Università degli Studi di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica
Settore Scientifico Disciplinare INF/01

Ph.D. Thesis

Space-Efficient Data Structures for Collections of Textual Data

Giuseppe Ottaviano

Supervisor
Prof. Roberto Grossi

Referee
Prof. Rajeev Raman

Referee
Prof. Jeffrey Scott Vitter

---

7 June 2013

Abstract

This thesis focuses on the design of succinct and compressed data structures for collections of string-based data, specifically sequences of semi-structured documents in textual format, sets of strings, and sequences of strings. The study of such collections is motivated by a large number of applications both in theory and practice.

For textual semi-structured data, we introduce the concept of *semi-index*, a succinct construction that speeds up the access to documents encoded with textual semi-structured formats, such as JSON and XML, by storing separately a compact description of their parse trees, hence avoiding the need to re-parse the documents every time they are read.

For string dictionaries, we describe a data structure based on a *path decomposition* of the compacted trie built on the string set. The tree topology is encoded using succinct data structures, while the node labels are compressed using a simple dictionary-based scheme. We also describe a variant of the path-decomposed trie for *scored string sets*, where each string has a score. This data structure can support efficiently *top-k completion queries*, that is, given a string p and an integer k, return the k highest scored strings among those prefixed by p.

For sequences of strings, we introduce the problem of *compressed indexed sequences of strings*, that is, representing indexed sequences of strings in nearly-optimal compressed space, both in the static and dynamic settings, while supporting supports random access, searching, and counting operations, both for exact matches and prefix search. We present a new data structure, the *Wavelet Trie*, that solves the problem by combining a Patricia trie with a wavelet tree. The Wavelet Trie improves on the state-of-the-art compressed data structures for sequences by supporting a dynamic alphabet and prefix queries.

Finally, we discuss the issue of the practical implementation of the succinct primitives used throughout the thesis for the experiments. These primitives are implemented as part of a publicly available library, *Succinct*, using state-of-the-art algorithms along with some improvements.

Contents

1 Introduction

String-based data structures lie at the core of most information retrieval, data mining, and database applications. One large-scale example is given by modern search engines and social networks; due to their centralized nature, these services must store, process, and serve a continuous massive stream of user-produced data, which mostly consists of collections of words, texts, URLs, queries, user ids, semi-structured metadata, and relations among them. These massive amounts of data must be made available to a vast number of users with query latencies in the order of milliseconds. At the same time, a large part of the users' interactions with the services is logged to be later analyzed; mining large databases of user actions enables discovery of usage patterns that can be used to improve the quality of the service. At the other end of the spectrum, mobile devices ship with increasingly large databases stored in memory. For instance, rich language models, which are basically scored string sets, are essential to provide accurate speech and handwritten text recognition, and predictive text entry.

In all these applications, space-efficiency is crucial. In order to answer queries with real-time latency guarantees, the data must be stored in the highest levels of the memory hierarchy: in many cases, it is not possible to afford a single disk access, let alone a network request. Besides, the higher the space efficiency, the more data can be stored with the same memory budget, which is the main cost in modern systems.

Traditionally, space efficiency is obtained by careful engineering of data structures, usually combined with general-purpose compression algorithms such as gzip. However, classical pointer-based data structures often have unavoidable overheads, for as clever the space optimizations might be. For example, a tree of n nodes needs $\Omega(\log n)$ bits per node to store the children pointers; as the tree grows larger, the size of the pointers can become much larger than the data stored in the node. Regarding general-purpose compression, most compression schemes can only be decompressed sequentially. To apply them to data structures, it is necessary to divide the data into blocks and compress them individually. The smaller are the blocks, the poorer is the compression ratio. On the other hand, if the blocks are too large the cost of decompressing the whole block to retrieve a single record becomes dominant. This is particularly inefficient when the data access patterns are highly non-sequential.

The field of *succinct data structures* promises to solve the aforementioned issues: a succinct data structure guarantees space bounds equal to the information-theoretic lower bound on the space needed to store the data, plus a second-order *negligible* term, which is usually called the *redundancy*. Still, a large set of operations can be supported with time complexity competitive with the equivalent classical data structures. A tree of n nodes, for example, can be stored in $2n + \mathrm{O}(n/\mathrm{polylog}\, n) = 2n + o(n)$ bits [101], while supporting in constant time a set of traversal operations even larger than a standard representation that stores children and parent pointers in each node, which would need $\Omega(n \log n)$ bits. Similarly, a text can be compressed down to its high-order entropy plus negligible terms, and still any character in the text can be accessed in constant time [100, 54, 42].

This thesis focuses on the design of succinct and compressed data structures for collections of string-based data, specifically sequences of semi-structured documents in textual format, sets of strings, and sequences of strings. We have chosen these particular types of data as they arise in a large number of practical scenarios, as outlined above.

Since our problems come from real-world applications, we give special attention to the practicality of the presented data structures, without however neglecting the theoretical guarantees. Succinct data structures have arisen mainly as a theoretical tool; nonetheless, a large amount of recent literature has shown that in many applications they can have performance competitive with classical data structures, while offering a significant reduction in space. The recent interest in succinct data structures for practical applications can be attributed to a few recent trends. First, the performance improvement of CPUs proceeds at a much higher pace than that of memory, making it convenient to trade more computation for fewer memory accesses. Second, 64-bit architectures are now ubiquitous, hence if the low-level primitives used in succinct data structures are implemented by means of *broadword algorithms* [76], double the bits can be processed in roughly the same time when compared to 32-bit architectures, thus greatly improving the efficiency. Lastly, data sizes are growing to a point where the $\Omega(\log n)$ overhead of pointer-based data structures becomes a bottleneck.

While most data structures benefit from some degree of algorithm engineering, for succinct data structures it becomes a necessary effort, as the asymptotic gains can be easily outweighted by the constants hidden in the big-Oh notation, with the result that the "negligible" redundancy, for realistic data sizes, is not negligible at all! On real datasets, an $\mathrm{O}(n)$ space data structure can be more space-efficient than an $o(n)$ one, and similarly a carefully optimized logarithmic-time algorithm can easily beat a constant-time one. For this reason, a chapter of this thesis is devoted to the practical implementation of succinct primitives, to investigate the impact of the theoretical ideas in a practical framework.

1.1 Thesis organization

The thesis is structured as follows. After a brief review of the basic concepts and tools, we present our contributions, which are summarized below. We then conclude the thesis with some directions for future work and open problems.

Semi-indexing textual semi-structured data. Semi-structured textual formats such as XML and JSON are gaining increasing popularity for the storage of document collections and rich logs. Their flexibility comes at the cost of having to load and parse a document entirely even if just a small part of it needs to be accessed. For instance, in data analytics massive collections are usually scanned sequentially, selecting a small number of attributes from each document.

In Chapter 3 we propose a technique to attach to a raw, unparsed document (even in compressed form) a "semi-index": a succinct data structure that supports operations on the document tree at speed comparable with an in-memory deserialized object.

After describing the general technique, we focus on the JSON format: our experiments show that avoiding the full loading and parsing step can give speed-ups of up to 10 times for on-disk documents using a small space overhead. The contents of this chapter are based on [97].

Compressed string dictionaries. In Chapter 4 we explore new succinct representations of tries, based on path-decomposition trees, and experimentally evaluate the corresponding reduction in space usage and running times, comparing with the state of the art. We study the following applications.

(1) Compressed string dictionary: We obtain data structures that outperform other state-of-the-art compressed dictionaries in space efficiency, while obtaining predictable query times that are competitive with data structures usually preferred by the practitioners. On real-world datasets our compressed tries obtain the smallest space (except for one dataset) and have the fastest lookup times, while retrieval times are within 20% slower than the best known solutions.

(2) Monotone minimal perfect hash for strings: Our compressed tries perform several times faster than other trie-based monotone perfect hash functions, while occupying nearly the same space. On real-world datasets our tries are approximately 2–5 times faster than previous solutions, with a space occupancy less than 10% larger.

The contents of this chapter are based on [60].

Top-k string completion. In Chapter 5 we present an application of the tries described in Chapter 4 to the problem of top-k string completion, that is, given a scored string set, the problem of retrieving the k highest scored strings that match a given prefix. This problem arises in several applications, such as auto-completion of search engine queries and predictive text entry in mobile devices.

We show that a path-decomposition strategy specifically tailored for the problem enables fast retrieval of top-k completions, while retaining the space reduction obtained with our compressed tries. The contents of this chapter are partly based on [65].

INDEXED SEQUENCES OF STRINGS. An *indexed sequence of strings* is a data structure for storing a *sequence of strings* that supports random access, searching, range counting and analytics operations, both for exact matches and prefix search. String sequences lie at the core of column-oriented databases, log processing, and other storage and query tasks. In these applications each string can appear several times and the order of the strings in the sequence is relevant. The prefix structure of the strings is relevant as well: common prefixes are sought in strings to extract interesting features from the sequence.

In Chapter 6 we introduce and study the problem of *compressed indexed sequence of strings*, i.e. representing indexed sequences of strings in nearly-optimal compressed space, both in the static and dynamic settings, while preserving provably good performance for the supported operations.

We present a new data structure for this problem, the *Wavelet Trie*, which combines the classical Patricia trie with the wavelet tree, a succinct data structure for storing compressed sequences. The resulting Wavelet Trie smoothly adapts to a sequence of strings that changes over time. It improves on the state-of-the-art compressed data structures by supporting a dynamic alphabet (i.e. the set of distinct strings) and prefix queries, both crucial requirements in the aforementioned applications, and on traditional indexes by reducing space occupancy to close to the entropy of the sequence. The contents of this chapter are based on [61].

PRACTICAL IMPLEMENTATION OF SUCCINCT DATA STRUCTURES. In Chapter 7 we discuss our implementation of the basic static succinct primitives such as Rank/Select bitvectors, balanced parentheses and sparse vectors, that form the basic building blocks of the data structures presented in chapters 3, 4, and 5.

In particular, for balanced parentheses we introduce the *Range-Min tree*, a variation of the Range-Min-Max tree [4, 101] which obtains faster operations times while occupying half the space.

In this chapter we also describe the *Succinct* library [107], a publicly available C++ library which contains all our implementations of the aforementioned succinct primitives, and discuss the engineering trade-offs and design principles made while writing the library. This library was used in all the experimental implementations of the data structures presented in this thesis.

2 Background and tools

In this chapter we briefly summarize the notation, tools and techniques that we will use throughout the thesis.

2.1 Basic notation

Given a set S, we will denote its *cardinality* with |S|. In order to represent ranges of integers, we will use $[n]$, where n is a natural number, to denote the set of the first n natural numbers, i.e. $\{0, \ldots, n-1\}$. When ambiguity can arise, we will use the alternative notation $[0, n)$. Note that $|[n]| = |[0, n)| = n$. All the logarithms will be in base 2, i.e. $\log(x) = \log_2(x)$, unless otherwise specified.

2.2 Sequences, strings, and bitvectors

Central to this thesis is the concept of *sequence*. In fact, it is so central that it will be given several names, such as *sequence*, *string*, *text*, *array*, *vector*, *list*. While all mathematically equivalent, each term has different nuances. For example, *sequence* is the most abstract. *Arrays* and *vectors* will usually be made up of numbers, while a *string*'s elements will be *characters*, for some meaningful definition of *character*; a *text* will just be a "long" string. A *list* is something meant to be iterated sequentially. Furthermore, in Chapter 6 we will make extensive use of *sequences of strings*; the term *sequences of sequences* would have been significantly more cumbersome. For this reasons, we will use these synonims depending on the context, trading some minimality for clarity.

A (finite) sequence of length n can be informally defined as an ordered collection of elements $s = s_0, \ldots, s_{n-1}$. A more formal and non-circular (although somewhat abstract) definition can be given as follows.

Definition 2.2.1 Let Σ be a finite set, $|\Sigma| = \sigma$, called the *alphabet*. Given a natural n, a *sequence s of length n drawn from* Σ is a function $s : [n] \to \Sigma$, defined as $i \mapsto s_i$. We will call s_i the *i-th element* of s, and $|s| = n$ the *length* of s.

Contrary to most literature which uses 1-based indices, all our definitions will be 0-based; this way, arithmetic expressions involving sequence indices are usually simpler.

To avoid confusion, we slightly abuse the term i-th to indicate s_i, so the 0-th element of a sequence will be the first and the $(n-1)$-th the last.

The set of sequences of length n drawn from Σ is usually denoted as Σ^n; Σ^0 is the singleton set of the empty sequence ϵ. Similarly, the set of strings of arbitrary length is denoted as $\Sigma^* = \bigcup_{i \in \mathbb{N}} \Sigma^i$. The elements of the alphabet Σ are usually called *symbols*.

An alternative notation for s_i is $s[i]$, which can be convenient when the subscript notation is ambiguous (for example when collections of sequences are involved). To refer to a contiguous range of a sequence, spanning from the i-th element to the $j-1$-th element, we will use the notation $s[i, j)$ or $s[i, j-1]$, depending on which one is more convenient. Note that $s[0, |s|) = s$, and $|s[i, j)| = j - i$.

BITVECTORS. A special case of particular importance is when Σ is $\{0, 1\}$. In this case the sequence is said to be binary, and called a *bitvector*. It will be represented as follows.

$$s = 0000111010010110000011100000111011110110$$

In this example, $|s| = 40$, $s_0 = 0$, and $s_4 = 1$.

STRINGS. A sequence drawn from an alphabet Σ of *characters*, for example from a natural language, it will be usually called a *string*, and is represented as follows.

$$s = \texttt{senselessness}$$

In this example, $|s| = 13$, $s_1 = \texttt{e}$, and $s_{12} = \texttt{s}$. For a string s we can define its *prefix* of length $\ell \leq |s|$ as $s[0, \ell)$. We say that p is a prefix of s if and only if $s[0, |p|) = p$.

ARRAYS. A sequence drawn from an alphabet Σ of *numbers* will be usually called an *array*, and is represented as follows.

$$s = \langle 4, 8, 16, 23, 42 \rangle$$

In this example, $|s| = 5$, $s_0 = 4$, and $s_3 = 23$.

SEQUENCES OF STRINGS. Lastly, a sequence $\mathrm{S} \in \Sigma^{**}$ drawn from an alphabet of strings Σ^*, which are drawn from an alphabet Σ, will be called a *sequence of strings*, and represented as follows.

$$\mathrm{S} = \langle \texttt{foo}, \texttt{bar}, \texttt{baz}, \texttt{foo}, \texttt{baz} \rangle$$

Note that in this case we use a capital S, to avoid confusion with strings that will be denoted as s. In this example, $|\mathrm{S}| = 5$, $\mathrm{S}_0 = \texttt{foo}$, and $\mathrm{S}_4 = \texttt{baz}$.

BINARY REPRESENTATION OF SEQUENCES. In many situations it will be useful to represent a sequence s drawn from an arbitrary alphabet Σ as a binary sequence. In order to do so, it is sufficient to map Σ injectively to the set of binary sequences of length $\lceil \log |\Sigma| \rceil$. The binary sequence a symbol maps to is called the *binary encoding* of the symbol. The *binary representation* of s is defined as the concatenation of the binary encodings of its symbols; its length is $|s| \lceil \log |\Sigma| \rceil$ bits.

2.2.1 *Sequence operations*

We now define some operations that can be performed on sequences. In particular, as we will see in the following sections, Rank and Select are powerful primitives that form the cornerstone of succinct data structures.

In the following, s is an arbitrary sequence. In the examples, it will be the following string.

$$s = \underset{0}{\texttt{s}}\ \underset{1}{\texttt{e}}\ \underset{2}{\texttt{n}}\ \underset{3}{\texttt{s}}\ \underset{4}{\texttt{e}}\ \underset{5}{\texttt{l}}\ \underset{6}{\texttt{e}}\ \underset{7}{\texttt{s}}\ \underset{8}{\texttt{s}}\ \underset{9}{\texttt{n}}\ \underset{10}{\texttt{e}}\ \underset{11}{\texttt{s}}\ \underset{12}{\texttt{s}}$$

The operations are defined as follows.

— $\text{Access}_s(i)$ retrieves the i-th element s_i. For example, $\text{Access}_s(2) = \texttt{n}$.

— $\text{Rank}_s(c, i)$ with $c \in \Sigma$ returns the number of occurrences of the symbol c in $s[0, i)$, i.e. those preceding position i. For example, $\text{Rank}_s(\texttt{s}, 0) = 0$, $\text{Rank}_s(\texttt{s}, 1) = 1$, and $\text{Rank}_s(\texttt{e}, 5) = 2$.

— $\text{Select}_s(c, i)$ returns the position of the i-th occurrence of the symbol c in s. For example, $\text{Select}_s(\texttt{e}, 0) = 1$, $\text{Select}_s(\texttt{e}, 1) = 4$, and $\text{Select}_s(\texttt{s}, 0) = 0$.

— $\text{Predecessor}_s(c, i)$ returns the position of the *rightmost* occurrence of c preceding or equal to i. For example, $\text{Predecessor}(\texttt{e}, 1) = 1$, $\text{Predecessor}(\texttt{e}, 2) = 1$, and $\text{Predecessor}(\texttt{l}, 12) = 5$.

Note that $\text{Rank}_s(c, \text{Select}_s(c, i)) = i$ and $\text{Select}_s(c, \text{Rank}_s(c, i)) = \text{Predecessor}_s(c, i)$.

When the subscript s can be omitted without ambiguity, we will use the alternative notation $\text{Access}(i)$, $\text{Rank}_c(i)$, $\text{Select}_c(i)$, and $\text{Predecessor}_c(i)$.

2.3 INFORMATION-THEORETICAL LOWER BOUNDS

In order to reason about the theoretical performance of space-efficient data structures, it will be useful to compare their space occupancy with lower bounds on the number of bits needed to represent the data. In this section we summarize the most common lower bounds, derived from information theory, for the combinatorial objects we will analyze, namely sequences, subsets, and trees.

SEQUENCES. If we do not have any prior information on the sequence s except for its alphabet Σ and its length n, then for any encoding scheme there will be at least one sequence that needs at least $n\lceil\log|\Sigma|\rceil$ bits, as a simple consequence of the pigeonhole principle. This number can then be considered as a *worst-case* lower bound on the space needed to represent a sequence. Naturally, this is hardly satisfying, as it is the space taken by the trivial binary representation. To obtain more realistic bounds, a solution is to adopt a *data-dependent* measure of the *compressibility* of a sequence, meaning that the measure is computed on a specific instance of the data.

A natural candidate would be the *Kolmogorov complexity* [82], which is defined as the minimum length among the encodings of the Turing machines that output a given sequence. However, it is impossible to compute such measure on a specific sequence, making it hard to derive effective bounds.

A much more fruitful definition was given by Shannon in his seminal paper on information theory [105]. Shannon was concerned with defining a notion of *information content* of a discrete random variable X ranging on a finite set Σ, whose distribution can be written as $(p_c)_{c\in\Sigma}$. After proposing a set of reasonable axioms, he concluded that the only function that would satisfy those axioms was

$$\mathrm{H}(\mathrm{X}) = -\sum_{c\in\Sigma} p_c \log(p_c),$$

which he called the *entropy* of X.

Given a sequence s we can define the *0th-order empirical entropy* of s as

$$\mathrm{H}_0(s) = -\sum_{c\in\Sigma} \frac{n_c}{n} \log \frac{n_c}{n},$$

where n_c is the number of occurrences of symbol c in s. If s is a binary sequence, and p is the fraction of 1s in s, we can rewrite the entropy as $\mathrm{H}_0(s) = -p\log p - (1-p)\log(1-p)$, which we also denote by $\mathrm{H}(p)$.

$\mathrm{H}_0(s)$ can be interpreted as an estimate of $\mathrm{H}(\mathrm{X})$ assuming that the elements of s are i.i.d. draws from the variable X; in this case, $p_c = n_c/n$ are the *empirical probabilities*. More importantly for our purposes, it can be shown [25] that $n\mathrm{H}_0(s)$ is a lower bound on the number of bits needed to encode s with an encoder that maps each symbol of Σ to a fixed binary string, thus independently of its position in s.

Building on H_0 we can define the *kth-order empirical entropy* H_k as follows. For $u \in \Sigma^k$, we define s_u as the sequence of the symbols that follow each occurrence of u in s. Then H_k is defined as

$$\mathrm{H}_k(s) = \sum_{u\in\Sigma^k} \frac{|s_u|}{n} \mathrm{H}_0(s_u),$$

and, similarly to H_0, $n\mathrm{H}_k(s)$ is a lower bound on the number of bits needed to encode s with an encoder that can choose the encoding of a symbol based on its k-context,

i.e. the k symbols preceding it. This notion is extremely useful in the analysis of text compressors [86]. However, in this thesis we will mostly be concerned with H_0.

SUBSETS. Let U be a set of size n, called the *universe*. A lower bound to the number of bits needed to store a subset X of U of size $|X| = m$ is the logarithm of the number of such subsets, that is $\mathscr{B}(m,n) = \lceil \log \binom{n}{m} \rceil$.

This number is linked to the entropy by the bound $\mathscr{B}(m,n) \leq nH(\frac{m}{n}) + O(1)$, which can be intuitively justified by noting that a subset S of m elements from an universe of n elements can be represented as a bitvector of length n with m ones, called the *characteristic function* of S.

In the following, we will implicitly make extensive use of the previous bound, as well as the bound $\mathscr{B}(m,n) \leq m \log(\frac{n}{m}) + O(m)$.

ORDINAL TREES. An *ordinal tree* is a rooted tree where the order of the children of each node is specified, so that it is possible to define the i-th child of a node, for i smaller than the node degree. The number of ordinal trees on n nodes is given by the *Catalan number* $\mathscr{C}_n = \frac{1}{n+1}\binom{2n}{n}$ [58]. By using the Stirling approximation it is easy to derive the approximation $\mathscr{C}_n \approx 4^n/(\pi n)^{3/2}$, which implies that $\log \mathscr{C}_n = 2n - \Theta(\log n)$. In other words, asymptotically, a tree needs at least 2 bits per node to be represented.

2.4 MODEL OF COMPUTATION

In order to analyze the time complexity of algorithms, we will need an abstract model of computation, which defines a set of elementary operations; the computational cost of an algorithm is then the number of elementary operations performed during its execution.

The *word-RAM* model tries to emulate closely the operations available on a realistic CPU. Specifically, the basic unit of memory is the *word*, a binary integer of w bits, and any unit of memory can be accessed in constant time. The operations defined are the familiar arithmetic and logical operations on words, including bitwise shift and boolean operations, and integer multiplication and division. We will represent such operations with the operators of the ubiquitous C programming language.

We will assume that $w = \Omega(\log n)$, where n is the size of the problem. Otherwise, the address of an arbitrary position of the input would not fit into a constant number of memory words, hence this assumption does not result in a significant loss of generality. Often a stronger assumption is made, namely that $w = \Theta(\log n)$, i.e. the word size is bounded by a multiple of the logarithm of the problem size. This assumption is often called *transdichotomous assumption* [48], because it breaks the dichotomy between the problem and the model of computation, tying the characteristics of the abstract machine to the size of the problem. However, in our work we will not need this assumption.

While the word-RAM model is reasonably realistic, it does not take into account some characteristics of a modern machine. For example, it assumes constant time access

to *any* location of memory, ignoring the latency of memory transfers. Other models of computation study the complexity of an algorithm when a non-uniform memory hierarchy is considered by counting the memory transfers across different levels of the hierarchy. The most prominent example is the *external-memory model* [113], which models the machine as a two-level memory hierarchy, with a "fast" memory of size M and a "slow" memory of unbounded size; the machine can transfer contiguous blocks of size B across the two levels, and the I/O complexity of an algorithm is defined as the number of such memory transfers (also called I/Os) throughout its execution. Also widely used is the *cache-oblivious model* [49], a variation of the external-memory model where M and B are *unknown* to the algorithm.

We will perform the analysis of our proposed algorithms using the word-RAM model, following the common practice adopted for the analysis of many succinct data structures. However, whenever necessary, we will also make informal considerations on the number of memory transfers to explain the empirical performance.

2.5 Succinct representations of sequences

A *succinct data structure* is a data structure that attains a space bound equal to the information-theoretical lower bound, plus a second-order, *negligible* term. At the same time, efficient operations are supported, with time comparable or equal to classical data structures for the same purpose. Succinct data structures were introduced by Jacobson in his seminal paper [68], where he presented a data structure to encode a bitvector of length n with $n + O(n \log\log n / \log n)$ bits, while supporting Access and Rank operations in constant time. The data structure was later improved by Clark [19], who showed how to implement Select in constant time as well.

2.5.1 *Fully indexable dictionaries*

Jacobson's result was later improved by Raman et al. [99], who introduced the notion of Fully Indexable Dictionary (FID), a data structure, which we refer to as RRR, to encode a set of m elements from an universe of size n in $\mathcal{B}(m,n) + O((n \log\log n)/\log n)$ bits. Thanks to the correspondence mentioned above, the data structure can be used to encode bitvectors of length n with m ones with the same space bound, while supporting Access, Rank, and Select in constant time.

2.5.2 *Elias-Fano representation*

The *Elias-Fano representation of monotone sequences* [32, 34] is an encoding scheme to represent a non-decreasing sequence of m integers $\langle x_1, \cdots, x_m \rangle$ from the universe $[0..n)$ occupying $2m + m \left\lceil \log \frac{n}{m} \right\rceil + o(m)$ bits, while supporting constant-time access to the i-th integer.

The scheme is very simple and elegant. Let $\ell = \lfloor \log(n/m) \rfloor$. Each integer x_i is first encoded in binary into $\lceil \log n \rceil$ bits, and the binary encoding is then split into the first $\lceil \log n \rceil - \ell$ *higher* bits and the last ℓ *lower* bits. The sequence of the higher bits is represented as a bitvector H of $\lceil m + n/2^\ell \rceil$ bits, where for each i, if h_i is the value of the higher bits of x_i, then the position $h_i + i$ of H is set to `1`; H is `0` elsewhere. The lower bits of each x_i are just concatenated into a bitvector L of $m\ell$ bits.

To retrieve the i-th integer we need to retrieve its higher and lower bits and concatenate them. The lower bits are easily retrieved from L. To retrieve the upper bits it is sufficient to note that $h_i = \text{Select}_{\text{H}}(\texttt{1}, i) - i$.

This data structure can be used to represent sparse bitvectors (i.e. where the number m of ones is small with respect to the size n of the bitvector), by encoding the sequence of the positions of the ones. Using this representation the retrieval of the the i-th integer can be interpreted as $\text{Select}_1(i)$, and similarly it is possible to support Rank_1.

2.5.3 *Wavelet trees*

The wavelet tree, introduced by Grossi et al. [59], is a data structure to represent sequences on an arbitrary alphabet Σ in compressed space, while supporting efficient Access/Rank/Select operations. The wavelet tree reduces the problem of storing a sequence to the storage of a set of $|\Sigma| - 1$ bitvectors organized in a tree structure.

The alphabet is recursively partitioned into two subsets, until each subset is a singleton (hence the leaves are in one-to-one correspondence with the symbols of Σ). The bitvector β at the root has one bit for each element of the sequence, where β_i is `0`/`1` if the i-th element belongs to the left/right subset of the alphabet. The sequence is then projected on the two subsets, obtaining two subsequences, and the process is repeated on the left and right subtrees. An example is shown in Figure 2.1.

The `0`s of one node are in one-to-one correspondence with the bits of the left node, while the `1`s are in correspondence with the bits of the right node, and the correspondence is given downwards by Rank and upwards by Select. Thanks to this mapping, it is possible to perform Access and Rank by traversing the tree top-down, and Select by traversing it bottom-up.

We briefly describe the Access (Rank and Select are similar): to access the i-th element, we access the i-th bit. If it is `0` we proceed recursively on the left subtree for the position $\text{Rank}(\texttt{0}, i)$, if it is `1` we proceed on the right subtree for the position $\text{Rank}(\texttt{1}, i)$. When we reach a leaf, its symbol is the i-th element of the sequence.

By representing each bitvector β with RRR, the space is $nH_0(S) + o(n \log|\Sigma|)$ bits, while operations take $O(\log|\Sigma|)$ time.

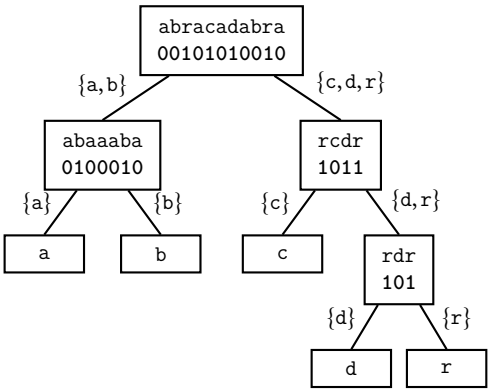


FIGURE 2.1: A wavelet tree for the string **abracadabra** from the alphabet $\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{r}\}$. Only the bitvectors and the tree structure are actually stored, the other information is for illustrative purposes only.

2.6 SUCCINCT REPRESENTATIONS OF TREES

In the same paper where he introduced succinct data structures for Rank and Select [68], Jacobson described the first succinct representation of ordinal trees, called *Level-order unary degree sequence* (LOUDS). A tree is encoded with LOUDS by concatenating the unary representations of the node degrees, namely $\mathtt{1}^d\mathtt{0}$ where d is the degree, in level-order (after adding a fake root). This yields a bitvector of $2n+1$ bits, where n is the number of nodes, and the positions of the zeros can be put in one-to-one correspondence with the nodes of the tree. It can be shown that, if p is the position of a node in the bitvector using this correspondence, then the following tree traversal operations can be reduced to sequence operations on the bitvector.

— $\text{FirstChild}(p) = \text{Select}_0(\text{Rank}_1(p)) + 1$;

— $\text{NextSibling}(p) = p + 1$;

— $\text{Parent}(p) = \text{Select}_1(\text{Rank}_0(p))$.

Hence, using the succinct representation of bitvectors, a tree can be encoded in $2n + o(n)$ bits while supporting constant-time basic traversal operations.

2.6.1 *Balanced parentheses (BP)*

A more powerful succinct representation of trees was given by Munro and Raman [91] by means of balanced parentheses sequences. A sequence of *balanced parentheses* (BP) is inductively defined as follows: an empty sequence is BP; if α and β are sequences of BP, then also $(\alpha)\beta$ is a sequence of BP, where (and) are called *mates*.

For example, the following is a sequence of BP.

$$s = \underset{0\,1\,2\,3\,4\,5\,6\,7\,8\,9}{((()(()()))}$$

These sequences are usually represented as bitvectors, where `1` represents `(` and `0` represents `)`. A sequence s of $2m$ BP can be encoded in $2m + o(m)$ bits [68, 91, 101] so that the following operations, among others, are supported in constant time.

- $\text{FindClose}_s(i)$, for a value i such that $s_i =$ `(`, returns the position $j > i$ such that $s_j =$ `)` is its mate. In the example, $\text{FindClose}_s(0) = 9$ and $\text{FindClose}_s(1) = 2$.
- $\text{FindOpen}_s(i)$, for a value i such that $s_i =$ `)`, returns the position $j < i$ such that $s_j =$ `(` is its mate. In the example, $\text{FindOpen}_s(9) = 0$ and $\text{FindClose}_s(2) = 1$.
- $\text{Enclose}_s(i)$, for a value i such that $s_i =$ `(`, returns the position $j < i$ such that $s_j =$ `(` and the pair of j and its mate enclose the pair of i and its mate. In the example, $\text{Enclose}_s(1) = 0$ and $\text{Enclose}_s(6) = 3$.
- $\text{Rank}_s(\texttt{(}, i)$ returns the pre-order index of the node corresponding to the parenthesis at position i and its mate; this is just the number of open parentheses preceding i.
- $\text{Excess}_s(i)$ returns the difference between the number of open parentheses and that of close parentheses in the first $i + 1$ positions of s. The sequence of parentheses is balanced if and only if this value is always non-negative, and it is easy to show that it equals $2\text{Rank}_s(\texttt{(}, i) - i$.
- $\pm 1\text{RMQ}_S(i, j)$ returns the leftmost position of minimal excess in the range $[i, j]$, i.e. $r \in [i, j]$ such that, for any $r' \in [i, j]$, either $\text{Excess}_S(r) < \text{Excess}_S(r')$, or $\text{Excess}_S(r) = \text{Excess}_S(r')$ and $r \leq r'$.

When no ambiguity is possible, we will drop the subscript s and use the alternative notation $\text{Rank}_{\texttt{(}}(i)$.

A sequence of BP implicitly represents an ordinal tree, where each node corresponds to a pair of mates. By identifying each node with the position p of its corresponding open parenthesis, several tree operations can be reduced to the operations defined above.

- $\text{FirstChild}(p) = p + 1$;
- $\text{NextSibling}(p) = \text{FindClose}(p) + 1$;
- $\text{Parent}(p) = \text{Enclose}(p)$;
- $\text{Depth}(p) = \text{Rank}_{\texttt{(}}(p)$;
- $\text{SubtreeSize}(p) = (\text{FindClose}(p) - p + 1)/2$.

Note that the last two operations cannot be supported with the LOUDS representation. Navarro and Sadakane [101] later introduced more powerful operations on BP sequences, which enable support for a larger set of tree operations in constant time.

2.6.2 *Depth-first unary degree sequence (DFUDS)*

Another tree representation based on balanced parentheses was introduced by Benoit et al. [9]. Called *depth-first unary degree sequence* (DFUDS), it is constructed by concatenating in depth-first order the node degrees encoded in unary with parentheses, i.e. a degree d is encoded as $(^d)$. It can be shown that by prepending an initial `(`, the obtained sequence of parentheses is balanced.

By identifying each node of the tree with the position p of beginning of its degree encoding, tree operations can be mapped to sequence operations as follows.

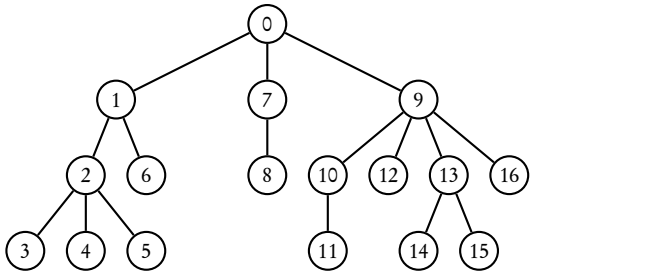
— $\text{Degree}(p) = \text{Select}_)(\text{Rank}_)(p) + 1) - p$;

— $\text{Child}(p, i) = \text{FindClose}(\text{Select}_)(\text{Rank}_)(p) + 1) - i) + 1$;

— $\text{Parent}(p) = \text{Select}_)(\text{Rank}_)(\text{FindOpen}(p - 1))) + 1$.

— $\text{SubtreeSize}(p) = (\text{FindClose}(\text{Enclose}(p)) - p)/2 + 1$

Compared to the BP representation we lose the Depth operation, but we gain the operation Child which returns the i-th child by performing a single FindClose.



LOUDS 10 1110 110 10 11110 1110 0 0 10 0 110 0 0 0 0 0 0 0
(0 1 7 9 2 6 8 10 12 13 16 3 4 5 11 14 15)

BP (((( ) ( ) ( )) ( )) (( )) ((( )) ( ) (( ) ( )) ( )))
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16)

DFUDS ((((() (() ((())))) ()) ((((() ())) (())))
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16)

FIGURE 2.2: LOUDS, BP, and DFUDS encodings of an ordinal tree.

2.6.3 *Representing binary trees*

We define a *binary tree* as a tree where each node is either an *internal node* which has *exactly* 2 children, or a *leaf*. It follows immediately that a binary tree with n internal nodes has $n + 1$ leaves. An example of binary trees is given by binary compacted tries. There is another popular definition of binary trees, used for instance in [91], where each

node is allowed to have *at most* 2 children, but a distinction is made between degree-1 nodes based on whether their child is left or right. The two definitions are equivalent if we consider the internal nodes in the first definition to be the nodes in the second definition, and the leaves to be the *absent children*. Hence, a tree with $2n+1$ nodes in the first definition is a tree with n nodes in the second. Choosing between the two definitions usually depends on whether the leaves have some significance for the application.

A simple way to represent a binary tree would be to see it as a ordinal tree and encode it with BP, taking $4n + o(n)$ bits for a tree with $2n+1$ nodes. It is however possible to do better: as noted in [91], binary trees with $2n+1$ nodes can be bijectively mapped to ordinal trees of n nodes, by considering the n internal nodes as the first-child next-sibling representation of a tree of n nodes; operations on the binary tree can be performed directly on the obtained tree. Such a tree can be encoded with BP, occupying just $2n + o(n)$ bits. Given a tree T we call such encoding FCNS(T).

We show here how the same representation can be obtained directly from the tree, without the conceptual step through the first-child next-sibling mapping. This makes reasoning about the operations considerably simpler. Furthermore, with this derivation the leaves have an explicit correspondence with positions in the parentheses sequence.

DEFINITION 2.6.1 Let T be a binary tree. We define its Depth-First Binary Sequence representation DFBS(T) recursively:

— If T is a leaf, then DFBS(T) =).

— If T is a node with subtrees T_{left} and T_{right}, then

$$\text{DFBS}(T) = (\,\text{DFBS}(T_{left})\,\text{DFBS}(T_{right}).$$

While in FCNS only internal nodes are mapped to (s, in this representation all the $2n+1$ nodes in depth-first order are in explicit correspondence to the $2n+1$ parentheses in the sequence, with (s corresponding to internal nodes and)s corresponding to leaves. This property is especially useful if satellite data need to be associated to the leaves.

The following lemma proves that the obtained sequence is isomorphic to FCNS(T).

LEMMA 2.6.2 For any binary tree T, it holds DFBS(T) = FCNS(T)).

Proof If T is a leaf, FCNS(T) is the empty sequence and DFBS(T) is the single parenthesis), so the lemma follows trivially. Otherwise, let $T_1, T_2, \ldots, T_i$ be the subtrees hanging off the rightmost root-to-leaf path in T; then by definition

$$\text{FCNS}(T) = (\,\text{FCNS}(T_1))\,(\,\text{FCNS}(T_2))\ldots(\,\text{FCNS}(T_i)).$$

By induction, we can rewrite it in the following way

$$\text{FCNS}(T) = (\,\text{DFBS}(T_1)(\,\text{DFBS}(T_2)\ldots(\,\text{DFBS}(T_i).$$

Note that T_1 is just T_{left}, while $\text{DFBS}(T_{\text{right}}) = (\,\text{DFBS}(T_2)\ldots(\,\text{DFBS}(T_i))$, where the last) is given by the final leaf in the rightmost root-to-leaf path. Hence

$$\text{DFBS}(T) = (\,\text{DFBS}(T_1)(\,\text{DFBS}(T_2)\ldots(\,\text{DFBS}(T_i)) = \text{FCNS}(T))$$

proving the inductive step. ∎

The above lemma implies that (DFBS(T) is a sequence of balanced parentheses. To make the construction self-contained, we prove it directly in the following lemma. The proof will also make clear how to perform navigational operations.

LEMMA 2.6.3 The sequence (DFBS(T) is a sequence of balanced parentheses.

Proof If T is a leaf, then (DFBS(T) = (). Otherwise, let

$$(\,\text{DFBS}(T) = ((\,\text{DFBS}(T_{\text{left}})\,\text{DFBS}(T_{\text{right}}).$$

By induction, both $(\,\text{DFBS}(T_{\text{left}})$ and $(\,\text{DFBS}(T_{\text{right}})$ are sequences of balanced parentheses. Inserting the first into the second yields a sequence of balanced parentheses. ∎

Note that internal nodes are of the form $(\,\text{DFBS}(T_{\text{left}})\,\text{DFBS}(T_{\text{right}})$ where T_{left} and T_{right} are the children. The previous lemma allows us to skip the $(\,\text{DFBS}(T_{\text{left}})$ subsequence with a single FindClose, hence the following corollary follows immediately.

COROLLARY 2.6.1 If position i in DFBS(T) corresponds to an internal node, then $\text{LeftChild}(i) = i + 1$ and $\text{RightChild}(i) = \text{FindClose}(i) + 1$.

The DFBS encoding is discussed also by Jansson et al. [69], however without mentioning that it is possible to perform traversal operations on it. They obtain the n-bit bound instead using a compressed version of DFUDS, which is more powerful, but also significantly more complex. Davoodi et al. [26] note that the encoding yields a sequence of balanced parentheses, but they do not mention the equivalence with FCNS.

Using this encoding it is also possible to generate random binary trees of a given size in a very intuitive way: to generate a tree of size $2n+1$, choose an odd number m between 1 and $2n-1$ from some distribution to be the size of the left subtree. Then recursively generate trees of sizes m and $2n-m$, obtaining the sequences α and β. The resulting tree is $(\alpha\beta$. We use this technique to benchmark BP implementations in Chapter 7.

2.7 STRING DICTIONARIES

Given a set of strings $\mathcal{S} \subset \Sigma^*$, a *string dictionary* is a data structure that stores the set $\mathcal{S}$ and supports retrieval of individual strings. Specifically, we are interested in *prefix-free* sets, which means that for any string $s \in \mathcal{S}$, no other string in $\mathcal{S}$ has s as a prefix. By making this assumption we do not lose generality, as any string set can be made prefix-free by adding a special character to the alphabet, called the *terminator*, and appending it to every string in the set. For example, this is the approach taken in C/C++ where the strings are *null-terminated*.

We define the following operations on string dictionaries.

- Lookup(s) returns a distinct integer in $[|\mathcal{S}|]$ for each string $s \in \mathcal{S}$, or $\perp$ if the string is not in $\mathcal{S}$;
- Access(i) returns the string s such that Lookup(s) $= i$.

In other words, Lookup and Access form a bijection between $\mathcal{S}$ and $[|\mathcal{S}|]$.

2.7.1 *Compacted tries*

The traditionally most popular data structure to represent a string dictionary is the *trie* [46]. A trie is a labeled tree whose root-to-leaf paths correspond to the strings in the string set. We will make use of a variant of the trie known as *compacted trie*. A compacted trie of a non-empty prefix-free set of strings is a tree built recursively as follows.

- The compacted trie of a single string is a node whose label is the string itself.
- Given a nonempty string set $\mathcal{S}$, the root of the tree is labeled with the longest common prefix α (possibly empty) of the strings in $\mathcal{S}$.

  For each character b such that the set $\mathcal{S}_b = \{\beta | \alpha b \beta \in \mathcal{S}\}$ is nonempty, the compacted trie built on $\mathcal{S}_b$ is attached to the root as a child.

  The edge is labeled with the *branching character* b. The number of characters in the label α is called the *skip*, and denoted with δ.

The compacted trie is a straightforward generalization to arbitrary alphabets of the *Patricia trie* [90], which was defined on binary strings. An useful characteristic of the Patricia trie is that each internal node has exactly two children.

For the sake of brevity, in the following we use the term *trie* to indicate a compacted trie, unless otherwise specified.

2.7.2 *Dynamic Patricia tries*

A Patricia trie can be easily made dynamic, supporting insertion and deletion of strings. We describe here the data structure in detail, as it will be needed in Chapter 6.

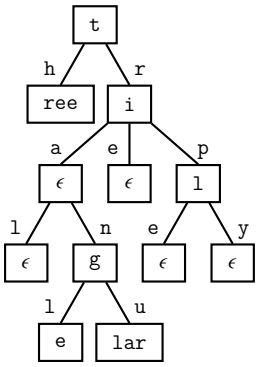


FIGURE 2.3: Example of a compacted trie for the string set {three, trial, triangle, triangular, trie, triple, triply}.

Each node contains two pointers to the children, one pointer to the label and one integer for its length. For k strings, this amounts to $O(kw)$ bits. Given this representation, all navigational operations are trivial. The total space is $O(kw) + |L|$ bits, where L is the concatenation of the labels in the compacted trie.

Insertion of a new string s *splits* an existing node, where the mismatch occurs, and adds a leaf. The label of the new internal node is set to point to the label of the split node, with the new label length (corresponding to the mismatch of s in the split node). The split node is modified accordingly. A new label is allocated with the suffix of s starting from the mismatch, and assigned to the new leaf node. This operation takes $O(|s|)$ time and the space grows by $O(w)$ bits plus the length of the new suffix, hence maintaining the space invariant.

When a new string is deleted, its leaf is deleted and the parent node and the other child of the parent need to be merged. The highest node that shares the label with the deleted leaf is found, and the label is deleted and replaced with a new string that is the concatenation of the labels from that node up to the merged node. The pointers in the path from the found node and the merged node are replaced accordingly. This operation takes $O(\hat{\ell})$ where $\hat{\ell}$ is the length of the longest string in the trie, and the space invariant is maintained.

2.8 RANGE MINIMUM QUERIES

Given a sequence A of elements drawn from a totally ordered universe, a *Range Minimum Query* (RMQ) for the range $[i, j]$ asks for the position of the minimum element in $A[i, j]$ (returning the leftmost in case of ties). The operation $RMQ_A(i, j)$ can be trivially implemented by scanning the interval $A[i, j]$, but this requires $O(j - i)$ operations and thus it can be as slow as $O(|A|)$.

The problem is intimately related to the *Least Common Ancestor* (LCA) problem on trees. In fact, the LCA problem can be solved by reducing it to an RMQ on a sequence

of integers derived from the tree, while RMQ can be solved by reducing it to an LCA in a tree derived from the sequence, called the *Cartesian Tree* [7].

Fischer and Heun [44] presented a solution to RMQ that uses a DFUDS representation of a tree called *2d-Min-Heap*. The RMQ operation on A is then reduced to a ± 1RMQ operation on the DFUDS sequence. As noted by Davoodi et al. [26], the 2d-Min-Heap is a tree transformation of the Cartesian Tree.

Using Fischer and Heun's data structure, it is possible to support the operation $\mathrm{RMQ}_{\mathrm{A}}(i, j)$ in constant time, and the data structure takes $2n + o(n)$ bits, without the need to store the sequence A.

3 SEMI-INDEXING TEXTUAL SEMI-STRUCTURED DATA

Semi-structured data formats have enjoyed popularity in the past decade and are virtually ubiquitous in Web technologies: extensibility and hierarchical organization—as opposed to flat tables or files—made them the format of choice for documents, data interchange, document databases, and configuration files.

The field of applications of semi-structured data is rapidly increasing. These formats are making their way into the realm of storage of *massive* datasets. Their characteristics of being schema-free makes them a perfect fit for the mantra *"Log first, ask questions later"*, as the schema of the log records is often evolving. Natural applications are crawler logs, query logs, user activity in social networks, to name a few.

In this domain JSON (*JavaScript Object Notation*, see [72]) in particular has been gaining momentum: the format is so simple and self-evident that its formal specification fits in a single page, and it is much less verbose than XML. In fact, both CouchDB [22] and MongoDB [88], two of the most used ([23, 89]) modern large-scale distributed schema-free document databases, are based on JSON, and Jaql [70] and Hive JSON SerDe [63] implement JSON I/O for Hadoop. These systems all share the same paradigm:

(*a*) Data is conceptually stored as a sequence of records, where each record is represented by a single JSON document. The records may have heterogeneous schema.

(*b*) The records are processed in MapReduce [27] fashion: during the Map phase the records are loaded sequentially and parsed, then the needed attributes are extracted for the computation of the Map itself and the subsequent Reduce phase.

In part (*b*) the extracted data is usually a *small* fraction of the records actually loaded and parsed: in logs such as the ones mentioned above, a single record can easily exceed hundreds of kilobytes, and it has to be loaded entirely even if just a single attribute is needed. If the data is on disk, the computation time is dominated by the I/O.

A typical way of addressing the problem of parsing documents and extracting attributes is to change the data representation by switching to a *binary*, easily traversable format. For instance XML has a standard binary representation, Binary XML ([11]),

and more sophisticated schemes that enable more powerful traversal operations and/or compression have been proposed in the literature (see [102]). Likewise MongoDB uses BSON [17], a binary representation for JSON. However, switching from a textual format to an ad-hoc binary format carries some drawbacks.

— In large-scale systems, the producer is often decoupled from the consumer, which gets the data through append-only or immutable, possibly compressed, distributed filesystems, so using simple self-evident standard formats is highly preferable.

— Binary data is not as easy to manually inspect, debug or process with scripting languages as textual data.

— If input/output is textual, back-and-forth conversions are needed.

— If existing infrastructure is based on textual formats, changing the storage format of already stored data can be extremely costly.

In fact, despite their advantages binary formats have not gained widespread adoption.

Surprisingly, even with a binary format it is not easy to support all the tree operations without incurring a significant space overhead. For example, BSON prepends to each element its size in bytes, enabling fast forward traversal by allowing to "skip" elements, but accessing the i-th element of an array cannot have sublinear I/O complexity.

SEMI-INDEX. In this chapter we introduce the notion of *semi-indexing* to speed up the access to the attributes of a textual semi-structured document without altering its storage format; instead, we accompany it with a small amount of redundancy.

A *semi-index* is a succinct encoding of the parse tree of the document together with a positional index that locates the nodes of the tree on the unparsed document. Navigation of the document is achieved by navigating the succinct parse tree and parsing on the fly just the leaf nodes that are needed, by pointing the parser at the correct location through the positional index. This way, a small part of the document has to be accessed: the I/O cost is greatly reduced if the documents are large, and on a slow medium such as a disk or a compressed or encrypted filesystem. Specifically, the I/O cost is proportional to the number of tree queries, regardless of the document size.

No explicit parsing tree is built, instead we employ a well-known balanced parenthesized representation and a suitable directory built on the latter. The resulting encoding is so small that it can be computed once and stored along with the document, without imposing a significant overhead.

We call our approach "semi-index" because it is an index on the *structure* of the document, rather than on its *content*: it represents a middleground between full indexing (where the preprocessing time and space can be non-negligible because the full content is indexed) and streaming (where data are not indexed at all).

The main feature is that the document in its textual semi-structured format (or *raw data*) is not altered in any way, and can be considered a *read-only random access oracle*. The combination of *raw data + semi-index* can thus support the same operations as an optimized binary format, while maintaining the advantages of keeping the raw data unaltered.

— Backward-compatibility: Existing consumers can just ignore the semi-index and read the raw data.

— The semi-index does not need to be built by the producer: the consumer can build it and cache it for later use.

— The raw data does not need to be given in explicit form, provided that a random-access primitive is given, while the semi-index is small enough that it can easily fit in fast memory. For example a compression format with random access can be used on the documents. We demonstrate this feature in the experimental analysis by compressing blockwise the data with *zlib*.

A semi-index can be engineered in several ways depending on the format grammar and the succinct data structures adopted for the purpose. Although the semi-indexing scheme is general, we focus on a concrete implementation using JSON as the underlying format for the sake of clarity.

In our experiments (Section 3.5) we show that query time is very fast, and speed-up using the precomputed semi-index on a MapReduce-like computation ranges from 2.5 to 4 times. Using a block-compressed input file further improves the running time when the document size is large, by trading I/O time for CPU time. This comes at the cost of a space overhead caused by the storage of the semi-index, but on our datasets the overhead does not exceed (and is typically much less than) around 10% of the raw data size.

When comparing against the performance of BSON, our algorithm is competitive on some datasets and significantly better on others. Overall, raw data + semi-index is never worse than BSON, despite the latter is an optimized binary format.

To our surprise, even if the semi-index is built on the fly right before attribute extraction, it is faster than parsing the document: thus semi-indexing can be also thought of as a fast parsing algorithm.

The main drawback of the semi-index is that it has a fixed additive overhead of 150–300 bytes (depending on the implementation), making it unsuitable for very small individual documents. This overhead can be however amortized for a *collection* of documents. In our experiments we follow this approach.

3.1 Related work

A similar approach for comma-separated-values files is presented in [67]. The authors describe a database engine that skips the ordinary phase of loading the data into the

database by performing queries directly on the flat textual files. To speed up the access to individual fields, a (sampled) set of pointers to the corresponding locations in the file is maintained, something similar to our *positional index*. This approach, however, is suitable only for tabular data.

Virtually all the work on indexing semi-structured data focuses on XML, but most techniques are easily adaptable to other semi-structured data formats, including JSON. For example, AgenceXML [24] and MarkLogic[66] convert JSON documents internally into XML documents, and Saxon [74] plans to follow the same route.

To the best of our knowledge, no work has been done on indexes on the *structure* of the *textual* document, either for XML or other formats. Rather, most works focus on providing indexes to support complex tree queries and queries on the *content*, but all of them use an ad-hoc binary representation of the data (see [57] for a survey on XML indexing techniques).

For the storage of XML data several approaches were proposed that simultaneously compress XML data while supporting efficient traversal, and they usually exploit the separation of tree structure and content (see [102] for a survey on XML storage schemes).

Some storage schemes employ succinct data structures: for example [28] uses a succinct tree to represent the XML structure, and [40] exploits compressed non-binary dictionaries to encode both the tree structure and the labels, while supporting subpath search operations.

The work in [118] is the closest to ours, as it uses a balanced-parentheses succinct tree representation of the document tree, but like the others it re-encodes the contents of the document to a custom binary format and discards the unparsed form.

Industrial XML parsers such as Xerces2 [3] tackle the cost of materializing the full document tree by keeping in memory only a summary of the tree structure with pointers to the textual XML, while parsing only the elements that are needed. This technique, known as Lazy XML parsing, needs however to scan the full document to parse the tree structure every time the document is loaded, hence the I/O complexity is no better than performing a full parse. Refinements of this approach such as double-lazy parsing [35] try to overcome the problem by splitting the XML file in several fragments stored in different files that link to each other. This however requires to alter the data, and it is XML-specific. Besides, each fragment that is accessed has to be fully scanned. Semi-indexing is similar to lazy parsing in that a pre-parsing is used to speed up the access to the document, but the result of semi-index preprocessing is small enough that can be saved along the document, while in lazy parsing the preprocessing has to be done every time the document is loaded.

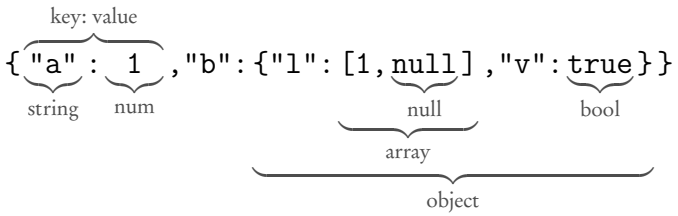
3.2 THE JSON FORMAT

Since in this chapter we will be using extensively the JSON format, we shortly summarize its data model and syntax in this section. JSON (JavaScript Object Notation) is a small

fragment of the Javascript syntax used to represent semi-structured data.

A JSON *value* can either be an atom (i.e. a string, a number, a boolean, or a *null*), an *object*, or an *array*. An object is an unordered list of key/value pairs, where a key is a string. An array is an ordered list of values (so one can ask for the i-th value in it).

A JSON *document* is just a value, usually an object. The following figure shows an example of a JSON document and its components.

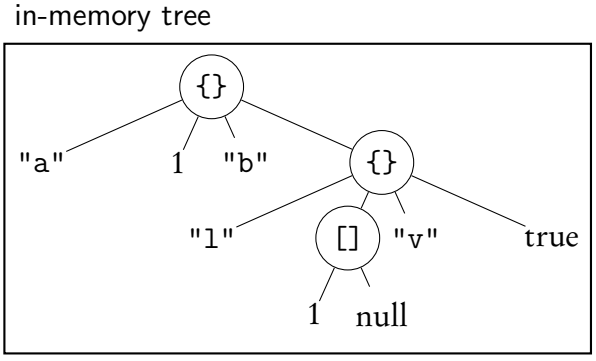


The document tree of a JSON value is the tree where the leaf nodes are the atoms and internal nodes are objects and arrays. The queries usually supported on the tree are the basic traversal operations, i.e. *parent* and *i-th* child, plus *labeled child* for objects. We use the Javascript notation to denote path queries, so for example in this document `a` is `1` and `b.l[1]` is `null`.

3.3 SEMI-INDEXING TECHNIQUE

We illustrate our technique with the JSON document shown in the example of Section 3.2.

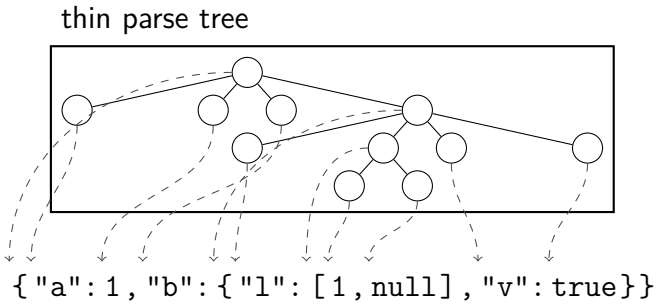
The most common way of handling textual semi-structured data is to *parse* it into an in-memory tree where the leaf nodes contain the parsed atoms. The tree is then queried for the requested attributes. Since the tree contains all the relevant data, the raw document is no longer needed, as shown in the figure below.




```
{"a":1,"b":{"l":[1,null],"v":true}}
```

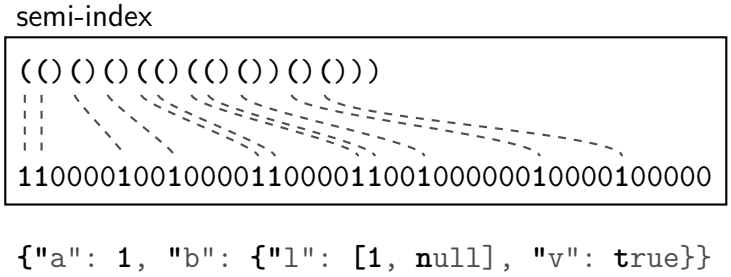
We would like to create a structure that allows us to navigate the document *without having to parse it*. One possible approach could be to replace the values in the nodes

of the parse tree with *pointers* to the first character of their corresponding phrases in the document. This clearly requires to store also the raw data along with the resulting "thinner" tree.



We can now navigate the parse tree, and parse just the values corresponding to the leaves that are actually needed. Note that if the grammar is LL(1)[1], we do not need to store the node type: it suffices to look at the first character of the node to recognize the production. So we are left with the tree data structure representing the topology of the parse tree, and a pointer for each node to the first character of its phrase in the raw text. This is very similar to the approach adopted by lazy parsers for XML.

Still, this requires building explicitly the parse tree every time the document is loaded. Instead, the parse tree and pointers can be encoded using a sequence of balanced parentheses for the first and a bitvector for the second. As we will show in Section 3.4, a quick scan of the document is sufficient to produce the two sequences, which form the semi-index.

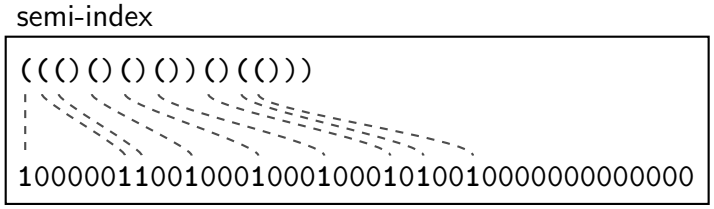


As we discuss in the experiments of Section 3.5, building the semi-index is much faster than full parsing. We merely store two binary sequences, encoding each parenthesis with a single bit, augmented with the succinct data structures needed to support the traversal operations. Since the binary sequence with the positions is sparse, it can be encoded in compressed format using very small space. Overall, the space needed to store

[1]Most semi-structured data formats are LL(1), including XML and JSON. If the grammar is not LL(1) an additional $\log P$ bits per node may be needed, where P is the number of productions.

the two sequences is very small, so they can be computed just once and then be stored for future use.

This scheme can be applied to other formats. For instance, the XML semi-index would look like the following figure.




```
<entry␣id="1" cat="c">t1<b>t2</b></entry>
```

Our approach is to employ a set of succinct structures to replace the functionality of the thinner parse tree, and obtain faster construction and query time (see Section 3.5). We can thus define the *semi-index*.

DEFINITION 3.3.1 A *semi-index* for a document D is a succinct encoding of (i) the topology of the parse tree T of D, and (ii) the pointers that originate from each node of T to the beginning of the corresponding phrase in D.

Let m denote the number of nodes in T. The general template to build the semi-index using an event parser is illustrated in Algorithm 3.1[2]. By *event parser* we mean a parser that simulates a depth-first traversal of the parse tree, generating an `open` event when the visit enters a node and a `close` event when it leaves it. (An example is the family of SAX parsers for XML.) If the event parser uses constant working memory and is one-pass, so does Algorithm 3.1. Thus it is possible to build the semi-index without having to build an explicit parse tree in memory. In the pseudocode, (i) `bp` is the balanced parentheses tree structure, and (ii) `positions` is the Elias-Fano representation of the pointers.

For the construction algorithm to be correct we need the following observation.

OBSERVATION 3.3.2 The sequence of pointers in a pre-order visit of the parse tree T induces a non-decreasing sequence of m positions in D. In other words, the sequence of positions of `open` events in an event parsing is non-decreasing.

Observation 3.3.2 allows us to use the Elias-Fano encoding, whose implementation is referred to as `EliasFanoSequence` in Algorithm 3.1, for the positions.

Algorithm 3.2 shows the pseudocode for some tree operations. The operation `get_node_pos` returns the position of the phrase in the document D corresponding to the node of T represented by the parenthesis at position `par_idx`. Operation `first_child`

ALGORITHM 3.1: Construction of semi-index using an event parser

```
def build_semi_index(s):
    positions = EliasFanoSequence()
    bp = BalancedParentheses()
    for event, pos in parse_events(s):
        if event == 'open':
            bp.append(1)
            positions.append(pos)
        elif event == 'close':
            bp.append(0)
```

ALGORITHM 3.2: Some tree operations on the semi-index

```
def get_node_pos(bp, positions, par_idx):
    node_idx = bp.rank(node)
    pos = positions.select(node_idx)
    return pos

def first_child(bp, par_idx):
    return par_idx + 1

def next_sibling(bp, par_idx):
    next = bp.find_close(par_idx) + 1
    if bp[next]:
        return next
    else:
        return None
```

returns the position of the parenthesis corresponding to the first child of the current node. Operation `next_child` returns the position of the next sibling (if any).

We now discuss the space usage of our encoding. As shown in our example, the tree topology can be encoded with the balanced parentheses representation, thus taking $2m + o(m)$ bits. Nodes are identified by the open parentheses, so that $\text{Rank}_{(}(i)$ gives the pre-order index of node i.

The pointers can be thus encoded in pre-order by using the Elias-Fano representation, taking another $2m + m\left\lceil \log \frac{n}{m} \right\rceil + o(m)$ bits. Summing the two figures leads to the following lemma.

LEMMA 3.3.3 A semi-index of a document D of n bytes such that the parse tree T has m nodes can be encoded in

$$4m + m\left\lceil \log \frac{n}{m} \right\rceil + o(m) \tag{3.1}$$

bits, while supporting each of the tree navigational operations in O(1) time.

The bound in (3.1) compares favorably against an explicit representation of the tree T and its text pointers: even if space-conscious, it would require 2 node pointers plus one text pointer, i.e. $m(2\log m + \log n)$ bits. For example, for a reasonably sized 1MB document with density `0.2` (1 node for each 5 bytes on average), the size of the data structure would be 1.4MB, that is 140% of the document itself!

In our practical implementation, we use the data structures described in Chapter 7. The overall space is approximately $5.5m + m\left\lceil \log \frac{n}{m} \right\rceil$ bits and the operations have $O(\log n)$ complexity. The encoding of the example above then takes 262kB, just 26.2% of the raw document. Even in case of a pathological high density document, i.e. $n = m$, the data structure would occupy $5.5m$ bits, i.e. an 68.7% overhead. Real-world documents, however, have very low densities (see Section 3.5).

3.4 ENGINEERING THE JSON SEMI-INDEX

In this section we describe a semi-index specifically tailored for JSON. It slightly deviates from the general scheme presented in Section 3.3, since it exploits the simplicity of the JSON grammar to gain a few more desirable properties, as we will see shortly.

As in the general scheme, we associate two bitvectors to the JSON document, `bp` and `positions`, that are built as follows.

— The structural elements of the document, i.e. the curly brackets `{}`, the square brackets `[]`, the comma `,` and the colon `:` are marked with `1` in the bitvector `positions`, which is encoded with the Elias-Fano representation.

[2]The pseudocode is actually working Python code, but we omitted the auxiliary functions and classes for the sake of brevity.

— For each structural element a pair of parentheses is appended to the `bp` (balanced parentheses) vector:

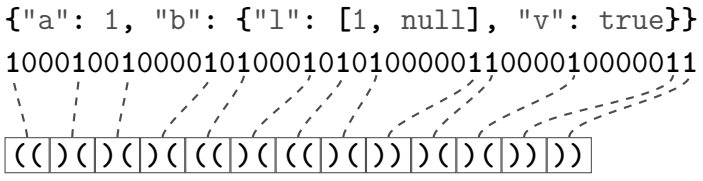
  – Brackets `{` and `[` open their own node (the *container*) and the first element of the list, so their encoding is `((`.

  – Brackets `}` and `]` close the last element of the list and their own node, so their encoding is `))`.[3]

  – Comma `,` closes the current element and opens the next, so its encoding is `)(`.

  – Colon `:` is treated like the comma, so key/value pairs are encoded simply as consecutive elements.

An example of the encoding is shown below: the JSON document (top), the `positions` bitvector (middle), and the `bp` bitvector (bottom). We implement `bp` as a binary sequence where `(` is encoded by `1`, and `)` is encoded by `0`.



This encoding allows a very simple algorithm for the construction of the semi-index: a one-pass scan of the document is sufficient, and it can be implemented in constant space (in particular, no stack is needed). As a result, building the semi-index is extremely fast. The pseudocode is shown in Algorithm 3.3.

Our ad-hoc encoding gives us two further features. First, each bit `1` in the bitvector `positions` is in one-to-one correspondence with pairs of consecutive parentheses in `bp`: there is no need to support a Rank operation to find the position in `bp` corresponding to a `1` in `positions`, as it is sufficient to divide by 2. Second, since the positions of closing elements (`}`, `]`, `,`) are marked in `positions`, it is possible to locate in constant time both endpoints of the phrase that represents a value in the JSON document, not just its starting position.

Navigation inside a JSON document proceeds as follows. Finding a key in an object is performed by iterating its subnodes in pairs and parsing the keys until the searched one is found. The pseudocode for this operation can be found in `object_get`, Algorithm 3.4.

The query algorithm makes a number of probes to the JSON document that is linear in the fanout of the object. This is not much of a problem since the fanout is usually

[3]The empty object `{}` and array `[]` have encoding `(())`, so they are special cases to be handled separately in navigation.

ALGORITHM 3.3: Construction of JSON semi-index

```
def build_semi_index(json):
    positions = EliasFanoBitVector()
    bp = BalancedParentheses()
    i = 0
    while i < len(json):
        c = json[i]
        if c in '[{':
            positions.append(1)
            bp.extend([1, 1])
        elif c in '}]':
            positions.append(1)
            bp.extend([0, 0])
        elif c in ',:':
            positions.append(1)
            bp.extend([0, 1])
        elif c == '"':
            # consume_string returns the position of
            # the matching '"'
            new_i = consume_string(json, i)
            length = new_i - i + 1
            i = new_i
            positions.extend([0] * length)
        else:
            positions.append(0)
        i += 1
```

ALGORITHM 3.4: Get position and object child by key

```
def get_pos(node):
    pos = positions.select(node / 2)
    pos += node % 2
    return pos

def object_get(json, node, key):
    # if node is odd, it is a comma, so skip it
    node += node % 2
    opening_pos = get_pos(node)
    if json[opening_pos] != '{':
        # not an object
        return None
    # empty objects are a special case
    if json[opening_pos + 1] == '}':
        return None

    node = node + 1
    node_pos = opening_pos + 1

    while True:
        if bp[node] == 0:
            return None
        node_close = bp.find_close(node)
        # node position after the colon
        val_open = node_close + 1
        # check if current key matches
        if parse(json, node_pos) == key:
            return val_open
        # otherwise skip to next key/value pair
        val_close = bp.find_close(val_open)
        node = val_close + 1
        # skip the comma
        node_pos = get_pos(node) + 1
```

small. Otherwise, if it is possible to ensure that the keys are in sorted order, binary search can be used to reduce the number of probes to the logarithm of the fanout.

Array access can be done similarly with forward iteration through FindClose, with backwards iteration by jumping to the parenthesis closing the container and iterating on the contents with FindOpen, or with the ith child if `bp` supports it. In any case, at most 3 accesses to the JSON document are made: the I/O complexity remains constant even if the runtime complexity may be linear, assuming that the semi-index fits in main memory.

We remark that in the design of the engineered JSON semi-index we have chosen simplicity over theoretical optimality. In general, other space/time/simplicity tradeoffs can be achieved by composing together other succinct data structures chosen from the vast repertoire in the literature, thus giving rise to a wide range of variations of the semi-indexing framework.

3.5 Experimental analysis

In this section we discuss the experimental analysis of the semi-index described in Section 3.4. The benchmark is aimed at the task of attribute extraction described in the chapter introduction.

— Each dataset consists in a text file whose lines are JSON documents. The file is read from disk.

— The query consists in a list of key/index paths, to define which we use the Javascript notation. For instance given the following document

```
{"a": 1, "b": {"v": [2, "x"], "l": true}}
```

the query `a,b.v[0],b.v[-1]` returns `[1, 2, "x"]`, i.e. the list of the extracted values encoded as a JSON list. Note that, borrowing a Python convention, negative indices count from the end of the array, so `-1` is the last element.

— Each benchmark measures the time needed to run the query on each document of the dataset and write the returned list as a line in the output file.

Implementation and testing details. The algorithms have been implemented in C++ and compiled with g++ 4.7. The tests were run on a dual core Intel Core 2 Duo E8400 with 6MiB L2 cache, 4GiB RAM and a 7200RPM SATA hard drive, running Linux 3.5.0 – 64-bit. Each test was run 3 times, and the timings averaged. Before each run the kernel page caches were dropped to ensure that all the data is read from disk. When not performing sequential scan, the input files were memory-mapped to let the kernel load lazily only the needed pages. For the construction of the semi-index each dataset is

considered as a single string composed by the concatenation of all the documents, so a single semi-index is built for each dataset and stored on a separate file. The positions in the positional index are thus absolute in the file, not relative to each single document. For both the Elias-Fano and balanced parentheses sequences we uses the implementations in the *Succinct* library, described in Chapter 7. Note that since the publication of [97] the experiments have been re-run with the latest version of the library and of the compiler, resulting in improved performance.

The source code used for the experiments is available at the URL `https://github.com/ot/semi_index`.

DOCUMENT COMPRESSION. To simulate the behavior on compressed file systems we implemented a very simple block compression scheme which we call `gzra` (for gzipped "random access"). The file is split into 16kB blocks which are compressed separately with `zlib` and indexed by an offset table. On decompression, blocks are decompressed as they are accessed. We keep an LRU cache of decompressed blocks (in our experiments we use a cache of 8 blocks). The on-disk representation is not optimized—it may be possible to shave some I/O cost by aligning the compressed blocks to the disk block boundaries.

DATASETS. The experiments were performed on a collection of datasets of different average document size and density. On one extreme of the spectrum there are datasets with small document size (`wp_events`), which should yield little or no speed-up, and very high density (`xmark`), which should give a high semi-index overhead. On the other extreme there is `wp_history` which has large documents and relatively small density. Specifically:

The Wikipedia data was obtained by converting to JSON the Wikipedia dumps [115], while we used the `xmlgen` tool from XMark [103] and converted the output to JSON to generate synthetic data of very high density.

— `wp_events`: Each document represents the metadata of one edit on Wikipedia.

— `delicious` [71]: Each document represents the metadata of the links bookmarked on Delicious in September 2009.

— `openlib_authors` [109]: Each document represents an author record in The Open Library.

— `wp_history`: Each document contains the full history of a Wikipedia page, including the text of each revision.

— `xmark`: Each document is generated using `xmlgen` from XMark with scale factor chosen uniformly in the range $[0.025, 0.075)$.

QUERIES. The queries performed on each dataset are shown in Table 3.1. We have chosen the queries to span several depths in the document trees (the XPath dataset `xmark` has deeper trees that allow for more complex queries). Some queries access negative indices in arrays, to include the contribution of the performance of backwards array iteration in our tests.

Dataset	Queries

`wp_events`	`id`
	`timestamp`
	`title`

`delicious`	`links[0].href`
	`tags[0].term`
	`tags[-1].term`

`openlib_authors`	`name` `last_modified.value`

`wp_history`	`id`
	`title`
	`revision[0].timestamp`
	`revision[-1].timestamp`

`xmark`	`people.person[-1].name`
	`regions.europe.item[0].quantity`
	`regions.europe.item[-1].name`
	`open_auctions.open_auction[0].current`

TABLE 3.1: Queries performed on each dataset

TESTING. For each dataset we measured the time needed to perform the following tasks.

— `wc`: The Unix command that counts the number of lines in a file. We use it as a baseline to measure the I/O time needed to scan sequentially the file without any processing of the data.

— `jsoncpp`: Query task reading each line, and parsing it using the JSONCpp library [73] (one of the most popular and efficient JSON C++ libraries). The requested values are output by querying the in-memory tree structure obtained from the parsing.

— `bson`: Query task using data pre-converted to BSON.

— `si_onthefly`: Query task reading each line, building on the fly the semi-index, and using it to perform the queries. Note that in this case the semi-index is not precomputed.

— `si`: Query task using a precomputed semi-index from a file on disk.

— `si_compr`: Like `si`, but instead of reading from the uncompressed JSON file, the input is read from a `gzra`-compressed file.

— `si_build`: Construction and storage to disk of the semi-index from the JSON file, which is then used by `si` and `si_compr`.

RESULTS. We summarize the running times for the above tests in Figure 3.1 and Table 3.3, and the space overhead in Figure 3.2 and Table 3.2, which also reports the statistics for each file in the dataset. We now comment in some detail these experimental findings.

A common feature on all the datasets is that the standard load-and-parse scheme using the JSONCpp library, and implemented as `jsoncpp`, has the worst performance. If time efficiency is an issue, the other methods are preferable.

BSON is a good candidate in this sense, since it uses pre-converted data, as implemented in `bson`, and always runs faster than `jsoncpp`. It is interesting to compare `bson` with our methods, which also run faster than `jsoncpp`.

Consider first the situation in which we do not use any preprocessing on the data: when performing the queries, we replace the full parsing of the documents with an on-the-fly construction of the semi-index, as implemented in `si_onthefly`. As shown in our tests, `si_onthefly` performs remarkably well even if it has to load the full document, as the semi-index construction is significantly faster than a full parsing. Compared to `bson`, which uses a pre-built index, the running times are slightly larger but quite close, and the I/O times are similar. Surprisingly, for file `wp_history`, `si_onthefly` is faster than `bson`: a possible reason is that in the latter the value sizes are interleaved with the values, causing a large I/O cost, whereas the semi-index is entirely contained in memory.

We now evaluate experimentally the benefit of using a pre-built and saved semi-index. The running times for `si_build` show that the construction of the semi-index is very fast, and mainly dominated by the I/O cost (as highlighted by the comparison with `wc`).

Using the pre-computed semi-index, `si` can query the dataset faster than `bson`, except for file `xmark` where it is slightly slower: as we shall see, when this file is in compressed format, the I/O time is significantly reduced. Also, on `wp_history` querying the last element of an array requires `bson` to scan all the file (as explained in the introduction), while `si` can jump straight to the correct position; overall `si` is 5 times faster than `bson` on this dataset. Note that contrarily to `bson`, `si` require less time than `wc` in some cases since it takes advantage of the semi-index to make random accesses to the file and retrieve just the needed blocks.

Dataset	Records	Average kBytes	Average nodes	Semi-index overhead
wp_events	1000000	0.36	24.82	8.27%
delicious	1252973	1.04	61.28	7.67%
openlib_authors	6406158	0.25	22.00	10.15%
wp_history	23000	127.27	203.87	0.33%
xmark	1000	2719.47	221221.48	10.31%

TABLE 3.2: Number of documents, average document size, average number of nodes, and semi-index space overhead (percent with respect to average document size) for each dataset used in the benchmark

The space overhead of the pre-built semi-index is reported in the last column of Table 3.2 and item `si_size` of Figure 3.2. The semi-index takes between 8% and 10% of the uncompressed input for all datasets except `wp_history`, where the overhead is practically negligible because data is sparse.

If the overall space occupancy is an issue, we can opt for a variant of our method `si`, as implemented in `si_compr`, where the dataset is kept compressed using the `gzra` format previously discussed. Note that this format, which is a variant of `gzip`, requires slightly more space but it allows for random block access to compressed data (e.g. compare items `gz_size` and `gzra_size` in Figure 3.2). When comparing to the space required by the binary format of `bson` (item `bson_size` in Figure 3.2), we obtain a significant saving, where the total space occupancy of the semi-index and the compressed dataset is the sum of the values of items `gzra_size` and `size_size` in Figure 3.2.

Regarding its time performance, `si_compr` is slighter slower than `bson` and `si` for sparse files, while it performs better for dense files such as `wp_history` and `xmark`: in the former case, the decompression cost dominates the access cost, while in the latter the I/O cost is dominant and the reduced file size improves it (while still taking advantage of semi-indexing). This is also why `si_compr` is faster than `wc` on some files, and obtains a 10x speed-up on `wp_history` over `jsoncpp`.

Summing up, on all the datasets `si` is between 2.5 and 4 times faster than `jsoncpp`, while for `si_compr` the speed-up is between 1.5x and 10x. The graphs suggest that compression enables better performance as the average document size increases.

3.6 MEMORY-EFFICIENT PARSING

In this section we describe an alternative application of the semi-indexing technique.

A fully deserialized document tree takes often much more memory than the serialized document itself. Hence in case of big documents it is very likely that the textual (XML or JSON) document fits in main memory but its deserialized version doesn't.

Industrial parsers such as Xerces2 [3] work around this problem by loading in memory only the tree structure of the document and going back to the unparsed document

Dataset	Wall clock time (seconds)						
	`wc`	`jsoncpp`	`bson`	`si_onthefly`	`si`	`si_compr`	`si_build`
`wp_events`	3.4	13.2 *(3.83)*	8.0 *(2.33)*	9.3 *(2.70)*	**4.2 *(1.23)***	9.0 *(2.62)*	4.9 *(1.43)*
`delicious`	11.6	38.3 *(3.30)*	14.0 *(1.21)*	20.2 *(1.74)*	**14.6 *(1.26)***	19.8 *(1.70)*	14.7 *(1.27)*
`openlib_authors`	15.2	64.0 *(4.21)*	36.9 *(2.43)*	49.2 *(3.24)*	**22.9 *(1.51)***	36.0 *(2.37)*	19.0 *(1.25)*
`wp_history`	26.9	45.9 *(1.71)*	40.4 *(1.50)*	31.1 *(1.16)*	9.6 *(0.36)*	**4.4 *(0.16)***	30.4 *(1.13)*
`xmark`	24.9	127.5 *(5.11)*	26.4 *(1.06)*	32.2 *(1.29)*	27.6 *(1.11)*	**14.4 *(0.58)***	33.4 *(1.34)*

TABLE 3.3: Running times for each dataset. Numbers in parentheses are the runtimes normalized on `wc` time. Numbers in bold are the ones within 10% from the best

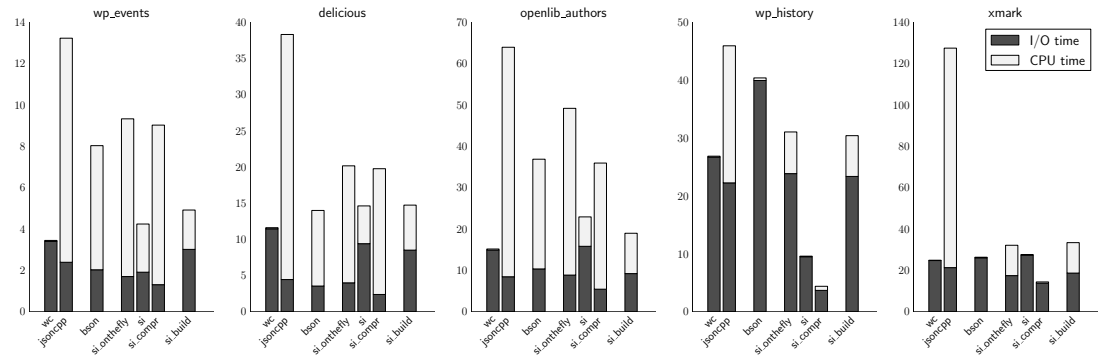


FIGURE 3.1: Wall clock times for each dataset as listed in Table 3.3. I/O time indicates the time the CPU waits for data from disk, while in CPU time the CPU is busy (but the kernel may use this time to prefetch pages from disk).

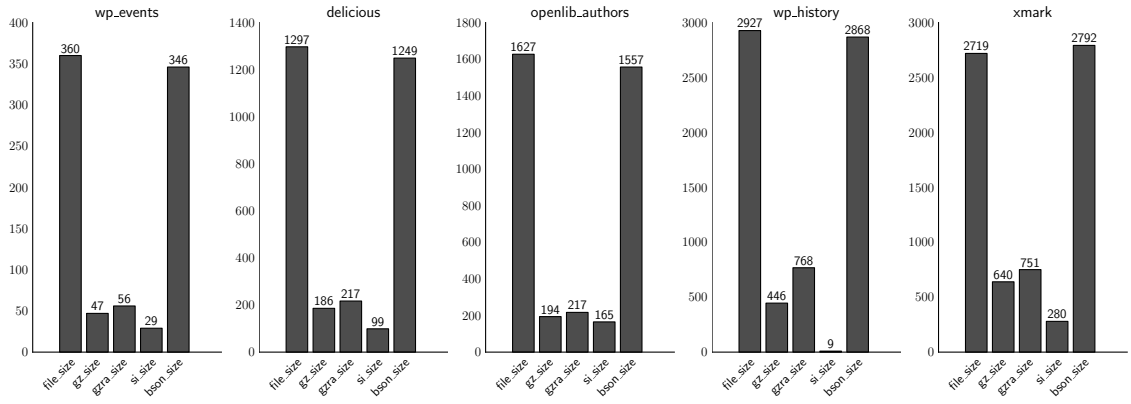


FIGURE 3.2: Space occupancy in MB of the file uncompressed (`file_size`), compressed with gzip (`gz_size`) and compressed with gzra (`gzra_size`), encoded in BSON (`bson_size`), and of the semi-index (`si_size`)

to parse the required elements. This approach however requires at least a pair of pointers per node, and usually much more. As shown in Section 3.3 for dense documents a pointer-based representation of the document tree can be more expensive than the document itself.

Since the construction of the semi-index is extremely fast, we suggest that a semi-index can be used in place of pointer-based data structures for lazy parsing.

Figure 3.3 shows the running times for `jsoncpp` and `si` construction and querying when the document is already in main memory, hence with no I/O involved. Note that query times using the semi-index for in-memory documents are just 10 times slower than by accessing a fully deserialized tree using `jsoncpp`. This is very reasonable, since the query time includes the time to parse the leaf attributes once the semi-index has identified their position in the unparsed document.

Thus the semi-index can be used as an alternative to explicit or lazy parsing in applications where memory is a concern, for example on mobile devices.

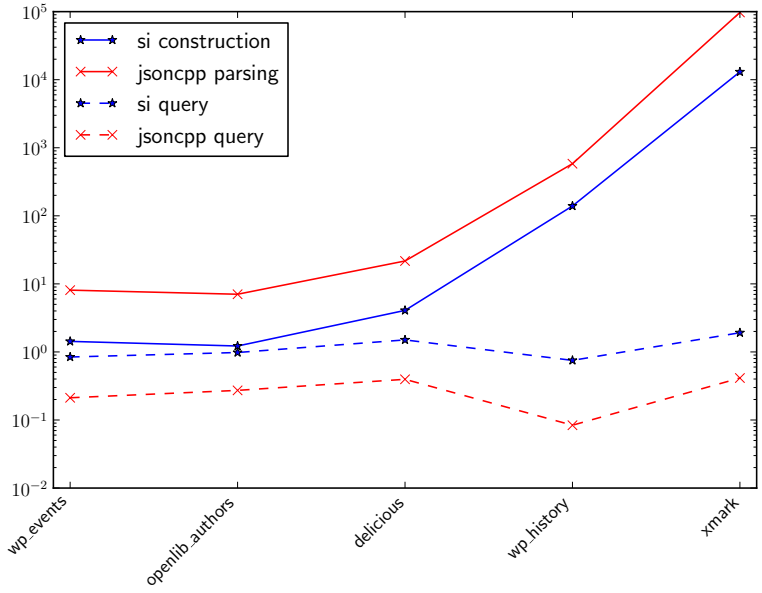


FIGURE 3.3: Timings (in microseconds, log-scale) for in-memory parsing and query operations.

4 Compressed string dictionaries

Tries are widely used data structures that turn a string set into a digital search tree. Several operations can be supported, such as mapping the strings to integers, r*etrie*ving a string from the trie, performing prefix searches, and many others. Thanks to their simplicity and functionality, they have enjoyed a remarkable popularity in a number of fields—Computational Biology, Data Compression, Data Mining, Information Retrieval, Natural Language Processing, Network Routing, Pattern Matching, Text Processing, and Web applications, to name a few—motivating the significant effort spent in the variety of their implementations over the last fifty years [75].

However their simplicity comes at a cost: as most tree structures, they generally suffer poor locality of reference due to pointer-chasing. This effect is amplified when using succinct representations of tries, where performing any basic navigational operation, such as visiting a child, requires accessing possibly several directories, usually with unpredictable memory access patterns. Tries are particularly affected as they are unbalanced structures: the height can be in the order of the number of strings in the set. Another issue with tries is that space savings are achieved only by exploiting the common prefixes in the string set, while it is not clear how to compress their nodes and their labels without incurring an unreasonable overhead in the running time.

In this chapter, we experiment with how *path decompositions* of tries help on both the above mentioned issues, inspired by the work presented in [39]. By using a *centroid* path decomposition, the height is guaranteed to be logarithmic in the number of strings, reducing dramatically the number of cache misses in a traversal; besides, for any path decomposition the node labels can be laid out in a way that enables efficient compression and decompression of a label in a sequential fashion.

4.1 Related work

The literature about space-efficient and cache-efficient tries is vast. Several papers address the issue of a cache-friendly access to a set of strings supporting prefix search [1, 8, 16, 38], but they do not deal with space issues except [8], which introduces an elegant variant of front coding.

Other papers aiming at succinct labeled trees and compressed data structures for

strings [4, 6, 9, 12, 40, 91, 101], support powerful operations—such as substring queries—and are very good in compressing data, but they do not exploit the memory hierarchy. Few papers [18, 39] combine (nearly) optimal information theoretic bounds for space occupancy with good cache efficient bounds, but no experimental analysis is performed.

More references on compressed string dictionaries can be found in [15].

4.2 STRING DICTIONARIES

In this section we describe an implementation of string dictionaries, as defined in Section 2.7, using path-decomposed tries.

PATH DECOMPOSITION. Our string dictionaries, inspired by the approach described in [39], are based on *path decompositions* of the compacted trie built on $\mathcal{S}$. A path decomposition $\mathcal{T}^c$ of a trie $\mathcal{T}$ is a tree where each node in $\mathcal{T}^c$ represents a path in $\mathcal{T}$. It is defined recursively in the following way: a root-to-leaf path in $\mathcal{T}$ is chosen and represented by the root node in $\mathcal{T}^c$. The same procedure is applied recursively to the sub-tries hanging off the chosen path, and the obtained trees become the children of the root, labeled with their corresponding branching character. Note that in the above procedure the order of the decomposed sub-tries as children of the root is arbitrary. Unlike [39], that arranges the sub-tries in *lexicographic order*, we arrange them in *bottom-to-top left-to-right* order; when using our succinct representation, this simplifies the traversal. Figure 4.1 shows a root-to-leaf path in $\mathcal{T}$ and its resulting node in $\mathcal{T}^c$.

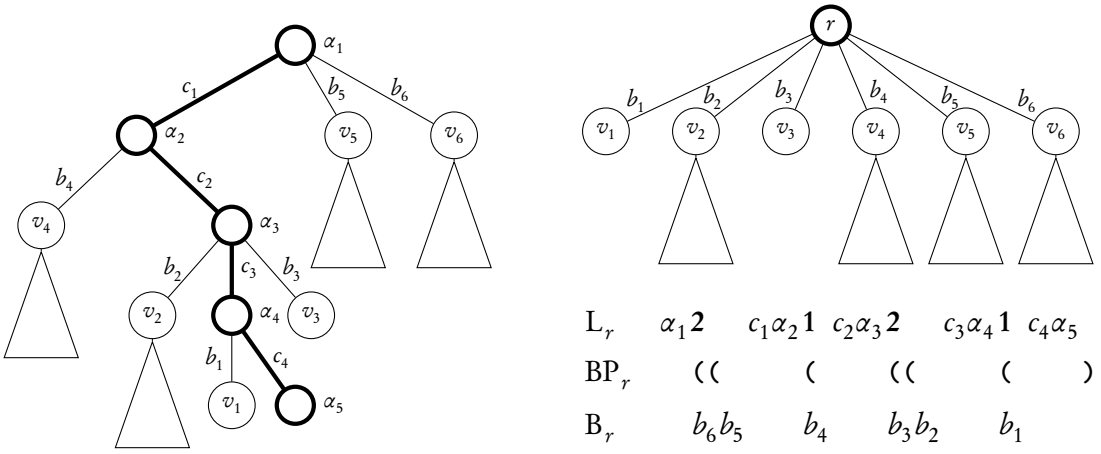


FIGURE 4.1: Path decomposition of a trie. The α_i denote the labels of the trie nodes, c_i and b_i the branching characters (depending on whether they are on the path or not).

There is a one-to-one correspondence between the paths of the two trees: root-to-*node* paths in $\mathcal{T}^c$ correspond to root-to-*leaf* paths in the trie $\mathcal{T}$, hence to strings in $\mathcal{S}$. This implies also that $\mathcal{T}^c$ has exactly $|\mathcal{S}|$ nodes, and that the height of $\mathcal{T}^c$ cannot be larger

than that of $\mathcal{T}$. Different strategies in choosing the paths in the decomposition give rise to different properties. We describe two such strategies.

— *Leftmost path*: Always choose the leftmost child.

— *Heavy path*: Always choose the *heavy* child, i.e. the one whose sub-trie has the most leaves (arbitrarily breaking ties). This is the strategy adopted in [39] and borrowed from [106].

OBSERVATION 4.2.1 If the leftmost path is used in the path decomposition, the depth-first order of the nodes in $\mathcal{T}^c$ is equal to the left-to-right order of their corresponding leaves in $\mathcal{T}$. Hence if $\mathcal{T}$ is lexicographically ordered, so is $\mathcal{T}^c$. We call it a *lexicographic path decomposition*.

OBSERVATION 4.2.2 If the heavy path is used in the path decomposition, the height of the resulting tree is bounded by $O(\log|\mathcal{S}|)$. We call such a decomposition a *centroid path decomposition*.

The two strategies enable a time/functionality trade-off: a lexicographic path decomposition guarantees that the indices returned by the Lookup are in lexicographic order, at the cost of a potentially linear height of the tree (but never higher than the trie $\mathcal{T}$). On the other hand, if the order of the indices is irrelevant, the centroid path decomposition gives logarithmic guarantees.[1]

We exploit a crucial property of path decompositions: since each node in $\mathcal{T}^c$ corresponds to a node-to-leaf path π in $\mathcal{T}$, the concatenation of the labels in π corresponds to a suffix of a string in $\mathcal{S}$. To simulate a traversal of $\mathcal{T}$ using $\mathcal{T}^c$ we only need to scan sequentially character-by-character the label of each node until we find the needed child node. Hence, any representation of the labels that supports *sequential access* (simpler than random access) is sufficient. Besides being cache-friendly, as we will see in the next section, this allows an efficient compression of the labels.

TREE REPRESENTATION. Each node v in the path-decomposed trie $\mathcal{T}^c$ is encoded with three sequences BP_v, B_v and L_v. Figure 4.1 shows an example for the root node.

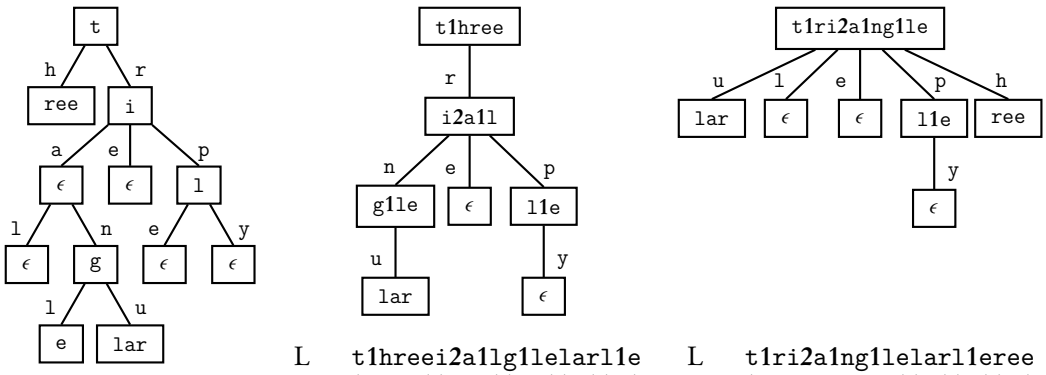
— The bitvector BP_v is a run of d_v open parentheses followed by a close parenthesis, where d_v is the degree of node v.

— The string B_v is the concatenation of the branching characters b_i of node v, written in reverse order, i.e. $\mathrm{B}_v = b_{d_v} \cdots b_1$. Note that they are in one-to-one correspondence with the `(`s in BP.

[1] In [39] the authors show how to have lexicographic indices in a centroid path-decomposed trie, using secondary support structures and arranging the nodes in a different order. The navigational operations are noticeably more complex, and require more powerful primitives on the underlying succinct tree, in particular for Access.

— The string L_v is the *label* of node v. We recall that each node in $\mathcal{T}^c$ represents a *path* in $\mathcal{T}$. To encode the path we augment the alphabet Σ with $|\Sigma| - 1$ special characters, $\Sigma' = \Sigma \cup \{\mathbf{1}, \mathbf{2}, \ldots, |\Sigma| - \mathbf{1}\}$, and alternate the label and the branching character of each node in the trie path with the number of sub-tries hanging off that node, encoded with the new special characters. More precisely, if the path in $\mathcal{T}$ corresponding to v is $w_1, c_1, \ldots, w_{k-1}, c_{k-1}, w_k$ where w_i are the trie nodes and c_i the edge labels, then $L = \alpha_{w_1} \tilde{d}_{w_1} c_1 \cdots \alpha_{w_{k-1}} \tilde{d}_{w_{k-1}} c_{k-1} \alpha_{w_k}$, where α_{w_i} is the node label of w_i and $\tilde{d}_{w_i}$ is the special character that represents $d_{w_i} - 1$ (all the degrees in $\mathcal{T}$ are at least 2). We call positions in L_v where the special characters $\tilde{d}_{w_i}$ occur *branching points*. Note that L_v is drawn from the larger alphabet Σ'; we will describe later how to encode it.

The sequences are then concatenated in depth-first order to obtain BP, B and L: if $v_1, \ldots, v_n$ are the nodes of $\mathcal{T}^c$ in depth-first order, then $BP = BP_{v_1} \cdots BP_{v_n}$, $B = B_{v_1} \cdots B_{v_n}$, and $L = L_{v_1} \cdots L_{v_n}$. By prepending an open parenthesis, BP represents the topology of $\mathcal{T}^c$ with the DFUDS encoding, hence it is possible to support fast traversal operations on the tree. Since the labels L_v are variable-sized, to support random-access to the beginning of each label we encode their endpoints in L using an Elias-Fano monotone sequence. Figure 4.2 shows an example of the three sequences for both lexicographic and centroid path decompositions on a small string set.



(a) Compacted trie (b) Lexicographic decomposition (c) Centroid decomposition

FIGURE 4.2: Example of a compacted trie for the set {three, trial, triangle, triangular, trie, triple, triply} and its lexicographic and centroid path decomposition trees. Note that the children in the path decomposition trees are in bottom-to-top left-to-right order.

TRIE OPERATIONS. Lookup is implemented recursively in a top-down fashion, starting from the root. To search for the string s in node v we scan simultaneously its label L_v and s. If the current character in the label is a special character, we add its value to an accumulator m (initially set to zero). Otherwise, we check if the current character in the label and the one in s match, in which case we proceed the scan with the next position in both. If instead they mismatch, the next character in the label must be a special character $\tilde{d}$, otherwise the string s is not in $\mathscr{S}$; in this case we return $\perp$ (likewise, if the string exceeds the label). The range between the accumulator m and $m+\tilde{d}$ indicates the children of v that correspond to the nodes in $\mathscr{T}$ branching from the prefix of s traversed up to the mismatch. To find the child it is then sufficient to find the matching branching character in B_v between in m and $m+\tilde{d}$. Since B is in correspondence with the open parentheses of BP, by performing a $\mathrm{Rank}_{(}$ of the current position in BP it is possible to find the starting point of B_v in B; also, because the characters in that range are sorted, a binary search can be performed. Again, $\perp$ is returned if no matching character is found. The search proceeds recursively in the found node with the suffix of s which starts immediately after the mismatch, until the string is fully traversed. The index of the) in BP corresponding to the found node is returned, i.e. the depth-first index of that node, which can be found with a $\mathrm{Rank}_{)}$ of the current position in BP. Note that it is possible to avoid all the Rank calls by using the standard trick of double-counting, i.e. exploiting the observation that between any two mates there is an equal number of (s and)s; this implies that if $j = \mathrm{FindClose}(i)$, then $\mathrm{Rank}_{)}(j) = \mathrm{Rank}_{)}(i) + (j - i - 1)/2$, so it is possible to keep track of the rank during the traversal. Likewise, $\mathrm{Rank}_{(}(i) = i - \mathrm{Rank}_{)}(i)$.

Access is performed similarly but in a bottom-up fashion. The initial position in BP is obtained by performing a $\mathrm{Select}_{)}$ of the node index returned by Lookup. Then the path is reconstructed by jumping recursively from the leaf thus found in $\mathscr{T}^c$ to the parent, until the root is reached. During the recursion we maintain the invariant that, at node v, the suffix of the string to be returned corresponding to the path below v has already been decoded. When visiting node v we know which child of v we are jumping from; hence, we scan its label L_v from the beginning until the position corresponding to that child is reached. The non-special characters seen during the scan are prepended to the string to be returned, and the recursion is carried on to the parent.

TIME COMPLEXITY. For the Lookup, for each node in the traversal we perform a sequential scan of the labels and a binary search on the branching character. If the string has length p, we can never see more than p special characters during the scan. Hence, if we assume constant-time FindClose in BP and Select in L, the total number of operations is $\mathrm{O}(p + h \log |\Sigma|)$.

To evaluate the cache efficiency, we can assume that the cost of sequentially scanning a small region of memory is dominated by the access to its first element, which we count as a *random memory access*. For the Lookup, the number of random memory accesses

is bounded by $O(h)$, where h is the height of the path decomposition tree. The Access is symmetric except that the binary search is not needed and $p \geq h$, so the number of operations is bounded by $O(p)$ where p is the length of the returned string. Again, the number of random memory accesses is bounded by $O(h)$.

LABELS ENCODING AND COMPRESSION. As previously mentioned, we need only to *scan* sequentially the label of each node from the beginning, so we can use any encoding that supports sequential scan with a constant amount of work per character. In the uncompressed trie, as a baseline, we simply use a vbyte encoding [117]. Since most bytes in the datasets do not exceed 127 in value, there is no noticeable space overhead. For a less sparse alphabet, more sophisticated encodings can be used.

The freedom in choosing the encoding allows us to explore other trade-offs. We take advantage of this to *compress the labels*, with an almost negligible overhead in the operations runtime.

We adopt a simple dictionary compression scheme for the labels: we choose a static dictionary of variable-sized words (that can be drawn from any alphabet) that will be stored along the tree explicitly, such that the overall size of the dictionary is bounded by a given parameter (constant) D. The node labels are then parsed into words of the dictionary, and the words are sorted according to their frequency in the parsing: a code is assigned to each word in decreasing order of frequency, so that more frequent words have smaller codes. The codes are then encoded using some variable-length integer encoding; we use vbyte to favor performance. To decompress the label, we scan the codes and for each code we scan the word in the dictionary, hence each character requires a constant amount of work.

We remark that the decompression algorithm is completely agnostic of how the dictionary was chosen and how the strings are parsed. For example, domain knowledge about the data could be exploited; in texts, the most frequent words would probably be a good choice.

Since we are looking for a general-purpose scheme, we adopt a modified version of the approximate Re-Pair [77] described in [21]: we initialize the dictionary to the alphabet Σ and scan the string to find the k most frequent pairs of codes. Then, we select all the pairs whose corresponding substrings fit in the dictionary and substitute them in the sequence. We then iterate until the dictionary is filled (or there are no more repeated pairs). From this we obtain simultaneously the dictionary and the parsing. To allow the labels to be accessed independently, we take care that no pairs are formed on label boundaries, as done in [15].

Note that while Re-Pair represents the words recursively as pairing rules, our dictionary stores the words literally, thus losing some space efficiency but fulfilling our requirement of constant amount of work per decoded character. If we used Re-Pair instead, accessing a single character from a recursive rule would have had a cost dependent

on the recursion depth.

IMPLEMENTATION NOTES. For the BP vector we use the balanced parentheses data structure described in Section 7.3, while to delimit the labels in L we use the Elias-Fano sequence implementation of Section 7.2.3. The search for the branching character is replaced by a linear search, which for the cardinalities considered (few tens of distinct symbols) is actually *faster* in practice. The dictionary is represented as the concatenation of the words encoded in 16-bit characters to fit the larger alphabet $\Sigma' = [0, 511)$. The dictionary size bound D is chosen to be 2^{16}, so that the word endpoints can be encoded in 16-bit pointers. The small size of the dictionary makes also more likely that (at least the most frequently accessed part of) it is kept in cache.

4.3 MONOTONE MINIMAL PERFECT HASH FOR STRINGS

Minimal perfect hash functions map a set of strings $\mathscr{S}$ bijectively into $[0, |\mathscr{S}|)$. *Monotone* minimal perfect hash functions [5] (or *monotone hashes*) also require that the mapping preserves the lexicographic order of the strings (not to be confused with generic order-preserving hashing, where the order to be preserved is arbitrary, thus incurring a $\Omega(|S| \log |S|)$ space lower bound). We remark that, as with standard minimal hash functions, the Lookup can return any number on strings outside of $\mathscr{S}$, hence the data structure does not have to *store* the string set.

The hollow trie [6] is a particular instance of monotone hash. It consists of a binary trie on $\mathscr{S}$, of which only the trie topology and the skips of the internal nodes are stored, in succinct form. To compute the hash value of a string x, a *blind search* is performed: the trie is traversed matching only the branching characters (bits, in this case) of x. If $x \in \mathscr{S}$, the leaf reached is the correct one, and its depth-first index is returned; otherwise, it has the longest prefix match with x, useful in some applications [38].

The cost of unbalancedness for hollow tries is even larger than that for normal tries: since the strings over Σ have to be converted to a binary alphabet, the height is potentially multiplied by $O(\log |\Sigma|)$ with respect to that of a trie on Σ. The experiments in [6] show indeed that the data structure has worse performance than the other monotone hashes analyzed in that paper, while it is among the most space-efficient.

PATH DECOMPOSITION WITH LEXICOGRAPHIC ORDER. To tackle their unbalancedness, we apply the centroid path decomposition idea to hollow tries. The construction presented in Section 4.2 cannot be used directly, because we want to both preserve the lexicographic ordering of the strings *and* guarantee the logarithmic height. However, both the binary alphabet and the fact that we do not need the Access operation come to the aid. First, inspired again by [39], we arrange the sub-tries in *lexicographic* order. This means that the sub-tries on the *left* of the path are arranged top-to-bottom, and precede all those on the *right* which are arranged bottom-to-top. In the path decomposition tree we

call *left* children the ones corresponding to sub-tries hanging off the left side of the path and *right* children the ones corresponding to those hanging on the right side. Figure 4.3 shows the new ordering.

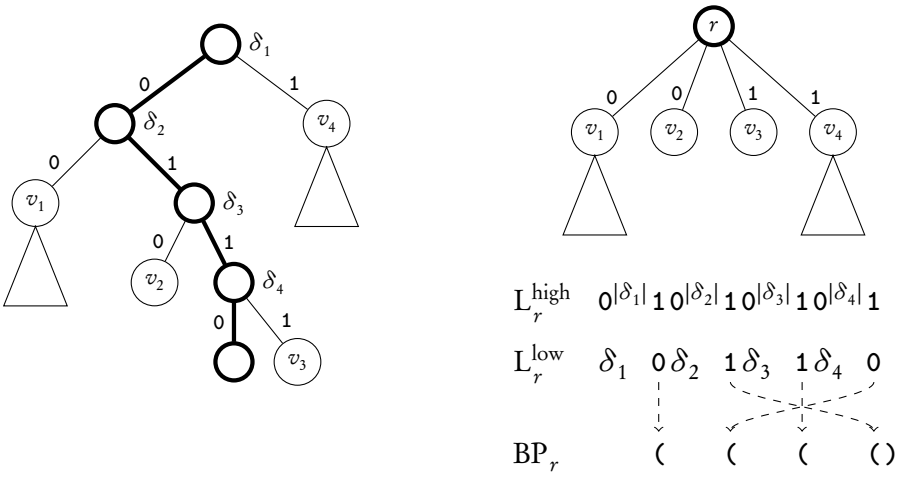


FIGURE 4.3: Path decomposition of a hollow trie. The δ_i denote the skips.

We now need a small change in the heavy path strategy: instead of breaking ties arbitrarily, we choose the *left* child. We call this strategy *left-biased heavy path*, which gives the following.

OBSERVATION 4.3.1 Every node-to-leaf left-biased heavy path in a binary trie ends with a left turn. Hence, every internal node of the resulting path decomposition has at least one right child.

Proof Suppose by contradiction that the path leaf is a right child. Then, either its left sibling is not a leaf, in which case the path is not heavy, or it is a leaf, then the tie would be resolved by choosing the left child. ■

TREE REPRESENTATION. The bitvector BP is defined as in Section 4.2. The label associated with each node is the sequence of skips interleaved with directions taken in the centroid path, excluding the leaf skip, as in Figure 4.3. Two aligned bitvectors L^{high} and L^{low} are used to represent the labels using an encoding inspired by γ codes [33]: the skips are incremented by one (to exclude `0` from the domain) and their binary representations (without the leading `1`) are interleaved with the path directions and concatenated in L^{low}. L^{high} consists of `0` runs of length corresponding to the lengths of the binary representations of the skips, followed by `1`s, so that the endpoints of (skip, direction) pair encodings in L^{low} are aligned to the `1`s in L^{high}. Thus a Select directory on L^{high} enables random access to the (skip, direction) pairs sequence. The labels of the node are concatenated in depth-first order: the `(`s in BP are in one-to-one correspondence with the (skip, direction) pairs.

TRIE OPERATIONS. As in Section 4.2, a trie traversal is simulated on the path decomposition tree. In the root node, the (skip, direction) pairs sequence is scanned (through L^{high} and L^{low}): during the scan the number of left and right children passed by is kept; when a mismatch in the string is found, the search proceeds in the corresponding child. Because of the ordering of the children, if the mismatch leads to a left child the child index is the number of left children seen in the scan, while if it leads to a right child it is the node degrees minus the number of right children seen (because the latter are represented from right to left). This correspondence between (skip, direction) pairs and child nodes (represented by `(`s in BP_v) is shown with dashed arrows in Figure 4.3. The search proceeds recursively until the string is fully traversed.

When the search ends, the depth-first order of the node found is not yet the number we are looking for: all the ancestors where we turned *left* come before the found node in depth-first but *after* it in the lexicographic order. Besides, if the found node is not a leaf, all the strings in the left sub-tries of the corresponding path are lexicographically smaller than the current string. It is easy to fix these issues: during the traversal we can count the number of left turns and subtract that from the final index. To account for the left sub-tries, using Observation 4.3.1 we can count the number of their leaves by jumping to the first right child with a FindClose: the number of nodes skipped in the jump is equal to the number of leaves in the left sub-tries of the node.

TIME COMPLEXITY. The running time of Lookup can be analyzed with a similar argument to that of the Lookup of Section 4.2: during the scan there cannot be more skips than the string length; besides, there is no binary search. Hence the number of operations is $O(\min(p, h))$, while the number of random memory accesses is bounded by $O(h)$.

IMPLEMENTATION NOTES. Since the `1`s in the sequence are at most 64 bits apart, to support Select on L^{high} we use the `darray64` variant of the `darray` [96], described in Section 7.2.2.

4.4 EXPERIMENTAL ANALYSIS

In this section we discuss a series of experiments we performed on both real-world and synthetic data. We performed several tests both to collect statistics that show how our path decompositions give an algorithmic advantage over standard tries, and to benchmark the implementations comparing them with other practical data structures.

We provide the source code at `http://github.com/ot/path_decomposed_tries` for the reader interested in further comparisons.

SETTING. The experiments were run on a 64-bit 2.53GHz Core i7 processor with 8MiB last-level cache and 24GiB RAM, running Windows Server 2008 R2. All the C++ code

was compiled with MSVC 10, while for Java we used the Sun JVM 6.

DATASETS. The tests were run on the following datasets.

— `enwiki-titles` (163MiB, 8.5M strings): All the page titles from English Wikipedia.

— `aol-queries` (224MiB, 10.2M strings): The queries in the AOL 2006 query log [2].

— `uk-2002` (1.3GiB, 18.5M strings): The URLs of a 2002 crawl of the `.uk` domain [13].

— `webbase-2001` (6.6GiB, 114.3M strings): The URLs in the Stanford WebBase from [78].

— `synthetic` (1.4GiB, 2.5M strings): The set of strings $\mathtt{d}^i\mathtt{c}^j\mathtt{b}^t\sigma_1\dots\sigma_k$ where i and j range in $[0,500)$, t ranges in $[0,10)$, σ_i are all distinct (but equal for each string) and $k = 100$. The resulting tries are very unbalanced while, at the same time, the strings are extremely compressible. Furthermore, the constant suffix $\sigma_1\dots\sigma_k$ stresses the linear search in data structures based on front coding such as HTFC, and the redundancy is not exploited by front coding and non-compressed tries.

AVERAGE HEIGHT. Table 4.1 compares the average height of plain tries with their path decomposition trees. In all the real-world datasets the centroid path decomposition cause a $\approx$ 2-3 times reduction in height compared to the standard compacted trie. The gap is even more dramatic in hollow tries, where the binarization of the strings causes a blow-up in height close to $\log|\Sigma|$, while the centroid path-decomposed tree height is very small, actually much smaller than $\log|\mathcal{S}|$. It is interesting to note that even if the lexicographic path decomposition is unbalanced, it still improves on the trie, due to the higher fan-out of the internal nodes.

The synthetic dataset is a pathological case for tries, but the centroid path-decomposition still maintains an extremely low average height.

STRING DICTIONARY DATA STRUCTURES. We compared the performance of our implementations of path-decomposed tries to other data structures. *Centroid* and *Centroid compr.* implement the centroid path-decomposed trie described in Section 4.2, in the versions without and with labels compression. Likewise, *Lex.* and *Lex. compr.* implement the lexicographic version.

Re-Pair and *HTFC* are respectively the Re-Pair and Hu-Tucker compressed Front Coding from [15]. For HTFC we chose bucket size 8 as the best space/time trade-off. Comparison with Front Coding is of particular interest as it is one of the data structures generally preferred by the practitioners.

	enwiki-titles	aol-queries	uk-2002	webbase-2001	synthetic
Compacted trie avg. height	9.8	11.0	16.5	18.1	504.4
Lex. avg. height	8.7	9.9	14.0	15.2	503.5
Centroid avg. height	5.2	5.2	5.9	6.2	2.8
Hollow avg. height	49.7	50.8	55.3	67.3	1005.3
Centroid hollow avg. height	7.9	8.0	8.4	9.2	2.8

TABLE 4.1: Average height: for tries the average height of the *leaves* is considered, while for path-decomposed tries all the nodes are considered (see the comments after Observation 4.2.2).

TX [110] is a popular open-source straightforward implementation of a (non-compacted) trie that uses LOUDS [68] to represent the tree. We made some small changes to avoid keeping the whole string set in memory during construction.

To measure the running times, we chose 1 million random (and randomly shuffled) strings from each dataset for the *Lookup* and 1 million random indices for the Access. Each test was averaged on 10 runs. The construction time was averaged on 3 runs.

Re-Pair, HTFC and TX do not support files bigger than 2GiB, so we could not run the tests on `webbase-2001`. Furthermore, Re-Pair did not complete the construction on `synthetic` in 6 hours, so we had to kill the process.

STRING DICTIONARIES RESULTS. The results of the tests can be seen in Table 4.2. On all datasets our compressed tries obtain the smallest space, except on `uk-2002` where they come a close second. The centroid versions have also the fastest Lookup times, while the Access time is better for Re-Pair and occasionally HTFC, whose time is although within 20% of that of the centroid trie. TX is consistently the largest and slowest on all the datasets.

Maybe surprisingly, the lexicographic trie is not much slower than the centroid trie for both Lookup and Access on real-world datasets. However, on the synthetic dataset the unbalanced tries are more than 20 times slower than the balanced ones. HTFC exhibits a less dramatic slowdown but still in the order of 5x on lookup compared to the centroid trie. Although this behavior does not occur on our real-world datasets, it shows that no assumptions can be made for unbalanced tries. For example in an adversarial environment an attacker could exploit this weakness to perform a denial of service attack.

We remark that the labels compression adds an almost negligible overhead in both Lookup and Access, due to the extremely simple dictionary scheme, while obtaining a very good compression. Hence unless the construction time is a concern (in which case other dictionary selection strategies can also be explored) it is always convenient to compress the labels.

	enwiki-titles 161 bps				aol-queries 185 bps				uk-2002 621 bps				webbase-2001 497 bps				synthetic 4836 bps			
	String dictionaries																			
	ctps	c.ratio	lkp	acs	ctps	c.ratio	lkp	acs	ctps	c.ratio	lkp	acs	ctps	c.ratio	lkp	acs	ctps	c.ratio	lkp	acs
Centroid compr.	6.1	**32.1%**	**2.5**	2.6	7.9	**31.5%**	2.7	2.7	8.5	**13.6%**	3.8	4.9	7.8	**13.5%**	4.8	**5.4**	13.7	**0.4%**	**4.2**	**13.5**
Lex. compr.	6.4	**31.9%**	3.2	3.1	8.0	**31.2%**	3.8	3.6	8.4	**13.5%**	5.9	6.6	8.5	**13.3%**	7.3	7.7	109.2	**0.4%**	90.9	96.3
Centroid	1.8	53.6%	**2.4**	2.4	2.0	55.6%	**2.4**	2.6	2.3	22.4%	**3.4**	4.2	**2.2**	24.3%	**4.3**	**5.0**	8.4	17.9%	5.1	**13.4**
Lex.	2.0	52.8%	3.1	3.2	2.2	55.0%	3.5	3.5	2.7	22.3%	5.5	6.2	2.6	24.3%	7.0	7.4	102.8	17.9%	119.8	114.6
Re-Pair [15]	60.0	41.5%	6.6	**1.2**	115.4	38.8%	7.3	**1.3**	326.4	**12.4%**	25.7	**3.1**	-	-	-	-	-	-	-	-
HTFC [15]	**0.4**	43.2%	3.7	2.2	**0.4**	40.9%	3.8	2.2	**0.9**	24.4%	7.0	4.7	-	-	-	-	**5.0**	19.1%	22.0	18.0
TX [110]	2.7	64.0%	9.7	9.1	3.3	69.4%	11.9	11.3	5.7	30.0%	42.1	42.0	-	-	-	-	44.6	25.3%	284.3	275.9
	Monotone hashes																			
	ctps	bps	lkp		ctps	bps	lkp		ctps	bps	lkp		ctps	bps	lkp		ctps	bps	lkp	
Centroid hollow	1.1	**8.40**	2.7		1.2	**8.73**	**2.8**		1.5	8.17	**3.3**		1.5	8.02	**4.4**		8.6	9.96	**11.1**	
Hollow	1.3	**7.72**	6.8		1.3	**8.05**	7.2		1.7	**7.48**	9.3		1.7	**7.33**	13.9		9.5	9.02	137.1	
Hollow [108]	**0.9**	**7.66**	14.6		**1.0**	**7.99**	16.6		**1.1**	**7.42**	18.5		**0.9**	**7.27**	22.4		**4.3**	**6.77**	462.7	
PaCo [108]	2.6	8.85	**2.4**		2.9	8.91	3.1		4.7	10.65	4.3		18.4	9.99	4.9		21.3	13.37	51.1	

TABLE 4.2: Experimental results. *bps* is *bits per string*, *ctps* is the average *construction time* per string, *c.ratio* is the *compression ratio* between the data structure and the original file sizes, *lkp* is the average Lookup time and *acs* the average Access time. All times are expressed in microseconds. The results within 10% of the best are in bold.

MONOTONE HASH DATA STRUCTURES. For monotone hashes, we compared our data structures with the implementations in [6]. *Centroid hollow* implements the centroid path-decomposed hollow trie described in Section 4.3. *Hollow* is a reimplementation of the hollow trie of [6], using a Range Min tree in place of a pioneer-based representation and the encoding described in Section 2.6.3. *Hollow (Sux)* and *PaCo (Sux)* are two implementations from [6]; the first is the hollow trie, the second a hybrid scheme: a Partially Compacted trie is used to partition the keys into buckets, then each bucket is hashed with an MWHC function. Among the structures in [6], PaCo gives the best trade-off between space and lookup time. The implementations are freely available as part of the Sux project [108].[2]

To measure construction time and lookup time we adopted the same strategy as for string dictionaries. For Sux, as suggested in [6], we performed 3 runs of lookups before measuring the lookup time, to let the JIT warm up and optimize the generated code.

MONOTONE HASH RESULTS. Table 4.2 shows the results for monotone hashes. On all real-world datasets the centroid hollow trie is ≈ 2-3 times faster than our implementation

[2]To be fair we need to say that Sux is implemented in Java while our structures are implemented in C++. However, the recent developments of the Java Virtual Machine have made the abstraction penalty gap smaller and smaller. Low-level optimized Java (as the one in Sux) can be on par of C++ for some tasks, and no slower than 50% with respect to C++ for most other tasks. We remark that the hollow trie construction is actually *faster* in the Sux version than in ours, although the algorithm is very similar.

of the hollow trie and ≈ 5 times faster than the Sux implementation. The centroid hollow trie is competitive with PaCo on all datasets, while taking less space and with a substantially simpler construction. The synthetic dataset in particular triggers the pathological behavior on all the unbalanced structures, with *Hollow*, *Hollow (Sux)* and *PaCo* being respectively 13, 41 and 5 times slower than *Centroid hollow*. Such a large performance gap suggests the same conclusion reached for string dictionaries: if *predictable* performance is needed, unbalanced structures should be avoided.

5 Top-k string completion

Virtually every modern application, either desktop, web, or mobile, features some kind of auto-completion of text-entry fields. Specifically, as the user enters a string one character at a time, the system presents k suggestions to speed up text entry, correct spelling mistakes, and help users formulate their intent. In its basic form, the suggestions are drawn from a static set of strings, each associated with a score. We call such a set a *scored string set*.

Definition 5.0.1 (Scored string set) A *scored string set* $\mathcal{S}$, $|\mathcal{S}| = n$, is a set of n pairs (s, r) where $s \in \Sigma^*$ is a string drawn from an alphabet Σ and r is an integer *score*.

Given a prefix string, the goal is to return the k strings matching the prefix with the highest scores. Formally, we define the problem of top-k completion as follows.

Definition 5.0.2 (Top-k completion) Given a string $p \in \Sigma^*$ and an integer k, a *top-k completion* query in the scored string set $\mathcal{S}$ returns the k highest scored pairs in $\mathcal{S}_p = \{(s, r) \in \mathcal{S} \mid p \text{ is a prefix of } s\}$ (or the whole set if $|\mathcal{S}_p| < k$).

In order to cover a large part of user intents, the string set $\mathcal{S}$ should contain as many strings as possible. For query auto-completion in web search engines, such set can easily be in the order of hundreds of millions of queries. Similarly, for predictive text entry, a database of n-grams can reach or even surpass these sizes. For this reason, it is extremely important to represent the scored string set in as little space as possible, while still supporting top-k completion queries efficiently. Luckily, many of the top-k completion application scenarios exhibit special properties which we can take advantage of to improve the space and time efficiency of the system. First, the scores associated with the strings often exhibit a power law distribution. Thus, most of the queries have low counts as scores that require only a few bits to encode. Second, the distribution of the prefixes that users enter one character at a time often approximates the distribution of the scores. In other words, in practical usages of top-k completion systems, prefixes of entries with higher scores tend to be queried more than those associated with lower scored entries. In fact, a common folklore optimization in practical trie implementations

is to sort the children of each node by decreasing score to speed up the lookup. Third, a large number of strings share common substrings; this makes them highly compressible.

In this chapter, we present an application of the path-decomposed trie of Chapter 4, where we apply a score-dependent path decomposition in order to support efficient top-k completion queries. We call this data structure *Score-Decomposed Trie*.

To experimentally compare this data structure against a baseline, we implemented a simple scheme based on a lexicographic string dictionary augmented with an RMQ data structure on the vector of the scores. We call this data structure *RMQ Trie*.

As we will see in Section 5.5, the Score-Decomposed Trie is both slightly smaller and significantly faster than the RMQ Trie.

5.1 Related Work

There is a vast literature on ranked retrieval, both in the classical and succinct settings. We report here the results closest to our work.

Using classical data structures, various studies have examined the task of word/phrase completion [87, 80, 81, 93, 111], though most do not consider datasets of more than a million strings or explore efficient algorithms on compressed data structures. In [80], Li et al. precompute and materialize the top-k completions of each possible word prefix and store them with each internal node of a trie. This requires a predetermined k and is space inefficient. Recently, Matani [87] described an index similar in principle to the proposed RMQ Trie structure in Section 5.2, but using a suboptimal data structure to perform RMQ. Although the system achieves sub-millisecond performance, both this and the previous work require storing the original string set in addition to the index.

From a theoretical point of view, Bialynicka-Birula and Grossi [10] introduced the notion of *rank-sensitive* data structures, and presented a generic framework to support ranked retrieval in range-reporting data structures, such as suffix trees and tries. However, the space overhead is superlinear, which makes it impractical for our purposes.

As the strings are often highly compressible, we would like data structures that approach the theoretic lower bound in terms of space. In this direction, recent advances have yielded many implementations of string dictionaries based on succinct data structure primitives [60, 15].

Hon et al. [64] used a combination of compressed suffix arrays [62, 41] and RMQ data structures to answer *top-k document retrieval* queries, which ask for the k highest-scored documents that contain the queried pattern as a *substring*, in compressed space. While this approach is strictly more powerful than top-k completion, as shown in [15], string dictionaries based on compressed suffix arrays are significantly slower than prefix-based data structures such as front-coding, which in turn is about as fast as compressed tries [60]. The RMQ Trie of Section 5.2 uses a similar approach, but it is based on a trie instead of a suffix array.

5.2 RMQ TRIE

In this section, we describe a simple scheme to *augment* any sorted string dictionary data structure with an RMQ data structure, in order to support top-k completion. We will use it as a baseline in our experiments.

As shown in Figure 5.1, if the string set $\mathcal{S}$ is represented with a trie, the set $\mathcal{S}_p$ of strings prefixed by p is a subtrie, hence, if the scores are arranged in DFS order in an array R, the scores of $\mathcal{S}_p$ are those in an interval $\mathrm{R}[a, b]$. This is true in general for any string dictionary data structure that maps the strings in $\mathcal{S}$ to $[0, |\mathcal{S}|)$ in lexicographic order. We call PrefixRange(p) the operation that, given p, returns the pair (a, b), or $\perp$ if no string matches the prefix.

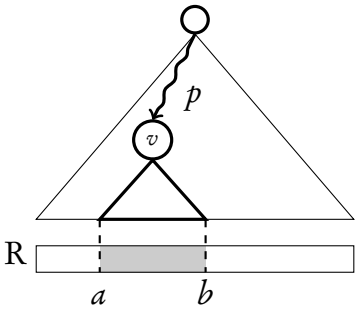


FIGURE 5.1: The scores of the strings prefixed by p correspond to the interval $[a, b]$ in the scores vector R.

To enumerate the completions of p in ranked order, we employ a standard recursive technique, used for example in [92, 64]. We build an RMQ data structure on top of R using an inverted ordering, i.e. the *minimum* is the highest score. The index of the first completion is then $i = \mathrm{RMQ}_{\mathrm{R}}(a, b)$. Now the index of the second completion is the one with highest score among $\mathrm{RMQ}(a, i-1)$ and $\mathrm{RMQ}(i+1, b)$, which splits again either $[a, i-1]$ or $[i+1, b]$ into two subintervals. In general, the index of the next completion is the highest scored RMQ among all the intervals obtained with this recursive splitting. By maintaining the intervals in a priority queue ordered by score, it is hence possible to find the top-k completion indices in $\mathrm{O}(k \log k)$. We can then perform k Access operations on the dictionary to retrieve the strings.

The space overhead of this data structure, beyond the space needed to store the trie and the scores, is just the space needed for the RMQ data structure, which is $2n + o(n)$ bits, where $n = |\mathcal{S}|$. If the trie supports PrefixRange of the prefix in time $\mathrm{T_P}$ and Access in time $\mathrm{T_A}$, the total time to retrieve the top-k completions is $\mathrm{O}(\mathrm{T_P} + k(\mathrm{T_A} + \log k))$.

The advantages of this scheme are its simplicity and modularity, since it is possible to re-use an existing dictionary data structure without any significant modification. In our experiments we use the lexicographic compressed trie of Chapter 4. The only change we needed to make was to implement the operation PrefixRange.

5.3 SCORE-DECOMPOSED TRIE

In this section, we introduce a compressed trie data structure specifically tailored to solve the top-k completion problem. The structure is based on the succinct path-decomposed tries described in Chapter 4, but with a different path decomposition that takes into account the scores.

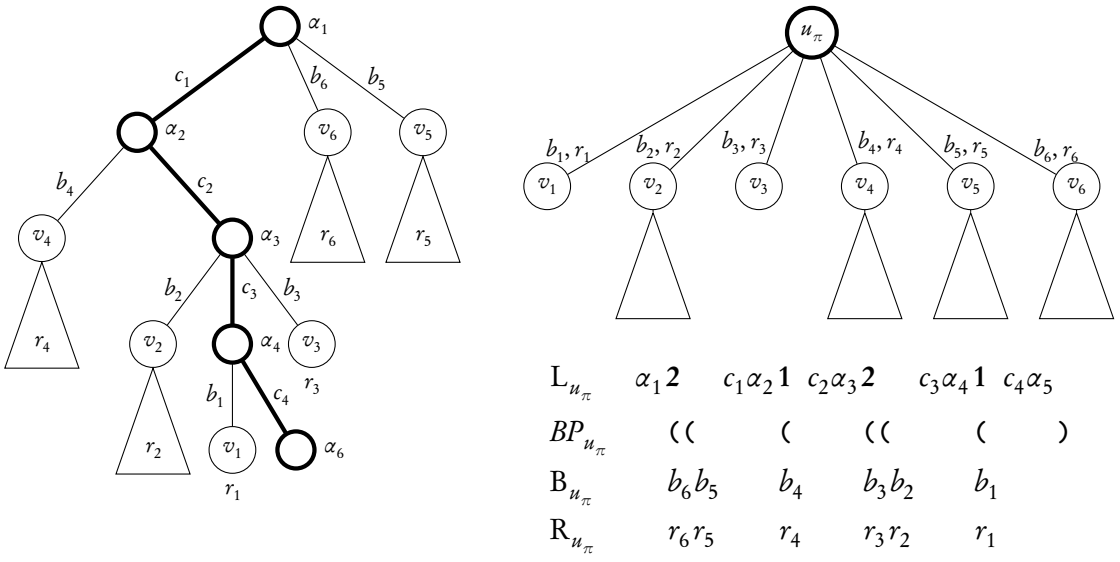


FIGURE 5.2: On the left, trie $\mathcal{T}$ with the decomposition path π highlighted. On the right, root node u_π in $\mathcal{T}^c$ and its encoding (spaces are for clarity only). In this example v_6 is arranged after v_5 because $r_5 > r_6$.

MAX-SCORE PATH DECOMPOSITION. Since each string corresponds to a leaf in $\mathcal{T}$, we can associate its score with the corresponding leaf. To define the path decomposition, we describe the strategy used to choose the decomposition path π and to order the subtries hanging off π as children of the root u_π. We define the *max-score path decomposition* as follows. We choose the path π as the root-to-leaf path whose leaf has the highest score (ties are broken arbitrarily). The subtries are ordered bottom-to-top, while subtries at the same level are arranged in decreasing order of score (the score of a subtrie is defined as the highest score in the subtrie).

To enable scored queries, we need to augment the data structure to store the scores. Following the notation of Figure 5.2, let u_π be the root node of $\mathcal{T}^c$ and $v_1, \ldots, v_d$ the nodes hanging off the path π. We call r_i the *highest score* in the subtrie rooted at v_i (if v_i is a leaf, r_i is just its corresponding score). We add r_i to the label of the edge leading to the corresponding child, such that the label becomes the pair (b_i, r_i).

ab	4
b	2
bba	1
caca	3
caccc	1
cbac	2
cbba	1

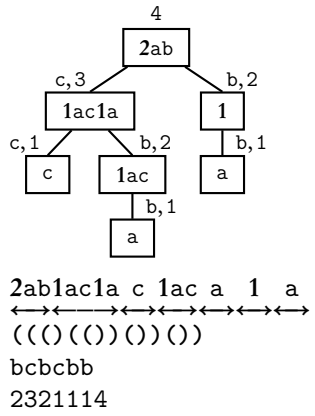


FIGURE 5.3: Score-Decomposed Trie example and its encoding.

SUCCINCT TREE REPRESENTATION. To represent the Score-Decomposed Trie, we use the same encoding described in Chapter 4; in addition, we need to store the scores in the edges along with the branching characters. We follow the same strategy used for the branching characters: we concatenate the r_i's in reverse order into a sequence R_{u_π}, and then concatenate the sequences R_u for each node u into a sequence R in DFS order. Finally, we append the root score to R. To compress the sequence R, we use the data structures described in Section 5.4.

TOP-k COMPLETIONS ENUMERATION. The operations Lookup and Access do not need any modification, as they do not depend on the particular path decomposition strategy used. We now describe how to support top-k completion queries.

Because of the *max-score* decomposition strategy, the highest score in each subtrie is exactly the score of the decomposition path for that subtrie. Hence if r_i is the highest score of the subtrie rooted in v_i, and u_i is the node in $\mathscr{T}^c$ corresponding to that subtrie, then r_i is the score of the string corresponding to u_i. This implies that for each (s, r) in $\mathscr{S}$, if u is the node corresponding to s, then r is stored in the incoming edge of u, except when u is the root u_π, whose score is stored separately. Another immediate consequence of the decomposition is that the tree has the *heap property*: the score of each node is less or equal to the score of its parent.

We exploit this property to retrieve the top-k completions. First, we follow the algorithm of the Lookup operation until the prefix p is exhausted, leading to the *locus node* u, that is the highest node whose corresponding string is prefixed by p. This takes time $\mathrm{O}(|\Sigma||p|)$. By construction, this is also the highest scored completion of p, so we can immediately report it. To find the next completions, we note that the prefix p ends at some position i in the label L_u. Thus, all the other completions must be in the subtrees whose roots are the children of u branching *after* position i. We call the set of such children the *seed set*, and add them into a priority queue.

To enumerate the completions in sorted order, we extract the highest scored node

from the priority queue, report the string corresponding to it, and add all its children to the priority queue. For the algorithm to be correct, we need to prove that, at each point in the enumeration, the node corresponding to the next completion is in the priority queue. This follows from the fact that every node u corresponding to a completion must be reached at some point, because it is a descendant of the seed set. Suppose that u is reported after a lower-scored node u'. This means that u was not in the priority queue when u' was reported, implying that u is a descendant of u'. But this would violate the heap property.

The previous algorithm still has a dependency on the number of children in each node, since all of them must be placed in the priority queue. With a slight modification in the algorithm, this dependency can be avoided. Note that in the construction, we sort the children branching off the same branching point in decreasing score order. Thus, we can delay the insertion of a node into the priority queue until after all other higher-scored nodes from the same branching point have already been extracted. For each node u, the number of branching points in L_u is at most $|\mathrm{L}_u|$. Hence, we add at most $|\mathrm{L}_u| + 1$ nodes to the priority queue: 1 for each branching point plus the next sibling, if any, of the extracted node at its branching point. Thus, the time to return k completions is $\mathrm{O}(l k \log l k)$ where l is the average length of the completions returned minus the prefix length $|p|$.

Note that, after the locus node is found, only $k - 1$ nodes need to be visited in order to return k completions. This property makes the Score-Decomposed Trie very suitable for succinct representations, where traversal operations constitute the bottleneck in the overall performance.

5.4 Score Compression

In both the data structures described in Section 5.2 and Section 5.3 it is necessary to store the array R of the scores, and perform random access quickly. It is crucial to effectively compress the scores: if stored, say, as 64 bit integers, they would take more than half of the overall space.

As noted in the introduction, many scoring functions (number of clicks/impressions, occurrence probability, ...) exhibit a power law distribution. Under this assumption, encoding the scores with γ-codes [33] (or, more in general, ζ-codes [14]) would give nearly optimal compression, but it would not be possible to support efficient random access to the array.

We implemented instead two different data structures for storing the scores, with different time/space trade-offs. In the first one, which we refer to as γ-array, we juxtapose the binary representations $\alpha_1, \alpha_2, \ldots$ of the integers (without the leading 1s) in a bitvector $\mathrm{L} = \alpha_1 \alpha_2 \ldots$. In another bitvector H we store a run of 0s for each α_i with the same length, followed by a 1, and prepend a 1 to the sequence, i.e. $\mathrm{H} = 1 0^{|\alpha_1|} 1 0^{|\alpha_2|} 1 \ldots$. To retrieve the i-th integer it is sufficient to note that its binary representation α_i is located in L

between position $\text{Select}_{\text{H}}(i) - i$ and position $\text{Select}_{\text{H}}(i+1) - i - 1$. Note also that the space occupancy is exactly the same as if the scores were compressed with γ-codes, plus the redundancy needed to support efficiently Select_{H}. To this end, in our implementation we use the `darray64` described in Section 7.2.2;

The second data structure which we call *packed-blocks array*, is inspired by *Frame of Reference* compression [52]. The scores array is divided into blocks of length l; within each block j the scores are encoded with b_j bits each, where b_j is the minimum number of bits sufficient to encode each value in the block. The block encodings are then juxtaposed in a bitvector B. To retrieve the endpoints of the blocks inside B we employ a two-level directory structure: the blocks are grouped into *super-blocks* of size L, and the endpoints of each block are stored relative to the beginning of the superblock in $O(\log(Lw))$ bits each, where w is the size in bits of the largest representable value. The endpoints of the superblock are then stored in $O(\log(nw))$ bits each. To retrieve a value, the endpoints of its block are retrieved using the directory structure; then b_j is found by dividing the size of the block by l. The overall time complexity is constant. In our implementation, we use $l = 16$, $L = 512$, 16-bit integers for the block endpoints, and 64-bit integers for the super-block endpoints.

5.5 Experimental Analysis

To evaluate the performance of the Score-Decomposed Trie (`sdt`), and of the baseline RMQ Trie (`rt`), we performed an experimental analysis on three datasets coming from different application scenarios.

Datasets. The tests were run on the following datasets.

— `AOL`: The set of queries and their impression counts from the AOL query log [2]. It consists of 10M unfiltered queries.

— `Wikipedia`: The set of page titles requested to Wikipedia in October 2012, and their request counts. The dataset was obtained from the daily request logs [116]. The titles were normalized (lowercasing, whitespace normalization, removal of trailing slashes) and only the titles with at least 5 requests were kept. This resulted in a dataset with 27M string-score pairs.

— `Unigrams`: All the unigrams from the Google English One Million n-grams dataset [56], together with the all-time occurrence counts. The size of the dataset is 3M string-score pairs.

In each dataset we subtracted from the scores their minimum, so that the smallest score is `0`. The minimum is then added back at query time.

TESTING DETAILS. The data structures have been implemented in C++ and compiled with g++ 4.7. The tests were run on a dual core Intel Core 2 Duo E8400 with 6MiB L2 cache and 4GiB RAM, running Linux 3.5.0 - 64-bit. Each test was run 10 times, and the running times averaged.

We tested the `rt` and `sdt` data structures in the `gamma` and `packed` variants, where `gamma` uses a γ-array to compress the scores vector R while `packed` uses a packed-blocks array.

For each dataset we sampled 1M strings with replacement according to the score distribution, and shuffled them randomly. On each data structure, we perform a top-10 query all the prefixes of length from 1 to 20 of each sampled string, and measure the average time per completion returned, denoted with cmpl in Table 5.2, To compare this number to the time to Access a random string-score pair, we also sampled 1M random indexes and measured the average Access time, denoted with acs in Table 5.2.

Dataset	raw	gzip	sdt_packed			sdt_gamma			rt_packed			rt_gamma		
			Total	S	R	Total	S	R	Total	S	R	Total	S	R
AOL	201.8	55.9	62.4	58.4	4.1	61.0	58.4	2.6	65.5	57.9	7.6	63.2	57.9	5.3
Wikipedia	224.1	64.1	68.8	59.8	9.0	67.7	59.8	8.0	72.0	59.3	12.7	69.9	59.3	10.7
Unigrams	100.1	38.3	35.6	22.3	13.3	37.4	22.3	15.1	39.4	21.7	17.7	39.5	21.7	17.8

TABLE 5.1: Space in bits per string-score pair.

RESULTS. Table 5.1 summarizes the average bits per score-string pair. `raw` is the space used to store the pairs in TAB-separated format (with the scores expressed in decimal), `gzip` the space for the `raw` file compressed with `gzip`. For {`sdt`, `rt`}_{`packed`, `gamma`}, Total represents the total bits per string-score pair, which can be broken down into the space taken by the strings S and that taken by the scores R. Note that for `rt` R includes the 2.7 bits per score needed by the RMQ data structure.

The `sdt` variants are the smallest across all datasets, attaining space close to `gzip` on `AOL` and `Wikipedia`, and even smaller on `Unigrams`. The `rt` variants are slightly larger, due to the additional space needed by the RMQ data structure. As predicted in Section 5.4, the γ-array obtain very good compression on all the datasets, while the packed-blocks array is slightly larger on `AOL` and `Wikipedia`, and slightly smaller on `Unigrams`. We attribute the good performance of the packed-blocks array to the fact that the scores are arranged in DFS order, which implies that the scores of strings which share long common prefixes are *clustered* in the array R. Since the scores of such strings are likely to be in the same order of magnitude, they require approximately the same number of bits to be represented. Hence, the waste induced by using the same number of bits for a block of adjacent values is relatively small.

Table 5.2 summarizes the average time for the queries. As expected, `sdt` is significantly faster than `rt`, with `sdt_packed` being about 4 times faster than `rt_packed` on `AOL` and

Wikipedia, and about 6 times faster on Unigrams. Note that the average time per completion cmpl for rt is close to that of accessing a random index in the trie, denoted as acs. On the other hand, with sdt the completion time is significantly smaller than the access time. As noted in Section 5.3, this is due to the fact that, after finding the node corresponding to the queried prefix, only one node per completion has to be accessed.

Dataset	sdt_packed		sdt_gamma		rt_packed		rt_gamma	
	cmpl	acs	cmpl	acs	cmpl	acs	cmpl	acs
AOL	0.8	3.0	1.0	3.0	3.5	3.3	3.6	3.5
Wikipedia	0.9	3.7	1.3	3.9	3.8	4.3	4.0	4.6
Unigrams	0.4	1.7	0.7	1.8	2.3	1.7	2.4	1.9

TABLE 5.2: Average time per found completion in microseconds on a top-10 query (cmpl), and average time to Access a random string-score pair (acs)

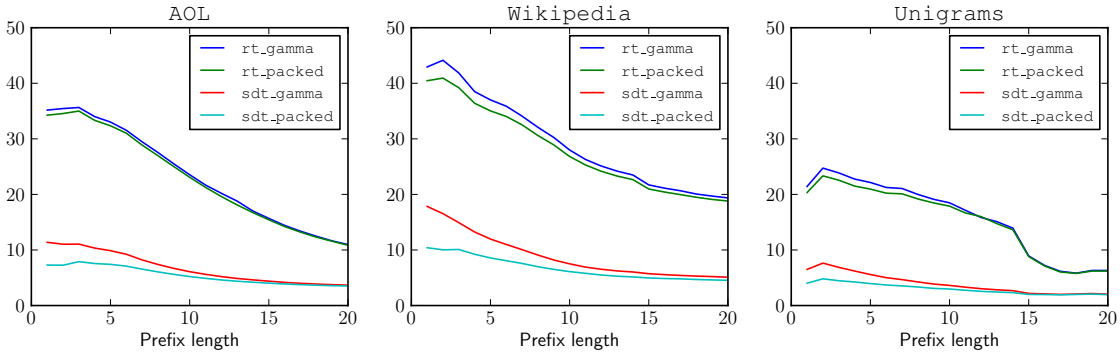


FIGURE 5.4: Time in microseconds to perform a top-10 query for different prefix lengths.

Figure 5.4 shows the average time for a top-10 query for different prefix lengths. It is immediate to see that for rt the time decreases as the prefix length increases, because the number of matched completions decreases as well. On the other hand, the performance of sdt is more consistent across prefix lengths, and faster than rt in all cases.

Regarding space compression, γ-array impose a severe slowdown on sdt when compared to the packed-blocks array, since whenever a node is traversed, one score for each branching point in the label must be retrieved. The slowdown is less pronounced for rt, where only the scores associated with the top-k completions are retrieved. Since the space overhead due to the packed-blocks array is negligible compared to γ-array (and in the case of Unigrams the overall space is even smaller), the former should be preferred in most cases.

5.6 Further considerations

In practical scenarios, auto-completion needs to support not only exact prefix matches, but also inexact matches due to differences in casing, accents, or spelling. One way to support case and accent insensitive match is to normalize both the dataset strings and the input prefix into lowercase unaccented characters before computing the completions. However, this removes the original casing and accents from the completions, which may be important for certain languages and scenarios.

An alternative technique is to use an approximate prefix matching algorithm on the trie, resulting in one or more candidate locus nodes. The top-k results for each of these nodes can be merged together to return the overall top-k approximate completions. For example, the algorithm described by Duan and Hsu [31] uses an error model learned from a database of user spelling mistakes to perform an approximate prefix search in a trie. Since both the RMQ Trie and the Score-Decomposed Trie support all the standard trie traversal operations, they support any algorithm that can be implemented on a trie.

In addition, some applications need to retrieve the top-k completions according to a *dynamic* score that depends on the prefix and completion. However, as the static score is usually a prominent component of the dynamic score, an approximate solution can be obtained by taking the top-k' completions with $k' > k$ according to the static score and re-ranking the completion list.

6 Indexed sequences of strings

Storage and indexing of sequences of strings lies at the core of many problems in databases and information retrieval. Column-oriented databases represent relations by storing individually each column as a sequence; if each column is indexed, efficient operations on the relations are possible. XML databases, taxonomies, and word tries are represented as labeled trees, that can be mapped to the sequence of its labels in a specific order; indexed operations on the sequence enable fast tree navigation. In data analytics query logs and access logs are simply sequences of strings; aggregate queries and counting queries can be performed efficiently with specific indexes. Textual search engines essentially reduce to representing a text as the sequence of its words, and queries locate the occurrences of given words in the text. Knowledge graphs and social graphs can be represented as binary relations on the set of their entities, which are identified by strings. Even the storage of non-string (for example, numeric) data can be often reduced to the storage of strings, as usually the values can be binarized in a natural way.

In addition to the standard exact search and count operations, in several applications it is necessary to support *prefix-based* operations. For example, in data analytics of query logs and access logs, the accessed URLs, paths (filesystem, network, ...), or any kind of hierarchical references can be stored in time order as a sequence $S = \langle S_0, \ldots, S_{n-1} \rangle$ of strings. Common prefixes denote common domains or a common parent directories, and sequence ranges $S[i', i)$ correspond to intervals of time. In this scenario, typical queries involve access statistics and reporting, that is, prefix-based range searching and counting (e.g. "what has been the most accessed domain during winter vacation?").

Another common requirement in most applications is that the sequence can change over time, either by updates at arbitrary positions (for example, in database applications), or by appending new strings at the end of the sequence (as in logging applications).

In this chapter we introduce and study the problem of *compressed indexed sequences of strings*, i.e. representing sequences of strings in nearly optimal compressed space, both in the static and dynamic settings, while supporting provably efficient access, search, and count (and update, in the dynamic setting) operations.

PROBLEM DEFINITION. Let S be a sequence of strings, as defined in Section 2.2. The set of distinct strings occurring in S is denoted as S_{set}.

An *indexed sequence of strings* is a data structure for storing S that supports random access, searching, and range counting, both for exact matches and prefix search. Specifically, we require that the data structure supports Access, Rank, and Select on S. By composing these three primitives it is possible to implement other powerful index operations. For example, functionality similar to inverted lists can be easily formulated in terms of Select. In addition, we introduce the following prefix-based versions of Rank and Select.

— $\text{RankPrefix}_S(p, i)$: return the number of strings in $S[0, i)$ that have prefix p.

— $\text{SelectPrefix}_S(p, i)$: find the position of the i-th string in S among those that have prefix p.

As usual, we will drop the subscript S when there is no risk of ambiguity.

Many operations can be easily formulated in terms of RankPrefix and SelectPrefix. Other useful operations, such as distinct values reporting, and majority element, are described in Section 6.4.

We define a *dynamic* version of the data structure, where the sequence S is allowed to change over time, by introducing the following operations, for any arbitrary string s.

— $\text{Insert}_S(s, i)$: update the sequence S as $S' = \langle S_0, \ldots, S_{i-1}, s, S_i, \ldots, S_{n-1} \rangle$ by inserting s immediately before S_i, or at the end if $i = |S|$.

— $\text{Append}_S(s)$: update the sequence S as $S' = \langle S_0, \ldots, S_{n-1}, s \rangle$ by appending s at the end. This is equivalent to $\text{Insert}_S(s, |S|)$.

— $\text{Delete}_S(i)$: update the sequence S as $S' = \langle S_0, \ldots, S_{i-1}, S_{i+1}, \ldots, S_{n-1} \rangle$ by deleting S_i.

Note that the above operations are can change S_{set}: if a previously unseen string is inserted or appended, S_{set} grows by one element; if the last occurrence of a string is deleted, S_{set} shrinks by one element.

WAVELET TRIE. To support the above defined operations, we introduce a new data structure, the *Wavelet Trie*. The Wavelet Trie is a generalization for string sequences of the wavelet tree, where the tree shape is given by the Patricia trie of the string set S_{set} of the sequence S. This enables efficient prefix operations and the ability to grow or shrink S_{set} as values are inserted or deleted. We first present a static version of the Wavelet Trie in Section 6.2. We then give in Section 6.3 an append-only dynamic version of the Wavelet Trie, meaning that elements can be appended only at the end, and a fully dynamic version.

Our time bounds are reported in Table 6.1. For a string s, let h_s denote the number of nodes traversed when looking up s in the binary Patricia Tree of $S_{set} \cup \{s\}$. Observe that

$h_s \leq |s| \log |\Sigma|$, where Σ is the alphabet of symbols from which s is drawn, and $|s| \log |\Sigma|$ is the length in bits of s (while $|s|$ denotes its number of symbols as usual). The cost for the queries on the static and append-only versions of the Wavelet Trie is $O(|s| + h_s)$ time, which is the *same* cost as searching in the binary Patricia trie, and appending s to S takes $O(|s| + h_s)$ time as well. The cost of the operations for the fully dynamic version incur a $O(\log n)$ slowdown. In both the append-only and fully-dynamic versions, there is no need to know S_{set} in advance.

	Query	Append	Insert	Delete	Space (in bits)
Static	$O(\|s\| + h_s)$	–	–	–	$LB + o(\tilde{h}n)$
Append-only	$O(\|s\| + h_s)$	$O(\|s\| + h_s)$	–	–	$LB + PT + o(\tilde{h}n)$
Fully-dynamic	$O(\|s\| + h_s \log n)$	$O(\|s\| + h_s \log n)$	$O(\|s\| + h_s \log n)$	$O(\|s\| + h_s \log n)^\dagger$	$LB + PT + O(nH_0)$

TABLE 6.1: Wavelet Trie bounds. *Query* is the cost of Access, Rank(Prefix), Select(Prefix), LB is the information theoretic lower bound $LT + nH_0$, and PT the space taken by the dynamic Patricia trie. †Deletion may take $O(\hat{\ell} + h_s \log n)$ time when deleting the last occurrence of a string, where $\hat{\ell}$ is the maximum string length in S_{set}.

All versions are nearly optimal in space as shown in Table 6.1. In particular, the lower bound LB(S) for storing an indexed sequence of strings can be derived from the lower bound $LT(S_{set})$ for storing S_{set} given in [39] plus the zero-order entropy bound $nH_0(S)$ for storing S as a sequence of symbols. The static version uses an additional number of bits that is just a lower order term $o(\tilde{h}n)$, where $\tilde{h}$ is the average height of the wavelet tree (Definition 6.2.4). The append-only version only adds $PT(S_{set}) = O(|S_{set}|w)$ bits for keeping $O(|S_{set}|)$ pointers to the dynamically allocated memory (assuming that we do not have control on the memory allocator on the machine). The fully dynamic version has a redundancy of $O(nH_0(S))$ bits.

6.1 RELATED WORK

Traditionally, indexed sequences of strings are stored by representing the sequence explicitly and indexing it using auxiliary data structures, such as B-Trees, Hash Indexes, Bitmap Indexes. These data structures have excellent performance and both external and cache-oblivious variants are well studied [113]. Space efficiency is however sacrificed: the total occupancy at least double the space of the sequence alone.

In the succinct data structures literature, instead, most compressed Rank/Select data structures assume that the alphabet from which the sequences are drawn is *integer* and *contiguous*, i.e the alphabet is $\{0, \ldots, \sigma - 1\}$. Non-integer alphabets (for sequences of strings, S_{set}) need to be mapped first to an integer range, as implicitly done, for example, in [20, 40]. However, in the mapping the string structure is lost, hence no prefix operations can be supported. Even if the mapping is lexicographic, which would allow an efficient

implementation of RankPrefix as a two-dimensional range query [84], it is not clear whether SelectPrefix can be supported efficiently.

In case prefix operations are not needed, dynamic variants of wavelet trees have been presented recently [79, 55, 85]. However, they all assume that the alphabet is known a priori, as the tree structure is static. This constraint is unacceptable in many applications, such as database storage, because the set of values of a column (or even its cardinality) is rarely known in advance; similarly in text indexing a new document can contain unseen words; in URL sequences, new URLs can be created at any moment. In fact, the existence of wavelet trees with dynamic alphabet was left as an open question in [55] and [85]. The Wavelet Trie was the first such data structure, answering positively the question.

Very recently, Navarro and Nekrich [95] introduced a dynamic compressed wavelet tree with optimal $O(\log n/\log\log n)$ time operations which supports unbounded alphabets. Using their data structure it is possible to maintain a sequence of strings by using a dynamic string dictionary on S_{set}, with an additive slowdown depending on the dictionary data structure. However, prefix operations are not supported, and the update time bounds are amortized, while the Wavelet Trie time bounds are worst-case.

6.2 The Wavelet Trie

We informally define the *Wavelet Trie* of a sequence of *binary* strings S as a wavelet tree on S (seen as a sequence on the alphabet $\Sigma = S_{set} \subset \{0,1\}^*$) whose tree structure is given by the Patricia trie of the string set S_{set}.

We assume that the strings are binary without loss of generality, since strings from larger alphabets can be binarized as described in Section 2.2. Likewise, we can assume that S_{set} is prefix-free; as noted in Chapter 2, any set of strings can be made prefix-free by appending a terminator symbol to each string.

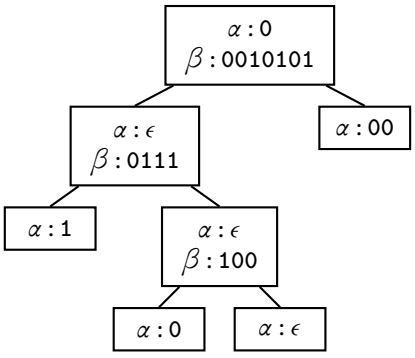


FIGURE 6.1: The Wavelet Trie of the sequence of strings ⟨0001, 0011, 0100, 00100, 0100, 00100, 0100⟩.

A formal definition of the Wavelet Trie can be given along the lines of the compacted trie definition of Section 2.7.

DEFINITION 6.2.1 Let S be a non-empty sequence of binary strings, $S = \langle S_0, \ldots, S_{n-1} \rangle$, $S_i \in \{0,1\}^*$, whose underlying string set S_{set} is prefix-free. The *Wavelet Trie* of S, denoted as WT(S), is built recursively as follows.

— If the sequence is constant, i.e. $S_i = \alpha$ for all i, the Wavelet Trie is a node labeled with α.

— Otherwise, let α be the longest common prefix of S. For any $0 \leq i < |S|$ we can write $S_i = \alpha b_i \gamma_i$, where b_i is a single bit. For $b \in \{0,1\}$ we can then define two sequences $S^b = \langle \gamma_i | b_i = b \rangle$, and the bitvector $\beta = \langle b_i \rangle$; in other words, S is partitioned in the two subsequences depending on whether the string begins with $\alpha 0$ or $\alpha 1$, the remaining suffixes form the two sequences S^0 and S^1, and the bitvector β discriminates whether the suffix γ_i is in S^0 or S^1. Then the Wavelet Trie of S is the tree whose root is labeled with α and β, and whose children (respectively labeled with 0 and 1) are the Wavelet Tries of the sequences S^0 and S^1.

An example is shown in Fig. 6.1. Leaves are labeled only with the common prefix α while internal nodes are labeled both with α and the bitvector β.

As it can be easily seen, the Wavelet Trie is a generalization of the wavelet tree on S: each node splits the underlying string set S_{set} in two subsets and a bitvector is used to tell which elements of the sequence belong to which subset. In fact, any wavelet tree can be seen as a Wavelet Trie through a specific mapping of the alphabet to binary strings. For example the original balanced wavelet tree can be obtained by mapping each element of the alphabet to a distinct string of $\lceil \log \sigma \rceil$ bits; another popular variant is the Huffman-tree shaped wavelet tree, which can be obtained as a Wavelet Trie by mapping each symbol to its Huffman code.

The algorithms described in [59] can be applied to the Wavelet Trie without any modification, so we immediately obtain the following.

LEMMA 6.2.2 The Wavelet Trie supports Access, Rank, and Select operations. In particular, if h_s is the number of internal nodes in the root-to-node path representing s in WT(S), the following holds.

— Access(i) performs $O(h_s)$ Rank operations on the bitvectors, where s is the resulting string.

— Rank(s, i) performs $O(h_s)$ Rank operations on the bitvectors.

— Select(s, i) performs $O(h_s)$ Select operations on the bitvectors.

Since $h_s \leq |s|$, the number of Rank/Select operations performed on the bitvectors is bounded by the length of the string.

PREFIX OPERATIONS. Definition 6.2.1 implies that for any prefix p occurring in at least one element of the sequence, the subsequence of strings prefixed by p is represented by a subtree of WT(S).

This simple property allows us to support two new operations, RankPrefix and SelectPrefix, as defined in the introduction. The implementation is identical to Rank and Select, with the following modifications: if n_p is the node obtained by prefix-searching p in the trie, for RankPrefix the top-down traversal stops at n_p; for SelectPrefix the bottom-up traversal starts at n_p. This proves the following lemma.

LEMMA 6.2.3 Let p be a prefix occurring in the sequence S. Then RankPrefix(p, i) performs $O(h_p)$ Rank operations on the bitvectors, and SelectPrefix(p, i) performs $O(h_p)$ Select operations on the bitvectors.

Note that, since S_{set} is prefix-free, Rank and Select on any string in S_{set} are equivalent to RankPrefix and SelectPrefix, hence it is sufficient to implement these two operations.

AVERAGE HEIGHT. To analyze the space occupied by the Wavelet Trie, we define the *average height*.

DEFINITION 6.2.4 The average height $\tilde{h}$ of a WT(S) is defined as $\tilde{h} = \frac{1}{n}\sum_{i=0}^{n-1} h_{S_i}$.

Note that the average is taken on the *sequence*, not on the set of distinct values. Hence we have $\tilde{h}n \leq \sum_{i=0}^{n-1} |S_i|$ (i.e. the total input size), but we expect $\tilde{h}n \ll \sum_{i=0}^{n-1} |S_i|$ in real situations, for example if short strings are more frequent than long strings, or they have long prefixes in common (exploiting the path compression of the Patricia trie). The quantity $\tilde{h}n$ is equal to the sum of the lengths of all the bitvectors β, since each string S_i contributes exactly one bit to all the internal nodes in its root-to-leaf path. Also, the root-to-leaf paths form a prefix-free encoding for S_{set}, and their concatenation for each element of S is an order-zero encoding for S, thus it cannot be smaller than the zero-order entropy of S, as summarized in the following lemma.

LEMMA 6.2.5 Let $\tilde{h}$ be the average height of WT(S). Then $H_0(S) \leq \tilde{h} \leq \frac{1}{n}\sum_{i=0}^{n-1} |S_i|$.

STATIC SUCCINCT REPRESENTATION. Our first representation of the Wavelet Trie is static. We show how by using suitable succinct data structures the space can be made very close to the information theoretic lower bound.

To store the Wavelet Trie we need to store its two components: the underlying Patricia trie and the bitvectors in the internal nodes.

Since the internal nodes in the tree underlying the Patricia trie have exactly two children, we can represent the tree topology using the DFBS representation of Section 2.6.3, which takes $2|S_{set}| + o(|S_{set}|)$ bits, while supporting traversal operations in constant time. If we denote the number of trie edges as $e = 2(|S_{set}| - 1)$, the space can be written as

$e + o(|S_{set}|)$. The e labels α of the nodes are concatenated in depth-first order in a single bitvector L. We use the partial sum data structure of [99] to delimit the labels in L. This adds $\mathscr{B}(e, |L| + e) + o(|S_{set}|)$ bits. The total space (in bits) occupied by the trie structure is hence $|L| + e + \mathscr{B}(e, |L| + e) + o(|S_{set}|)$.

We now recast the lower bound in [39] using our notation, specializing it for the case of binary strings.

THEOREM 6.2.6 ([39]) For a prefix-free string set S_{set}, the information-theoretic lower bound $LT(S_{set})$ for encoding S_{set} is given by $LT(S_{set}) = |L| + e + \mathscr{B}(e, |L| + e)$, where L is the bitvector containing the e labels α of the nodes concatenated in depth-first order.

It follows immediately that the trie space is just the lower bound LT plus a negligible overhead.

It remains to encode the bitvectors β. We use the RRR encoding, which takes $|\beta| H_0(\beta) + o(|\beta|)$ to compress the bitvector β and supports constant-time Rank/Select operations. In [59] it is shown that, regardless of the shape of the tree, the sum of the entropies of the bitvectors β's add up to the total entropy of the sequence, $nH_0(S)$, plus negligible terms.

With respect to the redundancy beyond $nH_0(S)$, however, we cannot assume that $|S_{set}| = o(n)$ and that the tree is balanced, as in [59] and most wavelet tree literature; in our applications, it is well possible that $|S_{set}| = \Theta(n)$, so a more careful analysis is needed. For the sake of clarity, we defer the technical lemmas to Section 6.2.1; Lemma 6.2.11 shows that in the general case the redundancy add up to $o(\tilde{h}n)$ bits.

We concatenate the RRR encodings of the bitvectors, and use again the partial sum structure of [99] to delimit the encodings, with an additional space occupancy of $o(\tilde{h}n)$. The bound is proven in Lemma 6.2.12. Overall, the set of bitvectors occupies $nH_0(S) + o(\tilde{h}n)$ bits.

All the operations can be supported with a trie traversal, which takes $O(|s|)$ time, and $O(h_s)$ Rank/Select operations on the bitvectors. Since the bitvector operations are constant-time, all the operations take $O(|s| + h_s)$ time. Putting together these observations, we obtain the following theorem.

THEOREM 6.2.7 The Wavelet Trie WT(S) of a sequence of binary strings S can be encoded in $LT(S_{set}) + nH_0(S) + o(\tilde{h}n)$ bits, while supporting the operations Access, Rank, Select, RankPrefix, and SelectPrefix on a string s in $O(|s| + h_s)$ time.

Note that when the tree is balanced both time and space bounds are basically equivalent to those of the standard wavelet tree. We remark that the space upper bound in Theorem 6.2.7 is just the information theoretic lower bound $LB(S) = LT(S_{set}) + nH_0(S)$ plus an overhead negligible in the input size.

6.2.1 *Multiple static bitvectors*

We prove here the lemmas required for the proof of Theorem 6.2.7.

LEMMA 6.2.8 Let S be a sequence of length n on an alphabet of cardinality σ, with each symbol of the alphabet occurring at least once. Then the following holds:

$$nH_0(S) \geq (\sigma - 1)\log n.$$

Proof The inequality is trivial when $\sigma = 1$. When there are at least two symbols, the minimum entropy is attained when $\sigma - 1$ symbols occur once and one symbol occurs the remaining $n - \sigma + 1$ times. To show this, suppose by contradiction that the minimum entropy is attained by a string where two symbols occur more than once, occurring respectively a and b times. Their contribution to the entropy term is $a \log \frac{n}{a} + b \log \frac{n}{b}$. This contribution can be written as $f(a)$ where

$$f(t) = t \log \frac{n}{t} + (b + a - t) \log \frac{n}{b + a - t},$$

but $f(t)$ has two strict minima in 1 and $b + a - 1$ among the positive integers, so the entropy term can be lowered by making one of the symbol absorb all but one the occurrences of the other, yielding a contradiction.

To prove the lemma, it is sufficient to see that the contribution to the entropy term of the $\sigma - 1$ singleton symbols is $(\sigma - 1)\log n$. ∎

LEMMA 6.2.9 $O(|S_{set}|)$ is bounded by $o(\tilde{h}n)$.

Proof It suffices to prove that

$$\frac{|S_{set}|}{\tilde{h}n}$$

is asymptotic to 0 as n grows. By Lemma 6.2.5 and Lemma 6.2.8, and assuming $|S_{set}| \geq 2$,

$$\frac{|S_{set}|}{\tilde{h}n} \leq \frac{|S_{set}|}{nH_0(S)} \leq \frac{|S_{set}|}{(|S_{set}| - 1)\log n} \leq \frac{2}{\log n},$$

which completes the proof. ∎

LEMMA 6.2.10 The sum of the redundancy of σ RRR bitvectors of $m_1, \ldots, m_\sigma$ bits respectively, where $\sum_i m_i = m$, can be bounded by

$$O\left(m \frac{\log\log \frac{m}{\sigma}}{\log \frac{m}{\sigma}} + \sigma\right).$$

Proof The redundancy of a single bitvector can be bounded by $c_1 \frac{m_i \log\log m_i}{\log m_i} + c_2$. Since $f(x) = \frac{x \log\log x}{\log x}$ is concave, we can apply Jensen's inequality:

$$\frac{1}{\sigma} \sum_i \left(c_1 \frac{m_i \log\log m_i}{\log m_i} + c_2 \right) \leq c_1 \frac{\frac{m}{\sigma} \log\log \frac{m}{\sigma}}{\log \frac{m}{\sigma}} + c_2.$$

The result follows by multiplying both sides by σ. ∎

LEMMA 6.2.11 The redundancy of the RRR bitvectors in WT(S) can be bounded by $o(\tilde{h}n)$.

Proof Since the bitvector lengths add up to $\tilde{h}n$, we can apply Lemma 6.2.10 and obtain that the redundancy are bounded by

$$\mathrm{O}\left(\tilde{h}n \frac{\log\log \frac{\tilde{h}n}{|\mathrm{S}_{\mathrm{set}}|}}{\log \frac{\tilde{h}n}{|\mathrm{S}_{\mathrm{set}}|}} + |\mathrm{S}_{\mathrm{set}}| \right).$$

The term in $|\mathrm{S}_{\mathrm{set}}|$ is already taken care of by Lemma 6.2.9. It suffices then to prove that

$$\frac{\log\log \frac{\tilde{h}n}{|\mathrm{S}_{\mathrm{set}}|}}{\log \frac{\tilde{h}n}{|\mathrm{S}_{\mathrm{set}}|}}$$

is negligible as n grows, and because $f(x) = \frac{\log\log x}{\log x}$ is asymptotic to 0, we just need to prove that $\frac{\tilde{h}n}{|\mathrm{S}_{\mathrm{set}}|}$ grows to infinity as n does. Using again Lemma 6.2.5 and Lemma 6.2.8 we obtain that

$$\frac{\tilde{h}n}{|\mathrm{S}_{\mathrm{set}}|} \geq \frac{n\mathrm{H}_0(\mathrm{S})}{|\mathrm{S}_{\mathrm{set}}|} \geq \frac{(|\mathrm{S}_{\mathrm{set}}| - 1)\log n}{|\mathrm{S}_{\mathrm{set}}|} \geq \frac{\log n}{2}$$

thus proving the lemma. ∎

LEMMA 6.2.12 The partial sum data structure used to delimit the RRR bitvectors in WT(S) occupies $o(\tilde{h}n)$ bits.

Proof By Lemma 6.2.11 the sum of the RRR encodings is $n\mathrm{H}_0(\mathrm{S}) + o(\tilde{h}n)$. To encode the $|\mathrm{S}_{\mathrm{set}}|$ delimiters, the partial sum structure of [99] takes

$$\begin{aligned} &|\mathrm{S}_{\mathrm{set}}| \log \left(\frac{n\mathrm{H}_0(\mathrm{S}) + o(\tilde{h}n) + |\mathrm{S}_{\mathrm{set}}|}{|\mathrm{S}_{\mathrm{set}}|} \right) + \mathrm{O}(|\mathrm{S}_{\mathrm{set}}|) \\ &\leq |\mathrm{S}_{\mathrm{set}}| \log \left(\frac{n\mathrm{H}_0(\mathrm{S})}{|\mathrm{S}_{\mathrm{set}}|} \right) + |\mathrm{S}_{\mathrm{set}}| \log \left(\frac{o(\tilde{h}n)}{|\mathrm{S}_{\mathrm{set}}|} \right) + \mathrm{O}(|\mathrm{S}_{\mathrm{set}}|). \end{aligned}$$

The third term is negligible by Lemma 6.2.9. The second just by dividing by $\tilde{h}n$ and noting that $f(x) = \frac{\log x}{x}$ is asymptotic to 0. It remains to show that the first term is $o(\tilde{h}n)$. Dividing by $\tilde{h}n$ and using again Lemma 6.2.5 we obtain

$$\frac{|\mathrm{S}_{\mathrm{set}}| \log\left(\frac{nH_0(\mathrm{S})}{|\mathrm{S}_{\mathrm{set}}|}\right)}{\tilde{h}n} \leq \frac{|\mathrm{S}_{\mathrm{set}}| \log\left(\frac{nH_0(\mathrm{S})}{|\mathrm{S}_{\mathrm{set}}|}\right)}{nH_0(\mathrm{S})} = \frac{\log\left(\frac{nH_0(\mathrm{S})}{|\mathrm{S}_{\mathrm{set}}|}\right)}{\frac{nH_0(\mathrm{S})}{|\mathrm{S}_{\mathrm{set}}|}}$$

By using again that $f(x) = \frac{\log x}{x}$ is asymptotic to 0 and proving as in Lemma 6.2.11 that $\frac{nH_0(\mathrm{S})}{|\mathrm{S}_{\mathrm{set}}|}$ grows to infinity as n does, the result follows. ∎

6.3 Dynamic Wavelet Tries

In this section we show how to implement dynamic updates to the Wavelet Trie.

Standard dynamic wavelet trees [79, 55, 85] replace the bitvectors in the nodes with *dynamic bitvectors with indels*, that support the insertion of deletion of bits at arbitrary positions. Insertion in the wavelet tree at position i can be performed by inserting `0` or `1` at position i of the root, whether the leaf corresponding to the value to be inserted is on the left or right subtree. A Rank operation is used to find the new position i' in the corresponding child. The algorithm proceeds recursively until a leaf is reached. Deletion is symmetric.

The same operations can be implemented on a Wavelet Trie. In addition, the Wavelet Trie supports insertion of strings that do not already occur in the sequence, and deletion of the last occurrence of a string, in both cases changing the alphabet $\mathrm{S}_{\mathrm{set}}$ and thus the shape of the tree. To do so we represent the underlying tree structure of the Wavelet Trie with a dynamic Patricia trie, as described in Section 2.7.2. We summarize the properties of a dynamic Patricia trie in the following lemma.

Lemma 6.3.1 A dynamic Patricia trie on k binary strings occupies $O(kw) + |\mathrm{L}|$ bits, where L is defined as in Theorem 6.2.6. Besides the standard traversal operations, the followint operations are supported.

- Insertion of a new string s in $O(|s|)$ time.
- Deletion of a string s in $O(\hat{\ell})$ time, where $\hat{\ell}$ is the length of the longest string in the trie (including s).

Updating the bitvectors. Each internal node of the trie is augmented with a bitvector β, as in the static Wavelet Trie. Inserting and deleting a string induce the following changes on the bitvectors βs.

- Insert(s, i): If the string is not present, we insert it into the Patricia trie, causing the split of an existing node: a new internal node and a new leaf are added. We

initialize the bitvector in the new internal node as a constant sequence of bits b if the split node is a b-labeled child of the new node; the length of the new bitvector is equal to the length of the sequence represented by the split node (i.e. the number of b bits in the parent node if the split node is a b-labeled child). The algorithm then follows as if the string was in the trie. This operation is shown in Figure 6.2.

Now we can assume the string is in the trie. Let prefix α and bitvector β be the labels in the root. Since the string is in the trie, it must be in the form $\alpha b \gamma$, where b is a bit. We insert b at position i in β and compute $i' = \text{Rank}(b, i)$ in β, and insert recursively γ in the b-labeled subtree of the root at position i'. We proceed until we reach a leaf.

— Delete(i): Let β be the bitvector in the root. We first find the bit corresponding to position i in the bitvector, $b = \text{Access}(i)$ in β. Then we compute $i' = \text{Rank}(b, i)$ in β, and delete recursively the string at position i' from the b-labeled subtree. We then delete the bit at position i from β.

We then check if the parent of the leaf node representing the string has a constant bitvector; in this case the string deleted was the last occurrence in the sequence. We can then delete the string from the Patricia trie, thus deleting an internal node (whose bitvector is now constant) and a leaf.

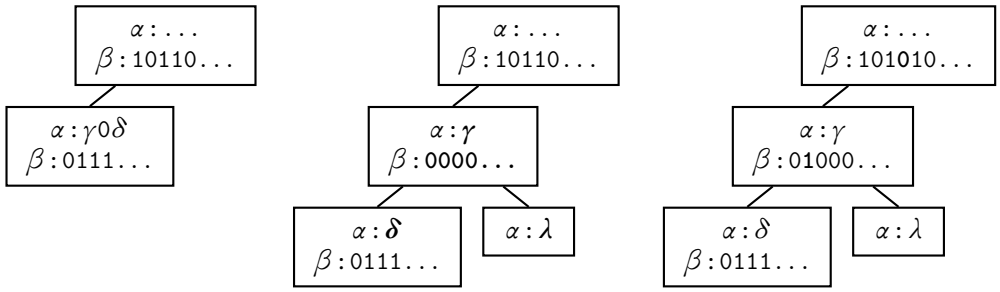


FIGURE 6.2: Insertion of the new string $s = \ldots\gamma 1\lambda$ at position 3. An existing node is split by adding a new internal node with a constant bitvector and a new leaf. The corresponding bits are then inserted in the root-to-leaf path nodes.

In both cases the number of operations (Rank, Insert, Delete) on the bitvectors is bounded by $O(h_s)$. The operations we need to perform on the bitvectors are the standard insertion/deletion, with one important exception: when a node is split, we need to create a new constant bitvector of arbitrary length. We call this operation Init(b, n), which fills an empty bitvector with n copies of the bit b. The following observation rules out for our purposes most existing dynamic bitvector constructions.

OBSERVATION 6.3.2 If the encoding of a constant (i.e. 0^n or 1^n) bitvector uses $\omega(f(n))$ memory words (of size w), Init(b, n) cannot be supported in $O(f(n))$ time.

Uncompressed bitvectors use $\Omega(n/w)$ words; the compressed bitvectors of [85, 55], although they have a desirable occupancy of $|\beta|\mathrm{H}_0(\beta) + o(|\beta|)$, have $\Omega(n \log\log n/(w \log n))$ words of redundancy. Since we aim for polylog operations, these constructions cannot be considered *as is*.

We first consider the case of append-only sequences. We remark that, in the Insert operation described above, when appending a string at the end of the sequence the bits inserted in the bitvectors are appended at the end, so it is sufficient that the bitvectors support an Append operation in place of a general Insert. Furthermore, Init can be implemented simply by adding a left offset in each bitvector, which increments each bitvector space by $\mathrm{O}(\log n)$, and queries can be offsetted back in constant time. Using the append-only bitvectors described in Section 6.3.1, and observing that the redundancy can be analyzed as in Section 6.2, we can state the following theorem.

THEOREM 6.3.3 The append-only Wavelet Trie on a dynamic sequence S supports the operations Access, Rank, Select, RankPrefix, SelectPrefix, and Append in $\mathrm{O}(|s| + h_s)$ time. The total space occupancy is $\mathrm{O}(|\mathrm{S}_{\mathrm{set}}|w) + |\mathrm{L}| + n\mathrm{H}_0(\mathrm{S}) + o(\tilde{h}n)$ bits, where L is defined as in Theorem 6.2.6.

Using instead the fully-dynamic bitvectors of Section 6.3.2, we can state the following theorem.

THEOREM 6.3.4 The dynamic Wavelet Trie on a dynamic sequence S supports the operations Access, Rank, Select, RankPrefix, SelectPrefix, and Insert in $\mathrm{O}(|s| + h_s \log n)$ time. Delete is supported in $\mathrm{O}(|s| + h_s \log n)$ time if s occurs more than once, otherwise time is $\mathrm{O}(\hat{\ell} + h_s \log n)$, where $\hat{\ell}$ is the length of the longest string. The total space occupancy is $\mathrm{O}(n\mathrm{H}_0(\mathrm{S}) + |\mathrm{S}_{\mathrm{set}}|w) + \mathrm{L}$ bits, where L is defined as in Theorem 6.2.6.

Note that, using the compact notation defined in the introduction, the space bound in Theorem 6.3.3 can be written as $\mathrm{LB}(\mathrm{S}) + \mathrm{PT}(\mathrm{S}_{\mathrm{set}}) + o(\tilde{h}n)$, while the one in Theorem 6.3.4 can be written as $\mathrm{LB}(\mathrm{S}) + \mathrm{PT}(\mathrm{S}_{\mathrm{set}}) + \mathrm{O}(n\mathrm{H}_0)$.

6.3.1 *Append-only bitvectors*

In this section we describe an append-only bitvector with constant-time Rank/Select/Append operations and nearly optimal space occupancy. The data structure uses RRR as a black-box data structure, assuming only its query time and space guarantees. We require the following *decomposable* property on RRR: given an input bitvector of n bits packed into $\mathrm{O}(n/w)$ words of size $w \geq \log n$, RRR can be built in $\mathrm{O}(n'/\log n)$ time for any chunk of $n' \geq \log n$ consecutive bits of the input bitvector, using table lookups and the Four-Russians trick; moreover, this $\mathrm{O}(n'/\log n)$-time work can be spread over $\mathrm{O}(n'/\log n)$ steps, each of $\mathrm{O}(1)$ time, that can be interleaved with other operations not involving the chunk at hand. This a quite mild requirement and, for this reason, it is a

general technique that can be applied to other static compressed bitvectors other than RRR with the same guarantees. Hence we believe that the following approach is of independent interest.

THEOREM 6.3.5 There exists an append-only bitvector that supports Access, Rank, Select, and Append on a bitvector β in O(1) time. The total space is $nH_0(\beta) + o(n)$ bits, where $n = |\beta|$.

Before describing the data structure and proving Theorem 6.3.5 we need to introduce some auxiliary lemmas.

LEMMA 6.3.6 (SMALL BITVECTORS) Let β' be a bitvector of size $n' = O(\text{polylog}(n))$. Then there exists a data structure that supports Access, Rank, Select, and Append on β' in O(1) time, while occupying O(polylog(n)) bits.

Proof It is sufficient to store explicitly all the answers to the queries Rank and Select in arrays of n' elements, thus taking $O(n' \log n') = O(\text{polylog}(n))$. Append can be supported in constant time by keeping a running count of the `1`s in the bitvector and the position of the last `0` and `1`, which are sufficient to compute the answers to the Rank and Select queries for the appended bit. ∎

LEMMA 6.3.7 (AMORTIZED CONSTANT-TIME) There exists a data structure that supports Access, Rank, and Select in O(1) time and Append in *amortized* O(1) time on a bitvector β of n bits. The total space occupancy is $nH_0(\beta) + o(n)$ bits.

Proof We split the input bitvector β into t smaller bitvectors $V_t, V_{t-1}, \ldots, V_1$, such that β is equal to the concatenation $V_t \cdot V_{t-1} \cdots V_1$ at any time. Let $n_i = |V_i| \geq 0$ be the length of V_i, and m_i be the number of `1`s in it, so that $\sum_{i=1}^{t} m_i = m$ and $\sum_{i=1}^{t} n_i = n$. Following Overmars's logarithmic method [98], we maintain a collection of static data structures on $V_t, V_{t-1}, \ldots, V_1$ that are periodically rebuilt.

(*a*) A data structure F_1 as described in Lemma 6.3.6 to store $\beta' = V_1$. Space is O(polylog(n)) bits.

(*b*) A collection of static data structures $F_t, F_{t-1}, \ldots, F_2$, where each F_i stores V_i using RRR. Space occupancy is $nH_0(\beta) + o(n)$ bits.

(*c*) Fusion Trees [47] of constant height storing the partial sums on the number of `1`s, $S_i^1 = \sum_{j=t}^{i+1} m_j$, where $S_t^1 = 0$, and symmetrically the partial sums on the number of `0`s, $S_i^0 = \sum_{j=t}^{i+1} (n_j - m_j)$, setting $S_t^1 = 0$. Predecessor takes O(1) time and construction is $O(t)$ time. Space occupancy is $O(t \log n) = o(n)$ bits.

We fix $r = c \log n_0$ for a suitable constant $c > 1$, where n_0 is the length $n > 2$ of the initial input bitvector β. We keep this choice of r until F_t is reconstructed: at that point, we set n_0 to the current length of β and we update r consistently. Based on this choice of r, we guarantee that $r = \Theta(\log n)$ at any time and introduce the following constraints: $n_1 \leq r$ and, for every $i > 1$, n_i is either 0 or $2^{i-2}r$. It follows immediately that $t = \Theta(\log n)$, and hence the Fusion Trees in (*c*) contain $\mathrm{O}(\log n)$ entries, thus guaranteeing constant height.

We now discuss the query operations. Rank(b, i) and Select(b, i) are performed as follows for a bit $b \in \{0, 1\}$. Using the data structure in (*c*), we identify the corresponding bitvector V_i along with the number S_i^b of occurrences of bit b in the preceding ones, $\mathrm{V}_t, \ldots, \mathrm{V}_{i+1}$. The returned value corresponds to the index i of F_i, which we query and combine the result with S_i^b: we output the sum of S_i^b with the result of Rank$(b, i - \sum_{j=t}^{i+1} n_i)$ query on F_i in the former case; we output Select$(b, i - \mathrm{S}_i^b)$ query on F_i in the latter. Hence, the cost is $\mathrm{O}(1)$ time.

It remains to show how to perform Append(b) operation. While $n_1 < r$ we just append the bit b to F_1, which takes constant time by Lemma 6.3.6. When n_1 reaches r, let j be the smallest index such that $n_j = 0$. Then $\sum_{i=1}^{j-1} n_i = 2^{j-2}r$, so we concatenate $\mathrm{V}_{j-1} \cdots \mathrm{V}_1$ and rename this concatenation V_j (no collision since it was $n_j = 0$). We then rebuild F_j on V_j and set F_i for $i < j$ to empty (updating $n_j, \ldots, n_1$). We also rebuild the Fusion Trees of (*c*), which takes an additional $\mathrm{O}(\log n)$ time. When F_t is rebuilt, we have that the new V_t corresponds to the whole current bitvector β, since $\mathrm{V}_{t-1}, \ldots, \mathrm{V}_1$ are empty. We thus set $n_0 := |\beta|$ and update r consequently. By observing that each F_j is rebuilt every $\mathrm{O}(n_j)$ Append operations and that RRR construction time is $\mathrm{O}(n_j / \log n)$, it follows that each Append is charged $\mathrm{O}(1/\log n)$ time on each F_j, thus totaling $\mathrm{O}(t/\log n) = \mathrm{O}(1)$ time. ∎

We now show how to de-amortize the bitvector of Lemma 6.3.7. In the de-amortization we have to keep copies of some bitvectors, so the $n\mathrm{H}_0$ term becomes $\mathrm{O}(n\mathrm{H}_0)$.

LEMMA 6.3.8 There exists a data structure that supports Access, Rank, Select, and Append in $\mathrm{O}(1)$ time on a bitvector β of n bits. The total space occupancy is $\mathrm{O}(n\mathrm{H}_0(\beta)) + o(n)$.

Proof To de-amortize the structure we follow Overmars's classical method of partial rebuilding [98]. The idea is to spread the construction of the RRR's F_j over the next $\mathrm{O}(n_j)$ Append operations, charging extra $\mathrm{O}(1)$ time each. We already saw in Lemma 6.3.7 that this suffices to cover all the costs. Moreover, we need to increase the speed of construction of F_j by a suitable constant factor with respect to the speed of arrival of the Append operations, so we are guaranteed that the construction of F_j is completed before the next construction of F_j is required by the argument shown in the proof of Lemma 6.3.7. We refer the reader to [98] for a thorough discussion of the technical details of this general technique.

While V_1 reaches its bound of r bits, we have a budget of $\Theta(r) = \Theta(\log n)$ operations that we can use to prepare the next version of the data structure. We use this budget to perform the following operations.

(*i*) Identify the smallest j such that $n_j = 0$ and start the construction of F_j by creating a *proxy bitvector* $\tilde{F}_j$ which references the existing $F_{j-1}, \ldots, F_1$ and Fusion Trees in (*c*), so that it can answer queries in $O(1)$ time as if it was the fully built F_j. When we switch to this version of the data structure, these $F_{j-1}, \ldots, F_1$ become accessible only inside $\tilde{F}_j$.

(*ii*) Build the Fusion Trees in (*c*) for the next reconstruction of the data structure. Note that this would require to know the final values of the n_is and m_is when V_1 is full and the reconstruction starts. Instead, we use the current values of n_i and m_i: only the values for the last non-empty segment will be wrong. We can *correct* the Fusion Trees by adding an additional *correction* value to the last non-empty segment; applying the correction at query time has constant-time overhead.

(*iii*) Build a new version of the data structure which references the new Fusion Trees, the existing bitvectors $F_t, \ldots, F_{j+1}$, the proxy bitvector $\tilde{F}_j$ and new empty bitvectors $F_{j-1}, \ldots, F_1$ (hence, $n_j = 2^{j-2}r$ and $n_{j-1} = \cdots = n_1 = 0$).

When n_1 reaches r, we can replace in constant time the data structure with the one that we just finished rebuilding.

At each Append operation, we use an additional $O(1)$ budget to advance the construction of the F_js from the proxies $\tilde{F}_j$s in a round-robin fashion. When the construction of one F_j is done, the proxy $\tilde{F}_j$ is discarded and replaced by F_j. Since, by the amortization argument in the proof of Lemma 6.3.7, each F_j is completely rebuilt by the time it has to be set to empty (and thus used for the next reconstruction), at most one copy of each bitvector has to be kept, thus the total space occupancy grows from $nH_0(\beta) + o(n)$ to $O(nH_0(\beta)) + o(n)$. Moreover, when r has to increase (and thus the n_i's should be updated), we proceed as in [98]. ∎

We can now use the de-amortized data structure to bootstrap a constant-time append-only bitvector with space occupancy $nH_0(\beta) + o(n)$, thus proving Theorem 6.3.5. The idea is to split the bitvector β into uniform blocks which are independently encoded with RRR, while the bitvector of Lemma 6.3.8 is used as a searchable partial sums data structure[1] to store the number of 0s and 1s in each block.

Proof (of Theorem 6.3.5) Let β be the input bitvector, and $L = \Theta(\log^{1+\varepsilon} n)$ be a power of two, for any given positive constant $\varepsilon > 0$.

[1] Other dynamic data structures for searchable partial sums exist in the literature, but to the best of our knowledge none of them is suitable for our purposes. For example, the data structure in [36] relies on the assumptions that all the values are *positive*, while in our case there can be blocks with no 0s or no 1s.

We split β into $n_{\mathrm{L}} = \lfloor n/\mathrm{L} \rfloor$ smaller bitvectors B_i's, each of length L and with $\hat{m}_i$ `1`s ($\mathtt{0} \leq \hat{m}_i \leq \mathrm{L}$), plus a residual bitvector B′ of length $\mathtt{0} \leq |\mathrm{B}'| < \mathrm{L}$: at any time $\beta = \mathrm{B}_1 \cdot \mathrm{B}_2 \cdots \mathrm{B}_{n_{\mathrm{L}}} \cdot \mathrm{B}'$. Using this partition, we maintain the following data structures:

(*i*) A collection $\hat{\mathrm{F}}_1, \hat{\mathrm{F}}_2, \ldots, \hat{\mathrm{F}}_{n_{\mathrm{L}}}$ of static data structures, where each $\hat{\mathrm{F}}_i$ stores B_i using RRR.

(*ii*) The data structure in Lemma 6.3.6 to store B′.

(*iii*) The data structure in Lemma 6.3.8 to store the partial sums $\hat{\mathrm{S}}_i^{\mathtt{1}} = \sum_{j=1}^{i-1} \hat{m}_j$, setting $\hat{\mathrm{S}}_1^{\mathtt{1}} = \mathtt{0}$. This is implemented by maintaining a bitvector that has a `1` for each position $\hat{\mathrm{S}}_i^{\mathtt{1}}$, and `0` elsewhere. Predecessor queries can be implemented by composing Rank and Select. The bitvector has length $n_{\mathrm{L}} + m$ and contains n_{L} `1`s. The partial sums $\hat{\mathrm{S}}_i^{\mathtt{0}} = \sum_{j=1}^{i-1} (\mathrm{L} - \hat{m}_j)$ are kept symmetrically in another bitvector.

Rank(b, i) and Select(b, i) are implemented as follows for a bit $b \in \{\mathtt{0}, \mathtt{1}\}$. Using the data structure in (*iii*), we identify the corresponding bitvector B_i in (*i*) or B′ in (*ii*) along with the number $\hat{\mathrm{S}}_i^b$ of occurrences of bit b in the preceding segments. In both cases, we query the corresponding dictionary and combine the result with $\hat{\mathrm{S}}_i^b$. These operations take O(1) time.

Now we focus on Append(b). At every Append operation, we append a `0` to the one of the bitvectors in (*iii*) depending on whether b is `0` or `1`, thus maintaining the partial sums invariant. This takes constant time. We guarantee that $|\mathrm{B}'| \leq \mathrm{L}$ bits: whenever $|\mathrm{B}'| = \mathrm{L}$, we conceptually create $\mathrm{B}_{n_{\mathrm{L}}+1} := \mathrm{B}'$, still keeping its data structure in (*ii*); reset B′ to be empty, creating the corresponding data structure in (*ii*); append a `1` to the bitvectors in (*iii*). We start building a new static compressed data structure $\hat{\mathrm{F}}_{n_{\mathrm{L}}+1}$ for $\mathrm{B}_{n_{\mathrm{L}}+1}$ using RRR in $\mathrm{O}(\mathrm{L}/\log n)$ steps of O(1) time each. During the construction of $\hat{\mathrm{F}}_{n_{\mathrm{L}}+1}$ the old B′ is still valid, so it can be used to answer the queries. As soon as the construction is completed, in $\mathrm{O}(\mathrm{L}/\log n)$ time, the old B′ can be discarded and queries can be now handled by $\hat{\mathrm{F}}_{n_{\mathrm{L}}+1}$. Meanwhile the new appended bits are handled in the new B′, in O(1) time each, using its new instance of (*ii*). By suitably tuning the speed of the operations, we can guarantee that by the time the new reset B′ has reached L/2 (appended) bits, the above O(L) steps have been completed for $\hat{\mathrm{F}}_{n_{\mathrm{L}}+1}$. Hence, the total cost of Append is just O(1) time in the worst case.

To complete our proof, we have to discuss what happens when we have to double $\mathrm{L} := 2 \times \mathrm{L}$. This is a standard task known as global rebuilding [98]. We rebuild RRR for the concatenation of B_1 and B_2, and deallocate the latter two after the construction; we then continue with RRR on the concatenation of B_3 and B_4, and deallocate them after the construction, and so on. Meanwhile, we build a copy (*iii*)' of the data structure in (*iii*) for the new parameter $2 \times \mathrm{L}$, following an incremental approach. At any time, we only have (*iii*)' and $\hat{\mathrm{F}}_{2i-1}, \hat{\mathrm{F}}_{2i}$ duplicated. The implementation of Rank and Select needs a

minor modification to deal with the already rebuilt segments. The global rebuilding is completed before we need again to double the value of L.

We now perform the space analysis. As for (*i*), we have to add up the space taken by $\hat{\mathrm{F}}_1, \ldots, \hat{\mathrm{F}}_{n_\mathrm{L}}$ plus that taken by the one being rebuilt using $\hat{\mathrm{F}}_{2i-1}, \hat{\mathrm{F}}_{2i}$. This sum can be upper bounded by $\sum_{i=1}^{n_\mathrm{L}} (\mathscr{B}(m_i, \mathrm{L}) + o(\mathrm{L})) + \mathrm{O}(\mathrm{L}) = \mathrm{H}_0(\beta) + o(n)$. The space for (*ii*) is $\mathrm{O}(\mathrm{polylog}(n)) = o(n)$. Finally, the occupancy of the S_i^1 partial sums in (*iii*) is $\mathscr{B}(n_\mathrm{L}, n_\mathrm{L} + m) + o(n_\mathrm{L} + m) = \mathrm{O}(n_\mathrm{L} \log(1 + m/n_\mathrm{L})) = \mathrm{O}(n \log n/\mathrm{L}) = o(n)$ bits, since the bitvector has length $n_\mathrm{L} + m$ and contains n_L `1`s. The analysis is symmetric for the S_i^0 partial sums, and for the copies in (*iii*)'. ∎

6.3.2 *Fully dynamic bitvectors*

We introduce a new dynamic bitvector construction which, although the entropy term has a constant greater than 1, supports logarithmic-time Init and Insert/Delete, thus it is suitable for the dynamic Wavelet Trie.

DEFINITION 6.3.9 A *Dynamic Bitvector with Indels and Initialization* is a data structure that maintains a sequence of bits $(b_1, \ldots, b_n) \in \{\mathtt{0}, \mathtt{1}\}^*$ and supports the standard Access/Rank/Select operations plus the following.

— Init(b, n) initializes the sequence to a run of n copies of the bit b.

— Insert(b, i) inserts the bit b immediately before b_i.

— Delete(i) deletes b_i from the sequence.

To support both insertion/deletion and initialization in logarithmic time we adapt the dynamic bitvector presented in Section 3.4 of [85]; in the paper, the bitvector is compressed using Gap Encoding, i.e. the bitvector $\mathtt{0}^{g_0}\mathtt{1}\mathtt{0}^{g_1}\mathtt{1}\ldots$ is encoded as the sequence of *gaps* $g_0, g_1, \ldots$, and the gaps are encoded using Elias delta code [33]. The resulting bit stream is split in chunks of $\Theta(\log n)$ (without breaking the codes) and a self-balancing binary search tree is built on the chunks, with partial counts in all the nodes. Chunks are split and merged upon insertions and deletions to maintain the chunk size invariant, and the tree rebalanced.

Because of gap encoding, the space has a linear dependence on the number of `1`s, hence by Observation 6.3.2 it is not suitable for our purposes. We make a simple modification that enables an efficient Init: in place of gap encoding and delta codes we use RLE and Elias gamma codes [33], as the authors of [45] do in their *practical dictionaries*. RLE encodes the bitvector $\mathtt{0}^{r_0}\mathtt{1}^{r_1}\mathtt{0}^{r_2}\mathtt{1}^{r_3}\ldots$ with the sequence of *runs* $r_0, r_1, r_2, r_3, \ldots$. The runs are encoded with Elias gamma codes. Init(b, n) can be trivially supported by creating a tree with a single leaf node, and encoding a run of n bits b in the node, which can be done in time $\mathrm{O}(\log n)$. In [37] it is proven the space of this encoding is bounded by $\mathrm{O}(n\mathrm{H}_0)$, but even if the coefficient of the entropy term is not 1 as in RRR bitvectors, the

experimental analysis performed in [45] shows that RLE bitvectors perform extremely well in practice. The rest of the data structure is left unchanged; we refer to [85] for the details.

THEOREM 6.3.10 The dynamic RLE+γ bitvector supports the operations Access, Rank, Select, Insert, Delete, and Init on a bitvector β in $O(\log n)$ time. The total space occupancy is $O(nH_0(\beta) + \log n)$ bits.

6.4 OTHER QUERY ALGORITHMS

In this section we describe range query algorithms on the Wavelet Trie that can be useful in particular in database applications and analytics. We note that the algorithms for *distinct values in range* and *range majority element* are similar to the *report* and *range quantile* algorithms presented in [50]; we restate them here for completeness, extending them to prefix operations. In the following we denote with $\mathscr{C}_{\text{op}}$ the cost of Access/Rank/Select on the bitvectors; $\mathscr{C}_{\text{op}}$ is $O(1)$ for static and append-only bitvectors, and $O(\log n)$ for fully dynamic bitvectors.

SEQUENTIAL ACCESS. Suppose we want to enumerate all the strings in the range $[l, r)$, i.e. $S[l, r)$. We could do it with $r - l$ calls to Access, but accessing each string S_i would cost $O(|S_i| + h_{S_i}\mathscr{C}_{\text{op}})$. We show instead how to enumerate the values of a range by enumerating the bits of each bitvector: suppose we have an iterator on root bitvector for the range $[l, r)$. Then if the current bit is 0, the next value is the next value given by the left subtree, while if it is 1 the next value is the next value of the right subtree. We proceed recursively by keeping an iterator on all the bitvectors of the internal nodes we traverse during the enumeration.

When we traverse an internal node for the first time, we perform a Rank to find the initial point, and create an iterator. Next time we traverse it, we just advance the iterator. Note that both RRR bitvectors and RLE bitvectors can support iterators with $O(1)$ advance to the next bit.

By using iterators instead of performing a Rank each time we traverse a node, a single Rank is needed for each traversed node, hence to extract the i-th string it takes $O(|S_i| + \frac{1}{r-l}\sum_{s \in S[l,r)_{\text{set}}} h_s \mathscr{C}_{\text{op}})$ amortized time.

If $S[l, r)_{\text{set}}$, that is the set of distinct strings occurring in $S[l, r)$, is large, then the actual time is smaller due to shared nodes in the paths. In fact, in the extreme case when the whole string set occurs in the range, the bound becomes $O(|S_i| + \frac{1}{r-l}|S_{\text{set}}|\mathscr{C}_{\text{op}})$ amortized time.

DISTINCT VALUES IN RANGE. Another useful query is the enumeration of the distinct values in the range $[l, r)$, which we called $S[l, r)_{\text{set}}$. Note that for each node the distinct values of the subsequence represented by the node are just the distinct values of the left

subtree plus the distinct values of the right subtree in the corresponding ranges. Hence, starting at the root, we compute the number of 0s in the range $[l, r)$ with two calls to Rank. If there are no 0s we just enumerate the distinct elements of the right child in the range $[\text{Rank}(1, l), \text{Rank}(1, r))$. If there are no 1s, we proceed symmetrically. If there are both 0s and 1s, the distinct values are the union of the distinct values of the left child in the range $[\text{Rank}(0, l), \text{Rank}(0, r))$ and those of the right child in the range $[\text{Rank}(1, l), \text{Rank}(1, r))$. Since we only traverse nodes that lead to values that are in the range, the total running time is $O(\sum_{s \in S[l,r)_{\text{set}}} (|s| + h_s \mathscr{C}_{\text{op}}))$, which is the same time as accessing the values, if we knew their positions. As a byproduct, we also get the number of occurrences of each value in the range.

We can stop early in the traversal, hence enumerating the distinct *prefixes* that satisfy some property. For example in an URL access log we can find efficiently the distinct hostnames in a given time range.

RANGE MAJORITY ELEMENT. The previous algorithm can be modified to check if there is a majority element in the range (i.e. one element that occurs more than $\frac{r-l}{2}$ times in $[l, r)$), and, if there is such an element, find it. Start at the root, and count the number of 0s and 1s in the range. If a bit b occurs more than $\frac{r-l}{2}$ times (note that there can be at most one) proceed recursively on the b-labeled subtree, otherwise there is no majority element in the range.

The total running time is $O(h \mathscr{C}_{\text{op}})$, where h is the height of the Wavelet Trie. In case of success, if the string found is s, the running time is just $O(h_s \mathscr{C}_{\text{op}})$. As for the distinct values, this can be applied to prefixes as well by stopping the traversal when the prefix we found until that point satisfies some property.

A similar algorithm can be used as an heuristic to find all the values that occur in the range at least t times, by proceeding as in the enumeration of distinct elements but discarding the branches whose bit has less than t occurrences in the parent. While no theoretical guarantees can be given, this heuristic should perform very well with power-law distributions and high values of t, which are the cases of interest in most data analytics applications.

6.5 PROBABILISTICALLY BALANCED DYNAMIC WAVELET TREES

In this section we show how to use the Wavelet Trie to maintain a dynamic wavelet tree on a sequence from a bounded alphabet with operations that with high probability do not depend on the universe size.

A compelling example is given by numeric data: to represent with a wavelet tree a sequence of integers, say in $\{0, \ldots, 2^{64} - 1\}$, if the tree structure is static it must cover the whole universe, even if the sequence only contains integers from a much smaller subset. Similarly, a text sequence in Unicode typically contains few hundreds of distinct characters, far fewer than the $\approx 2^{17}$ (and growing) defined in the standard.

Formally, we wish to maintain a sequence of symbols $S = \langle S_0, \ldots, S_{n-1} \rangle$ drawn from an alphabet $\Sigma \subseteq U = \{0, \ldots, u-1\}$, where we call U the *universe* and Σ the *working alphabet*, with Σ typically much smaller than U and not known a priori. We want to support the standard Access, Rank, Select, Insert, and Delete but we are willing to give up RankPrefix and SelectPrefix, which would not be useful anyway for non-string data.

We can immediately use the Wavelet Trie on S, by mapping injectively the values of U to strings of length $\lceil \log u \rceil$. This supports all the required operations with a space bound that depends only logarithmically in u, but the height of the resulting trie could be as much as $\log u$, while a balanced tree would require only $\log |\Sigma|$.

To maintain a balanced tree without having to deal with complex rotations we employ a simple randomized technique that will yield a balanced tree with high probability. The main idea is to randomly permute the universe U with an easily computable permutation, such that the probability that the alphabet Σ will produce an unbalanced trie is negligible.

To this end we use the hashing technique described in [29]. We map the universe U onto itself by the function $h_a(x) = ax \pmod{2^{\lceil \log u \rceil}}$ where a is chosen at random among the odd integers in $[1, 2^{\lceil \log u \rceil} - 1]$ when the data structure is initialized. Note that h_a is a bijection, with the inverse given by $h^{-1}(x) = a^{-1}x \pmod{2^{\lceil \log u \rceil}}$. The result of the hash function is considered as a binary string of $\lceil \log u \rceil$ bits written LSB-to-MSB, and operations on the wavelet tree are defined by composition of the hash function with operations on the Wavelet Trie; in other words, the values are hashed and inserted in a Wavelet Trie, and when retrieved the hash inverse is applied.

To prove that the resulting trie is balanced we use the following lemma from [29].

LEMMA 6.5.1 ([29]) Let $\Sigma \subseteq U$ be any subset of the universe, and $\ell = \lceil (\alpha + 2) \log |\Sigma| \rceil$ so that $\ell \leq \lceil \log u \rceil$. Then the following holds

$$\text{Prob}\left(\forall x, y \in \Sigma \quad h_a(x) \not\equiv h_a(y) \pmod{2^\ell}\right) \geq 1 - |\Sigma|^{-\alpha}$$

where the probability is on the choice of a.

In our case, the lemma implies that with very high probability the hashes of the values in Σ are distinguished by the first ℓ bits, where ℓ is logarithmic in $|\Sigma|$. The trie on the hashes cannot be taller than ℓ, hence it is balanced. The space occupancy is that of the Wavelet Trie built on the hashes. We can bound |L|, the length of the concatenation of trie labels in Theorem 6.2.6, by the total sum of the lengths of the hashes, hence proving the following theorem.

THEOREM 6.5.2 The randomized wavelet tree on a dynamic sequence $S = \langle S_0, \ldots, S_{n-1} \rangle$ where $S_i \in \Sigma \subseteq U = \{0, \ldots, u-1\}$ supports the operations Access, Rank, Select, Insert, and Delete in time $O(\log u + h \log n)$, where $h \leq (\alpha+2) \log |\Sigma|$ with probability $1 - |\Sigma|^{-\alpha}$ (and $h \leq \lceil \log u \rceil$ in the worst case).

The total space occupancy is $O(nH_0(S) + |\Sigma|w) + |\Sigma| \log u$ bits.

7 Practical implementation of succinct data structures

In this chapter we describe the implementation details of the succinct primitives used in the experiments of the previous chapters. As noted in Chapter 1, a thorough engineering of is essential to obtain good performance for succinct data structures. We keep two main goals in mind: *(i)* reduce as much as possible the actual space requirements on realistic data sizes, and *(ii)* guarantee fast query times.

With respect to point *(i)*, the $o(n)$ redundancy typical of most succinct data structures actually dominates the space occupancy in practical implementations. For example, as shown by Vigna [112], because of the large constants hidden in the asymptotic notation, the classical Jacobson-Clark constant-time Rank/Select $o(n)$-space data structure becomes competitive in space with $O(n)$-space solutions only when n is larger than 2^{100}, a data size that is completely unrealistic even in the foreseeable future.

Regarding point *(ii)*, because of the complex memory management of modern CPUs and operating systems, as the data sizes grow large the actual behaviour of the machine diverges significantly from the model of computation. We demonstrate this with the following toy problem. We fill an array A with n random 32-bit integers, fix a constant K, and define the following algorithm.

```
1 function TOY(i)
2     j ← i
3     for k in [0,K) do
4         j ← A[hash(j) mod n]
5     return j
```

The algorithm is clearly constant-time in most computational models, as the loop only performs K iterations. Since each iteration depends on the hashed value of the result of the previous one, the CPU cannot predict the access pattern to A. This toy problem mimicks the memory access patterns of many algorithms on succinct data structures, which use several look-up tables and directories with unpredictable access patterns.

We implemented it in C++ using a fast hash function [114] which does not use multiplication, division, or lookup tables, so that it affects the running time as little as

possible, and measured the average time of invoking TOY(i) for i running from 0 to 20 millions, for several values of K and n. The tests were run on an Intel i7-2600 3.4Ghz CPU with 8MiB of last-level cache and 16GiB of RAM, running Linux 3.2.0 – 64bit. Results are shown in Figure 7.1.

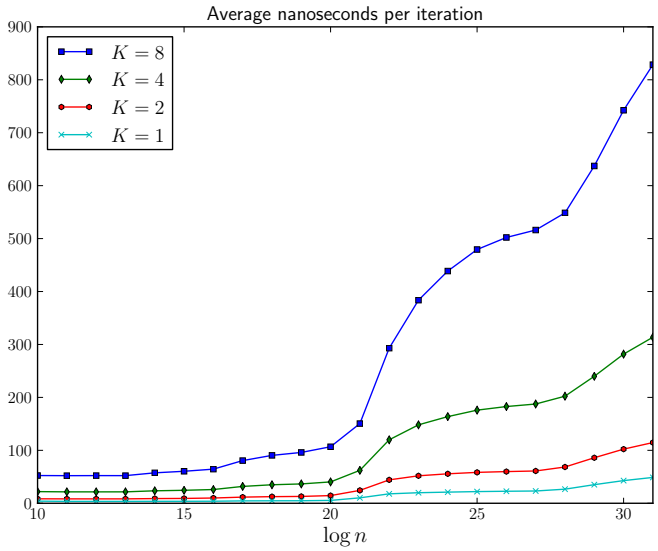


FIGURE 7.1: Average TOY time in nanoseconds for increasing values of K and n.

It is possible to see a small increase in running time as the array A does not fit anymore into L1/L2 cache, that is for $\log n > 16$, and a large slowdown when the array does stops fitting into L3 (last-level) cache, that is for $\log n > 21$. After that, the behavior turns into noticeably super-constant, and actually closer to logarithmic. The effect is more visible as K grows. As explained by González et al. [53], this can be mainly attributed to the translation of virtual memory addresses to physical addresses, performed by the operating system kernel. Even if this translation layer can be skipped, the memory access times in modern CPUs are still highly non-uniform [30].

In light of these considerations, an O(1)-time algorithm may not necessarily be faster than, say, an O($\log n$)-time one, when n is limited by the memory available on an actual machine. Likewise, an $o(n)$-space data structure be not be more space-efficient than an O(n)-space one. Hence, in our implementations we adopted an empirical approach, building and improving on the state of the art of engineered succinct data structures.

7.1 THE *Succinct* LIBRARY

We implemented the data structures described in this chapter as part of the *Succinct* C++ library [107]. The library is available with a permissive license, in the hope that it will be useful both in research and applications. While similar in functionality to other existing

C++ libraries such as libcds [83], SDSL [104], and Sux[108], we made some radically different architectural choices, which we describe below.

MULTI-PLATFORM SUPPORT. The library is tested under Linux, Mac OS X, and Microsoft Windows, compiled with gcc, clang and MSVC.

MEMORY MAPPING. As in most static data structures libraries, all the data structures in *Succinct* can be serialized to disk. However, as opposed to libcds, SDSL, and Sux, deserialization is performed by *memory mapping* the data structure, rather than *loading* it into memory.

While being slightly less flexible, memory mapping has several advantages over loading. For example, for short-running processes it is often not necessary to load the full data structure in memory; instead the kernel will load only the relevant pages. If such pages were accessed recently, they are likely to be still in the kernel's page cache, thus making the startup even faster. If several processes access the same structure, the memory pages that hold it are shared among all the processes; with loading, instead, each process keeps its own private copy of the data. Lastly, if the system runs out of memory, it can just un-map unused pages; with loading, it has to *swap* them to disk, thus increasing the I/O pressure.

For convenience we implemented a mini-framework for serialization/memory mapping which uses template metaprogramming to describe recursively a data structure by implementing a single function that lists the members of a class. The mini-framework then automatically implements serialization and mapping functions, together with profiling functions to print out a breakdown of the space occupied by the various components of an instance of a class, which is extremely useful when optimizing the data structures.

TEMPLATES OVER POLYMORPHISM. We chose to avoid dynamic polymorphism and make extensive use of C++ templates instead. This allowed us to write idiomatic and modular C++ code without the overhead of virtual functions. Manual inspection of the generated code confirmed that, thanks to the ability to inline functions, there is virtually no abstraction penalty on all the compilers we tested the code on.

64-BIT SUPPORT. Like Sux and parts of SDSL, *Succinct* is designed to take advantage of 64-bit architectures, which allow us to use efficient broadword algorithms [76] to speed up several operations on memory words. Another advantage is that the data structures are not limited to 2^{32} elements or less like 32-bit based implementations, a crucial requirement for large datasets, which are the ones that benefit the most from succinct data structures. We also make use of CPU instructions that are not exposed to C++ but are widely available, such as those to retrieve the MSB and LSB of a word, or

to reverse its bytes. While all these operations can all be implemented with broadword algorithms, the specialized instructions are faster.

7.2 BITVECTORS

Succinct implements uncompressed bitvectors as packed arrays of 64-bit words in the class `bit_vector`. The operations supported are the basic ones, such as Access and iteration. Additional classes build on `bit_vector` to support more powerful operations.

7.2.1 *Supporting Rank and Select*

To support Rank we implemented the `rank9` data structure [112] in the `rs_bit_vector` class. `rank9` divides the bitvector into blocks of 64 bits, and into super-blocks of 512 bits. The block ranks relative to the super-blocks then fit into 9 bits, thus all the block ranks for a super-block can be fitted in a 64-bit integers. The in-block rank can be computer by population count of a word, which is implemented using a broadword algorithm [76]. The absolute super-block ranks are stored as plain 64-bit words, and are interleaved with the words representing the block ranks, so that a Rank operation incurs only 2 cache misses, one to retrieve the super-block and the block rank, and one to retrieve the block in the bitvector.

To support Select_1 we use a single-level variation of hinted binary search [112]. We store the position of every 1024th `1` in the bitvector; every other `1` will be between two sampled `1`s, so we can perform a binary search on the super-block ranks that span the range of the two sampled `1`s. Once the super-block is found, the block can be found using a broadword search in the word of the block ranks, and then a select-in-word broadword algorithm is used. While the binary search can require $O(\log n)$ operations, for non-pathological bitvectors the range of superblocks that spans two consecutive sampled `1`s is very small. Even for pathological bitvectors, in our experiments the slowdown is almost negligible. An array of sampled `0` positions can be used similarly to support Select_0.

The space overhead for `rank9` is given by the two 64-bit words for the block and super-block ranks per 512-bit super-block, thus it is 25%. The hints for Select add an overhead that depends on the density, and can be as high as 6.25% for a bitvector that contains only `1`s, but it is generally much lower.

7.2.2 *Supporting Select only*

When there is no need to support Rank, Select can be supported with a simpler data structure described in [96], where it is called `darray`. We implement the same data structure but with larger block sizes, as we can take advantage of 64-bit operations. A similar data structure is called `simple` in [112]. The classes in *Succinct* are called `darray1` for Select_1 and `darray0` for Select_0.

As in the previous section, we only describe the data structure to support Select_1. Support for Select_0 is symmetric. For clarity, our notation slightly deviates from [96].

The bitvector B is first divided into super-blocks of variable length, whose endpoints are located at every L_1-th occurrence of a `1` (so that each super-block, possibly except the last, contains exactly L_1 ones). For each super-block we distinguish two cases, depending on whether the super-block length is smaller or larger than a fixed number T. In the first case (dense super-block), the positions of the `1`s relative to the super-block can be expressed in $\lceil \log T \rceil$ bits, so we further divide the super-block into blocks with L_2 `1`s each, and store the position of the first `1` of each block in $\lceil \log T \rceil$ bits into a *block inventory array*. In the second case (sparse super-block), we write explicitly all the positions of the `1`s into an *overflow array*. For each super-block, we also write a pointer to either its starting position in B if it is dense, or to its overflow array into a *super-block inventory* if it is sparse.

To perform a $\text{Select}_1(i)$ we first look up the $\lfloor i/L_1 \rfloor$-th pointer in the super-block inventory, and check whether the super-block is dense or sparse. In the first case we look up in the block inventory the position of the first one if i's block, and scan the block using broadword operations to find the i-th `1`. If the block is sparse, we just look up its position in the overflow array.

In our implementation we use $L_1 = 1024$, $L_2 = 32$, and $T = 2^{16}$, so that we can use 16-bit integers for the block inventory. For the super-block inventory and the overflow array we use 64-bit integers. The space overhead is dominated by the block inventory, as both the overflow array and the super-block inventory occupy negligible space. With the constants above, each `1` has a 0.5 bit overhead, so the total overhead is slightly above $d \cdot 50\%$, where d is the density of the bitvector.

A slight variation is implemented in the class `darray64`. This data structure assumes that the maximum distance between two consecutive `1`s is 64. In this case we can tune the constants so that $64L_1 \leq T$; in this case the overflow array is not needed. Specifically, we use the same $L_1 = 1024$ as in the `darray`, but since the bitvector is expected to be dense we use a value of $L_2 = 64$. The space overhead is thus roughly half of `darray`.

The `darray64` is used in Chapter 4 to store the bitvector H in the hollow tries, and in Chapter 5 to implement the γ-array. In both cases it is guaranteed that the `1`s are at most 64 bits apart.

7.2.3 *Elias-Fano encoding*

We use the Elias-Fano encoding to represent sparse bitvectors in the class `elias_fano`. The scheme is very simple to implement, and efficient practical implementations are described in [62, 96, 112]. In [96] the data structure is called *sarray*.

The class is a straigtforward implementation of the data structure as described in Section 2.5.2, where we use `darray1` for H to support Select_H.

7.3 Balanced Parentheses

To support FindClose, FindOpen, Enclose, and ±1RMQ on a balanced parentheses sequence we use a variation of the Range Min-Max tree [101, 4] in the class `bp_vector`. This data structure has $O(n)$ redundancy and $O(\log n)$ time operations, but as shown in [4] it is very space-efficient in practice and competitive in time with constant-time implementations.

The Range Min-Max tree is a data structure on $\{-1, 0, +1\}$ sequences that supports a *forward search* operation FwdSearch(i, x): given a position i and a value x, return the leftmost position $j > i$ such that the sum of the values in the sequence in the interval $[i, j]$ is equal to x.

The application to balanced parentheses is straightforward: if the sequence takes value $+1$ on open parentheses on -1 on close parentheses, the cumulative sum of the sequence at position i is the excess. Then, FindClose(i) = FwdSearch$(i, 0) - 1$. In other words, it is the leftmost position j following i such that between i and j there is the same number of open and close parentheses.

Backward search is defined symmetrically, returning the rightmost position j preceding i such that the sum of the values in the interval $[j, i]$ is equal to x; it can be used likewise to implement FindOpen(i) as BwdSearch$(i + 1, 0)$ and Enclose(i) as BwdSearch$(i, -1)$.

The data structure is defined as follows: the sequence is divided into *blocks* of the same size; for each block k are stored the minimum m_k and the maximum M_k of the cumulative sum of the sequence (for balanced parentheses, the excess) within the block. A *tree* is formed over the blocks, which become the leaves, and each node stores the minimum and the maximum cumulative sum among the leaves of its subtree.

To perform the forward search we define the *target value* y as the cumulative sum of the sequence at i, plus x; the result of the forward search is of the leftmost occurrence of the target value y following i in the sequence of the cumulative sums. To find its position we traverse the tree to find the first block k following i where the value x plus the cumulative sum of the sequence at i is between m_k and M_k. Since the sequence has values in $\{-1, 0, +1\}$, the block k contains all the intermediate values between m_k and M_k, and so it must contain y. A linear search is then performed within the block.

The operation ±1RMQ(i, j) can also be implemented using the Range Min-Max tree. By traversing the tree it is possible to find the block with minimum excess among those between i and j that do not contain either i or j. Then a linear search is used to find the minimum in the block containing i, in the one containing j, and in that found by the tree search. The minimum among the three is returned.

7.3.1 *The Range Min tree*

The Range Min-Max tree search works for arbitrary values of x. However, to implement FindClose, FindOpen, and Enclose it is sufficient to support nonpositive values of x. In this case, we can simplify the data structure obtaining smaller space occupancy and better performance. We apply the following modifications, and call the resulting data structure *Range Min tree*.

HALVING THE TREE SPACE OCCUPANCY. We discard the maxima in the tree nodes, and store only the minima. The block search in the tree then returns the leftmost block whose minimum is smaller than y. The following lemma guarantees that the forward search is correct. A symmetric argument holds for the backward search.

LEMMA 7.3.1 Let $j = \text{FwdSearch}(i, x)$ for $x \leq 0$. Then the Range Min tree block search finds the block that contains j.

Proof Since $x \leq 0$ and the sequence has values in $\{-1, 0, +1\}$, the values of the cumulative sums following i and preceding j must be all greater than the cumulative sum at j. Hence the leftmost block k that has minimum smaller than y must also contain y. ∎

The implementation of $\pm$1RMQ described above works without any modification, as it only needs the block minima.

Discarding the node maxima effectively halves the space occupancy of the tree, compared to the Range Min-Max tree. Although we expected that the reduction in space would also be beneficial for cache locality, the experiments did not show any measurable improvement in query times.

BROADWORD IN-BLOCK SEARCH. The in-block forward and backward search performance is crucial as it is the inner loop of the search. In practical implementations [4, 104] it is usually performed byte-by-byte with a lookup table that contains the solution for each possible combination of byte and excess (hence $256*8 = 2048$ bytes). This algorithm, which we will call *Lookup/Loop* in the following, involves many branches and accesses to a fairly big lookup tables for each byte. Suppose, instead, that we know which byte contains the closing parenthesis; we can then use the lookup table only on that byte.

To find that byte we can use the same trick as in the Range Min tree: the first byte with min-excess smaller than the target excess must contain the closing parenthesis. This byte can be found by using a pre-computed lookup table with the min-excess of every possible byte (hence 256 bytes), and checking all the bytes in the word with a loop, while updating the current excess at each loop iteration by using a second lookup table. We call this algorithm *RangeMin/Loop*.

The above algorithm still involves some hard to predict branches inside the loop. To get rid of them, we use a hybrid lookup table/broadword approach, which we call *RangeMin/Broadword*.

We divide the block into machine words. For each word w we compute the word m_8 where the i-th byte contains the min-excess of the i-th byte in w with inverted sign, so that it is non-negative, by using the same pre-computed lookup table used in RangeMin/Loop. At the same time we compute the byte counts c_8 of w, where the i-th byte contains the number of 1s in the i-th byte of w, using a broadword algorithm [76].

Using the equality $\text{Excess}(i) = 2 \cdot \text{Rank}_(i) - i$ we can easily compute the excess for each byte of w: if e_w is the excess at the starting position of w, the word e_8 whose i-th byte contains the excess of the i-th byte of w can be obtained through the following formula:[1]

$$e_8 = (e_w + ((2 * c_8 - \texttt{0x...08080808}) \texttt{ << } 8)) * \texttt{0x...01010101}.$$

Now we have all we need: the closing parenthesis is in the byte where the excess function crosses the target excess, in other words in the byte whose excess added to the min-excess is smaller than the target. Hence we are looking for the first byte position in which e_8 is smaller than m_8 (recall that the bytes in m_8 are negated). This can be done using the $\leq_8$ operation described in [76] to compute a mask $l_8 = e_8 \leq_8 m_8$, where the i-th byte is 1 if and only if the i-th byte of e_8 is smaller than the i-th byte of m_8. If the l_8 is zero, the word does not contain the closing parenthesis; otherwise, an LSB operation quickly returns the index of the byte containing the solution. The same algorithm can be applied symmetrically for the FindOpen.

Overall, for 64-bit words we perform 8 lookups from a very small table, a few tens of arithmetic operations and one single branch (to check whether the word contains the solution or not). In the following section we show that RangeMin/Broadword is significantly faster than Lookup/Loop.

IMPLEMENTATION DETAILS. We build our Range Min tree implementation on top of `rs_bit_vector`, so that we can support Excess in constant time by using Rank. We use blocks of 256 bits, and group them blocks into super-blocks of 32 blocks. We store the block excess minima relative to the super-block. On the super-blocks we build a complete implicit binary tree, storing the absolute block minima. Since the super-blocks contain 8192 parentheses, we can use signed 16-bit integers to store the relative excess minima, while we use 32-bit integers to store the absolute excess minima, thus limiting the excess in the range $[-2^{32}, 2^{32} - 1]$, however this limitation can be removed by using 64-bit integers, for a relatively small space overhead. We use RangeMin/Broadword for the in-word FindClose and FindOpen, and a linear scan using 8-bit lookup tables for the in-block $\pm$1RMQ.

[1] A subtlety is needed here to prove that the search is correct: the excess can be negative, hence the carry in the subtraction corrupts the bytes after the first byte that contains the zero. However, this means that the word *contains* the solution, and the closing parenthesis is in the byte that precedes the one where the sampled excess becomes negative.

7.3.2 *Experimental analysis*

We evaluated the performance of our Range-Min tree implementation with different variations of the in-block search, and against other Range Min-Max tree implementations. To this end, we generated a set of random binary trees of several sizes, encoded with the DFBS representation described in Section 2.6.3, and measured the average FindClose time while performing random root-to-leaf traversals, to mimick typical visit operations. Both the random trees and the random path directions are equal across benchmarks of different implementations. Each result is averaged over 20 runs on different random trees, and the lengths of the random paths sum up to 10 millions per run.

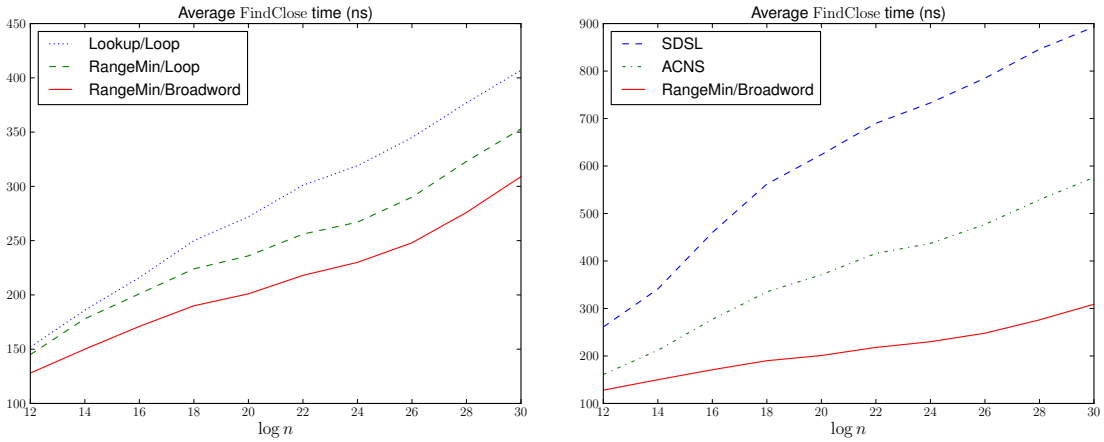
We benchmarked our implementations of Lookup/Loop, RangeMin/Loop and RangeMin/Broadword, the implementation of [4], which we refer to as ACNS, and the implementation of [104], which we refer to as SDSL. The experiments were run on an Intel Core i5 1.6Ghz running Mac OS X Lion, compiled with the stock compiler GCC 4.2.1. The results are shown in Figure 7.2 and Table 7.1.

	Redundancy	Average FindClose time (ns)									
$\log n$		12	14	16	18	20	22	24	26	28	30
SDSL [104]	42%	261	341	460	562	624	690	733	785	846	893
ACNS [4]	15%	161	212	277	335	371	416	437	477	529	576
Lookup/Loop	32%	152	186	216	250	272	301	319	345	377	407
RangeMin/Loop	32%	145	178	201	224	236	256	267	290	323	353
RangeMin/Broadword	32%	128	150	171	190	201	218	230	248	276	309

TABLE 7.1: Average space redundancy and FindClose time in nanoseconds for different implementations. Each implementation was tested on sequences of different lengths, with $\log n$ being the logarithm of the sequence length.

On the longest sequences, RangeMin/Loop is $\sim$ 20% faster than Lookup/Loop, while RangeMin/Broadword is $\sim$ 15% faster than RangeMin/Loop. Overall, RangeMin/Broadword is $\sim$ 30% faster than RangeMin/Loop. We also tested our implementations with newer versions of GCC and MSVC, and found out that the performance difference between RangeMin/Loop and RangeMin/Broadword vanishes, still being 20% to 30% faster than Lookup/Loop, due to better optimizations. This suggests that the choice between RangeMin/Loop and RangeMin/Broadword should depend on the particular compiler/architecture combination, with RangeMin/Broadword being a safe default.

When compared with other implementations, our implementation with RangeMin/Broadword is both faster and smaller than SDSL, and faster than ACNS, while occupying more space. It should be noted, however, that the difference in space is entirely caused by the data structure used to support Rank (and thus Excess): while ACNS uses an implementation with an 8% overhead, we use the broadword data structure of [112], which has a 25% overhead. If we account for this, the space used by the two implementations for supporting BP operations is roughly the same.



(a) Comparison of Lookup/Loop, RangeMin/Loop and RangeMin/Broadword

(b) Comparison against other implementations

FIGURE 7.2: Average FindClose time in nanoseconds for increasing sequence length.

7.4 RANGE MINIMUM QUERIES

To support Range Minimum Queries we use the 2d-Min-Heap described by Fischer and Heun [44], which, as noted by Davoodi et al. [26], is an alternative representation of the Cartesian tree. In *Succinct*, the class is called `cartesian_tree`, and it supports sequences of values with any arbitrary ordering function.

As in [44], we build the DFUDS representation of the 2d-Min-Heap in the bitvector U, which we store in a `bp_vector` to support FindOpen and ±1RMQ.

The RMQ of i and j can then be reduced to the RMQ on the excess sequence E of U, denoted as ±1RMQ, with the following algorithm. The indices are slightly different from [44] because all our primitives are 0-based.

LEMMA 7.4.1 ([44, SECTION 5.1]) The following algorithm, where all the operations are performed on U, returns $\text{RMQ}(i, j)$.

1 $x \leftarrow \text{Select}_)(i+1)$
2 $y \leftarrow \text{Select}_)(j+1)$
3 $w \leftarrow \pm1\text{RMQ}(x, y)$
4 **if** $\text{Rank}_)(\text{FindOpen}(w-1)) = i+1$ **then**
5 **return** i
6 **else**
7 **return** $\text{Rank}_)(w-1)$

Our implementation differs at line 4 in Lemma 7.4.1, which checks whether the node at position x is the parent of $w-1$. We replace it with the following line.

4 **if** $\text{Excess}(\text{Select}_)(i)+1) \leq \text{Excess}(w)$ **then**

Since, in our implementation, Select is significantly faster than FindOpen, the whole algorithm speeds up by 10–20% with this change. The following lemma proves that the two conditions are equivalent.

LEMMA 7.4.2 In the algorithm described in Lemma 7.4.1, $\text{Excess}(\text{Select}_)(i)+1) \leq \text{Excess}(w)$ if and only if the node at $w-1$ in U is a child of the node at $\text{Select}_)(i+1)$.

Proof Let $t = \text{Select}_)(i)$ and $x = \text{Select}_)(i+1)$. If the node at $w-1$ is a child of the node at x, then its mate z is between t and x as shown below:

$$\underset{t}{)}\,\underset{z}{(}\cdots\,\underset{x}{()}\cdots\cdots\cdots\,\underset{w-1}{)}$$

If follows that $\text{Excess}(t+1) \leq \text{Excess}(z)$; since $\text{Excess}(z) = \text{Excess}(w)$, we obtain $\text{Excess}(t+1) \leq \text{Excess}(w)$.

Conversely, suppose $\text{Excess}(t+1) \leq \text{Excess}(w)$. Since $w = \pm 1\text{RMQ}(x,y)$, it holds $\text{Excess}(w) < \text{Excess}(x)$. Hence there must be a position z such that $t+1 \leq z < x$ and $\text{Excess}(z) = \text{Excess}(w)$. To prove that z is the mate of $w-1$, it suffices to note that the excess between $z+1$ and $w-1$ is strictly greater than $\text{Excess}(w)$, again because w is the leftmost excess minimum in the range $[x,y]$. ∎

7.4.1 *Experimental analysis*

We compared our implementation of RMQ against two other publicly available implementations. The first is Fischer's constant-time succinct scheme [43], which divides the sequence A into a two-level block structure and computes and stores the cartesian tree of each block. Using this structure, the search is narrowed down to 5 positions of A, which are accessed explicitly to find the overall minimum. For this reason, the RMQ algorithm needs random access to the sequence A. The second implementation is the SDSL [104] implementation of the succinct RMQ scheme described by Fischer and Gog [51], which uses a succinct encoding of the cartesian tree and a new operation, called `rr_enclose`, which is implemented using a Range Min-Max tree. According to [51], this data structure is both more space-efficient and faster than the original implementation of the Fischer and Heun's succinct 2d-Min-Heap [44]. Unlike the first data structure, and like our implementation, the SDSL implementation does not need to access the sequence A, which can be discarded after construction.

To perform the experiments we generated arrays of increasing lengths filled with random integers from the interval $[0, 1024)$, performed 10M random RMQ queries, and averaged the running time. All the code was compiled with g++ 4.7. The tests were run on a dual core Intel Core 2 Duo E8400 with 6MiB L2 cache and 4GiB RAM, running Linux 3.5.0 - 64-bit. Each test was run 10 times, and the running times averaged. The results are shown in Figure 7.3 and Table 7.2.

	Bits per value	Average RMQ time (ns)									
$\log n$		10	12	14	16	18	20	22	24	26	28
Fischer O(1) [43]	6.6†	80	104	119	128	139	165	415	596	761	901
SDSL [104]	2.5	834	1476	2267	2523	2796	2887	2960	3088	3765	4273
Succinct	2.7	270	331	394	466	536	599	641	718	1213	1654

TABLE 7.2: Average space per value in bits and RMQ time in nanoseconds for different implementations. † The succinct scheme of Fischer also needs to store the sequence A.

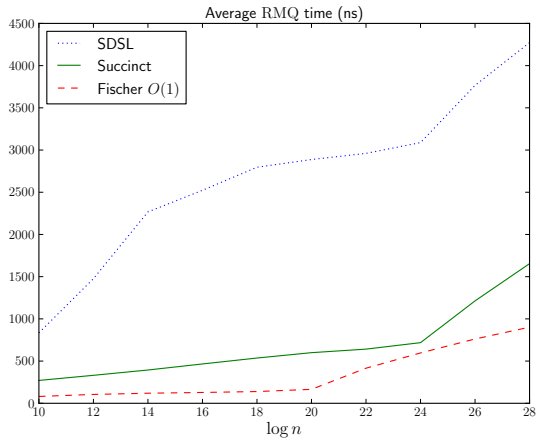


FIGURE 7.3: Average RMQ time in nanoseconds for increasing sequence length.

Fischer's constant-time scheme is overall the fastest, about 2 times faster than Succinct, but it is also the one with the largest overhead, at 6.6 bits per element, plus the space to store A. The implementation requires to store A as an explicit integer array (specifically, 32-bit integers); if A was stored in compressed form, the cost to random-access the 5 values of A during the RMQ query would add up in the query time. SDSL and our implementation obtain very similar space overheads, with Succinct slightly larger due to the `rank9` data structure. However, Succinct is 3 times faster than SDSL.

The plot in Figure 7.3 shows that the CPU cache has a very large effect on the running time. While Fischer's scheme is theoretically constant-time, it exhibits a logarithmic-time behavior once the sequence A does not fit anymore in cache, which happens at about $\log n = 20$. Succinct shows a logarithmic behavior across all data sizes, with a jump in the constant as the sequence length passes $\log n = 24$; the cut point is higher than the constant-time implementation because of the higher space efficiency of the succinct 2d-Min-Heap compared to the explicit storage of the array A. SDSL has significantly worse performance than both the others, and it seems to have two cut points, one at $\log n = 14$ and one at $\log n = 24$, probably due to the use of large lookup tables.

8 Open problems and future directions

We conclude this thesis by presenting some future directions of research, which we believe to be the most promising and interesting.

MORE POWERFUL SEMI-INDEXES. The semi-index presented in Chapter 3 supports tree traversal operations similar to those supported by standard pointer-based representations. However, in many applications more powerful search operations are needed, from simple sub-path searches to full regular expressions on trees. There is a vast literature, for example, on XML indexing (see references in [57]), but the existing indexes are not space-efficient. Compressed data structures for labeled trees [40] support sub-path queries, but, as discussed in the chapter, they require to change the representation of the data. An interesting development would be to augment the semi-index with succinct indexes on the *content* of the documents in order to support powerful search operations, for example by indexing the object keys, while maintaining its space-efficiency.

EXTERNAL-MEMORY COMPRESSED DICTIONARIES. The centroid path-decomposed tries described in Chapter 4 have shown very good performance in the experiments in internal memory, thanks to their $O(\log n)$ height. In an external memory scenario, however, the height is still far from the optimal $O(\log_B n)$. Ferragina et al. [39] presented a cache-oblivious succinct data structure to store a set of strings, but unlike our tries the labels are not compressed, and the data structure does not seem to be practical for realistic data sizes without a significant algorithm engineering effort. An efficient implementation of an external-memory or cache oblivious compressed trie with provable performance guarantees would be of very high practical interest. An I/O-efficient construction algorithm would also be useful in this scenario, as both the intermediate data and the final data structure are unlikely to fit into main memory.

BETTER LABEL COMPRESSION IN TRIES. While the label compression described in Chapter 4 works reasonably well in practice, it is non-optimal: the Re-Pair algorithm [77] optimizes for an objective function that does not take into account the size of the dictionary, but only the number of rules generated; furthermore, it is a greedy

non-optimal algorithm even for the original objective function. For these reasons, we believe that the compression ratio can be improved significantly without changing the data structure, just using an optimal algorithm for the computation of the dictionary and of the parsing. However it is not clear whether this problem is tractable, even in an approximate fashion.

DYNAMIC COMPRESSED DICTIONARIES. In many applications of string dictionaries it is necessary to update the dictionary, by inserting or deleting strings. However, we are not aware of any succinct *dynamic* string dictionary. It is natural to ask whether our compressed tries can be made dynamic without a significant overhead in time and space.

Another interesting development would be a data structure for top-k completions where the string set is fixed but the scores can change over time, which is a realistic scenario in many applications.

PRACTICAL IMPLEMENTATION OF WAVELET TRIES. Wavelet trees have proven very effective in practical applications, and several efficient implementations have been proposed (see Navarro [94] for a survey). The static Wavelet Trie described in Chapter 6 is a straightforward extension, and it should be easy to implement and very practical. However, we expect that the main applications of Wavelet Tries arise in dynamic settings; in this scenario, an efficient implementation of dynamic compressed bitvectors would be necessary. Unfortunately, we are not aware of any practical implementation of dynamic Rank/Select bitvectors, either append-only or fully-dynamic. Engineering efficient dynamic compressed bitvectors is thus an extremely promising line of research, which would have direct application in several other scenarios as well, such as construction of BWT and FM-indexes in compressed working space [79, 55, 85].

BETTER BOUNDS FOR WAVELET TRIES. Navarro and Nekrich [95] recently introduced a dynamic wavelet tree on integer alphabets with optimal $O(\log n / \log\log n)$ operations, although amortized for the update operations, and optimal space bounds. Even on balanced Wavelet Tries, this is better by a factor of $O(\log|\Sigma| \log\log n)$, although the Wavelet Trie bounds are worst-case. It is possible to extend their wavelet tree to sequences of strings by using a dynamic mapping between strings and integers, but it is not clear whether RankPrefix and SelectPrefix can be supported efficiently. It is thus natural to ask whether it is possible to obtain a Wavelet Trie with worst-case bounds closer to $O(\log n / \log\log n)$, while still supporting prefix operations.

EXTERNAL-MEMORY WAVELET TRIES. A natural application of Wavelet Tries is to store table columns in a relational database scenario. In these applications it is usually necessary to store the data structure on disk, which makes the Wavelet Trie unsuitable because of both the unbalancedness of the structure and the number of random memory

accesses made by the dynamic bitvectors, which is $O(\log n)$ regardless of the disk block size. Hence, an important open question is whether it is possible to design a data structure that supports all the operations supported by the Wavelet Trie, with better time bounds in the external memory or cache oblivious models.

BIBLIOGRAPHY

[1] A. Acharya, H. Zhu, and K. Shen. Adaptive algorithms for cache-efficient trie search. In *Algorithm Engineering and Experimentation, International Workshop (ALENEX)*, pages 296–311, 1999. → 41

[2] AOL search data. `http://www.gregsadetsky.com/aol-data/`, 2006. → 50, 61

[3] Apache Xerces2 XML Parser. `http://xerces.apache.org/xerces-j/`. → 24, 37

[4] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 84–97, 2010. → 4, 42, 90, 91, 93

[5] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Monotone minimal perfect hashing: searching a sorted table with O(1) accesses. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 785–794, 2009. → 47

[6] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Theory and practice of monotone minimal perfect hashing. *ACM Journal of Experimental Algorithmics*, 16:3.2:3.1–3.2:3.26, 2011. → 42, 47, 52

[7] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 88–94, 2000. → 19

[8] M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul. Cache-oblivious string B-trees. In *Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 233–242, 2006. → 41

[9] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005. → 14, 42

[10] I. Bialynicka-Birula and R. Grossi. Rank-sensitive data structures. In *String Processing and Information Retrieval, 12th International Conference (SPIRE)*, pages 79–90, 2005. → 56

[11] Binary XML. `http://en.wikipedia.org/wiki/Binary_XML`. → 21

[12] D. K. Blandford and G. E. Blelloch. Compact dictionaries for variable-length keys and data with applications. *ACM Transactions on Algorithms*, 4(2), 2008. → 42

[13] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004. → 50

[14] P. Boldi and S. Vigna. Codes for the World Wide Web. *Internet Mathematics*, 2(4):407–429, 2005. → 60

[15] N. R. Brisaboa, R. Cánovas, F. Claude, M. A. Martínez-Prieto, and G. Navarro. Compressed string dictionaries. In *Experimental Algorithms - 10th International Symposium (SEA)*, pages 136–147, 2011. → 42, 46, 50, 52, 56

[16] G. S. Brodal and R. Fagerberg. Cache-oblivious string dictionaries. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 581–590, 2006. → 41

[17] BSON specification. `http://bsonspec.org/`. → 22

[18] S.-Y. Chiu, W.-K. Hon, R. Shah, and J. S. Vitter. I/O-efficient compressed text indexes: From theory to practice. In *Data Compression Conference (DCC)*, pages 426–434, 2010. → 42

[19] D. R. Clark. *Compact pat trees*. PhD thesis, University of Waterloo, Waterloo, Ont., Canada, Canada, 1998. Ph.D. Thesis, UMI Order No. GAXNQ-21335. → 10

[20] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *String Processing and Information Retrieval, 15th International Symposium (SPIRE)*, pages 176–187, 2008. → 67

[21] F. Claude and G. Navarro. Fast and compact web graph representations. *ACM Transactions on the Web*, 4(4), 2010. → 46

[22] CouchDB. `http://couchdb.apache.org/`. → 21

[23] CouchDB in the wild. `http://wiki.apache.org/couchdb/CouchDB_in_the_wild`. → 21

[24] A. Couthures. JSON for XForms. In *Proc. XMLPrague 2011*, 2011. → 24

[25] T. M. Cover and J. A. Thomas. *Elements of information theory*. John Wiley and Sons, Inc., 1991. → 8

[26] P. Davoodi, R. Raman, and S. S. Rao. Succinct representations of binary trees for range minimum queries. In *Computing and Combinatorics - 18th Annual International Conference (COCOON)*, pages 396–407, 2012. → 16, 19, 94

[27] J. Dean and S. Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010. → 21

[28] O. Delpratt, R. Raman, and N. Rahman. Engineering succinct DOM. In *11th International Conference on Extending Database Technology (EDBT)*, pages 49–60, 2008. → 24

[29] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997. → 84

[30] U. Drepper. What every programmer should know about memory. `http://www.akkadia.org/drepper/cpumemory.pdf`, 2007. → 86

[31] H. Duan and B.-J. P. Hsu. Online spelling correction for query completion. In *Proceedings of the 20th International Conference on World Wide Web (WWW)*, pages 117–126, 2011. → 64

[32] P. Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21(2):246–260, 1974. → 10

[33] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975. → 48, 60, 81

[34] R. Fano. On the number of bits required to implement an associative memory. *Computer Structures Group, Project MAC, MIT, Cambridge, Mass., nd*, 1971. Memorandum 61. → 10

[35] F. Farfán, V. Hristidis, and R. Rangaswami. 2LP: A double-lazy XML parser. *Information Systems*, 34(1):145–163, 2009. → 24

[36] A. Farzan and J. I. Munro. Succinct representation of dynamic trees. *Theoretical Computer Science*, 412(24):2668–2678, 2011. → 79

[37] P. Ferragina, R. Giancarlo, and G. Manzini. The myriad virtues of wavelet trees. *Information and Computation*, 207(8):849–866, 2009. → 81

[38] P. Ferragina and R. Grossi. The String B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999. → 41, 47

[39] P. Ferragina, R. Grossi, A. Gupta, R. Shah, and J. S. Vitter. On searching compressed string collections cache-obliviously. In *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, 2008. → 41, 42, 43, 47, 67, 71, 97

[40] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM*, 57(1), 2009. → 24, 42, 67, 97

[41] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005. → 56

[42] P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. *Theoretical Computer Science*, 372(1):115–121, 2007. → 2

[43] J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, First International Symposium (ESCAPE)*, pages 459–470, 2007. → 95, 96

[44] J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011. → 19, 94, 95

[45] L. Foschini, R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Transactions on Algorithms*, 2(4):611–639, 2006. → 81, 82

[46] E. Fredkin. Trie memory. *Communications of the ACM*, 3:490–499, 1960. → 17

[47] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993. → 77

[48] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, 1994. → 9

[49] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4, 2012. → 10

[50] T. Gagie, G. Navarro, and S. J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426:25–41, 2012. → 82

[51] S. Gog and J. Fischer. Advantages of shared data structures for sequences of balanced parentheses. In *Data Compression Conference (DCC)*, pages 406–415, 2010. → 95

[52] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *Proceedings of the Fourteenth International Conference on Data Engineering (ICDE)*, pages 370–379, 1998. → 61

[53] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of Rank and Select queries. In *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, 2005. → 86

[54] R. González and G. Navarro. Statistical encoding of succinct data structures. In *Combinatorial Pattern Matching, 17th Annual Symposium (CPM)*, pages 294–305, 2006. → 2

[55] R. González and G. Navarro. Rank/Select on dynamic compressed sequences and applications. *Theoretical Computer Science*, 410(43):4414–4422, 2009. → 68, 74, 76, 98

[56] Google Books n-grams. `http://storage.googleapis.com/books/ngrams/books/datasetsv2.html`. → 61

[57] G. Gou and R. Chirkova. Efficiently querying large XML data repositories: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(10):1381–1403, 2007. → 24, 97

[58] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete mathematics: a foundation for computer science*. Addison-Wesley Longman Publishing Co., Inc., 1989. → 9

[59] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003. → 11, 69, 71

[60] R. Grossi and G. Ottaviano. Fast compressed tries through path decompositions. In *Meeting on Algorithm Engineering & Experiments (ALENEX)*, pages 65–74, 2012. → 3, 56

[61] R. Grossi and G. Ottaviano. The Wavelet Trie: maintaining an indexed sequence of strings in compressed space. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 203–214, 2012. → 4

[62] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005. → 56, 89

[63] Hive JSON SerDe. `http://code.google.com/p/hive-json-serde/`. → 21

[64] W.-K. Hon, R. Shah, and J. S. Vitter. Space-efficient framework for top-k string retrieval problems. In *50th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 713–722, 2009. → 56, 57

[65] B.-J. P. Hsu and G. Ottaviano. Space-efficient data structures for top-k completion. In *Proceedings of the 22th International Conference on World Wide Web (WWW)*, 2013. → 4

[66] J. Hunter. A JSON facade on MarkLogic server. In *Proc. XMLPrague 2011*, 2011. → 24

[67] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my data files. here are my queries. where are my results? In *5th International Conference on Innovative Data Systems Research (CIDR)*, 2011. → 23

[68] G. Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989. → 10, 12, 13, 51

[69] J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees with applications. *Journal of Computer and System Sciences*, 78(2):619–631, 2012. → 16

[70] Jaql. `http://code.google.com/p/jaql/`. → 21

[71] JSON dump of Delicious bookmarks, September 2009. `http://infochimps.com/datasets/delicious-bookmarks-september-2009`. → 34

[72] JSON specification. `http://json.org/`. → 21

[73] JsonCpp. `http://jsoncpp.sourceforge.net/`. → 35

[74] M. Kay. Ten reasons why Saxon XQuery is fast. *IEEE Data Engineering Bulletin*, 31(4):65–74, 2008. → 24

[75] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, USA, 2 edition, 1998. → 41

[76] D. E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley Professional, 12th edition, 2009. → 2, 87, 88, 92

[77] N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *Data Compression Conference (DCC)*, pages 296–305, 1999. → 46, 97

[78] Laboratory for Web Algorithmics - Datasets. `http://law.dsi.unimi.it/datasets.php`, 2011. → 50

[79] S. Lee and K. Park. Dynamic compressed representation of texts with rank/select. *Journal of Computing Science and Engineering*, 3(1):15–26, 2009. → 68, 74, 98

[80] G. Li, S. Ji, C. Li, and J. Feng. Efficient type-ahead search on relational data: a TASTIER approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 695–706, 2009. → 56

[81] G. Li, J. Wang, C. Li, and J. Feng. Supporting efficient top-k queries in type-ahead search. In *The 35th International ACM SIGIR conference on research and development in Information Retrieval (SIGIR)*, pages 355–364, 2012. → 56

[82] M. Li and P. M. Vitnyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer, 3 edition, 2008. → 8

[83] libcds - Compact Data Structures Library. `http://libcds.recoded.cl/`, 2011. → 87

[84] V. Mäkinen and G. Navarro. Position-restricted substring searching. In *Theoretical Informatics, 7th Latin American Symposium (LATIN)*, pages 703–714, 2006. → 68

[85] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, 4(3), 2008. → 68, 74, 76, 81, 82, 98

[86] G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001. → 9

[87] D. Matani. An O($k \log n$) algorithm for prefix based ranked autocomplete. `http://www.dhruvbird.com/autocomplete.pdf`, 2011. Preprint. → 56

[88] MongoDB. `http://www.mongodb.org/`. → 21

[89] MongoDB Production Deployments. `http://www.mongodb.org/display/DOCS/Production+Deployments`. → 21

[90] D. R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968. → 17

[91] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001. → 12, 13, 14, 15, 42

[92] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 657–666, 2002. → 57

[93] A. Nandi and H. V. Jagadish. Effective phrase prediction. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*, pages 219–230, 2007. → 56

[94] G. Navarro. Wavelet trees for all. In *Combinatorial Pattern Matching - 23rd Annual Symposium (CPM)*, pages 2–26, 2012. → 98

[95] G. Navarro and Y. Nekrich. Optimal dynamic sequence representations. In *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2013. → 68, 98

[96] D. Okanohara and K. Sadakane. Practical entropy-compressed Rank/Select dictionary. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007. → 49, 88, 89

[97] G. Ottaviano and R. Grossi. Semi-indexing semi-structured data in tiny space. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management (CIKM)*, pages 1485–1494, 2011. → 3, 34

[98] M. H. Overmars. *The design of dynamic data structures*, volume 156 of *Lecture Notes in Computer Science*. Springer-Verlag, 1983. → 77, 78, 79, 80

[99] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding n-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007. → 10, 71, 73

[100] K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1230–1239, 2006. → 2

[101] K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 134–149, 2010. → 2, 4, 13, 42, 90

[102] S. Sakr. XML compression techniques: A survey and comparison. *Journal of Computer and System Sciences*, 75(5):303–322, 2009. → 22, 24

[103] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: a benchmark for XML data management. In *Proceedings of the 28th international conference on Very Large Data Bases (VLDB)*, pages 974–985, 2002. → 34

[104] SDSL 0.9 - Succinct Data Structure Library. http://www.uni-ulm.de/in/theo/research/sdsl.html. → 87, 91, 93, 95, 96

[105] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 1948. → 8

[106] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983. → 43

[107] *Succinct* library. http://github.com/ot/succinct, 2011. → 4, 86

[108] Sux 0.7 and Sux4J 2.0.1 - Implementing Succinct Data Structures. http://sux.dsi.unimi.it/, 2011. → 52, 87

[109] The Open Library, JSON dump of author records. http://infochimps.com/datasets/the-open-library. → 34

[110] Tx 0.18 - Succinct Trie Data Structure. http://code.google.com/p/tx-trie/, 2010. → 51, 52

[111] R. Vernica and C. Li. Efficient top-k algorithms for fuzzy search in string collections. In *roceedings of the First International Workshop on Keyword Search on Structured Data (KEYS)*, 2009. → 56

[112] S. Vigna. Broadword implementation of Rank/Select queries. In *Experimental Algorithms, 7th International Workshop (WEA)*, pages 154–168, 2008. → 85, 88, 89, 93

[113] J. S. Vitter. *Algorithms and Data Structures for External Memory*. Foundations and trends in theoretical computer science. Now Publishers, 2008. → 10, 67

[114] T. Wang. Integer hash function. http://www.cris.com/~Ttwang/tech/inthash.htm, 2007. → 85

[115] Wikipedia database dumps. http://download.wikimedia.org/. → 34

[116] Wikipedia raw page counts. http://dumps.wikimedia.org/other/pagecounts-raw/. → 61

[117] H. E. Williams and J. Zobel. Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201, 1999. → 46

[118] R. K. Wong, F. Lam, and W. M. Shui. Querying and maintaining a compact XML storage. In *Proceedings of the 16th International Conference on World Wide Web (WWW)*, pages 1073–1082, 2007. → 24