geq 4$, the RMQC + CSA retrieval time does not decrease when more documents are requested.

Fig. 9.15 compares the two grid representations: the K^2-treap and the wavelet tree. Interestingly, for the small collection (ENWIKI-SML^w) the K^2-treap is slower than the wavelet tree, but it is considerably faster for larger collections (ENWIKI-BIG^w and GOV2^w). In detail, for the small collection, the wavelet tree is up to twice as fast as the K^2-treap, and for the larger ones, the wavelet tree is up to twice as slow. This result is also different when compared to the character alphabet collections, where in most cases the wavelet tree is faster than the K^2-treap. This is due to the x-range of the query: since the pattern length is $m = 1$, the x-ranges in the queries bigger for the large collections. This affects negatively the wavelet tree, because the two $rank_b$ operations used to map the interval $[x_1, x_2]$ are far apart and require separate cache misses. Instead, the K^2-treap has a higher chance of having the heaviest

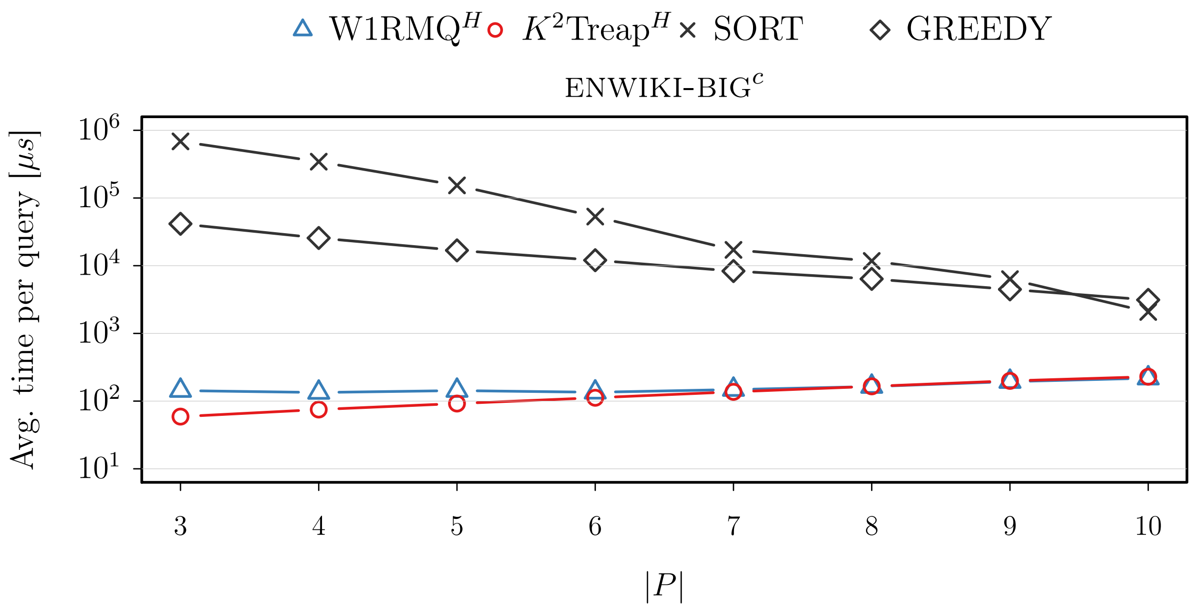


Figure 9.12: Average time per query in microseconds for different pattern lengths and fixed $k = 10$ value for ENWIKI-BIG^c.

point of each subgrid inside the bigger query area, and thus it might find results sooner.

Fig. 9.16 shows the comparison between our indexes and the baselines (GREEDY and SORT). For the small Wikipedia collection our indexes are one order of magnitude faster than GREEDY for all values up to $k \leq 64$, taking less than $100\mu s$ on average. On the other hand, the naive SORT is up to 1,000 times slower than our indexes for small $k \leq 8$ values, and 10 times slower for $k = 256$. For the bigger texts, SORT is not considered since it required more than 5 seconds to execute. In these cases, our indexes are undisputedly the fastest alternative, being about 1,000 times faster for $k = 10$ and almost 100 times faster for the largest k value. Note that from the two alternatives, K^2Treap^H is faster than W1RMQ^H as the grid search is considerably faster when performed on the K^2-treap than on the wavelet tree.

9.4 Discussion

In this part we have proposed a more efficient and conceptually simpler implementation of the optimal solution of Navarro and Nekrich [119]. We have sharply engineered and compressed the optimal solution, while retaining the best ideas that led to its optimal time. Our first index, WRMQ, presents a first naive approach of the optimal implementation, while the indexes K^2Treap^H and W1RMQ^{H}shows remarkable improvement over this naive approach, in terms of space and time. We also present an efficient construction method, which we carefully engineered so that that it is possible to handle large IR collections. We also make use of state-of-the-art compact data structures implementations, so that our indexes are able to support large alphabets.

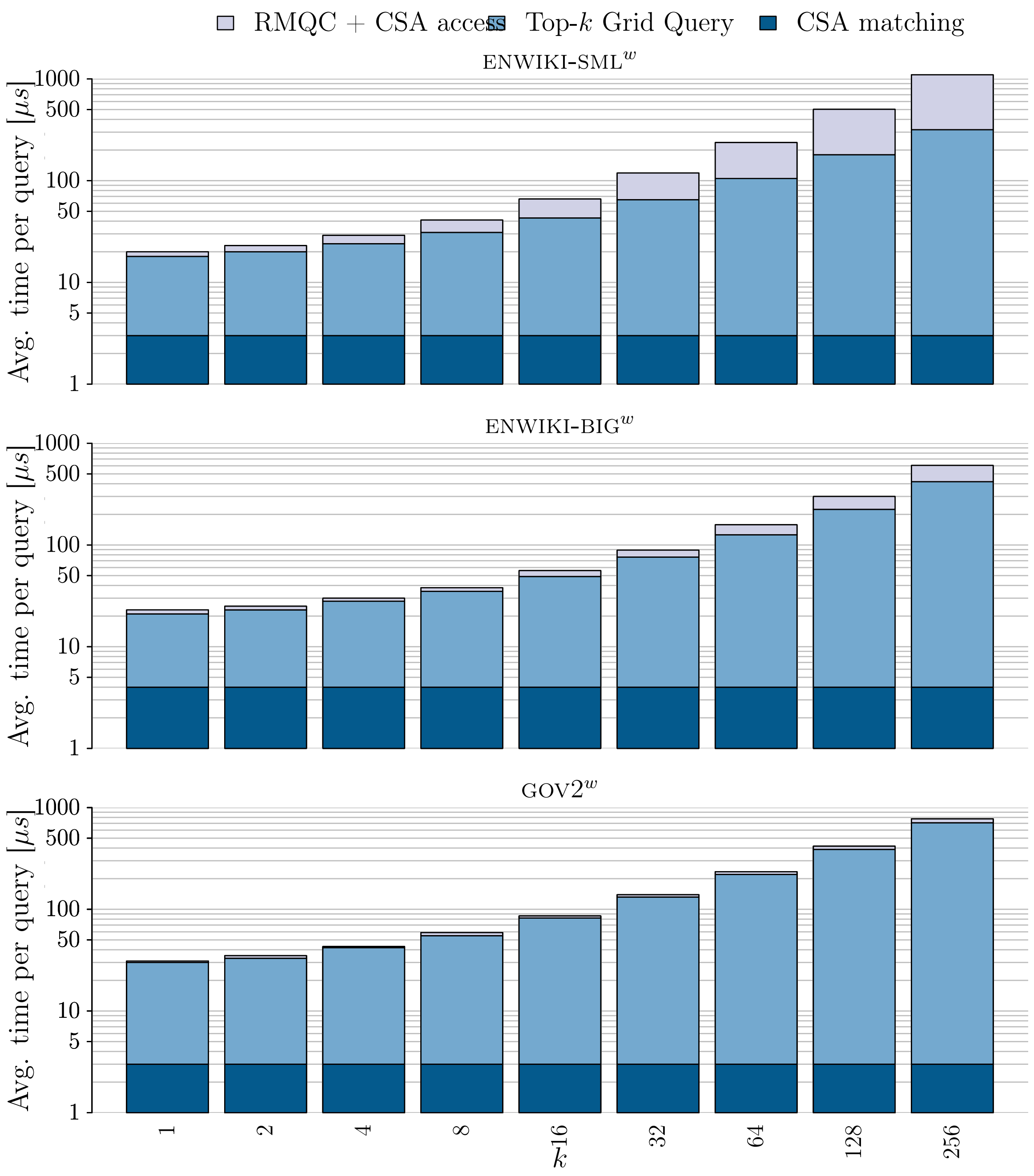


Figure 9.13: Detailed breakdown of average query time for index $K^2\text{Treap}^H$ on word alphabet collections.

A relevant open problem is to reduce the space further while retaining competitive query times; as shown in this work, current existing approaches that uses less space are hundreds of times slower. Another interesting direction is to compete with inverted indexes in weighted Boolean queries, which are considerably harder than phrase queries for our indexes. Finally, we plan to adapt our indexes and extend them to support more complex relevance measures, such as BM25.

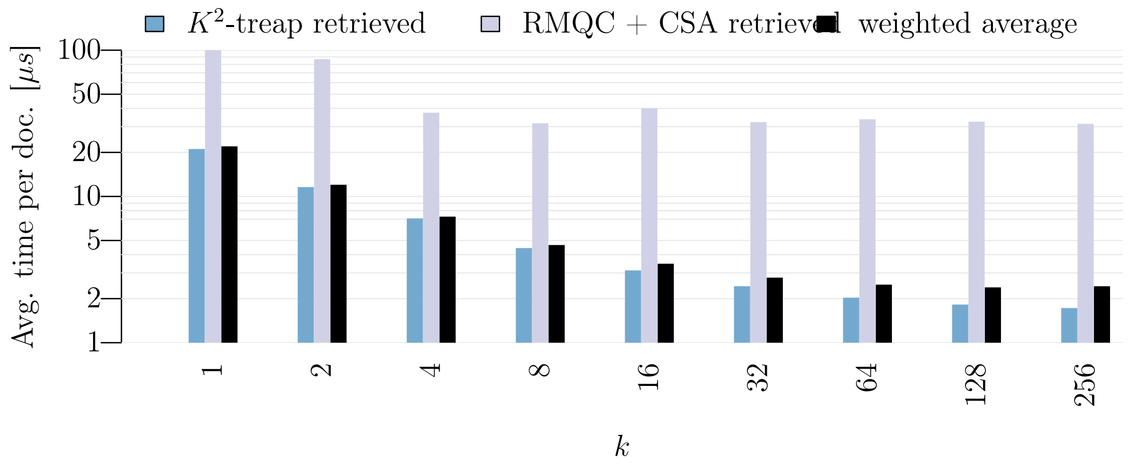


Figure 9.14: Query times for IDX_GN on GOV2^w, with $m = 1$. Left: Query time depending on k with a detailed breakdown of the three query phases. Right: Average time per document (mixed), per K^2-treap retrieved document and RMQC+CSA retrieved document. CSA matching time is included in all cases.

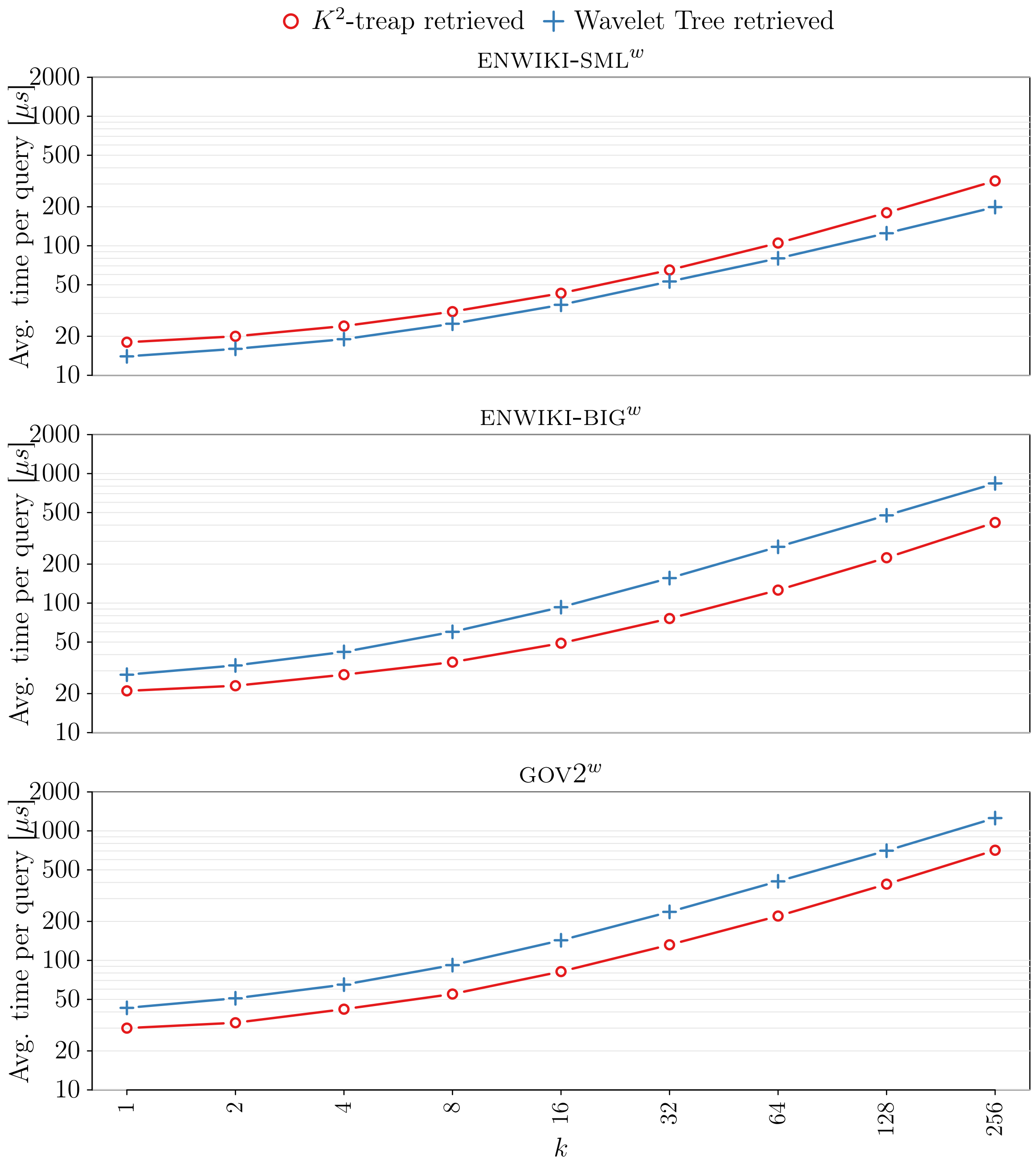


Figure 9.15: Comparison of the average time per query to retrieve the top-k results using a K^2-treap or the wavelet tree for representing G in word alphabet collections.

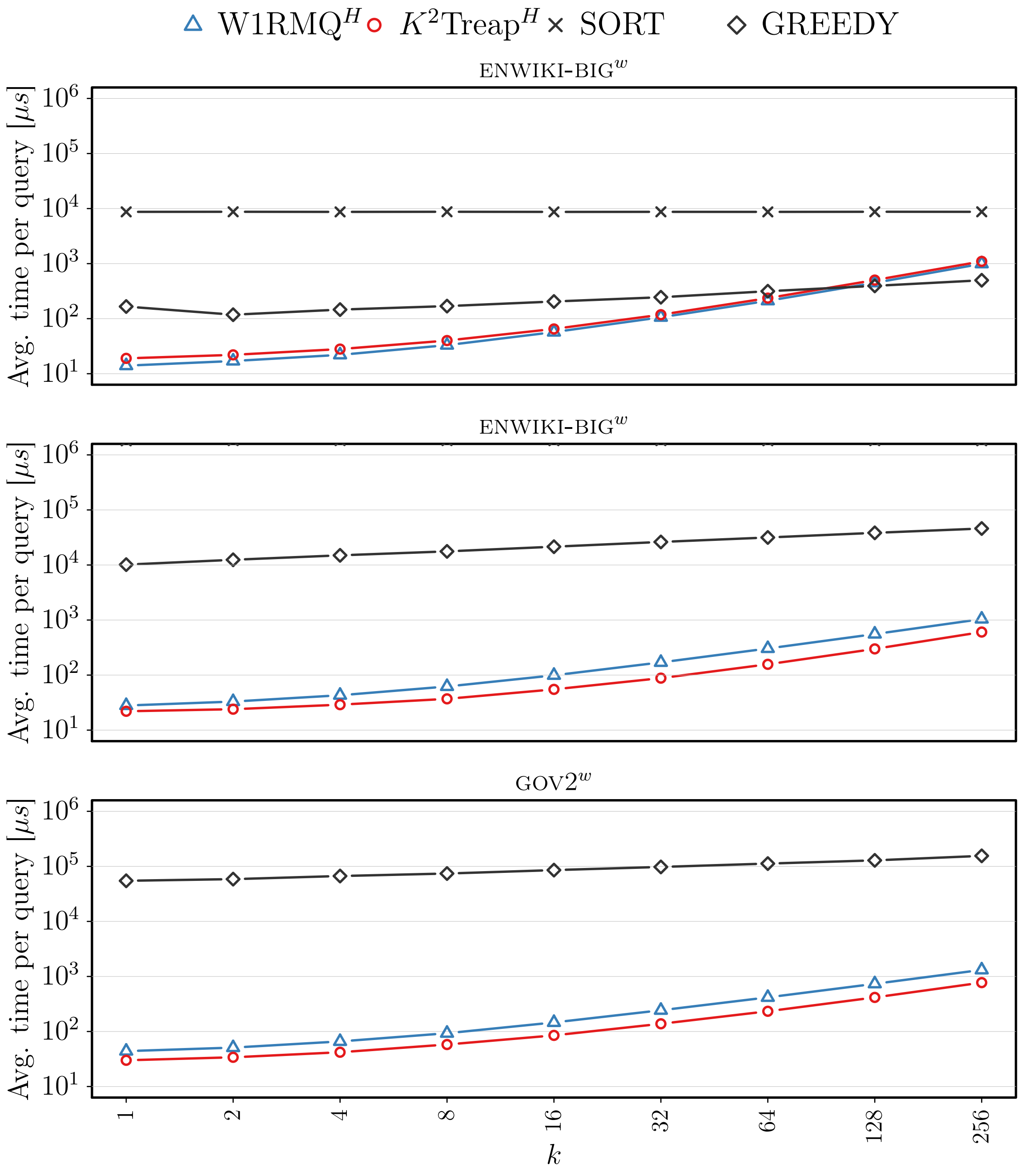


Figure 9.16: Average time per query in microseconds for different k values and fixed pattern length $m = 1$ on word alphabet collections. Results that required more than 5 seconds for SORT are not shown.

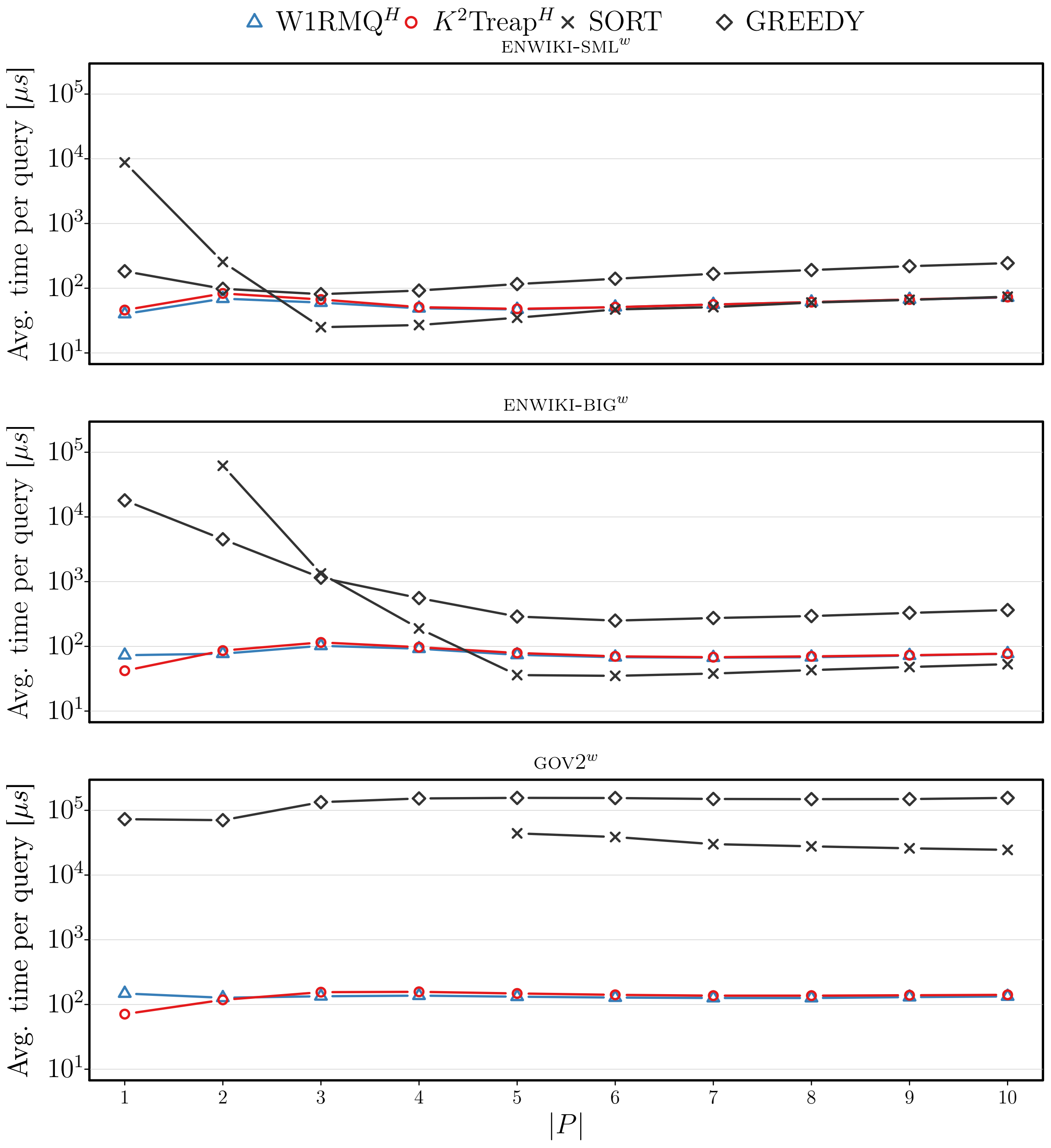


Figure 9.17: Average time per query in microseconds for different pattern lengths. Results that required more than 5 seconds for SORT are not shown.

Part IV

Extensions and Evangelization

Chapter 10

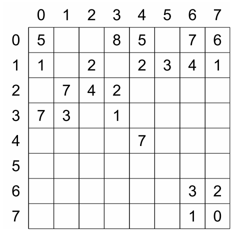
Aggregated 2D Queries on Clustered Points

In this chapter we present a generalization of the technique that was used in the design of the K^2-treap data structure (see Section 7.2) to support other types of two-dimensional aggregated queries. We show how to extend the K^2-tree to solve the problem of efficiently retrieving the number of points that lie in a two-dimensional range $[x_1, x_2] \times [y_1, y_2]$. Our experimental evaluation shows that by using little space overhead (less than a 30%) on top of the K^2-tree, the modified version of the data structure performs queries several orders of magnitude faster than the original K^2-tree, especially when the query ranges are large.

The K^2-treap data structure appears in the proceedings of the 21st String Processing and Information Retrieval conference (SPIRE), 2014, held in Ouro Preto, Brazil. The generalization of the K^2-treap as a technique that employs the K^2-tree to perform aggregated two-dimensional queries over clustered set of points has been published in the journal Information Systems, in March of 2016. This work was developed in collaboration with Nieves R. Brisaboa, Guillermo de Bernardo from University of A. Coruña, Spain and Diego Seco, from University of Concepción.

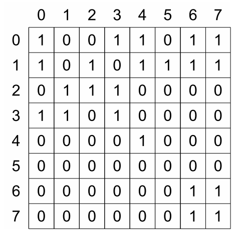
10.1 Motivation

Many problems in different domains can be interpreted geometrically by modeling data records as multidimensional points and transforming queries about the original records into queries on the point sets [26, Chapter 5]. In 2D, for example, orthogonal range queries on a grid can be used to solve queries of the form *"report all employees born between y_0 and y_1 who earn between s_1 and s_2 dollars"*, which are very common in databases. By the same token, other range aggregated queries (e.g., top-k, counting, quantile, majority, etc.) have proven to be useful for analyses about the data in various domains, such as OLAP databases, Geographic Information Systems, Information Retrieval, Data Mining, among others [120]. For example, top-k queries on a OLAP database of sales can be used to find the sellers with



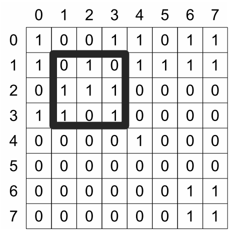
	0	1	2	3	4	5	6	7
0	5			8	5		7	6
1	1		2		2	3	4	1
2		7	4	2				
3	7	3		1				
4					7			
5								
6							3	2
7							1	0

Figure 10.1: Weighted matrix.



	0	1	2	3	4	5	6	7
0	1	0	0	1	1	0	1	1
1	1	0	1	0	1	1	1	1
2	0	1	1	1	0	0	0	0
3	1	1	0	1	0	0	0	0
4	0	0	0	0	1	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	1
7	0	0	0	0	0	0	1	1

Figure 10.2: Binary matrix.



	0	1	2	3	4	5	6	7
0	1	0	0	1	1	0	1	1
1	1	0	1	0	1	1	1	1
2	0	1	1	1	0	0	0	0
3	1	1	0	1	0	0	0	0
4	0	0	0	0	1	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	1
7	0	0	0	0	0	0	1	1

Figure 10.3: Range query.

most sales in a time period. In this example, the two dimensions of the grid are sellers ids (arranged hierarchically in order to allow queries for sellers, stores, areas, etc.) and time (in periods of hours, days, weeks, etc.), and the weights associated to the data points are the number of sales made by a seller during a time slice. Thus, the query asks for the k heaviest points in some range $Q = [i_1, i_2] \times [t_1, t_2]$ of the grid.

This approach of modeling problems using a geometrical formulation is well-known and there are many classical representations that support the queries required by the model and solve them efficiently. Range trees [25] and *kd*-trees [24] are two paradigmatic examples. Some of these classical data structures are even optimal both in query time and space. However, such classical representations usually do not take advantage of the distribution of the data in order to reduce the space requirements. When dealing with massive data, which is the case of some of the aforementioned data mining applications, the use of space-efficient data structures can make the difference between maintaining the data in main memory or having to resort to (orders-of-magnitude-slower) external memory.

In this work we consider the case where we have clustered points in a 2D grid, which is a common scenario in domains such as Geographic Information Systems, Web graphs, social networks, etc. There are some well-known principles such as Tobler's first law of geography [156], which states that *near things are more related than distant things*, or the locality of reference in the temporal dimension, that induce clusters in datasets where such dimensions are considered (either explicitly or implicitly). This is also the case in other not so well-known domains; for example, in the Web graph [30] these clusters appear when the Web pages are sorted by URL. Hence, we take advantage of these clusters in order to reduce the space of the data structures for aggregated 2D queries.

The K^2-tree (see Section 2.9) is a good data structure to solve range queries on clustered points and it has been extensively evaluated in different domains [144, 54, 41]. We extend this data structure in order to answer aggregated range queries, using a general technique, and we show how to adapt this general technique to support range counting queries within a 2D range, and evaluate its performance for a variety of domains.

We consider two dimensional grids with n columns and m rows, where each cell a_{ij} can either be empty or contain a weight in the range $[0, d-1]$ (see Figure. 10.1). For some problems, we will omit the weights and just consider the non-empty cells, which can be represented as a binary matrix (Figure. 10.2) in which each cell contains a 1 (if there is a weighted point in the original matrix) or a 0 (in other case).

Let t be the number of 1s in the binary matrix (i.e., the number of weighted points). If we can partition the t points into c clusters, not necessarily disjoint and $c << t$, we will say the points are clustered. This definition is used in Gagie et al. [73] to show that a quadtree in such case has $O(c \log u + \sum_i t_i \log l_i)$, where $u = max(n, m)$, and t_i and l_i are the number of points and the diameter of cluster i, respectively.

A range query $Q = [x_1, x_2] \times [y_1, y_2]$ defines a query rectangle with all the columns in the range $[x_1, x_2]$ and the rows in $[y_1, y_2]$ (see Fig. 10.3). An aggregated range query defines, in addition to the range, an aggregate function that must be applied to the data points in the range query. Examples of aggregated queries are $\text{COUNT}(Q)$, which counts the number of data points in the range query, $\text{MAX}/\text{MIN}(Q)$, which computes the maximum (alt. minimum) value in the query range, and its generalization top-k, which retrieves the k lightest (alt. heaviest) points in the query range. These top-k queries are also referred in the literature as ranked range queries. For the range query q in Fig. 10.3 the result of $\text{COUNT}(q)$ is 6, $\text{MAX}(q)$ returns 7, $\text{MIN}(q)$ returns 1, and the top-3 heaviest elements are 7, 4 and 3.

There are other interesting data-analysis queries on two-dimensional grids. For example, $\text{QUANTILE}(Q, k)$, which returns the k-th smallest value in Q, or $\text{MAJORITY}(Q, \alpha)$, which retrieves those values in Q that appear with relative frequency larger than α. These and more have been studied in [120], where the authors propose space-efficient data structures with good time performance. We restrict ourselves to an emblematic subset of these queries, and propose data structures that are even more space-efficient when the points in the set are clustered.

10.2 Augmenting the K^2-tree

Let $M[n \times n]$ be a matrix in which cells can be empty or contain weight in the range $[0, d-1]$ and let $BM[n \times n]$ be a binary matrix in which each cell contains a zero or a one. We say that BM^M represents the topology of M if $BM^M[i][j] = 1$ when $M[i][j]$ is not empty, and $BM^M[i][j] = 0$, otherwise.

On the one hand, we store the topology of the data. On the other hand, we store the weights associated with each non-empty cell. For the topology, which can be regarded as a binary matrix, we use a K^2-tree. Recall from Section 2.9 that this data structure requires little space when the data points are clustered. The K^2-tree performs a recursive division of the space into K^2 submatrices until it reaches an individual cell or a completely empty submatrix, and each of these submatrices is represented in the data-structure with a bit. Therefore, each bit in the K^2-tree represents a submatrix of the original matrix which size depends on its depth on the recursive division.

A level-wise traversal of the K^2-tree can be used to map each node to an aggregated value that summarizes the weights in the corresponding submatrix. The specific value of this summary depends on the operation. For example, for ranked range queries (see Section 7.2) the summary represents the maximum weight in the corresponding submatrix, whereas for counting queries, it represents the number of non-empty cells in the submatrix. However, a common property is that the value associated with a node aggregates information about its K^2 children. Therefore, we use a sort of differential encoding [48] to encode the values of each node with respect to the value of its parent. In other words, the information of a node (such as its summary and number of children) is used to represent the information of its children in a more compact way. In order to access the original (non-compressed) information of a node we need to access its parent first (i.e., the operations in this technique are restricted to root-to-leaves traversals).

To summarize, we use a K^2-tree to represent the topology of the data and augment each node of such a tree with additional values that represents the aggregate information related with the operation to be supported. These aggregated values are differentially encoded with respect to information of the parent node in order to reduce the space necessary to store them. In the following two sections we show how both the K^2-tree and the differential encoded values can be tuned to efficiently solve two types of aggregated queries.

Finally, note that we present our results for matrices of size $n \times n$. This is not a loss of generality, as we can extend a matrix $M'[n \times m]$ with zeros to complete a square matrix $M[n \times n]$ (w.l.o.g. we assume $m \leq n$). As the topology of the matrix is represented with a K^2-tree, this does not cause a significant overhead because the K^2-tree is efficient to handle large areas of zeros. Actually, we round n up to the next power of K [35].

10.2.1 Supporting Counting Queries

In addition to the K^2-tree, the augmented data structure stores additional information to support range counting queries efficiently. In Figure 10.4 we show a conceptual K^2-tree in which each node has been annotated with the number of elements in its corresponding submatrix. Note that this is the same example of Figure 7.2 considering as non-empty cells those with weight larger than 0.

This conceptual tree is traversed level-wise, reading the sequence of counts from left to right in each level. All these counts are stored in a sequence *counts* using a variant of the differential encoding technique presented in Section 10.2. Let v be a node of the K^2-tree, $children(v)$ the number of children of v, and $count(v)$ the number of elements in the submatrix represented by v. Then, $\overline{count(v')} = \frac{count(v)}{children(v)}$ represents the expected number of elements in the submatrix associated with each child v' of v, assuming an uniform distribution. Thus, the count of each node v' is stored as the difference of the actual count and its expected value. In the running example, the root has three children and there are 22 elements in the matrix. Each of the corresponding submatrices is expected to contain $\lfloor 22/3 \rfloor = 7$ elements whereas they actually contain 10, 7 and 5 elements, respectively. Hence, the differential enconding stores $10 - 7 = 3$, $7 - 7 = 0$, and $5 - 7 = -2$.

The result of this differential encoding is a new sequence of values smaller than the original, but which may contain negative values. In order to map this sequence, in a unique and reversible way, to a sequence of non-negative values we use the folklore *overlap and interleave* scheme, which maps a negative number $-i$ to the i^{th}-odd number $(2i - 1)$ and a positive number j to the j^{th} even number $(2j)$. Finally, to exploit the small values while allowing

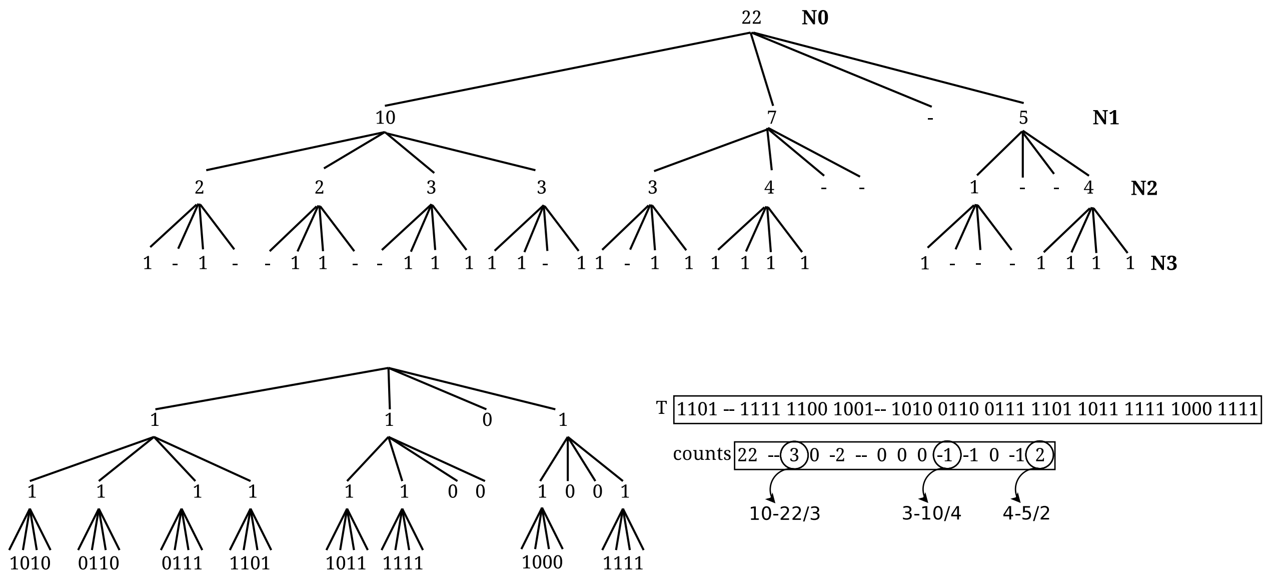

22 N0
10 7 - 5 N1
2 2 3 3 3 4 - - 1 - - 4 N2
1 - 1 - - 1 1 - - 1 1 1 1 1 - 1 1 - 1 1 1 1 1 1 1 - - - 1 1 1 1 N3
1 1 0 1
1 1 1 1 1 1 0 0 1 0 0 1
1010 0110 0111 1101 1011 1111 1000 1111
T 1101 -- 1111 1100 1001-- 1010 0110 0111 1101 1011 1111 1000 1111
counts 22 -- 3 0 -2 -- 0 0 0 -1 -1 0 -1 2
10-22/3
3-10/4
4-5/2

Figure 10.4: Storage of the conceptual tree in our data structures.

efficient direct access to the sequence, we represent *counts* with Direct Access Codes [34].

As *counts* corresponds with a level-wise traversal of the K^2-tree, it is not necessary to store this additional information for all the levels of the tree. In this way, we provide a parametrized implementation that sets the space overhead by defining the number of levels for which counting information is stored. This provides a space-time trade-off and, in the Experimental Evaluation, we study the influence of this parameter.

Range Counting Queries

The base of the range counting algorithm is the range reporting algorithm of the K^2-tree shown in Algorithm 3. We modify this divide-and-conquer algorithm in order to navigate both the K^2-tree and *counts* at the same time. Given a query range $Q = [x_1, x_2] \times [y_1, y_2]$, we start accessing the K^2-tree root, and set $c_0 = counts[0]$ and $result = 0$. Then, the algorithm explores all the children of the root that intersect Q and decodes the counting value of the node from the *counts* array and the absolute counting value of the root. The process continues recursively for each children until it finds a completely empty submatrix, in which case we do not increment *result*, or a matrix completely contained inside Q, in which case we increment result with the counting value of such matrix.

To clarify the procedure, let us introduce some notation and basic functions. We name the nodes with the position of their first bit in the bitmap T that represents the K^2-tree. Then, $v = 0$ is the root, $v = K^2$ is its first non-empty child, and so on. Recall that $(rank_1(T, v-1)+1) \cdot K^2$ gives the position in T where the children of v start and each of them is represented with K^2 bits. We can obtain the number of children of v as $NumChildren(v) = rank_1(T, v + K^2) - rank_1(T, v - 1)$. Non-empty nodes store their differential encoding in *counts* in level order, so we must be able to compute the level order of a node in constant time. Node v stores K^2 bits that indicate which children of v are non-empty. If the i^{th} bit is set to 1, then the level order of that child node is $rank_1(T, v+i)$, with $i \in [0, K^2-1]$.

Given a node v with absolute counting value c_v and $NumChildren(v)$ children, we expect each child v' to represent $\overline{count(v')} = \frac{c_v}{NumChildren(v)}$ elements. Let p be the level order number of child v'. Then, the absolute counting value of v' can be computed as $c_{v'} = counts[p] + \overline{count(v')}$. Note that *counts* is stored using DACs, which support efficient direct access. By updating $c_{v'}$ before visiting each node v', we can complete the recursive procedure.

Let us consider a query example $q = [0,1] \times [0,2]$. We start at the root with $c_0 = 22$ and $result = 0$. The root has three children, but only the first one, stored at position 4 in T, intersects q. Each child is expected to represent $\lfloor 22/3 \rfloor = 7$ elements, so we set $c_4 = counts[rank_1(T, 0+0)] + 7 = 3 + 7 = 10$. Similarly, we recurse on the first and third children of this node. On the first branch of the recursion, we process node 16 and set $c_{16} = counts[rank_1(T, 4+0)] + \lfloor 10/4 \rfloor = counts[4] + 2 = 2$. As the submatrix corresponding with this node is contained inside q, we add 2 to the result and stop the recursion on this branch. On the other one, we have to recurse until the leaves to sum the other element to the result, and obtain the final count of three.

Other Supported Queries

This data structure obviously supports all the queries that can be implemented on a K^2-tree, such as range reporting or emptiness queries. More interesting is that it can also support other types of queries with minor modifications. A natural generalization of range counting queries are aggregated sum queries. In this case, we consider a matrix $M[n \times n]$ where each cell can either be empty or contain a weight in the range $[0, d-1]$. We perform the same data partitioning on the conceptual binary matrix that represents the non-empty cells. In other words, we use a K^2-tree to represent the topology of the matrix. Then, instead of augmenting the nodes with the count of the non-empty cells in the corresponding submatrix, we store the sum of the weights contained in such submatrix. The same differential encoding used for range counting can be used to store these sums. In this case, however, the space-efficiency achieved by the data structure depends not only on the clustering of the data, but also on the distribution of the weights. The encoding achieves its best performance when the sums of the weights of all the children of a node are similar.

10.3 Experiments and Results

In this section we present the conducted empirical evaluation of range counting queries. We first present the experiment setup (baselines and datasets), then an evaluation in terms of the space required by the different solutions, and finally a comparison of the running times of such solutions.

We ran all our experiments on a dedicated server with 4 Intel(R) Xeon(R) E5520 CPU cores at 2.27GHz 8MB cache and 72GB of RAM memory. The machine runs Ubuntu GNU/Linux version 9.10 with kernel 2.6.31-19-server (64 bits) and gcc 4.4.1. All the data structures were implemented in C/C++ and compiled with full optimizations.

All bitmaps that are employed use a bitmap representation that supports RANK and SELECT using 5% of extra space. The wavelet tree that was employed to implement the solution of [120] is a pointer less version obtained from `LIBCDS`[1]. This wavelet tree is augmented with a RMQ data structure (recall Section 2.12 at each level.

10.3.1 Experiment Setup

We use grid datasets coming from three different real domains: Geographic Information Systems (GIS), Social Networks (SN) and Web Graphs (WEB). For GIS data we use the Geonames dataset[2], which contains the geographic coordinates (latitude and longitude) of more than 6 million populated places, and converted it into three grids with different resolutions: Geo-sparse, Geo-med, and Geo-dense. The higher the resolution, the sparser the matrix. These datasets allow for evaluating the influence of data sparsity in the different proposals. For SN we use three social networks (dblp-2011, enwiki-2013 and ljournal-2008) obtained from the Laboratory for Web Algorithmics[3] [31, 28]. Finally, in the WEB domain we consider the grid associated with the adjacency matrix of three Web graphs (indochina-2004, uk-2002 and uk-2007-5) obtained from the same Web site. The existence of clusters in these datasets is very dissimilar. In general, GIS datasets do not preset many clusters whereas data points in the WEB datasets are very clustered. SN represent an intermediate collection in terms of clustering.

In this experiment, we compare our proposal to speed up range counting queries on a K^2-tree, named *rck2tree*, with the original K^2-tree. In this evaluation we show the space-time trade-off offered by the *rck2tree* structure, which is parametrized by the number of levels of the original K^2-tree augmented with counting information. As a baseline from the succinct data structures area, we include a representation of grids based on wavelet trees (recall Section 7.1), named *wtgrid* in the following discussion. As explained in Section 2.7.4, this representation requires a bitmap to map from a general grid to a grid with at most one point per column. In our experiments we store this bitmap with either plain bitmaps or `RRR`, the one that requires less space. As for the wavelet tree itself, we use a balanced tree with just one pointer per level and the bitmaps of each level are compressed with `RRR`. In other words, we use a configuration of this representation that aims to reduce the space. Note, however, that we use the implementation available in LIBCDS [46], which uses $O(\sigma)$ counters to speed up queries. As we will show in the experiments, this data structure, even with the compression of the bitmaps that represent the nodes of the wavelet tree, does not take full advantage of the existence of clusters in the data points.

Table 10.1 shows the main characteristics of the datasets used: name of the dataset, size of the grid (u)[4], number of points it contains (n) and the space achieved by the baseline *wtgrid*,

[1]http://www.github.com/fclaude/libcds/

[2]http://www.geonames.org/

[3]http://law.di.unimi.it

[4]Note that, unlike the grids used in the previous scenario, these are square grids, and thus u represents both the number of rows and columns.

Table 10.1: Real datasets used, and space required to represent them.

Dataset	Type	Grid (u)	Points (n)	wtgrid (bpp)	K^2-tree (bpp)	*rck2tree*^{4} (bpp)	*rck2tree*^{8} (bpp)	*rck2tree*^{16} (bpp)
Geo-dense	GIS	524,288	6,049,875	17.736	14.084	14.085	14.356	18.138
Geo-med	GIS	4,194,304	6,080,640	26.588	26.545	26.564	29.276	36.875
Geo-sparse	GIS	67,108,864	6,081,520	44.019	41.619	41.997	48.802	56.979
dblp-2011	SN	986,324	6,707,236	19.797	9.839	9.844	10.935	13.124
enwiki-2013	SN	4,206,785	101,355,853	19.031	14.664	14.673	16.016	19.818
ljournal-2008	SN	5,363,260	79,023,142	20.126	13.658	13.673	15.011	18.076
indochina-2004	WEB	7,414,866	194,109,311	14.747	1.725	1.729	1.770	2.13
uk-2002	WEB	18,520,486	298,113,762	16.447	2.779	2.797	2.888	3.451
uk-2007-5	WEB	105,896,555	3,738,733,648	16.005	1.483	1.488	1.547	1.919

by the original K^2-tree and by different configurations of our *rck2tree* proposal. Unlike the previous scenario, the space is measured in bits per point because these matrices are very sparse, which results in very low values of bits per cell.

10.3.2 Space Comparison

As expected, the representation based on wavelet trees, *wtgrid*, is not competitive in terms of space, specially when the points in the grid are clustered. Even though the nodes of the wavelet tree are compressed with `RRR`, this representation is not able to capture the regularities induced by the clusters. In the WEB domain, where the points are very clustered, the *wtgrid* representation requires up to 10 times the space of the K^2-tree. Therefore, the latter allows for the processing in main memory of much larger datasets. In domains where the data points are not that clustered, space-savings are still significant but not as outstanding as in the previous case.

Second, we analyze the space overhead offered by the *rck2tree* in comparison with the original K^2-tree. As mentioned above, this overhead depends on the number of levels of the K^2-tree that are augmented with additional range counting information. In these experiments, we show the results of three configurations in which 4, 8 and 16 levels were augmented, respectively. As Table 10.1 shows, the space overhead is almost negligible for the *rck2tree*^{4} and it ranges between 25% and 40% for *rck2tree*^{16}. In the next section we will show the influence of this extra space in the performance of the data structure to solve range counting queries.

It is interesting to notice that the space overhead is lower in the domains in which the K^2-performs the best. The K^2-tree performs better in sparse domains where data points are clustered [35], for example, in the Web graph. In the largest WEB dataset, uk-2007-5, the K^2-tree requires about 1.5 bits per point, and we can augment the whole tree with range counting information using less than 0.5 extra bits per point (this is an overhead of less than 30%). Sparse and clustered matrices result in less values being stored in the augmented data structure (as in the original K^2-tree, we do not need to store information for submatrices full of zeros). In Figure 10.5 we show the space overhead in two of our dataset collections, GIS and WEB.

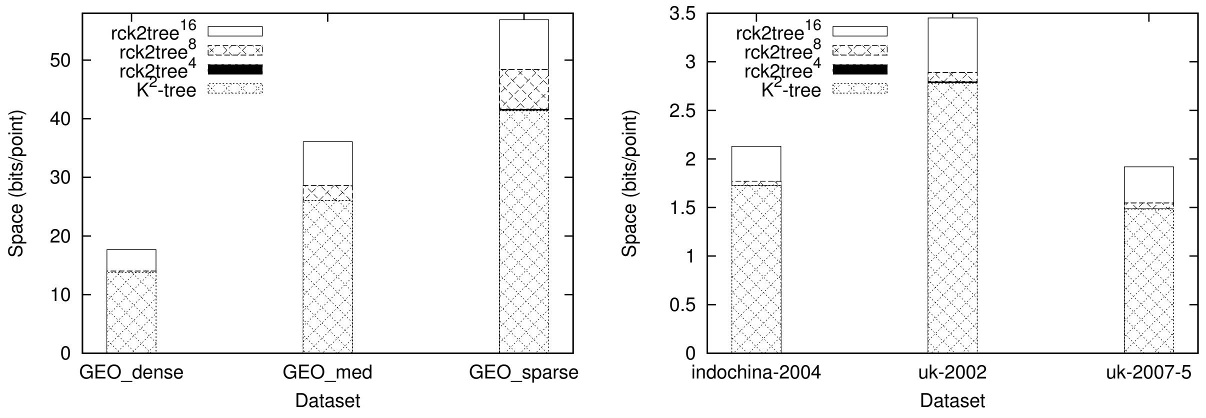


Figure 10.5: Space overhead in GIS and WEB datasets.

Note also that, unlike the K^2-tree, the space overhead is not drastically increased in sparse, but not clustered, datasets (for example, Geo-sparse). Isolated points are represented in the original K^2-tree with K^2 bits per level, which results in a strong degradation of such representation. The K^2-tree uses more than 40 bits per point in the Geo-sparse dataset, which is a lot more than the about 2 bits per point in the, also sparse, WEB datasets. However, the space overhead of the $rck2tree^{16}$ (with respect to the K^2-tree) is about 40% in Geo-sparse and 30% in uk-2007-5 (i.e., a difference of 10 percentage points). Isolated points are represented with approximately two bits per level in the range counting data structure, as it represents the difference between the expected number of elements in the submatrix, one, and the actual number, which is also one[5]. If we consider the other configurations that store range counting values for less levels of the K^2-tree the difference is even smaller. This is expected because there are less isolated points in higher levels of the K^2-tree.

10.3.3 Query Processing

In this section we analyze the efficiency of range counting queries, comparing our augmented K^2-tree with the original data structure and with the *wtgrid*. For each dataset, we build multiple sets of queries with different query selectivities (i.e., size of spatial ranges). All the query sets were generated for fixed query selectivity. A query selectivity of $X\%$ means that each query in the set covers $X\%$ of the cells in the dataset. Each query set contains 1,000 queries where the spatial window is placed at a random position within the matrix.

As in the space comparison, we show the results of three different configurations of our augmented data structure, in which 4, 8 and 16 levels were augmented with range counting data. Fig. 10.4 shows the time required to perform range counting queries in some of our real datasets, for different values of query selectivity. For each domain (GIS, SN and WEB), we only show the results of the two most different datasets, as the other datasets do not change the conclusions of this comparison.

[5]Recall that these data are represented using DACs, which require at least two bits, one two encode the difference and one two indicate that such value is encoded using just one block.

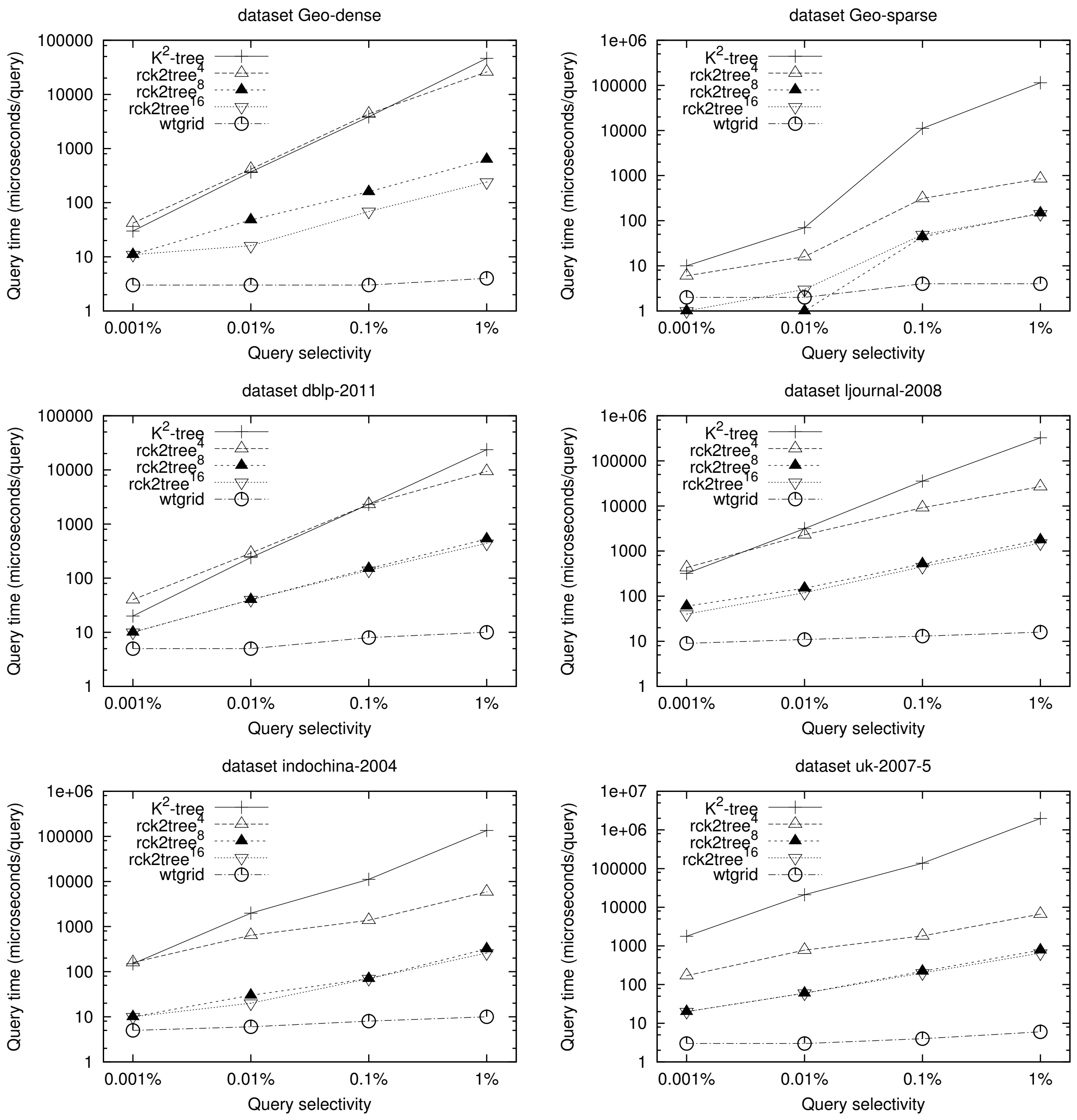


Figure 10.6: Query times of range counting queries in real datasets.

The augmented data structure consistently outperforms the original K^2-tree for all domains and query selectivities. The only exception is the $rck2tree^4$ in the two SN datasets and Geo-dense for very selective queries (i.e., for the smallest sets of queries). The influence of the query selectivity in the results is evident. The larger the query, the higher the impact of the additional range counting data in the performance of the data structure. Larger queries are expected to stop the recursion of the range counting algorithm in higher levels of the K^2-tree because those queries are more likely to contain the whole area covered by nodes of the K^2-tree. Recall that a node of the K^2-tree at level i represents $u^2/(K^2)^i$ cells. In our experiments we use a configuration of the K^2-tree in which the first six levels use a value of $K_1 = 4$ (thus, partitioning the space in $K_1^2 = 16$ regions) and the remaining levels use a value of $K_2 = 2$. Therefore, for the *rck2tree*^{4} to improve the performance of the range counting queries, those queries must contain at least $u^2/(4^2)^4 = u^2/2^{16}$ cells. In the *rck2tree*^{8} and *rck2tree*^{16}, which contain range counting data for more levels of the K^2-tree, this value is much lower and thus, even small queries are likely to stop the recursion of the query algorithm. For larger queries, the improvement in the performance reaches several orders of magnitude in some datasets (note the logarithmic scale).

In most datasets, the performance of *rck2tree*^{8} and *rck2tree*^{16} is very similar, therefore the former is preferable as it requires less extra space. Hence, in general, we can recommend the use of the *rck2tree*^{8} configuration to speed up range counting queries. If the queries are not very selective, and there are strong memory consumption constraints, the *rck2tree*^{4} can be an alternative as it requires almost the same space of the original K^2-tree.

The *wtgrid* is consistently not only faster than the data structures based on the K^2-tree, but also less sensitive to the size of the query. However, as we showed above, it also requires more space. To reinforce these conclusions, Figure 10.7 shows the space-time trade-offs of the different configurations. We selected two representative datasets that were not used in the previous experiment, enwiki-2013 (SN) and uk-2002 (WEB), and two query selectivities for each of them, 0.001% and 0.1%. Each point in the lines named *rck2tree* represents a different configuration with 4, 8 and 16 levels augmented with range counting information, respectively from left to right.

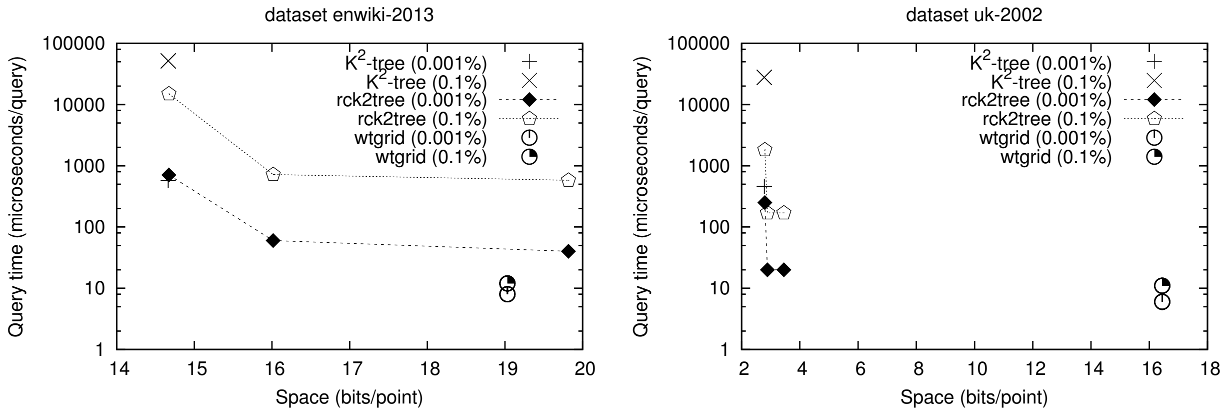


Figure 10.7: Space-time trade-offs of the compared range counting variants.

These graphs show that the *rck2tree*^{8} configuration offers the most interesting trade-off as it requires just a bit more space than the original K^2-tree and it speeds up range counting queries significantly. This effect is more evident for larger queries, but even for very selective queries the improvement is significant. The *wtgrid* competes in a completely different area of the trade-off, being the fastest data structure, but also the one that requires more space.

10.4 Discussion

We have introduced a technique to solve aggregated 2D range queries on grids, which requires little space when the data points in the grid are clustered. We use a K^2-tree to represent the topology of the data (i.e., the positions of the data points) and augment each node of the tree with additional aggregated information that depends on the operation to be supported. The aggregated information in each node is differentially encoded with respect to its parent in order to reduce the space overhead.

Our experimental evaluation shows that with a little space overhead (less than a 30%) on top of the K^2-tree, this data structure performs queries several orders of magnitude faster than the original K^2-tree, specially when the query ranges are large. These results are consistent in the different databases tested, which included domains with different levels of clustering in the data points. For example, in Web graphs the data points are very clustered, which is not the case in GIS applications. We also compared our proposal with a wavelet tree-based solution and the experiments show that, although the wavelet tree is faster, our proposal requires less space (up to 10 times less space when the points are clustered). Thus, we provide a new point in the space-time trade-off which allows for the processing of much larger datasets.

Although we have presented the two types of queries separately, this does not mean that an independent data structure would be required for each type of aggregated query. The topology of the data can be represented by a unique K^2-tree and each type of aggregated query just adds additional aggregated and differentially encoded information. However, some specific optimizations on the K^2-tree, such as the one presented for ranked range queries, may not be possible for all types of queries.

The technique can be generalized to represent grids in higher dimensions, by replacing our underlying K^2-tree with its generalization to d dimensions, the K^d-tree [55] (not to be confused with kd-trees [24]). The algorithms stay identical, but an empirical evaluation is left for future work. In the worst case, a grid of t points on $[n]^d$ will require $O(t \log \frac{n^d}{t})$ bits, which is of the same order of the data, and much less space will be used on clustered data. Instead, an extension of the wavelet tree will require $O(n \log^d n)$ bits, which quickly becomes impractical. Indeed, any structure able to report the points in a range in polylogarithmic time requires $\Omega(n(\log n/\log\log n)^{d-1})$ words of space [43], and with polylogarithmic space one needs time at least $\Omega(\log n(\log n/\log\log n)^{\lfloor d/2 \rfloor -2})$ [7]. As with top-k queries one can report all the points in a range, there is no hope to obtain good worst-case time and space bounds in high dimensions, and thus heuristics like K^d-treaps are the only practical approaches (kd-trees do offer linear space, but their time guarantee is rather loose, $O(n^{1-1/d})$ for n points on $[n]^d$).

Chapter 11

Web Access Logs

This chapter describes how to extend the ideas from top-k query processing (see Part III) to other scenarios that are of interest for IR. In this case, we explore the idea of using a compressed text representation (`SSA`) to represent the server logs that are formed when any user visits a website. In order to support IR operations, while being space-efficient, we design an index that makes use different compact data structures and document retrieval algorithms, and we adapt them to this particular scenario. The result is a space-efficient data structure that is able to process a set of operations that are of interest to web usage mining processes.

This work appears in the proceedings of 21st String Processing and Information Retrieval conference (SPIRE), held in Ouro Preto, Brazil, and was developed in collaboration with Francisco Claude from Universidad Diego Portales, Chile.

11.1 Motivation

Web Usage Mining (WUM) [90] is the process of extracting useful= information from web server access logs, which allows web site administrators, designers and engineers to understand the users' interaction with their web site. This process is used to improve the layout of the web site to better suit their users, or to analyze the performance of their systems in order to apply smart prefetching techniques for faster response, among other applications.

One particular WUM task is to predict the path of web pages that the user is going to traverse within a website. Accurately predicting the web user access behavior can minimize the user perception of latency, which is an important measure of the website quality of service [153, 61, 60]. This is achieved by fetching the web page *before* the user requests it. Another application is as a recommendation technique [154, 146]: the prediction can be displayed to the user, giving insight as to what the user might be looking for, therefore improving the user's experience. Other relevant mining operations include determining how frequently the path has been followed, which users have followed the path, and so on.

The prediction problem can be formalized as follows. Access logs obtained from web servers are used to extract the user's website visit path as an ordered sequence of web pages $S_u = \langle v_1, v_2, v_3, \ldots, v_m \rangle$ (several sessions of the same user u might be concatenated into S_u). Therefore the system records the set $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ of the accesses of each user. Given a new visit sequence (or path) P that is currently being performed by a user, the predicting task has to predict which page will be visited next. One näive approach to this problem is to first return the k web pages that have been visited most commonly by users after following the same path P, that is, we consider each time P appears as a substring of some S_u, and pick the most common symbols following those occurrences of P. After this process is done, more complex recommendation or machine learning algorithms [110] can be employed to accurately predict the next web page that is going to be visited. One particular challenge is that this operation needs to be done in an *on-line* manner, that is, we have to efficiently update our results as new requests are appended at the end of P. The system will eventually add those requests S' to the corresponding S_u sequences in $\mathcal{S}$, via periodic updates. At query time, the set $\mathcal{S}$ can be taken as static. The other mining operations are defined analogously.

Another interesting operation coming from WUM and general data mining is to retrieve the top-k most frequent sequences [83, 130] of a certain length. These are commonly used in retailing, add-on sales, customer satisfaction and in many other fields. On the other hand, least frequent sequences can be used to identify outliers or sets of suspicious activities on a site.

We present a space-efficient data structure for representing web access logs, based on the Burrows-Wheeler Transform (BWT) [38] (see Section 2.14.1). Our index is able to efficiently process various queries of interest, while representing the data in compressed form. In this work we focus on the following key operations; others are described in the Conclusions.

- ACCESS(u, i) : Access the i-th web page visited by user u.
- USERPATH(u) : Return the complete path done by user u.
- COUNT(P) : Count the number of times that path P has been done by any user.
- MOST_COMMON(P, k) : Return the k most common web pages visited after a path P.
- LIST_USERS(P, k) : Return k distinct users that have followed path P.
- MOST_FREQUENT(k, q) : Return the k most frequent paths of length q done by the users.

Our experimental results show that our index is able to represent the web access logs using 0.85–1.03 the size of their plain representations, thereby *replacing* them by a representation that uses about the same space but efficiently answers various queries, within microseconds in most cases. Our index can be easily deployed in other types of applications that handle ordered sequences, such as GPS trajectories, stock price series, customer buying history, and so on.

To our knowledge, this is the first compressed representation of logs that answers queries specific of WUM applications.

11.2 Our Index

In this section, we describe the main components of our index and how they are employed to solve the queries previously defined.

11.2.1 Construction

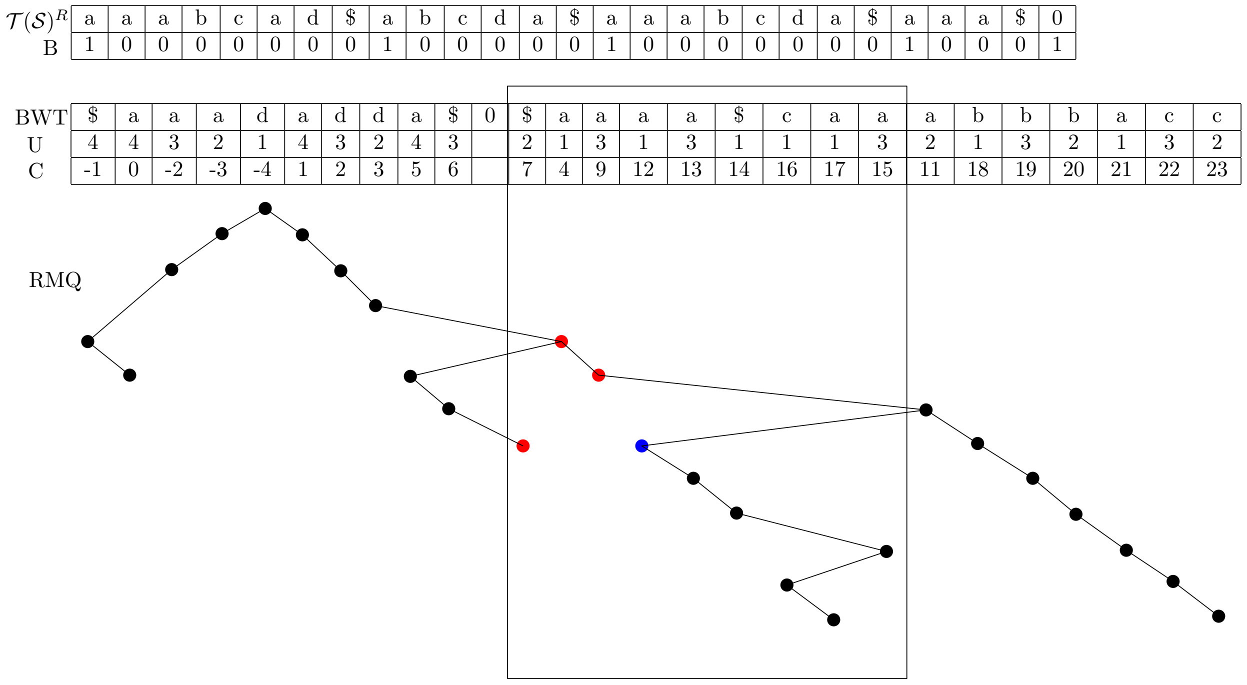
We start by concatenating all ordered web access sequences $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ from all users into a sequence $\mathcal{T}(\mathcal{S})$ of size $\sum_{i=1}^{n} |S_i| = N$ over an alphabet of size σ, where σ is the amount of distinct web pages visited by any user in the log file. Instead of building the BWT to index $\mathcal{T}(\mathcal{S})$, we construct the index over $\mathcal{T}(\mathcal{S})^R$, that is, $\mathcal{T}(\mathcal{S})$ has each S_i reversed. This simple trick allows us to maintain a range of elements that match the sequence of requests so far, while performing backward search (see Algorithm 4) , and thus allowing us to add new arriving requests by just performing one more step. We employ the `SSA` index to represent and replace $\mathcal{T}(\mathcal{S})^R$, however, having the `SSA` of $\mathcal{T}(\mathcal{S})^R$ is not enough to reconstruct the information from the log. We also need to store the user identifier associated with each position in the sequence. To do so without spending $N \log n$ bits to associate a user id to each position in the BWT of $\mathcal{T}(\mathcal{S})^R$, we construct a bitmap B of length N and mark with a 1 the positions where each S_i ends in $\mathcal{T}(\mathcal{S})^R$. We later index B to solve RANK and SELECT queries in constant time using compressed space [136]. This is enough to obtain the user associated to a location in the BWT of the sequence, by locating its original position p in the sequence and then performing $\text{RANK}_1(B, p)$.

For listing the distinct users (strings) where path P occurs, we implement Muthukrishnan's document listing algorithm (see Section 2.15 and Algorithm 6). We construct a temporary array $U[i] = \text{RANK}_1(B, i)$, for $1 \leq i \leq N$, that stores the user ids and then permute the values so that the ids are aligned to the $BWT(\mathcal{T}(\mathcal{S})^R)$ sequence. Another integer array C is constructed by setting $C[i] = \text{SELECT}_{U[i]}\left(U, \text{RANK}_{U[i]}(U, i) - 1\right)$[1] for all $0 \leq i \leq N$, and then build a RMQ data structure on C. We keep this structure and discard C and U.

The RMQ data structure requires $2N + o(N)$ bits. The `SSA` index requires $NH_k(\mathcal{T}(\mathcal{S})) + o(N \log \sigma)$ bits. The representation of bitmap B takes $H_0(B) + o(N) \leq N + o(N)$ bits. Note that we do not store the users, nor the frequencies in an explicit way.

We show in Figure 11.1 the main components of the index built over a set of example sequences $S_1 = dacbaaa, S_2 = adcba, S_3 = adcbaaa$ and $S_4 = aaa$. The dollar (\$) symbol is used to represent the end of each sequence and the zero symbol is used to mark the end of the concatenation of the sequences $\mathcal{T}(\mathcal{S})^R$.At the bottom we show the topology of the Cartesian tree built over the C array representing the RMQ data structure. Recall that arrays U and C are not represented and are only shown for guidance.

[1]To avoid corner cases, we define $\text{SELECT}_{U[i]}(U, 0) = -1$



$\mathcal{T}(\mathcal{S})^R$	a	a	a	b	c	a	d	\$	a	b	c	d	a	\$	a	a	a	b	c	d	a	\$	a	a	a	\$	0
B	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	1

BWT	\$	a	a	a	d	a	d	d	a	\$	0	\$	a	a	a	a	\$	c	a	a	a	b	b	b	a	c	c
U	4	4	3	2	1	4	3	2	4	3		2	1	3	1	3	1	1	1	3	2	1	3	2	1	3	2
C	-1	0	-2	-3	-4	1	2	3	5	6		7	4	9	12	13	14	16	17	15	11	18	19	20	21	22	23

Figure 11.1: Layout of our index organization for representing Web Access Logs over an example set of sequences.

11.2.2 Queries

ACCESS(u, i)**.** To obtain the i-th web page visited by user u within the sequence, we need to locate the position in the original sequence $\mathcal{T}(\mathcal{S})^R$ where the user's session begins. We can do this by computing $p = select_1(B, u+1) - 1 - i$ and then applying EXTRACT as $\mathcal{T}(\mathcal{S})^R[p, p]$ on the SSA index, in $O(s_a \log \sigma)$ time.

USERPATH(u)**.** To obtain the path done by user u, we compute $p_1 = \text{SELECT}_1(B, u)$ and $p_2 = \text{SELECT}_1(B, u+1) - 1$ and then EXTRACT on $\mathcal{T}(\mathcal{S})^R[p_1, p_2]$ using the `SSA` index. This takes $O((\ell + s_a) \log \sigma)$ time, where $\ell = p_2 - p_1$ is the length of the extracted path.

COUNT(P)**.** Given a path P of length m we can count its occurrences, by just performing COUNT (P) on the SSA index in $O(m \log \sigma)$ time.

MOST_COMMON(P, k)**.** We describe this operation incrementally. Assume we have already processed the sequence of requests $P = r_1, r_2, \ldots, r_{m-1}$. Our invariant is that we know the interval $[sp, ep]$ corresponding to path P, and this is sufficient to answer query MOST_COMMONP, k. Now a new request r_m arrives at the end of P. Then we proceed as follows:

1. Update the range $[sp, ep]$ in $BWT(\mathcal{T}(\mathcal{S}))$ using r_m, in $O(\log \sigma)$ time [67].
2. Retrieve the k most frequent symbols in $BWT[sp, ep]$, which are precisely those preceding the occurrences of P^R in $\mathcal{T}(\mathcal{S})^R$, or following P in $\mathcal{T}(\mathcal{S})$.

The second step is done with the heuristic proposed by Culpepper et al. [53] to retrieve the k most frequent symbols in a range of a sequence represented with a wavelet tree.

It is a greedy algorithm that starts at the wavelet tree root and maps the range (in constant time, using RANK on the bitmap of the wavelet tree node) to its children, until reaching the leaves. The traversal is prioritized by visiting the longest ranges first, and reporting the symbols corresponding to the first k leaves found in the process. The worst-case performance of this algorithm is bounded by the number of different symbols present in the string range. This is smaller than both σ and the size of the range. By using more sophisticated data structures (that nevertheless do not add much space) [88, 125], a worst case of $O(k + \text{polylog}\, n)$ time can be guaranteed. Note that, if we want to list *all* the request that have followed P, we can use an optimal algorithm based on depth-first traversal of the wavelet tree [74]. We show the details of this procedure in Algorithm 19.

LIST_USERS(P). To list k (or all) distinct users that have followed path P we first locate the starting and ending points $[sp, ep]$ for the given path, using the SSA index in $O(m \log \sigma)$ time (we can also proceed incrementally as in operation MOST_COMMON. Then we apply the optimal document listing algorihtm [115, 140]. Each value $C[p] < sp$, for $sp \leq p \leq ep$, signals a distinct value of $U[p]$ in $U[sp, ep]$. Recall that we do not have C or U anymore, but the procedure for extracting the list of users can be emulated with the RMQ data structure over array C and the bitmap B. Listing k distinct users for path P takes $O((k \cdot s_a + p) \log \sigma)$ time, since for each user u we need to take a position in the BWT and map it to the corresponding position in the original $\mathcal{T}(\mathcal{S})^R$, in $O(s_a \log \sigma)$ time. Figure 11.1 shows an example of listing users. The framed region represents the range sp, ep. Red nodes in the RMQ tree represent the position of the retrieved users, while the blue node represents the last visited node before returning.

MOST_FREQUENT(k, n). We want to retrieve the k most frequent paths of a certain length q done by the users in the system. We start by pushing into a priority queue all ranges obtained by performing a backward-search of paths of length 1 for each possible symbol (σ at most). Now, we extract the biggest range from the priority queue as well as the path that created that range. We execute the same procedure again, creating new paths by appending to the extracted path one further symbol (trying the σ possible ones in the worst case) and we push the ranges obtained by performing the backward search on these new paths into the priority queue. When we extract a path of length q, we report it and remove it from the priority queue. The procedure ends when k paths are reported or when the priority queue is empty. In the worst case, this operation can take $O(\sigma^q)$.

This method may perform poorly when the alphabet σ is large. An optimization is to avoid trying out all the σ characters to extend the current path, but just those symbols that do appear in the current range. Those are found by traversing the wavelet tree from the root towards all the leaves that contain some symbol in the current range [74].

Data Structure	**Msnbc**	**Kosarak**	**Spanish**
`SSA`	3,175,504	12,500,964	75,667,016
RMQ	1,770,690	2,807,570	39,538,238
Users Bitmap	608,044	777,804	9,413,732
Total	**5,554,246**	**16,086,346**	**124,618,994**
Plain	**5,782,495**	**18,884,286**	**120,613,202**
Ratio	**0,96**	**0,85**	**1,03**

Table 11.1: Space usage, in bytes, of the data structures used in our index. Plain corresponds to the sum of the space usage of plain representations of the sequence and users. Ratio corresponds to the total index size divided by the plain representation size.

11.3 Experiments and Results

11.3.1 Setup and Implementations

We used dedicated server with 16 processors Intel Xeon E5-2609 at 2.4GHz, with 256 GB of RAM and 10 MB of cache. The operating system is Linux with kernel 3.11.0-15 64 bits. We used g++ compiler version 4.8.1 with full optimizations (-O3) flags.

We implemented the `SSA` index using the public available wavelet tree implementation obtained from `LIBCDS` and developed the heuristic proposed by Culpepper et al. [53] on top of that implementation. The wavelet tree needed for the `SSA` index uses a `RRR` (see Section 2.6) compressed bitmap representation. The bitmap B needed to retrieve the users is used in plain form (`RG`). We implemented the RMQ data structure based on compact tree representations [14], which in practice requires $2.38n$ bits. Our implementation is available at `https://gitlab.com/fclaude/wum-index/`.

11.3.2 Experimental Data

We used web access sequences from the public available Msnbc, Kosarak, and Spanish datasets. The Msnbc dataset comes from Internet Information Services log files of msnbc.com for a complete day of September, 28 of 1999. It contains web access sequences from $989,818$ users with an average of 5.7 web page categories visits per sequence, the alphabet size of this dataset is $\sigma = 17$. The Kosarak dataset contains the click-stream data obtained from a Hungarian on-line news portal. It contains sequences of $990,000$ users with an average of 8.1 web page visits per sequence and an alphabet size $\sigma = 41,270$. Finally, the Spanish dataset contains the visitors' click-stream obtained from a Spanish on-line news portal during September 2012. This dataset consists of $9,606,228$ sequences of news-categories that were visited, with an average of 12.3 categories visited per sequence and an alphabet size of $\sigma = 42$.

11.3.3 Space Usage

Table 11.1 shows the space usage of each data structure for each dataset. Row "Plain" shows the space required to represent the original sequence $\mathcal{T}(\mathcal{S})^R$ using an array of $N \log \sigma$ bits plus an array to represent the users using $n \log N$ bits. The table shows that our index is able to compress the sequence by up to 15%, on the Kosarak dataset, while requiring only 3% extra space at most, on the Spanish dataset. Within this space, we are able to support the aforementioned operations, while at the same time can reconstruct the original sequence and the users information.

We evaluated alternatives to the `SSA`, such as the *Compressed Suffix Array* (`CSA`) [139] and the *Compressed Suffix Tree* (`CST`) [142] (recall Section 2.14.1), using the ones provided by the `SDSL-LITE`. Table 11.2 compares their space usage to our `SSA` at representing the sequence. The `SSA` is a better choice in this case, using up to 33% less space than the others. The `CST` could be used to compute some of the operations, since it is naturally a faster alternative. In fact, we evaluated this alternative for counting the occurrences of a sequence, and it is in practice four to twenty times faster than the `SSA`. We discarded it since the space requirement makes it unpractical for massive datasets. In fact, the `CST` needs to be augmented in order to support all the operations presented in this work, which would increase its memory usage.

11.3.4 Time Performance

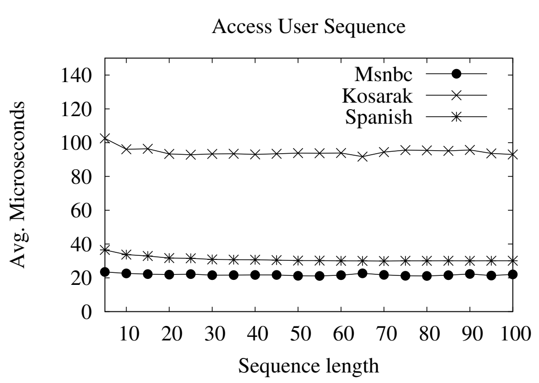
To evaluate the main operations using our proposed data structure, we generate query paths by choosing uniformly at random a position in the original sequences, and then extracting a path of the desired length.

Figure 11.2 (left) shows the time to access positions chosen uniformly at random from users whose traversal log has length 1 to 100. The time does not depend on the length, but directly on $s_a \log \sigma$. Fig. 11.2 (right) shows the time per symbol extracted, when we access the whole sequence associated with a user. We can see that for users with short interactions the SSA sampling has a greater effect, and this is amortized when accessing longer sequences, that is, the term $O(s_a \log \sigma)$ is spread among more extracted symbols and the time converges to $\log \sigma$.

We then measured the time to count the number of times a certain path appears in the access sequence. Figure 11.3 (left) shows the time per query. As expected, it grows linearly with the length of the path being counted, and the $\log \sigma$ term determines the slope of the

Data Structure	**Msnbc**	**Kosarak**	**Spanish**
`SSA`	1,08	0,77	0,85
`CSA`	1,61	0,87	1,13
`CST`	3,82	1,43	2,96

Table 11.2: Comparison of the space consumption of the `SSA`, `CSA`, and `CST` for representing sequence $\mathcal{T}(\mathcal{S})^R$. We show the ratio of each index space over the plain representation of the sequence without the user's information by using $N \log \sigma$ bits.



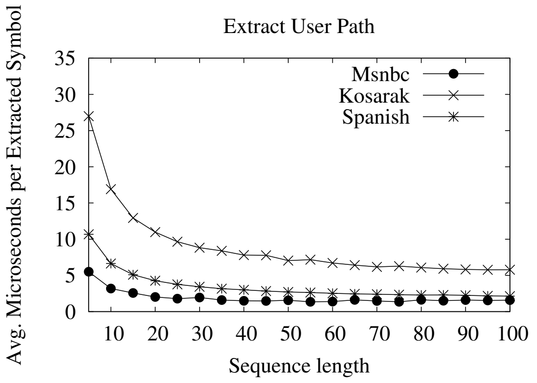
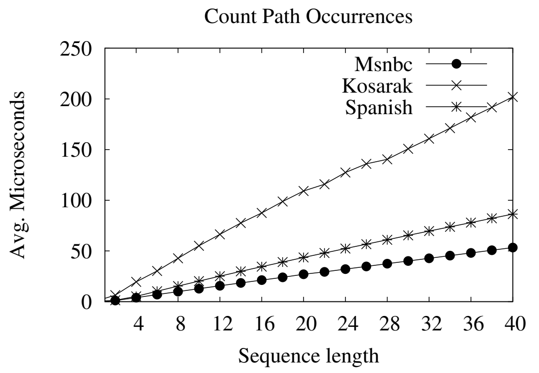


Figure 11.2: On the left, average microseconds to perform ACCESS (u, i); on the right, average microseconds per extracted symbol for the operation USER_PATH (u).



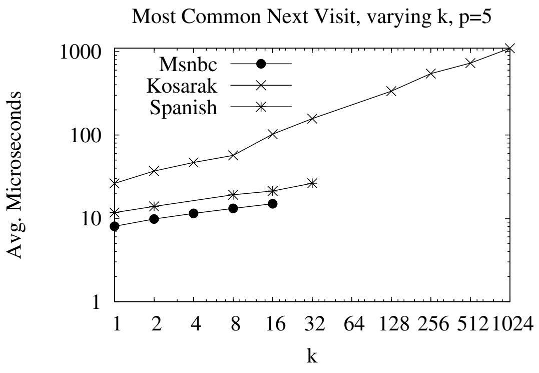
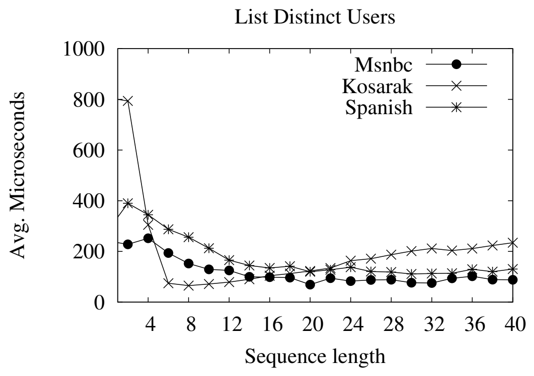


Figure 11.3: On the left, average microseconds to perform COUNT (P) on the right, average microseconds for the MOST_FREQUENT operation with varying k using patterns of fixed length $p = 5$.

line. On Figure 11.3 (right) we show the results for the MOST_COMMON operation. The x-axis and y-axis are in log scale. For datasets containing small alphabets such as Msnbc ($\sigma = 17$) and Spanish ($\sigma = 42$) this operation is performed in under 30 microseconds for all possible values of k (note it is impossible to obtain more than σ distinct symbols), and shows a logarithmic behavior. We also note that the slope of the logarithm depends on the value of σ, as shown in the Kosarak dataset. The operation, however, is still reasonably fast, taking less than 1 millisecond for $k = 1024$.

Figure 11.4 (left) shows the time for retrieving the set of users that followed a given access pattern in the system. For shorter sequences, the index has to retrieve a bigger set of users, as these sequences are more likely to appear. As the sequences grow in length, the time decreases, since the resulting set is also smaller. The behaviour for Kosarak, which after a certain point starts increasing in time per query, can be explained by the fact that determining the range $[sp, ep]$ grows linearly with the length of the pattern, and at some point it dominates the query time. This is also expected, at a later point, for Msnbc and Spanish.

Finally, Figure 11.4 (right) shows the query time for operation



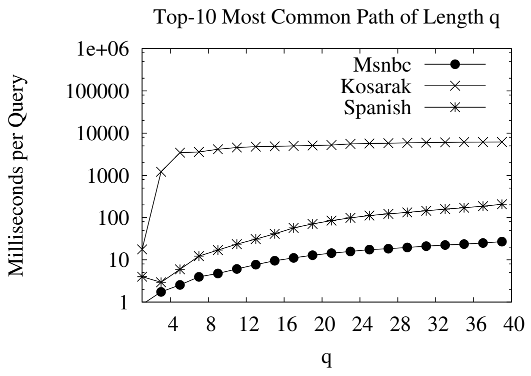


Figure 11.4: On the left, average microseconds to list distinct users that traversed paths with varying lengths. On the right, the time required to perform top-10 most common path operation for varying pattern lengths.

MOST_FREQUENT (k, q). Our first implementation tried following all symbols at every step of the algorithm. This worked quite well for small alphabets but had a very bad performance on the Kosarak dataset. This plot shows the implementation traversing the wavelet tree at each step to only follow symbols that do appear in the range. This gives a slightly worse performance for small alphabets, but a considerable speedup (1–2 orders of magnitude) for larger ones.

11.4 Discussion

We introduced a new data structure for handling web access sequences in compressed form that fully replaces the original dataset and also supports useful operations for typical WUM processes. Our experiments show that our index uses about the same size of the plain representation of the data, and within this space supports various relevant operations within tens of microseconds. This is competitive; consider that in most common scenarios the systems have to reply over a network, which have considerable latency and transfer time. Ours is the first compressed representation tailored for this scenario.

We have not yet fully explored other possible operations of interest in log mining that can be supported with our arrangements. For example, we can count the number of users that followed some path in constant time using $2n + o(n)$ bits using document counting [140], compute the k users that have followed a path most frequently using top-k document retrieval [88], and others.

Our index can be easily adapted to custom scenarios by adding satellite information, such as duration of the visit, actions (buy, login/logout, comment, etc.), browser information, location, and others, to each event in the log and include the information by mapping it to the BWT transform for later processing. This enables our index to be applied in other scenarios involving ordered sequences, such as GPS trajectories, stock price series, customer buying history, and so on.

Chapter 12

PcapWT: Indexing Network Packet Traces

In this chapter we present a solution that is inspired by the dual sorted inverted index data structure introduced in Section 3.3. We explore the idea of indexing large network packet traces using wavelet trees, and we show that our index is competitive in terms of space and time to the state of the art tools that network administrators use regularly.

The results of this work appear in the IEEE Journal on Computer Networks, 2015 and was done in collaboration with Young-Hwan Kim from INRIA, France, Diego Dujovne from Universidad Diego Portales, Chile, Thierry Turletti and Walid Dabbous from INRIA, France.

12.1 Motivation

The volume of network packet traces becomes larger and larger, and much more complex than before. The reason is that the speed and size of networks have increasingly expanded. Thus, much more H/W resources (e.g., massive storage space, stronger computing power, and so on) are necessary to deal with big traces including numerous packets and complex traffic patterns. Furthermore, efficient tools are required to analyze the data effectively. However, traditional tools such as *tcpdump* [94], *tcpflow* [62], or *tcpstat* [85] are inefficient at handling very large packet traces.

In order to enhance packet trace analysis, a number of contributions have been published, such as fast packet filtering [58], packet trace compression [86, 133], and network statistics information extraction [63]. Moreover, a few practical tools have also been recently proposed, such as *PCAP-Index* [4], Compressed Pcap Packet Indexing Program (CPPIP) [1], and *pcapIndex* [71, 5]. Overall, they use extra data sets for fast search, which are extracted from the original trace.

However, these tools result in poor performance, as we show in Section 12.4.2. *PCAP-*

Index running on top of a database requires too much space for handling the index data, and also its packet extraction procedure is considerably slow. CPPIP can save storage space by compressing original traces, but it can only support a few simple queries. *PcapIndex* is much faster than these tools, and is able to retrieve and extract matched packets from large size trace files (up to 4.2 GB). However, the index data size built by *pcapIndex* is abnormally increased when the number of packets is in the millions. Thus, there is a trade-off between the index data size and the packet extraction performance when processing a large packet trace.

12.2 Related Work

Today most network trace analysis tools, such as *tcpdump* [94] and *wireshark* [49], run on a single thread, and their complexity increases linearly with the volume of original trace files. Possible solutions to improve their performance include using a higher processor clock speed, replacing the SSD by a Hard Disk Drive (HDD), or splitting the large packet trace into multiple pieces. For that reason, several tools have been recently published to enhance performance of packet extraction on large traces, such as `CPPIP` [1], `PCAP-INDEX` [4], and `pcapIndex` [71].

`CPPIP` uses bgzip [103] (i.e., a block compression/decompression software) to reduce the volume of original packet trace files. This tool extracts an index data from the compressed trace file, and filters out matched packets directly from the compressed file. Although this tool is able to reduce the storage space for the original trace files, it can only support simple queries, such as packet number and received timestamp. In addition, CPPIP needs a significant amount of space (about 7% of the original trace file) to store the index data.

`PCAP-INDEX` uses a database (SQLite Ver. 3) to build the index data and to perform packet lookup. Thus, this tool is more flexible than other command-line based tools to express queries. However, as discussed in Section 12.4.2, the performance obtained is poor in terms of time needed to build index data, packet lookup time and index data size.

`PCAPIINDEX`, having a similar name as `PCAP-INDEX`, is currently part of a commercial network monitoring solution [5]. In order to reduce both index data size and packet lookup time, this tool adopts an encoding method using a bitmap compression technique based on a pre-defined codebook, named COMPressed Adaptive index [72]. Thus, it obtains better performance than CPPIP and `PCAP-INDEX`, in terms of index data size and packet extraction time. In comparison with `tcpdump`, this tool can reduce packet extraction time up to 1,100 times. Moreover, its index data size only takes about 7 MB per GB of the original trace file. However, as discussed in Section 12.4.2, `PCAP-INDEX` is not as fast compared to what is mentioned in the work of Fusco et al. [71], when it extracts a large amount of packets from a big trace file. In particular, the index data size rapidly increases when the volume of original trace file is greater than 4.2 GB.

12.3 The Design of `pcapWT`

In this section, we describe how to build the index data, and explain the process for lookup querying and for extracting packets. The overall processes consist of six steps, as shown in Figures 12.1 through 12.5.

12.3.1 Building Index Data

The first step is to extract index data from a packet trace file, such as packet offset, Ethernet source (Src) / destination (Dst) addresses, IPv4 Src/Dst addresses, Src/Dst port numbers, and Layer 4 (L4) protocol type. Here, the packet offset stands for the distance in bytes between two consecutive packets, which corresponds to the packet size in bytes. The reason why the offset is replaced by the packet size is that sequentially accumulated values consume a lot of memory. In addition, such large numbers are inefficient for the WT compression, since they increase the required number of bits to represent them. As shown in Figure 12.1, all index data are stored into each array, and they include the same number of elements with the number of total packets in the source trace file.

However, as mentioned in Section 2.7, the size of the wavelet tree compressed data is highly affected by the number of bits per element and the number of elements. For instance, when an element set (e.g., packet sizes referring packet offsets) of index data is $S = \{100, 200, 200, 300..., n\}$, the required number of bits per element is ($\lceil \log \sigma \rceil$), where σ is the maximum value among these input elements. Thus, for enhancing the compression performance, the elements have to be converted into sequential positive integer numbers ($\Sigma = \{1, 2, ..., \sigma\}$), and long arrays must be divided into multiple pieces ($S = \{S_1, S_2, ..., S_N\}$). For example, as shown in Figure 12.2, IPv4 addresses are mapped into positive integer numbers in consecutive order. In addition, to increase efficiency, we use a balanced tree (B-tree) to map the addresses with a positive integer. Consequently, a trace file accompanies a number of mapping tables, and the tables are provided as part of the index data.

In the last step for building index data, as shown in Figure 12.3, we create a wavelet tree on chunks of data every one million elements, and each chunk is stored in an independent file. This value has been empirically set to minimize the compressed index data size.

Meanwhile, the packet offset is not compressed by wavelet tree, because the large value of σ causes significant degradation of the compression performance. As an alternative, we use an efficient array provided by the `LIBCDS` package which removes redundant bits.

12.3.2 Packet Lookup and Extraction

The query syntax to perform the lookup and extraction is as follows,

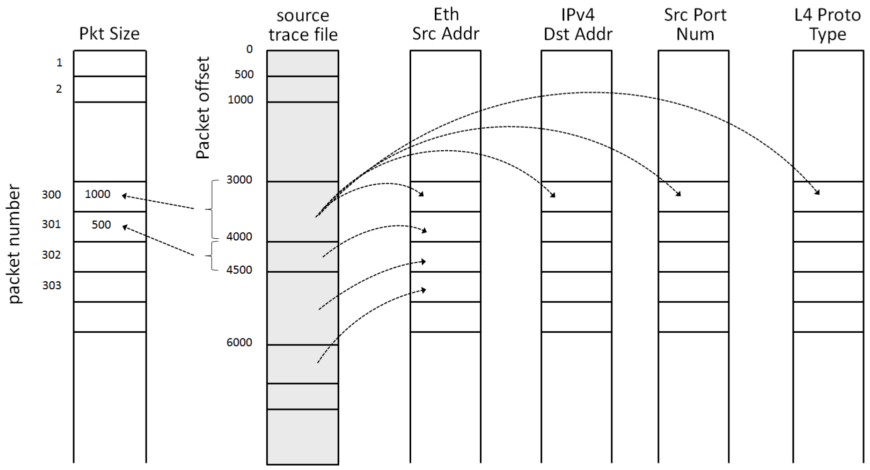


Figure 12.1: Extracting index data from a source packet trace file, such as packet size, Ethernet Src/Dst address, IPv4 Src/Dst address, Src/Dst port number, and Layer 4 protocol type.

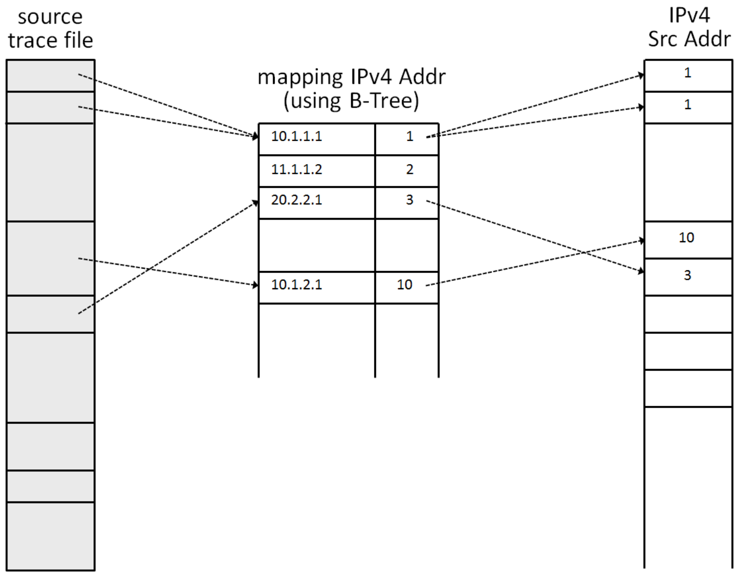


Figure 12.2: Mapping between the extracted index data and positive integers in consecutive order for reducing the index data size and packet lookup time.

$$\begin{aligned} Q &:= Q_{type}\ Q_{operand} \\ &:= Q_1\ and/or\ Q_2 \end{aligned}$$

A basic query consists of a query type (Q_{type}) and a query operand ($Q_{operand}$). The query type defines the kind of index data extracted from the traces, and `pcapWT` supports *pkt_num*, *eth_src*, *eth_dst*, *ipv4_src*, *ipv4_dst*, *port_src*, *port_dst*, and *l4_proto*. The query operand is the value that needs to be found on a data set of index data which is indicated by the query type. For example, the Q_{type} is *ipv4_src* and the valid $Q_{operand}$ is 10.1.1.1. The query syntax also allows to perform intersection (*and*) or union (*or*) of different queries, thus we can construct more complex queries, such as Q_1 *and* Q_2 *and* ... *or* Q_n.

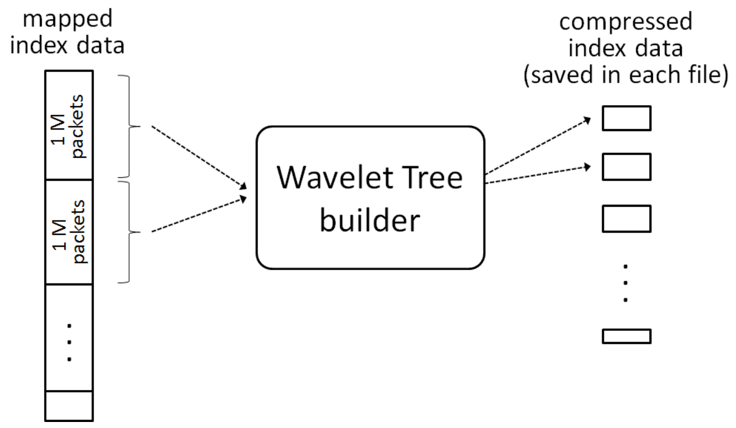


Figure 12.3: Building wavelet tree chunks from the mapped index data once every one million packets. This partitioning can reduce the size of index data.

Recall the basic functions described in Section 2.7. The main idea is that we can get the number of matched packets (m) containing the query operand by performing RANK of the wavelet tree. Next, we can figure out indexing numbers of matched packets by iterative executing of SELECT.

First, the procedure to perform the lookup has to reconstruct the packet offset by accumulating the packet sizes. For example, if the packet sizes are $P = \{100, 200, 200, 300, ..., p_n\}$, their offsets are $O = \{0, 100, 300, 500, 800, ..., o_n\}$. Then, it loads an index data related to the query. At this step, the index data is located in main memory, however it does not need to be decompressed to execute a lookup.

The matched packets are marked on a matching table, which is composed of a Boolean array with the same number of entries as the number of packets included in the source pcap file. If there are multiple queries concatenated by *and*/*or* operators, those steps are repeated -from loading related index data- through updating the matching table. As shown in Figure 12.4, the matching table assigns 1 to its entries for the matched packets which are found by the first query or queries located after the *or* operator. On the other hand, if the packets are found by queries located after the *and* operator, the table assigns 1 only to entries kept 1, otherwise 0. Once the lookup procedure is completed, two lists are created where offsets and packet sizes for the final matched packets are marked 1 on the matching table.

The performance of packet extraction highly depends on the capability of storage devices. In particular, this function performs random read and sequential write from/to files. The random access performance is significantly degraded in comparison with sequential access, in terms of read and write throughput. Therefore, we need an efficient way to compensate the performance degradation. SSDs have a very short access delay (e.g., below 1 millisecond) compared to HDDs, thus we can reduce the idle time by distributing the I/O requests into multiple processes, without significant delay overhead. Consequently, the proposed design uses multiple threads to handle multiple requests simultaneously, which are performed by OpenMP provided by gcc [3].

As shown in Figure 12.5, a number of threads are used to balance the overall workload for the packet lookup and extraction tasks. For example, if the number of matched packets is m and the number of threads is w, each thread will retrieve m/w packets to an output file simultaneously.

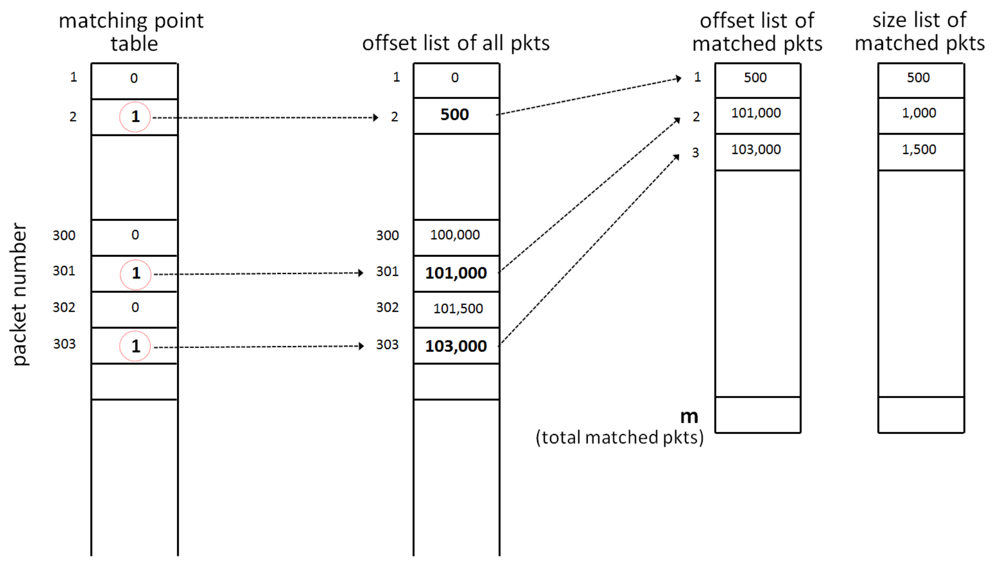


Figure 12.4: Listing up the final matched packets marked by '1' at the matching point table. Then the offset position and the size of selected packet are written into the offset list and the size list, respectively.

12.4 Performance Evaluation

`pcapWT` has been developed using gcc and `LIBCDS` on the Linux platform. In particular, `pcapWT` and `LIBCDS` have been compiled with g++ (Ver. 4.4.7) with the highest optimization flag (`-O3`). In addition, we set the `-fopenmp` option of g++ for enabling OpenMP. The machine used for benchmarking consists of CentOS 6.4 (kernel Ver. 2.6.32 for 64 bits x86 system), Intel i7-2600k processor including 8 threads (4 cores), 4 GB of main memory, a 60 GB SSD (OCZ Agility 3, low-grade product), and a 500 GB HDD (Toshiba HDD2F22).

12.4.1 Packet Traces Datasets

`pcapWT` supports *pcap* [94], one of the most popular packet trace formats. Table 12.1 shows four sample pcap files used in the performance evaluation, such as simulated traffic (ST) and real traffic 1/2/3 (RT-1/2/3). ST was generated by the ns-3 network simulator [2], and contains 15 million packets in a 4.2 GB pcap file. RT-1 was generated by capturing packets in our local network, and also contains 15 million packets in a 3.6 GB pcap file. ST consists of 8 endpoints at the transport layer, whereas RT-1 has 2416 endpoints. Meanwhile, those two samples are smaller than 4.2 GB because `pcapIndex` cannot support larger pcap files than the volume.

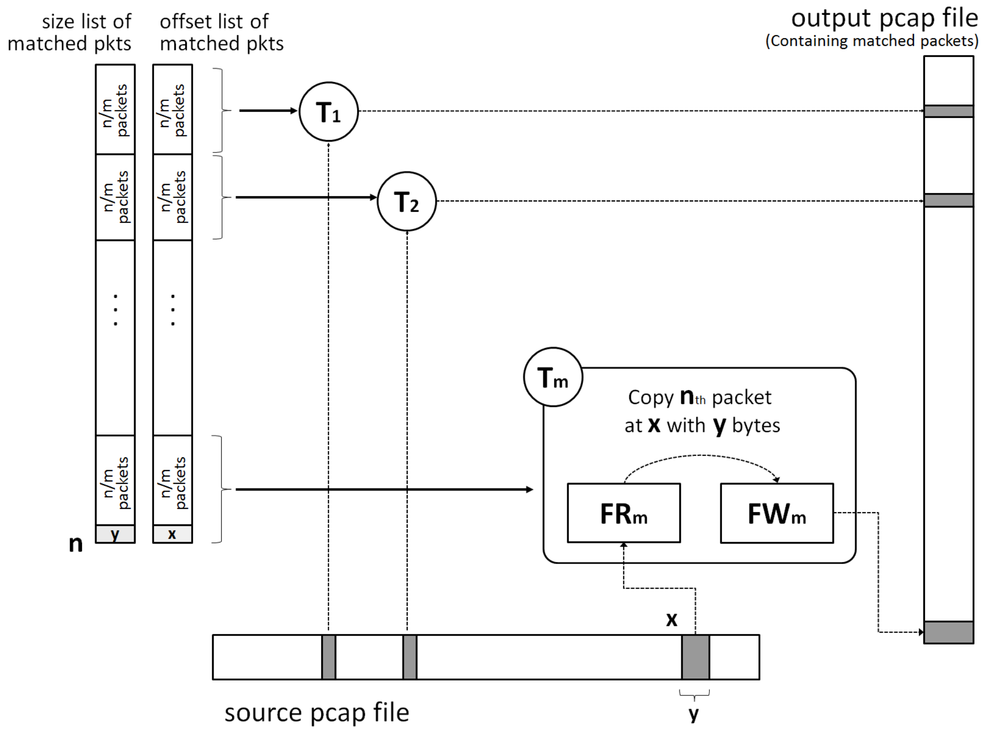

size list of matched pkts
offset list of matched pkts
output pcap file
(Containing matched packets)
n/m packets
T1
T2
Tm
Copy nth packet at x with y bytes
FRm
FWm
n
y
x
source pcap file

Figure 12.5: Retrieving the packets using multi-threaded file I/O. Multiple threads (m) evenly divide the number of the final matched packets (n), and each thread $(T_1, T_2, ..., T_m)$ copies the assigned packets from the source to output pcap file, simultaneously.

RT-2 and RT-3 are used to evaluate larger source trace files than 4.2 GB. Those two samples are generated by adding virtual packets to RT-1 in order to compare the results with RT-1 directly. Thus, RT-2 contains 115 million packets in a 13.4 GB pcap file, and the latter contains 200 million packets in a 21.6 GB pcap file.

As mentioned above, the packet extraction performance depends on the random read and write capabilities. Thus, we have manipulated those two samples in order to contain a larger amount of packets than ST and RT-1, instead of reducing their average packet sizes.

	ST	RT-1	RT-2	RT-3
Number of packets [million]	15	15	115	200
File size [GB]	4.2	3.6	13.4	21.6
User data size [GB]	3.9	3.4	11.5	18.4
Average packet size [Byte]	261.7	224.1	100.1	92.2
Number of end-points	8	2416	2417	2417

Table 12.1: Description of four sample pcap files: simulated traffic (ST) and real traffic 1/2/3/ (RT-1/2/3).

12.4.2 Baselines

We analyze the performances of other tools based on packet indexing, such as Compressed Pcap Packet Indexing Program (CPPIP) [1], PCAP-INDEX [4], and pcapIndex [5].

In order to build the index data from RT-1, pcapIndex and CPPIP need 27 MB (0.8% [1]) and 289 MB (8.0%) of storage, respectively. However, PCAP-INDEX needs around 2.6 GB (72%), even though PCAP-INDEX deals with a few simple fields (e.g., addresses, port numbers, protocol types, and so on). This is due to the database, which is not configured to use data compression. Moreover, PCAP-INDEX needs around 265 seconds to complete, whereas pcapIndex and CPPIP take only around 25 and 40 seconds, respectively. The reason is that PCAP-INDEX requires a lot of time to inject the index data into the database.

CPPIP and pcapIndex take around 20 and 30 seconds to extract the largest traffic corresponding to 48% of the volume and 36% of the total number of the packets of RT-1, respectively. However, the available operations are not the same ones used as in the prior case, since this tool supports only simple filtering queries using a range of packet timestamps and striding (e.g., selecting every n-th packets). Moreover, PCAP-INDEX takes about 80 seconds (packet lookup time from the database about 30 seconds and file writing time about 50 seconds), which is significantly slower compared to the other tools. Even tcpdump takes less than 32 seconds in the same use case.

CPPIP affords the advantage of being able to extract packets directly from a compressed trace file. However, some efforts are needed to support more complex filtering queries and to reduce the index data size as much as that of pcapIndex. PCAP-INDEX is not satisfactory from many points of views, even though it uses a database. In contrast, pcapIndex is the only reliable one among those tools, since it provides not only comprehensive query operations, as well as remarkable performances in terms of index data size and packet extraction time. Therefore, we select pcapIndex to compare performance with pcapWT in the following.

12.4.3 Space

pcapWT aims to minimize the index data size, for example to compress it below 1% of the volume of original trace files. As shown in Table 12.4.3, pcapWT is successful for all sample files.

More precisely, as mentioned in Section 2.7, wavelet trees supports different bitmap representation methods (i.e., RG and RRR) and its encoding unit size that can be set by the user. Figure 12.6 shows that RRR is slightly better at reducing index data size than RG. On the other hand, according to Figures 12.7 and 12.8, RG is more efficient than RRR, in terms of index data building time and the packet lookup time. Especially, as shown in Figure 12.6, RG using a wide sampling provides mostly complementary performance between those features. Thus, we decided to use RG(50) as the default configuration.

[1] a percentage of the volume of original trace file.

In addition, for improving compression efficiency and reducing search time at the same time, we use Huffman shaped wavelet tree which combines multi-layered wavelet trees and entropy compression of the bitmap sequences. Moreover, we minimize redundant bits (e.g., consecutive 0 bits) that can be generated in between multi-layered wavelet trees, using Huffman compression.

In contrast, `pcapIndex` obtains an abnormally large index data size and causes a segmentation fault when building the index data for trace files larger than 4.2 GB. In the case of RT-2, the index data size corresponds to 6.86% of the volume of the original trace file, and around 3.0 GB of main memory is required. Furthermore, this tool fails to build the index data for RT-3 because of lack of memory. Note that RT-2 and RT-3 have been generated by adding virtual packets to RT-1, in order to provide a direct comparison with the index data size and packet extraction performance of RT-1. More precisely, the virtual packets are equal except for the timestamp field. Thus, the index data from those samples has to be minimized regardless of the number of packets. Nevertheless, the index data size produced by `pcapIndex` exponentially increases with the number of packets.

	pcapIndex [%]	pcapWT [%]
ST	0.45	0.30
RT-1	0.78	0.86
RT-2	6.86	0.93
RT-3	Error	0.96

Table 12.2: A comparison of the index data size produced by *pcapIndex* and `pcapWT` with RG(50). The percentage indicates the ratio between the index data size and the volume of the original trace file.

12.4.4 Multi-threaded File I/O

As we can observe in Figure 12.9, `pcapIndex` and `pcapWT` have almost the same packet extraction performance on both the SSD and the HDD when using a single thread. Meanwhile, when `pcapWT` extracts a large portion (88% of the total packets) from ST on the SSD, 32 threads can reduce the time down to 26%, compared to when using a single thread. However, multi-threading on HDD does not provide any gain. Even worse, the time is increased to 24% when using 32 threads. In this case, multi-threading generates frequent random accesses which increase delay and degrade performance on HDD.

12.4.5 Packet Extraction

To analyze the performance of packet retrieval, we use various queries, and extract a large amount of packets. As shown in Table 12.3, simple queries (from A through E) extract packets specified by a pair of the query type and its operand, and complex queries (from F through J) indicate combining simple queries by using the union operator. For example,

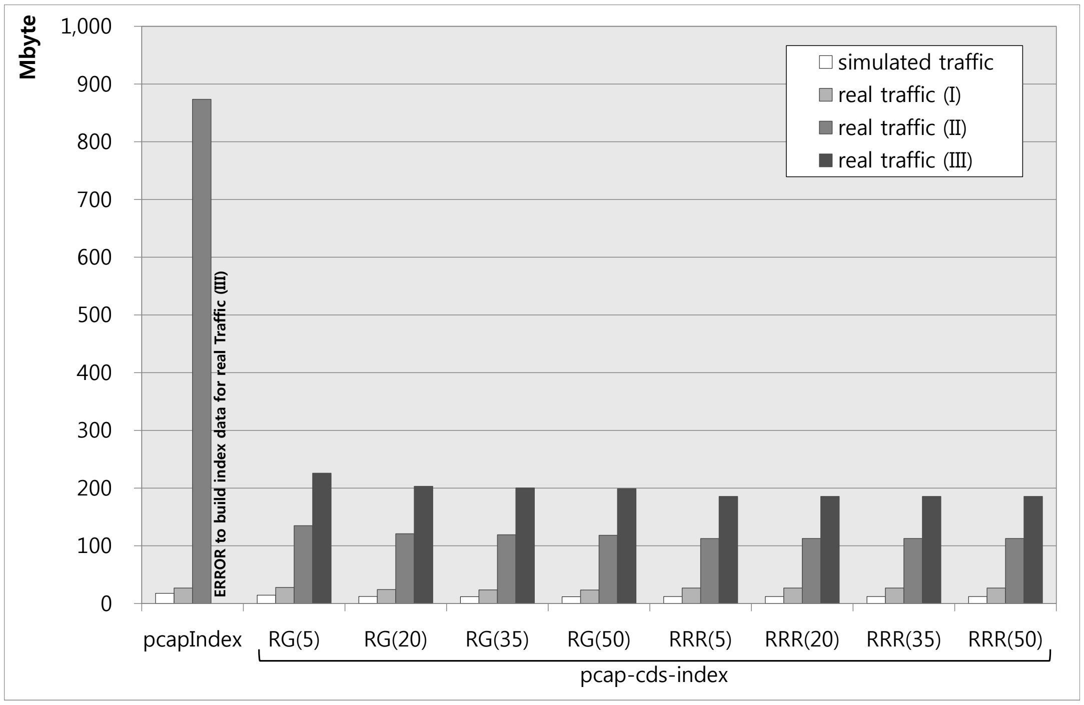


Figure 12.6: A comparison of the index data size by by bit sequence representations (RG and RRR) and block size parameters (5, 20, 35, 50).

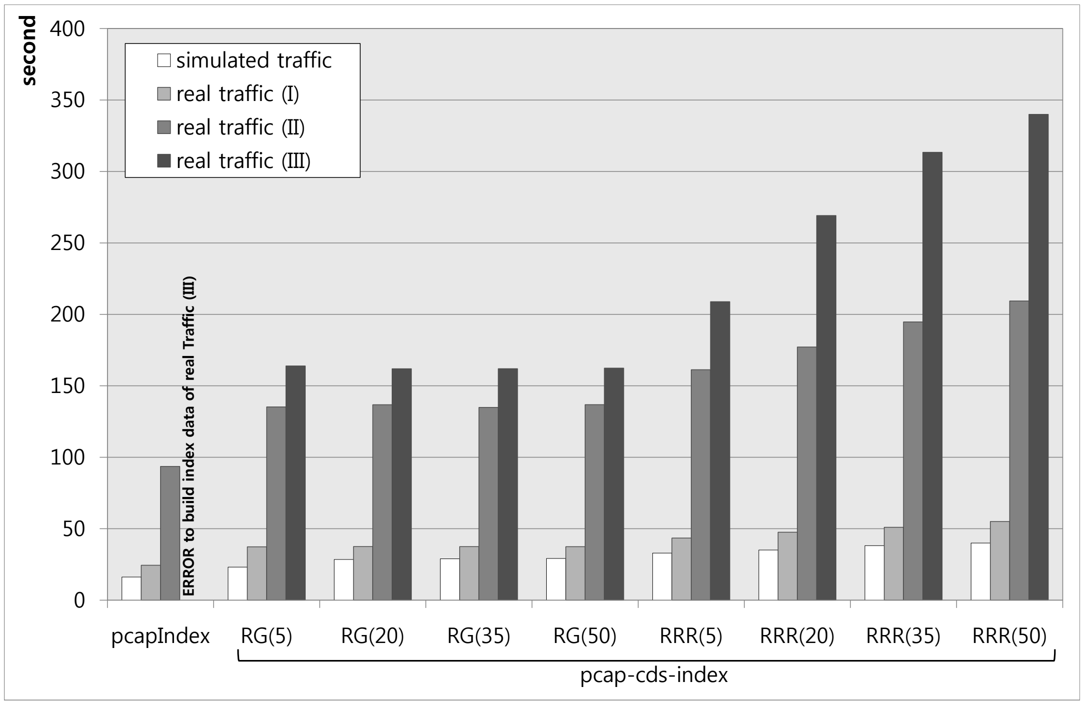


Figure 12.7: A comparison of the building time of index data by bit sequence representations (RG and RRR) and block size parameters (5, 20, 35, 50).

query A extracts 4,030 packets from RT-1/2/3, corresponding to 0.03%, 0.003%, and 0.002% in comparison with the total numbers of packets, respectively.

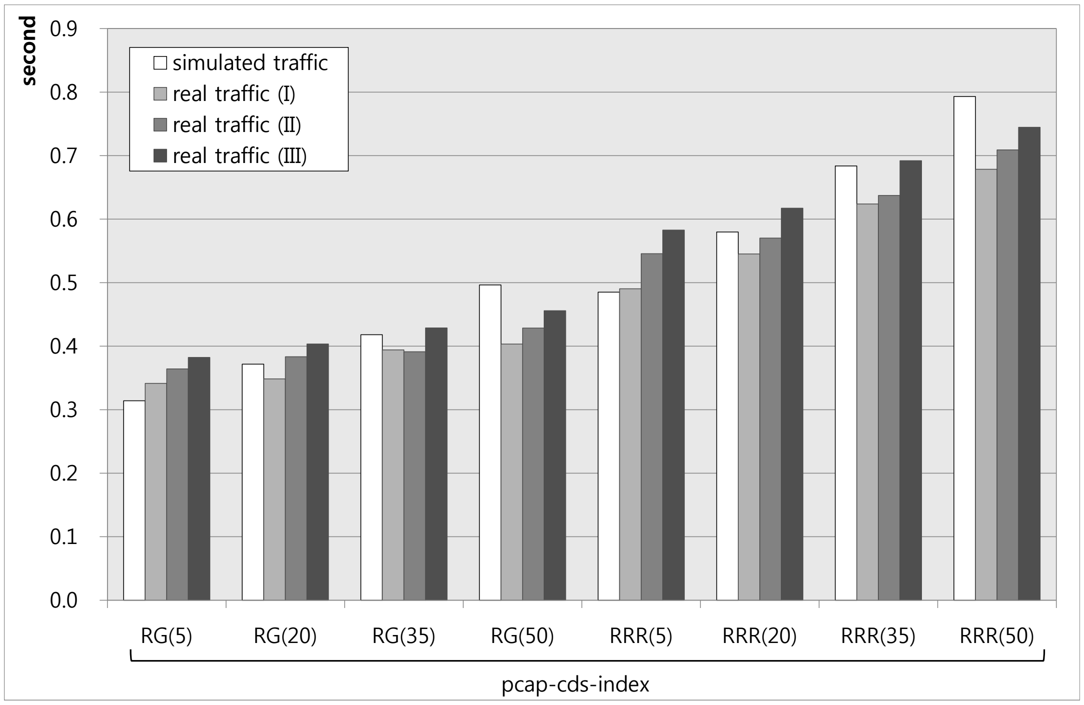


Figure 12.8: A comparison of the packet lookup time for different sampling sizes. In case of ST, the number of the retrieval targets is 9.6 million packets (i.e., 63.8% in comparison to the total number of packets included in this sample.), and the volume of the target packets corresponds to 1.82 GB (i.e., 43.7% in comparison with the original volume). In case of RT-1/2/3, the number of target packets are all same at 5.6 million packets and 0.97 GB, whereas the percentages of the number of the target packets are 37.5%, 3.8%, and 2.8%, and the percentages of the volumes of them correspond to 23.3%, 7.3%, and 4.5%, respectively.

In experiments using RT-1, `pcapIndex` and `pcapWT` are faster than `TCPDUMP` in retrieving small amounts of packets, such as queries A, B, C, F, G, and J. On the contrary, the gap of packet extraction performance becomes smaller between `TCPDUMP` and those two tools using the packet indexing scheme. The reason is that the performance depends highly on the number of packets extracted. Nevertheless, `pcapWT` (using 32 threads) is faster than `TCPDUMP` in experiments performing complex queries.

In experiments using larger samples (i.e., RT-2/3), we evaluate the performance of `pcapWT` and compare it with `TCPDUMP`, since `pcapIndex` cannot support such large traces. In these experiments, `pcapWT` uses a prompt mode which supports iterative operations once the index data is loaded. Thus, this mode can avoid unnecessary time loading the same index data repetitively. The results are shown in Figure 12.10.

12.5 Discussion

In network analysis, packet traces have always been important to analyze, since they record the complete information exchanged through networks. However, as network links, traffic

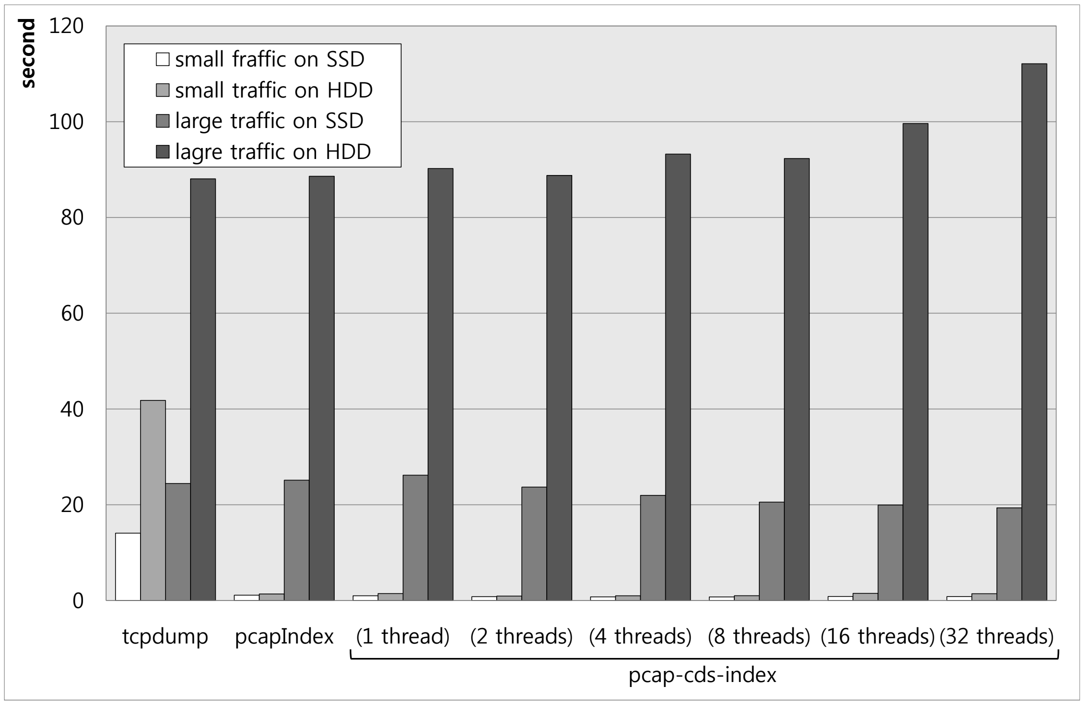


Figure 12.9: A performance comparison of packet extraction with various number of threads for parallel file I/O. In this test, `pcapWT` (shown as `pcap-cds-index` extracts two different size of traffics from ST: one small traffic containing 1.8 million packets (12.2%) corresponding to 146 MB and one large traffic containing 13.2 million packets (88.0%) corresponding to 3.0 GB. Note, the percentages indicate the ratio of the extracted packets compared to the total number of packets included in the sample.

Query (IP Addr)	Matched packets				Total packet size			
	$[\times 10^3]$	$[\%]^{(1)}$			[MB]	$[\%]^{(2)}$		
		(I)	(II)	(III)		(I)	(II)	(III)
A	4	0.03	0.00	0.00	0.3	0.01	0.00	0.00
B	521	3.5	0.35	0.26	754	20.94	5.65	3.48
C	1,161	7.7	0.77	0.58	298	8.28	2.23	1.38
D	5,631	37.5	3.75	2.82	971	26.96	7.27	4.49
E	5,428	36.2	3.62	2.71	1731	48.07	12.97	8.00
F = $\{A \cup B\}$	524	3.5	0.35	0.26	755	20.97	5.66	3.49
G = $\{F \cup C\}$	1,686	11.2	1.12	0.84	1053	29.24	7.89	4.87
H = $\{G \cup D\}$	7,317	48.8	4.88	3.66	2024	56.21	15.16	9.35
I = $\{H \cup E\}$	12,225	81.5	8.15	6.11	3000	83.31	22.47	13.86
J = $\{B \cap E\}$	521	3.5	0.35	0.26	754	20.94	5.65	3.48

Table 12.3: Packet filtering queries and specifications of RT-1/2/3. The percentage values indicate (1) the ratio of the number of matched packets and (2) the ratio of the size of matched packets compared to the packet number and the volume of the original trace file, respectively. Note that all numbers are rounded up.

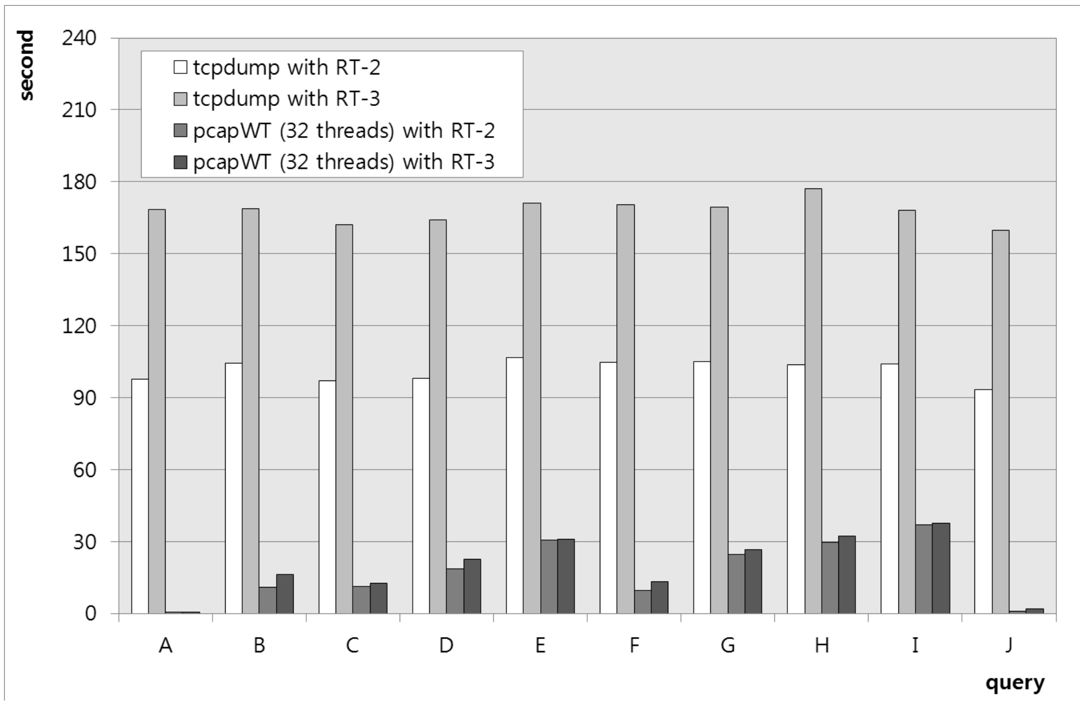


Figure 12.10: A performance comparison of packet extraction using RT-2/3.

complexity, and applications throughput have significantly increased, the analysis of huge amounts of those traces is becoming a challenging task, and traditional analysis tools are not efficient at dealing with such traces. Even though a few tools have been recently published for fast packet extraction, their performance is still not satisfactory in terms of index data size, and packet extraction delay.

In this work, we propose a new tool designed to process efficiently very large network traces, which allows not only enhancing packet extraction performance on large traces, but also reducing storage requirements. `pcapWT` uses a wavelet tree data structure and a multi-threading scheme for parallel file I/O. Our benchmarks including SSD show that `pcapWT` is about 10 to 20% faster than `pcapIndex` in the worst case scenario and about 200% in the best cases. Moreover, this tool allows reducing the storage space required to store the index data by about 10 to 35% compared to `pcapIndex`.

Part V

And Finally...

Chapter 13

Conclusions and Future Work

13.1 Summary of Contributions

In this thesis we have addressed the problem of speeding up queries at the first stage of query processing in Information Retrieval systems: given a user's query, retrieve as fast as possible the top-k most relevant candidate documents using a fast and simple ranking scheme. We have designed, implemented and evaluated new compact data structures and algorithms, and showed how to assemble them to create space-efficient indexes for information retrieval on natural language. We experimentally showed that these new indexes obtain interesting space/time tradeoffs when compared to the current state-of-the-art.

This section summarizes the main contributions of this thesis:

- We have presented a new representation of inverted indexes, dubbed *Inverted Treaps*. Inverted Treaps are able to simultaneously represent the docid and the weight ordering of a posting list. We use them to design efficient ranked union and intersection algorithms that simultaneously filter out documents by docid and frequency. The treap also allows us to represent both docids and frequencies in differential form, to improve the compression of the posting lists.

  We have also presented an engineered version of the theoretical index dubbed *Dual-Sorted* [122], and compared it against inverted index alternatives, including the inverted treaps.

  Experimental results show that under the tf-idf scoring scheme, inverted treaps use about the same space as competing alternatives like Block-Max and Dual-Sorted, but they are significantly faster: from 20 times faster on one-word queries to 3–10 times faster on ranked unions and 1–2 times faster on ranked intersections. Inverted treaps are generally the fastest alternative for $k \leq 100$, and on one-word and two-word queries, which are the most popular ones. In addition, we have shown that treaps can handle insertions of new documents, with a 50%–100% degradation in space, construction and query time performance.
- We proposed a more efficient and conceptually simpler implementation of the optimal

solution introduced by Navarro and Nekrich [119] for the top-k document retrieval problem. We presented three new indexes: WRMQ is a first naive approach to the optimal implementation, our second index, K^2Treap^H, is a remarkable improvement over the first approach thanks to the use of a new compact data structure, the K^2-treap, and by simplifying components of the optimal solution. Finally, our third index, W1RMQ^H, can be regarded as merged version of the first and second alternatives, which provides similar space/time tradeoffs. Although top-k document retrieval indexes traditionally consider the collection as general sequences, we show how these indexes can be used in natural language collections, in which case the index is much more powerful than inverted indexes at handling phrase queries. We also present an efficient construction method so that it is possible to handle large IR collections. Our experimental results showed that these indexes are orders of magnitude faster than the current state-of-the-art for most types of collections. Remarkably, our indexes are within the same space requirements as the best alternatives.

- We have extended the main ideas previously developed in this thesis, and make extensive use of compact data structure for tackling other scenarios that are important for IR.

  We presented a generalization of the technique used in the design of the K^2-treap data structure (based on the treap data structure introduced in Part II) to support other types of two-dimensional aggregated queries. We showed a particular case that solves the problem of counting the points that lie in a two-dimensional range $[x_1, x_2] \times [y_1, y_2]$. Our experimental evaluation showed that using little space overhead (less than a 30%) on top of the K^2-tree, this data structure performs queries several orders of magnitude faster than the original K^2-tree, especially when the query ranges are large.

  Based on the work developed for top-k document retrieval on general sequences, we presented an index for handling web access sequences in compressed form that fully replaces the original dataset and also supports useful operations for typical web usage mining. Our experiments showed that our index uses about the same size of the plain representation of the data, and within this space supports various relevant operations within tens of microseconds.

  Finally, we proposed a new tool to efficiently process very large network traces, which allows enhancing packet extraction performance on large traces while reducing the storage requirements. This tool is based on the idea of the Dual-Sorted index, introduced in Part II. Our index, dubbed `pcapWT`, uses a wavelet tree data structure and a multi-threading scheme for parallel file I/O. Our experimental evaluation showed that `pcapWT` is 10% to 20% faster than existing networking tools in the worst case and about 200% in the best scenarios. Finally, our index reduces the space required to store the indexed data by about 10% to 35% when compared to the space required by other typical tools.

13.2 Future Work

In this section we detail some future plans after this thesis. We describe the most interesting ones for each contribution.

- A future research line for experimenting with the inverted treap representation of inverted indexes is to study the effect of reassigning docids. Some results [59] show that reassignment can significantly improve both space and processing time. Another line of research is to introduce a mechanism to efficiently separate lists with higher frequencies or impacts. We believe that, since an important part of our gain owes to the separation of lists with frequency $f_0 = 1$, this can lead to important gains. It is also interesting to explore how this idea could impact on schemes like Block-Max and Dual-Sorted.
- An interesting direction to investigate is to evaluate if our efficient top-k document retrieval indexes for general string collections are able to compete with inverted indexes for weighted Boolean queries, which are considerably harder than phrase queries for our indexes. In addition, our current implementation supports term frequency as the relevance score. A possible next step is to support more complex relevance measures, such as BM25. Finally, A relevant open problem is to reduce the space further while retaining competitive query times; as shown in our experiments, current existing approaches that use less space are hundreds of times slower.
- The generalized technique to support aggregated queries presented in Chapter 10 can be generalized to represent grids in higher dimensions, by replacing our underlying K^2-tree with its generalization to d dimensions, the K^d-tree [55] (not to be confused with kd-trees [24]). The algorithms remain identical, but an empirical evaluation is left for future work.
- Our space-efficient web access logs index, can be further extended to support more complex scenarios by adding satellite information, such as duration of the visit, actions, browser information, location, and others, to each event in the log. How our index performs in practice for these extended scenarios remains uncertain.
- Compact data structures is an active field of research, yet, many computer science research areas have not taken advantage of the new developments coming from this area. We have shown that wavelet trees (Chapter 12) or top-k document retrieval indexes (Chapter 11) can be employed to assemble new indexes that offer interesting space/time tradeoffs in other scenarios of interest for IR. We believe that there are even more IR-related problems that will benefit from the use of compact data structures. An example is the representation of the item/user matrix employed in traditional collaborative filtering performed in recommender systems. The use of wavelet-trees or K^2-trees arises as natural, space-efficient alternatives to current space-demanding representations. Another case is the use of compact top-k document retrieval indexes (Part III) to efficiently represent large language models while providing fast access and operations on them. As an example of this line of research, Shareghi et al. [149] have recently showed that it is possible to use compressed suffix trees to represent language models that supports fast operations.

Bibliography

[1] Cppip. URL `http://blogs.cisco.com/tag/pcap/`. Last visited on 18/12/2015.

[2] Ns-3 Network Simulator. URL `http://www.nsnam.org/`. Last visited on 18/12/2015.

[3] Openmp. URL `http://gcc.gnu.org/projects/gomp/`. Last visited on 18/12/2015.

[4] Pcap-index. URL `https://github.com/taterhead/PCAP-Index/`. Last visited on 18/12/2015.

[5] PcapIndex. URL `http://www.ntop.org/products/n2disk/`. Last visited on 18/12/2015.

[6] D. Abhishek and J. Ankit. Indexing the world wide web: The journey so far. In *Next Generation Search Engines: Advanced Models for Information Retrieval*, pages 1–28, 2012.

[7] P. Afshani, L. Arge, and K. G. Larsen. Higher-dimensional orthogonal range reporting and rectangle stabbing in the pointer machine model. In *Proceedings of the Twenty-eighth Annual Symposium on Computational Geometry*, pages 323–332, 2012.

[8] A. Andersson and S. Nilsson. Faster searching in tries and quadtrees - an analysis of level compression. In *Proc. of the 2nd Annual European Symposium on Algorithms (ESA)*, LNCS 855, pages 82–93, 1994.

[9] V. Anh, O. Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *Proc. of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '01, pages 35–42. ACM, 2001.

[10] V. Anh and A. Moffat. Impact transformation: Effective and efficient web retrieval. In *Proc. of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '02, pages 3–10, 2002.

[11] V. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, 2005.

[12] V. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 372–379, 2006.

[13] V. Anh and A. Moffat. Index compression using 64-bit words. *Software Practice and Experience*, 40(2):131–147, 2010.

[14] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Proc. 11th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 84–97, 2010.

[15] N. Asadi and J. Lin. Fast, incremental inverted indexing in main memory for Web-scale collections. *CoRR*, abs/1305.0699, 2013.

[16] R. Baeza-Yates, A. Moffat, and G. Navarro. Searching large text collections. In *Handbook of Massive Data Sets*, pages 195–244. Kluwer, 2002.

[17] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, 2nd edition, 2011.

[18] R. Baeza-Yates and A. Salinger. Experimental analysis of a fast intersection algorithm for sorted sequences. In *Proc. 12th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 13–24, 2005.

[19] J. Barbay, F. Claude, T. Gagie, G. Navarro, and Y. Nekrich. Efficient fully-compressed sequence representations. *Algorithmica*, 69(1):232–268, 2014.

[20] J. Barbay, A. López-Ortiz, T. Lu, and A. Salinger. An experimental investigation of set intersection algorithms for text searching. *ACM Journal of Experimental Algorithmics*, 14:art. 7, 2009.

[21] D. Belazzougui, G. Navarro, and D. Valenzuela. Improved compressed indexes for full-text document retrieval. *Journal of Discrete Algorithms*, 18:3–13, 2013.

[22] M. Bender and M. Farach-Colton. The LCA problem revisited. In *Proceedings of the 9th Latin American Theoretical Informatics (LATIN)*, LNCS 1776, pages 88–94, 2000.

[23] D. Benoit, E. Demaine, I. Munro, R. Raman, and S. Rao V. Raman. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, December 2005.

[24] J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[25] J. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.

[26] M. Berg, O. Cheong, M. Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications.* Springer-Verlag TELOS, 3rd edition, 2008.

[27] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.

[28] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multireso-

lution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web (WWW)*. ACM Press, 2011.

[29] P. Boldi, M. Santini, and S. Vigna. Permuting web graphs. In *Proceedings of the 6th International Workshop on Algorithms and Models for the Web-Graph*, pages 116–126, 2009.

[30] P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. In *Proc. of the 13th International Conference on World Wide Web (WWW)*, pages 595–602, 2004.

[31] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the 13th International World Wide Web Conference (WWW)*, pages 595–601, 2004.

[32] N. Brisaboa, G. de Bernardo, R. Konow, and G. Navarro. k^2-treaps: Range top-k queries in compact space. In *Proc. 21st International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 8799, pages 215–226, 2014.

[33] N. Brisaboa, G. de Bernardo, R. Konow, G. Navarro, and D. Seco. Aggregated 2d range queries on clustered points. *Information Systems*, 60:34–49, 2016.

[34] N. Brisaboa, S. Ladra, and G. Navarro. DACs: Bringing direct access to variable-length codes. *Information Processing Management.*, 49(1):392–404, 2013.

[35] N. Brisaboa, S. Ladra, and G. Navarro. Compact representation of web graphs with extended functionality. *Information Systems*, 39(1):152–174, 2014.

[36] A. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. 12th CIKM*, pages 426–434, 2003.

[37] C. Buckle and A. Lewit. Optimization of inverted vector searches. In *Proc. of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 97–110, 1985.

[38] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, 1994.

[39] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-time search at Twitter. In *Proceedings of the 28th IEEE conference on Data Engineering (ICDE)*, pages 1360–1369, 2012.

[40] S. Büttcher, C. Clarke, and G. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines.* MIT Press, 2010.

[41] D. Caro, M. Rodríguez, and N. Brisaboa. Data structures for temporal graphs based on compact sequence representations. *Information Systerms*, 51:1–26, 2015.

[42] B. Chazelle. Functional approach to data structures and its use in multidimensional searching. *SIAM Journal of Computation*, 17(3):427–462, 1988.

[43] B. Chazelle. Lower bounds for orthogonal range searching I: The reporting case. *Journal of the ACM*, 37(2):200–212, 1990.

[44] D. Clark. *Compact PAT trees.* PhD thesis, University of Waterloo, 1997.

[45] F. Claude, R. Konow, and G. Navarro. Efficient representation of web access logs. In *Proc. 21st International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 8799, pages 65–76, 2014.

[46] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 176–187, 2008.

[47] F. Claude, P. Nicholson, and D. Seco. Differentially encoded search trees. In *Proc. 22nd Data Compression Conference (DCC)*, pages 357–366, 2012.

[48] F. Claude, P. Nicholson, and D. Seco. On the compression of search trees. *Information Processing Management*, 50(2):272–283, 2014.

[49] G. Combs. Wireshark. pages 12–02, 2007. Last visited on 18/12/2015.

[50] B. Croft, D. Metzler, and T. Strohman. *Search Engines: Information Retrieval in Practice.* Pearson Education, 2009.

[51] J. Culpepper and A. Moffat. Compact set representation for information retrieval. In *Proceedings of the 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 137–148, 2007.

[52] S. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *ACM Transactions on Information Systems*, 29(1):1:1–1:25, 2010.

[53] S. Culpepper, G. Navarro, S. Puglisi, and A. Turpin. Top-k ranked document search in general text databases. In *Proc. 18th Annual European Symposium on Algorithms (ESA B)*, LNCS 6347, pages 194–205 (part II), 2010.

[54] G. de Bernardo, S. Alvarez-García, N. Brisaboa, G. Navarro, and O. Pedreira. Compact querieable representations of raster data. In *Proc. 20th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 8214, pages 96–108, 2013.

[55] G. de Bernardo, S. Álvarez-García, N. Brisaboa, G. Navarro, and O. Pedreira. Compact querieable representations of raster data. In *Proc. 20th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 8214, pages 96–108, 2013.

[56] J. Dean. Challenges in building large-scale information retrieval systems. In *ACM Workshop on Web Search and Data Mining*, 2009.

[57] E. Demaine, A. López-Ortiz, and I. Munro. Adaptive set intersections, unions, and differences. In *Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 743–752, 2000.

[58] L. Deri. High-speed Dynamic Packet Filtering. *Journal of Network and Systems Management*, 15(3):401–415, 2007.

[59] S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In *Proc. 34th SIGIR*, pages 993–1002, 2011.

[60] J. Domènech, J. Gil, J. Sahuquillo, and A. Pont. Web prefetching performance metrics: a survey. *Performance and Evaluation*, 63(9):988–1004, 2006.

[61] X. Dongshan and S. Junyi. A new markov model for web access prediction. *Computing in Science and Engg.*, 4(6):34–39, 2002.

[62] J. Elson. Tcpflow. URL `http://www.circlemud.org/jelson/software/tcpflow/`, 2009. Last visited on 18/12/2013.

[63] C. Estan, K. Keys, D. Moore, and G. Varghese. Building A Better NetFlow. *ACM SIGCOMM Computer Communication Review*, 34(4):245–256, 2004.

[64] H. Ferrada and G. Navarro. Efficient compressed indexing for approximate top-k string retrieval. In *Proc. 21st International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 8799, pages 18–30, 2014.

[65] H. Ferrada and G. Navarro. Improved range minimum queries. In *Proc. 26th Data Compression Conference (DCC)*, pages 516–525, 2016.

[66] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13(12), 2009.

[67] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

[68] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 3(2):20–45, 2007.

[69] J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011.

[70] M. Fontoura, V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Zien. Evaluation strategies for top-k queries over memory-resident inverted indexes. *Proceedings of the VLDB Endowment*, 4(12):1213–1224, 2011.

[71] F. Fusco, X. Dimitropoulos, and M. Vlachos L. Deri. PcapIndex: An ndex for Network Packet Traces with Legacy Compatibility. *ACM SIGCOMM Computer Communication Review*, 42(1):47–53, 2012.

[72] F. Fusco and M. Vlachos M. Stoecklin. NET-FLi: On-the-fly Compression, Archiving and Indexing of Streaming Network Traffic. *Proc. of the VLDB Endowment*, 3(1-2):1382–1393, 2010.

[73] T. Gagie, J. González-Nova, S. Ladra, G. Navarro, and D. Seco. Faster compressed quadtrees. In *Proc. 25th Data Compression Conference (DCC)*, pages 93–102, 2015.

[74] T. Gagie, G. Navarro, and S. J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426-427:25–41, 2012.

[75] T. Gagie, S. Puglisi, and A. Turpin. Range quantile queries: Another virtue of wavelet trees. In *Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 1–6, 2009.

[76] S. Gog. *Compressed Suffix Trees: Design, Construction, and Applications.* PhD thesis, Univ. of Ulm, Germany, 2011.

[77] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. 13th International Symposium on Experimental Algorithms, (SEA)*, pages 326–337, 2014.

[78] S. Gog and G. Navarro. Improved single-term top-k document retrieval. In *Proc. 17th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 24–32. SIAM Press, 2015.

[79] A. Golynski, I. Munro, and S. Rao. Rank/select operations on large alphabets: A tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.

[80] R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, 2005.

[81] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 841–850, 2003.

[82] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[83] Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. Frequent pattern mining: current status and future directions. *Data Mining and Knowledge Discovery*, 15(1):55–86, 2007.

[84] H. Heaps. *Information Retrieval - Computational and Theoretical Aspects.* Academic Press, NY, 1978.

[85] P. Herman. Tcpstat tool. URL `http://www.frenchfries.net/paul/tcpstat/`, 2001. Last visited on 18/12/2013.

[86] R. Holanda, J. Verdu, J. Garcıa, and M. Valero. Performance Analysis of A New Packet Trace Compressor Based on TCP Flow Clustering. *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 219–225, 2005.

[87] W. Hon, M. Patil, R. Shah, and S.-B. Wu. Efficient index for retrieving top-k most frequent documents. *Journal of Discrete Algorithms*, 8(4):402–417, 2010.

[88] W. Hon, R. Shah, and J. S. Vitter. Space-efficient framework for top-k string retrieval problems. In *Proc. 50th FOCS*, pages 713–722, 2009.

[89] D. Huffman. A method for the construction of minimum-redundancy codes. *Proc. of the IRE*, 40(9):1098–1101, Sept.

[90] T. Hussain, S. Asghar, and N. Masood. Web usage mining: A survey on preprocessing of web log file. In *Proc. of the Information and Emerging Technologies conference (ICIET)*, pages 1–6, 2010.

[91] B. Iwona and R. Grossi. Rank-sensitive data structures. In *Proc. 11th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 79–90, 2005.

[92] M. Petri J. Culpepper and F. Scholer. Efficient in-memory top-k document retrieval. In *Proc. 35th International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 225–234, 2012.

[93] G. Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, SFCS '89, pages 549–554, 1989.

[94] V. Jacobson, C. Leres, and S. McCanne. Tcpdump. URL `http://www.tcpdump.org/`, 2003. Last visited on 18/12/2013.

[95] K. Jones, S. Walker, and S. Robertson. A probabilistic model of information retrieval: development and comparative experiments: Part 2. *Information Processing & Management*, 36(6):809–840, 2000.

[96] A. Kane and F. Tompa. Skewed partial bitvectors for list intersection. In *Proc. of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '14, pages 263–272, 2014.

[97] J. Karkkainen, D. Kempa, and S. Puglisi. Hybrid compression of bitvectors for the fm-index. In *Proceeding of the Data Compression Conference (DCC)*, pages 302–311, 2014.

[98] Y Kim, R. Konow, D. Dujovne, T. Turletti, W. Dabbous, and G. Navarro. PcapWT: An efficient packet extraction tool for large volume network traces. *Computer Networks*, 79:91–102, 2015.

[99] R. Konow and G. Navarro. Dual-sorted inverted lists in practice. In *Proc. 19th SPIRE*, LNCS 7608, pages 295–306, 2012.

[100] R. Konow and G. Navarro. Faster compact top-k document retrieval. In *Proc. 23rd Data Compression Conference (DCC)*, pages 351–360, 2013.

[101] R. Konow, G. Navarro, C. Clarke, and A. López-Ortíz. Faster and smaller inverted indices with treaps. In *Proc. 36th Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 193–202, 2013.

[102] N. J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proc. of the IEEE*, 88(11):1722–1732, 2000.

[103] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, G. Marth N. Homer, and G. Abecasis R. Durbin. The Sequence Alignment/Map Format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[104] J. Lin and A. Trotman. Anytime ranking for impact-ordered indexes. In *Proceedings of the ACM SIGIR International Conference on the Theory of Information Retrieval (ICTIR)*, pages 25–36, 2015.

[105] T. Liu. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval*, 3(3):225–331, 2009.

[106] V. Mäkinen and G. Navarro. Position-restricted substring searching. In *Proceedings of the 7th Latin American Theoretical Informatics Symposium (LATIN'06)*, LNCS 3887, pages 703–714, 2006.

[107] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[108] C. Martínez and S. Roura. Randomized binary search trees. *J. ACM*, 45(2):288–323, 1997.

[109] E. McCreight. Priority search trees. *SIAM J. Comp.*, 14(2):257–276, 1985.

[110] B. Mobasher. Data mining for web personalization. In *The adaptive web*, pages 90–135. 2007.

[111] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, 1996.

[112] B. Prosenjit H. Meng M. Aniland P. Morin. Succinct orthogonal range search structures on a grid with applications to text indexing. In *Proc. of the 11th International Symposium on Algorithms and Data Structures, WADS*, pages 98–109, 2009.

[113] I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1180, pages 37–42, 1996.

[114] I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computation*, 31(3):762–776, 2002.

[115] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '02, pages 657–666, 2002.

[116] G. Navarro. Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. *ACM Computing Surveys*, 46(4):article 52, 2014. 47 pages.

[117] G. Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

[118] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.

[119] G. Navarro and Y. Nekrich. Top-k document retrieval in optimal time and linear space. In *Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1066–1078, 2012.

[120] G. Navarro, Y. Nekrich, and L. Russo. Space-efficient data-analysis queries on grids. *Theoretical Computer Science*, 482:60–72, 2013.

[121] G. Navarro and E. Providel. Fast, small, simple rank/select on bitmaps. In *Proc 13th International Symposium on Experimental Algorithms (SEA)*, pages 295–306, 2012.

[122] G. Navarro and S. Puglisi. Dual-sorted inverted lists. In *Proceedings of the 17th international conference on String processing and information retrieval (SPIRE)*, LNCS 6393, pages 309–321, 2010.

[123] G. Navarro, S. J. Puglisi, and J. Sirén. Document retrieval on repetitive collections. In *Proc. 22nd Annual European Symposium on Agorithms (ESA B)*, LNCS 8737, pages 725–736, 2014.

[124] G. Navarro, S. J. Puglisi, and D. Valenzuela. General document retrieval in compact space. *ACM Journal of Experimental Algorithmics*, 19(2):article 3, 2014. 46 pages.

[125] G. Navarro and D. Valenzuela. Space-efficient top-k document retrieval. In *Proc. 11th SEA*, pages 307–319, 2012.

[126] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 8th ALENEX*, 2007.

[127] G. Ottaviano and R. Venturini. Partitioned elias-fano indexes. In *Proceedings of the 37th international ACM SIGIR conference on Research and development in information retrieval*, pages 273–282, 2014.

[128] R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31(2):353–363, 2001.

[129] M. Patil, S. Thankachan, R. Shah, W. Hon, J. Vitter, and S. Chandrasekaran. Inverted indexes for phrases and strings. In *Proc. of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, pages 555–564. ACM, 2011.

[130] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, and Hua Zhu. Mining access patterns efficiently from web logs. In *Knowledge Discovery and Data Mining. Current Issues*

and New Applications, pages 396–407. Springer, 2000.

[131] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, 1996.

[132] M. Petri, S. Culpepper, and A. Moffat. Exploring the magic of WAND. In *Proceedings of the rhe Australasian Document Computing Symposium (ADCS)*, pages 58–65, 2013.

[133] P. Politopoulos, E. Markatos, and S. Ioannidis. Evaluation of Compression of Remote Network Monitoring Data Streams. In *Proc. of IEEE Network Operations and Management Symposium (NOMS) Workshops*, pages 109–115, 2008.

[134] E. Providel. Efficient solutions for select operations in binary sequences. MSc. Thesis, 2011. Department of Computer Science, University of Chile.

[135] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (SODA'02)*, pages 233–242, 2002.

[136] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorthms*, 3(4), 2007.

[137] F. Reza. *An Introduction to Information Theory*. Dover Books on Mathematics Series. 1961.

[138] R. Rice and J. Plaunt. Adaptive variable-length coding for efficient compression of spacecraft television data. *IEEE Transactions on Communication Technology*, 19(6):889 –897, 1971.

[139] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *ACM Journal on Algoritms*, 48(2):294–313, 2003.

[140] K. Sadakane. Compressed suffix trees with full functionality. *Theoretical Computer Systems*, 41(4):589–607, 2007.

[141] K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 134–149, 2010.

[142] Kunihiko Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Algorithms*, 5(1):12–22, 2007.

[143] D. Salomon. *Data Compression: The Complete Reference*. Springer Verlag, 4th edition, 2007.

[144] S.Álvarez-García, N. Brisaboa, J. Fernández, M. Martínez-Prieto, and G. Navarro. Compressed vertical partitioning for efficient RDF management. *Knowledge and Information Systems*, 44(2):439–474, 2015.

[145] P. Sanders and F. Transier. Intersection in integer inverted indices. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

[146] S. Nadiand M. Saraee and M. Davarpanah-Jazi. A fuzzy recommender system for dynamic prediction of user's behavior. In *Proc. ICITST*, pages 1–5, 2010.

[147] F. Scholer, H. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. 25th SIGIR*, pages 222–229, 2002.

[148] R. Seidel and C. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[149] E. Shareghi, M. Petri, G. Haffari, and T. Cohn. Compact, efficient and unlimited capacity: Language modeling with compressed suffix trees. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing, (EMNLP)*, pages 2409–2418, 2015.

[150] A. Spink, D. Wolfram, M. Jansen, and S. Tefko. Searching the web: The public and their queries. *Journal of the Society of Information Science Technology*, 52(3):226–234, 2001.

[151] A. Stepanov, A. Gangolli, D. Rose, R. Ernst, and P. Oberoi. Simd-based decoding of posting lists. In *Proc. of the 20th ACM International Conference on Information and Knowledge Management*, pages 317–326, 2011.

[152] T. Strohman and B. Croft. Efficient document retrieval in main memory. In *Proc. 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 175–182, 2007.

[153] Z. Su, Q. Yang, Y. Lu, and H. Zhang. Whatnext: A prediction system for web requests using n-gram sequence models. In *Proc. of the First International Conference on Web Information Systems Engineering (WISE)*, pages 214–224, 2000.

[154] C. Sumathi, R. Valli, and T. Santhanam. Automatic recommendation of web pages in web usage mining. *Intl. J. Comp. Sci. Eng.*, 2:3046–3052, 2010.

[155] W. Szpankowski. A generalized suffix tree and its (un)expected asymptotic behaviors. *SIAM Journal on Computing*, 22(6):1176–1198, 1993.

[156] W. Tobler. A computer movie simulating urban growth in the detroit region. *Economic Geography*, 46(2):234–240, 1970.

[157] A. Trotman. Compression, simd, and postings lists. In *Proceedings of the 2014 Australasian Document Computing Symposium (ADCS)*, pages 50:50–50:57, 2014.

[158] A. Trotman, X. Jia, and M. Crane. Towards an efficient and effective search engine. In *SIGIR 2012 Workshop on Open Source Information Retrieval*, pages 40–47, 2012.

[159] H. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Information*

Processing Management, 31(6):831–850, 1995.

[160] S. Vigna. Broadword implementation of rank/select queries. In *Experimental Algorithms*, pages 154–168. Springer, 2008.

[161] S. Vigna. Quasi-succinct indices. In *Proc. of the Sixth ACM International Conference on Web Search and Data Mining (WSDM)*, pages 83–92, 2013.

[162] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.

[163] L. Wang, J. Lin, and D. Metzler. A cascade ranking model for efficient ranked retrieval. In *Proc. 34th International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 105–114, 2011.

[164] P. Weiner. Linear pattern matching algorithms. In *Proc. Switching and Automata Theory*, pages 1–11, 1973.

[165] H. Williams and J. Zobel. Compressing integers for fast file access. *ACM Computing Journal*, 42(3):193–201, 1999.

[166] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann, 2nd edition, 1999.

[167] L. Wu, W. Lin, X. Xiao, and Y. Xu. LSII: an indexing structure for exact real-time search on microblogs. In *Proceedings of the 29th IEEE conference on Data Engineering (ICDE)*, pages 482–493, 2013.

[168] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. of the 18th international conference on World wide web (WWW)*, pages 401–410, 2009.

[169] G. Zipf. *Human Behaviour and the Principle of Least Effort*. Addison-Wesley, 1949.

[170] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

[171] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2):art. 6, 2006.

[172] M. Zukowski, H. Sandor, N. Niels, and P. Boncz. Super-scalar ram-cpu cache compression. In *Proc. of the 22nd International Conference on Data Engineering*, pages 59–. IEEE Computer Society, 2006.