Scalable succinct indexing for large text collections

A thesis submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

Matthias Petri
M.Sc.

School of Computer Science and Information Technology
College of Science, Engineering, and Health
RMIT University
Melbourne, Victoria, Australia.

July 2013

Declaration

I certify that except where due acknowledgment has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; and, any editorial work, paid or unpaid, carried out by a third party is acknowledged.

Matthias Petri
School of Computer Science and Information Technology
RMIT University
July 2013

Credits

Portions of the material in this thesis have previously appeared in the following refereed publications:

- J. S. Culpepper, M. Petri, and S. J. Puglisi. Revisiting bounded context block-sorting transformations. Software Practice and Experience, 42(8):pp 1037-1054, Aug 2012.
- M. Petri and J. S. Culpepper. Efficient indexing algorithms for approximate pattern matching in text. Proceedings of the 17th Annual Australasian Document Computing Symposium (ADCS 2012) , pp 9-16, December 2012. (best paper)
- J. S. Culpepper, M. Petri, and F. Scholer. Efficient in-memory top-k document retrieval. Proceedings of the 35th Annual International Conference on Research and Development in Information Retrieval (SIGIR 2012) , pp 225-234, August 2012.
- M. Petri, G. Navarro, J. S. Culpepper, and S. J. Puglisi. Backwards search in context-bound text transformations. Proceedings of the First International Conference on Data Compression, Communication and Processing (CCP 2011) , IEEE Press, pp 82-91 Jun 2011.
- S. Gog, and M. Petri. Optimizing Succinct Data Structures. Software Practice and Experience, (to appear)

Parts of my thesis also contributed to the following workshop papers and non refereed publications:

- M. Yasukawa, J. S. Culpepper, F. Scholer, and M. Petri. RMIT and Gunma University at NTCIR-9 Intent Task. In Proceedings of the NTCIR-9 Workshop Meeting, December 2011.
- M. Petri, J. S. Culpepper, and F. Scholer. RMIT at TREC 2011 Microblog Track. In Proceedings of the 20th Text REtrieval Conference (TREC 2011), November 2011

Acknowledgements

I'm offering my insubstantial gratitude to my supervisors Shane Culpepper and Falk Scholar for providing guidance and supporting me throughout these years.

I would also like to thank my parents for the support they provided me throughout my life. I must also acknowledge my partner and best friend, Irry, without whose reassurance, love and editorial assistance, I would not have finished this thesis.

This work was supported by RMIT University and NICTA.

Contents

List of Figures

List of Tables

Abstract

Self-indexes save space by emulating operations of traditional data structures using basic operations on bitvectors. Succinct text indexes provide *full-text search* functionality which is traditionally provided by suffix trees and suffix arrays for a given text, while using space equivalent to the compressed representation of the text. Succinct text indexes can therefore provide full-text search functionality over inputs much larger than what is viable using traditional uncompressed suffix-based data structures.

Fields such as Information Retrieval involve the processing of massive text collections. However, the in-memory space requirements of succinct text indexes during construction have hampered their adoption for large text collections. One promising approach to support larger data sets is to avoid constructing the full suffix array by using alternative indexing representations.

This thesis focuses on several aspects related to the scalability of text indexes to larger data sets. We identify practical improvements in the core building blocks of all succinct text indexing algorithms, and subsequently improve the index performance on large data sets. We evaluate our findings using several standard text collections and demonstrate: (1) the practical applications of our improved indexing techniques; and (2) that succinct text indexes are a practical alternative to inverted indexes for a variety of top-ϕranked document retrieval problems.

Notation

Here we give an overview of the notation used in this thesis.

Symbol	Description
BWT	Fully sorted Burrows Wheeler Transform.
k-BWT	Context-Bound Burrows Wheeler Transform sorted to depth k.
v-BWT	Context-Bound Burrows Wheeler Transform with threshold v.
T	Input sequence of length n.
n	Length of the bitvector or text T.
σ	Size of the alphabet, Σ, all symbols in T are drawn from.
T^{bwt}	Text T transformed using the BWT corresponding to the last column (L) of $\mathcal{M}$.
T^{kbwt}	Text T transformed using the k-BWT. corresponding to the last column (L) of $\mathcal{M}_k$.
$T^{v\text{-BWT}}$	Text T transformed using the v-BWT. Corresponds to the last column of $\mathcal{M}_v$.
k	Sorting depth of the k-BWT.
v	Threshold of the v-BWT. Equal to the maximum size of context groups in $\mathcal{M}_v$.
k_{min}	Minimum sorting depth of the v-BWT.
k_{max}	Maximum sorting depth of the v-BWT.
Q	$Q[c]$ stores the number of symbols in T^{bwt} smaller than c.
Y	Error threshold of the approximate pattern matching algorithm.
D_k	Bitvector describing the k-group boundaries in $\mathcal{M}_k$.
D_2, D_3	Bitvectors describing the 2 and 3 group boundaries in $\mathcal{M}_2$ and $\mathcal{M}_3$.
D_v	Bitvector describing the context group boundaries in $\mathcal{M}_v$.
d_v^i	Size of the context group containing row i at sorting depth v.
$\mathcal{M}$	Matrix $\mathcal{M}$ whose rows contain the cyclic rotations of T in lexicographical order.
$\mathcal{M}_k$	Matrix $\mathcal{M}$ of rotations with the rotations stably k-sorted.
$\mathcal{M}_3$	Matrix $\mathcal{M}$ of rotations with the rotations stably 3-sorted.
$\mathcal{M}_v$	Matrix $\mathcal{M}$ of rotations with the rotations so no context group is $> v$.

L	Last column of the transform matrix which corresponds to the BWT or k-BWT.
F	First column of the transform matrix.
LF	Last to first column mapping of $\mathcal{M}$ used to recover T from T^{bwt}.
LF_k	Last to first column mapping of $\mathcal{M}_k$ used to recover T from T^{kbwt}.
C_i	Context group referring to the i-th largest k long prefix in $\mathcal{M}_k$.
C^{abc}	Context group with the same $k = 3$ prefix `abc` in $\mathcal{M}_k$.
SA	Suffix Array where suffixes are fully lexicographically sorted.
SA_k	Suffix Array where suffixes are lexicographically sorted up to depth k.
$kblock$	Initial block size of the external k-BWT construction algorithm.
γ	Branching factor of the external merge construction algorithm.
I	Position of the original text in $\mathcal{M}$ and the start of the reversal algorithm.
P	Pattern of length m.
$\langle sp, ep \rangle$	Range of rows in $\mathcal{M}$ prefixed by P.
B / bv	Uncompressed bitvector of size n.
B_{rrr}	H_0 compressed bitvector of Raman et al. [2002].
K	Block size of the H_0 compressed bitvector of representation Raman et al. [2002].
b_i	Block b_i of size K bits in B_{rrr} represented as $< \kappa_i, \lambda_i >$.
κ_i, λ_i	For each b_i in B_{rrr}, κ_i represents the block class and λ_i the offset of b_i in κ_i.
C	Array in B_{rrr} storing the class κ_i types of each block.
O	Array in B_{rrr} storing the class offsets λ_i types of each block.
S	Array in B_{rrr} storing $rank$ samples Raman et al. [2002].
r	Size of a superblock in SEL-CLARK covering $\log n \log \log n$ one bits.
$long, block, mini$	In SEL-CLARK, depending on r, a superblock is represented as a $long$ block, or multiple $block$s and $mini$-blocks.
r'	Size of a $block$ in SEL-CLARK covering $r / \log r \log \log n$ one bits.
R_s, R_b	Superblock and block array of the rank structure of González et al. [2005].
HP	Hugepages support of the operating system.
RANK-V	Rank structure of Vigna [2008] using 25% overhead.
RANK-IL	Interleaved rank structure proposed in 3.2.
SEL-C	Engineered select structure proposed in 3.3.
W	Number of one bits in the superblock of SEL-C.
SEL-CLARK	Faithful implementation of constant time select of Clark [1996].
SEL-BS	Binary Search select of González et al. [2005].

SEL-BSH	Cache-friendly Binary Search select of González et al. [2005].
SEL-V9	Select method of Vigna [2008] built on top of RANK-V.
SEL-VS	Engineered select method of Vigna [2008].
TAAT	Term-at-a-time query processing.
DAAT	Document-at-a-time query processing.
D	Collection of documents comprising the collection.
d	Number of documents in the collection.
D_i	i-th document in the document collection D.
N	Number of distinct terms in the collection.
q	Search query q consisting of query terms $q_0 \dots q_j$.
$\|q\|$	Number of query terms in the query.
q_i	Individual query term of the bag-of-words query q.
$S(q, \mathcal{D}_i)$	Similarity ranking function.
f_{q_i}	Number of documents containing one or more occurrence of q_i.
F_{q_i}	Number of occurrences of q_i in the collection.
$f_{q_i,j}$	Number of occurrences of q_i in a document D_j.
BM25	Standard similarity ranking function [Robertson et al., 1994a].
k_1, b	Tuneable parameters for the **BM25** similarity metric. Usually $k_1 = 1.2$, $b = 0.75$.
ϕ	Number of relevant documents to be retrieved.
ϕ'	Larger query threshold for each query term to be retrieved.
DA	Map each SA$[i]$ to the corresponding document, DA$[i]$, the suffix SA$[i]$ occurs in.
WT_{DA}	Wavelet tree over DA.
HSV	Skeleton suffix tree-based structure of Hon et al. [2009].
g	Sample rate with which the **HSV** structure pre-stores values.

Chapter 1

Introduction

Researching an entry in an encyclopedia or the address of a restaurant was once considered a time consuming manual labor task. In contrast, these tasks today require little to no effort from an average computer user and can be performed instantaneously. This can be attributed to the availability of fast *text search* over many different data sources. Applications such as search engines, genome databases, and spam filters process large amounts of data. For example, popular social media platforms produce over 340 million messages per day.[1] Genome databases such as GenBank store 180 million sequences in 587 GB.[2] Therefore, being able to efficiently *locate* relevant information — text search — becomes especially important as manual search becomes impractical. While the way text search is used may be different for many applications, it can often be reduced to one of the core problems in computer science: *exact pattern matching*. Formally, this classic problem is defined as:

Definition 1 *[Gusfield, 1997] Given a string P of length m called the pattern and a longer string T of length n called the text, the exact pattern matching problem is to find all occurrences, if any, of P in T.*

Exact pattern matching is fundamental to many practical problems ranging from word processing to natural language processing. Nevertheless, some specific applications of pattern matching such as computational biology face a more difficult problem: the text (or genome sequence) can contain *errors*. For example, errors in the text can result from mutations in genetic code or difficulties in the process of sequencing the genome sequence. The existence of errors therefore complicates the task of exploring genome sequences using traditional pattern matching algorithms. In this context, pattern

[1] `http://blog.twitter.com/2012/03/twitter-turns-six.html`
[2] `ftp://ftp.ncbi.nih.gov/genbank/gbrel.txt`

matching *allowing errors* is an important subproblem. Formally, this is known as *approximate* pattern matching and is defined as:

Definition 2 *[Navarro, 2001] Given a text T, a pattern P, a non-negative integer Y and a distance metric $d()$, find all occurrences, if any, of P in T where the distance $d()$ between the occurrence and P is less than or equal to Y.*

Here, distance is any arbitrary metric used to numerically quantify the *similarity* between two sequences such as the *Edit Distance* [Sellers, 1980].

The concept of *similarity* is also important in the area of *Information Retrieval* (IR) where the relevance of a document is expressed using a *similarity metric*. Most IR processes solve the more abstract pattern matching problem called the *ranked document search* problem. In this context a search pattern is often referred to as a *query* consisting of one or more *query terms*. The text T is further partitioned into a set of *documents*. Unlike traditional pattern matching, documents instead of text positions are returned to the user. The documents are ranked by their *relevance* to the *information need* formulated by the user through the query [Croft et al., 2009]. In practice, the notion of relevance is "emulated" by a similarity metric. Therefore, a search returns a subset of documents in the text which are most *similar* to the search query. Formally, we define the *ranked document search* problem as:

Definition 3 *Given a query q consisting of one or more query terms q_i, a non-negative integer ϕ and a text T partitioned into d documents $\{D_1, D_2, \ldots, D_d\}$, return the top-$\phi$ documents ordered by a similarity measure $S(q, D_i)$.*

A document is considered relevant if it helps to satisfy the information need of the user. Unsurprisingly, the relevance of a document can not always be objectively assessed as it relies on the perception of the individual user. Therefore, modelling and evaluating relevance, expressed through the similarity measure used to rank documents, is a difficult problem in a continuously evolving area of research [Hawking et al., 2001]. The *effectiveness* of an algorithm solving the ranked document search problem describes the *quality* of the results returned to the user. Evaluating the effectiveness of an algorithm is non-trivial as effectiveness is generally measured by user satisfaction which itself can be subjective and difficult to measure [Al-Maskari et al., 2007].

In practice, there are two general approaches to solve the *exact*, *approximate* and *ranked document* text search problems discussed above: *online* and *offline* pattern matching. Online *exact* pattern matching algorithms preprocess the pattern P and locate all occurrences of P by performing

a complete scan over the text T. Several theoretically optimal [Knuth et al., 1977] and practically fast [Horspool, 1980] online exact pattern matching algorithms exist, and are used in popular text searching tools such as `grep`. However, online pattern matching algorithms have one major disadvantage: search time is proportional to the size of the text. In contrast, *offline* pattern matching allows searching for any pattern P in a text T using time proportional only to the size of the pattern [Weiner, 1973]. To achieve this improvement in run time performance, offline pattern matching algorithms preprocess the text to create an *index*. The index consists of auxiliary data, stored on disk or in-memory, which is used to allow fast pattern matching over the text. In this perspective there exists a classical time and space trade-off between the two approaches to pattern matching: compared to online pattern matching, offline pattern matching algorithms require additional space to reduce the time required to perform search. In this thesis we focus only on offline pattern matching and text indexing.

The most common index used in IR to solve the ranked document search problem is the *inverted index*. The inverted index consists of two main components, the *vocabulary* and the *postings lists*. During index construction each document in the text is segmented into an ordered set of terms. For English text, terms refer to the words in a document. Each unique term in the text is added to the vocabulary. For each unique term, the documents containing the term (and optionally, the positions of the term within the documents) are stored in compressed form in a *postings list*. A variety of time and space trade-offs exist in regards to storing and accessing both main components of the inverted index [Zobel and Moffat, 2006]. During query processing, the vocabulary is used as a look-up table to retrieve the postings lists of the query terms. The ranked document search problem is then answered by processing the retrieved postings lists. The processing of the retrieved lists varies depending on the query type, desired quality of the result set, storage location and compression method of the individual lists [Zobel and Moffat, 2006]. For example, storing postings lists on disk is generally coupled with compression schemes which require sequential processing, whereas postings lists stored in-memory can support efficient random access to increase query run time performance. The choices made during index construction and query time can therefore impact both efficiency and effectiveness of query processing [Zobel and Moffat, 2006]. The most prominent example of inverted index-based text search is the Google search engine. It provides search capabilities over the World Wide Web by periodically crawling public websites and creating a highly engineered inverted index [Barroso et al., 2003].

While inverted indexes are widely used in practice today, they exhibit several inherent limitations. Inverted indexes are built around the notion of *terms*. Each document in the text is *segmented*, that is partitioned, at construction time into an ordered set of terms. In practice, segmenting involves

additional term normalization techniques such as *stemming* and *stopping* [Zobel and Moffat, 2006]. Stemming refers to reducing words to their base form. For example, the words "swimming" and "swimmer" are reduced to the term "swim", which is then included in the vocabulary. Stopping refers to removing terms such as "the" or "and" from the vocabulary. Both techniques are used to increase the efficiency of the index. Stemming reduces the number of postings lists whereas stopping specifically excludes words which occur frequently but contain little semantic content. In essence, inverted indexes are carefully engineered to store and retrieve auxiliary information about terms in the text to solve the ranked document search problem efficiently. However, relying on the notion of *terms* as the basic building blocks of the index has several drawbacks. At query time, a query consisting of one or more query terms is evaluated. The posting list for each query term is retrieved using the vocabulary and processed. Only those query terms contained in the vocabulary which was created during index construction can be processed. Therefore, an inverted index can not be used to solve the exact pattern matching problem as only terms selected during the construction process are searchable via the index. For example, if stopping is used the index does not contain the term "the", while stemming reduces the occurrences of "swimmer" to the term "swim". Therefore, determining the exact positions of the pattern "the swimmer" is not possible with a traditional inverted index. A different problem with a term-based index is the non-trivial task of segmenting certain texts. For example, agglutinative languages such as German or Hungarian form new words by combining existing words. Many East Asian languages do not contain explicit word boundaries in sentences. Therefore, the segmentation of non-English text can be error-prone and complex [Nie et al., 2000]. Finally, a separate problem of inverted indexes is the requirement of document separation. Many aspects of inverted indexes are geared towards document-based collections. For example, most similarity metrics used to solve the ranked document search problem emulate "relevance" by computing statistics at a document and term level over the text. Overall many aspects of inverted indexes are engineered to efficiently solve the ranked document search problem over a collection of documents consisting of terms. Using inverted indexes over other types of text, or to solve other types of text search problems can therefore be problematic.

A second family of text indexes are based on *suffix trees*, which do not exhibit the problems described above. A suffix tree is a data structure that can solve the exact pattern matching problem over a text in theoretically optimal time [Weiner, 1973]. Conceptually, a suffix tree indexes all *suffixes* of a given text by building a trie over every suffix in the text. Text search is performed by walking the trie from the root node along matching edge labels, to a subtree where each leaf corresponds to a match of the search pattern in the text. Unlike inverted indexes, the suffix tree does not require the text to be either document- or term-based. Therefore, a suffix tree can support search for any

pattern selected during query time. Suffix trees can also be used to efficiently solve the approximate pattern matching problem [Navarro, 2001]. Unfortunately, suffix trees exhibit one major limitation: the most efficient implementation uses up to twenty times the space of the original text [Kurtz, 1999]. *Suffix arrays* can provide the same search functionality as suffix trees. Conceptually, suffix arrays map the lexicographically sorted suffixes corresponding to the leaves in the suffix tree to positions in an array. The suffix array, in conjunction with several auxiliary structures, can then be used to emulate the suffix tree [Manber and Myers, 1993]. Still, both suffix-based indexes exhibit large space requirements which make them usable for only small text collections when contrasted to the petabyte-size data sets indexed by inverted indexes.

The space requirements of many data structures can be reduced by using space-efficient — *succinct* — alternative representations. Succinct data structures use space comparable to the compressed representation of the underlying data while providing the same functionality as the equivalent uncompressed data structure. In particular, succinct representations of suffix trees and suffix arrays require space equivalent to the *compressed* representation of the indexed text while providing identical functionality. The main component of suffix-based succinct text indexes is the *Burrows-Wheeler Transform* (BWT). The transform, originally used in data compression, permutes the original text by sorting all rotations of the text in lexicographical order. Interestingly there exists a duality between the suffix array and the BWT. The duality allows *emulating* search with the suffix array while using the much smaller and more compressible BWT. In practice, during query time, this translates to significantly smaller space requirements when compared to an uncompressed suffix tree or suffix array [Ohlebusch et al., 2010]. However, one of the main problems of all succinct text indexes is the large space required during index construction. From this perspective the problem that succinct text indexes intended to address — reducing the space requirements — persists. This is especially problematic as this contradicts the main goal of succinct data structures: reducing the space requirements of the equivalent uncompressed data structures.

In this thesis we investigate two aspects related to the scalability of succinct text indexes on large data sets. The main difficulty in indexing larger data sets using succinct text indexes is *construction cost*. For fast construction, all suffix-based succinct text indexes require the *uncompressed* suffix array to be created during construction. However, more space efficient solutions exist [Ferragina et al., 2012]. While a succinct text index only requires space equivalent to the compressed representation of the data set, a regular suffix array requires up to nine times the size of the data set in RAM during construction [Puglisi et al., 2007]. For example, constructing the suffix array for a 3 GB text requires up to 27 GB of main memory. In contrast, the resulting succinct text index, depending on the compressibility of the data set, can be stored and queried using only 1.8 GB of RAM.

In response to this problem, we investigate an alternative suffix array representation which can be constructed more efficiently. Specifically we explore a suffix array representation which only *partially* sorts suffixes, up to a certain depth k, in lexicographical order. This alternative suffix array representation is equivalent to a previously unexplored variation of the BWT called the k-BWT [Schindler, 1997; Yokoo, 1999]. We show that the k-BWT can be constructed more efficiently than the BWT. Furthermore we propose a novel external memory construction algorithm for the k-BWT which outperforms state-of-the-art external suffix array construction algorithms. Next, we prove that k-BWT-based succinct text indexes can be used to provide equivalent search functionality. Additionally, we discuss benefits of the k-BWT index and compare it to traditional BWT indexes. Finally, we propose a new, *variable-depth* text transformation: the v-BWT. We show that the transform can be used in succinct text indexes, and provide applications of the transform to approximate pattern matching.

The second aspect we explore in this thesis is the engineering of succinct text indexes for large-scale data sets. We provide an extensive evaluation of the performance and resource usage of succinct text indexes on various data sets. We propose several carefully engineered, cache-efficient, data structures which form the building blocks of all succinct data structures. In addition, we provide an extensive empirical evaluation of several commonly used succinct indexes to compare our new representations. To our knowledge, our improvements result in succinct text indexes that are smaller or faster than all current state-of-the-art implementations.

Finally, we apply succinct text indexes in the context of IR. We construct indexes over data sets commonly used in IR evaluation, and compare the performance of succinct text indexes on standard IR tasks with inverted indexes. We provide the first evaluation of succinct text indexes comparing both the efficiency and effectiveness using standard IR evaluation metrics. We find that, while succinct text indexes use more space than inverted indexes, succinct text indexes can be competitive in regards to efficiency and effectiveness.

1.1 Thesis Structure and Contributions

In Chapter 2 we discuss previous work and basic concepts related to our work. We provide an overview of the notation, terminology, mathematical concepts, and the experimental setup used throughout this thesis. We describe the data sets, software, hardware, and common methodology used for the conducted experiments. We introduce the basic operations $rank$ and $select$, first on bitvectors and later on general sequences. Last we discuss text transformations, specifically the BWT, and introduce several fundamental succinct text indexes and basic concepts of document retrieval.

In Chapter 3, we describe our improvements to the basic building blocks of many succinct text in-

dexes. Our first contribution is two carefully engineered, cache-efficient data structures solving $rank$ and $select$ on uncompressed bitvectors. We further provide the first "true to theory" implementation of Clark's $select$ data structure [Clark, 1996] and compare it to our newly engineered implementation. We then propose a new $select$ algorithm for 64-bit words which is the basic building block of many $select$ data structures operating on 64-bit words. Our second contribution includes practical enhancements to compressed bitvectors [Raman et al., 2002] which result in significant performance improvements, in both compression effectiveness and run-time performance. Our final contribution is an extensive empirical evaluation of our modified data structures. We first evaluate the performance of each individual data structure by comparing it to the current state-of-the-art. We find that our uncompressed bitvector representations are faster for all data sets tested. We further find that our compressed bitvector representations provide better compression effectiveness and run-time performance, depending on the chosen parameter, than previous implementations. Finally we evaluate the effect of our improvements on succinct text indexes. We first ensure that our non-optimized implementations are competitive with commonly used test implementations. Finally we show that our improvements of the "basic building blocks" of succinct text indexes significantly improve the performance of the index in terms of both time and space. Overall, we provide representations that are either smaller or faster than all existing state-of-the-art implementations.

In Chapter 4, we provide an extensive evaluation of the k-BWT. Our first contribution in this chapter is a formal definition of the k-BWT and its auxiliary structures and algorithms. We provide an extensive evaluation of in-memory forward construction algorithms. Our second major contribution in this chapter is a fast, efficient external memory construction k-BWT algorithm which outperforms all existing suffix array construction algorithms. Next we propose a new k-BWT reversal algorithm which explicitly stores the information required to reverse the transform instead of recovering it during the reversal process. We analyse the time and space trade-offs of our new algorithm and compare it to the BWT and k-BWT reversal algorithms. We find that algorithms which perform well in theory are outperformed by more inefficient algorithms in practice. Last, we provide an extensive evaluation of the compression effectiveness of the k-BWT when used in a standard compression system.

In Chapter 5, we extend our previous work on the k-BWT. Specifically, we investigate searching in a k-BWT transformed text. The main contribution is providing a formal proof that performing backward search using the k-BWT is possible. We further provide a detailed walk-through example of our approach, discuss theoretical space bounds as well as practical improvements. Next we evaluate the practical space usage of our approach when compared to a regular inverted file-based k-gram index. Last we examine the applicability of the k-BWT to emulate a k-gram inverted index to solve the approximate pattern matching problem.

Chapter 6 further extends our work on context-bound text transformations. Instead of using a fixed threshold k, we investigate other sorting thresholds using variable sorting depths. We show that this new variable depth transform (v-BWT) is reversible and can be constructed efficiently using modified radixsort algorithms. Next we discuss applying our results of performing search in k-BWT transformed text to the v-BWT. Last we show that the transform can be used in the context of approximate pattern matching. We discuss and evaluate using the v-BWT in a k-gram index in the context of approximate pattern matching. Specifically, we show trade-offs between verification cost and index size for k-BWT and v-BWT based approximate text indexes.

As our last contribution, we apply succinct text indexes within the area of Information Retrieval in Chapter 7. We use self-indexes in the context of document retrieval to solve the ranked document search problem defined in Definition 3. Specifically, we use a hybrid self-index approach to solve a subset of important top-ϕ document retrieval problems – *bag-of-words* queries. We evaluate our approach using a comprehensive efficiency analysis comparing in-memory inverted indexes with top-ϕ self-indexing algorithms for bag-of-words queries on standard Information Retrieval text collections an order of magnitude larger than any other prior experimental study. To our knowledge, this is the first comparison of self-indexes in the context of document retrieval for standard, realistic sized text collections using a commonly used similarity metric – **BM25**. Overall we show that self-indexes can be competitive in terms of both run time efficiency and effectiveness to standard inverted index baselines. We find that space usage of character-based succinct text indexes supporting document retrieval is not competitive to term-based inverted indexes. However, self-indexes can efficiently support advanced operations such as phrase queries which are computationally expensive to support using inverted indexes.

In Chapter 8 we discuss our contributions and possible extensions to work presented in this thesis. In Section 8.1 we consider several areas of future work. We discuss problems with construction and parallelism of succinct data structures and their basic building blocks as well as extensions and applications of both context-bound text transformations discussed in this thesis. Finally we discuss self-indexes in the context of Information Retrieval. We focus on (1) extensions to efficient top-ϕ ranked retrieval and (2) using features provided by self-indexes which are computationally expensive using traditional index types. To conclude we summarize our contributions in Section 8.2.

Chapter 2

Background

Succinct text indexes are generally composed of several underlying data structures. In this chapter we describe the fundamental techniques used in succinct text indexes, providing the background and related work for the subsequent chapters of this thesis. We discuss notation used throughout this thesis in Section 2.1. We then define the two fundamental operations, $rank$ and $select$, which are used by many succinct data structures. In Section 2.2 we describe algorithms and data structures which perform both operations on computer words, uncompressed bitvectors and compressed bitvectors. General sequences are covered in Section 2.3.

The BWT is an important component in many succinct text indexes and compression systems. We introduce the BWT and other text transformations in Section 2.4.1. Finally we discuss previous work and give an overview in the area of suffix-based text indexing in Section 2.5. We discuss suffix trees and suffix arrays in Section 2.5.1. Building on these data structures we introduce the main text index used throughout this thesis: the *FM-Index*. We also briefly discuss compressed suffix tree representations in Section 2.5.3 and alternative text indexes such as the inverted index in Section 2.5.4.

One of the application areas we focus on in this thesis is document retrieval, where documents instead of text positions are returned during search. We discuss document retrieval in Section 2.6 by providing an overview of two relatively distinct areas of research. We discuss theoretical solutions to the document listing and top-ϕ most frequent document retrieval problem. The second area of document retrieval we focus on is ranked document retrieval which originated from IR. Here our focus is on practical algorithms and data structures used to increase the performance of inverted indexes solving the top-ϕ ranked document search problem. Finally we describe the data sets, software, hardware, and methodology used for experiments conducted throughout the thesis in Section 2.7.

2.1 Basic Notation

Throughout this document we use the *word RAM* model of computation in which all standard arithmetic and bit operations on word-sized operands take $\mathcal{O}(1)$ time. We always assume that a computer word w is larger than $\log n$, where n is the size of our data set. All logarithms are of base 2 unless stated otherwise.

The following notation is used to refer to a sequence X of size n: Let $X[0 \ldots n-1]$ be equal to X. The subsequence starting at position $X[i]$ of length $j-i+1$ is referred to as $X[i \ldots j]$. The symbol stored at position i is indicated by $X[i]$ or x_i. The first symbol in the sequence is x_0. The last symbol is referred to as x_{n-1}. All symbols in X are drawn from an *alphabet* Σ of size $\sigma = |\Sigma|$. The number of occurrences of symbol c in X is referred to as n_c and $H_0(X)$ refers to the zero-th order entropy of X which is defined as

$$H_0(X) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c}.$$

It represents the average number of bits required to encode a symbol by any compressor of a source that randomly produces symbols with probability n_c/n. For completeness, $0 \log 0 = 0$ is assumed in the formula above. Any such compressor requires at least $nH_0(X)$ bits to represent X. We further define $H_k(X)$ as the k-th order entropy of X. The metric, $nH_k(X)$, represents a lower bound on the number of bits required by any compressor assigning uniquely decodable fixed length codes that depend on the context of a symbol up to length k. A bit sequence (or a bitvector) B corresponds to a sequence drawn from the binary alphabet $\Sigma = \{0, 1\}$ of size $\sigma = 2$. When performing pattern matching we define a pattern P of length m which is to be searched for in a text T of size n.

2.2 Rank and Select on Bitvectors

Succinct data structures save space by emulating operations of traditional data structures using basic operations on bitvectors. Two essential functions $rank$ and $select$ used by many succinct data structures were first introduced by Jacobsen [1988, chap. 5] over a bitvector B of length n as;

$rank(B, i, c)$:	Return the number of times symbol c occurs in the prefix $B[0..i-1]$.
$select(B, i, c)$:	Return the position of the i-th occurrence of symbol c in $B[0..n-1]$.

Where $c \in \{0, 1\}$ represents the binary case. Interestingly, there exists a duality between the two

functions:

$$rank(B, select(B, i, c), c) = select(B, rank(B, i, c), c) = i.$$

In the following we will briefly discuss the basic concepts behind several data structures that implement $rank$ or $select$ efficiently.

2.2.1 Elementary Bit Operations

In practice, most implementations reduce $rank$ and $select$ to solving both operations on a single computer word. We refer to these operations as $rank_{64}$ and $select_{64}$ as most architectures today use 64-bit words. The $rank_{64}$ operation is often also referred to as *population count* or $popcnt_{64}(x)$ which has been widely studied in literature [Knuth, 2011; Warren, 2003]. Population count refers to counting the number of one bits in a given computer word x. The classical divide and conquer approach described by Knuth [2011, p. 143] is shown in Figures 2.1 and 2.2. In the first step, the word is split up into 2-bit chunks. For each chunk the number of one bits are calculated in parallel as shown in Line 2 in the pseudo code. Next, two 2-bit sums are combined into a 4-bit sum in Lines $3-4$. The code uses a combination of shift/and/additions to cleverly, for two 2-bit chunks, sum up the number of 1 bits. For example, a chunk `1011` is first right shifted and masked to become `0010`. This represents the number of bits in the left 2-bit chunk. The result of this operation is then added to `0011`, the number of 1-bits in the right 2-bit chunk. As the resulting number of 1-bits in a 4-bit chunk can never exceed 4, the addition is guaranteed to not overflow. In step 3, the four bit sums are combined into 8-bit sums using the same technique used in the previous steps. Finally, the population count of the initial word is calculated by calculating adding up the 8-bit sums as shown in Line 6. The formula calculates $x \mod 255$ which is equal to the $w_1 + w_2 \ldots + w_8$ [Knuth, 2011].

González et al. [2005] and earlier Warren [2003] find that multiple byte-wise accesses to a lookup table storing all 2^8 pre-calculated $popcnt_8(x)$ values performs better in practice than more complex bit manipulation methods such as the one described above. Recently, Suciu et al. [2011] found that new processors provide efficient hardware support for calculating population count which outperforms the lookup table methods used by González et al. [2005].

Unlike $rank_{64}$/$popcnt_{64}$, which has applications outside the field of succinct data structures, $select_{64}$ has not seen as much attention in the research community. González et al. [2005] find that performing sequential $popcnt_8()$ operations over a computer word followed by sequential bit scan performs best in practice. Vigna [2008] proposes a $select_{64}$ algorithm which builds on the divide and conquer popcount approach shown in Figure 2.2 and is faster than the sequential scan approach

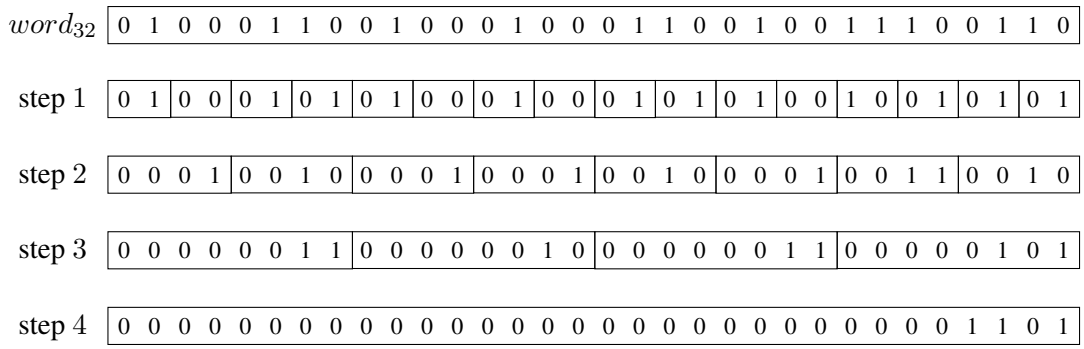

Figure 2.1: Folklore divide and conquer approach to calculate the population count of a 32*-bit word (*$word_{32}$*) using* 3 *steps to calculate the* 2*-bit sums (Step* 1*),* 4*-bit sums (Step 2),* 8*-bit sums (Step 3) and the final* 32 *bit population count (Step* 4*) described by Knuth [2011, p. 143].*

```
1 uint32_t popcount(uint64_t x) {
2     x = x -((x>>1) & 0x5555555555555555ULL);
3     x = (x & 0x3333333333333333ULL) +
4         ((x >> 2) & 0x3333333333333333ULL);
5     x = (x + (x >> 4)) & 0x0F0F0F0F0F0F0F0FULL;
6     return (0x0101010101010101ULL*x >> 56);
7 }
```

Figure 2.2: Pseudo-code of folklore divide and conquer population count calculating the 2*-bit sums in Line* 2*, the* 4 *bit sums in lines* $3-4$*, the* 8*-bit sums in line* 5 *and the final population count in line* 6 *as described by Knuth [2011].*

of González et al. [2005]. Unfortunately, there is no direct hardware support for $select_{64}$ and current implementations of $select_{64}$ are roughly six times slower than $rank_{64}$.

2.2.2 Uncompressed Rank on Bitvectors

A naive solution to answer $rank$ on an uncompressed bitvector B is to scan B in worst case linear, $\mathcal{O}(n)$, time counting bits. This approach requires no additional space. Jacobsen [1988] proposes a worst case constant time data structure to support $rank(B, i, 1)$ using $o(n)$ bits of additional space. The same structure can be used to answer $rank(B, i, 0)$ at no additional cost as $rank(B, i, 0) = i - rank(B, i, 1)$.

The data structure consists of a two-level dictionary shown in Figure 2.3. First, partition B into chunks of size $s = \log^2 n$. For each chunk i store, using $\log n$ bits, the precomputed value of

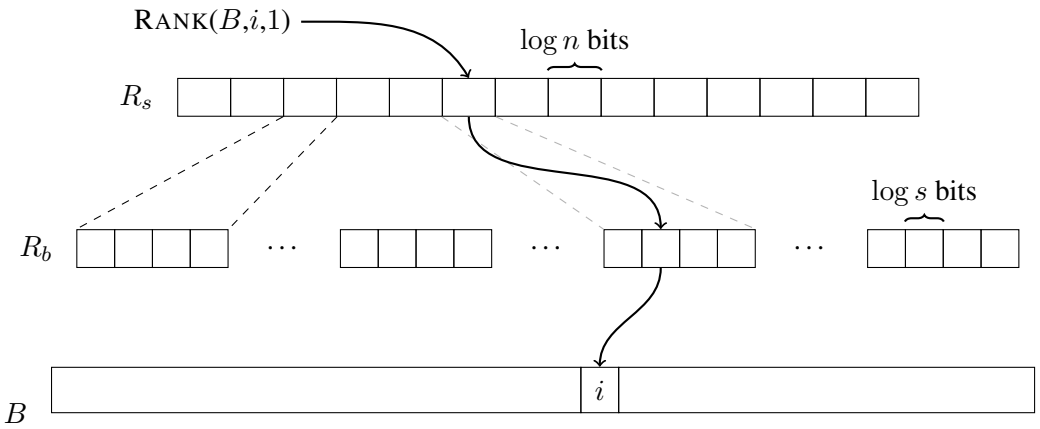

Figure 2.3: Two-level dictionary structure of Jacobsen [1988] to solve $rank$ in constant time using $o(n)$ additional space with n/s superblocks R_s of size $\log n$ and blocks R_b of size $\log s$.

$rank(B, s \cdot i, 1)$ in $R_s[i]$ at a total cost of $n/\log n$ bits. Practical implementations refer to the top level blocks (R_s) as superblocks [González et al., 2005]. Each superblock is further subdivided into blocks of size $b = \log n$. For each block $j = i \mod b$, we store the number of one bits at a cost of $\log s$ bits from the start of the corresponding superblock in $R_b[j]$. Total cost of R_b is $n \log\log n/\log n$ bits. The total cost of storing both R_s and R_b is therefore $n \log\log n/\log n + n/\log n$ bits $\in o(n)$.

To perform a $rank(B, i, 1)$ query, we first calculate the corresponding superblock $R_s[i/s]$. Next we calculate the corresponding block $R_b[i/b]$. Finally, we process the last computer word v in B containing i. We perform *population count* ($popcnt$) on v up to position i by masking the remaining bits in v. Therefore, $rank(B, i, 1)$ can be computed as $R_s[i/s] + R_b[i/b] + rank_{64}(v)$. Overall we perform a constant number of operations and thus $rank(B, i, 1)$ can be computed in $\mathcal{O}(1)$ time.

In practice, several time-space tradeoffs exist when implementing $rank$ data structures efficiently. For a 512 MB bitvector, the two-level structure described above has a 34% space overhead. González et al. [2005] evaluate different chunk sizes s and b to achieve a certain overhead space overhead. For example, for $b = 32$ and $s = b \log n$, the total overhead of the two-level structure is 38% in addition to storing B. Both Jacobsen [1988] and González et al. [2005] further propose a more space efficient one-level data structure. Instead of storing two levels, only R_s is stored. To calculate $rank(B, i, 1)$, after accessing $R_s[i/s]$, the bitvector is processed sequentially using the $popcnt$ operation starting at position $\lfloor i/s \rfloor$ up until position i. Therefore, $rank$ is calculated in $\mathcal{O}(s/w)$ time, where w is the size of a computer word. González et al. [2005] implement the two-level structure as two separate arrays R_s and R_b on top of B.

To perform one $rank$ operation, three memory accesses to different memory locations have to be performed which could potentially result in 3 TLB and cache misses. Vigna [2008] proposed *interleaving* arrays R_s and R_b to reduce the maximum number of cache misses to 2. When the superblock $R_s[i/s]$ is accessed, all second level blocks are also loaded into the cache as they are stored adjacent to the superblock. The data structure is engineered as follows; (1) Store B as an array of 64 bit integers. Additionally store an array of 64 bit integers containing the precomputed $rank$ values. Each superblock $R_s[i]$ is stored using 64 bits. (2) Using 63 bits or 8 bytes, store seven 9 bit counts representing the blocks for the corresponding superblock. Using 9 bits per block, counting the size of a superblock is fixed to 512 bits as there can be at most $2^9 - 1$ one bits in a superblock. Seven sums are sufficient to subdivide the 512 bit superblock into eight 64 bit words which can be processed efficiently using $popcnt$.

2.2.3 Uncompressed Select on Bitvectors

The $select$ operation can solved in $\mathcal{O}(\log n)$ time by performing $\log n$ $rank$ operations over B using the data structure shown in Figure 2.3. The $rank$ data structure proposed by Jacobsen [1988] can also perform $select$ in $\mathcal{O}(\log n)$ time using $\mathcal{O}(n)$ bits of space in addition to storing the bitvector. The first constant time $select$ data structure was proposed by Clark [1996, Section 2.2.2, pp. 30-32] and later published by Munro [1996]. We refer to this structure as Clark's constant time $select$ structure. The data structure consists of up to three levels similar to the $rank$ structure shown in Figure 2.3 and uses $3n/\log\log n + \sqrt{n} + \log n \log\log n \in o(n)$ bits of space. Let p_i be the i-th one-bit in B. Instead of storing values at constant intervals (the block size), the position of each $s = \log n \log\log n$-th one bit (p_s,p_{2s} and so on) in B is stored explicitly, using $\log n$ bits each, in an array $super$ using $n/\log\log n$ bits. Unlike the $rank$ data structure, the sampling intervals depend on the position of the one bits and are therefore not guaranteed to be evenly distributed over B. Depending on the distance r between two sampled positions p_{is} an $p_{(i+1)s}$, different samples are stored. If $r \geq \lg^2 n \lg\lg n$ — the one positions in the range are *sparse* — thus each one position can explicitly be stored in $long$, using $\log n$ bits for each $\log n \log\log n$ ones in $\left[p_{is}, p_{(i+1)s}\right]$, at a total cost of $r/\log\log n$ bits. Otherwise the section is *dense*, that is $r < \lg^2 n \lg\lg n$, and we further subdivide $\left[p_{is}, p_{(i+1)s}\right]$. For every $t = \log r \log\log n$-th one bit in the range, the position relative to p_{is} is stored, using $\log r$ bits each, in $block$ at a total cost of $r/\log\log n$ bits. At most $r/\log r \log\log n$ relative positions r_i are stored for each superblock. The block is further subdivided if the distance r' between to relative positions r_i and r_{i+1} is $r' \geq \log r' \log r \log\log n$. In this case, all remaining positions in $[r_i, r_{i+1}]$ are stored in the $mini$ array using $r'/\log\log n$ bits. If $r' < \log r' \log r \log\log n$, r' is asymptotically

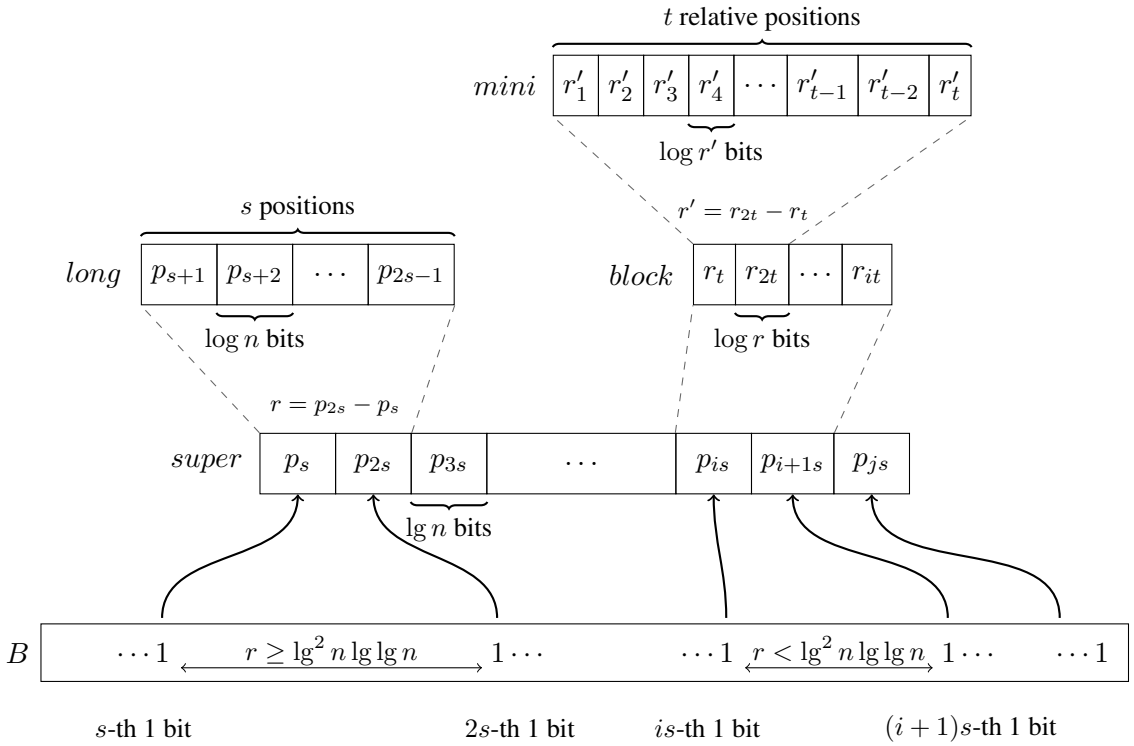

Figure 2.4: Three-level dictionary structure of Clark [1996] to solve select in constant time. Store every $s = \lg n \lg\lg n$ one bit position p_i of B in S_s. If the distance $r = p_{2s} - p_s$ is larger than $\lg^2 n \lg\lg n$, store each of the s one bit positions explicitly in S_l using $\log n$ bits each. Otherwise subdivide r and store every $t = \lg r \lg\lg n$ relative on bit position, using $\lg r$ bits in S_b Further subdivide each relative position r_t if the distance $r' = r_{2t} - r_t$ is larger than $\lg r' \lg r \lg^2 \lg n$ and store, in S_m, each relative position in $\lg r'$ bits. The total worst case cost is $3n \lg\lg n$ bits plus the cost of storing the final lookup table to process B if the position is not stored explicitly.

also smaller than $\log n$. Thus, the final position can be retrieved by processing B in constant time during query time. The structure of the data structure is shown in Figure 2.4. It is important to note that Clark's structure only provides constant time guarantees in an asymptotic sense as the range in B that has to be processed is bound above by $16(\log\log n)^4$, which can in practice be much larger than $\log n$ [Kim et al., 2005].

González et al. [2005] implement Clark's structure. However, their structure always uses 60% overhead whereas Clark's structure only uses this overhead in the worst case. They find that using binary search is faster than using Clark's structure for bitvectors of sizes up to 8 MB.

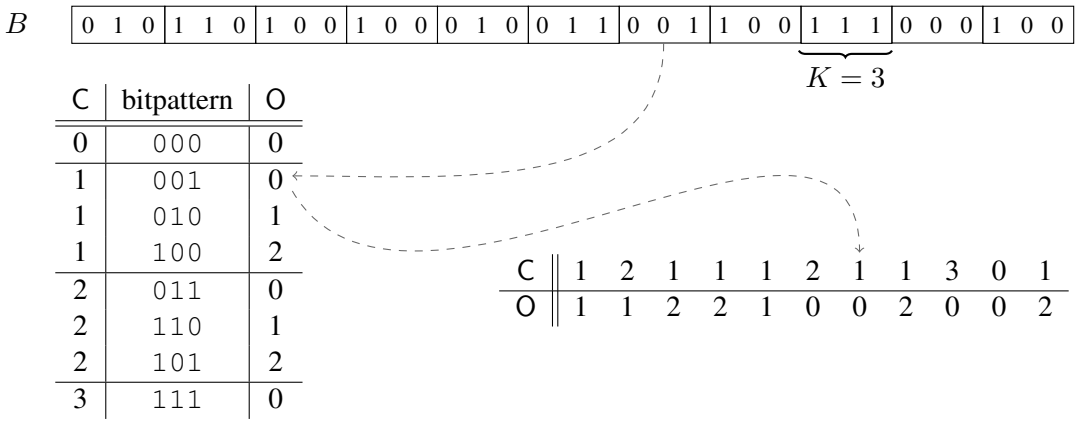

Figure 2.5: H_0 compressed bitvector representation (B_{RRR}) of Raman et al. [2002] of a given bitvector B supporting $rank$,$select$ and $access$ in $\mathcal{O}(1)$ time using quotienting to "hash" blocks of size $K = 3$ into class identifiers C *and class offsets* O*.*

2.2.4 Rank and Select on Compressed Bitvectors

Previous sections discussed solutions to support operations $rank$ and $select$ in constant time over an uncompressed bitvector B of size n while only using $o(n)$ bits of extra space. Sparse bitvectors, in which the number of ones, m, is significantly smaller than $n/2$ can be stored in compressed form while still providing the same constant time bounds on $rank$, $select$ and $access$. In this section we discuss a data structure proposed by Pagh [1999] and refined by Raman et al. [2002]. The structure is commonly known as "RRR" compressed bitvectors after the names of the authors of [Raman et al., 2002]. It provides constant time $rank$, $select$ and $access$ over binary sequences using a compressed representation requiring only $\lceil \log \binom{n}{m} \rceil + \mathcal{O}(n \log \log n / \log n)$ bits of space which is bound above by $nH_0(B) + o(n)$ bits of space using Stirling's formula. Raman et al. [2002] refer to their structure as a *fully indexable dictionary* or *FID*.

The main technique used in the "RRR" data structure is related to *quotienting* [Pagh, 1999] and *most-significant-bit bucketing* [Raman et al., 2002] commonly used in perfect hashing. All values hashed to a bucket share the same key, which from an information theoretic point of view, allows the amount of information that needs to be stored for each key inside the bucket to be reduced. For example, in Figure 2.5, all bit patterns of length 3 are sorted into buckets depending on the number of one bits in the pattern. As there are only a certain number of permutations of a bit pattern containing m ones, we can save space by storing which bit pattern is hashed to the bucket by enumerating all possible bit patterns and storing only the offset.

To achieve compression, the original bitvector B is divided into blocks of fixed length K. The information stored in each block is split into two parts: first the number κ_i of ones in the block i. There are $\binom{K}{\kappa_i}$ possible permutations of b_i containing κ_i ones. Second, storing an additional number $\lambda_i \in [0, \binom{K}{\kappa_i} - 1]$ is enough to uniquely encode and decode block b_i. For example in Figure 2.5, $\kappa_i = 2, \lambda_i = 0$ uniquely identifies block `011`. Each κ_i is stored in a vector $\mathsf{C}[0 \dots \frac{n}{K}]$ of $\lceil \log(K+1) \rceil$-bit integers requiring only $n/K\lceil \log(K+1) \rceil$ bits. For example, for $K = \log(n)/2$, the space required to store C is $\mathcal{O}(n \log\log n / \log n) \in o(n)$ bits. Compression is achieved by representing each λ_i with only $\lceil \log(\binom{K}{\kappa_i} + 1) \rceil$ bits and storing all λ_i consecutively in an offset vector O. This implies the size of each offset λ_i varies depending on the number of combinations for a specific class type κ_i. For example, in Figure 2.5, class type 1 requires 2 bits for each offset while class 0 only requires 1 bit. The total space required to store O therefore is $\sum_{i=0}^{n/K} \lceil \log(\binom{K}{\kappa_i} + 1) \rceil$ which is bound above by $nH_0(B) + \mathcal{O}(n/\log n)$ bits. To efficiently answer $access$ and $rank$ queries in $\mathcal{O}(1)$ time, pointers into the offset array (O) are stored for every $t = \log^2 n/2$ blocks at a cost of $\mathcal{O}(n/\log n)$ bits. For each block b_{jt} the element $\mathsf{S}[j]$ contains the starting position of λ_{jt} in O and the rank to the start of the block $rank(B, jtK, 1)$. The position and rank value for each element is then stored inside the larger block relative to the sampled position (the cost of one element is bound above by $\mathcal{O}(\log\log n)$) at a total cost of $\mathcal{O}(n \log\log n / \log n)$ bits. For a detailed proof of this space bound see Pagh [1999, Prop. 4]. Using similar techniques as proposed by Clark [1996] and described in Section 2.2.3, $select$ can also be performed in constant time using $\mathcal{O}(n \log\log n / \log n)$ bits of extra space [Raman et al., 2002]. Finally, the table storing the class – offset – bit sequence mapping can be stored in $K2^K + K^2$ bits which for small K is negligible. Summing up the space usage of all the parts, the overall a structure supporting $rank$,$select$ and $access$ in constant time can therefore be stored in $nH_0(B) + \mathcal{O}(n \log\log n / \log n)$ bits of space.

In practice, both $access(B, i)$ and $rank(B, i, 1)$ can be answered in time $\mathcal{O}(t)$, where t is the sample rate with which positions in O are stored explicitly [Claude and Navarro, 2008]. First we determine the block index $i' = \lfloor i/K \rfloor$ of bit i. Second, we calculate the intermediate block $\tilde{i} = \lfloor \frac{i'}{t} \rfloor t$ prior to i' which contains a pointer into O. Third, the sum Δ of the binary lengths of each λ_j ($\tilde{i} \leq j \leq i' - 1$) is calculated by sequentially scanning C, adding $\lceil \log(\binom{K}{\kappa_j} + 1) \rceil$ for each block b_j. Finally, $b_{i'}$, the block containing position i, can be reconstructed by accessing $\kappa_{i'}$ and $\lambda_{i'}$ directly as they are encoded at position $\mathsf{S}[\tilde{i}] + \Delta$ with $\lceil \log(\binom{K}{\kappa_{i'}} + 1) \rceil$ bits in O. Having recovered $b_{i'}$ from O and C, we can answer $access(B, i)$ by returning the bit at index $i \bmod K$. Operation $rank(B, i, 1)$ can be answered by adding $\mathsf{S}[\tilde{i}+1]$, the sum of values in $\mathsf{C}[t\tilde{i}..i'-1]$, and $rank(b_{i'}, i \bmod K)$. In practice $select$ is performed in $\mathcal{O}(\log n)$ time by first performing binary search over rank samples in S and then sequentially scanning the blocks between the target interval Claude and Navarro [2008]. Claude and

Navarro's implementation uses blocks of size $K = 15$ and does the decoding of $b_{i'}$ from $(\kappa_{i'}, \lambda_{i'})$ via a lookup table of size 64 kB. They experimentally show that "RRR" compressed bitvectors perform $rank$ and $select$ roughly 4 times slower than uncompressed representations discussed above. The size of their "RRR" compressed bitvectors, B_{RRR}, was consistently roughly 50% of the size of the original bitvector which however depends on the compressibility of B as the space is bound by $H_0(B)$.

Using Lookup-tables for larger K is not practical. Navarro and Providel [2012] recently proposed a solution to overcome this obstacle by removing the need for the lookup-table and spending more time on the decoding process: they encode and decode blocks in $\mathcal{O}(K)$ time on-the-fly which improves the compression effectiveness of B_{RRR} by allowing larger K to be used. Golynski et al. [2007] improve on the space bounds of [Raman et al., 2002] and provide a fully indexable dictionary using only $nH_0(B) + \mathcal{O}(n \log\log n / \log^2 n)$ bits. Pătraşcu [2008] reduces the space bound further to $nH_0(B) + \mathcal{O}(n/\log^c n)$ while providing $rank$ and $select$ in $\mathcal{O}(c)$ time. Unfortunately, to this point, these results are only theoretical and no practical implementation exists. Okanohara and Sadakane [2007] presented an implementation for very sparse bitvectors (the *sdarray*) following the idea of Elias [1974] to store monotonically increasing numbers. They provide several other bitvector representations supporting $rank$ and $select$ with different time space trade-offs [Okanohara and Sadakane, 2007].

2.3 Rank and Select on Sequences

Many succinct data structures rely on being able to perform $rank(B, i, c)$ and $select(B, i, c)$ on alphabets larger than two. The $rank$ and $select$ data structures discussed in the previous section only support both operations over binary alphabets. A *wavelet tree* can be used to support both $rank$ and $select$ operations efficiently over larger alphabets. The wavelet tree is the key component of many succinct text indexes such as the FM-Index and has been studied extensively in previous work. Here we discuss the basic concept of the wavelet trees, several key results related to space and time efficiency, alternative data structures and several advanced operations composed of multiple $rank$ and $select$ operations which provide additional functionality over a sequenced indexed by a wavelet tree.

2.3.1 Wavelet Tree Fundamentals

The wavelet tree was initially proposed by Grossi et al. [2003, Section. 4.2] as an "additional" result in a paper proposing new compressed text indexes. The basic concept is shown in Figure 2.6. A sequence T of size n over an alphabet of size $\sigma = |\Sigma|$ is decomposed into multiple binary sequences

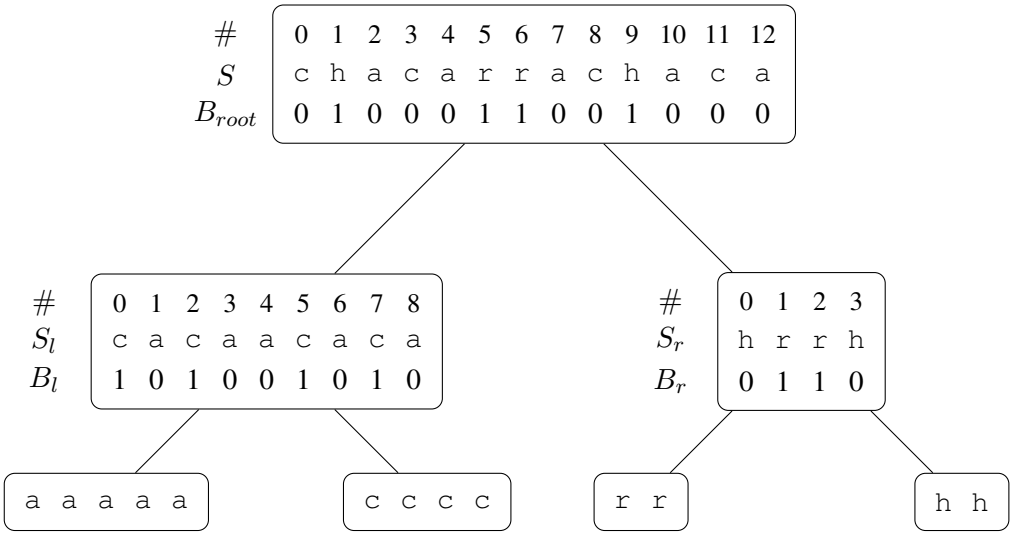

Figure 2.6: Wavelet Tree over the sequence $T =$`chacarrachaca` *with* $\Sigma = |\{a, c, h, r\}| = 4$*. The sequence is decomposed into a binary tree of height* $\log \sigma$ *by splitting the alphabet at each node into half. Symbols represented in the right subtree are marked using a bitvector at each level.*

at each node of a binary tree. At each node, the symbols associated with the node are represented with either 0 if they are smaller or equal the median symbol in the alphabet and 1 otherwise. The zero or one bits are stored in a bitvector at each node in the tree. Symbols associated with zero are mapped into the left subtree and symbols associated with one are mapped into the right subtree. For example, in Figure 2.6, the alphabet $\Sigma = \{a, c, h, r\}$ is split evenly at each level into two subsets resulting in a balanced binary wavelet tree of height $\lceil \log \sigma \rceil$. In the root node, symbols a, c are mapped to the left subtree and symbols h, r are mapped to the right subtree. This process is repeated recursively in each subtree. At the leaf level each of the σ nodes correspond to one unique symbol in the sequence. The number of bits stored in all nodes in any level of the wavelet tree is n. Therefore, at most $n\lceil \log \sigma \rceil$ bits for the binary sequences in all tree nodes plus the space required to store the tree structure are required to store the wavelet tree. We refer to the wavelet tree representation of T as $WT(T)$.

A standard, uncompressed wavelet tree supports both $rank(T, i, c)$ and $select(S, i, c)$ in $\mathcal{O}(\log \sigma)$ time. Additionally, we can retrieve i-th symbol in T, $T[i]$, also in $\mathcal{O}(\log \sigma)$ time. This operation is usually referred to as $access(T, i)$. To support $rank$ and $access$ over T we additionally build the constant time binary $rank$ structure of Jacobsen [1988] over all bit sequences at a total cost of $o(n\lceil \log \sigma \rceil)$. We can therefore perform constant time $rank$ operations over the binary sequence in each of the wavelet tree nodes.

We now briefly discuss how to perform $rank(T, 9, \mathtt{c})$ in the sequence $T =$`chacarrachaca` by

"walking" the wavelet tree shown in Figure 2.6 in $\mathcal{O}(\log \sigma)$ time. Let B_{root} be the bit sequence of the root node. We know that symbol c is associated with 0 at the root level as it is in the "lower half" of the alphabet $\Sigma = \{a, c, h, r\}$. Therefore we perform $rank(B_{root}, 9, 0)$: we count the number of zeros before position 9 which is 6. This implies that there are 6 symbols in $T[0 \dots 9]$ which are either `a` or `c` as those are the only symbols mapped to 0 in B_{root}. Next we recurse to the left subtree as we mapped symbol `c` to zero. Let B_l be the binary sequence representing the left child of the root node. We now only examine $B_l[0 \dots 5]$, which corresponds to the first 6 symbols in the node. As symbol `c` is mapped to 1 in B_l, we perform $rank(B_l, 5, 1) = 3$ to get the number of `c`'s in $T[0 \dots 9]$. We performed $\log \sigma$ constant time $rank$ operations on the binary sequences in the wavelet tree nodes at a total cost of $\mathcal{O}(\log \sigma)$.

Next we discuss $access(S, i)$. Being able to perform $access(S, i)$ implies that we can recover T from $WT(T)$ by performing $T[0] = access(S, 0)$ to $T[n-1] = access(S, n-1)$. We now show the process of recovering $T[8]$ from $WT(T)$ using the sample wavelet tree shown in Figure 2.6. First, we access $B_{root}[8] = 0$ in constant time. This implies that the symbol $T[8]$ is mapped to 0 in root of the wavelet tree. Next we perform $rank(B_{root}, 8, 0) = 6$ to determine the number of symbols in $T[0 \dots 8]$ also mapped to zero. We now recurse to the left child of the root node and access $B_l[5] = 1$ again in constant time. We access the sixth symbol in B_l as 6 symbols in $T[8]$ is the sixth symbol in $T[0 \dots 8]$ mapped to zero. As $B_l[5] = 1$, we recurse to the right sub tree of the current node to find that $T[8] =$ `c`. Overall we again performed $\log \sigma$ constant time operations at a total cost of $\mathcal{O}(\log \sigma)$.

Last we discuss $select(T, i, c)$. To support the operation in $\mathcal{O}(\log \sigma)$ time we additionally require constant time $select$ operations on binary sequences provided by the three-level structure proposed by Clark [1996] discussed in detail in Section 2.2.3. We again use the example shown in Figure 2.6 to show how to perform $select(T, 3,$ `c`$)$: Return the position of the third `c` in T. Assume we can access the leaf node corresponding to symbol c in constant time. We now perform a bottom-up traversal of the wavelet tree starting at c's leaf node in the wavelet tree. First, we know that in the parent node of c's leaf node, all occurrences of c are marked in B_l with 1 as c's leaf node is the right child of the parent node. Therefore we perform $select(B_l, 3, 1) = 5$ to determine what is the position of the third 1 in B_l. This implies that the position of the third c in T corresponds to the sixth (position 5 in B_l corresponds to the sixth element in B_l) 0 in B_{root}. We therefore perform $select(B_{root}, 6, 0) = 8$ to answer $select(T, 3,$ `c`$) = 8$ in $\mathcal{O}(\log \sigma)$ time.

In practice the wavelet tree is not stored explicitly using pointers and nodes but the bit sequences at each level are concatenated [Claude and Navarro, 2008] resulting in a pointer-less tree representation. This requires a continuous alphabet. In practice, all symbols in T are therefore remapped into a continuous range $[0 \dots \sigma - 1]$. Using a continuous alphabet, the tree shape can further be determined

implicitly during query time at the cost of additional $rank$ operations [Claude and Navarro, 2008].

2.3.2 Alternative Wavelet Tree Representations

Wavelet trees have been a very active research area since the initial paper by Grossi et al. [2003]. There exist several extensive surveys giving an exhaustive overview of wavelet trees in prior art [Grossi et al., 2011; Navarro, 2012; Christos, 2012]. Here we briefly discuss several key results related to wavelet trees and alternative representations.

Compression

The standard binary wavelet tree using the uncompressed $rank$ and $select$ support structures of Clark [1996] and Jacobsen [1988] uses $n \log \sigma + o(n \log \sigma)$ bits of space while supporting $rank$, $select$ and $access$ in $\mathcal{O}(\log \sigma)$ time. Various techniques in previous work exist to improve both the time and space bound of wavelet trees. Most of these techniques either change the *shape* of the tree or the underlying bitvector representation.

Using Huffman codes [Huffman, 1952] to encode each symbol in T instead of using fixed length codes in $[0 \dots \sigma - 1]$ results a Huffman-shaped wavelet tree. The size of the wavelet tree is therefore reduced to $n(H_0(T) + 1) + o(n(H_0(T) + 1))$ bits as the average length of each symbol is at most $H_0(T) + 1$ [Mäkinen and Navarro, 2004]. Assuming operations on symbols (for example $rank(T, 5, \texttt{f})$) are performed with the same probability as their frequency in T, a Huffman-shaped wavelet tree, on average, further provides improved query performance for all operations of $\mathcal{O}(H_0(T) + 1)$ which can however become $\mathcal{O}(\log n)$ in the worst case. In their initial paper Grossi et al. [2003] remarked that independent of the shape, compressing the bitvectors inside the wavelet tree will also result in wavelet tree being H_0 compressed. Using the H_0 compressed bitvectors discussed in Section 2.2.4 a balanced wavelet tree uses only $nH_0(T) + o(nH_0)$ bits of space while providing $\mathcal{O}(\log \sigma)$ query performance. The same space complexity can be achieved using a Huffman-shaped wavelet tree using uncompressed bitvectors. In practice, using a Huffman-shaped wavelet tree is much faster, as the average query time is decreased ($\mathcal{O}(H_0(T) + 1)$ compared to $\mathcal{O}(\log \sigma)$) and operations $rank$ and $select$ on uncompressed bitvectors is much faster than their compressed counterparts [Claude and Navarro, 2008].

Other time and space trade-offs have been proposed. Grossi et al. [2011] propose using Hu-Tucker codes [Hu and Tucker, 1971] instead of Huffman codes to create a Hu-Tucker shaped wavelet tree. Ferragina et al. [2007] propose multi-ary wavelet trees with a branching factor of z to obtain run-time complexities of $\mathcal{O}(1 + \log_z \sigma)$ at increased space complexity. Grossi et al. [2011] propose a

run-length encoded wavelet tree. Navarro et al. [2011] use the grammar compressor RePair [Larsson and Moffat, 2000] in conjunction with wavelet trees to exploit repetitions in T to achieve better space bounds. The wavelet tree is the main component of many succinct indexes where a wavelet tree over $BWT(T)$ is used to achieve higher order entropy ($H_k(T)$) space bounds. This is discussed in detail in Section 2.5.

Alternatives to Wavelet Trees

Claude and Navarro [2012a] propose the *wavelet matrix*, which reorders the bitmaps in a traditional wavelet tree to allow, in practice, faster $access$ operations while providing matching performance in both theory and practice for $rank$ and $select$ operations. *Alphabet partitioning* proposed by Barbay et al. [2010] uses a wavelet tree as part of a more complex data structure which repartitions the alphabet into classes based on the frequency of each symbol. This technique is very similar to the *quotienting* technique described in Section 2.2.4. It provides $access$ and $rank$ in $\mathcal{O}(\log\log\sigma)$ time while supporting $select$ in constant time at total space complexity of $nH_0(T)+o(n)(H_0(T)+1)$ bits. Golynski et al. [2006] propose two data structures for large alphabets that do not use wavelet trees: The first structure supports only $rank$ and $select$ in $\mathcal{O}(\log\log\sigma)$ time at a cost of $nH_0(T) + \mathcal{O}(n)$ bits. The second structure supports $rank$ and $access$ in $\mathcal{O}(\log\log\sigma)$ time and $select$ in $\mathcal{O}(1)$ time at a cost of $n\log\sigma + o(n\log\sigma)$ bits. Claude and Navarro [2008] evaluate the structures of Golynski et al. and find that they are competitive for large alphabets but use up to twice the space of the most efficient wavelet tree representation.

2.3.3 Advanced Operations on Wavelet Trees

Three advanced operations – consisting of multiple $rank$ calls within a wavelet tree – provide additional functionality when using a wavelet tree over a sequence of numbers instead of text symbols.

Quantile Queries

Until recently wavelet trees have only been used to perform $rank$ operations on text characters. Gagie et al. [2009] proposed *range quantile queries* (RQQ) over balanced wavelet trees. RQQ return, for a given rank, the number with that rank in a given sublist $T[i\ldots j]$ in $\mathcal{O}(\log\sigma)$ time. For example, the query $RQQ(T[i,j], \frac{j-i+1}{2})$ would return the median in the range $[i,j]$ of sequence T. Range Quantile Queries therefore allow access to any position in a subrange $[i,j]$ of T as if $T[i,j]$ were in sorted order. Figure 2.7 shows the balanced wavelet tree over sequence $T =$`7012271385649` for

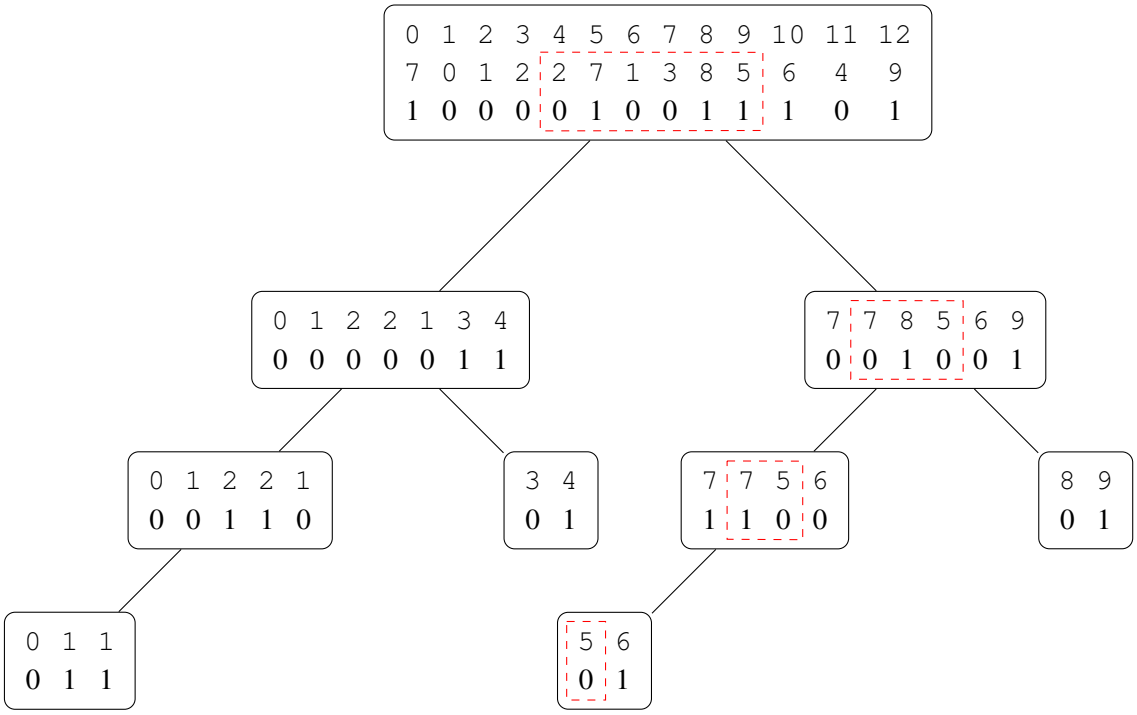

Figure 2.7: Range Quantile Queries (RQQ) on balanced Wavelet Trees over the sequence $T =$`7012271385649` *in the range* $T[4, 9]$ *for the rank* 3 *which represents the item at position* 4 *in the sorted representation of* $T[4, 9]$.

the quantile query $RQQ(T[4, 9], 3)$ in detail. In this example, we are interested in the value of the item $T[8]$ as if $T[4, 9]$ were sorted.

Given a range $T[i, j]$ and a balanced wavelet tree over T, we determine the value of "rank" r as follows. Using two binary *rank* queries at each level we determine how many 0 bits are present in the current range. For example, in Figure 2.7, there are three zero bits in the root level of the wavelet tree corresponding to range $T[4, 9]$. Let $r = 3$ be the desired "rank" for our example, which corresponds to the fourth position, in the sorted representation of $T[4, 9]$. As there are only 3 zero bits in the root level corresponding to $T[4, 9]$, we can infer that the number we are looking for is represented by one in the root level. So, recurse to right subtree, map the range, and perform the same comparison until we reach a leaf node. Note that we subtract the number of zeros we have seen from r before recursing. Overall we perform $\mathcal{O}(\log \sigma)$ binary constant time *rank* operations to answer the quantile query.

Gagie et al. [2009] suggest to use quantile queries to solve the *Document Listing Problem* first discussed by Muthukrishnan [2002]. Another interesting property of quantile queries is the fact that

the RQQ operation, without extra cost, also returns the frequency of the item. Therefore, one can, using multiple RQQ queries, traverse the sub-array $T[i, j]$ in sorted order.

Top-ϕ Most Frequent Items

Culpepper et al. [2010] propose two algorithms to return the top-ϕ most frequent items in multiple given sub-ranges $T[i, j]$,$T[f, j] \ldots, T[a, b]$. They show that both algorithms can be competitive with inverted index-based top-ϕ retrieval systems when solving the top-ϕ *Document Listing Problem* (see Section 2.6.2).

The first algorithm, top-ϕ **GREEDY** retrieval, uses a priority queue to traverse the wavelet tree in a greedy fashion by mapping all ranges $[i, j] \ldots [a, b]$ depending on the size of the remaining range. As ranges are mapped to the different sub-trees in the wavelet tree, they become smaller. If only the largest ranges are being processed, it is guaranteed that the first ϕ leaves visited will correspond to the ϕ most frequent items in all ranges. However, in the worst case the greedy approach will explore the complete tree similar to a depth-first traversal.

The second algorithm, **QUANTILE** probing, exploits properties of quantile queries to return the top-ϕ most frequent items in a range $T[i, j]$. The main idea behind the algorithm is as follows. If $S[1..m]$ is a sorted array of numbers and a number occurs more than $m/2$ times, it has to be stored in position $S[m/2]$. The same logic applies for an item that occurs at least $m/4$ times: an item occurring at least $m/4$ times has to be stored at position $S[m/4]$, $S[m/2]$, $S[3m/4]$. Interestingly, for an unsorted array T, we can query positions $m/2$ or $m/4$ as if T was sorted using range quantile queries in $\mathcal{O}(\log \sigma)$ time. Therefore, quantile probing retrieves the top-ϕ most frequent documents by repeatedly issuing RQQ queries within a range $T[i, j]$ until enough positions are queried to guarantee the top-ϕ most frequent items have been "seen". In their experimental evaluation Culpepper et al. [2010] show that greedy retrieval outperforms quantile probing while being significantly faster than other document listing solutions.

Set Intersection

Gagie et al. [2012c] propose several advanced Information Retrieval algorithms based on wavelet trees. They show how to perform *set intersection* using a wavelet tree using two algorithms. The first algorithm, similar to greedy top-ϕ retrieval, maps the ranges to intersect from the root node to all sub-trees until a leaf node is reached. The second algorithm uses the wavelet tree to emulate finger search within the ranges as if they were sorted. They show that their algorithms approaches the lower bound of *alternation*, which measures the number of times *switching* occurs when obtaining the union of

two sorted sequences [Barbay and Kenyon, 2002].

2.4 Text Transformations

Text transformations are often used to provide better compression in storage systems as well as text indexes. The Burrows-Wheeler Transform is one of the main transform used in many compression systems and succinct text indexes. The transform was initially proposed by Burrows and Wheeler as part of a compression system which utilizes the fact that the transform is reversible and at the same time makes the input text more compressible. In this section we provide an overview of the transform, the inverse transform, a more in depth analysis of several important properties of the transform as well as an overview similar text transformations.

2.4.1 The Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) was published in 1994 by Michael Burrows and David Wheeler as part of an alternative compression system to LZ compression [Burrows and Wheeler, 1994]. The transform itself was discovered by David Wheeler in 1983 but was not published until 10 years later [Burrows and Wheeler, 1994]. Unlike LZ compression which sequentially processes a given input, a BWT-based compression system processes the input in blocks. Each input block T of size n is processed as follows. First, all cyclic rotations of the input block are stored in a conceptual matrix $\mathcal{M}$ of size $n \times n$ as shown in Figure 2.8. Next, all rotations are sorted lexicographically to produce a row-ordered matrix $\mathcal{M}$. The BWT of the input block can then be retrieved from the last column of $\mathcal{M}$. Instead of sorting $\mathcal{M}$ at the worst case cost of $\mathcal{O}(n^2)$ time and space, Burrows and Wheeler further show that instead, the sorting step can be reduced to sorting all suffixes of a modified input string T', where a symbol lexicographically smaller than all symbols occurring in T is appended to T. This ensures that all suffixes are unique and no two suffixes are equal. In previous work, this symbol is often depicted as `'$'`. To sort the input suffixes, Burrows and Wheeler build and traverse a suffix tree in linear time which is not applicable in practice due to large constant factors. Instead constructing a suffix array can be used to sort the suffixes in linear time using roughly $9n$ space.

The BWT can be reversed in linear time without the need to store any additional information. Conceptually, this is done by partially reconstructing $\mathcal{M}$ from T^{bwt}. First, the first column in $\mathcal{M}$ is recovered by performing a counting sort on T^{bwt} in linear time. Second, an "LF"-mapping which maps the symbols of T^{bwt} in the last column (L) to their corresponding positions in the first column (F) is created in linear time. Finally, using this mapping and the position of "$" the T' can be reconstructed in reverse order in linear time from T^{bwt} as follows. Consider the example transform

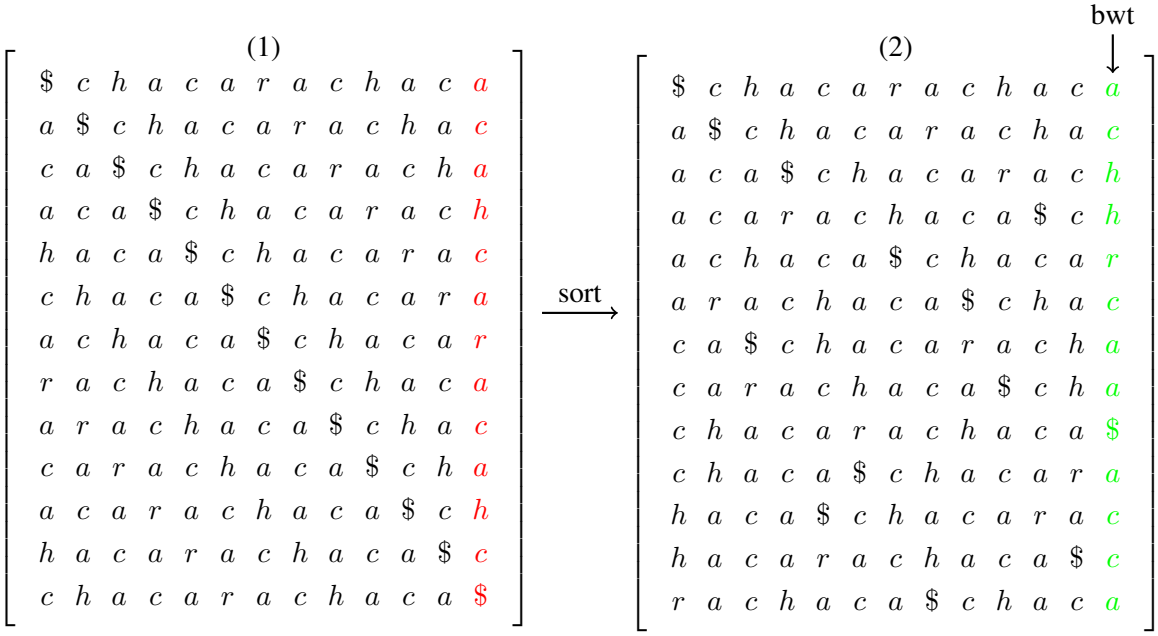

Figure 2.8: Forward BWT *of text* T =chacarachaca$ *in two steps. In the first step (1), all rotations of the input text are stored in the rows of matrix* $\mathcal{M}$*, which are then sorted in step (2) in lexicographical order. The Burrows-Wheeler transformation of the input text* T *can then be obtained from the last column of the sorted matrix* $\mathcal{M}$ *as* T^{bwt} =achhrcaa$acca.

in Figure 2.8. The position of "$" is in row 8. Thus symbol "$" is recovered and the LF mapping is used to jump to row $LF[8] = 0$. To process the current row 0, symbol 0 is prepended to the output $T^{bwt}[0] =$ 'a' and next row $LF[0] = 6$ is processed. Overall, n jumps are performed to recover T' =chacarachaca$ in reverse order from T^{bwt} =achhrcaa$acca in $\mathcal{O}(n)$ time without the need to store any additional information. The $j = LF[i]$ mapping can be computed on-the-fly using the formula:

$$j = Q[c] + rank(T^{bwt}, c, i).$$

where $c = T^{bwt}[i]$ and $Q[c]$ corresponds to starting position of symbols c in F. Using a wavelet tree this can be computed in $\mathcal{O}(\log \sigma)$ time. The process is visualized in Figure 2.9. The string T is recovered from T^{bwt} =achhrcaa$acca as follows. Initially set $s = I = 8$, the position of '$' in T^{bwt} and thus the position of the original text T in $\mathcal{M}$. Thus $T[n-1] = T^{bwt}[s]$. Next the preceding row in $\mathcal{M}$ is determined by performing $s = LF[s] = 0$. Therefore $T[n-2] = T^{bwt}[0]$ is recovered. This process is performed until all n symbols of T are recovered.

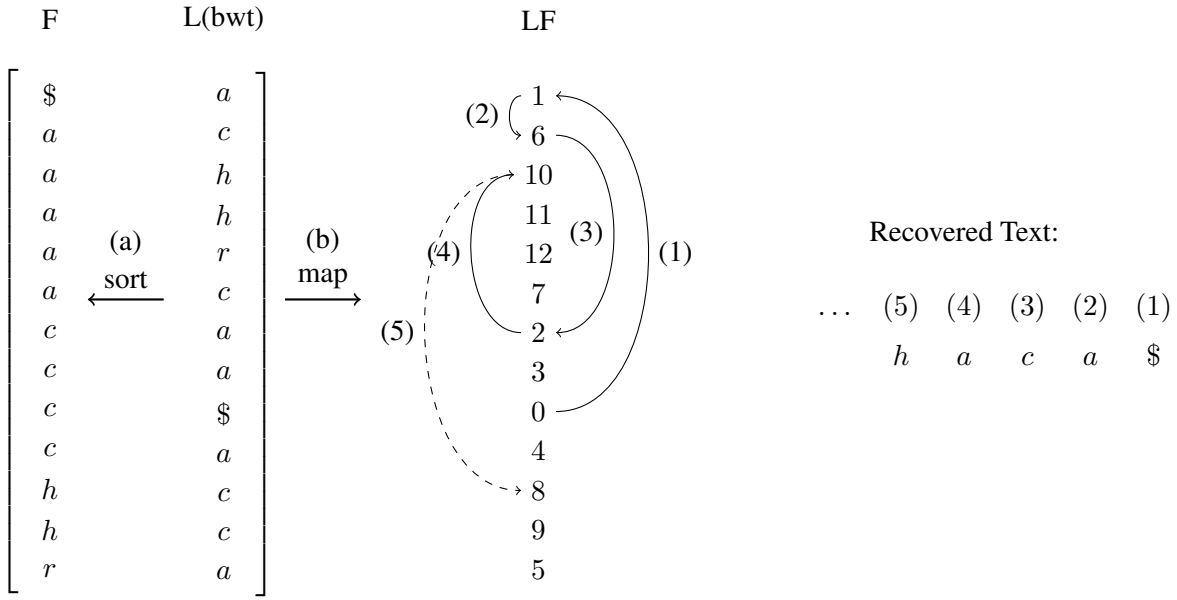

Figure 2.9: Reverse BWT *recovers the original text* T =chacarachaca$ *in three steps from* T^{bwt} =achhrcaa$acca. *In the first step (a) the first column of* $\mathcal{M}$ *is recovered by sorting* T^{bwt}. *In the second step (b), the mapping between the positions of the symbols in the last column* L*, and the first column* F *is created. Last, the* LF*-mapping is used to recover* T *in reverse order from* T^{bwt} *by jumping within the* LF*-mapping starting at position* I *of "$" in* T^{bwt}.

Burrows and Wheeler initially intended their transform to be used to improve the compression effectiveness of a given text by permuting the text to group symbols with a similar context together. Each row in $\mathcal{M}$ represents one position in the text. The last column is the transform output. Each row is a cyclic rotation of the initial text. Therefore, sorting each row lexicographically groups symbols in the BWT with the same context together. For example, symbol "h" is always preceded by symbol "c" in T. Therefore, they are grouped together in T^{bwt}. For English text, grouping together symbols with similar context tends to generate long runs of identical symbols in T^{bwt}. These runs make T^{bwt} more compressible, which can then exploited in subsequent compression steps. However, over time many other applications and improvements of the BWT have been discovered which we will elucidate below.

The most commonly used BWT compressor, `bzip2`, was originally developed by Julian Seward. [1] The compressor splits the input text into blocks of sizes 100 kB (-1) up to 900 kB (-9) to achieve various compression ratios and uses, as suggested by Burrows and Wheeler, Move-To-Front and Huffman coding. Seward subsequently proposed several improvements to the implementation of

[1] available at `bzip2.org`

the forward and reverse transform [Seward, 2000; 2001]. One of the main problems with BWT inversion is the non sequential "jumping" required when processing the LF-mapping. This causes cache misses as pointed out by Seward [2001]. Recently, Kärkkäinen and Puglisi [2011a] and Kärkkäinen et al. [2012] provide cache efficient BWT reversal algorithms which use *superalphabets* (combining two symbols into one) and parallel recovery to improve the recovery process by up to a factor of 4. A second problem of BWT reversal is space usage. The LF-mapping uses $n \log n$ bits of space which is one of the reasons `bzip2` only processes small blocks of the text at a time. Recently, Ferragina et al. [2012] show how to perform BWT inversion using sequential scans in external memory. Several "medium" space inversion algorithms have further been proposed by Kärkkäinen and Puglisi [2010].

The forward BWT transform can be reduced to suffix sorting. Therefore, recent advances in fast, practical suffix array construction can also speed up the forward BWT transform [Maniscalco and Puglisi, 2006; Puglisi et al., 2007]. Kärkkäinen [2007] propose a BWT construction algorithm using only $2n$ space which uses block wise suffix array construction and merging. Ferragina et al. [2012] show how to construct the BWT in external memory. Sirén [2009] propose a compressed suffix array construction algorithm which can be used to construct the BWT in small space and in parallel. Bauer et al. [2011] show how to construct the BWT for a collection of strings, a common scenario in DNA sequencing, in small space.

2.4.2 Applications of the Burrows-Wheeler Transform

The Burrows-Wheeler Transform has many applications in text processing and storage. Here we focus on the three common applications of the BWT.

Compression

Burrows and Wheeler initially proposed their transform to be used as the first step in a compression system [Burrows and Wheeler, 1994]. Seward provides `bzip2`, which is the most popular BWT-based compressor. All major operating systems and compression tools can decompress `bzip2` compressed files. Most BWT-based compression systems consist of a sequence of processing steps as shown in Figure 2.10. First the text is transformed using the BWT. Next the runs in the transformed output are used to skew the alphabet using a *symbol ranking* algorithm which is then either *run-length encoded* or directly compressed using an *entropy coder* such as Huffman.

Many "post-BWT" algorithms have been proposed to increase the effectiveness and efficiency of BWT based compressors. The most common second stage algorithm is move-to-front encoding proposed by Bentley et al. [1986]. Gagie and Manzini [2007] show that commonly used move-

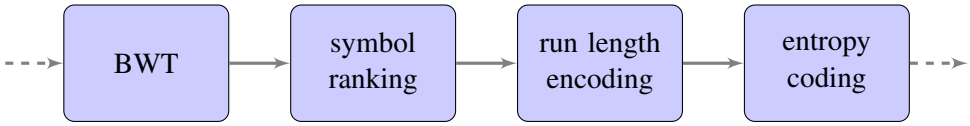

Figure 2.10: BWT*-based compression systems with the* BWT *as the first step followed by a second step a symbol ranking algorithm to skew the alphabet with optional run length encoding. The transformed and processed text is encoded using an entropy coder such as Huffman to produce the compressed output.*

to-front encoding is not optimal. Fenwick [1996] evaluates the compression effectiveness of BWT-based compressors on a standardized compression test corpus. Abel [2010] evaluates several second stage BWT post-processing algorithms. Deorowicz [2002] proposes weighted frequency counting in combination with a higher-order arithmetic coder which to our knowledge results in the most effective BWT-based compressor. Most "post-BWT" algorithms try to exploit the locality of BWT symbol runs to skew the alphabet of the string to be compressed by the entropy encoder. This process is related to the list-update problem which has been studied in both theory and practice in relation to memory paging and scheduling [Sleator and Tarjan, 1985; Bachrach and El-Yaniv, 1997; Albers and Lauer, 2008]. Transforming and compressing words instead of characters has further been studied by Moffat and Isal [2005].

Manzini [2001] shows that BWT-based compressor can achieve the k-th order entropy (H_k) within a constant factor $(5 + \epsilon)$ for any input string. Kaplan et al. [2007] provide a simpler analysis while providing lower bounds for different BWT-based compression systems. Ferragina et al. [2005] propose *Compression Boosting*, which uses the BWT to optimally partition any input text into blocks which can then be compressed more effectively using an arbitrary compressor such as LZ77.

Text Indexing

Recently, the BWT has become an integral part in many text indexing data structures which allow searching in a compressed representation of the input text without the need to decompress the index. These text indexes are described in more detail in Section 2.5.

Sequence Alignment

The BWT has found many applications in Bioinformatics. Langmead et al. [2009] propose the widely used sequence alignment tool "Bowtie" which could align 25 million short DNA sequence reads per "CPU hour" in 2009. The tool is built on top of a succinct text index proposed by Ferragina and

Manzini [2000] which uses the BWT as a key component. Li and Durbin [2009] propose the "BWA" tool which, similar to "Bowtie", uses a BWT-based text index to provide sequence alignment up to 10 times faster than existing tools at the time. Li and Durbin [2010] combine a BWT-based index with a Smith-Waterman dynamic programming algorithm to allow efficient long sequence read alignment. Many more tools like SOAP2/WCD/WCD-Express exist which take advantage of the duality between the BWT and suffix arrays to provide fast, space efficient sequence alignment tools [Li et al., 2009; Hazelhurst and Liptk, 2011]

2.4.3 Context-Bound Burrows-Wheeler Transform

When constructing the BWT via suffix array construction, one of the main problems is the worst case $\mathcal{O}(n)$ cost of a single suffix comparison. Independently, Schindler [1997] and Yokoo [1999] propose to limit the number of characters to be compared during a single suffix comparison to at most k. This implies that the matrix $\mathcal{M}$ is only sorted up to depth k. We refer to the partially sorted matrix as $\mathcal{M}_k$. The context based on which the symbols in the transform are ordered by is therefore also bound by k. The transform is commonly referred to as the Context-Bound Burrows-Wheeler Transform or k-BWT.

The difference between the k-BWT output and the full BWT output is minor. The only difference between both transforms occurs in context groups larger than one. If all context groups are of size one, the output of the k-BWT is equal to that of the BWT. This is only guaranteed for $k = n$. However, even for small k of around 10, the output is very similar for most input texts which results in similar compression effectiveness when replacing the BWT with the k-BWT even for small k [Schindler, 1997]. We explore the k-BWT in more detail in Chapters 4 and 5.

2.4.4 Alternative Text Transformations

Several other text transformations have been proposed for specific use cases. We briefly discuss these transformations and how they relate to the full BWT.

RadixZip

Vo and Manku [2007] propose *RadixZip*, a transform used for permuting column based data such as database tables or log files. Similar to the BWT, data is grouped together if they have similar context. The context in *RadixZip* is restricted to the previous columns which can improve the compression effectiveness over the BWT for highly correlated token streams.

XBW

Ferragina et al. [2006] propose the XBW, a BWT like transform for compressing (and searching) XML data. Similar to *RadixZip*, the data is sorted based on the context contained in the hierarchical tree structure. The transformed text can be used to answer certain XQuery queries efficiently.

Generalized Radix Permute Transform

Inagaki et al. [2009] propose the Generalized Radix Permute transform (GRP) which generalizes the k-BWT, *RadixZip* and the full BWT via two parameters l and d, where $l = 1$ and $d = 1$ represents the full BWT. The k-BWT is represented as $l = 1$ and $d = k$. *RadixZip* is represented as $d = 0$. Yokoo [2010] additionally provide a linear time inverse GRT transform.

Geometric Burrows-Wheeler Transform

Chien et al. [2001] propose the Geometric Burrows-Wheeler Transform (GBWT). The transform converts a given text into a sets of points on a two dimensional plane. The authors show that queries on the converted point set can be done more efficiently in external memory. They further show a connection between range searching in two dimensional space and text indexing. In addition, the authors propose a reverse transform, *Points2Text*, which transforms a set of points into a text string.

Dynamic Burrows-Wheeler Transform

Salson et al. [2008] propose the Dynamic BurrowsWheeler Transform. They show how to insert, delete and substitute symbols and substrings in a BWT transformed text by reordering the output. In their experiments they show that reordering can be cheaper than rebuilding the complete BWT from scratch up to a certain number of insertions and deletions.

2.5 Succinct Text Indexes

Succinct text indexes have become a large part of Stringology research in recent years. Suffix arrays and suffix trees have numerous applications in the area of Bioinformatics and provide the foundations of most succinct text indexes today. Here we give a brief overview of the main ideas used in succinct text indexes today. First we briefly review suffix trees and arrays. Next we introduce their compressed equivalent, the compressed suffix array and suffix tree. Last we discuss alternative succinct text indexing techniques and succinct text indexes in practice.

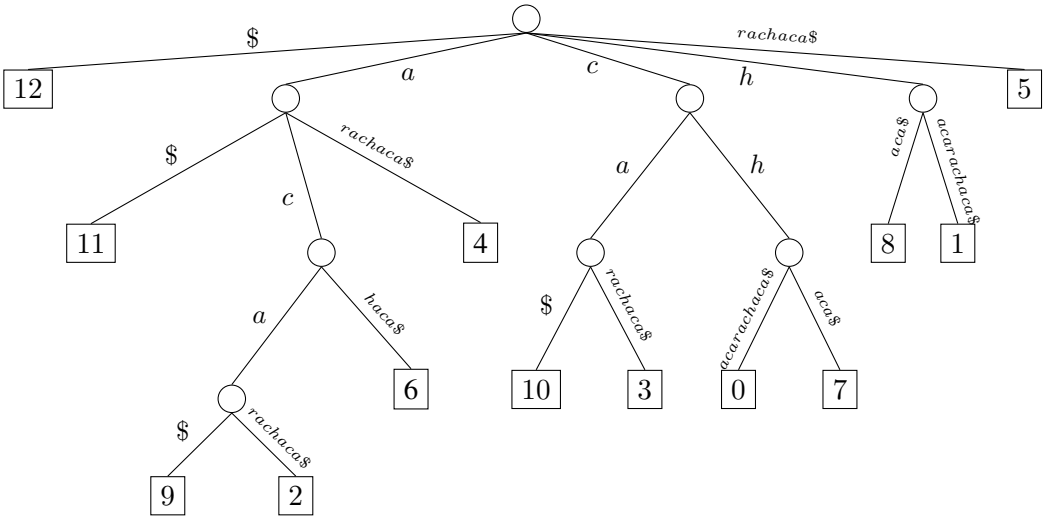

Figure 2.11: Suffix tree of text `chacarachaca$` *with leaf notes corresponding to the equivalent suffix array ordering.*

2.5.1 Suffix Arrays and Suffix Trees

Suffix Trees and Suffix Arrays are the two classical indexing data structures often discussed in textbooks. Suffix Trees are based on the notion of a *trie*. A trie can be seen as a prefix tree, where each leaf node x_i in conjunction with all edges on the path from the root node to x_i – the prefix – represent a given element in the trie [Knuth, 1998]. Each edge represents one symbol in the prefix of the string represented by node x_i. The leaf nodes of a trie represent the elements stored in the trie, whereas all internal nodes represent prefixes of these elements. Given a text T, a *suffix trie* represents all suffixes in T as a trie structure where each leaf node represent a suffix starting at a specific position in T. Weiner [1973] first showed how to use such a trie structure (in his paper called a "bi-Tree") to perform pattern matching for a pattern P of length m in time linear to the size of the pattern. Starting from the root node, we can walk the suffix trie in $\mathcal{O}(m)$ time to arrive at a node representing all suffixes in T prefixed by P. A *suffix tree* over a text T extends the idea of a suffix trie by merging all nodes in the trie which have only one child. This implies that edge labels can now be multiple characters long. A similar idea was used in the PATRICIA tree proposed by Morrison [1968]. Suffix trees are also called compact prefix trees or radix trees. Figure 2.11 shows the suffix tree over the text `chacarachaca$`. The leaf nodes show the positions of each suffix in T. The concatenated edge labels represent the suffix at each leaf node. Unlike suffix tries, *suffix trees* are guaranteed to only have $\mathcal{O}(n)$ nodes and can be constructed efficiently in $\mathcal{O}(n)$ time [Weiner, 1973; Ukkonen, 1995].

Similar to pattern matching in tries, we walk the tree starting from the root node. However, for

a pattern P, we are not guaranteed to stop at a node in the suffix tree as transitioning between nodes can result in multiple character comparisons per transition. For example, searching for pattern $P = acarad$ in the suffix tree shown in Figure 2.11, we first transition from the root node using symbol `a`. Next we transition using symbol `c`. Next we transition using `ra`. Our comparison stops during the transition with a mismatch at position $P[5]$ which implies that P does not occur in T. Overall matching can still be performed in $\mathcal{O}(m)$ time. Unfortunately, suffix trees are generally considered impractical due to their large space requirements in practice. The most efficient implementations require at least twenty times the space of the original text [Kurtz, 1999].

Manber and Myers [1993] proposed *suffix arrays* as a simplification to suffix trees. A suffix array (SA) over a text T of length n consists of a permutation of the positions of all suffixes in T. The permutation is obtained by sorting all suffixes in T in lexicographically increasing order. The same order can be obtained by traversing the suffix tree leaves from left to right in Figure 2.11. Therefore, for our sample string `chacarachaca$` the suffix array consists of $\text{SA} = [12, 11, 9, 2, 6, 4, 10, 3, 0, 7, 8, 1, 5]$. Storing the suffix array takes $n \log n$ bits of space, which when compared to the most efficient suffix tree implementation is only four times (for 32 bit or eight times for 64 bit) larger than the original text. Suffix arrays can be built efficiently in theory and practice. In 2003 several linear time suffix array construction algorithms were proposed [Ko and Aluru, 2005; Kärkkäinen and Sanders, 2003]. In practice, the fastest algorithms have a worst case complexity of $\mathcal{O}(n^2 \log n)$ and generally use a technique called induced suffix sorting [Itoh and Tanaka, 1999; Nong et al., 2011]. Recently, Nong et al. [2011] proposed a worst case linear time suffix array construction algorithm which also uses induced suffix sorting while still providing good theoretical bounds. Unlike all other proposed linear time suffix array construction algorithms, Mori [2012] show that the algorithm proposed by Nong et al. [2011] is also competitive in practice. Overall, suffix arrays can be constructed roughly six times faster than suffix trees using less space [Puglisi et al., 2007].

Searching for a pattern P of length m in T using a suffix array is more complicated than in a suffix tree. The simplest approach performs two binary searches over SA, at each step performing a $\mathcal{O}(m)$ string comparison between the current position in the suffix array $T[\text{SA}[i]]$ and P. The two binary searches determine a range $\text{SA}[sp, ep]$, where all suffixes of T are prefixed by P. Counting the number of occurrences of P in T can therefore be done in $\mathcal{O}(m \log n)$ time. This bound can be reduced to $\mathcal{O}(m + \log n)$ using the *Longest Common Prefix* (lcp) array [Manber and Myers, 1993]. The lcp is commonly defined as follows. Let $lcp(X, Y)$ be the length of the common prefix of two strings X and Y. The lcp array is then defined as

$$lcp[i] = lcp(T[\text{SA}[i-1]], T[\text{SA}[i]]).$$

for $i \leq 1 < n$ and $lcp[0] = -1$. The lcp array can also be constructed efficiently in theory and practice. Kasai et al. [2001] propose a linear time construction algorithm. Gog and Ohlebusch [2011] propose the currently fastest practical algorithm which however has a worst case complexity of $\mathcal{O}(n^2)$. The lcp array can be used to avoid performing $\mathcal{O}(m)$ character comparisons at each step of the binary search process as we can determine the length of the matching prefix of the pattern and the suffixes within the range of the binary search process. This reduces the search time to $\mathcal{O}(m + \log n)$ [Manber and Myers, 1993]. Every algorithm using a suffix tree can be implemented with the same asymptotic complexity using a suffix array with the help of the lcp array and other axillary data structures [Abouelhoda et al., 2004].

Interestingly, there exists a duality between the suffix array and the Burrows-Wheeler Transform (BWT) discussed in Section 2.4.1. This duality is the basis of many succinct text indexes and is defined as

$$T^{bwt}[i] = T[\text{SA}[i] - 1 \mod n].$$

Therefore, the BWT can be constructed efficiently by constructing the suffix array over T and applying the formula above to obtain T^{bwt} from T and SA.

Many succinct text indexes are measured by their ability to perform certain operations efficiently. Generally the following three operations are supported over a text T of length n given a pattern P of length m;

`count`(P, m):	Return the number of times pattern P occurs in $T[0..n-1]$.
`locate`(P, m):	Return all positions of pattern P in $T[0..n-1]$.
`extract`(i, j):	Extract $T[i..j]$ of length $l = j - i + 1$ from the index.

An uncompressed suffix array in conjunction with the lcp array can solve `count` in $\mathcal{O}(m+\log n)$ time, `extract` in $\mathcal{O}(l)$ time by accessing $T[i..j]$ directly and `locate` in $\mathcal{O}(m + \log n + occ)$ time where occ is the number of occurrences of P in T which also determines the size of the range $\langle sp, ep \rangle$ in SA.

2.5.2 Compressed Suffix Arrays

The functionality of suffix arrays can be provided in compressed space. We focus on two main index types: the FM-Index proposed by Ferragina and Manzini [2000] and the compressed suffix array of Sadakane [2003]. We realize there have been many improvements to these basic index types as succinct indexing has been a very active field of research over the past decade. However, many of

these indexes directly build on these two ground breaking indexing data structures which are still used today.

FM-Index

Ferragina and Manzini [2000] proposed what is now called the FM-Index. an *opportunistic* index data structure whose space usage depends on the compressibility of the text T to be indexed. Today, these data structures are commonly referred to as compressed text indexes. The space usage of compressed data structure is usually defined as a function of the compressibility (H_0 or H_k) of the text. If the text to be indexed is more compressible, the compressed text indexes requires less space. They exploit the duality of the BWT and the suffix array described above by proposing a new search algorithm: *backward search*. Backward search uses the BWT to emulate searching in a suffix array. That is, for a given pattern P of length m, the range $\langle sp, ep\rangle$ in the SA of T is determined by performing $rank$ operations over the BWT. An example of backward search is shown in Figure 2.12 for the pattern $P = cha$ and our sample text $T =$`chacarachaca$`. Backward search works as follows. A row i in $\mathcal{M}$ represents the suffix array position $\text{SA}[i]$ of a suffix $S\$$. If $T^{bwt}[i] = c$, we know that the suffix $\text{SA}[i]$ is preceded in T by c: $cS\$$. Therefore, performing $j = LF[i]$ will return the row j in $\mathcal{M}$ prefixed by $cS\$$. The backward search algorithm maintains a range $\langle sp, ep\rangle$ representing the rows in $\mathcal{M}$ prefixed by a suffix of the pattern $P[i..m-1]$ at step i. After m steps, we have determined all rows in $\mathcal{M}$ prefixed by $P[0..m-1]$ which corresponds to the same range in the SA of T.

First we determine the range in $\mathcal{M}$ starting with the last symbol in P by using the cumulative count array Q which marks the starting position of each symbol in F: $sp = Q[c]$ and $ep = Q[c+1]-1$. In our example, $sp_0 = 1$ and $ep_0 = 5$. Next we perform two $rank$ operations, counting the number of times the second last symbol ($P[1] = h$) occurs before $\langle sp_0, ep_0\rangle$ and within the range. This operation computes the LF mapping for the first and last occurrence of the currently processed suffix of P. Recall the LF mapping discussed in Section 2.4.1, which is used to recover T from T^{bwt} by stepping through the BWT of the text in reverse order. Backward search emulates the LF mapping by performing $rank$ operations on T^{bwt} instead of storing the mapping explicitly. The new range $\langle sp_1, ep_1\rangle$ is computed by adding to the starting position of all rows in $\mathcal{M}$ prefixed by $P[1] = h$, $Q[h]$, the number of times h occurs before sp_0 in T^{bwt} which is 0. Therefore $sp_1 = Q[h] = 10$. Next, ep_1 is mapped by counting the number of rows in $\langle 0, ep_0\rangle$ of $\mathcal{M}$ preceded by h, which is equivalent to the number of times h occurs in $T^{bwt}[0, ep_0]$. In the last step, we computer the number of times symbol $P[0] = c$ occurs before $T^{bwt}[sp_1]$ and within $T^{bwt}[0, ep_1]$ to retrieve all rows $\langle sp_2, ep_2\rangle$ in $\mathcal{M}$ prefixed by $P = cha$. The pseudo code for the algorithm is shown in Figure 2.13.

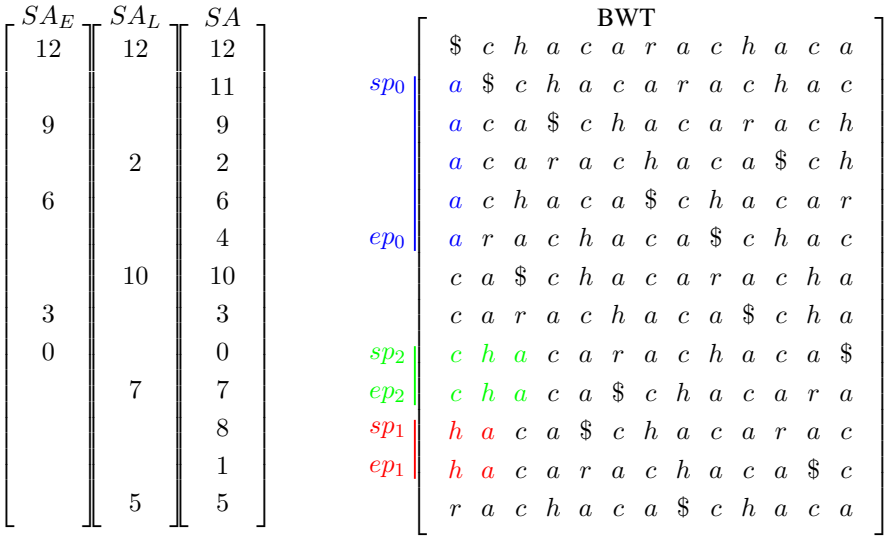

Figure 2.12: Backward search procedure for $P = cha$ and $T =$`chacarachaca$` *by maintaining the range $\langle sp, ep\rangle$ while processing P backwards.*

Overall, we perform $\mathcal{O}(m)$ *rank* operations which can be performed in $\mathcal{O}(m \log \sigma)$ total time using a wavelet tree [Ferragina et al., 2004]. Different time and space trade-offs exist depending on which wavelet tree representation is used. In the original paper, Ferragina and Manzini use a different structure to perform *rank* on the compressed representation of T^{bwt} as the wavelet tree had not been proposed yet. They proposed a technique similar to the *rank* structure shown in Section 2.2.2 over a compressed representation of T^{bwt} which uses move-to-front encoding, run-length encoding and an entropy coder for each symbol c in T^{bwt}. Querying the structure is very complex and only efficient in theory [Navarro and Mäkinen, 2007]. The wavelet tree-based FM-Index however is very simple and fast in practice [Ferragina et al., 2008]. Several wavelet tree-based FM-Indexes have been proposed over time which improve the time and space complexity of the original structure by storing the rank structure more efficiently. Mäkinen and Navarro [2005] show how to perform *rank* over a run length encoded representation of T^{bwt} by storing extra bitvectors, and achieve overall compression relative to the k-th order empirical entropy (H_k). Ferragina et al. [2004] show that building wavelet trees over more compressible chunks of the T^{bwt} – similar to compression boosting [Ferragina et al., 2005] – achieves compression effectiveness bound by the k-th order empirical entropy (H_k). The same compression effectiveness can be achieved when combining a Huffman-shaped wavelet tree over T^{bwt} with compressed bitvector representations [Mäkinen and Navarro, 2007]. Recently, Kärkkäinen and Puglisi [2011b] show via an information theoretic argument that this can also be achieved by parti-

```
1  ⟨sp, ep⟩ backward_search(P[0..m − 1]) {
2           i = m − 1
3           c = P[i]
4           sp = Q[c], ep = Q[c + 1] − 1
5           while ( (sp < ep) and (i ≥ 1) )
6                    c = P[i − 1]
7                    sp = Q[c] + rank(T^bwt, c, sp)
8                    ep = Q[c] + rank(T^bwt, c, ep + 1) − 1
9                    i = i − 1
10          return ⟨sp, ep⟩
11 }
```

Figure 2.13: Pseudo code of backward search used in the FM-Index which determines the range $\langle sp, ep\rangle$ by performing at most m iterations of LF using a wavelet tree over the BWT *of T and a commutative count array Q to represent F.*

tioning T^{bwt} into fixed length blocks. Note that the time complexity of $count$ for wavelet tree-based FM-Indexes depends on the time complexity of performing $rank$ over the wavelet tree. A regular binary wavelet tree requires $\mathcal{O}(m \log \sigma)$ time to perform $count$ for a pattern of length m. For example, Huffman-shaped wavelet tree requires only $\mathcal{O}(mH_0(T))$ binary $rank$ operations. Multi-ary wavelet trees can further be used to reduce the time complexity of an individual $rank$ operation [Ferragina et al., 2007].

Interestingly, the FM-Index is also considered to be a *self-index* which implies that the index contains enough information to recover the original text or any substring $T[i..j]$. The FM-Index therefore supports $\texttt{extract}(i, j)$. In addition, the FM-Index also supports locating all positions of P in T which is equivalent to the `locate` operation. Ferragina and Manzini [2000] use suffix array sampling to allow extracting and locating in a FM-Index. Instead of storing SA completely, we only store every t-th element at a total cost of $n/t \log n$ bits in SA_L. Consider our example in Figure 2.12. Assume we want to access $\text{SA}[5]$ which is 4 but is not sampled it in SA_L. Therefore, we perform $LF(5)$ which is 7. We still have not stored $\text{SA}_L[7]$, therefore, we again perform $LF(7) = 3$. We now arrived at a row in $\mathcal{M}$ for we which we store $\text{SA}_L[3] = 2$. As we performed two $LF()$ operations we know that $\text{SA}[5] = 4$. Due to the regular sampling of SA, we are guaranteed to perform at most t $LF()$ operations at a cost of $\mathcal{O}(t \log \sigma)$. The same idea can be used to allow extracting parts of T from the index. However, the sampling has to occur in *text order*. That is, instead of sampling $\text{SA}[0], \text{SA}[3], \text{SA}[6]$ and so on, we sample the positions in SA that store $\text{SA}[x_i] = 0, \text{SA}[x_j] = 3, \text{SA}[x_j] = 6$, etc. Ferragina and Manzini [2000] use a sample interval of $t = \log^{1+\epsilon} n$

for any $\epsilon > 0$ at a total cost of $\mathcal{O}(n/\log^{\epsilon} n)$ extra space. In addition, a bitvector supporting $rank$ and $select$ is stored to mark each sampled row in $\mathcal{M}$ to determine an occurrence of a sampled position when performing $LF()$. This can be done efficiently using the techniques described in Section 2.2. All occurrences of P can be retrieved in $\mathcal{O}(occ\ \log^{1+\epsilon} n)\ LF()$ steps.

Compressed Suffix Array of Sadakane

Sadakane [2002; 2003] proposed a version of the *Compressed Suffix Array* (CSA) which is an extension of work by Grossi and Vitter [2000]. The key component of the compressed suffix array is the inverse function of $LF()$ used by FM-Index type indexes called $\psi()$ (PSI). The function is defined as follows:

$$\psi(i) = j \qquad \text{such that} \qquad \text{SA}[j] = (\text{SA}[i] \mod n) + 1.$$

The original compressed suffix array of Sadakane [2003] stores the array $\psi[]$ in compressed form. The ψ array contains runs of numbers similar to those present in inverted indexes which can be compressed efficiently [Witten et al., 1999]. The compressed suffix array of Sadakane [2002] provides similar search capabilities than the FM-Index. In this thesis we do not use this data structure. We therefore refer to Sadakane [2002; 2003] for a more detailed description of the data structure.

2.5.3 Compressed Suffix Trees

Similar to the way compressed suffix arrays provide the same functionality of suffix arrays in compressed space, there exist *compressed suffix trees* (CST) which emulate suffix trees in compressed space. Compressed suffix trees use compressed suffix arrays (for example the FM-Index), compressed representations of the suffix tree shape, and compressed representations of the lcp array to perform operations supported by uncompressed suffix trees. There are generally two types of compressed suffix trees proposed in previous work [Ohlebusch et al., 2010]. The first is based on the fully functional compressed suffix tree proposed by Sadakane [2007b]. This CST uses a compressed suffix array implementation such as the FM-Index or the alternative representation of Sadakane [2002]. The space used by the compressed suffix array depends on the chosen index type. In addition to the compressed suffix array, the CST uses a total $6n$ bits to represent the lcp array ($2n$ bits) as well a balanced parenthesis (BP) representation of the tree structure in $4n$ bits. A suffix tree of a text of length n has at most $2n - 1$ nodes. A balanced parenthesis representation of a tree of $2n$ nodes requires at most $4n$ bits of space. The BP representation was initially proposed by Jacobsen [1989] to succinctly represent static unlabeled trees. The space usage and time complexity of the tree was later

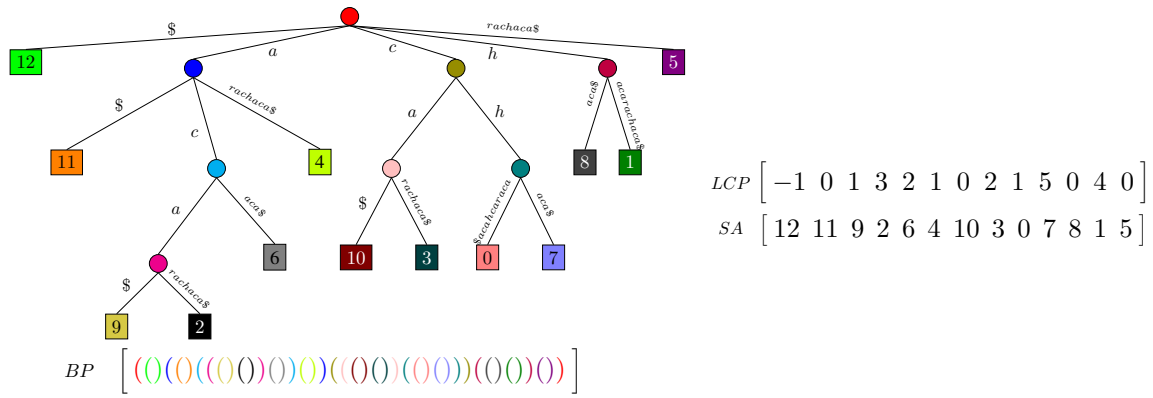

Figure 2.14: Components of a compressed suffix tree of text T =`chacarachaca$` *with the balanced parenthesis representation (*BP*), the suffix array (*SA*) and the longest common prefix array (lcp). Note the colours of the parenthesis representation corresponding to the nodes in the original suffix tree (left).*

improved by Munro and Raman [1997] and first applied to suffix trees by Munro et al. [2001]. The balanced parenthesis of a tree is built by performing a pre-order traversal of the tree starting from the root. When entering a node write '(' or 1 bit. When leaving the node write ')' or a 0 bit. Overall, for a tree of n nodes, $2n$ bits are used to represent the sequence of parenthesis. An example of a balanced parenthesis representation of a suffix tree is shown in Figure 2.14 for our running example of the text T =`chacarachaca$`. The BP representation can be used to perform many suffix tree navigation options in constant time using *rank* and *select* over the BP bitvector [Ohlebusch et al., 2010; Sadakane, 2007b]. For example, finding the parent of a node represented by a given parenthesis at position $bp[i]$ is equivalent to finding the parenthesis pair *enclosing* position i. This operations is commonly called `enclose`(i) and can be performed in constant time [Munro and Raman, 1997].

As seen in Figure 2.14, the *lcp* representation can be used to represent the depth of a node in the suffix tree. For example, all nodes with an *lcp* value of 0 are direct children of the root node, whereas suffixes with larger *lcp* values are children of internal nodes in the suffix tree. Interestingly, the complete tree structure is implicitly stored inside the *lcp* array. The second type of compressed suffix trees do not store the tree shape explicitly using a balanced parenthesis representation, but implicitly deduce the shape from the *lcp* array [Fischer et al., 2009; Ohlebusch et al., 2009]. Tree navigation is supported by supporting three operations on a compressed representation of the *lcp* array efficiently: Range Minimum Queries (RMQ), Next Smaller Value (NSV) and Previous Smaller Value (PSV). Overall, the implicit storage of the tree shape is more space efficient, but results in operations on

the compressed suffix tree being slower in practice [Ohlebusch et al., 2010]. Recently, full functioning compressed suffix trees have been proposed unifying both general approaches [Navarro and Sadakane, 2013].

2.5.4 Alternative Text Indexes

Next we give a brief overview of other approaches to text indexing which are not based on the suffix array or suffix tree. We discuss grammar and Lempel-Ziv compression based indexes and then give a brief introduction to inverted indexes, which are extensively used in practice.

Grammar Compressed and LZ Text Indexes

The main focus of Stringology research and practical applications of succinct text indexes are compressed suffix-based text indexes. However, there exist several approaches of non-suffix-based self-indexes. Kreft and Navarro [2012] propose an LZ77 like encoding called *LZ-End*. They propose a self-index-based on this encoding which works well for inputs with long repetitions. The approach is an extension of a LZ77-based text-index proposed by [Kärkkäinen and Ukkonen, 1996] which still requires the original text in plain form.

Claude and Navarro [2012b] show how to perform search on a special grammar compressed index. Gagie et al. [2012a] propose a self-index-based on context-free grammars. Maruyama et al. [2013] propose a different grammar compressed-based self index based on *edit-sensitive parsing* which is competitive to FM-based indexes in practice.

Inverted Indexes

One of the most widely used text index data structures is the inverted index [Witten et al., 1999; Zobel and Moffat, 2006]. Unlike the full-text indexes discussed above, traditional inverted indexes are token and document based. The index can only be used to search for tokens chosen during index time. Further, occurrences of a pattern are generally indexed on a document level. That is, for a given term t_i, which documents contain t_i. An inverted index consists of three main components as shown in Figure 2.15. The *vocabulary* stores the indexed terms, and for each term, term statistics and a pointer to the postings list for that term. The *postings lists* store, for a given token t_i, the documents containing the token. Often additional information such as the number of times t_i occurs in document D_i is stored. This is called the term document frequency $f_{d,t}$. To allow reporting of the exact occurrence positions within a document D_i, position offsets pos_{D_i} can be stored within a postings list. Finally, the inverted index contains a *document store* which contains the source

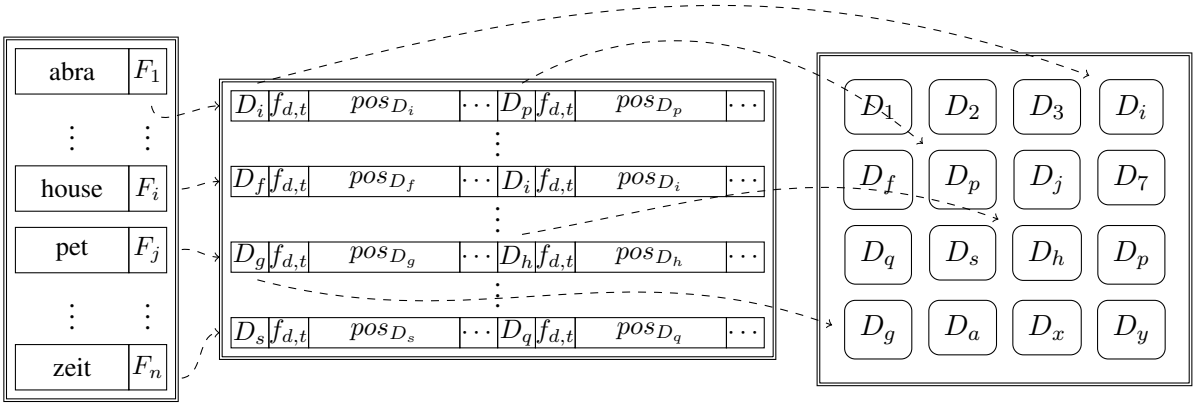

Figure 2.15: Components of an inverted index with the vocabulary a), the postings lists b) and the document store c).

document and maps it given document D_i, allowing access during query time. Trade-offs exist for all parts of the inverted index. Here we briefly give an overview for the different parts of the inverted index.

Vocabulary The vocabulary can be stored as a hash table, as a trie, a B-tree or as a suffix tree. Compressed suffix trees or suffix arrays can also be used to represent the vocabulary. To compress the vocabulary, *Front Coding* can be used to take advantage of long lcp values of entries in the vocabulary [Witten et al., 1999]. Brisaboa et al. [2011] propose several other compressed string dictionaries which can also be used to represent the vocabulary of an inverted index. Their best dictionary is based on a combination of *Front Coding* and *Hu-Tucker*-Codes [Hu and Tucker, 1971]. Generally, the vocabulary only contributes a small amount to the total size of the index, but has to be accessed for every query operation. Therefore, faster query times are more important than compressibility.

Postings lists Postings lists comprise the main part of the inverted index. For most inverted index types, postings lists consist of sequences of monotonically increasing document numbers which can be compressed effectively. Similar to the runs in $\psi[i]$ in the compressed suffix array, individual postings lists can be compressed very effectively using difference encoding (often called d-gaps in this context) and Elias δ-codes [Zobel and Moffat, 2006]. However, in practice a byte-wise encoding

such as v-byte encoding proposed by Scholer et al. [2002] or word aligned codes of Anh and Moffat [2005] are used instead at a small loss of compression effectiveness while, improving processing speed of individual postings lists significantly.

Postings lists processing Processing of postings list during query time has also been an active area of research. Most queries processed by inverted indexes are *ranked top-ϕ document retrieval* queries. Given a query, return the top ϕ most relevant documents based on a similarity function S. Answering these queries efficiently has been key to making inverted indexes very fast in practice. The key concept in efficient evaluation of postings lists is the improvement of efficiency of the inverted index while not affecting the effectiveness of the results returned by the query evaluation system.

To efficiently answer ranked queries, partially processing postings lists while retaining retrieval effectiveness has been an active area of research. Two competing approaches are used in practice. These approaches can be classified as *Term-At-A-Time processing* (**TAAT**) or *Document-At-A-Time* (**DAAT**) processing [Turtle and Flood, 1995]. **TAAT** processing evaluates each postings list, corresponding to a single token t_i, at a time, whereas **DAAT** processing evaluates the postings lists of all tokens in the query at the same time.

Boolean conjunctive queries, or intersection, is another important query in IR which can be answered efficiently using inverted indexes. Boolean conjunctive queries can be seen as a form of *set intersection*, where each postings list is seen as a set of document identifiers. Set intersection, especially on compressed postings lists has been extensively studied in theory as well as in practice [Barbay and Kenyon, 2008; Culpepper and Moffat, 2010; Barbay et al., 2009].

Document store For most queries processed by an inverted index, accessing the plain-text documents is not necessary. However, post-processing the results to contain short summaries or snippets of the return documents is a common feature of many IR systems based on inverted indexes [Turpin et al., 2007]. Hoobin et al. [2011] classifies the methods to compress and access the document store into two categories: Semi-static methods and adaptive methods. Semi-static Methods make two passes of the collection. The first pass gathers statistics over the collection which are then used to compress the collection and create the document store. Hoobin et al. [2011] propose a compression scheme called (RLZ) or relative LZ compression which is based on LZ77. The proposed compression scheme allows fast random and sequential access to documents while compressing a large collection (426 GB) to roughly 10% of their original size.

2.6 Document Retrieval

Previously we only focused on the exact pattern matching problem, that is, locating or counting all occurrences of a pattern P in a text T. Here we discuss a set of related problems. Instead of a monolithic text collection, the text T, represents a document collection D, consisting of d concatenated documents $\{D_1, D_1, D_d\}$.

2.6.1 Document Listing Problem

Performing search on a collection of documents instead of a single string has been the dominant search paradigm in the field of IR for many decades. However, from a theory perspective, Muthukrishnan [2002] formally introduced the *document listing* problem as:

Definition 4 *A* **document listing** *search takes a pattern P of length m and a text T of length n partitioned into d documents $\{D_1, D_2, \ldots, D_d\}$ and returns all documents ($docc$) containing P exactly once.*

The *document listing problem* is related to the *occurrence listing problem* discussed in Section 2.6.1, where one returns all occurrences (occ) of a pattern P in T. As discussed in Section 2.5.1, using a suffix tree, the occurrence listing problem can be solved optimally using $\mathcal{O}(n)$ preprocessing time and $\mathcal{O}(m + occ)$ query time.

Range Minimum Queries (RMQ) can be used to return all distinct documents containing P in $\mathcal{O}(m + docc)$ time [Muthukrishnan, 2002]. The algorithm works as follows. First, define an array DA, usually referred to as the *Document Array*, which maps each suffix position to its corresponding document:

$$\text{DA}[i] = D_j \qquad \text{if} \qquad \text{SA}[i] \in D_j.$$

at a total cost of $n \log d$ bits. Using DA, we can map each suffix to its corresponding document in $\mathcal{O}(1)$ time. Next, Muthukrishnan [2002] defines a second array G as

$$G[i] = j \qquad \text{if} \quad j < i, \text{DA}[i] = \text{DA}[j] = D_k.$$

where j is the next smaller suffix in SA within the same document D_k. If there is no j, such that the suffix $\text{SA}[i]$ is the smallest suffix in a given document, we set $G[i] = -1$. G therefore links all suffix positions within a document together as $\text{DA}[G[i]] = \text{DA}[i]$ and $\text{SA}[G[i]] < \text{SA}[i]$.

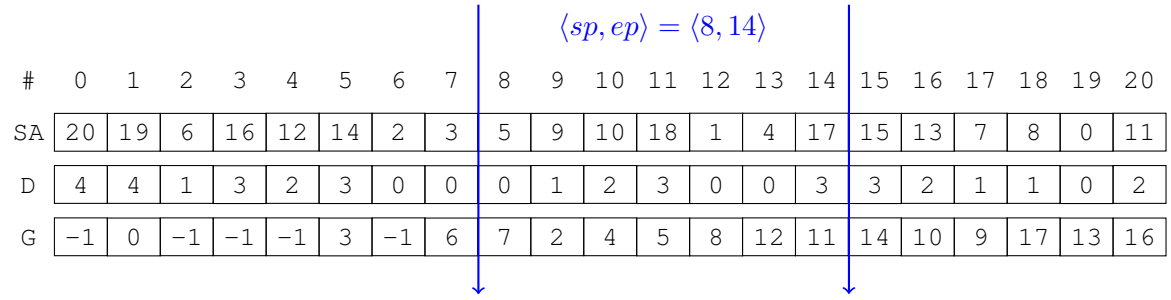

Figure 2.16: Document listing approach of Muthukrishnan [2002] with DA *array and* G *array used to uniquely list each document in* $\langle sp, ep \rangle$.

Figure 2.16 shows an example of the two arrays DA and G. Note that every $G[i]$ "links" to position i to the next smaller suffix SA$[j]$ (to the left) where DA$[i]$ = DA$[j]$. For example, $G[14] = 11$ as DA$[14] = 3$ and the next occurrence of 3 in DA is at position $G[11] = 3$. Given a range $\langle sp, ep \rangle$ in SA corresponding to all occurrences of P in T the algorithm returns all distinct documents matching P as follows. First we perform $j = RMQ(G[sp, ep])$ to find the position of the smallest element in $G[sp, ep]$ in constant time. We now check if $G[j] < sp$. If this is the case we output DA$[j]$ as DA$[j]$ is the leftmost occurrence of that document in DA$[sp, ep]$. Next we recurse and perform $RMQ(G[sp, j-1])$ and $RMQ(G[j+1, ep])$. Again we only output if the value is smaller than sp. This guarantees that we output each document exactly once. The recursion stops once the $RMQ(G[x, y])$ query returns a $G[j]$ larger than sp, which implies that there are no candidates within $G[x, y]$ which have to be processed. Overall we perform $\mathcal{O}(docc)$ RMQ queries which can be performed in constant time each. Therefore, the overall time to list all documents containing P is $\mathcal{O}(m + docc)$. In our example shown in Figure 2.16, we first perform $RMQ(G[8, 14]) = 9$ as $G[9] = 2$ is the smallest element in $G[8, 14]$. We therefore output DA$[9] = 1$. Next we recurse and perform $RMQ(G[8, 8]) = 8$ and $RMQ(G[10, 14]) = 10$. We output both DA$[10] = 2$ and DA$[8] = 0$ as both $G[10] = 4$ and $G[8] = 7$ are smaller than $sp = 8$. We continue to recurse using $RMQ(G[11, 14])$ and output DA$[11]$ which stops the recursion as $RMQ(G[12, 14]) = 8$, which is not smaller than sp. The space usage of the solution in addition to the suffix array or suffix tree is $n \log d$ bits for storing DA as well as $n \log n$ bits for G plus $2n + o(n)$ bits to support RMQ in constant time [Fischer, 2010]. Overall the space usage is $\mathcal{O}(n \log n)$ bits.

Välimäki and Mäkinen [2007] reduce the space requirements of the solution proposed by Muthukrishnan [2002]. First, instead of using a suffix tree or suffix array, a compressed suffix array such as the FM-Index discussed in Section 2.5.2 is used. Additionally, it is possible to calculate $G[i]$ on-the-fly

using a *wavelet tree* over DA, as:

$$G[i] = select(\text{DA}, \text{DA}[i], rank(\text{DA}, \text{DA}[i], i) - 1).$$

That is, the position of the lexicographically next smaller suffix within the same document can be calculated on-the-fly in $\mathcal{O}(\log d)$ time. RMQ queries over G can be answered without accessing G using $2n + o(n)$ bits [Fischer, 2010]. Therefore, the space cost of solving the *document listing* problem is reduced to the cost of storing the compressed suffix array, the wavelet tree over DA plus $2n + o(n)$ bits to perform RMQ queries over G.

Sadakane [2007a] uses a similar approach as Välimäki and Mäkinen [2007], but instead of storing DA explicitly uses a bitvector B_D to mark the document boundaries within T. $\text{DA}[i]$ can then be calculated using $\text{DA}[i] = rank(B_D, \text{SA}[i], 1)$. Note that this operation is not constant as $\text{SA}[i]$ has to be calculated from the compressed suffix array as shown in Section 2.5.2. Additionally, Sadakane stores a compressed suffix array of each document, csa_{D_i}, to calculate the frequency with which the pattern occurs in document D_i. The total cost of storing all csa_{D_i} is bound above by the cost of storing the compressed suffix array of all concatenated documents [Hon et al., 2009]. To calculate the frequency, Sadakane stores an additional RMQ structure to determine the leftmost and rightmost occurrence of document D_i in $D[sp, ep]$ in constant time. These are then mapped, in constant time to the corresponding positions in csa_{D_i} to calculate the frequency by determining the relative position of the leftmost and rightmost match of P in csa_{D_i}.

Range Quantile Queries (RQQ) discussed in detail in Section 2.3.3 can be used to solve the document listing problem in $\mathcal{O}(m + doc \log \sigma)$ time using a wavelet tree over DA [Gagie et al., 2009]. The method additionally allows retrieving the frequency of P in each document D_i at no extra cost.

Muthukrishnan [2002] formally introduces two related problems of independent interest: First the *Document Mining Problem* where, for a given P, all documents containing P at least k times are to be returned. Second, the *repeats problem*, where all documents d_i are to be found where P occurs at least twice, where the two occurrences are separated by at most k symbols.

2.6.2 Top-ϕ Document Listing Problem

The *top-ϕ document listing problem* is related to the document listing problem discussed above. Formally it is defined as:

Definition 5 *A* **top-ϕ document search** *takes a pattern P, an integer $0 < \phi \leq d$, and a text T of*

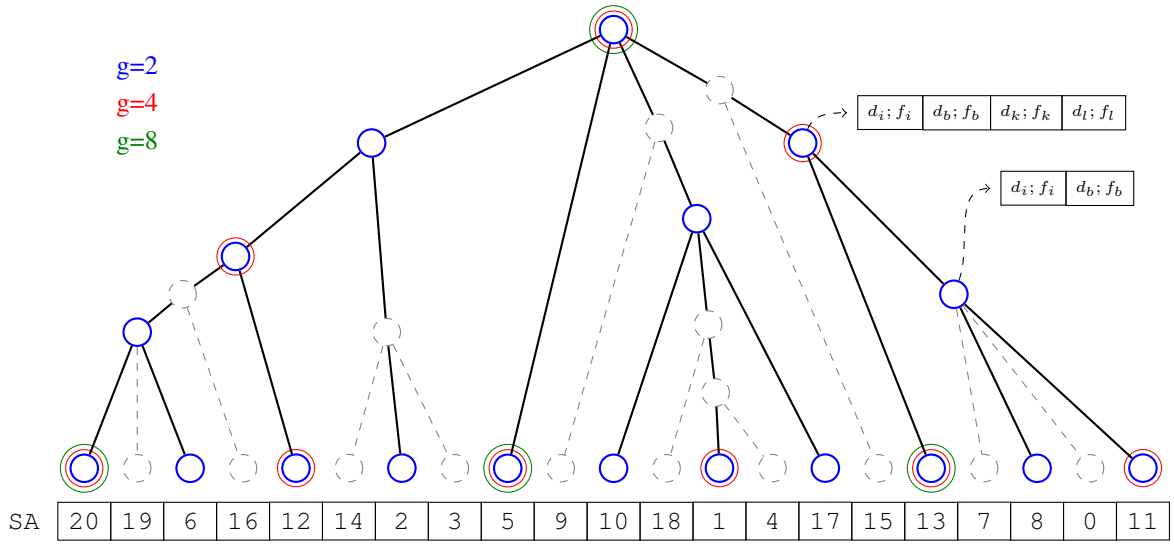

Figure 2.17: Top-ϕ retrieval structure of Hon et al. [2009] with multiple skeleton suffix trees marked at intervals $g = 2, 4, 8$ with FLists attached to each marked node.

length n partitioned into d documents $\{D_1, D_2, \ldots, D_d\}$ and returns the top-ϕ documents ordered by the number of times P occurs in each document D_i.

Essentially, instead of exhaustively returning *all* documents matching P, we only return the ϕ documents where P occurs most frequently. In the following we discuss two main approaches used in the literature. First, we discuss the skeleton suffix tree-based approach of Hon et al. [2009] commonly referred to as HSV after the authors. Next we review an alternative, practical approach based on wavelet trees proposed by Culpepper et al. [2010]. Last we give an overview of recent enhancements to both approaches as well as alternative approaches.

Skeleton Suffix Tree of Hon et al. [2009]

Hon et al. [2009] were the first to consider the top-ϕ retrieval problem. They provide a linear space data structure to solve the problem in optimal time. The main idea is as follows. For a given P and the corresponding range $\langle sp, ep \rangle$ in SA obtained via a CSA, ensure that at most $2g$ positions have to processed by pre-computing the "answers" for certain parts of the suffix array. Here $g = \phi \log^{2+\epsilon} n$.

The structure is built as follows. First create a suffix tree over T. Next mark the leaves of the suffix tree, or the corresponding positions in the suffix array, at fixed intervals $g = \phi \log^{2+\epsilon} n$ for $\phi = 2$. Next, mark the lowest common ancestors (LCA) of all marked nodes. This corresponds to all

blue nodes marked in the example shown in Figure 2.17. We refer to the marked nodes as the skeleton suffix tree τ_2. We perform the same marking for $\phi = 4$ and $\phi = 8$ which is shown in Figure 2.17 as red and green nodes respectively. We refer to these trees as τ_4 and τ_8. In theory we create skeleton suffix trees for $\phi = 2, 4, 8, 16...\phi_{max}$, where at most $\log n$ trees are stored in total. Overall, in each skeleton tree we mark at most $2n/g$ nodes. At each marked v node we store the top-ϕ most frequent documents in the range $\text{SA}[l, r]$ where l and r are the leftmost and rightmost leaf descending from v. Therefore for the tree τ_2 we store the top-2 most frequent documents at each node. For each skeleton tree we mark at most $2n/(\phi \log^{2+\epsilon} n)$ nodes. At each node we store $\mathcal{O}(\phi)$ items. Therefore, each tree τ_k requires $2n/(\log^{2+\epsilon} n)$ words or $2n/(\log^{1+\epsilon} n)$ bits of space. We store at most $\log n$ trees which results in $2n/(\log^{\epsilon} n)$ bits total space usage for all skeleton trees and pre-calculated top-ϕ values. In addition to the structure described above we store a compressed suffix array over T at a cost of the size of the compressed suffix array as discussed in Section 2.5.2. Similar to the approach of Sadakane [2007a] discussed above, we additionally store a compressed suffix array for each document D_i. This is used during query processing in case the pre-computed values are not sufficient to answer a given query.

Queries are answered as follows. First, for a given query, P and ϕ_q, we determine the correct skeleton tree τ_ϕ by rounding ϕ_q up to the nearest power of two. Next we determine the range $\text{SA}[sp, ep]$ corresponding to P using the CSA. Using $\langle sp, ep \rangle$ we start traversing τ_ϕ. Each node in τ_ϕ corresponds to a range in $\text{SA}[i, j]$ covered by the leaf nodes in the corresponding suffix tree. We traverse the tree to find the first node v, starting from the root, whose range $\langle v_l, v_r \rangle$ contains $\langle sp, ep \rangle$. Hon et al. [2009] show that $sp - l_v$ plus $r_v - ep$ can not be larger than $2g$, which bounds the number of items to be processed during query time. Note however that g depends on ϕ, the number of items pre-stored in τ_ϕ. For large ϕ, potentially large sections of SA, specifically $\text{SA}[l_v, sp]$ and $\text{SA}[ep, r_v]$, have to be processed.

Wavelet Tree Approach of Culpepper et al. [2010]

A more pragmatic approach was proposed by Culpepper et al. [2010]. They create a balanced wavelet tree over the *document array* used by Muthukrishnan [2002] and Välimäki and Mäkinen [2007]. Two new algorithms to traverse a wavelet tree and retrieve the top-ϕ most frequent documents are presented: *greedy* and *quantile probing*. The algorithms are discussed in detail in Section 2.3.3. In practice the greedy wavelet tree approach outperforms other top-ϕ document approaches and can answer phrase queries faster than inverted index based approaches.

Alternatives

Several improvements to the three main approaches have been proposed over time. Grammar compression can be used to reduce the size of the wavelet tree over the DA array at the cost of reduced query performance [Navarro et al., 2011]. *Monotone minimum perfect hash functions* [Belazzougui et al., 2009] can be used to reduce the space requirements of the *document array* in combination with the skeleton suffix tree [Hon et al., 2009; Belazzougui and Navarro, 2011]. Navarro and Nekrich [2012] transform the top-ϕ problem into performing range queries on a grid and show how to solve the problem in optimal $\mathcal{O}(m + \phi)$ time.

2.6.3 Top-ϕ Ranked Document Listing Problem

In traditional IR systems, retrieving the top-ϕ documents for a given a pattern in frequency order is often not sufficient to answer the *information need* of a user. Instead, IR systems solve the *ranked* retrieval problem defined as

Definition 6 *Given a query q consisting of one or more query terms q_i, a non negative integer ϕ and a text T partitioned into d documents $\{D_1, D_2, \ldots, D_d\}$, and returns the top-$\phi$ documents ordered by a similarity measure $S(q, D_i)$.*

A *similarity measure* S is used to compute the *relevance* of a document D_i to the query q. A query can consist of multiple *query terms* which are often treated as an unordered set or a "bag-of-words" query [Croft et al., 2009; Baeza-Yates and Ribeiro-Neto, 2011]. Many different similarity measures have been proposed over time. One of the most important similarity measures is **BM25**. Formally it can be defined as:

$$\textbf{BM25} = \sum_{q_i \in q} \log\left(\frac{d - f_{q_i} + 0.5}{f_{q_i} + 0.5}\right) \cdot \text{TF}_{\textbf{BM25}}$$

$$\text{TF}_{\textbf{BM25}} = \frac{f_{q_i,j} \cdot (k_1 + 1)}{f_{q_i,j} + k_1 \cdot ((1 - b) + (b \cdot \ell_j / \ell_{avg}))}$$

Here, d is the number of documents in the collection, f_{q_i} is the number of distinct document appearances of q_i, $f_{q_i,j}$ is the number of occurrences of term q_i in document D_j, $k_1 = 1.2$, $b = 0.75$, ℓ_j is the number of symbols in the j-th document, and ℓ_{avg} is the average document length of ℓ_j over the whole collection. The free parameters k_1 and b can be tuned for specific collections to improve effectiveness, but we use the standard Okapi parameters suggested by Robertson et al.

[1994b]. Specifically we use a simplified version of the original formula which sets the additional parameter k_3 to zero [Zobel and Moffat, 2006].

In the bag-of-words retrieval model, each document is seen as an unordered set of terms (a *bag*). Thus each query term can be evaluated independently as a disjunctive boolean query over all terms. In case of **BM25**, the similarity of a document to the query q is calculated by accumulating the *term frequency* scores (TF) normalized by the inverse document frequency (the logarithmic term in the summation above) for each document for each query term. The efficiency of an index solving the top-ϕ ranked document search problem thus depends on the ability to compute the similarity measure for a document efficiently.

Inverted indexes have been heavily optimized to efficiently support retrieving the top-ϕ most "important" documents based on a given *similarity measure*. The main optimization technique used is early termination. Once the number of processed inverted lists can no longer *significantly* affect the outcome of the top-ϕ computation, the result is returned. Several approaches have been proposed to minimize the number of lists/elements to be processed. Early termination techniques exist for both **TAAT** and **DAAT** processing.

Term-at-a-Time Processing (TAAT)

For **TAAT** processing, a fixed number of accumulators are allocated, and the rank contribution – the contribution of a query term to the overall similarity of a document to q – is incrementally calculated for each query term, one term-at-a-time, in increasing document order. When inverted files are stored on disk, the advantages of this method are clear. The inverted file for each term can be read into memory, and processed sequentially. However, when ϕ is small relative to the total number of matching documents in collection, **TAAT** can be inefficient, particularly when the number of terms in the query increases, since all of the inverted lists must be processed before knowing the full rank score of each document. In early work, Buckley and Lewit [1985] proposed using a heap of size ϕ to allow posting lists to be evaluated in **TAAT** order. Processing is terminated when the sum of the contributions of the remaining lists cannot displace the minimum score in the heap.

Moffat and Zobel [1996] improved on this pruning approach with two heuristics: **STOP** and **CONTINUE**. The **STOP** strategy is somewhat similar to the method of Buckley and Lewit [1985], but the terms are processed in order of document frequency from least frequent to most frequent. When the threshold of ϕ accumulators is reached, processing stops. In contrast, the **CONTINUE** method allows the current accumulators to be updated, but new accumulators cannot be added. These accumulator pruning strategies only approximate the true top-ϕ result list.

The true-ϕ result list refers to the ranking of documents after all postings list of all query terms have been completely evaluated, or the computed result list is guaranteed to be equal to that of the complete evaluation. Methods which cannot fulfil this requirement are considered *approximate*. Approximate results are generally acceptable as long as the effectiveness (that is, the quality of the result list) of the approach is not *statistically significantly* different from the true-ϕ result list.

If approximate results are acceptable, the **TAAT** approach can be made even more efficient using *score-at-a-time* or *impact ordering* [Anh and Moffat, 2006]. The key idea of impact ordering is to precompute the TF for each document a term appears in. Next, the TF values are quantized into a variable number of buckets, and the buckets (or blocks) are sorted for each term in decreasing impact order. Now, the top-ϕ representative can be generated by sequentially processing each of the highest ranking term contribution blocks until a termination threshold is reached. The authors refer to this blockwise processing method as *score-at-a-time* processing. Despite not using the full TF contribution for each term, Anh and Moffat [2006] demonstrate that the effectiveness of impact ordered indexes is not significantly reduced, while efficiency is dramatically improved.

Turtle and Flood propose a simple pruning strategy for both **TAAT** and **DAAT** processing called **MAXSCORE**. Using **MAXSCORE**, the processing of a postings-list stops if the sum of remaining the postings list can not change the current top-ϕ result list.

Document-at-a-Time Processing (DAAT)

The alternative approach is to process all of the terms simultaneously, one document at a time [Büttcher et al., 2010]. The advantage of this approach is that the final rank score is known as each document is processed, so it is relatively easy to maintain a heap containing exactly ϕ scores. The disadvantage is that all of the term posting lists are cycled through for each iteration of the algorithm requiring non-sequential disk reads or memory accesses for multi-word queries. However, for *in-memory* ranked retrieval, **DAAT** tends to work very well in practice.

Fontoura et al. [2011] compare several **TAAT** and **DAAT** based *in-memory* inverted indexing strategies. The authors present novel adaptations of **MAXSCORE** and **WAND** [Broder et al., 2003] to significantly improve query efficiency of in-memory inverted indexes. **WAND** uses a two-level approach to only evaluate promising postings lists completely.

The authors go on to show further efficiency gains in **DAAT** style processing by splitting query terms into two groups: rare terms and common terms. The exact split is based on a fixed threshold selected at query time.

The *quality* of the returned result list is often evaluated using human accessors with one of many

effectiveness measures such as *precision* or *recall*. In general, the similarity between the retrieved result list and a "gold standard" list generated by human accessors is compared to judge the quality of the results. In the IR field, statistical analysis is used to determine if the lists are *significantly* different at a given confidence interval using paired t-tests or other metrics.

Alternative Index-based Approaches

From a theory perspective, Hon et al. [2009] propose a suffix tree based index storing document identifiers based on importance in each node of the suffix tree. Overall the index uses linear space and achieves $\mathcal{O}(m + \phi \log \phi)$ query time. Navarro and Nekrich [2012] show a top-ϕ approach using range queries on a point grid can also be used to retrieve documents based on importance in optimal $\mathcal{O}(m + \phi)$ time. However, these approaches only handle singleton term queries and are not designed to answer "bag-of-words" queries.

2.7 Experimental Setup

We now describe the experimental setup we will use for all our experiments. First we describe the hardware used for the experiments. Next we describe the data sets, methodology and tools used in this thesis.

2.7.1 Hardware

Our main experimental machine was a server equipped with $2\times$ Intel Xeon E5640 processors each with a 12 MB L3 cache. The total memory is 144 GB of DDR3 DRAM, 72 GB directly attached to each socket. This implies that the machine is operating on a Non-Uniform Memory Access (NUMA) architecture, where access time to RAM depends on the location of the position to be accessed. Each processor uses a 32 kB L1 data cache and a 256 kB L2 cache per core. The L1 cache line size is 64 bytes. The processor additionally supports memory page sizes of 4 kB, 2 MB and 1 GB. During the experiments we only used one thread of a core. Each CPU further includes a two-level translation lookaside buffer (TLB) with 64 4 kB-page entries in the first level and 512 in the second level. The TLB has 4 dedicated entries for 1 GB-pages in its first level. We refer to this machine as LARGE.

For certain cache/TLB sensitive experiments we use a second machine to sanity check our results on small test instances. A desktop machine equipped with a Intel Core i5-3470 Processor with 6 MB L3 cache, 16 GB of DDR3 DRAM. The processor uses 32 kB L1 data cache and 256 kB L2 cache per core. We refer to this machine as DESKTOP.

2.7.2 Software

Ubuntu Linux version 12.04 served as the operating system on LARGE and version 12.10 on DESKTOP. We used the `g++` compiler version 4.6.3 on LARGE and version 4.7.2 on DESKTOP and used the basic compile options `-O3 -DNDEBUG -funroll-loop`.

Most of the implementations are included in the open source C++ template library SDSL which is available at `http://github.com/simongog/sdsl` [Gog, 2011]. Our fork of the library with additional baselines is available at `http://github.com/mpetri/sdsl`. The library includes a test framework which was used to check the correctness of the implementations. The library also provides methods to measure the space usage of data structures and a provides support for timings based on the `getrusage` and `getTimeOfDay` functions. We state the real elapsed time in all results.

The *papi* library version 4.4.0, which is capable of reading performance counters of CPUs, was used to measure cache and TLB misses. It is available under `http://icl.cs.utk.edu/papi/`. Note that we choose to not use *cachegrind*, a popular cache measurement tool provided in the *valgrind* debugging and profiling tool suite. Cachegrind performs simulations to determine the number of cache misses caused by a program. However, this simulation does not elucidate cache misses and TLB misses caused during address translation, which can affect the performance of data structure on large data sets. As the size of the page table grows with the size of the data, cache misses and TLB misses resulting from accesses to the page table become more frequent and thus more relevant to the overall performance of a data structure.

2.7.3 Data Sets

Over the course of our experiments we use the following data sets:

- The *Pizza&Chili* Corpus: Ferragina et al. [2008] perform an extensive evaluation of compressed text indexes by providing multiple baseline implementations and a widely used test corpus available at `http://pizzachili.dcc.uchile.cl/texts.html`. The corpus consists of files up to 1 GB in size from different domains such as Bioinformatics to source code and English text, and has been used by many researches as a reference collection in experiments [Claude and Navarro, 2008; Navarro and Providel, 2012]. However, the 200 MB test instances are the largest files which are available for all categories on the website. From the *Pizza&Chili* Corpus we use the 200 MB test instances of SOURCES (source code), XML(taken from DBLP), ENGLISH (English text from the Wall Street Journal), DNA (P&C) and PRO-

TEINS (protein sequences). A detailed description of each data set is available on the corpus website.

- To go beyond the 200 MB limit, we created a 64 GB file from the 2009 CLUEWEB web crawl available at `http://lemurproject.org/clueweb09.php/`. The first 64 files in the directory `ClueWeb09/disk1/ClueWeb09_English_1/enwp00/` were concatenated and null bytes in the text were replaced by 0xFF-bytes. We refer to this data set as WEB. Prefixes of size X are denoted by WEB-X.

- Our second larger text is a 3.6 GB genomic sequence file called DNA. We again refer to prefixes of size X as DNA-X. It was created by concatenating the "Soft-masked" assembly sequence of the human genome (hg19/GRCH37) and the December 2008 assembly of the cat genome (catChrV17e) in FASTA format. We removed all comment/section separators and replaced them with a separator token to fix the alphabet size.

From all data sets we created bitvectors denoted by the WT-X extension. Each bitvector was produced by taking a prefix of size X of a given file, calculating the Huffman-shaped wavelet tree WT of the Burrows-Wheeler-Transform of this prefix and concatenating all the bitvectors of WT. For example, WEB-WT-1 GB refers to the 1 GB prefix of the wavelet tree of the BWT of WEB.

For various experiments on bitvectors we further create synthetic data sets as in previous studies on *rank* and *select* [González et al., 2005; Vigna, 2008]. We created bitvectors of varying sizes and densities to evaluate our data structures. The instance sizes range from 1 MB to 64 GB, quadrupling in size each time. For a given density d, the random bitvectors are generated as follows: first, we set the seed of the random number generator to 4711 (`srand(4711)`) to enable reproducibility. We then set $B[i]$ to one with probability d.

2.8 Summary and Conclusion

In this chapter we provided background to text indexing, succinct data structures, succinct text indexes and document retrieval. First we focused on two basic operations *rank* and *select* which form the basis of most succinct data structures. We gave an overview of previous work on implementing both operations over computer words, bitvectors and general sequences. For *rank* on computer words we described a classic population count method [Knuth, 2011], and the initial proposal of Jacobsen [1989] to solve *rank* efficiently on uncompressed bitvectors, and the constant time *select* data structure of Clark [1996]. For both data structures we discussed practical alternatives and empirical evaluations, such as the study of González et al. [2005] which enable additional trade-offs for supporting

both operations over uncompressed bitvectors. Complementary to uncompressed bitvectors we described a compressed bitvector representation which also support $rank$ and $select$ efficiently [Raman et al., 2002].

Using either compressed or uncompressed bitvectors, a wavelet tree can support $rank$ and $select$ over general sequences by decomposing the operations of a non-binary alphabet to $rank$ and $select$ operations on bitvectors [Grossi et al., 2003]. We described the basic concept of wavelet trees and alternative representations. In practice the wavelet tree provides the good performance, a wide variety of time-space trade-offs and additional operations [Mäkinen and Navarro, 2005; Gagie et al., 2012c]. Wavelet trees are one of the key components to implement succinct text indexes [Ferragina et al., 2008].

The main type of succinct text index we focus on in this thesis is the FM-Index [Ferragina and Manzini, 2000]. We discussed how the BWT [Burrows and Wheeler, 1994] is used in the FM-Index to emulate the suffix array by performing *backward search* using a wavelet tree. Additionally, we described in detail the three main operations $count$, $extract$ and $locate$ supported by the FM-Index.

We further discussed alternative index types including inverted indexes which are used in practice to solve the top-ϕ ranked document search problem [Zobel and Moffat, 2006]. Lastly we provided an overview of different document retrieval techniques. We discussed data structures solving the document listing problem [Muthukrishnan, 2002; Välimäki and Mäkinen, 2007] and the top-ϕ most frequent document search problem [Culpepper et al., 2010]. From a practical perspective we discussed inverted index query processing techniques used to solve the top-ϕ ranked document search problem efficiently [Turtle and Flood, 1995].

In the next chapter, we focus on *engineering* practical $rank$ and $select$ data structures on both uncompressed and compressed bitvectors. We also discuss improvements to wavelet tree processing in the context of succinct text indexes as well as improvements to the construction cost of different succinct data structures.

Chapter 3

Optimized Succinct Data Structures

Main memory is often one of the key constraints in processing large amounts of data efficiently. Traditional data structures such as the binary search tree use several times the space of the underlying data to provide additional functionality such as *search* efficiently. For large data sets, there is often not enough main memory available to retain these classical data structures in-memory completely. Several solutions to this problem such as using external-memory or distributing the data structure over machines multiple exist. Succinct data structures can also be used to process large amounts of data which cannot usually fit into main memory using traditional data structures. They require space close to the information theoretic lower bound needed to represent the underlying objects, while at the same time providing the functionality of their classical counterparts. In theory, operations on succinct data structures can be carried out in almost the same time complexity as in their uncompressed counterparts. Optimizing succinct data structures to efficiently operate on large data sets is essential as, from a practical point of view, this is the only time they should replace their equivalent uncompressed data structure.

In addition to memory constraints, which can be mitigated by succinct data structures, processing power has also become a bottleneck for computationally large tasks. For years, programmers could rely on advances in CPU manufacturing which increased processor speeds with every generation. Unfortunately, this is no longer the case, and the speed of processors has been stagnant for the last five years [McKenney, 2011]. New processor generations, in recent years, provide better performance by including multiple processing cores which, unfortunately, do not affect single-threaded program execution. The instruction set supported by current processors has also become more versatile. Additional instruction sets such as SSE 4.2 can perform critical operations more efficiently. However, these instruction sets often require additional effort by the programmer to be effective [Suciu et al.,

2011]. Using these techniques is usually referred to as broadword programming, and has already been successfully applied to the field of succinct data structures [Vigna, 2008].

Another emerging problem is the cost of memory access. Each memory access performed by a program requires the operating system to translate the virtual address of a memory location to its physical location in RAM. All address mappings are stored in a process-specific page table in RAM. Today, all CPUs contain a fast address cache called the Translation Lookaside Buffer (TLB). The TLB is used to cache address translation results similar to the way the first level (L1) cache is used to store data from recently accessed memory locations. If the TLB does not contain the requested address mapping, the in-memory page table has to be queried. This is generally referred to as a TLB miss which similar to a L1/L2 cache miss affects runtime performance. Accessing main memory at random locations results in frequent TLB misses as the amount of memory locations cached in the TLB is limited. Operating systems provide features such as hugepages to improve the runtime performance of in-memory data structures. Hugepages allow the increase of the default memory page size of 4 kB to up to 16 GB which can significantly decrease the cost of address translation as a larger area of memory can be "serviced" by the TLB. Succinct data structures such as FM-Indexes exhibit random memory access patterns when performing operations such as $count$, yet to our knowledge, the effect of hugepages on the performance of succinct data structures has not yet been explored.

Unfortunately, the running time of operations on succinct data structures tends to be slower than the original data structure they emulate in practice. Succinct data structures should therefore only be used where memory constraints prohibit the use of traditional data structures [Ferragina et al., 2008]. Twenty years ago, Ian Munro conjectured that we no longer live in a 32-bit world [Munro, 1996]. Yet, many experimental studies involving succinct data structures are still based on small data sets, which somewhat ironically contradicts the motivation behind developing and using succinct data structures. From a theoretical perspective, the cost to access a given word of memory is constant; in reality, the cost of address translation associated with each memory access becomes more significant as the size of the in-memory data structure increases. This effect however can not be clearly measured when performing experiments on small data sets.

In this chapter we focus on improving basic operations ($rank$ and $select$) on bitvectors used in many succinct data structures. We specifically focus on the performance on data sets much larger than in previous studies. We explore cache-friendly layouts, new architecture-dependent parameters, and previously unexplored operating system features such as hugepages. Specifically, we show that our improvements to these basic operations translate to substantial performance improvements of FM-type succinct text indexes. Within this chapter we gradually shift our focus from basic bit operations on words, to bitvectors, wavelet trees and succinct data structures. Our contributions and the structure

of this chapter can be summarized as follows:

1. First we explore improvements to basic bit operations on a single computer word used in most succinct data structures. Throughout the chapter we show how improvements to these operations affect the operations of larger data structures (Section 3.1).

2. We focus on performing $rank$ and $select$ on uncompressed bitvectors in Sections 3.2 and 3.3.

3. We discuss $rank$ and $select$ in the context of compressed bitvectors in Sections 3.4.

4. We analyse the effects of our optimizations of $rank$ on wavelet trees. We additionally show that performing $rank$ on a wavelet tree cache efficiently can further increase run time performance (Section 3.5).

5. We provide an extensive empirical evaluation by building on the experimental studies of Ferragina et al. [2008], Vigna [2008] and González et al. [2005]. We show how the speed-up of basic bit operations propagates through the different levels of succinct data structures: from binary $rank$ and $select$ over bitvectors to FM-Indexes. To the best of our knowledge, we are the first to explore the behaviour of these succinct data structures on massive data sets (64 GB compared to commonly used 200 MB) (Section 3.6).

3.1 Faster Basic Bit Operations

Answering operations $rank(B, i, c)$ and $select(B, i, c)$ requires computing $rank_{64}$ (often called $popcnt$) and $select_{64}$ on a 64-bit word. The first step to an efficient implementation of a $rank$ and $select$ data structure is to improve the performance of the basic operation $rank_{64}$ and $select_{64}$ which perform $rank$ and $select$ on a 64 bit computer word. In this section we compare different $rank_{64}$ and $select_{64}$ implementations. Additionally, a new, branchless $select_{64}$ implementation is proposed which outperforms all existing approaches.

3.1.1 Faster $Rank$ Operations on Computer Words

Various $popcnt$ methods have been proposed in previous work. In Section 2.2.1 we discuss several of these methods in detail. Here we refer to the table-based $popcnt$ approach proposed by González et al. [2005] as (pop_{table}). The broadword technique discussed by Knuth [2011, p. 143] is referred to as pop_{bw} in the sequel. Recently, researchers have started using advanced CPU instructions (for example from the SSE instruction set) to perform population count more efficiently [Suciu et al.,

	Time in [ns]					
	$rank_{64}$			$select_{64}$		
	pop_{table}	pop_{bw}	pop_{blt}	sel_{table}	sel_{bw}	sel_{blt}
LARGE	5.47	3.97	1.00	24.93	14.56	6.69
DESKTOP	2.88	1.96	0.56	16.02	8.44	3.96

Table 3.1: Mean time over 100 *million iterations to perform one operation in nanoseconds for different* $rank_{64}$ *and* $select_{64}$ *implementations.*

2011]. At the end of 2008 both INTEL and AMD released processors with built-in CPU instructions to solve $popcnt$ efficiently (down to 1 CPU cycle; see e.g. Fog [2012]) on 64-bit words. This method is referred to as pop_{blt} for built-in. Performance comparisons of the different methods on our two experimental machines (described in Section 2.7) are shown in Table 3.1.

Using the fastest methods on a modern computer, a $popcnt$ operation can be performed in several nanoseconds. To measure the performance differences of the different methods correctly, 10^8 operations are performed on random values for each method in this experiment. An array of random numbers is sequentially processed to avoid generating random numbers on-the-fly. Note that the table lookup method pop_{table} which was considered the fastest in 2005 is now roughly five times slower than using the CPU internal pop_{blt} instructions. The broadword technique pop_{bw} used by Vigna is roughly 3 times slower than the pop_{blt} method. Note that there exist other $popcnt$ algorithms in practical use and in literature. We focus here on those commonly used in the context of succinct data structures. We further note that more extensive comparisons also show that pop_{blt} is currently the fastest $popcnt$ algorithm. For a more in-depth comparison of different population count methods we refer to the extensive empirical study of Suciu et al. [2011].

3.1.2 Faster $Select$ Operations on Computer Words

Unfortunately, the development of efficient $select$ operations on 64-bit integers ($select_{64}$) has not been as rapid as for $rank_{64}$. There are no direct CPU instructions available to return the position of the i-th set bit. Method pop_{table} solves the problem in two steps: first, the byte which contains the i-th set bit is determined by performing sequential byte-wise popcounts using a version of pop_{table}. The final byte is scanned bit-by-bit to retrieve the position of the i-th bit. Vigna [2008] proposed a broadword computing method. This method is referred to as sel_{bw}. Figure 3.1 shows a faster variant of Vigna's method (sel_{bw}) which uses two small additional lookup tables and a builtin CPU instruction. Given a 64-bit word x and a number i, the position of the i-th set bit in x is determined

```
 1 uint32_t select64(uint64_t x, uint32_t i) {
 2     uint64_t s = x, b; uint64_t j = i;
 3     s = s-((s>>1) & 0x5555555555555555ULL);
 4     s = (s & 0x3333333333333333ULL)
 5         + ((s >> 2) & 0x3333333333333333ULL);
 6     s = (s + (s >> 4)) & 0x0F0F0F0F0F0F0F0FULL;
 7     s = s*0x0101010101010101ULL;
 8     b = (s+PsOverflow[i]) & 0x8080808080808080ULL;
 9     int bytenr = __builtin_ctzll(b) >> 3;
10     s <<= 8;
11     j -= ((uint8_t*)&s)[bytenr];
12     pos = (bytenr << 3) + Select256[((i-1) << 8)
13             + ((x>>(bytenr<<3))&0xFFULL)];
14     return pos;
15 }
```

Figure 3.1: Fast branchless $select_{64}$ method using built in CPU *instructions and a final table lookup. Returns the position of the i-th set bit in x with improvements to previous versions highlighted in grey.*

as follows. In the first step the byte b_i in x which contains the i-th set bit is determined. In the second step a lookup table is used to select the j-th set bit in b_i, which corresponds to the i-th set bit in word x.

In detail, first the divide and conquer approach described by Knuth is used to calculate the number of ones in each byte in x (lines 2-6). Line 7 calculates the cumulative sums of the byte counts using one multiplication. Next these cumulative sums are used to determine b_i. This can be done by adding a mask (`PsOverflow[i]`) depending on i to the cumulative sums (line 8). After the addition the most significant bit (MSB) of each byte is set, if its corresponding sum was larger than i (line 8). The first byte with a sum larger or equal to i contains the i-th bit in x. A second mask is used to set the remaining 7 bits in each byte to zero. Now the position of b_i corresponds to the number of trailing zeros divided by eight (line $9-10$). A CPU instruction (`builtin_ctzll`) is used to select the leftmost non-overflowed byte b_i. In the second step, the rank j of the i-th bit in b_i is determined by subtracting the number of set bits in x occurring before b_x (line 11). Last, a final lookup table (`Select256`) is used to select the j-th bit in b_i and return the position (pos) of j in the word x.

Next, the new method (sel_{blt}) is compared to the sequential table lookup (sel_{table}) based method used by González et al. [2005] and to the broadword method (sel_{bw}) proposed by Vigna [2008] which to our knowledge is the currently fastest method available. A performance comparison of different

$select_{64}$ methods on our two test machines is shown in Table 3.1. Again, 100 million operations are performed over pre-computed $select$ queries. The new approach (sel_{blt}) is only roughly 6 times slower than the fastest $rank_{64}$ method. Using incremental table lookups (sel_{table}) to determine the correct byte within the 64-bit integer and using bit-wise processing of the target byte is roughly 3 times slower than the fastest method. The broadword method (sel_{bw}) proposed by Vigna [2008][1] is roughly 2 times slower. This relative behaviour between the different methods is similar on both test machines, while all operations on the newer machine, DESKTOP, run roughly twice as fast as LARGE. Overall, careful engineering can significantly improve the performance of $rank$ and $select$ on 64 bit words. Throughout this chapter the effect of these improvements is evaluated on more complex succinct data structures.

3.2 Optimizing $Rank$ on Uncompressed Bitvectors

Most succinct data structures do not directly perform $rank$ on computer words. Instead the abstraction of a bitvector is used. In this section we explore optimizing performing $rank$ on an uncompressed bitvector B ($rank(B, i, 1)$). We first propose a new cache- and TLB-friendly bitvector representation in Section 3.2.1. Second we evaluate our structure in Section 3.2.2. We investigate different time-space trade-offs for our new structure over bitvectors of size 1 GB. In addition, we investigate the scalability of our data structure compared to other uncompressed $rank$ solutions. Here we discuss the effects of efficient $rank_{64}$ implementations, operating system features such as hugepages and bitvector size on the performance of the $rank$ structure. We specifically analyse the TLB and cache performance of all structures to elucidate the impact of the different features on the overall performance of the data structure.

3.2.1 A Cache-aware $Rank$ Data Structure on Uncompressed Bitvectors

In practice, $rank(B, i, 1)$ is answered as described in Section 2.2.2 using a two-level structure proposed by Jacobsen [1988]. As several authors already pointed out, in addition to fast basic bit-operations, minimizing cache and TLB misses is the key to fast succinct data structures [Vigna, 2008; González et al., 2005]. Using two block levels results in 3 cache misses to answer a rank query: one access to R_s and R_b each and a third access to the bitvector to perform $popcnt$ within the target block. To reduce cache and TLB misses, Vigna proposed to *interleave* R_s and R_b [Vigna, 2008]. Each superblock $R_s[i]$ which contains absolute rank samples is followed in-memory by the corresponding blocks $R_b[j..k]$ containing the relative rank samples. The size of the blocks are chosen

[1] available at `http://sux.dsi.unimi.it/select.php`

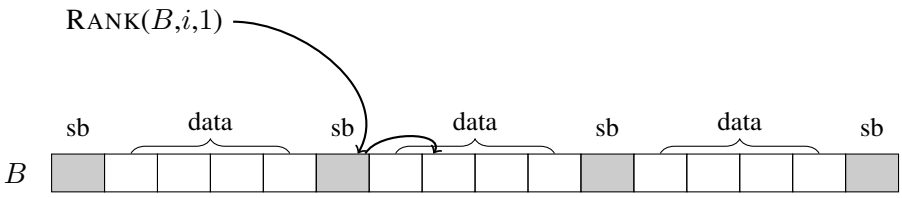

Figure 3.2: Interleaved bitvector (B) showing superblocks (sb) and data interleaved for optimal cache performance. A $rank$ operation first jumps to the interleaved superblock and then iteratively processes the following data block, with block size $4 \times 64 = 256$ bits.

as follows: 64 bits are used to store each superblock value. The second 64 bits are used to store seven 9-bit block counts. Omitting the first block count, as it is always 0, one can answer $rank$ queries in constant time for a superblock size of $s = (7 + 1) \cdot 2^9 = 512$. This reduces the number of cache and TLB misses to two: one to access the pre-calculated counts and a second to perform the population counts within the bitvector. The total space overhead of this approach is $128/512$ bits $= 25\%$. In the following, we call the implementation of this approach RANK-V.

Extending Vigna's approach, we propose to interleave the pre-computed values and the bitvector data as shown in Figure 3.2. We further only store one level of pre-computed values similar to one of the methods proposed in [González et al., 2005]. We conjecture that, for large enough bitvectors and fast $popcnt$ methods, the advantage of only having one cache and TLB miss will outweigh the cost of performing a longer sequential scan over the block as part of block will already be high up in the cache hierarchy.

For each block b we therefore only store a 64-bit cumulative rank count. For a block size of 256 bits, the space overhead of our method is 25%, the same as the solution proposed by Vigna. We call our 25% overhead implementation RANK-IL. More space-efficient versions can be achieved by choosing a larger block size. For example a block size of 1024 results in 6.25% extra space. The same space can also be achieved by a variation of Vigna's approach, which we call RANK-V5. Set the superblock size to $s = 2048$ and the block size to $b = 6 \cdot 64 = 384$ bits. The superblock values are still stored in a 64-bit word and the 5 positive relative 11-bit counts in a second 64-bit word. A rank query then requires two memory accesses plus at most 6 $popcnt$ operations.

3.2.2 Evaluating the Performance of the Cache-aware $Rank$ Data Structure

Figure 3.3 shows the time space trade-offs for our new interleaved bitvector (RANK-IL) for different block sizes for a 1 GB bitvector. The time in nanoseconds for one $rank$ operation averaged over 100

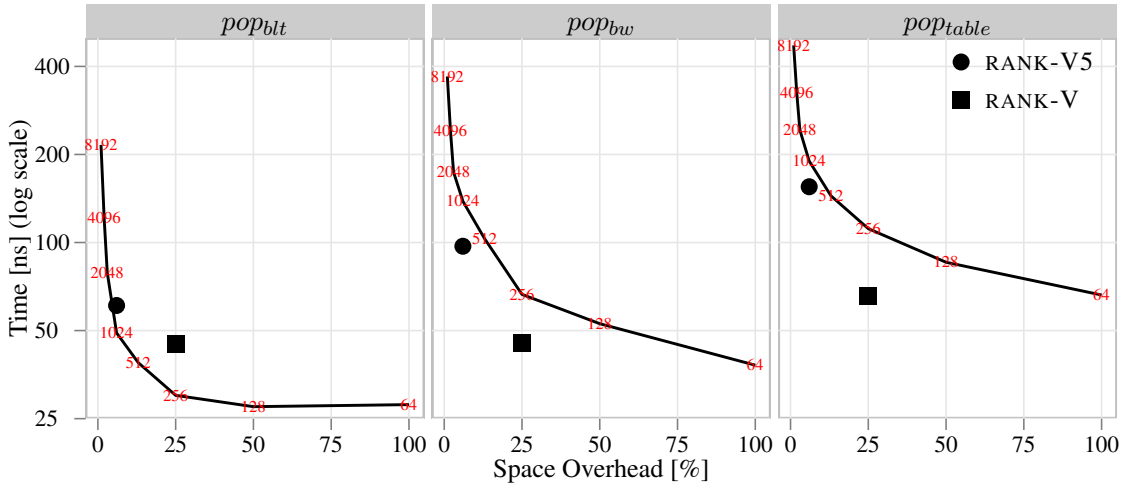

*Figure 3.3: Time-Space trade-offs for our interleaved bitvector representation (*RANK-IL*) for varying block sizes (red) over a* 1 *GB bitvector for different* $popcnt$ *implementations. Also includes time-space trade-offs for* RANK-V *and* RANK-V5 *for comparison.*

million uniformly distributed queries on our small test machine (DESKTOP) is measured. The space overhead shown is the additional space needed as a percentage of the original bitvector size. For blocksize $b = 64$ the space overhead is 100%, as we store a 64-bit superblock for each 64-bit block. As calculated above, RANK-V uses 25% space overhead whereas RANK-V5 uses 6.25%. The best time-space trade-offs for RANK-IL are achieved for block sizes 256 to 2048. The effect of different $popcnt$ implementation is significant (as discussed in Section 3.1) : for the slower pop_{table} and pop_{bw} implementations, the representations proposed by Vigna [2008] outperform our implementation. For the fast pop_{blt} implementation RANK-IL outperforms both RANK-V and RANK-V5 when compared to block sizes of equal space overhead. In fact, RANK-IL using pop_{blt} with space overhead of 3%, which corresponds to a blocksize of $b = 2048$, is faster than RANK-IL using pop_{table} and a blocksize of $b = 128$ (corresponding to a space overhead of 50%). This observation can be explained by the fact that RANK-IL uses sequential $popcnt$ scans proportional to the size of the block size. The running time therefore depends on the block size as well as the underlying $popcnt$ implementation. Both RANK-V and RANK-V5 also profit from faster $popcnt$. RANK-V using pop_{blt} is roughly 40% faster whereas RANK-V5 is roughly twice as fast using pop_{blt}.

Scalability of the Cache-aware $Rank$ Data Structure

The performance of our representation can change depending on the size of the input. Here we evaluate our new representation for different bitvector sizes. As RANK-IL is more cache-efficient than RANK-V, it is expected that RANK-IL outperforms RANK-V as the size of the bitvector increases. Figure 3.4 shows the run time performance of RANK-IL and RANK-V for bitvectors of sizes 1 MB up to 64 GB. Again, 100 million operations on our large test machine (LARGE) are performed and the mean time per operation in nanoseconds is reported. In this experiment, the two most efficient $popcnt$ implementations, pop_{bw} and pop_{blt}, are used to parametrize the previously fastest implementation RANK-V and our new, fully interleaved representation RANK-IL with it. Additionally, different memory page sizes are also evaluated. The standard 4 kB pages (no HP) or 1 GB hugepages (HP) are used, since address translation also determines the performance of the data structure. The results of our random query experiment is depicted in Figure 3.4. Note that the performance is not affected by the density of the bitvector, since accessing the cumulative counts and performing $popcnt$ does not depend on the underlying data. The cost of performing a single $access$ operation on the bitvector is also included in each subgraph as baseline and as a practical lower bound, since a $rank$ operation can not be faster than reading a single bit. Figure 3.4 shows that RANK-V is better than RANK-IL for test instances of larger size if $popcnt$ method $pop_{bw}$is used, and RANK-IL outperforms RANK-V if pop_{blt} is used. It also shows that the runtime of the $access$ and $rank$ operations increases with the size of the data structure, if the standard 4 kB-pages are used.

Performance of a Single $Access$ Operation. The $access$ operation can be used to explain the observed runtime performance with increasing input size. Thus we first discuss the cost of accessing a single bit in a bitvector of increasing size before providing a more detailed explanation for the $rank$ structures. In our explanation we heavily rely on the cache and memory structure of our large test machine (LARGE) described in detail in Section 2.7.1. The only cost of an $access$ operation is the address translation and the memory access to the bitvector. For 4 kB-pages the 512 elements of the TLB can store translations of up to 512×4 kB $= 2$ MB of memory. For a 1 MB-bitvector all address translations can be performed using only the TLB. After a few initial TLB misses for the first random accesses and no more TLB misses occur. The main cost is therefore the L2 miss, which occurs when fetching the bitvector data from L3 cache.

For bitvectors in the range from 2 MB to 12 MB, the bitvector still fits in L3 cache, but TLB misses occur as more memory pages are accessed than TLB entries are available. The handling of a TLB miss is cheap, since the page table for a 12 MB index is about 12 kB large and fits in L1 cache.

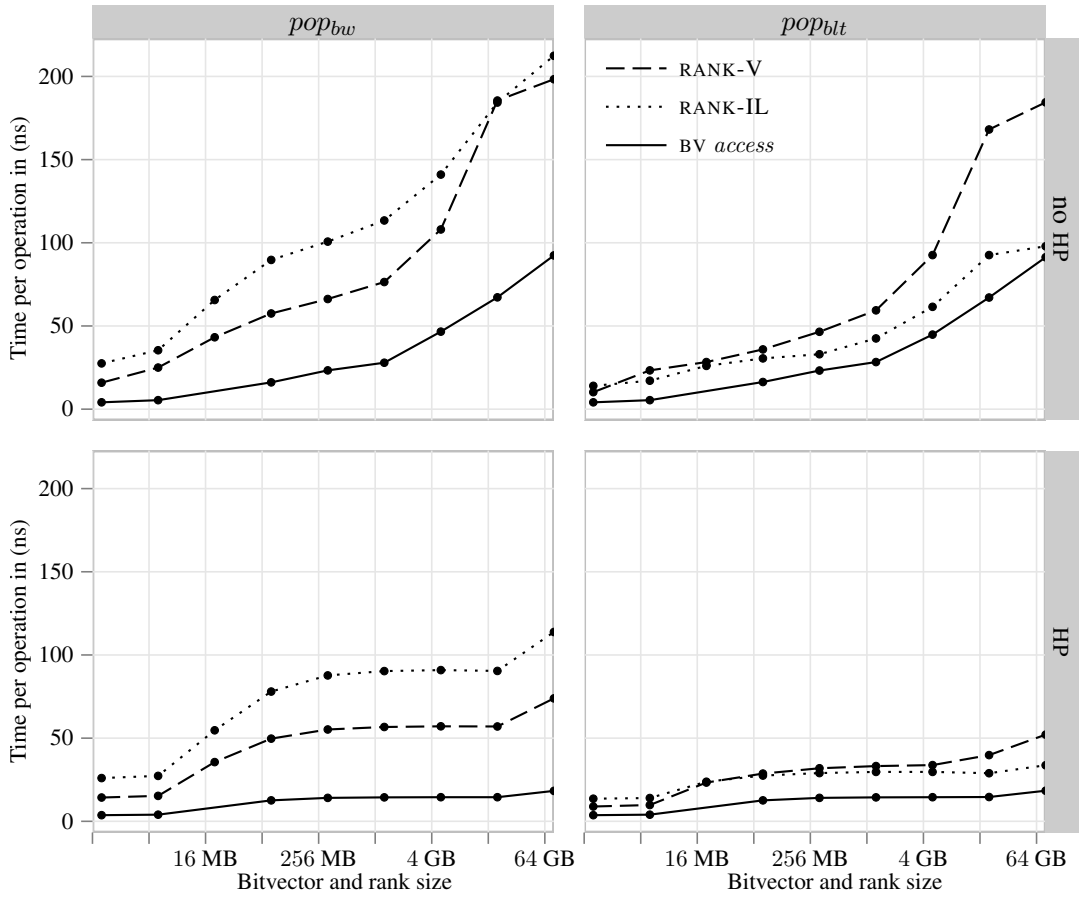

Figure 3.4: Time for single random rank operations on uncompressed bitvectors. The top row shows performance with standard 4 *kB pages, the bottom row shows performance with* 1 *GB pages. The left column shows performance using broadword method proposed by Vigna, right column with* SSE.

This is still the case for bitvectors in the range from 12 MB to 32 MB, but the bitvector itself can not be completely held in the L3 cache (as our L3 cache is 12 MB large) and therefore L3 misses occur and the bitvector data is transfered form RAM. In the range from 32 MB to 256 MB the page table is of size 32 kB to 256 kB and does not fit in L1 cache any more. Therefore, each TLB miss now forces a L1 miss to fetch the page table entry from L2 cache. In the range from 256 MB to 12 GB the page table is larger than the L2 cache and so each *access* operation can cause a L2 miss to update the TLB.

Finally, for bitvectors larger than 12 GB, the page table is larger than the L3 cache. Therefore looking up the address of the page table entry can now itself result in a TLB miss, which in turn

	TLB misses / L1 cache misses per operation							
	1 MB	16 MB	64 MB	256 MB	1 GB	4 GB	16 GB	64 GB
4 kB pages (no HP)								
RANK-V	0.0 /2.0	1.6/3.1	1.9/3.8	1.9/ 4.0	2.1/4.3	2.8/5.0	3.6/5.7	3.9/6.0
RANK-IL	0.0 /2.0	0.9/2.3	0.9/2.0	1.0/ 3.0	1.0/3.2	1.2/3.7	1.3/4.0	1.9/4.3
BV *access*	0.0 /1.0	0.8/1.6	0.9/2.0	1.0/ 2.0	1.0/2.1	1.4/2.6	1.8/2.9	1.9/3.1
1 GB pages (HP)								
RANK-V	0.0 /2.0	0.0/2.1	0.3/2.1	1.4/ 2.1	1.8/2.1	2.0/2.1	2.0/2.1	2.0/2.1
RANK-IL	0.0 /2.0	0.0/2.0	0.2/2.0	0.9/ 2.0	1.0/2.0	1.1/2.0	1.1/2.0	1.1/2.0
BV *access*	0.0 /1.0	0.0/1.1	0.0/0.9	0.7/ 1.1	0.9/1.1	0.9/1.1	1.0/1.1	1.0/1.1

Table 3.2: Average TLB and Level 1 cache misses for a single rank or access operation on uncompressed bitvectors over 10 million queries for 4 kB and 1 GB pages.

can be handled by one L1 miss to access an entry of the upper level of the hierarchical page table. Accessing and loading the found page table entry into memory causes then another L3 miss. In total one *access* operation can result in 2 TLB and 3 cache misses when 4 kB pages are used. This can be seen in Table 3.2: for a 64 GB bitvector, the mean number of TLB misses per *access* query is 1.9, and the mean number of L1 cache misses is 3.1.

Runtime Performance of RANK-V and RANK-IL. The performance of both *rank* operations can now be explained based the discussion of *access* above. The cost of RANK-V are two memory accesses plus one *popcnt* operation. For the 64 GB instance, the TLB and L1 misses are two times the misses for one access. We can observe in Figure 3.4 that the runtime is also doubled. Using the slow *popcnt* adds extra overhead. The new RANK-IL performs only one memory access to the superblock and sequentially reads, for a block size of 256 bits, at most 32 bytes (equal to four 64 bit words) additional bytes. These bytes are not necessarily aligned. This therefore results in one extra L1 cache miss on top of the cost of one access operation.

The use of 1 GB-pages reduces the number of TLB misses. For smaller instance sizes no misses occur. For large bitvectors, the second TLB miss caused by the access to the large page table is no longer needed. We observe in Table 3.2 and Figure 3.4 that the transition from not having a TLB miss to having a TLB miss occurs before the 1 GB size is reached. A reason for this might be that the operating system also stores the kernel itself in one hugepage, which in turn affects the number of TLB entries available to other programs. Further, note that the runtime for the 64 GB bitvector

instance increases slightly. This can be explained by the fact that the whole data structure including the 25% $rank$ overhead, is 80 GB in size and therefore larger than one NUMA node (recall that our system provides non-uniform memory access – NUMA – as discussed in Section 2.7.1), which is 72 GB. As the memory access in NUMA architectures is non uniform, an extra cost might occur when accessing memory outside the current NUMA node which would explain the increase in run time for our 64 GB test case.

Overall, the combination of the pop_{blt} and HP features results in a significant performance improvement for the $rank$ structures compared to the pop_{bw} and no HP version. Our improvements allow performing $rank(B, i, 1)$ at a cost close to that of one memory access, as we can perform all calculations by accessing the cache hierarchy, and are therefore almost optimal.

3.3 Improving $Select$ on Uncompressed Bitvectors

In addition to $rank$, many succinct data structures also use the $select$ operation on bitvectors to emulate the functionality of a regular data structure. Here the implementation and optimization of $select$ on uncompressed bitvectors is investigated. First, an overview of non-constant $select$ implementations used in practice is given (Section 3.3.1). Next, an implementation of the constant time $select$ structure proposed by Clark [1996] is discussed, which is more faithful to the space complexities then previous implementations (Section 3.3.2). The block type distribution of the implementation is analysed, which has also not been done before in previous work (Section 3.3.2). From this analysis a simpler, more space efficient constant time $select$ structure (Section 3.3.3) is proposed which performs well in practice and outperforms commonly used $\mathcal{O}(\log n)$ time solutions as well as our faithful implementation of Clark [1996]'s structure in terms of both space and time. Last we perform an extensive evaluation of different $select$ data structures. Specifically, (1) scalability of different solutions to larger data sets (2) effects of faster $rank_{64}$ and $select_{64}$ operations (3) effects of the hugepage operating system feature (4) TLB and cache miss performance and (5) construction cost are investigated. Additionally, implementations of Vigna [2008] are included in the evaluation to show that the data structures tested are competitive.

3.3.1 Non Constant Time $Select$ Implementations

A $select(B, i, 1)$ operation can be solved by performing $\mathcal{O}(\log n)$ $rank$ operations over a bitvector to determine the position of the i-th one bit using any of the $rank$ representations discussed in Section 3.2. González et al. [2005] show that using binary search on superblock values followed by sequential search of the determined superblock is several time faster. González et al. [2005] sequen-

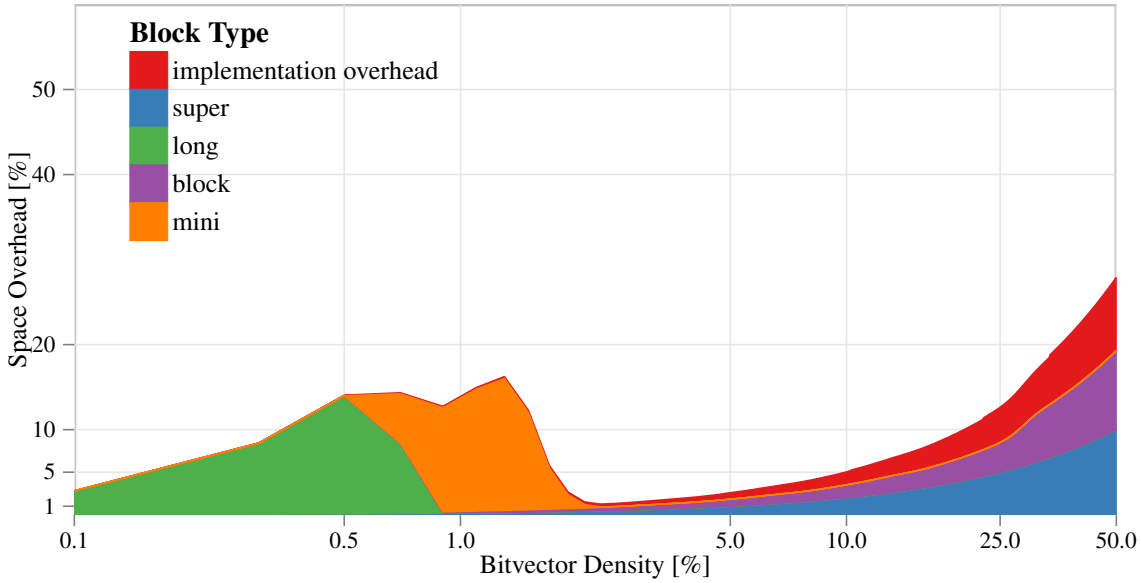

Figure 3.5: Space overhead in percent and block type distribution of constant time select of Clark [1996] of random uniform bitvectors (See Section 2.7.3) of size 128 *MB with densities of* 0.1% *(sparse) up to* 50% *(dense).*

tially search within the superblock by using byte-wise *popcnt*s and a bitwise search for the correct bit in the final byte. This approach is referred to as SEL-BS. However, performing binary search for large inputs is inefficient as the samples of the first binary search steps lie far apart, and can therefore cause cache and TLB misses. In this context, we use a second implementation of binary search *select*, called SEL-BSH, which uses an auxiliary array H. The array contains 1024 rank samples of the first 10 steps of each possible binary search in heap-order. SEL-BSH uses H to improve the locality of the first 10 steps of the binary search.

3.3.2 Faithful Implementation of the Constant Time Structure of Clark [1996]

Clark [1996] presents a sub-linear space constant time 3-level data structure for *select* described in detail in Section 2.2.3. González et al. [2005]'s verbatim implementation of Clark's structure caused 60% percent space overhead in addition to the original bitvector, and was slower than SEL-BS for inputs smaller than 32 MB. We implemented Clark's structure more space efficiently while still staying faithful to the description given by Clark [1996]. This implementation is referred to as SEL-CLARK.

Figure 3.5 shows the space usage of our implementation for bitvector densities of a 1 GB bitvector

for densities ranging from 0.1% (sparse) up to 50% (dense). Recall that depending on the size r of the superblock, the block may be represented as $long$ blocks or a combination of $block$s and $mini$ blocks. Note that our implementation uses at most 28% percent space in addition to the original bitvector. The implementation overhead is at most 9% for dense bitvectors. Next the distribution of the different block types in SEL-CLARK as described in Section 2.2.3 is discussed. The space usage of $super$-blocks increases as the density increases. As the density increases, the distance r covered by a superblock consisting of $\log n \log \log n$ one bits decreases. Therefore, the number of superblocks stored increases. For very low densities (less than 1%), the size of a superblock is large as the $\log n \log \log n$ one bits are spread out over a large area of the bitvector. Therefore each position is stored explicitly in $long$-blocks. For densities around 1% the range r is too short for all positions to be stored explicitly. The superblock is divided into blocks. However, the subdivided blocks are too large to be efficiently answered by accessing the bitvector directly. Therefore, $mini$-blocks store each position explicitly. As the density increases further, the subdivided blocks become smaller and $mini$-blocks do not need to be stored explicitly as queries into subdivided blocks can be answered efficiently through accessing the bitvector. For densities larger than 3 percent only blocks and superblocks are stored. Overall, for all densities our implementation requires significantly less space than the worst case 60% overhead. Note that the distribution of the different block types depends on the density of the bitvector. In a random uniform bitvector used in the previous experiment, only a certain type of block distribution can occur as the one bits are evenly spaced over the whole bitvector. However, real data is only rarely uniformly distributed.

The space usage of our faithful implementation (SEL-CLARK) of Clark [1996] structure for real world data sets with non-uniform distribution of one bits is evaluated. Figure 3.6 shows the block distribution and overall space overhead for different real world data sets described in detail in Section 2.7.3. Each bitvector corresponds to the Huffman-shaped wavelet tree over the BWT of a given data set. This corresponds to the same wavelet tree used in an FM-Index. Note that the density of a wavelet tree is roughly 50%. For all tests files the space overhead is roughly 30%. The majority of the space is evenly distributed between superblocks, blocks and implementation overhead. For some files *long*-blocks and *mini*-blocks are also stored, but only contribute a small percentage to the overall space overhead.

3.3.3 Engineering Constant Time $Select$ on Uncompressed Bitvectors

There are two main problem of our faithful implementation of Clark's structure. First is the computational overhead during query time and the second is implementation overhead. For a given query

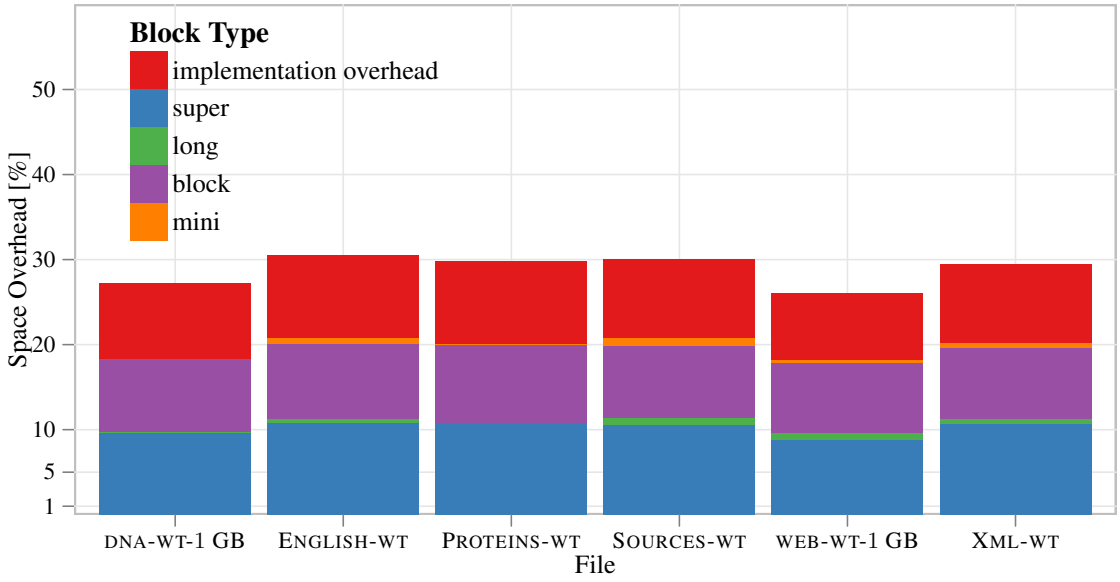

Figure 3.6: Space overhead in percent and block type distribution of constant time select of Clark [1996] of bitvectors of the wavelet tree representation of a Huffman shaped wavelet tree over the test files described in Section 2.7.3.

$select(B, i, 1)$, the size r of the superblock determines how the query can be answered. To determine if a given superblock is further divided, $\log\ r$ has to be calculated during query time. Next, if a super block is subdivided, the size r' of the correct subblock is also calculated during query time. Finally, to determine if *mini*-blocks are stored, $\log r'$ has to be calculated. These operations contribute significantly to the cost of answering a single *select* query with the structure. Further, Clark's structure is constant in a theoretical sense only due to fact that any potential scan in the original bitvector is asymptotically smaller than $\log n$ and can therefore be answered in constant time in the word-RAM model. However, in practice the upper bound of the final scan "area", $16(\log \log n)^4$, can be significantly larger than $\log n$. In both Figure 3.6 and in Figure 3.5 it can be seen that for most densities, no *mini*-blocks are used. However, they do contribute significantly to the complexity and space overhead of our implementation.

We now present an implementation of a simplified version of Clark's proposal, called SEL-C. Let $m \leq n$ be the number of ones in the bitvector B. The bitvector B is divided in *superblocks* by storing the position of every $W = 4096$-th 1-bit. Each position is stored explicitly at a cost of $\lceil \log n \rceil \leq 64$ bits. In total $\lceil \frac{n}{64} \rceil$ bits or 1.6% additional space is needed to answer *select* queries for every $W = 4096$-th 1-bit directly. Let $x = select(Wi + 1)$ and $y = select(W(i + 1) + 1)$ be the border position of the i-th superblock. A superblock is called *long*, if its size $r = y - x$ is larger or

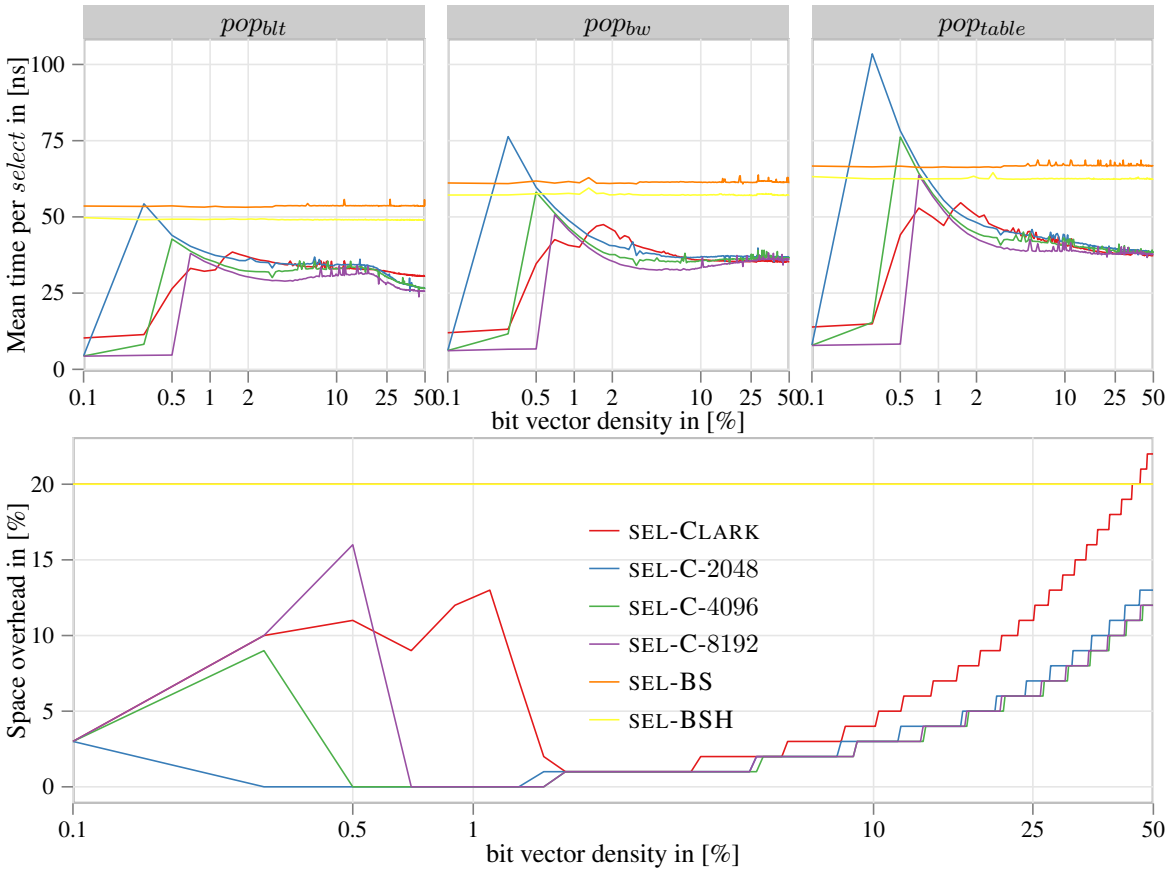

Figure 3.7: Space overhead in percent and mean time per select query in nano seconds for all uncompressed select solutions over a bitvector of size 1 *GB for densities of* 0.1% *to* 50%. SEL-C *is evaluated for* $W = 2048$, 4096 *and* 8192.

equal $\log^4 n$. As a result, storing each of the 4096 positions explicitly requires only $4096/\log^3 n$ bits per position which translates to only $\leq 10\%$ overhead per bit in a 1 GB bitvector. If $r < \log^4 n$, the superblock is classified as *short,* and it is further subdivided by storing the position of every 64-th 1-bit relative to the left border of the superblock. Within these sub-blocks, the original bitvector is scanned to calculate the final position. This requires at most $\log r \leq \log(\log^4 n) = 4 \log \log n$ bits per entry. Hence an upper bound for the space overhead is $\frac{4096}{64} \cdot \log r = 64 \cdot \log r$, which in the worst case ($r = 4096$) results in an overhead of $\frac{64 \cdot \log r}{4096} = \frac{64 \cdot 12}{4096} = 18.75\%$. For the important case of *dense bitvectors*, the typical space overhead is much smaller than the worst case, since $r \approx 2 \cdot 4096$ which translates into $\frac{64 \cdot \log(8196)}{8196} = 10.2\%$ overhead per input bit.

Time and Space Trade-offs of the Engineered Constant Time *Select* Implementation

Figure 3.7 shows the time and space efficiency of all uncompressed *select* implementations discussed above for a bitvector of size 1 GB for densities ranging from 0.1% up to 50%. Again, the mean time per query operation averaged over 100 million queries on our large test machine (LARGE) is reported. Further, SEL-C is evaluated for $W = 2048$, 4096 and 8192 while using the three different *popcnt* implementations evaluated in Section 3.1. The space usage of SEL-BS and SEL-BSH is identical but larger for all densities. Only SEL-CLARK uses roughly 5% more space than the binary search-based methods for densities at around 50% while providing much faster runtime performance. For small densities, the space utilization of all SEL-C-based methods and SEL-CLARK is interesting. For very low densities, the methods store the positions of all one bits explicitly. The space therefore increases till the threshold $\log^4 n$ is reached. This can be clearly observed in Figure 3.7. The smallest block size $W = 2048$ reaches the threshold first and but is not visible as it occurs at a density smaller than 0.1%. The larger block sizes $W = 4096$ and $W = 8192$ reach the threshold later (green and purple peak). Thus in the bottom row of Figure 3.7, one can observe the switch between block types. After the peak is hit, then the space usage decreases as not all positions are stored explicitly. For densities larger than 1%, the space usage again increases as more relative positions are stored. As calculated above, for densities around 50%, the space overhead is roughly 10%.

The running time of the different implementations is shown in the top row of Figure 3.7. Overall, the fast pop_{blt} implementation improves the running time of all implementations. The constant time implementations SEL-C and SEL-CLARK benefit more as they potentially perform long sequential scans in the original bitvector. The constant time implementations are always faster than the binary search-based methods SEL-BS and SEL-BSH. Only for small densities, the SEL-C implementations perform similarly to SEL-BS and SEL-BSH. For densities larger than 1%, both SEL-C and SEL-CLARK always outperform SEL-BS and SEL-BSH. The different SEL-C implementations perform similar. Only for small densities (less than 0.5%) the performance differs: the number of one bits within a super block is determined by W (the block size). As W increases, the range covered by the super block also becomes larger. Therefore, as W increases from 2048 to 4096 to 8192, the density threshold where the data structure switches from storing every position explicitly to resorting to storing relative positions and scanning the bitvector is different. For $W = 2048$, the space usage already decreases for density 0.1%. This implies that as the density increases to 0.5%, less positions are stored explicitly. For $W = 4096$, the space usage still increases till density 0.3%. This means that more positions are stored explicitly and only at density 0.3% does the data structure switch to storing relative positions. For $W = 8192$, this threshold is at density 0.5%. At these thresholds, the

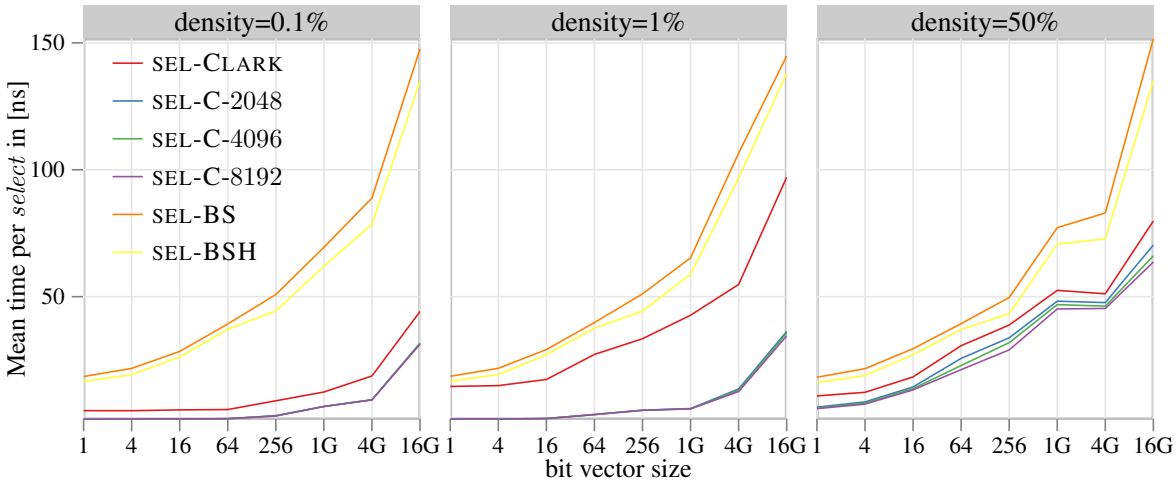

Figure 3.8: Mean time per select query in nanoseconds for all uncompressed select solutions over a bitvectors of sizes 1 *MB to* 16 *GB for densities of* 0.1%*,* 1% *and* 50%*.* SEL-C *is evaluated for* $W = 2048, 4096$ *and* 8192*.*

performance of SEL-C drastically decreases. As the switch occurs, the range covered by the super blocks is just barely smaller than $\log^4 n$. This implies that, to answer a *select* query, large portions of the bitvector have to be sequentially processed. Interestingly, the SEL-CLARK implementation does not experience this problem. The data structure is carefully designed to avoid this problem by switching between block types to cap the maximum space usage. However, as seen in Figure 3.5, at around 1% density, SEL-CLARK uses *mini* blocks, which drastically increase the space usage compared to all SEL-C, which does not use *mini* blocks. This can also be observed in Figure 3.7, where for densities around 1%, SEL-CLARK uses more space than SEL-C, but allows faster retrieval times.

To summarize, for bitvectors of size 1 GB, our reference implementation of Clark [1996] outperforms the binary search methods for all densities except 0.3%. For high density bitvectors (50%), our reference implementation uses roughly the same space as the binary search methods but is significantly faster. Our optimized SEL-C implementation is simpler, performs similar to our reference implementation of SEL-CLARK, but uses significantly less space. For high densities, the space overhead is roughly 10% compared to the 20% space overhead of SEL-BS and SEL-BSH.

Scalability of the Engineered Constant Time $Select$ Implementation

The performance of our implementations changes as the size of the used data sets increases. Figure 3.8 shows the performance of our data structures for varying data sizes. In this experiment, only pop_{blt} is used as only the performance over different bitvector sizes is of interest. bitvectors of densities 0.1%, 1% and 50% with variables sizes ranging from 1 MB to 16 GB are created. Again the mean time for 100 million operations performed on our large test machine (LARGE) is reported. For all densities and sizes the constant time solutions outperform the binary search-based implementations. For density 0.1%, SEL-C is roughly six times faster than SEL-BS and SEL-BSH for large bitvectors. For very dense bitvectors, SEL-C is only two times faster. SEL-C always outperforms SEL-CLARK. Interestingly, SEL-CLARK performs worse for bitvectorswith density 1%. This can be explained by looking at the block type distribution graph shown in Figure 3.5. For bitvectors of density 1%, SEL-CLARK uses many *mini*-blocks. These blocks are stored in compressed form which requires more computational overhead to answer $select$ queries. The hinted binary search implementation (SEL-BSH) performs significantly better than SEL-BS as the size of the bitvector increases.

3.3.4 Comparing different $Select$ Implementations

Finally a similar experiment as discussed above is performed, but, we now measure the effect of hugepages (HP) and our SSE enhancements on the overall performance of our engineered $select$ implementation, SEL-C, as well as SEL-BS and SEL-BSH. We choose not to include SEL-CLARK in the following experiments as SEL-C always outperforms SEL-CLARK. In addition two further $select(B, i, 1)$ implementations called SELECT9 and SIMPLE proposed by Vigna [2008] are evaluated to provide a better understanding how our data structures compare to other state-of-the-art proposals. The implementations are referred to as SEL-V9 and SEL-VS. SEL-V9 builds select capabilities on top of RANK-V. In addition to the $rank$ counts, a two level inventory is stored at a total cost of 57.5% space overhead. SEL-VS similar to SEL-V9, uses a multi-level inventory to perform select but does not require RANK-V during query time which results in less space-overhead. Similar to the select structure of Clark [1996], a three-level structure is used in SEL-VS. However, SEL-VS is algorithmically engineered to use only 64-bit and 16-bit words instead of using integers of size $\log r$ which are more expensive to access and store. For very large superblocks, SEL-VS further stores the absolute positions of each set bit using a 64-bit word, whereas Clark's structure stores the position in $\log r$ bits relative to the beginning of the superblock. For smaller superblocks, the positions are not stored explicitly. Instead, two inventories are used to find a region in the original bitvector which is then scanned sequentially to find the correct i-th bit position.

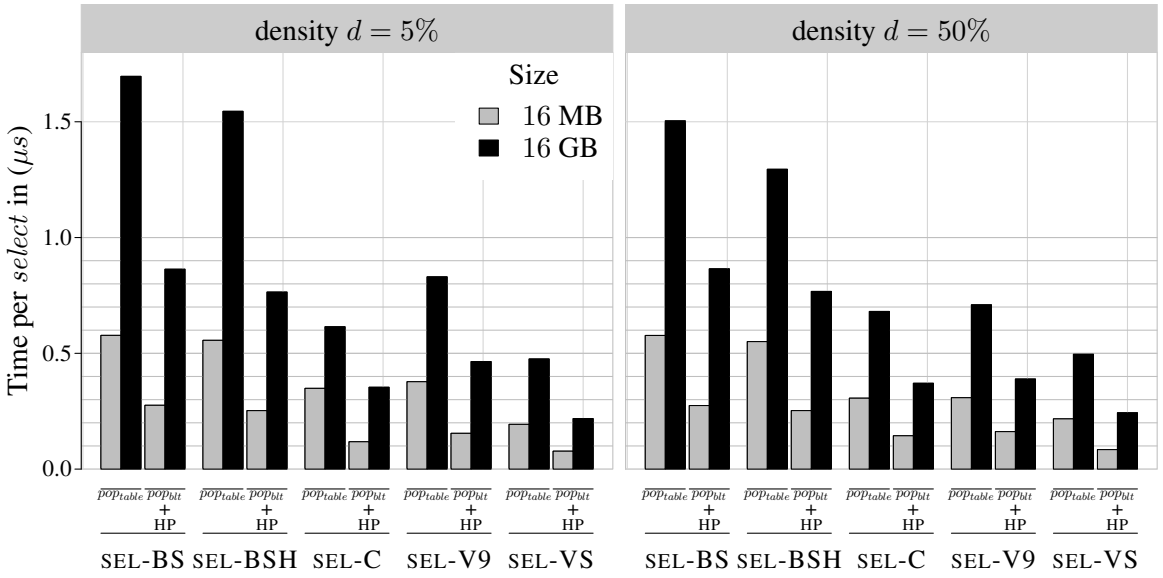

Figure 3.9: Average time for a single select operation on uncompressed bitvectors dependent on the size (16 MB/16 GB), the density (5/50) of the input, the implementation (SEL-BS/SEL-BSH/SEL-C-4096/SEL-V9/SEL-VS) and the used environmental features (pop_{table}/pop_{blt}+HP).

Address Translation Cost

Address translation can have an effect on the performance of *select* data structures. To elucidate the effect of address translation, two bitvectors of size 16 MB and 16 GB are created, and the time for one random *select* query over bitvectors of densities 5% and 50% is evaluated. This is in line with densities $(5, 20, 50)$ chosen by previous studies such as González et al. [2005]. For each density 100 million select queries are performed and the mean time per query as well as the mean number of TLB and L1 cache misses per *select* query is measured. Only timing results are shown in Figure 3.9. In Table 3.3 the mean number of cache and TLB misses per *select* query caused by the different implementations for densities 5%, 20% and 50% are shown.

For the large data set and 5% density, the constant time select method SEL-C is roughly 60% faster than fast as the cache-friendly binary search select implementation SEL-BSH and twice as fast as the unoptimized SEL-BS implementation. For the small data set at the same density, SEL-C is twice as fast as both SEL-BSH and SEL-BS. For 50% density SEL-C is twice as fast than both SEL-BS and SEL-BSH for the small and large data set. To conclude, for all data sets and all densities, constant time select (SEL-C) always outperform both binary search select implementations. Our cache-friendly implementation (SEL-BSH) generally outperforms the non optimized SEL-BS implementation by 10 to 20 percent for larger data sets. For small data sets, no effect can be seen as

the cache-friendly "hinting" has no effect. For all 16 MB test instances, the bitvector and the support structure fit in one 1 GB hugepage. Therefore, all structures benefit from enabling the features (pop_{blt}+HP). All structures become roughly twice as fast for the small test case. Overall, the two-level structure SEL-C outperforms SEL-BS, SEL-BSH and SEL-V9 for small instances, but is slower than SEL-VS. Further note that the performance of SEL-BSH and SEL-BS is almost identical as address translation for small instances does not significantly affect runtime performance (see our analysis of $rank$ for details). For the 16 GB instances, all run times increase due to the increasing cost of TLB and cache misses as described in the discussion of our $rank$ experiment in Section 3.2. As expected SEL-BSH outperforms SEL-BS since the first 10 memory accesses cause no TLB miss. However, the unoptimized implementation (pop_{table}+ 4 kB pages) of our 2-level SEL-C approach outperforms both binary search approaches even with SSE+ HP enabled. Without the features, SEL-C is slightly faster than SEL-V9 for $d = 5\%$ and roughly the same speed for $d = 50\%$. For all instances SEL-VS outperforms our new two-level approach SEL-C.

L1 cache and TLB performance

Cache performance of all $select$ implementations in our experiments are shown in Table 3.3. Note that the cache and TLB performance of all $select$ implementations is up to two times worse than the equivalent $rank$ implementations shown in Table 3.2. Overall SEL-C is always more cache-efficient than both binary search implementations for all data sets. Our cache friendly binary search implementation (SEL-BSH) shows significantly better L1 cache and TLB miss performance. For the large data set, SEL-BS causes, on average, 25.6 TLB misses using 4 kB pages. For a bitvector of size 16 GB there are $33,554,432$ superblocks. Performing binary search over the superblocks takes $\log(33,554,432) = 26$ steps. However, as a page is 4 kB, in the "final" steps of the binary search only "jump" inside a page, not causing any additional TLB misses. One additional TLB miss is caused by accessing the bitvector to perform the final scan inside the block. This explains the average number of 25.6 misses for 4 kB pages. SEL-BSH causes 10.3 TLB misses on average. 1024 $rank$ samples are stored in heap order to reduce the number of cache misses during binary search. These samples can be used to perform binary search over the superblocks while not causing TLB misses in the first 10 steps. Therefore, only 16 accesses to the superblock vector are required, which explains the reduction in total TLB misses for SEL-BSH. Hugepages (HP) also affect the performance of SEL-BS and SEL-BSH. With 1 GB pages, the complete superblock array fits into at most two pages causing at most 2 TLB misses during the binary search steps. Next, we additionally access the bitvector to perform the sequential scan which can also at most two additional TLB miss (in the worst case a scan

	TLB misses / L1 cache misses per operation				overhead in %	
	SEL-BS		SEL-BSH		SEL-BS	SEL-BSH
	pop_{table}	pop_{blt}+HP	pop_{table}	pop_{blt}+HP		
density $d = 5$						
16 MB	1.2 / 13.8	0.0 / 12.5	1.3 / 10.9	0.0 / 10.1	19	19
16 GB	25.6 / 67.9	7.4 / 68.7	10.3 / 43.3	5.9 / 40.3	20	19
density $d = 20$						
16 MB	1.2 / 14.2	0.0 / 13.8	1.3 / 11.1	0.0 / 10.9	19	19
16 GB	25.6 / 68.4	7.4 / 68.8	10.3 / 43.5	6.0 / 40.2	20	20
density $d = 50$						
16 MB	1.3 / 13.8	0.1 / 13.2	1.5 / 10.6	0.1 / 10.6	19	19
16 GB	25.7 / 68.0	7.5 / 69.0	10.6 / 44.0	6.6 / 40.6	20	20
	SEL-C		SEL-VS		SEL-C	SEL-VS
	pop_{table}	pop_{blt}+HP	pop_{table}	pop_{blt}+HP		
density $d = 5$						
16 MB	0.9 / 7.9	0.0 / 7.5	1.3 / 4.3	0.0 / 3.6	2	11
16 GB	4.9 / 13.7	3.9 / 10.2	3.0 / 7.8	2.8 / 3.6	2	11
density $d = 20$						
16 MB	1.3 / 7.3	0.0 / 6.7	1.4 / 4.5	0.0 / 3.7	5	11
16 GB	5.0 / 13.0	4.4 / 8.9	3.0 / 8.1	2.9 / 3.8	5	11
density $d = 50$						
16 MB	2.1 / 6.9	0.0 / 5.7	1.2 / 5.2	0.0 / 4.3	12	7
16 GB	5.2 / 12.9	4.9 / 8.1	3.0 / 8.8	2.8 / 4.4	12	7

*Table 3.3: Average number of TLB misses/L1 cache misses per select operation dependent on implementation (*SEL-BS*/*SEL-C-4096*/*SEL-CLARK*), used* $rank_{64}$ *method (*pop_{table}*/*pop_{blt}*) and activation of* 1 *GB pages (*HP*) on bitvectors of different sizes (*16 *MB/*16 *GB) and densities (*5*/*20*/*50*).*

over a page boundary causes a second TLB miss). When performing $select_{64}$, an additional lookup table is accessed to process the last byte which can also cause an additional TLB miss. Therefore, the number of 7 TLB misses, on average, seems reasonable. SEL-BSH only causes 2.7 TLB misses on average. First the samples are accessed. Next the superblock and the bitvector are accessed which can also cause additional TLB misses. Note that our analysis does not completely explain the measured number of TLB misses. However, this could be caused by the fact that there are only 4 TLB entries for hugepages available in the system (see Section 2.7.1). Our program could further access memory

regions outside of the hugepage mapped memory area which could cause normal TLB misses which are also counted in our measurements. However, our analysis only tries to explain the "order of magnitude" of TLB misses measured. SEL-VS uses only byte aligned accesses to words of size 16 or 64 bits. SEL-C stores positions bit-compressed. This results in slower runtime performance and an increased number of L1 cache misses.

Table 3.3 further shows the space overhead required for each select structure. The binary search methods use 20% overhead. The size of the "hints" in SEL-BSH is only 8 kB and is therefore negligible. For densities $d = 5$ and $d = 20$ SEL-C is substantially smaller than SEL-VS. However, for $d = 50$ SEL-VS is smaller than SEL-C.

Comparing different *Select* Implementations on Real World Data

As suggested by Vigna [2008], it is important to evaluate the performance of select structures on unevenly distributed bitvectors. We therefore evaluate the performance of all select structures discussed above on several real world data sets. The "real world" bitvector instances are extracted from the Huffman-shaped wavelet tree as described in Section 2.7.1. Figure 3.10 shows the time-space trade-offs of each structure for all test instances. We contrast the mean time per $select$ operation over 10 million random queries with the space overhead of each structure in percent of the original bitvector. Note that the real world data sets roughly have about 50% density as they represent a Huffman-shaped wavelet tree. Another property is that they contain long runs as well as evenly distributed regions. Long runs occur frequently in the bitvectors of human generated texts, which contains words and are structured. We observe in Figure 3.10 that overall SEL-BS and SEL-BSH are not competitive. SEL-V9 is faster than the binary search methods but requires 57% overhead. SEL-VS is always the fastest method but the space usage varies between different data sets. For WEB-WT-1GB, SEL-VS uses 35% overhead whereas SEL-C uses only 13%. The same behavior can be observed for SOURCES-WT-200MB, ENGLISH-WT-200MB and DBLP.XML-WT-200MB. On the one hand, for bitvectors with few runs (PROTEINS-WT-200MB, DNA-WT-200MB,DNA-WT-1GB) the space usage of SEL-VS is about the same as for evenly distributed bitvectors and therefore less than SEL-C.

The performance of $select_{64}$ on run time of $select$ on bitvectorsis also of independent interest. For all evaluated $select$ structures, only one $select_{64}$ operation is performed to determine the position of the i-th bit in a target word x. Before this step, potentially many other operations such as $popcnt$s in a sequential scan or binary search are performed to determine x. We measure the effect of using a slow $select_{64}$ method (sel_{table}) compared the fastest method (sel_{blt}) of the target word on the overall

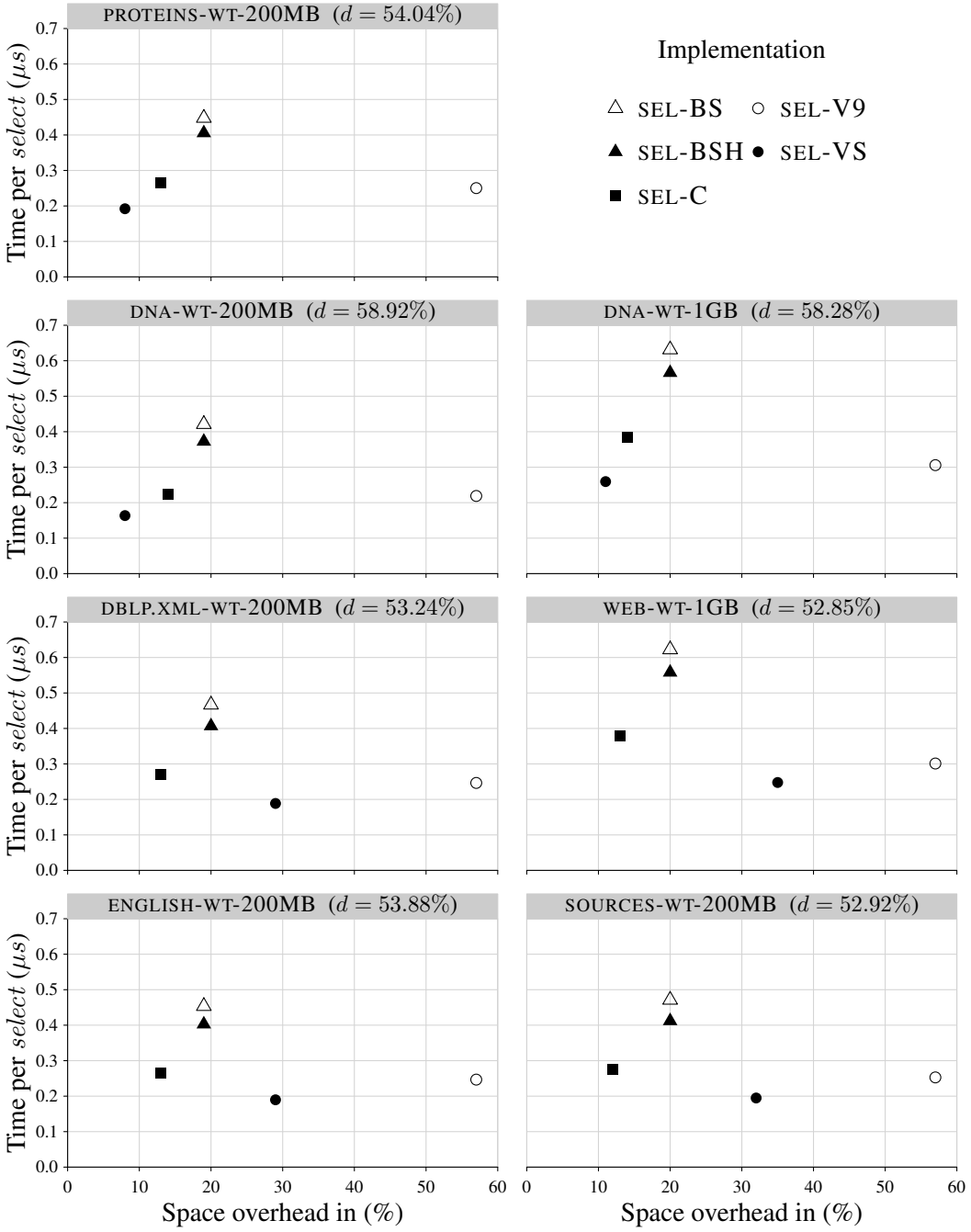

Figure 3.10: Time-space trade-offs for a single select operation on uncompressed bitvectors for different "real world" data sets of the implementations SEL-BS, SEL-BSH, SEL-C-4096, SEL-V9 *and* SEL-VS. *For all implementations the features* pop_{blt}+HP *were used.*

performance of the structure. For small instances, the running time for SEL-C and SEL-VS decreases by 10 to 20%, for the binary search structures it decreased by 7%. For large instances, the running time for SEL-C and SEL-VS decreases only by 2% to 5%. The binary search solutions do not benefit from improved $select_{64}$ on large instances as the running time is dominated by binary search and the resulting TLB misses. Interestingly, the running time for SEL-V9 decreases by 30% for all instances while all other $select$ structures improve mostly for small instances.

Construction cost of a $select$ structure should also be considered when evaluating different data structures. We use both $popcnt$ and $select_{64}$ methods during the construction of our SEL-C structure. We observed an improvement in construction cost by up to an order of magnitude (up to 30 times faster for the random bitvectors used in our experiments) compared to versions with simple bit-by-bit processing during construction.

To summarize, SEL-C and SEL-CLARK are more TLB and cache efficient than the commonly used binary search alternatives. This is not noticeable for small data sets but for large data sets, the constant time solutions are significantly more cache and TLB friendly than the binary search alternatives. This can be directly observed in the overall performance of the constant select implementations discussed in this chapter. Both SEL-C and SEL-CLARK perform similar to state-of-the-art engineered select data structures while using less space and having much faster construction time. Especially the hugepage feature available in many modern operating systems can lead to significant performance improvements.

3.4 Optimizing $Rank$ and $Select$ on Compressed Bitvectors

To save space, succinct data structures often use compressed bitvector representations instead of uncompressed bitvectors. In this section we try to improve the H_0 compressed bitvector representation of Raman et al. [2002] (RRR) discussed in detail in Section 2.2.4. Here we specifically focus on a proposal by Navarro and Providel [2012]. First we review the idea of on-the-fly encoding and decoding of blocks. Next we discuss practical improvements to the on-the-fly encoding and decoding scheme which we later evaluate in an extensive evaluation. In our experiments we show: (1) how the different components of RRR contribute to the size of the bitvector (2) how many blocks are affected by our optimizations and (3) how our optimizations affect the performance of $rank$, $select$ and $access$ on the compressed bitvector representation.

Recall that bitvectors are split into blocks of size K. The blocks are then represented in succinct form using two components C and O. Additionally, samples to allow efficient $rank$ and $select$ over the compressed bitvector are stored in a third class S. Previously the block size was constrained

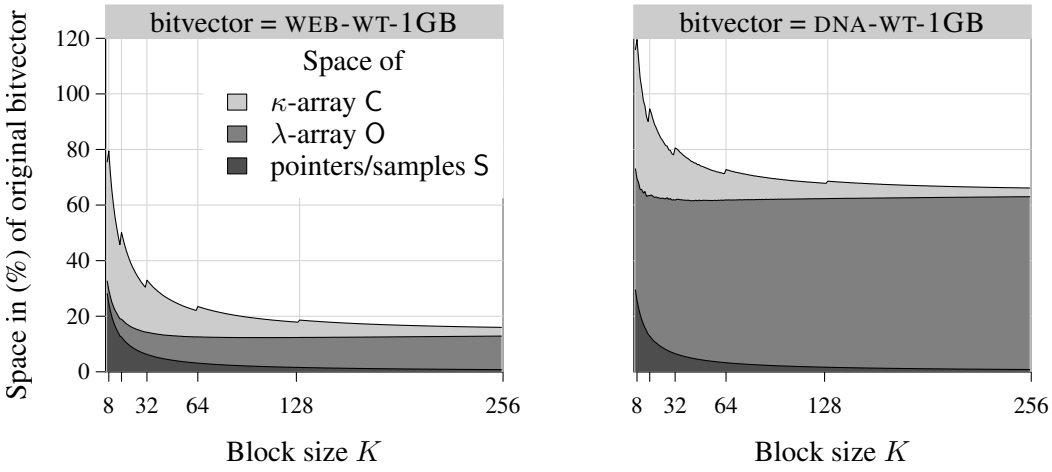

Figure 3.11: Space consumption of the three different parts of the H_0-compressed bitvector as a function of block size K for two different bitvectors. The sample rate t was set to 32. The original bitvectors of size 1 GB were extracted from wavelet trees of WEB *and* DNA *text.*

to $K \leq 15$ by the size of the lookup-table required for encoding and decoding individual blocks. Navarro and Providel [2012] propose an on-the-fly decoding scheme based on Pascal's triangle which does not require the lookup-table, and thus allows for the use of larger block sizes In their initial proposal, Navarro and Providel use block sizes of up to $K = 64$. In this section we first evaluate the effect of K on the compressibility of the different class types which to our knowledge has not been done before. We use the results of our evaluation as a motivation to provide an implementation which uses block sizes K up to 255 bits. We further provide several implementation tricks which in turn translate to significant performance improvements in practice.

The total space of O is bounded by about $nH_0 + \frac{n}{K}$ bits (see [Pagh, 1999]). Bitvector C is bounded by $n\frac{\log K}{K}$ bits, and bitvector S by $2n\frac{\log n}{tK}$ bits. If the bitvector is compressible – that is $H_0 \ll 1$ – then the size of C is dominant if K is small. Figure 3.11 shows the space usage of the individual components (C,O,S) for two, "real world" bitvectors WEB-WT-1GB and DNA-WT-1GB. For WEB-WT-1GB, which is highly compressible, a block size of $K = 15$ compresses the up to 50% of size of the original bitvector. Here the samples S, contribute and the block class type C contribute significantly towards the overall space usage of the compressed representation. For $K = 64$, the space used by the samples S, is proportionally small compared to the class type C and the offsets O. However, the class types still significantly contributes to the overall space usage. For $K = 255$, the majority of the space is used by the offsets (O). The second data set, DNA-WT-1GB, is not very compressible. Therefore, increasing the block size K beyond 64 does not significantly reduce the

overall size of the compressed representation. However, for larger block sizes the contributions of C and S are not significant. Overall, for compressible data, it is advantageous to increase the block size of H_0 compressed bitvectors beyond the limits explored in previous work.

On-the-fly Encoding and Decoding

The implementation of Claude and Navarro [2008] uses blocks of size $K = 15$ and decodes b_i (block i at position $i \times K$) from (κ_i, λ_i) (the class type κ_i of the block and the offset λ_i within the block) via a lookup table. Unfortunately, for larger K lookup tables are not practical. Navarro and Providel [2012] recently proposed a solution to overcome this obstacle by spending more time on the decoding process: they encode and decode blocks in $\mathcal{O}(K)$ time on-the-fly. During the encoding process λ_i is computed from a block b_i with κ_i set bits as follows: initially $\lambda = 0$. First, the least significant bit in b_i is considered. There are $\binom{K-1}{\kappa_i}$ blocks ending on zero and $\binom{K-1}{\kappa_i-1}$ ending on one. If the last bit is one, λ_i is increased by $\binom{K-1}{\kappa}$ (that is the number of blocks of ending with zero); otherwise λ_i is not changed. In the next step, K is decreased by one, and κ_i is decreased by one if the last bit was set. b_i is shifted to the right and we reevaluate the least significant bit. The process ends when $\kappa_i = 0$. This is shown in detail in Figure 3.12 where we encode $b_i =$`0101011` with $K = 7$ and $k_i = 4$. We start with $\binom{K}{k_i} = \binom{7}{4}$ and process the least significant bit which is 1. As we are processing a one bit, we add $\binom{K-1}{k_i} = \binom{6}{4} = 15$ to λ_i. After every step we decrease K by one. If we processed a one bit we also decrease k_i as there is one less one left to process. Next we process the second last one bit. We add $\binom{K-1}{k_i} = \binom{5}{3} = 10$ to λ_i and again decrease both K and k_i. Next we process the first 0 bit which implies we only decrease K. Next we process the next 1 bit by adding $\binom{3}{2} = 3$ to λ_i and again decreasing both K and k_i. We process the next 0 bit by only decreasing K. Finally we process the last 1 bit by adding $\binom{1}{1} = 1$ to λ_i. We decrease k_i which is now 0 and we finish the encoding process in $\mathcal{O}(K)$ time. Therefore, $b_i =$`0101011` is encoded as $k_i = 4$ and $\lambda_i = 15 + 10 + 3 + 1 = 29$. In Figure 3.12, we "walk" the encoding process is visualized as the red path in Pascal's triangle. At each step we decrease K by one. We walk "left" by decreasing k_i each time we process a one bit. If a one bit is processed we add the "blue" coefficients to λ_i.

A block b_i can be recovered from λ_i and κ_i as follows: if $\lambda_i \geq \binom{K-1}{\kappa_i}$, then the least significant bit in b was set. In this case λ_i is decreased by $\binom{K-1}{\kappa_i}$ and κ_i by one. Otherwise the least significant bit was a zero. In the next step, K is decreased by one. The decoding process ends when $\kappa_i = 0$. In Figure 3.12, this would imply "walking" the same path as in the encoding process, deciding at each step, if we processed a one bit or zero bit based on the value of λ_i and $\binom{K-1}{\kappa_i}$. On-the-fly decoding requires only $\mathcal{O}(K)$ simple arithmetic operations (subtraction, shifts and comparisons) and a lookup

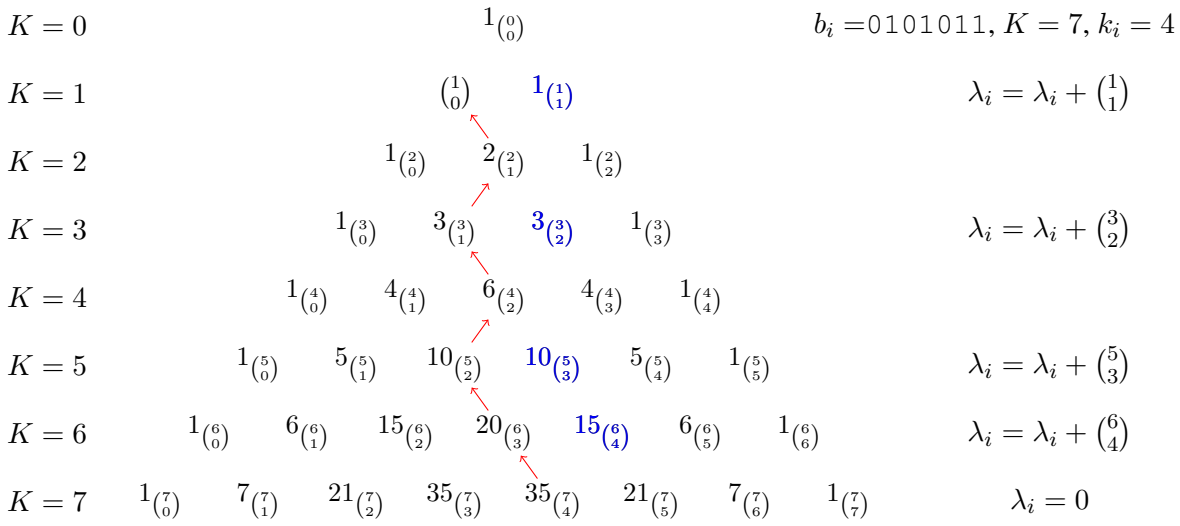

Figure 3.12: On-the-fly encoding of λ_i by walking Pascal's triangle for b_i =0101011, $K = 7$, $k_i = 4$. We start the encoding process at $\binom{K}{k_i} = \binom{7}{4}$. Each time we process a one bit, we decrease k_i by one and add $\binom{K-1}{k_i}$ to λ_i. Every step we decrease K by one. The process stops once $k_i = 0$.

table of size K^2 to calculate the binomial coefficients. Navarro and Providel [2012] use block size of $K = 63$, which reduces the size of C to 98 MB. This is still as big as C (97 MB) for WEB-WT-1GB, and it was reported that the use of $K > 63$ results in runtimes orders of magnitudes slower than for smaller K [Navarro and Providel, 2012].

Practical Optimizations to On-the-fly Decoding

On-the-fly decoding requires only $\mathcal{O}(K)$ time to recover a block b_i from k_i and λ_i. However, this process can be improved by applying the following optimizations.

First, the *access* operation can immediately return the result if block b_i contains only zeros or ones. We call this block *uniform*. A *uniform* block during the decoding process will either return $b_i = 0^K$ or $b_i = 1^K$. Therefore, depending on k_i, we can immediately return the correct result (a zero or one bit) for *access* without scanning O or accessing S. The *rank* operation can immediately return its result if the $(\tilde{i}+1)$-th and $\tilde{i}$-th rank sample in S differ by 0 or tK, saving the scanning of C. At first glance, this optimization seems to be trivial, however in text indexing the bitvectors of wavelet trees often contain long runs of zero and ones and we therefore conjecture that this optimization will be effective in practice. Figure 3.13 shows the percentage of *uniform* blocks for variable block sizes K over bitvectors of wavelet trees over BWT of several real world data sets described in Section 2.7.3.

For WEB-WT-1 GB up to 90% of the blocks in the compressed bitvector are uniform. With our optimization we will therefore almost never have to access the offset array O to decode a block. For other files the percentage of affected block ranges from 30% to 80%. This can be explained as follows. The BWT over a given sequence S tends to group symbols with similar context in S together. This results in long runs of identical symbols in the BWT. These runs are then "translated" into runs in the wavelet tree representation of the BWT. At each level, the runs are mapped to the same subtree, which again results in a run of zero or one bits in the compressed bitvector representation. Therefore, our optimization is especially useful when using B_{RRR} in the context of succinct text indexes, where generally a wavelet tree over the BWT is used to represent a sequence S. However, sequences without long runs of zeros or ones will not exhibit the same percentage of *uniform* blocks and will thus not benefit from this optimization.

Second, if a block b_i has to be decoded, than we can apply an optimization if the block contains only few ones: we first check if $\kappa_i \leq \frac{K}{\log K}$. That is, if the block is *sparse*. Especially for large block sizes K, it is faster to determine the κ_i positions of ones in b_i by performing κ_i binary searches over the columns in Pascal's triangle instead of performing a complete sequential scan. Figure 3.13 shows the percentage of *sparse* blocks for variable block sizes K over bitvectors of wavelet trees over the BWT of several real world data sets described in Section 2.7.3. As there are very long runs of *uniform* blocks in the shown data sets, the number of *sparse* blocks is less substantial. We therefore suspect our second optimization to be less effective than our first. However, for data sets without long runs, we suspect there to be more *sparse* blocks.

Experimental Evaluation

Recall that larger block sizes K for the H_0-compressed bitvector representation can lead to significant space savings, as shown in Figure 3.11. In this section we explore how the runtime performance of H_0-compressed bitvectors. First we evaluate the impact of our optimizations on the performance of $rank$ and $access$. Figure 3.14 shows the improvements in running time for both operations for WEB-WT-1 GB and DNA-WT-1 GB analysed in Figure 3.13. Recall that for WEB, almost 90% of the blocks are *uniform*. Therefore, the first optimization on *uniform* blocks described above is very effective. This can be clearly observed in Figure 3.14, as the optimized version of both $rank$ and $select$ is roughly twice as fast. Figure 3.13 shows that DNA-WT-1 GB contains fewer blocks that are affected by our optimizations. This can be clearly seen in Figure 3.14, as the optimized version is only slightly faster. The second proposed optimization – decoding with binary searching Pascal's triangle if only few set bits are in the block – only improved the runtime for the random bitvectors of

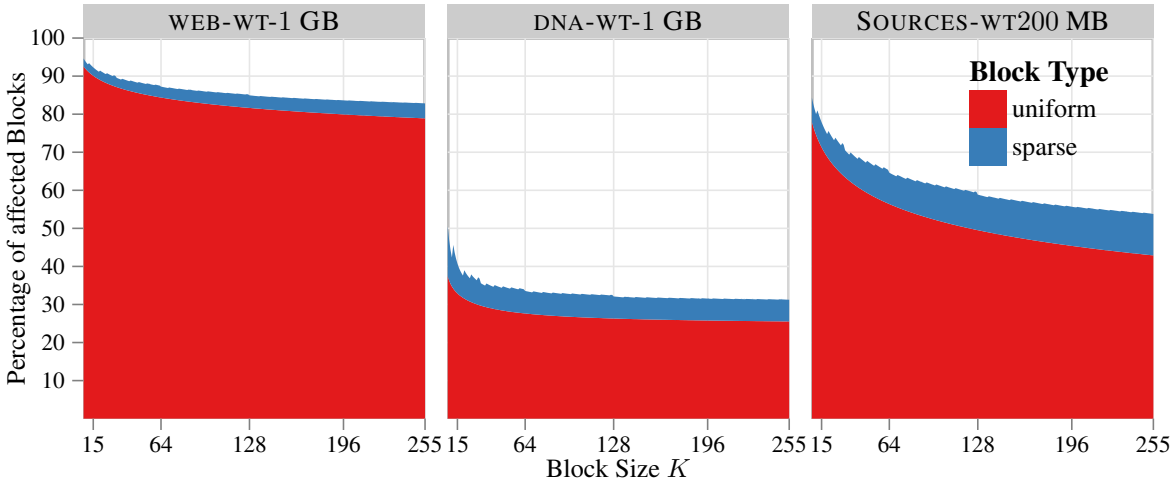

Figure 3.13: Percentage of blocks in a H_0 compressed bitvector representation which can be optimized using technique one (uniform blocks) and technique two (sparse blocks) for wavelet trees of files WEB, DNA *and* SOURCES *described in Section 2.7.3.*

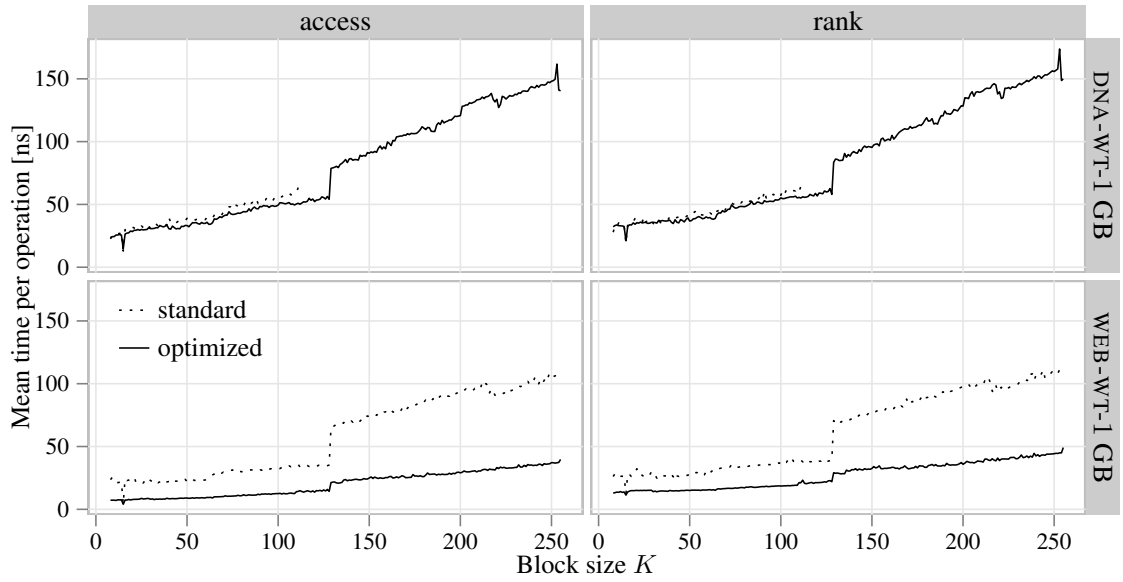

Figure 3.14: Run time improvements of operations rank and access caused by the proposed optimizations on H_0 compressed bitvectors with long repetitions using variable block sizes K.

densities $d \leq 5\%$. The runtime was reduced by 10% for $d = 5\%$ and 50% for $d = 1\%$.

Next we evaluate how the performance of all operations is affected by the choice of K. In our implementation of B_{RRR} we used built-in 64- and 128-bit integers for block sizes $K \leq 64$ and $K \leq 128$, the latter with SSE. For $K \geq 129$ we used our own tailored class for 256-bit integers. Note

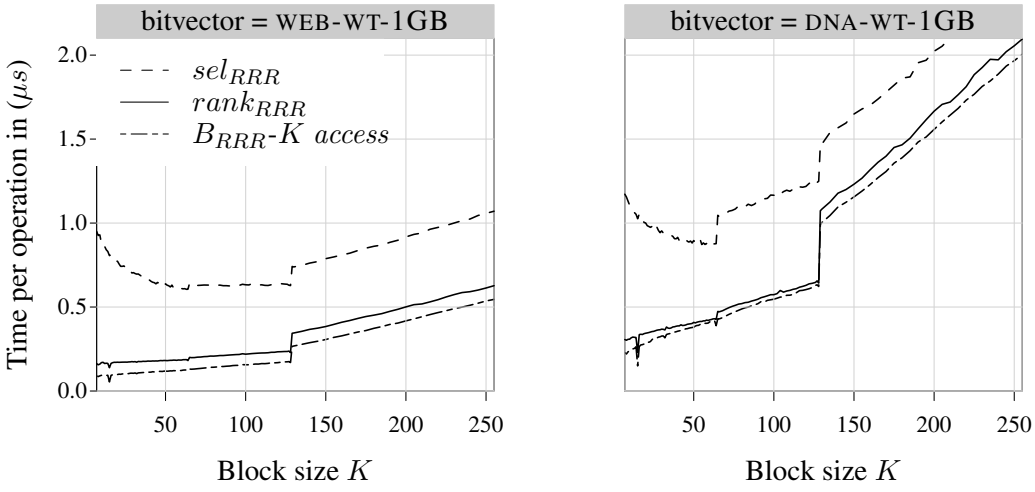

Figure 3.15: Query times for the operations on the H_0-compressed bitvector as function of block size K. The sample rate t was set to 32.

that the size of the lookup tables for Pascal's triangle for the on-the-fly decoding is therefore 32 kB, 256 kB and 2 MB for the three different integer types. In the special case of $K = 15$ we do not use on-the-fly decoding but use on access to a lookup table of size 64 kB to get the 15-bit block instead. In this case we also use broadword computing to calculate the sum of multiple block types $\kappa_i s$. The latter can be done, since the $\kappa_i s$ are stored in nibbles and we can applying line $6-7$ of Algorithm 3.1 and get the sum as the last byte of b then. For all other block sizes we sum up each κ_i individually and use the on-the-fly decoding in the last block $b_{i'}$.

Figure 3.15 depicts the resulting runtime for the three operations $access$, $rank$, and $select$ as function of K for the bitvectors WEB-WT-1GB and DNA-WT-1GB. We first concentrate on the operations $access$ and $rank$. Note that the specialized implementation for $K = 15$ can be recognized clearly: it is about twice as fast as the on-the-fly decompression with comparable block sizes.

We can observe that the runtime of the on-thy-fly decoding version linearly depends on the block size. The constant of the linear correlation is determined by two factors: the used integer type and the structure of the original bitvector, where the latter has a stronger impact than the former. The reason is the number of uniform blocks in WEB-WT-1GB is much larger than in DNA-WT-1GB as shown in Figure 3.13. Thus, for WEB-WT-1GB we decode significantly less blocks than in DNA-WT-1GB which leads to overall faster $rank$ and $access$ operations. The time for $select$, which is implemented using binary search, slightly decreases when we increase the block size and native arithmetic is used. For $K \geq 64$ the decoding costs become too dominant and we can not observe the reduced time for the binary search on less rank samples any more. The effect of the integer type can be clearly observed

by the transitions from using 64- or 128-bit integers to the own tailored 256-bit integers which also have to use a lookup table which does not fit in the L2 cache.

Overall, the run time the runtime of the on-thy-fly decoding version linearly depends on the block size K. The performance of H_0 compressed bitvectors directly correlated with the compressibility of the data set. More runs in the BWT of a text allow the text to be compressed more efficiently. As shown in Figure 3.15, the query time can also be greatly reduced on compressible texts using our proposed optimizations.

3.5 Optimizing Wavelet Trees

Especially in succinct text indexes, operations $rank$ and $select$ are performed not on binary alphabets over a bitvector but over a sequence of symbols with a larger alphabet. A wavelet tree (described in detail in Section 2.3) is commonly used to decompose performing $rank$ and $select$ over sequences of larger alphabets to performing $rank$ and $select$ on multiple bitvectors. Thus wavelet trees profit from the optimizations on compressed and uncompressed bitvectors discussed in this chapter.

In this section we describe two additional optimization techniques for wavelet trees. First we discuss cache-efficient processing of wavelet trees while performing $rank$ in the context of backwards search. Second, we discuss bit-parallel wavelet tree construction which outperforms traditional bit-by-bit construction by up to 150 times.

3.5.1 Cache-Aware Wavelet Tree Processing

In the context of creating cache-friendly succinct data structures, performing $rank$ operations on wavelet trees can often also be performed in a cache-aware way. Consider a regular COUNT query in a FM-Index discussed in Section 2.5.2 in the context of backwards search. During each step, one symbol of the pattern is processed by performing two $rank$ operations on the wavelet tree of the BWT. Interestingly, both $rank$ operations calculate the $rank$ of the same symbol c, before and after the current matching range $\langle sp, ep \rangle$. Therefore, both $rank$ operations, in each processing step follow the same path in the wavelet tree. The range $\langle sp, ep \rangle$ generally also decreases with each processing step of the pattern. Consequently, the two $rank$ operations can thus be expected to occur closer to each other as the pattern is processed. Using a wavelet tree in a FM-Index can be performed cache-aware by performing the two $rank$ operations at each level of the wavelet tree at the same time. In the lower levels of the wavelet tree, it is expected that during both binary $rank$ operations, superblocks close to each other are accessed. Therefore, performing both $rank$ operations simultaneously can lead to increased COUNT performance due to cache effects.

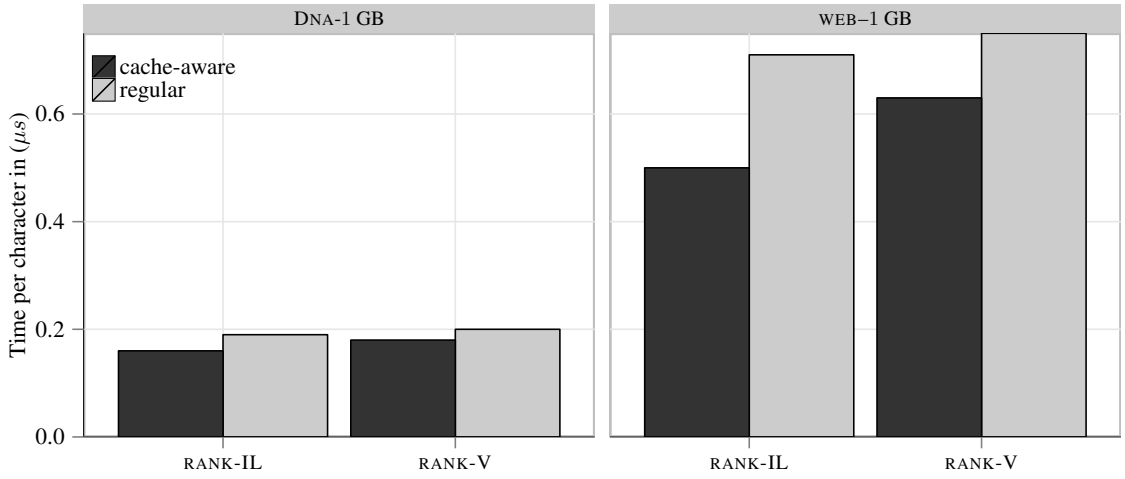

Figure 3.16: Wavelet tree construction time for two data sets. Run-aware construction utilizing SSE *is compared to bit-by-bit wavelet tree construction in* LIBCDS.

To test this assumption, both regular wavelet tree processing and cache-aware processing are compared by replicating the COUNT experiment in the experimental study of Ferragina et al. [2008]. Overall $50,000$ count queries of length 20 are performed. The mean time taken to process one character in a query is shown in Figure 3.16. The FM-Index is parametrized using a Huffman-shaped wavelet tree and the cache friendly bitvector representations RANK-V and RANK-IL which were evaluated in Section 3.2. The performance for the DNA and WEB data sets of size 1 GB is evaluated. Interestingly, the cache effects are only significant for the WEB data set, whereas the DNA data set shows no run time improvements. This can be explained as follows. Random patterns of length 20 occur more frequently in the DNA data set. The total number of occurrences for all patterns is 46 times larger for the DNA data set compared to the WEB data set. Therefore, the $\langle sp, ep \rangle$ ranges of each pattern are larger throughout the backward search steps of each pattern. This results in more cache misses even when both $rank$ operations are performed at the same time as the ranges are to large to produce a cache effect during simultaneous $rank$ processing. For the WEB data set and RANK-IL, the run time performance increases by 30% whereas the RANK-V-based wavelet tree is only 10% faster in the cache-aware setting. Overall, if patterns do not occur often in the text, performing cache-aware processing of the wavelet tree can lead to substantial gain in run time performance during pattern matching.

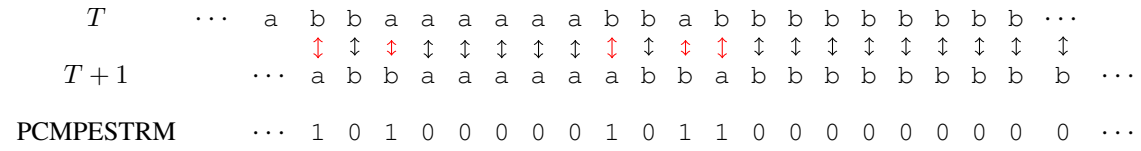

Figure 3.17: SSE*-enabled wavelet tree run detection using the* SSE4.2 *instruction* `PCMPESTRM` *to detect runs in the text by shifting the text by one symbol and comparing the shifted string* $T+1$ *to the original string* T*.*

3.5.2 Wavelet Tree Construction

Wavelet trees need to be constructed as part of the creation of most succinct text index representations. Performing construction efficiently is therefore important. Regular bit-by-bit wavelet tree construction used in popular compressed data structure libraries such as LIBCDS can take longer than constructing the BWT [2]. In contrast, efficient wavelet tree construction such as the run-aware version proposed by Simon Gog in his SDSL library[3] is up to several orders of magnitudes faster than bit-by-bit processing [4]. Thus, the construction of the BWT is the major bottle neck during succinct text index construction, whereas wavelet tree construction can be performed efficiently. Here we will briefly discuss the efficient wavelet tree construction algorithm used in the SDSL library. We then show how using SSE instructions can further decrease the construction time of wavelet trees by up to 30%.

Wavelet trees used in succinct text indexes are mostly built over the BWT of a given text. If the text is compressible, the BWT tends to contain long runs of symbols due to the their similar context. Therefore, instead of processing each symbol individually, processing runs of symbols during wavelet tree construction can improve the overall construction cost. Detecting runs efficiently can significantly improve the run time performance of wavelet tree construction. Instead of keeping track of the preceding symbol while processing the text we utilize *parallel string comparison* instructions available in SSE 4.2.

To detect runs in the BWT efficiently, we use the parallel string comparison instruction `PCMPESTRM` which can be configured to return a bitvector indicating character mismatches. We compare the original string T with the string $T+1$ which corresponds to T shifted one symbol to the right. The `PCMPESTRM`[5] instruction then returns a bitvector indicating, with a set bit, the beginning of runs

[2] see later experiments in this section
[3] `https://github.com/simongog/sdsl`
[4] personal communication with Simon Gog
[5] See Intel SSE4 Programming Reference

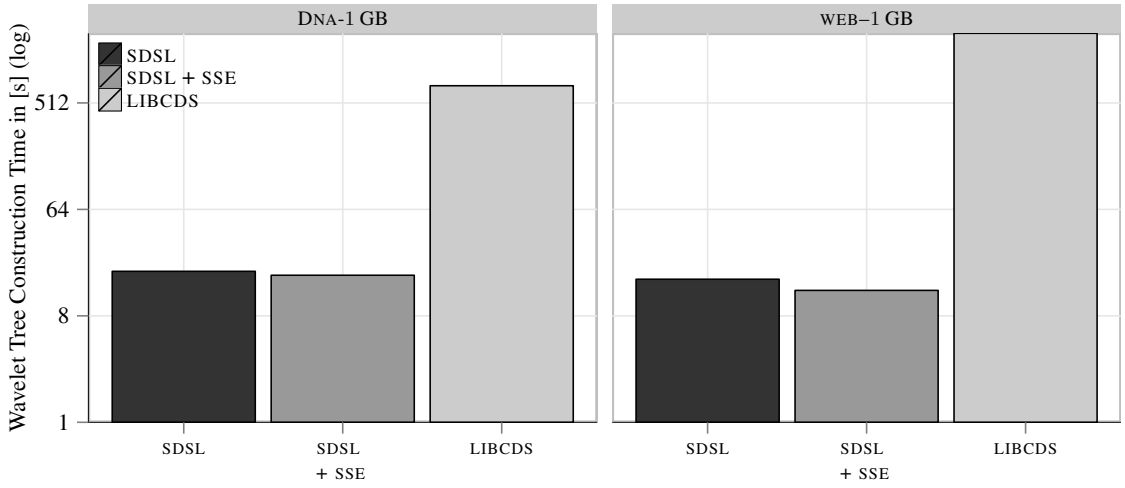

Figure 3.18: Wavelet Tree construction time for several real world data sets. Run-aware construction utilizing SSE *is compared to bit-by-bit wavelet tree construction in* LIBCDS. *Note the log scale of the y-axis.*

in the BWT. We can then use $rank_{64}$ and $select_{64}$ to efficiently determine run-length and starting positions of each run. The main idea is shown in Figure 3.17. Next three different Huffman-shaped wavelet tree construction algorithms are compared empirically.

Experimental Evaluation of Wavelet Tree Construction

We compare regular bit-by-bit construction used in the LIBCDS library to the run-aware construction method used in SDSL and our SSE enhanced run-aware method. We construct the wavelet tree over the BWT of the DNA and WEB data set of size 1 GB. The results are shown in Figure 3.18. Note that the y-axis is shown as a log scale. For both data sets the bit-by-bit construction method is much slower than the run aware methods. For WEB, the standard run aware method is 120 times faster as there are many runs in the WEB data set. For the DNA data set, the standard run aware method is 37 times faster. The SSE enabled method is 15% faster than the standard run aware method for the DNA data set and 25% faster for the WEB data set. This can not be clearly seen in Figure 3.18, as both methods are much faster than the bit-by-bit construction method. Interestingly, constructing a Huffman shaped wavelet tree using the bit-by-bit method is more expensive than constructing the BWT from the original text: constructing the BWT for WEB requires 181 seconds whereas the wavelet tree construction algorithm used in LIBCDS requires 1995 seconds. In contrast, our SSE enabled run-aware algorithm uses only 13 seconds. Overall, efficient wavelet tree construction is an important

aspect of efficient construction of succinct text indexes.

3.6 Effects on Succinct Text Indexes

In this section we investigate the effect caused by using faster $rank$ and $select$ implementations on the performance of succinct text indexes. We mainly focus on FM-Indexes discussed in Section 2.5. First we establish that our implementations are competitive to commonly used implementations proposed by Ferragina et al. [2008]'s study using the *Pizza&Chili* corpus. We then show how our implementations profit from the optimizations on bitvector representations discussed above.

For simplicity, we only focus on $count$ queries in our evaluation. A $count$ query for a pattern of length m can be reduced to about $2mH_0$ $rank$ queries if a Huffman-shaped wavelet trees is used, or at most $2m(H_0 + 2)$ $rank$ and $2m$ $select$ queries if the run-length encoded wavelet trees is used. We therefore expect that our improvements in the previous section will directly translate into corresponding speedups of our FM-Index. We specifically do not evaluate $extract$ and $locate$ queries as their underlying algorithm relies on sample parameters which introduce additional dimensions in the experimental evaluation.

3.6.1 Text Index Implementations

We compare our implementations and improvements the widely used baselines provided by Ferragina et al. as part of an extensive study [Ferragina et al., 2008]. They provide multiple implementations of common index types as well as the widely used *Pizza&Chili*-corpus discussed in Section 2.7.3. Here we give a short description of the baseline implementations which we will compare against:

SSA The Succinct Suffix Array [Mäkinen and Navarro, 2004] is an FM-Index based on a Huffman-shaped wavelet tree. The wavelet tree uses uncompressed bitvectors and a one level rank data structure with 5% space overhead and $popcnt$ implementation pop_{table}. Implementation by Veli Mäkinen and Rodrigo González

SSA-RRR Same index as SSA but the H_0-compressed bitvector representation of Pagh [1999]; Raman et al. [2002] implemented by Claude and Navarro [2008] is used.

RLFM Run-length wavelet tree [Mäkinen and Navarro, 2005] implementation by Veli Mäkinen and Rodrigo González. The two indicator bitvectors for the run heads in the BWT are represented by uncompressed bitvectors and the same rank structure as in SSA. The *select* operation is solved by a binary search over the rank samples.

SAu Plain suffix array [Manber and Myers, 1993] implementation by Veli Mäkinen and Rodrigo González.

We compiled the implementation with all optimizations and added flag `-m32` since our platform is 64-bit and the code is 32-bit which limits the usage of these indexes to small inputs.

Our Corresponding Implementations We create comparable SDSL indexes for the presented baseline indexes by parametrizing SDSL FM-Indexes and CSAs with comparable basic data structures. The following list gives an overview.

FM-HF-V5 FM-Index based on a Huffman-shaped wavelet tree (`wt_huff`) which is parametrized with the uncompressed bitvector (`bit_vector`) and the 6.25% overhead rank structure `rank_support_v5<>`. Corresponds to the baseline index SSA.

FM-HF-R^{3}15 Same as FM-HF-V5 except that the wavelet tree is parametrized with the compressed bitvector `rrr_vector<15>` and its associated rank and select structures. Corresponds to the baseline index SSA-RRR.

FM-HF-R^3K A family of more space-efficient FM-Indexes. Realized by parametrizing the wavelet tree of FM-HF-V5 by the `rrr_vector<`K`>`

FM-RLMN FM-Index based on a run-length compressed wavelet tree (`wt_rlmn`) which is configured with compressed bitvectors (BV-SD) for the two indicator bitvectors. Corresponds to the baseline index RLFM.

CSA-SADA The SDSL CSA class (`csa_sada`) parametrized with Elias-δ coder and Ψ sampling density of 128. Corresponds to CSA.

Note that all these indexes do not contain SA or ISA samples which are not required to answer *count* queries efficiently.

	SSA		SSA-RRR		RLFM		SAu	
	Time (μs)	Space (%)	Time (μs)	Space (%)	Time (μs)	Space (%)	Time (μs)	Space (%)
200 MB test instance								
XML	1.300	69	1.776	34	6.108	52	0.436	500
DNA (P&C)	0.548	29	0.812	30	1.216	62	0.388	500
ENGLISH	1.156	60	1.676	40	1.744	67	0.372	500
PROTEINS	1.024	56	1.668	55	1.704	76	0.368	500
SOURCES	1.356	72	1.960	41	1.864	61	0.364	500

Table 3.5: Time and space performance of four Pizza&Chili FM-Index implementations on five different inputs for count queries. The space of an index is stated in percent of the input text. The time is the average time to match one character in a count query. It was determined by executing $50,000$ *queries, each for patterns of length* 20*, which were extracted from the corresponding texts at random positions.*

3.6.2 Baseline Comparison

To validate our results we first compare the competitiveness of our implementations to baselines commonly used in the succinct data structure community. Here we reproduce the $count$ experiment from Ferragina et al. [2008]'s study. They report the average time to match one character in a $count$ query. It was determined by executing $50,000$ queries, each for patterns of length 20, which were extracted from the corresponding texts at random positions. Our machine is much faster than that of Ferragina et al. To compare our implementation to the baseline we therefore reproduce the counting experiment from Section 5.2 (Table VI) in their paper. Table 3.5 shows the results of the reproduced experiment. Note that re-running the experiments on our machine improves the running times of all baseline implementations by a factor of two as the hardware (LARGE) has become much faster. The space consumption is identical to those reported by Ferragina et al. in their experimental study.

Next we compare the running time and space consumption of Ferragina et al.'s baselines to our corresponding implementation in SDSL. Note that we do not use any of our proposed improvements (like pop_{blt} or hugepages) to allow for a fair comparison of both implementations. The results are shown in Table 3.6. Comparing our results to the *Pizza&Chili* performance shown in Table 3.5, as expected, the space usage of SSA and FM-HF-V5 are almost the same since the rank structure only differ by 1.5% in size. The runtime of FM-HF-V5 is slightly faster since the 5% overhead $rank$ implementation has to scans blocks 66% larger than that of RANK-V5. The space of FM-HF-R^{3}15 is slightly smaller than that of SSA-RRR since we use bit-compressed integer vectors whenever possible. The time is again faster, since we use our algorithmic optimizations described in Section

	FM-HF-V5		FM-HF-R^{3}15		FM-RLMN		FM-HF-R^{3}63	
	Time (μs)	Space (%)	Time (μs)	Space (%)	Time (μs)	Space (%)	Time (μs)	Space (%)
200 MB test instance								
XML	1.096	70	1.316	32	3.456	34	2.048	17
DNA (P&C)	0.404	29	0.728	28	1.484	79	1.660	24
ENGLISH	0.976	61	1.472	38	2.272	69	2.648	27
PROTEINS	0.872	56	1.604	53	2.128	89	3.228	48
SOURCES	1.208	73	1.688	39	2.460	53	2.820	26

Table 3.6: Space and time performance of four SDSL *FM-Index implementations. The experiment was the same as in Table 3.5. All implementations in this experiment use method pop_{table} for $popcnt$.*

3.4. We get an interesting result for the space of the run-length compressed FM-Indexes. While our implementation FM-RLMN which uses the compressed bitvector representation is smaller than RLFM for two test cases (XML, SOURCES) it is larger for the rest. This is caused by the fact that the Burrow-Wheeler-Transform (BWT) of the latter test cases contains to many runs and so the bitvectors are too dense to achieve compression using BV-SD. Furthermore the compressed bitvector causes a slowdown. Unexpectedly, FM-RLMN is two times faster than RLFM on the XML instance.

The last column of Table 3.6 shows the effect of using a better compressed B_{RRR}-K (that is increasing K) in FM-HF-R^3K. As expected, the space significantly reduces for the compressible texts and the runtime doubles compared to FM-HF-R^{3}15. Taking the data from Figure 3.15 (showing the performance of RRR for increasing K), we could have expected an even greater slowdown as RRR using $K = 63$ is substantially slower than RRR using $K = 15$.

3.6.3 Effects of Our Optimizations

Here we show the effect of applying our environmental features to the performance of our indexes. We use the same $count$ experiment we used to compare our implementations against the *Pizza&Chili* baselines. Additionally, we now apply all of the optimizations discussed in this chapter. Instead of the slow pop_{table} we now use pop_{blt}. We further use HP to minimize the effect of address translation. Table 3.7 shows the run time performance of our implementations on the large test machine (LARGE) as well as the speed up in percent (Δt) over the unoptimized implementation.

The runtime of FM-HF-V5 is reduced by about 40% for all test cases when SSE and HP are activated. This means that our implementation is twice as fast as the highly optimized *Pizza&Chili* counterpart. For the DNA (P&C)-instance it is even faster than the suffix array solution which takes

17 times the space of FM-HF-V5. The effect on the Huffman-shaped wavelet tree-based FM-Indexes using B_{RRR}-K is not substantial. For both $K = 15$ and $K = 63$, decoding the individual blocks in the compressed representation is the bottleneck, and thus B_{RRR}-K can not be improved by the applied features. Our second compressed solution FM-RLMN profits mostly from the SSE features (pop_{blt} and sel_{blt}) for two reasons: first, the $rank$ operations on the underlying Huffman-shaped wavelet tree on uncompressed bitvectors is accelerated, and second, the $rank$ queries in BV-SD which translate to $select$ queries on SEL-C are faster through sel_{blt}, see Figure 3.9.

3.6.4 Effects of Our New Bitvector Representations on Index Performance

We specifically evaluate the usefulness of our new bitvector representation (RANK-IL) as well as our optimizations to H_0 compressed bitvectors (B_{RRR}). The Huffman-shaped wavelet tree is parametrised with the two 25% overhead rank structures RANK-V and RANK-IL. We refer to the resulting indexes as FM-HF-V and FM-HF-1L. We further evaluate FM-Indexes using H_0 compressed bitvectors (B_{RRR}) using larger block sizes $K = 127$ and $K = 255$. We expect that the runtime performance of these indexes directly correlate with the runtime shown in the evaluation of compressed bitvectors shown in Figure 3.15 and discussed in Section 3.4. We use test instance sizes of 64 MB and 64 GB to elucidate the difference between small and large test instances. We use prefixes of WEB-64 GB and again use the same methodology to get the average query time per character of a $count$ query: $50,000$ patterns of length 20 are extracted from random positions of the corresponding input prior to the experiment and are then queried.

The results of the experiments are shown in Table 3.8 and Figure 3.19. We first discuss Table 3.8 which shows the run time performance of each index type depending on the data set and different feature sets. For FM-HF-R^3K a performance improvement caused by applying optimizations is only achieved for the large test instance. Enabling SSE(pop_{blt}) for $K = 15$ results in faster runtime performance, since a shift and $popcnt$ is used to determine the rank to the right offset, after a block b'_i is decoded by a table lookup. The on-the-fly decoding based solution with $K \neq 15$ stops decoding when the offset is reached and therefore do not profit from SSE. All FM-HF-R^3K profit from HP, but the effect decreases as K gets larger and the decoding dominates the runtime. We briefly discuss possible reasons why HP does not affect the indexes for the small test instance. For the 64 MB the size of the indexes is at most 30% (see Figure 3.19) of the original bitvector of size 64 MB. Therefore the page table holding the data structure in-memory is roughly 20 kB large, which fits in L1 cache. Therefore, a TLB miss can be resolved efficiently via an access to the L1 cache which would explain why the hugepage feature has almost no effect. FM-RLMN profits from each feature, especially from

	FM-HF-V5		FM-HF-R^{3}15		FM-RLMN		FM-HF-R^{3}63	
	t (μs)	Δt (%)	t (μs)	Δt (%)	t (μs)	Δt (%)	t (μs)	Δt (%)
200 MB test instance								
XML								
pop_{table}	1.096		1.316		3.456		2.048	
pop_{bw}	0.924	-16	1.324	$+1$	2.600	-25	2.064	$+1$
pop_{blt}	0.764	-30	1.304	-1	1.876	-46	2.036	-1
pop_{blt}+HP	0.644	-41	1.140	-13	1.764	-49	1.924	-6
DNA (P&C)								
pop_{table}	0.404		0.728		1.484		1.660	
pop_{bw}	0.348	-14	0.724	-1	1.408	-5	1.652	$+0$
pop_{blt}	0.280	-31	0.716	-2	1.248	-16	1.660	$+0$
pop_{blt}+HP	0.252	-38	0.628	-14	1.356	-9	1.688	$+2$
ENGLISH								
pop_{table}	0.976		1.472		2.272		2.648	
pop_{bw}	0.832	-15	1.480	$+1$	2.108	-7	2.644	$+0$
pop_{blt}	0.688	-30	1.472	$+0$	1.900	-16	2.660	$+0$
pop_{blt}+HP	0.600	-39	1.300	-12	2.200	-3	2.756	$+4$
PROTEINS								
pop_{table}	0.872		1.604		2.128		3.228	
pop_{bw}	0.752	-14	1.596	$+0$	1.980	-7	3.264	$+1$
pop_{blt}	0.624	-28	1.560	-3	1.824	-14	3.244	$+0$
pop_{blt}+HP	0.552	-37	1.408	-12	2.100	-1	3.432	$+6$
SOURCES								
pop_{table}	1.208		1.688		2.460		2.820	
pop_{bw}	1.044	-14	1.680	$+0$	2.196	-11	2.804	-1
pop_{blt}	0.856	-29	1.656	-2	1.996	-19	2.824	$+0$
pop_{blt}+HP	0.744	-38	1.476	-13	2.276	-7	2.652	-6

Table 3.7: Time performance of the our FM-Index implementations of Table 3.6 dependent on the used $popcnt$ method and the usage of 1 GB pages (HP) to avoid TLB misses running on LARGE. The time difference is stated as relative difference to the corresponding implementation which uses method pop_{table} for $popcnt$.

	FM-HF-R^3K				FM-RLMN	FM-HF-V	FM-HF-1L
	$K = 15$	$K = 63$	$K = 127$	$K = 255$			
	t (μs)	t (μs)	t (μs)	t (μs)	t (μs)	t (μs)	t (μs)
WEB-64M							
pop_{table}	1.183	2.085	3.509	13.864	2.362	0.789	0.843
pop_{bw}	1.194	2.091	3.482	14.027	1.855	0.684	0.733
pop_{blt}	1.182	2.094	3.489	14.019	1.447	0.651	0.590
pop_{blt}+HP	1.008	1.991	3.393	13.714	1.316	0.519	0.508
WEB-64G							
pop_{table}	3.488	3.675	4.794	17.571	8.237	2.440	2.684
pop_{bw}	3.526	3.707	5.621	16.940	6.991	2.244	1.952
pop_{blt}	2.945	3.724	4.961	15.756	4.960	2.224	1.700
pop_{blt}+HP	1.780	2.796	4.353	15.496	4.177	1.085	0.870

Table 3.8: Mean time t for $count$ of different FM-Index implementations dependent on instance size of 64 MB and 64 GB while using different sets of features.

replacing the lookup table version for $popcnt$ and $select_{64}$ to the broadword of SSE versions. Since FM-RLMN is a rather complex structure the different $rank$ and $select$ sub data structures "compete" for cache when implemented by lookup tables.

We now we discuss the impact on the FM-Indexes based on the rank structure RANK-V and RANK-IL for uncompressed bitvectors (FM-HF-V and FM-HF-1L). For the small instance, the difference between the slowest configuration (pop_{table}) and the fastest is around 40% for both implementations. For large instances, the effect of hugepages is especially noticeable. Using only pop_{blt} in conjunction with hugepages (HP) is twice as fast as using only pop_{blt}. This effect is much larger than when comparing the basic $rank$ structure itself as shown in Section 3.2, which can be explained as follows. During the process of *backward search* the $rank$ operations on the bitvector are performed in pairs. We first map the start sp of each range $\langle sp, ep\rangle$. Next we map the end ep of the same range. The difference in the positions of the $rank$ call is therefore *not* random. As the *backward search* process continues, the range $\langle sp, ep\rangle$ becomes smaller. Therefore, using hugepages, the $rank$ operation of ep will likely reside inside the same memory page as that of sp. As expected the FM-Index based on RANK-IL is slower than the RANK-V-based index when no optimization is used, but is faster when SSE and hugepages are activated. This reflects the outcome for the basic data structure in Figure 3.4. Interestingly, FM-HF-1L using all optimizations (pop_{blt} and HP) can perform $count$ on the large data set at the same cost as the using no optimizations on the small 64 MB data set. Further, note that the HP feature significantly improves the run time performance of all indexes. In our previous experiments using the *Pizza&Chili* corpus this was not visible. This highlights one of the weaknesses

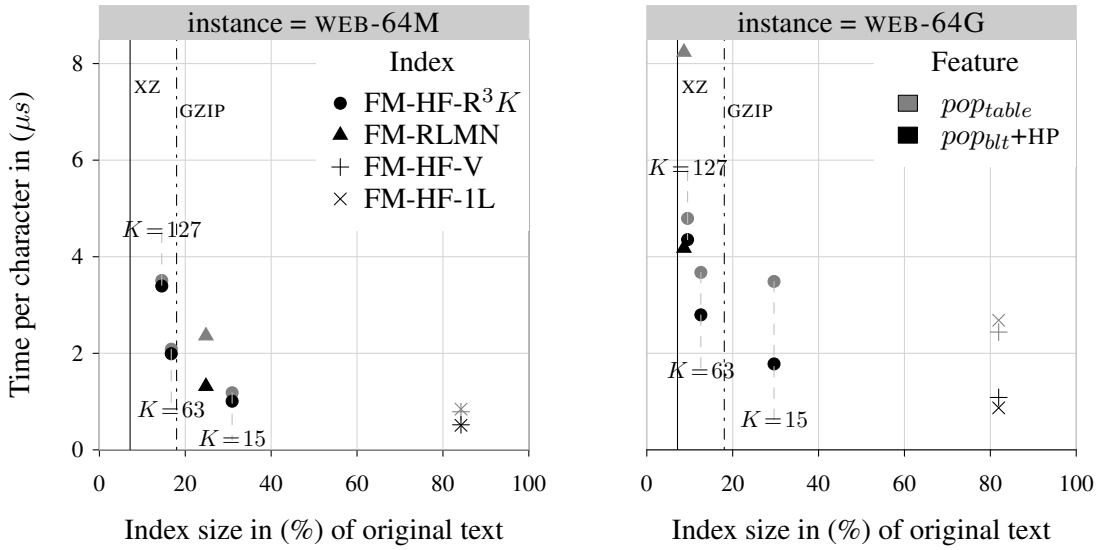

Figure 3.19: Time and space trade-offs of our index implementations on input instances of size 64 *MB and* 64 *GB with compression effectiveness baselines using standard compression utilities* XZ *and* GZIP *with option* `--best`.

of the *Pizza&Chili* corpus. While sizes of 200 MB for the provided data sets was still sufficient for the time it was created (2008), today evaluation on larger data sets can lead to very different results.

Figure 3.19 shows the different time-space trade-offs achieved by our indexes. It further shows the effect of our optimizations, as well as comparative compression effectiveness baselines using standard compression utilities XZ and GZIP. Note that $K = 255$ was not included as the running time is too high. Interestingly, the compression effectiveness of FM-HF-R^3K= 255 outperforms GZIP and is within 5% of XZ, the best of-the-shelf compression utility available, while still providing query functionality in the microsecond range. For the large test instance, even FM-HF-R^3K= 127 achieves compression close to that of XZ. For all implementations the "move" on the y-axis shows the running time improvements when using our optimizations. The difference is very noticeable for $K = 15$ for the large data set, but only minor for the small data set as discussed above. The compressed representation using our optimizations is faster than FM-HF-V using no optimizations.

3.7 Summary and Conclusion

In this chapter we investigated optimizing different aspects of succinct data structures. Specifically, we investigated the basic building blocks of succinct text indexes. First we evaluated and engineered common operations $rank_{64}$ and $select_{64}$ on computer words. Next we moved one level higher in

the succinct data structure hierarchy by investigating $rank$ and $select$ on bitvectors. We proposed a simple cache-friendly rank data structure (RANK-IL) and a faithful implementation of the select structure of Clark [1996] which adheres to the theoretical bounds of the structure. We further provided a more space efficient, faster and practical solution (SEL-C) for select data structures on uncompressed bitvectors as well as an improved compressed bitvector representation.

We empirically demonstrated that our proposed improvements outperform existing implementations. Our experiments further explored the behaviour of $rank$ and $select$ data structures for binary and general sequences in different scenarios. In our experiments, we varied data size, implementation, used instruction sets and operation system features. Overall we provided, to our knowledge, the fastest FM-Index (FM-HF-1L using pop_{blt} and HP) and the smallest FM-Index (FM-HF-R$^3K = 255$) with compression effectiveness close to that of the state-of-the-art compressor XZ while still providing query functionality in the microsecond range.

We further discovered that the performance of the basic succinct data structures $rank$ and $select$ can be increased by using advance CPU instructions – the built-in $popcnt$ operation and our new suggested select routine – and the hugepage feature. We have demonstrated that these improvements propagate directly to more complex succinct data structures like FM-Indexes. Especially the hugepage feature was not yet explored in literature on succinct data structures and string processing. This is surprising as it seems natural to do so, since succinct data structures usually have memory access patterns which cause many TLB and cache misses.

We think that exploiting the hugepage feature as well as creating more cache-efficient succinct data structures is an important step to making succinct structures perform efficiently on larger data sets. One of the main problems when processing large data sets is locality of access. For small data sets, large parts of a succinct data structure can reside close to the CPU in various levels of the cache hierarchy. As the data size increases, this is no longer the case which could clearly be observed in our empirical evaluation. Both the hugepage feature as well as cache-efficient data structure layouts and access can mitigate the problem of performing many random, uncached, accesses to main memory. Exploring these features is an important aspects in providing efficient succinct data structures which perform well on both small and large data sets to the extent where they are competitive against classical uncompressed structures in more scenarios.

Chapter 4

Revisiting Context-Bound Text Transformations

The BWT is one of the main components of many compression systems and succinct text indexes. The transform produces a permutation of a string T of size n, denoted T^{bwt}, by sorting the n cyclic rotations of T into full lexicographical order and taking the last column of the resulting $n \times n$ matrix $\mathcal{M}$ to be T^{bwt}. Since the transform was introduced in 1994, many empirical evaluations and theoretical investigations have followed [Fenwick, 1996; Manzini, 2001].

Most BWT-based compression systems and succinct text indexes fully sort the cyclic rotations of T, and nearly all current empirical studies of both compression systems and succinct text indexes assume a full sorting of rotations. However, a full sorting of the rotations is resource intensive. The construction of the fully sorted BWT is generally performed by construction the suffix array (SA) over T. In-memory suffix array construction can be performed efficiently, but requires up to $n + n \log n$ bits of space in available main memory [Puglisi et al., 2007]. In practice, the fastest construction algorithms uses 9 times the space of the original text during construction [Puglisi et al., 2007]. More space efficient algorithms exist which however are much slower in practice [Ferragina et al., 2012]. This is especially problematic for suffix-based succinct text indexes. Succinct text indexes can provide search over a text T in space roughly equal to compressed representation of T. Thus, full text indexes require much less space during operation when compared to the construction cost where constructing the full BWT is the main resource constraint. Expensive construction of the BWT therefore prohibits the use of succinct full indexes on large text sequences which in turn has undeniably hampered the adoption of succinct text indexes in practice.

In independent work, Schindler [1997] and Yokoo [1999] described an alternative approach in

which the n rotations are only *partially* sorted to a fixed prefix depth, k. This modified transform is referred to as the k-BWT. By limiting the sort depth to k, sorting can be accomplished in $\mathcal{O}(nk)$ time using radix sort, and is very fast in practice. Moreover, Schindler reported nearly identical compression effectiveness to the full transform, even for small values of k. The algorithm developed by Schindler [1997] was subsequently made available in the general purpose compression tool SZIP.

Interestingly, the research community has not given much attention to the k-BWT. In this chapter we revisit the k-BWT. Our contributions and the structure of this chapter can be summarized as follows:

1. We provide a formal definition of the k-BWT and its auxiliary structures and algorithms in Section 4.1.

2. Our second major contribution in this chapter is a fast, efficient external memory k-BWT construction algorithm which outperforms all existing full suffix array construction algorithms in Section 4.2. This allows the construction of the k-BWT for data sets much larger than those that are commonly indexed using traditional, BWT based, succinct text indexes.

3. We conduct an extensive evaluation of existing in-memory k-BWT construction algorithms in Section 4.1.2. We further discuss problems with adopting popular induced suffix sorting techniques to constructing the k-BWT.

4. Our next contribution is a new linear-time k-BWT reversal algorithm which explicitly stores information required to reverse the transform instead of recovering it during the reversal process. We analyse the time and space trade-offs of our new algorithm and compare it to the normal BWT and k-BWT reversal algorithms. We find that algorithms performing well in theory are outperformed by more inefficient algorithms in practice.

5. We further investigate a previously undocumented locality of access property inherent to all k-BWT reversal algorithms in Section 4.3.4. This locality allows fast transform reversal for small k.

6. Our last contribution is the first thorough empirical analysis of state-of-the-art k-BWT algorithms for the forward and inverse transforms, compression effectiveness, and associated trade-offs in Section 4.4.

4.1 Forward Context-Bound Text Transformations

In this section the forward k-BWT transform is discussed. We formally introduce the forward k-BWT and evaluate the efficiency of different forward transform algorithms.

4.1.1 The Regular Burrows-Wheeler Transform

The BWT is a popular text transformation used in many compression systems and succinct text indexes. A text T of size n is permuted using the transform by sorting all suffixes of T in lexicographical order. This can be viewed as sorting conceptual matrix $\mathcal{M}$ of size $n \times n$. The output of the transform corresponds to the last column (L) of $\mathcal{M}$ where $L[i] = \mathcal{M}[n-1][i]$ for $0 \leq i < n$. This is shown in Figure 4.1 (left) where, for the input text $T =$`chacarachaca$`, $T^{bwt} =$`achhrcaa$acca`. The regular BWT transform is described in detail in Section 2.4.1.

4.1.2 The Context-Bound Burrows-Wheeler Transform

When constructing the BWT via suffix array construction, one of the main problem is the worst case $\mathcal{O}(n)$ cost of a *single* suffix comparison. Independently, Schindler [1997] and Yokoo [1999] propose to limit the number of characters to be compared during a single suffix comparison to at most k. This implies that the matrix $\mathcal{M}$ is only sorted up to depth k. The partially sorted matrix is referred to as $\mathcal{M}_k$. The context based on which the symbols in the transform are ordered by is therefore also bound by k. This partial ordering is referred to as a k-ordering of rotations into k-order, and to the process itself as a k-sort. If two or more rotations are equal under k-order, they fall into the same *context group* or k-group and are said to be k-equal. Within a k-group all k-equal suffixes appear in text-order and are therefore stably sorted. The output of the transform of an input T is referred to as T^{kbwt}. The sorted matrix $\mathcal{M}_k$ can be partitioned into context groups. The context group containing the i-th largest prefix of length k is denoted as C_i. All context groups are strictly lexicographically ordered where $C_{i-1} < C_i < C_{i+1}$. In the practical examples in this section a specific k-group is referred to as C^{abc}. Within C^{abc} all rows are prefixed by `abc`. There are at most n context groups in $\mathcal{M}_k$. In this case, $\mathcal{M}_k$ is equal to $\mathcal{M}$ and therefore the k-BWT is equal to the BWT. This is however only guaranteed for $k = n$. An example of the k-BWT is shown in Figure 4.1 (right) for $k = 2$.

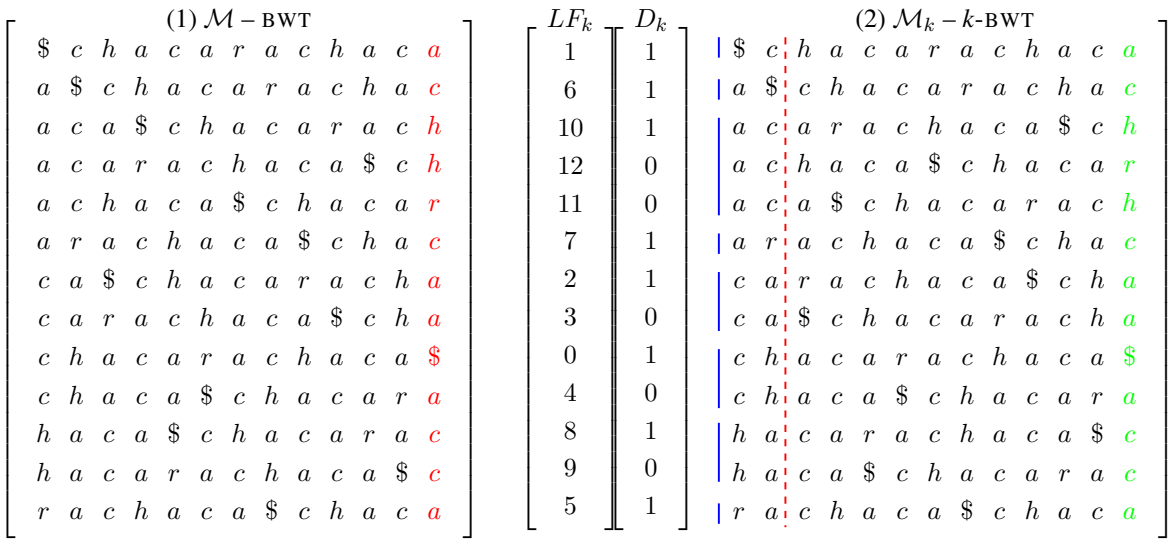

Figure 4.1: Comparison of regular BWT *and context-bound* BWT *for* $k = 2$ *for the input text* `chacarachaca$`. *The full* BWT *(left) is sorted completely whereas the* k-BWT *is only sorted up to a depth of* 2. *Rows with the same* $k = 2$ *long prefix are grouped together in context groups which are marked by the bitvector* D_k. *The* LF *mapping needed to recover* T *from* T^{kbwt} *is shown as* LF_k *and* T^{kbwt} =`achrhcaa$acca`.

The context group boundaries can be can be marked in a bitvector D_k of length n. The bitvector is formally defined as follows:

Definition 7 *For any* $0 \leq k < n$, *let* $D_k[0, n-1]$ *be a bitvector, such that* $D_k[0] = 1$ *and, for* $1 \leq i < n$,

$$D_k[i] = \begin{cases} 0 & \text{if } \mathcal{M}_k[i][0, k-1] = \mathcal{M}_k[i-1][0, k-1]\,, \\ 1 & \text{if } \mathcal{M}_k[i][0, k-1] \neq \mathcal{M}_k[i-1][0, k-1]\,. \end{cases}$$

For our example shown in Figure 4.1, D_k is 1110011010101. So, $D_k[i] = 1$ if the rotations at rows i and $i-1$ in $\mathcal{M}_k$ are in different k-groups, and $D_k[i] = 0$ if they belong to the same k-group. Thus, a k-group containing $v+1$ members is indicated by a substring 10^v in D_k.

The difference between the BWT and the k-BWT for the input text `chacarachaca$` for $k = 2$ is shown in Figure 4.1. Note that the difference between the k-BWT output and the full BWT output is minor. The only difference between both transforms occurs in context groups larger than one. Even for small k of around 10, the output is very similar for most input texts which results in similar

compression effectiveness when replacing the BWT with the k-BWT [Schindler, 1997]. Compression trade-offs of the k-BWT are explored in detail in Section 4.4.

4.1.3 Engineering In-memory k-BWT Construction

Next, engineering an efficient forward k-BWT transform is discussed. First, traditional radix sort and multi-key quicksort schemes are investigated and modified to construct the k-BWT. Then the problem with using popular induced suffix sorting techniques for constructing the k-BWT is investigated. Last, we perform an experimental evaluation of the different forward transforms.

Radixsort-Based k-BWT Construction

The most straightforward way to perform k-BWT construction is radixsort. Schindler [1997] provides the compression tool SZIP. It uses a specialized radix sort implementation which uses counting sort to compute the $k = 2$ sorting order before switching to regular a radix sort at a total cost of $\mathcal{O}(kn)$ time. This requires $2\sigma \log n$ bits of extra space to perform counting sort. Kärkkäinen and Rantala [2008] propose three cache-efficient radixsort-based string sorting algorithms. The most efficient method (CE2-S) method uses $2\sigma \log n + n \log \sigma + n \log n$ bits of extra space. We refer to this method as RDX-CACHE.

Multi-Key Quicksort Construction

Another common string sorting algorithm is the multi-key quicksort [Bentley and Sedgewick, 1997]. Multi-key quicksort combines quicksort and radixsort to sort strings consisting of multiple characters. The algorithm partitions the strings at the current sorting level, d, into three partitions: smaller, equal or larger than the pivot character. Each partition is processed recursively. For all symbols equal to the pivot character the sorting depth is increased to $d + 1$. For the two other partitions a new pivot element at the same depth d is chosen. This algorithm can be trivially adapted to construct the k-BWT as follows. First, the maximum recursion depth is limited to k. Second, for partitions of size larger than one which are sorted up to depth k, additional work is required. Each of the fully sorted partitions is a context group. Quicksort is not stable. Therefore, each "final" context group is further sorted based on the integer values of the suffix positions. This ensures that within a context group, all suffix positions are stored in their initial text order. This algorithm is referred to as MK-QSORT.

Induced Suffix Sorting-Based Construction

Using induced sorting, as little as 30% of the suffixes must be sorted to obtain the final suffix array and the resulting T^{bwt} [Maniscalco and Puglisi, 2006]. Algorithms using induced suffix sorting perform very well in practice despite having a non-optimal worst case time complexity. The main idea behind induced sorting is as follows. First all text positions are classified into different types. Itoh and Tanaka [1999] classify the positions into two types whereas only the suffixes of one of the type have to be sorted explicitly. The correct position of all suffixes of the remaining type can be *induced* by performing one pass over the partially sorted suffix array. The algorithm sorts around 50% of the suffixes explicitly and induces the order of the remaining suffixes.

Itoh and Tanaka [1999] partition all suffixes in T into two types, A and B, by comparing any two prefixes of two adjacent suffixes T:

$$T[i] = \begin{cases} \textbf{Type A} & \text{if } T[i] > T[i+1]\,, \\ \textbf{Type B} & \text{if } T[i] \leq T[i+1]\,. \end{cases}$$

Each individual type B suffix S_i is then assigned a bucket in SA according to the first character $T[i]$. All type B buckets are then sorted into the correct lexicographical order. One pass is then performed to induce the remaining unsorted A type suffixes. This process is shown in Figure 4.2.

Unfortunately this technique cannot easily be adopted to constructing the k-deep suffix array SA_k and therefore the k-BWT. The main problem is the *induction* step described above. When inducing the order of the type A suffixes in the last step, each "induction" step increases the sorting order by one. Therefore, inducing the sorting order of a position $\text{SA}[i]$ from a position sorted up to a depth of k will result in $\text{SA}[i]$ being sorted up to depth $k+1$. To enable performing induced sorting to construct SA_k and the k-BWT we enhance the initial algorithm of Itoh and Tanaka [1999] to be *context aware*. When sorting a B-type bucket to depth k, the context group boundaries D_k within the bucket are implicitly determined. From the definition of types A and B it can be deduced that, for a given symbol c, all types A and B suffixes are in different context groups as $T[i] > T[i+1]$ for type A and $T[i] \leq T[i+1]$ for type B. To induce a k context group the "source" context therefore is required to be $k-1$ sorted. This is shown in Figure 4.3. A context group C^{hab} is not induced from the $k=3$ contexts C^{aba} to C^{abd} but from the larger context group C^{ab} with depth $k=2$.

The algorithm of Itoh and Tanaka [1999] is modified to be context aware as follows. During the induction step the partially sorted suffix array is processed on a context by context basis. Initially all B types blocks are sorted to depth $k-1$ with which the correct k sorting order of all suffix array

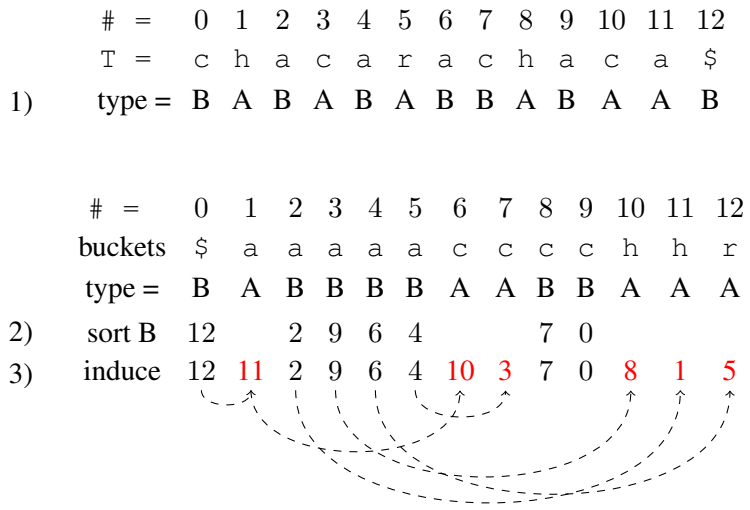

Figure 4.2: Induced sorting for $T =$`chacarachaca$` *of Itoh and Tanaka [1999] in three steps: First determine all type* B *suffixes. Second sort all type* B *buckets and finally induce all type* A *suffixes.*

positions from a type B suffix can be induced. In the last step the sorting depth of the B type buckets is increased from $k-1$ to k. Inducing from a type A position is more complex. Suppose the context group C^{hab} is of type A and was induced from a type B context group C^{ab}. However, C^{hab} is also part of a larger $k=2$ group C^{ha}. Therefore, each time a type A position is induced from another type A suffix the sort order increases. Further, note that this case can easily be detected during the first scan of T where the suffix type of each text position is determined. Two adjacent type A suffixes in T implies that one of the positions will be induced from a type A suffix during the induction step. We therefore mark these positions during the first pass over T. During the induction step, if within a type A bucket, multiple positions in the same "target" context group are induced we again mark this context group. In a last pass the marked context groups are restored to return the correct k-order. This method is referred to as INDUCE-K.

Empirical Evaluation

Next the performance of the in-memory forward transform algorithms is evaluated. The running time of the different algorithms is compared to the fastest in-memory suffix array construction algorithm DIVSUFSORT maintained by Yuta Mori.[1] The performance of our different methods discussed above

[1] available at `https://code.google.com/p/libdivsufsort/`

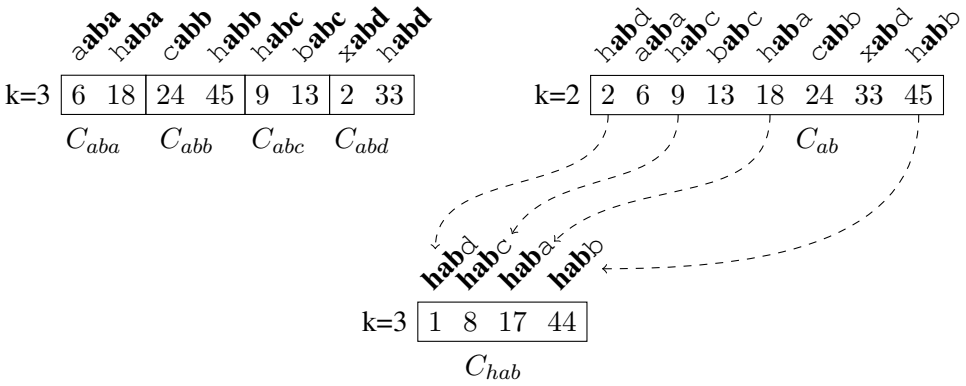

Figure 4.3: Context aware induction step where the context C^{hab} is induced from the sorting order of context C^{ab} instead of the $k = 3$ context groups.

is evaluated using the 200 MB test cases of the *Pizza&Chili* corpus (see Section 2.7.3) as well as a 200 MB prefix (WSJ) of the of the *Wall Street Journal* extracted from the Disk 2 of the TREC data collection. The results are shown in Figure 4.4.

Overall, MK-QSORT is not competitive and only outperforms DIVSUFSORT for small k on the DNA data set. For all other test cases and k values it is slower. This can be explained by the amount of work performed by the algorithm. First the context groups are sorted into k-order. Next, each context group larger than one is additionally sorted to ensure all suffix positions within the group are in ascending order. This cannot be performed during the first sorting step as quicksort is inherently not stable. Thus, for small sorting depths, essentially two sorting stages are performed which affects the run time performance of the overall algorithm. The induced suffix sorting method is faster than DIVSUFSORT for small k for all test cases. For larger k, DIVSUFSORT outperforms the induced-based method which requires more "bookkeeping" of context boundaries during the induction step as k becomes larger. The additional steps required to ensure the correct k ordering do not justify the use of induced suffix sorting as cache efficient radixsort-based algorithms always perform better than INDUCE-K. The cache-efficient radix sort, RDX-CACHE, performs best for all test cases. For small sorting depths, it is roughly four times faster than DIVSUFSORT. As k becomes larger, it slowly approaches the running time of DIVSUFSORT. Still, for all test cases and different k it is faster than DIVSUFSORT and all other methods. Only for SOURCES and $k = 10$, DIVSUFSORT is as fast as RDX-CACHE. We further experimented with a combination of "super alphabets" and a fast SSE-based string comparison function in conjunction with RDX-CACHE and all other methods but could not find any significant improvements in the running time of the algorithms. Note that the running time

	$\|\Sigma\|$	Size	LCP_{mean}	LCP_{max}	$\text{LCP}_{\text{median}}$	$\mathcal{H}_0$
WSJ	89	50	16	1,344	12	4.62
DNA	4	50	31	14,836	13	1.98
XML	96	50	42	1,005	30	5.23
SOURCES	227	50	168	71,651	15	5.53
WEB-3 GB	126	3072	6011	55,6673	120	5.35
DNA-3 GB	8	3072	223860	21,049,999	15	2.12

*Table 4.1: Statistical properties of the k-*BWT *benchmark collection.*

of DIVSUFSORT should not be affected by k. In our experiments we reran constructing the full BWT using DIVSUFSORT every time we measure the data points for the *k*-BWT methods. Therefore, a small discrepancy in the expected "constant" running time for different k can be observed in Figure 4.4. Interestingly, the SOURCES file can be constructed much faster using DIVSUFSORT than the other test files, whereas the *k*-BWT-based methods are not much faster for this test case compared to the other test cases.

This can be explained as follows. Consider the statistical properties of our test collection in Table 4.1. The mean and max *lcp* values of the SOURCES collection are much higher than for all other test cases. However, the median *lcp* value ($\text{LCP}_{\text{median}}$) is similar. This implies that there are several suffix positions which have to be sorted to a high depth. Not having to perform these expensive suffix comparisons is therefore beneficial to the overall running time of an algorithm. DIVSUFSORT uses induced suffix sorting methods which sort only 35% of all suffix positions. Thus less expensive suffix comparisons are performed for the SOURCES test case. Therefore, algorithms sorting only a small number of these suffixes (DIVSUFSORT) performs especially well on the SOURCES test file.

In conclusion, for all files with $k < 8$, the *k*-BWT transform can be constructed faster than the BWT. For $k < 5$ the *k*-BWT transform can be constructed twice as fast. The *k*-BWT sorts the initial matrix $\mathcal{M}_k$ up to a depth of k, but the BWT must sort the matrix fully. Even though the construction of the transform is performed using fast suffix array construction algorithms, we still expect the *k*-BWT transform to be more efficient as the average number of character comparisons required to compare two individual suffixes when constructing the full BWT tends to be significantly larger than k.

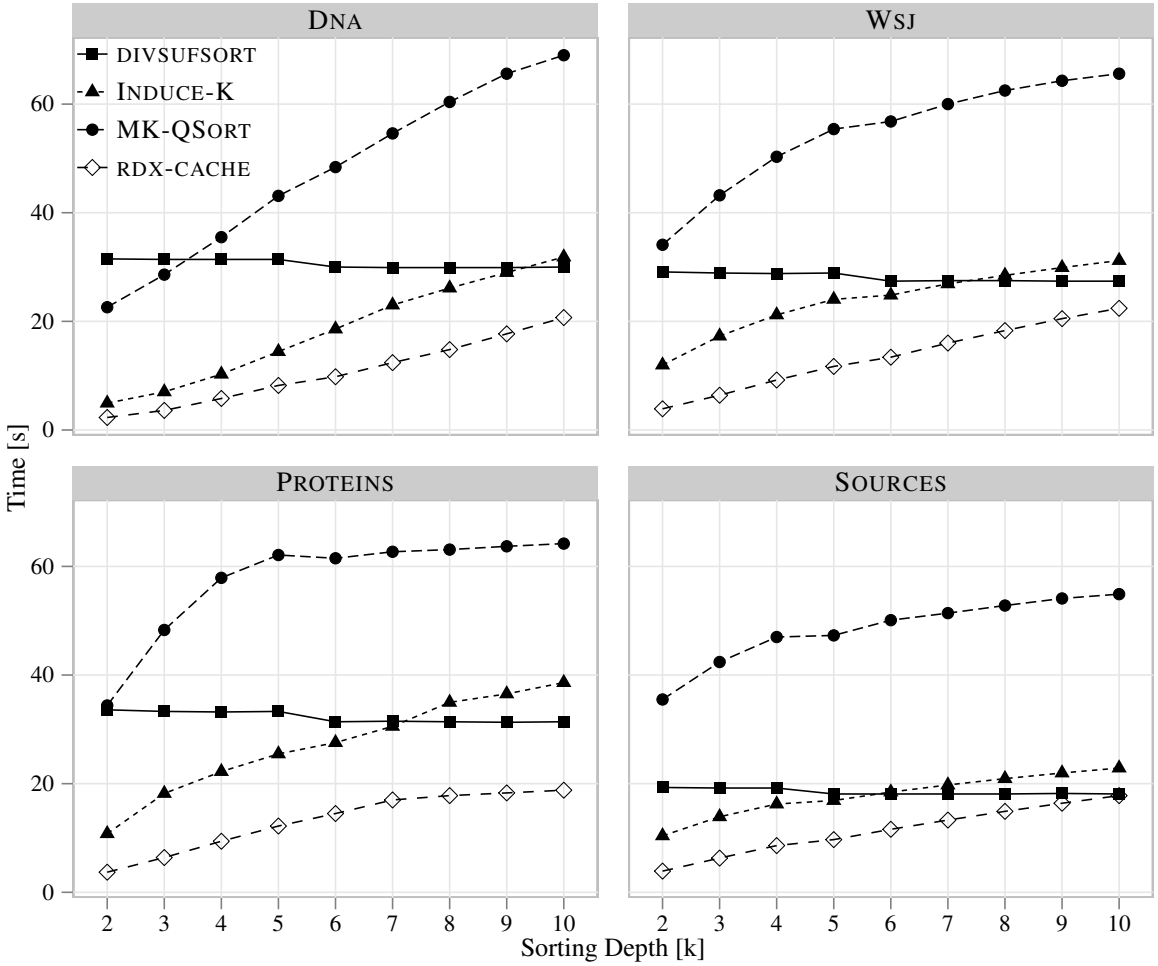

*Figure 4.4: In-memory k-*BWT *forward transform efficiency for the* 200 *MB Pizza&Chili test files for variable k.*

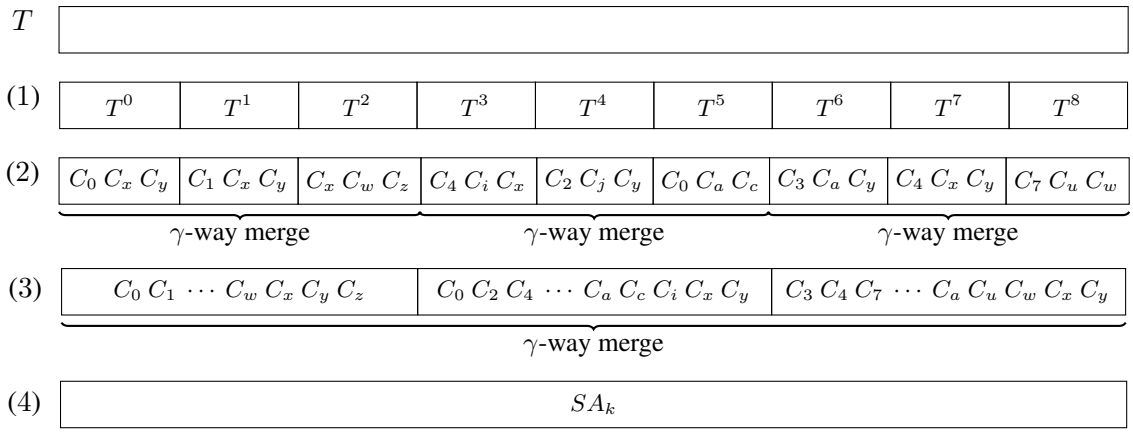

Figure 4.5: External k-BWT construction of a text T using four steps. (1) Split up T into chunks T^i of size $kblock$ and construct $\mathrm{SA}_k(T_i)$ in-memory. (2-3) merge $kblock$ blocks at the same time to create the final SA_k.

4.2 External Memory-Based k-BWT Construction

Next we propose a simple but efficient external memory construction algorithm for the k-BWT. First the algorithm is described in detail. Next the algorithm is empirically compared to state-of-the-art external full BWT construction algorithms.

Algorithm Description

First split the initial text T into blocks T^i of size $kblock$. For each block the suffix array of depth k, SA_k, is constructed and written to disk. After all blocks are processed, a γ-way merge of γ blocks at a time is performed. The merging phase is repeated until only one block comprising of the complete suffix array SA_k remains from which the k-BWT can be extracted. This is shown in Figure 4.5.

Merging two blocks is performed as follows. Each block is composed of several context groups as shown in Figure 4.5. Therefore merging blocks is performed on a context group basis. Merging the individual context group can be performed efficiently. Individual context groups C^{abc} are lexicographically ordered across the initial text blocks T^i. Therefore, C^{abc} in block T^0 is lexicographically smaller than C^{abd} in T^1. If a context group C^{abc} occurs in multiple blocks T^x and T^y, merging C^{abc} can be performed by concatenating C^{abc} of T^y to C^{abc} of T^x. Inside a context group C^{abc}, all suffix positions are stored in their initial text order. All suffix positions of C^{abc} of T^x are guaranteed to be lexicographically smaller than the positions of C^{abc} in T^y. This is shown in Figure 4.6.

The initial SA_k for the initial blocks of text T is created at a total cost of $\mathcal{O}(nk)$ time. The

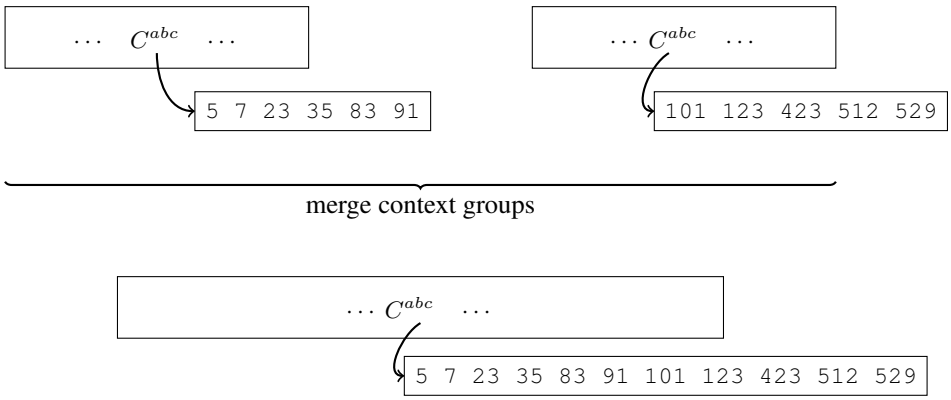

Figure 4.6: Context group merge phase of the external k-BWT algorithm. (1) The context group C^{abc} occurs in two blocks and is merged by concatenating the suffixes stored in the right block to the suffixes in the left block.

text is split into $n' = n/kblock$ blocks where $kblock$ is the size of the initial text blocks T^i. Next, $\log_\gamma(n')$ merge phases are performed to create the final SA_k and the k-BWT. In the worst case each context group is of size 1. This implies that the k-BWT is equal to the regular BWT. In this case, no "merging" of two context groups occurs as each context group occurs only once in T. However, we conjecture that for small k, individual context groups will be large on average, which in turn allows fast construction using our simple γ-way context group merge construction algorithm.

Empirical Evaluation

Next an implementation of our γ-way context group merge algorithm is compared to state-of-the-art external full BWT algorithms. Our approach is compared to the following suffix array construction algorithms: (1) ESAIS [2] proposed by Bingmann et al. [2013] (2) BWTDISK[3] proposed by Ferragina et al. [2010]. The machine DESKTOP described in Section 2.7.1 is used in this experiment. The main memory of the machine is further restricted to to 1 GB via the grub bootloader option `mem=1024m`.

First our approach is compared to the two external construction baselines: ESAIS and BWTDISK. The full SA and the full BWT are built with both external memory algorithms, and the running time is compared to that of our construction algorithm for $k = 2, 4, 6, 8, 10$ for two data sets: WEB-3 GB and DNA-3 GB using only 1 GB main memory. In this experiment the initial block size is set to $kblock = 100$ MB. Due to the long running time of the experiment we only perform each experiment three times and report the mean running time. Varying branching factors $\gamma = 2, 4, 16, 32, 64$ are used

[2] available at `http://panthema.net/2012/`
[3] available at `http://people.unipmn.it/manzini/bwtdisk/`

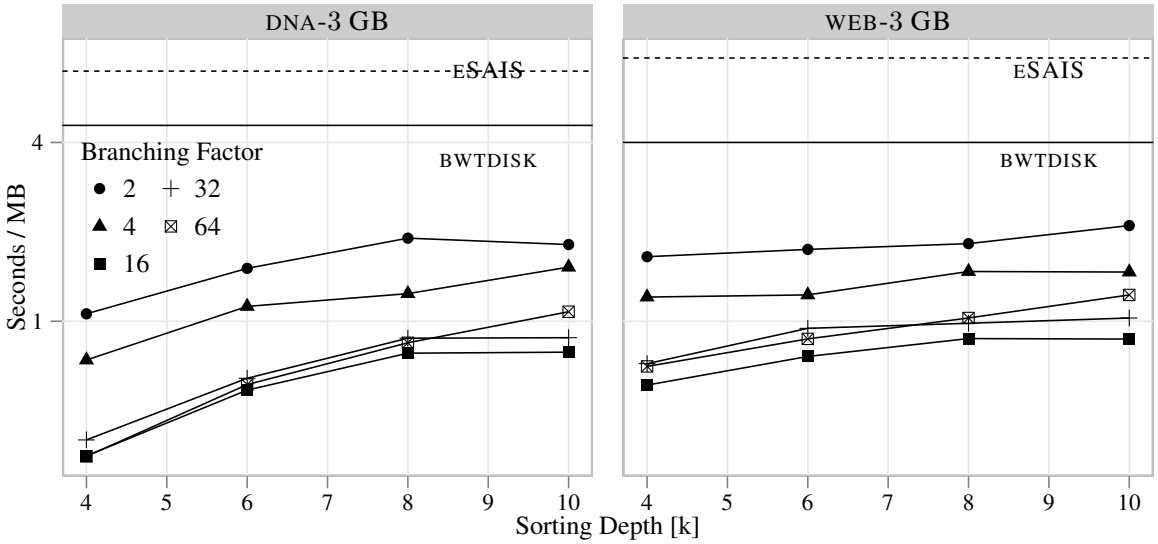

Figure 4.7: External k-BWT construction using different branching factors for WEB-3 GB *and* DNA-3 GB. *Time is shown in seconds per megabyte of input data to take into account the cost of disk I/O.*

for our new construction algorithm. The results of the experiment are shown in Figure 4.7. The time in seconds normalized by the input text size for different k is shown on the y-axis. The k-BWT can be constructed faster than the full BWT with both external memory algorithms. The k-BWT construction is roughly 4 times faster than the fastest BWT external construction algorithm. When comparing to the full BWT construction algorithms, the sorting depth k does not significantly affect the performance of our algorithms. While the algorithm is faster for small k, it remains constant for $k = 8, 10$. This will however change as we further increase k as the number of possible context groups grows exponentially ($|\Sigma|^k$) in the worst case. The branching factor affects the running time of our algorithm. We split each file into $n' = 3072/100 = 31$ blocks. For $\gamma = 2$, we perform $\log_2(31) = 5$ merge phases. For $\gamma = 4$ we only perform 3 merge phases. For all other branching factors we only perform one merge phase. Surprisingly, the "older" BWTDISK algorithm outperforms the more recent external suffix array construction algorithm ESAIS. Figure 4.8 shows additional results only using $\gamma = 16$ and $kblock = 100$ MB but larger values of k. Here the sorting depth for which the BWT can be constructed more efficiently than the k-BWT is shown for both data sets. For DNA-3 GB, BWTDISK begins to outperform external k-BWT construction between $k = 16$ and 32. ESAIS only outperforms the k-BWT construction algorithm for k larger than 64. For the WEB-3 GB data set, k-BWT construction is faster than both BWT construction methods up to $k = 64$.

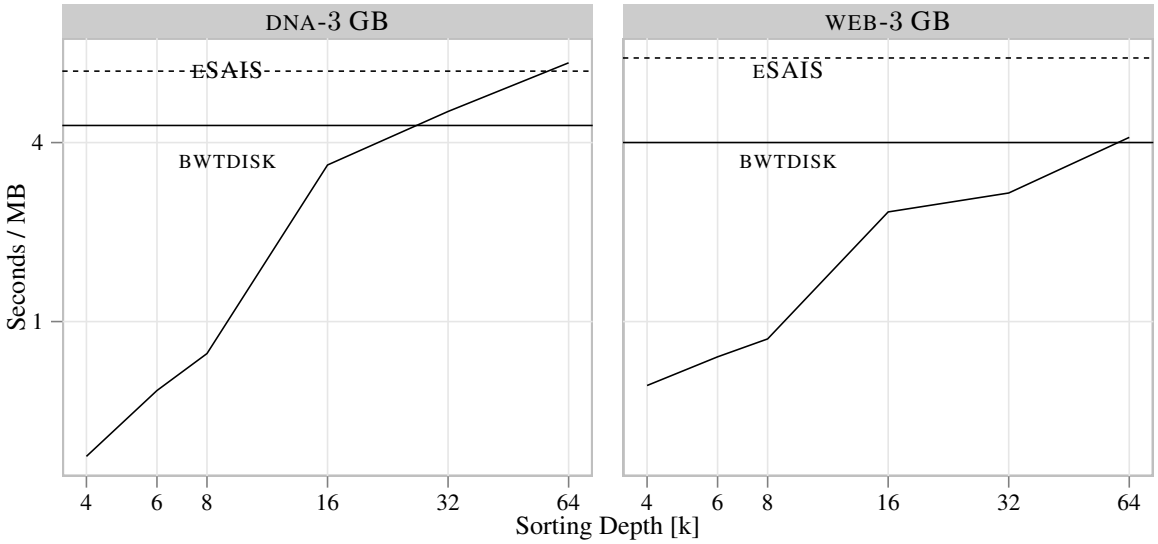

Figure 4.8: Extern k-BWT construction for larger values of k for WEB-3 GB *and* DNA-3 GB *using* $\gamma = 16$ *and* $kblock = 100$ *MB showing the trade-off where* BWT *construction in external memory outperforms external k-BWT construction.*

The disparity between the performance for the different data sets can be explained by the number of context for larger values of k. Consider the properties of the DNA-3 GB and WEB-3 GB data sets in Table 4.1. The median sorting depth required ($\text{LCP}_{\text{median}}$) to create context groups of size one for DNA-3 GB is approximately 15. Therefore, for sorting depths larger than 30, the k-BWT closely resembles the BWT, and can no longer be constructed more efficiently using the external k-BWT construction algorithm. However, the median sorting depths of WEB-3 GB is much larger. Therefore, as shown in Figure 4.8, constructing the k-BWT for large values of k can be performed efficiently for WEB-3 GB as there are exist still large context groups which can be merged efficiently using our algorithm.

Next the running time of our algorithms is evaluated for increasing input sizes. The k-BWT is constructed for input prefixes from 1 GB to 10 GB of the WEB data set for $k = 2, 4, 6, 8, 10$. This experiment uses the initial block size of $kblock = 100$ MB and the branching factor of $\gamma = 16$ during the merging phase. Figure 4.9 shows the results of the experiment. For WEB-1 GB and WEB-2 GB the running time is similar for all k. As the input size increases, the difference between the running time for the different sorting depths increases. This is caused by the increasing number of context groups for larger k. More context groups impose an additional merging cost, which outweighs the additional time required to perform the initial k-sort. Overall the running time increases linearly with the

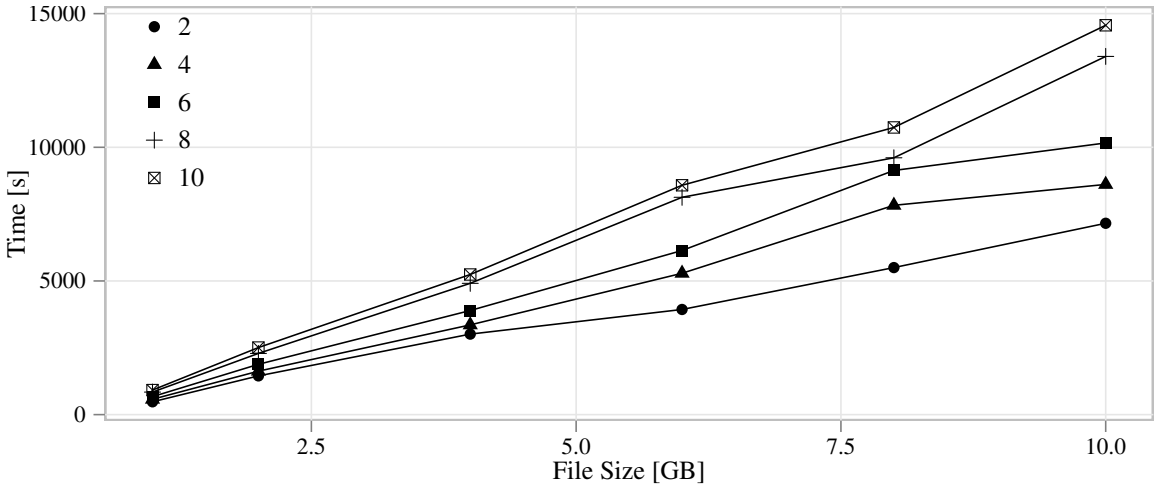

Figure 4.9: External k-BWT construction cost of the WEB *data set for prefixes of size* 1 *GB to* 10 *GB and* $k = 2, 4, 6, 8$*. In this experiment we use* $kblock = 100$ *MB and* $\gamma = 16$*.*

size of the input data for all sorting depths measured. This will change as the number of merge phases increases, where we expect to see a significant decrease in run time performance. For an input text of size 25700 MB and an initial block size of $kblock = 100$ MB, a total of $\log_{\gamma=16}(25700/100) = 3$ merge phases would be performed instead of two for all smaller inputs.

The k-BWT can be efficiently constructed using a simple external γ-way merge algorithm. The k-BWT can be constructed for large inputs using only a fixed amount of memory. The construction cost increases as the sorting depth increases due to the increasing number of context groups. Unfortunately, as k increases, the number of context groups is not bound and, in the worst case, our algorithm would merge context groups of size 1. This is however not the case in practice where our algorithm outperforms state-of-the-art external BWT and suffix array construction algorithms by a factor of four.

4.3 Reversing Context-Bound Text Transformations

In this section we discuss the reverse k-BWT transform. Reversing the regular BWT transform is briefly revisited to refresh the notation used throughout this section. Next the reverse k-BWT transform is formally introduced. Last the efficiency of different aspects of the k-BWT reversal algorithms is empirically evaluated.

4.3.1 Reversing the Regular Burrows-Wheeler Transform

The BWT can be reversed in linear time without the need to store any additional information. Conceptually, this is done by partially reconstructing $\mathcal{M}$ from T^{bwt}. First, the first column (F) in $\mathcal{M}$ is recovered by performing counting sort on T^{bwt}, which represents the last column (L) in $\mathcal{M}$, in linear time. Thus $T^{bwt} = L[0]L[1]...L[n]$. To reverse the transform the position I, which corresponds to the row in $\mathcal{M}$ where the original string T appears, is also stored. Let *LF* be the mapping of each symbol in L to its corresponding position in F for $\mathcal{M}$. If row j of $\mathcal{M}$ contains rotation i then function $\textbf{PRED}(j)$ returns the row containing rotation $i-1$. Observe that the rows of $\mathcal{M}$ in Figure 4.1 (left) – up to the \$ symbol on each row – are in fact the suffixes of T in lexicographical order. Note the following important properties of $\mathcal{M}$ that are necessary for the reverse transform:

1. Given the i^{th} row of $\mathcal{M}$, the last character $L[i]$ precedes the first character $F[i]$ in the original text. So, $T = \ldots L[i]F[i]\ldots$.
2. Let $c = L[i]$ and let r_i be the frequency of symbol c in $L[0..i-1]$. If $\mathcal{M}[j]$ is the r_i th row of $\mathcal{M}$, starting with c, then the symbol corresponding to $L[i]$ in the first column is located at $F[j]$. As such, $L[i]$ and $F[j]$ correspond to the same symbol in the input string T since F and L are both sorted by the text *following* the symbol occurrences.

The BWT algorithm is reversible using the following procedure, which produces the original string T in reverse order.

1. Count the number of occurrences of each symbol in T^{bwt} and compute an array $Q[0..|\Sigma|]$, such that $Q[c]$ stores the number of occurrences of symbols $\{\$, 1, .., \sigma-1\}$ in T^{bwt} (and equivalently T). The count $Q[c]+1$ gives the first occurrence of the symbol c in F.
2. Next, construct the *LF* mapping, $LF[0..n]$, by making a pass over T^{bwt} and setting $LF[i] = Q[L[i]] + \textbf{OCC}(L, L[i], i)$, where the function $\textbf{OCC}(A, b, i)$ returns the number of occurrences of symbol b in string $A[0..i-1]$. The mapping *LF* precisely defines the **PRED** function.
3. Reconstruct T backwards as follows: set $s = I$ and for each $i \in n-1\ldots 0$ do $T[i] \leftarrow T^{bwt}[s]$ and $s \leftarrow LF[s]$.

4.3.2 Reversing the k-BWT

The key difference between reversing the full and partial transforms is the ease of implementing the **PRED** function. For $\mathcal{M}$ and the full BWT, $\textbf{PRED}(i)$ is easily derived from $LF[i]$. Let LF_k be the k-BWT equivalent last to first column mapping for $\mathcal{M}_k$. While $LF[i]$ represents $\textbf{PRED}(i)$ for the regular

BWT, LF_k does not necessarily represent **PRED**. The mapping $LF[i]$ returns the position in $\mathcal{M}$ of the rotation $j = \mathcal{M}[i]-1$ in the fully sorted BWT, which is also the row containing the rotation preceding shift $\mathcal{M}[i] - 1$. Unfortunately, $LF_k[i]$ provides no such guarantee for the predecessor. It only points to a row with a rotation that shares the same k-prefix as the true predecessor, or all rows k-equal to the true predecessor. For example, consider $LF_k[10] = 8$ in Figure 4.1. Observe that row 8 is not the predecessor of rotation 10 – the actual predecessor is 9. However, rows 8 and 9 do share a common prefix `ch` of length $k = 2$. Using only LF_k would recover the text "`$chaca$chaca$`" instead of "`chacarachaca$`". During the reconstruction of T from T^{kbwt}, using LF_k, the character \$ is processed first instead of the correct choice `a` as LF_k only points to the preceding *context* and not the correct preceding character or row.

Nevertheless, LF_k can be used to properly simulate **PRED** as in LF. However, the boundaries of the k-groups in $\mathcal{M}_k$ previously defined (see Definition 7) as the bitvector D_k are also required in the process. This method was originally described by Schindler [1997]. For now, we assume the context group boundaries (D_k) are available before reconstruction without loss of generality.

Recall that the rotations are defined to be in ascending order (due to the stability of the sorting process) within each k-group. During reversal, the rows in a given k-group must be visited last to first, by virtue of the fact that the string is recovered from the last character to the first. The complete reversal algorithm for k-BWT is shown below:

```
1  reverseKBWT(T^kbwt, I, n) {
2      j = I
3      for i = n − 1 to 0 do {
4          T[i] = T^kbwt[j]
5          g ← GROUP(j)
6          j ← PRED(j) ≡ g − GROUPPOS[g]
7          GROUPPOS[g] = GROUPPOS[g] + 1
8      }
9      return T
10 }
```

As previously discussed, LF_k cannot be used to implement the **PRED** function directly as $LF_k[j]$ only guarantees to point to a symbol in the same k-group. So, the correct predecessor must be derived from the current k-group. Therefore, the preceding k-group g ($g =$ **GROUP**(j)) is located. Then, the correct preceding character j within the k-group is determined by maintaining the count of all

of the previously decoded characters within g (**GROUPPOS**(g)). All known methods for reversing the k-BWT use a variation of this general procedure. The methods only differ in how the k-group boundaries (the D_k vector) are reconstructed prior to recovering T from T^{kbwt}.

4.3.3 Recovering the k-group Boundaries

Here a variation of the simple $\mathcal{O}(nk)$ algorithm of Schindler [1997] to recover D_k from T^{kbwt} is described. First recover the $k = 1$ context group boundaries (D_1) from T^{kbwt} by performing count sort. From D_1 the context bounds for sorting depth $k = 2$ (D_2) can recovered as follows. With each pass over T^{kbwt}, the sorting depth is increased by one. Each context group C_i in D_1 is processed as follows: While processing C_i, the first occurrence of each symbol c is recorded. Consider processing the context group C^a the running example (see Figure 4.1). In row 1 in $\mathcal{M}_k$ symbol $T^{kbwt}[1] = c$ occurs for the first time in C^a. This implies that there is a $k = 2$ context group C^{ca}. Therefore, $D_2[6] = 1$ is set to one as the start of all c contexts begins in row 6 and $T^{kbwt}[1]$ is the first time symbol c is encountered within C^a. This implies that the first context of order two in $\mathcal{M}_k$ is C^{ca}. The second time symbol c occurs in C^a is in row 5 as $T^{kbwt}[5] = c$. Therefore, the context C^{ca} is of length two and $D_2[7] = 0$. By performing k passes over T^{kbwt} the k-deep context boundaries (D_k) can be recovered in $\mathcal{O}(nk)$ time. A formal description of the algorithm is shown below:

```
1  recoverDk(T^kbwt, n) {
2      Q[0...σ−1] = CumCounts(T^kbwt, n) , D_1 = ExtractBounds(Q)
3      for j = 2 to k do {
4          CT[0...σ−1] = Q[0...σ−1]
5          LastSeen[0...σ−1] = −1
6          for i = 0 to n−1 do {
7              if D_{j−1}[i] == 1
8                  ctxstart = i
9              if LastSeen[T^kbwt[i]] < ctxstart
10                 D_j[CT[T^kbwt[i]]] = 0
11             else
12                 D_j[CT[T^kbwt[i]]] = 1
13             CT[T^kbwt[i]]] = CT[T^kbwt[i]] + 1
14             LastSeen[T^kbwt[i]] = i
15         }
16     }
17     return D_k
18 }
```

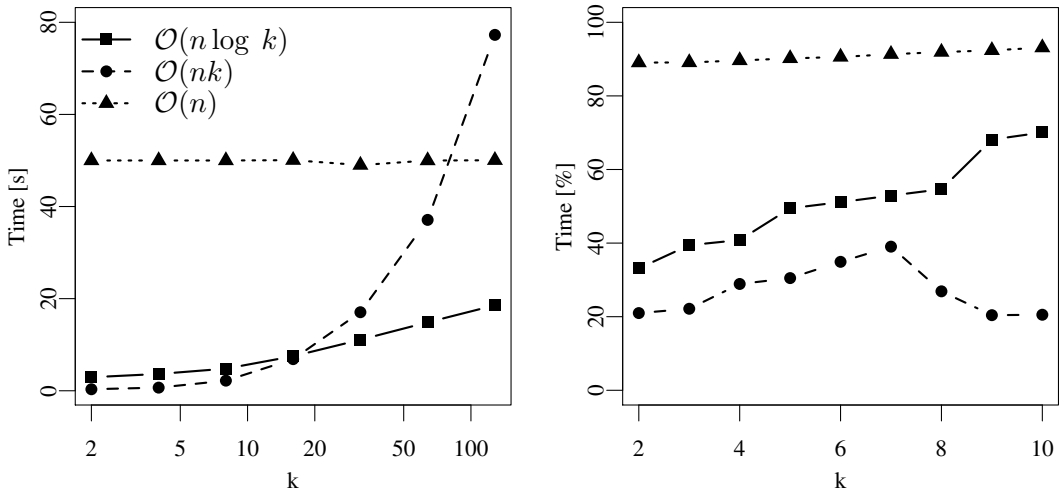

Figure 4.10: Context reconstruction time in seconds for the DNA *data set for variable k for all k-*BWT *algorithms (left) and as a percentage of the total reconstruction time (right).*

Recently, Nong and Zhang [2007] described an asymptotically more efficient method for computing D_k which requires $\mathcal{O}(n \log k)$ time. The key observation made by Nong and Zhang is that a k-order context is composed of two $(k/2)$-order contexts. Due to the cyclic properties inherent to the permutation matrix, the $(k/2)$-order context positions can be extracted from the previous iteration. Using a dynamic programming approach, the algorithm uses $D_{i/2}$ to produce D_i, where $i \leq k$, to recover D_k in $\mathcal{O}(n \log k)$ time. Nong et al. [2008] also propose an algorithm to reduce the time for recovering the k-order contexts to $\mathcal{O}(n)$ by computing the lengths of longest-common-prefixes (lcp) for each adjacent row in $\mathcal{M}_k$. The k-order contexts can then be retrieved by simply traversing the list of lcp values and marking a k-order context boundary each time $lcp \geq k$. Figure 4.10 compares only the cost of context reconstruction for each of these algorithms. The left subfigure shows the aggregate time for reconstruction and the right subfigure shows the percentage of total runtime consumed during context reconstruction for DNA. Note the superior performance of the $\mathcal{O}(nk)$ algorithm for even moderate values of k. For $k = 100$, the $\mathcal{O}(n \log k)$ algorithm is still more than twice as fast as the linear variant. The constant factors of $\mathcal{O}(n)$ algorithm are not overcome until k is very large. This is further evaluated empirically in Table 4.2.

The context reconstruction time is independent of the text input, unlike the compression effectiveness and inverse transform efficiency. As shown in Figure 4.10 (right), context reconstruction time using the fastest practical $\mathcal{O}(nk)$ algorithm is still a significant contributor to the overall time complexity of k-BWT inversion. As can be observed in Figure 4.10, while the total cost of recovering

the context bounds for the $\mathcal{O}(nk)$ algorithm increases (left), the percentage of the total recovery cost decreases for $k > 7$. This implies that the actual cost of recovering T from T^{kbwt} using D_k *decreases* for larger k. Upon further investigation we discovered a cache effect during the actual reversal procedure of the *k*-BWT. This cache effect is shown in Figure 4.13 and will be discussed in more detail in Section 4.3.4. Overall, the reconstruction of the context boundaries contributes up to 40% to the total running time of reconstructing T from T^{kbwt}. Thus, methods which can alleviate this cost merit further consideration.

Consequently, storing the contexts explicitly instead of computing the bounds during the reverse transform is a sensible space-time trade-off to consider. For small k, the number of distinct contexts is small, and, can thus be stored efficiently. The contexts can be represented naively as a bitvector of size n bits, but the vector is sparse for small values of k, making the context information more compressible. A pragmatic choice to explicitly encode the context group boundaries is compressed d-gap encoding. It is used to store the one bit positions in D_k explicitly, as the method is highly effective when the number of contexts is much smaller than n [Culpepper and Moffat, 2005]. Furthermore, the sequential decoding limitation imposed by d-gap encoding is not problematic since random access to the context vector during reconstruction is not required. The additional cost, measured in loss of entropy as a result of having to store additional information, to explicitly store the contexts relative to the cost of reconstructing the context information using the $\mathcal{O}(nk)$ reconstruction algorithm is shown in Figure 4.11. For WSJ, XML and SOURCES, the cost of storing context information is negligible for $k < 6$. Context information for the DNA collection can be stored effectively for $k < 10$. Overall, there is no significant loss in compression effectiveness for moderate values of k.

Another important dimension to consider is the working space required by the inversion process. Table 4.2 shows the time and space requirements with constant factors, for all context reconstruction algorithms, including our explicit storage algorithm. The empirical space usage is estimated using the program MEMUSAGE available from the PIZZA & CHILI CORPUS website. The $\mathcal{O}(n \log k)$ and $\mathcal{O}(n)$ algorithms require up to five times more space than the $\mathcal{O}(nk)$ algorithm to recover the original text. This space is further reduced by eliminating the context reconstruction entirely, such as is done by our explicit context boundary encoding algorithm. Our algorithm requires only $5n$ space and can be inverted in $\mathcal{O}(n)$ time, but is only preferable when k is small as the cost of storing D_k increases with k.

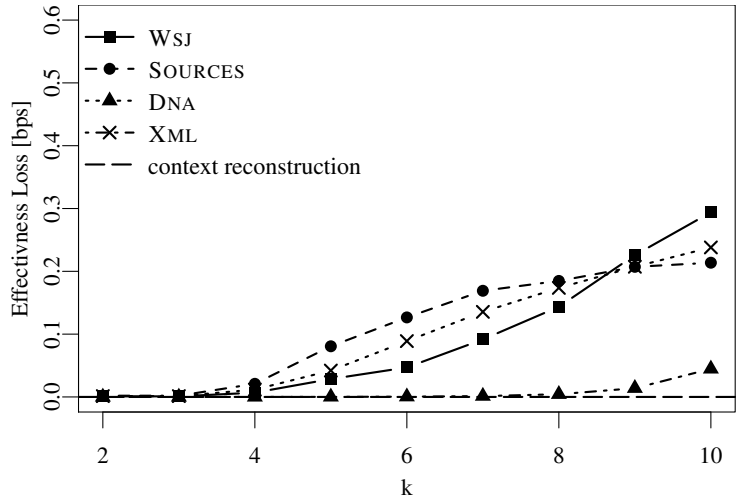

Figure 4.11: Entropy loss resulting from explicitly storing the contexts with d-gap encoding relative to reconstruction.

	Theoretical		Empirical
Algorithm	Time	Space	Space(bytes)
[Schindler, 1997]	$\mathcal{O}(nk)$	$\mathcal{O}(n)$	$5n$
[Nong and Zhang, 2007]	$\mathcal{O}(n \log k)$	$\mathcal{O}(n)$	$22n$
[Nong et al., 2008]	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$26n$
Our Algorithm	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$5n$

Table 4.2: Time and space bounds for inverse k-BWT context reconstruction.

4.3.4 Inverse Transform Efficiency

The recovery of the original string from T^{bwt} and T^{kbwt} is similar. The fundamental difference between BWT and k-BWT inversion is the need to recover context boundaries. As discussed in Section 4.3.3, several methods exist to recover the original context information. Intuitively, we expect BWT inversion to be more efficient than any of the k-BWT methods, and the $\mathcal{O}(nk)$ k-BWT algorithm to be less efficient than the $\mathcal{O}(n)$ or $\mathcal{O}(n \log k)$ algorithms. This is not the case in practice as discussed in the previous Section. We therefore use the fastest practical $\mathcal{O}(nk)$ k-BWT context recovery algorithm in conjunction with the k-BWT reversal algorithm described in Section 4.3.2 to evaluate the efficiency of the reverse k-BWT transform relative to the efficiency of the full BWT.

Figure 4.12 shows the efficiency of the basic $\mathcal{O}(nk)$ context recovery algorithm in conjunction with the k-BWT reversal algorithm discussed in Section 4.3.2. The figure shows the efficiency of the k-BWT reversal procedure *relative* to the efficiency of the full BWT algorithm. For small values of k,

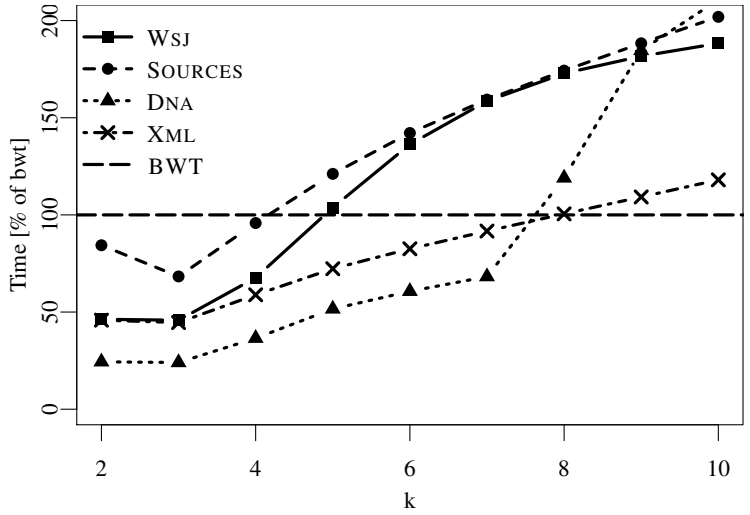

Figure 4.12: Reverse-transform efficiency of k-BWT *using the* $\mathcal{O}(nk)$ *algorithm for* WSJ, SOURCES, DNA *and* XML *in time relative to* BWT *for each respective file.*

the inverse k-BWT outperforms BWT, regardless of text input. The inverse k-BWT is up to three times faster than BWT inversion. The exact performance depends on the composition of the collection, but clearly k-BWT inversion can be more efficient than BWT inversion.

Locality of Access in the Inverse k-BWT

The efficiency of k-BWT is contradictory to previous results and somewhat surprising as the k-BWT algorithm performs additional work to reconstruct context information during inversion. Further experimental investigation into the discrepancy revealed a significant caching effect when k is small. In order to quantify the unexpected efficiency gain, the locality of access in the k-BWT inversion process is evaluated next. Cache performance is measured using the PAPI library described in Section 2.7.2.

Figure 4.13 shows the percentage of cache misses for variable k-order contexts during the reconstruction of the original text for each test collection. The cache misses are shown *proportional* to BWT inversion for identical files. Both algorithms perform the same task – reconstructing the text using the *LF* mapping – but, exhibit a profound difference in cache behavior. For low order contexts, k-BWT has 90% fewer cache misses than BWT. This remarkable difference is a natural by-product of k-BWT inversion. During inversion, the algorithm jumps between different contexts, but each partially sorted context is encoded by increasing position in the original text. The inherent locality of reference in each context allows the operating system to cycle between each context with fewer page-faults, resulting in fewer overall cache misses.

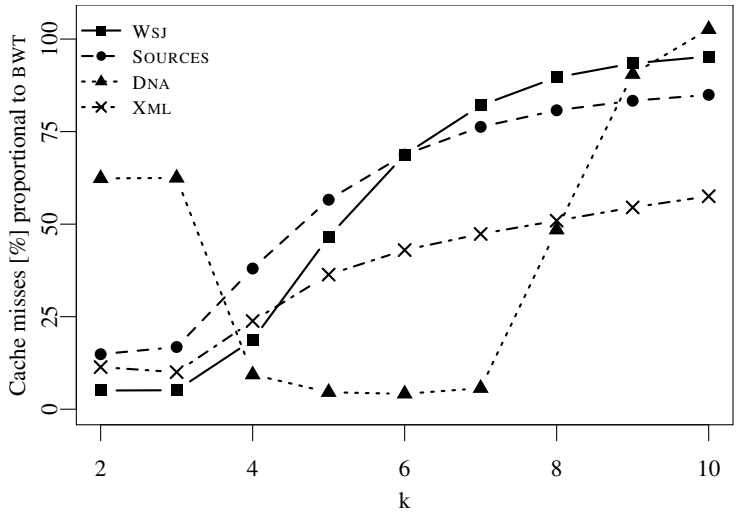

Figure 4.13: Cache misses for variable k in percent proportional to the cache misses of BWT *for the respective file.*

However, this observation does not hold true for DNA, where the cache misses for $k < 4$ are higher than for $k \geq 4$. This in contrast to the alternative collections where smaller values of k always result in better cache performance. Upon further investigation, our experiments show that k-BWT algorithms do not benefit from the cache effect when $|\Sigma|$ is very small. For small k and $|\Sigma|$, the number of cache misses is unusually high in DNA. Since the total number of contexts is bound by $|\Sigma|^k$, the cache can not be fully utilized: each context has only a page size p_s of data pre-fetched in the cache. Once this data segment is processed, a cache miss occurs, and a new page is loaded. At most $p_s \cdot |\Sigma|^k$ data is cached at any given time, resulting in more cache misses as data is processed using fewer distinct memory pages. Therefore, balancing the number of distinct contexts and available memory pages can have a dramatic impact on overall efficiency. So, when $|\Sigma|$ is small, using larger values of k is desirable to allow more context groups to be cached.

4.4 Context-Bound Text Transformations in Data Compression

Last we evaluate the compression performance of the k-BWT compared to the regular BWT. We first focus on effectiveness for different values of k. Finally we give a comprehensive time and space trade-off analysis to conclude the chapter.

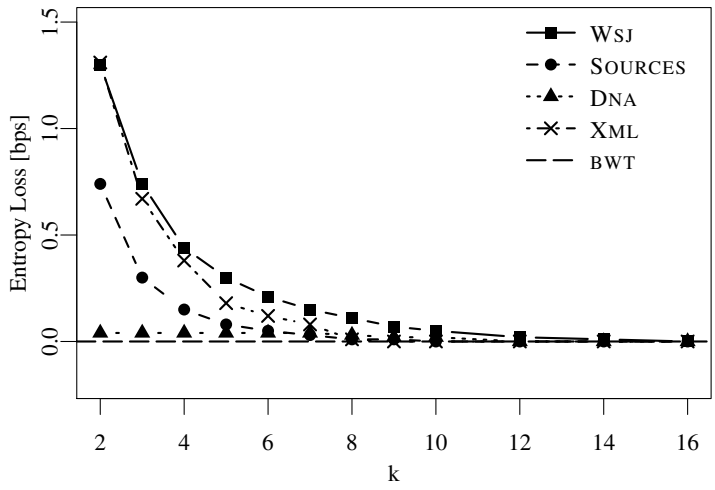

Figure 4.14: Effectiveness of k-BWT *for* WSJ, SOURCES, DNA *and* XML. *Effectiveness is measured in entropy loss relative to* BWT.

4.4.1 Compression Effectiveness

The BWT algorithm groups symbols with similar context, allowing for more effective text compression. The transform is the first of a series of steps in a typical data compression system as described in Section 2.4.2. The k-BWT algorithm sorts the permutation matrix to a depth of k, and symbols with similar k-order contexts are grouped together. Consecutive contexts with an $lcp > k$ will not necessarily be exploited in k-BWT (but they may be by chance). Therefore, a k-BWT compression system can be less effective than a system using BWT.

First, the loss in compression effectiveness of the k-BWT relative to the sorting depth k is evaluated. Effectiveness is measured in bits per symbol (bps) using the zero-order empirical-entropy $\mathcal{H}_0$ (defined in Section 2.1). The effectiveness is measured by replacing the BWT with the k-BWT in a standard transform-based compression system [Burrows and Wheeler, 1994]. In the second stage of the compression system a standard Move-To-Front transformation (MTF) is used [Bentley et al., 1986]. Figure 4.14 shows the zero-order empirical-entropy after the MTF step for increasing values of k. The effectiveness is reported as the entropy loss in bits per symbol *relative* to BWT. As k increases, the performance of k-BWT approaches the effectiveness achievable using a fully sorted BWT. For $k = 6$, the limited context transform achieves nearly identical compression effectiveness. For DNA, the performance of k-BWT is independent of k. This is unsurprising since DNA compression usually requires additional symbol model steps to improve compression effectiveness [Apostolico and Lonardi, 2000; Chen et al., 2000]. The WSJ, SOURCES and XML collections can be compressed

effectively using contexts as low as order-5. Overall, using a limited order context transform supports efficient compression, and achieves impressive compression effectiveness, even when k is small.

Next the compression effectiveness of full compression systems in conjunction with the k-BWT is evaluated. Each file is first transformed using BWT or k-BWT, then compressed with MTF and RLE [Burrows and Wheeler, 1994], or compression boosting (BOOST) [Ferragina et al., 2005]. In the next step a variety of standard entropy coders are then applied to compress the transformed input. In this experiment a Huffman coder (HUFF) [Huffman, 1952], a Range Coder (RANGE) [Schindler, 1998] and an Arithmetic coder (ARITH) [Witten et al., 1986] are used. Table 4.3 shows the compression effectiveness in bits per symbol and efficiency in seconds of k-BWT for $k = 2, 4, 8, 10$, boosting and BWT using the different compression systems. All compression systems show similar compression effectiveness relative to sorting depth. Huffman encoding is more efficient than Range or Arithmetic coding, but results in worse effectiveness. Compression boosting using the BWT outperforms all systems in terms of effectiveness, but is an order of magnitude slower than all other combinations. When $k > 6$, the effectiveness loss is below 10 percent in k-BWT-based compression systems. When $k > 8$, the compression loss drops less to than 3 percent. The experiments show that for small k, compression effectiveness is close to that of BWT, but is much more efficient.

4.4.2 Inverse Transform Effectiveness and Efficiency Trade-offs

The k-BWT can achieve near identical compression results as the BWT for small values of k. Next the effectiveness and efficiency trade-offs of different inversion algorithms are evaluated. The following algorithms are compared: the BWT, Schindler's $\mathcal{O}(nk)$ context recovery algorithm in conjunction with the k-BWT, and the $\mathcal{O}(n)$ algorithm which explicitly stores context information in conjunction with the k-BWT. The $\mathcal{O}(n \log k)$ and $\mathcal{O}(n)$ k-BWT context reconstruction algorithms of Nong et al. are not evaluated as they are always less efficient than Schindler's $\mathcal{O}(nk)$ algorithm when $k < 10$ (see Section 4.3.3). Figure 4.15 summarizes trade-offs between efficiency and effectiveness of the various inversion algorithms. Note that each sub-graph is scaled individually to increase readability.

As k gets large, the cost of storing an increasingly dense context vector begins to offset any gains that might be achievable by sorting to a higher k for our $\mathcal{O}(n)$ algorithm. The WSJ test collection can be decoded twice as fast using k-BWT algorithms with little compression loss. The XML collection can be recovered up to 50% faster and approaches BWT effectiveness at $k = 4$. The SOURCES collection can be compressed effectively and efficiently using low order contexts and the DNA collection decompresses three times as fast at identical levels of effectiveness. Implicit context storage has negligible impact on compression effectiveness for small k for all test collections.

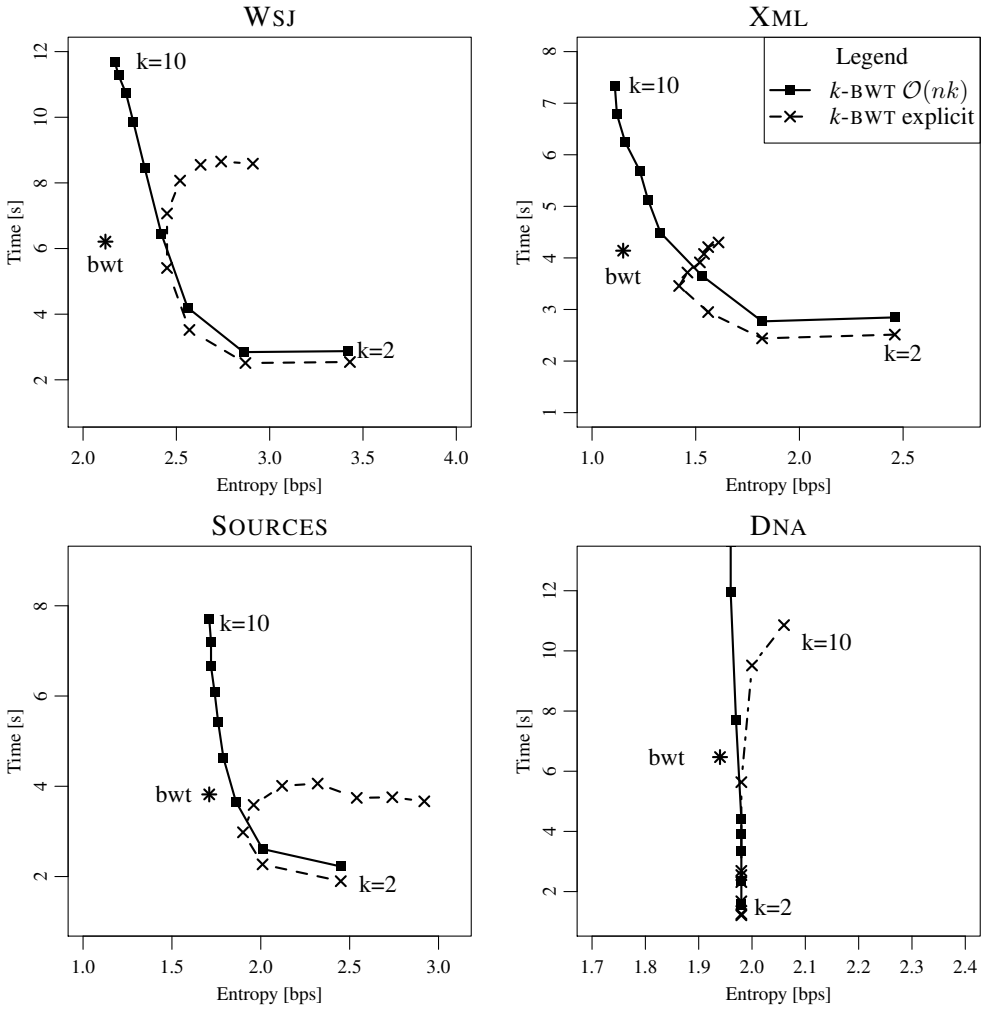

Figure 4.15: Efficiency vs effectiveness for block-sort inversions.

Context-bound transforms achieve similar compression effectiveness to full transform-based compression systems, but can compress faster. Figure 4.16 shows the effectiveness and efficiency trade-offs achieved using our k-bound forward transform for different compression systems. Note again that each sub-graph is scaled individually to increase readability. We compare the effectiveness and efficiency achieved for BWT and k-BWT compression systems where $2 \leq k \leq 10$. We also include the common compression systems BZIP, SZIP of [Schindler, 1997], and GZIP for reference. We do not show boosting as it is an order of magnitude slower than all other compression systems tested. Note the total I/O costs are included in this comparison to allow a fair comparison to the "off the shelf" compression systems.

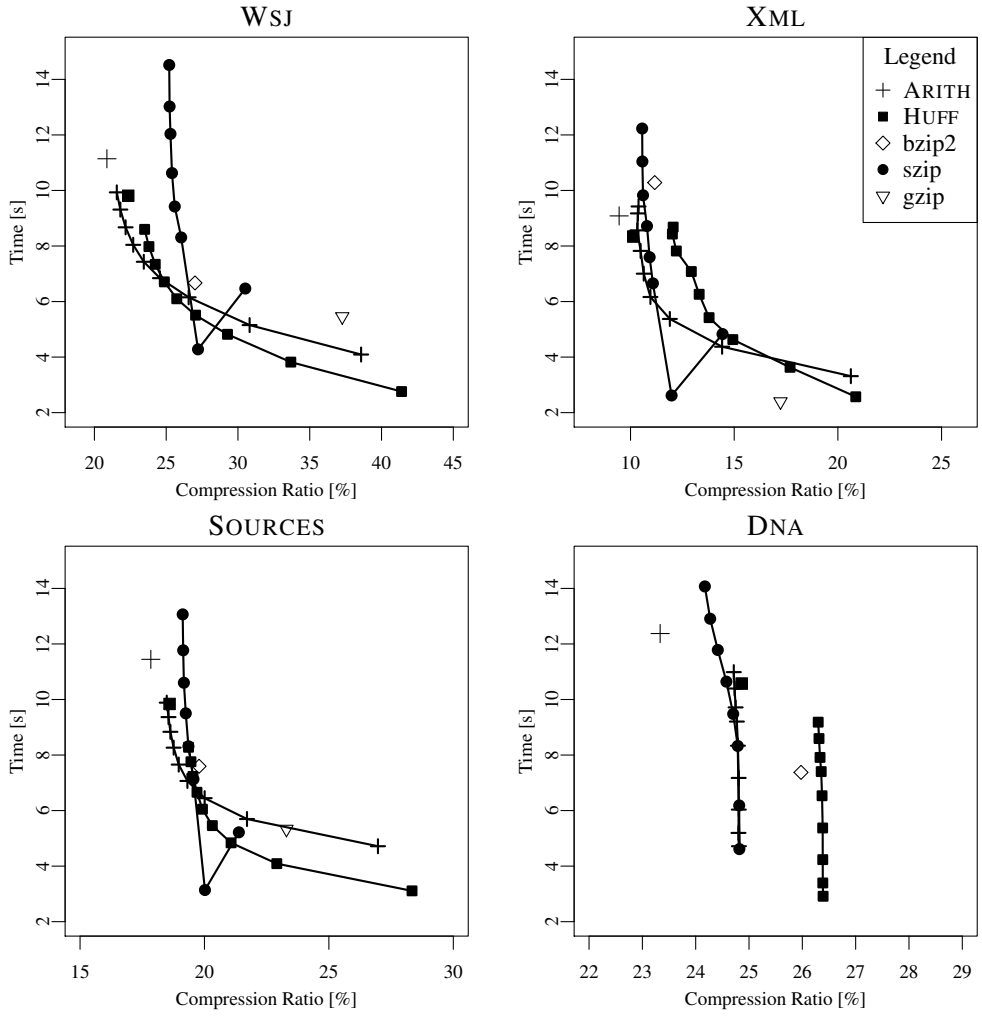

Figure 4.16: Efficiency vs effectiveness for bounded context-based transformation systems.

For WSJ, the k-BWT compressor is faster than GZIP and at the same time achieves better compression. All compression backends show a similar effectiveness and efficiency trade-off. The BWT system is shown in the graph as an unconnected point at the top of the curve as k-BWT systems approach the full BWT as k approaches n. The BWT compression systems achieve slightly better compression but are less efficient when $k < n$. The BZIP2 compressor is fast but does not achieve comparable compression effectiveness due to the fixed block size of 900 kB. The SZIP compressor is less efficient than our k-BWT system, for all k other than $k = 4$. For $k = 4$, SZIP uses a custom "super alphabet" method to only sort the text once, resulting in noticeably faster compression. This special case method can also be exploited in our compression system, but is an implementation detail

we do not explore here. The SOURCES and XML collections show similar behaviour. The k-BWT compressor achieves remarkable compression effectiveness on the XML collection even for small k, but this result is more an artefact of a specific, highly compressible collection than the compression system. The DNA collection is generally not compressible without more specialized modelling steps, and thus does not show any significant trade-offs. Note that for GZIP using the -9 parameter for best compression, no result is shown in the DNA graph as the running time is an order of magnitude larger than the other compressors. We also performed the same experiments for the LZMA-based compressor. The results are not shown in the graph as the graph as the running time are at least 3 times, and up to 11 time slower for DNA, slower than the next slowest other compressor shown in Figure 4.16 while achieving, on average, 3% better compression effectiveness.

4.5 Summary and Conclusion

The BWT is the main component of many succinct text indexes. Constructing the BWT is computationally expensive as a full suffix sort is performed in practice. Sorting all suffixes of a text in-memory requires up to nine times the space of the original text. This makes the construction of the BWT one of the major obstacle when scaling the size of succinct text indexes in practice.

In this chapter we revisited an alternative to the BWT, the k-BWT which can be constructed more space efficiently in external memory than the regular BWT. We performed an comprehensive evaluation of bounded context length block sorting showed that the k-BWT can be computed efficiently in-memory. We also proposed a new reversal algorithm which explicitly stores the context information to avoid expensive context reconstruction. Interestingly, by using a simple external merge-sort like algorithm, the k-BWT can be constructed efficiently using only a fixed amount of main memory. The external k-BWT construction algorithm outperforms existing state-of-the-art external full BWT and suffix array construction algorithms in our experiments. This makes the k-BWT especially interesting for large scale text-indexes and compression systems where constructing the full BWT or the full suffix array is not feasible. We further showed that existing, context recovery algorithms that are theoretically efficient do not perform well in practice due to large constant factors and space usage.

The k-BWT can be constructed more efficiently for small k in-memory as well in external-memory. Replacing the BWT with the k-BWT in succinct text indexes would allow larger indexes to be constructed. However, replacing the BWT transform is not straightforward. The FM-Index [Ferragina and Manzini, 2000] relies on a duality between the suffix array and the BWT to provide operations $count$, $locate$ and $extract$ in succinct space. Due to the incomplete lexicographical ordering of the k-BWT suffixes, the duality between the suffix array and the k-BWT is incomplete. In the

next chapter we investigate supporting the standard operations of a FM-Index using the k-BWT.

		$\mathcal{H}_0$	MTF-RLE					
			ARITH		RANGE		HUFF	
			bps	sec	bps	sec	bps	sec
WSJ	BWT	2.12	1.70	11.14	1.67	9.93	1.78	9.80
	k-BWT-2	3.42	3.08	4.09	3.08	2.88	3.31	**2.75**
	k-BWT-4	2.56	2.12	6.15	2.12	4.94	2.34	4.81
	k-BWT-8	2.33	1.77	8.67	1.77	7.46	1.89	7.33
	k-BWT-10	2.17	1.72	9.93	1.72	8.72	1.87	8.59
	BOOST	-	**1.65**	108.07	1.66	60.53	1.68	87.06
SOURCES	BWT	1.71	1.43	11.44	1.42	9.67	1.48	9.83
	k-BWT-2	2.45	2.15	4.71	2.16	**2.94**	2.26	3.10
	k-BWT-4	1.86	1.60	6.44	1.60	4.67	1.68	4.83
	k-BWT-8	1.72	1.50	8.83	1.49	7.06	1.56	7.22
	k-BWT-10	1.71	1.47	9.88	1.48	8.11	1.53	8.27
	BOOST	-	**1.65**	108.07	1.66	60.53	1.68	87.06
XML	BWT	1.15	0.77	9.08	0.77	8.35	0.81	8.34
	k-BWT-2	2.46	1.65	3.31	1.65	2.58	1.66	**2.57**
	k-BWT-4	1.53	0.95	5.37	0.95	4.64	1.12	4.63
	k-BWT-8	1.16	0.82	8.56	0.82	7.83	0.96	7.82
	k-BWT-10	1.15	0.82	9.42	0.82	8.69	0.94	8.68
	BOOST	-	**0.74**	84.06	0.75	48.79	0.75	72.67
DNA	BWT	1.94	1.89	12.37	1.86	10.69	1.98	10.57
	k-BWT-2	1.98	1.98	4.71	1.98	3.03	2.11	**2.91**
	k-BWT-4	1.98	1.98	6.03	1.98	4.35	2.10	4.23
	k-BWT-8	1.97	1.97	9.71	1.97	8.03	2.10	7.91
	k-BWT-10	1.96	1.97	10.98	1.97	9.30	2.10	9.18
	BOOST	-	**1.86**	99.90	1.86	58.23	1.96	92.14

Table 4.3: Effectiveness and efficiency for common transform-based compression system combinations. The $\mathcal{H}_0$ result is not shown for BOOST *since this method requires each context block to be compressed separately, resulting in effectiveness approaching $\mathcal{H}_k$.*

Chapter 5

Searching Context-Bound Text Transformations

Many self-indexing methods are derived from a *suffix array* (SA) [Manber and Myers, 1993]. A SA supports search over a text T by sorting all suffixes in lexicographical order. However, the SA occupies $n \log n$ bits of space which is required during search time in addition to the original text. This makes using the suffix array prohibitive when searching over large text collections. For example, searching in 5 GB large text using a suffix array requires 45 GB of main memory.

The BWT permutes a text T into a more compressible representation. Remarkably, the output of the BWT can be used in conjunction with a *wavelet tree* [Grossi et al., 2003] to emulate the search capabilities of a suffix array while using space proportional to the compressed is constructed over the BWT output so that jumps between lexicographically ordered suffixes in T are possible. Processing a pattern by jumping between suffix positions in the BWT is commonly referred to as *backwards search* [Ferragina and Manzini, 2000]. The duality between the SA and the BWT over T is therefore the key component in the compressibility and search functionality of most *succinct text indexes*. Unfortunately, constructing the BWT requires the SA to be constructed. Thus, while succinct text indexes are space-efficient, constructing them is still resource intensive due to the space required to construct the SA.

Prior to the discovery of *succinct text indexes*, suffix arrays and suffix trees were in theory the most efficient data structures to perform full-text search. From a practical perspective, a folklore method for reducing space called a k-gram index was commonly used for substring searches [Ullman, 1977; Sutinen and Tarhio, 1996; Puglisi et al., 2006]. A k-gram index records the occurrences of each distinct substring of length k in an attempt to mimic the efficiency of inverted indexes. While suffix

arrays (and the BWT) are often more convenient when searching for general patterns, there are various applications where using a k-gram index is still advantageous. Since text suffixes need to be sorted only up to the first k symbols, a k-gram index can be built using less space and time, and are more I/O-friendly than full suffix arrays. Also, searches for patterns of a fixed length k can be performed very efficiently, and are returned with all occurrences by increasing text position order.

An example where a k-gram index may be more convenient than a suffix array is for indexed approximate searching [Navarro et al., 2001a]. One can backtrack in a suffix array, but the time is exponential in the error threshold allowed. It is more efficient to split the pattern into subpatterns that are searched for with a lower threshold (or even exactly). Using this approach, the full search can be completed by backtracking or by generating a neighbourhood of all possible k-grams that match the subpattern within the error threshold. The approximate occurrences of the subpattern must be merged, and the occurrence lists for the distinct pieces are processed to find areas where an occurrence may be present. False matches must be filtered out from each possible occurrence using an on-line pattern matching algorithm. This process requires subpatterns of a fixed length k, and having potential matches returned in text position order is vital to efficient intersection and union operations. A suffix array needs additional query time and space to sort the possible occurrences in text order. In fact, one of the most prolific genomic search systems, BLAST, is reliant on a k-gram index and not SA-based algorithms for this reason [Altschul et al., 1990]. The k of interest in BLAST is around 11–12 for DNA, and 3–4 for protein sequences. However, the space cost to explicitly store all possible k-grams grows exponentially with k in the worst case, limiting the substring segment sizes used in practice.

In chapter 4 we explored construction and trade-offs related to the k-BWT. We showed that the k-BWT can be constructed more efficiently in external memory for large text collections compared to the regular k-BWT. However, in addition to constructing the k-BWT for large data, *searching* in a k-BWT transformed text is non-trivial. In this chapter, we explore a variant of succinct text indexes derived from k-BWT. In the BWT, the symbols in T are fully sorted in lexicographical order of contexts. The k-BWT only sorts the suffixes in T up to a certain length k. Here we explore the potential of the k-BWT as a succinct text indexes representation of text that offers k-gram index search capabilities. Our contributions and the structure of this chapter can be summarized as follows:

1. First we briefly refresh the *backward search* procedure to introduce notation in Section 5.1.

2. We present the first backwards search algorithm for k-BWT permuted text in Section 5.3.

3. We show that it is possible to search for patterns in the same way as in a k-gram index, retriev-

ing the occurrences in text position order.

4. We also demonstrate the recovery of any substring of the original text. Surprisingly, operations within k-BWT appear to be significantly more challenging than on a full BWT, and so our solution trades some compression effectiveness when k is small.

5.1 Backwards Search in the BWT

Here we briefly review the BWT transform and the *backward search* procedure used to emulate search in a suffix array (SA) using the BWT which is described in more detail in Sections 2.4.1 and 2.5.2. We further introduce the notation used throughout this chapter.

The BWT produces a permutation of a string T of size n over an alphabet of size σ, denoted T^{bwt}, by sorting the n cyclic rotations of T into full lexicographical order, and taking the last column of the resulting $n \times n$ matrix $\mathcal{M}$ to be T^{bwt}. To produce T^{bwt} for a given text T, it is not necessary to construct $\mathcal{M}$ as there is a duality between T^{bwt} and the SA over a text T: $T^{bwt}[i] = T[\text{SA}[i] - 1 \bmod n]$.

The original text T can be recovered from T^{bwt} in linear time. To recover T from only T^{bwt} we first recover the first column, F, in $\mathcal{M}$ by sorting the last column ($L = T^{bwt}$), in lexicographical order. By mapping the symbols in L to their respective positions in F with $L[i] = F[j]$ (usually referred to as the LF mapping, $j = \text{LF}(i)$), we can recover T backwards as $T[n-1] = T^{bwt}[0] = \$$ and $T[j-1] = T^{bwt}[\text{LF}(i)]$ if and only if $T[j] = T^{bwt}[i]$. The LF mapping is computed using the equation

$$\text{LF}(i) = \text{LF}_c(i, c) = Q[c] + rank(T^{bwt}, i, c)\ . \tag{5.1}$$

where c is the symbol $T^{bwt}[i]$, and $Q[c]$ stores the number of symbols in T^{bwt} smaller than c. Using the LF mapping we can recover T starting with position of symbol \$ in T^{bwt}. This is described and visualized in detail in Section 2.4.1.

5.1.1 Searching in the BWT

Performing a search in BWT is performed by processing the pattern in reverse order. The algorithm is commonly referred to as *backwards search* and can be summarized as follows. Recall that all rows are sorted in lexicographical order in $\mathcal{M}$. Therefore, for a pattern P, all occurrences of P in T must have a corresponding row in $\mathcal{M}$ within a range $\langle sp, ep \rangle$, where $T[\text{SA}[sp], \text{SA}[sp] + m - 1] = T[\text{SA}[ep], \text{SA}[ep] + m - 1] = P$. That is, the rows in $\mathcal{M}$ are prefixed by P. The backwards search

procedure determines the $\langle sp_m, ep_m \rangle$ of rows in $\mathcal{M}$ prefixed by P. To determine the range within $\mathcal{M}$, we first determine the range $\langle sp_m, ep_m \rangle$ within $\mathcal{M}$ that corresponds to $P[m-1]$ using $Q[P[m-1]]$. Then, for each symbol $j = m-1$ down to 0 in P, we iteratively find $\langle sp_j, ep_j \rangle$ by calculating the number of rows within $\langle sp_{j+1}, ep_{j+1} \rangle$ that are preceded by the symbol $P[j]$ in T. For a given row j, the LF mapping can be used to determine the row in $\mathcal{M}$ representing the symbol preceding j in T. The preceding row is determined by counting the number of occurrences of $c = T^{bwt}[j]$ *before* the current row and ranking these occurrences within $Q[c]$. Assume we have located $\langle sp_{j+1}, ep_{j+1} \rangle$, which corresponds to the rows prefixed by $P[j+1, m]$. Then

$$sp_j \;=\; \text{LF}(sp_{j+1} - 1, P[j]) + 1, \tag{5.2}$$

will calculate the position in F of the first occurrence of $P[j]$ within $\langle sp_{j+1}, ep_{j+1} \rangle$, and thus compute the start of our range of rows within $\mathcal{M}$ that correspond to $P[j, m]$. Similarly, we compute

$$ep_j \;=\; \text{LF}(ep_{j+1}, P[j]). \tag{5.3}$$

This process is visualized and described in detail in Section 2.5.2. The function LF can also be used to recover the original text or any substring $T[i..j]$. This is referred to as the *extract* operation. Additionally, the suffix array values of each element in $\langle sp, ep \rangle$ can be extracted using the LF in conjunction with a sampled version of the suffix array. This is referred to as the *locate* operation. Both operations are discussed in detail in Section 2.5.2.

5.2 Context-Bound Text Transformations

Here we briefly review the k-BWT and the notation used in the rest of this chapter. The k-BWT is a variation of the BWT which partially sorts the n rotations based on k length prefixes. Unlike the full BWT, the k-BWT only sorts the permutation matrix $\mathcal{M}$ up to a depth k ($\mathcal{M}_k$). The transform itself has been described in detail in Sections 2.4.3 and 4.1.2.

Figure 5.1 shows the $\mathcal{M}_2$ rotations of the k-BWT for the string $T =$ `chacarachaca$` and $k = 2$, producing T^{kbwt}. Due to the fixed sorting depth, multiple suffixes can be treated as k-equal during the sorting stage. These suffixes are grouped in *context groups* or k-groups, where suffixes in the k-sorted suffix array (SA_k) are stored in ascending order according to their position in T for each context group. The k-group boundaries can be can be marked in a bitvector, D_k. For our example shown in Figure 5.1, D_k is 1110011010101. The bitvector is formally defined in Definition 7 in Section 4.1.2.

D_k	SA_k	SA	F	T^{kbwt}
1	12	12	\$	a
1	11	11	a \$	c
1	2	9	a c a r a c h a c a \$	h
0	6	2	a c h a c a \$	h
0	9	6	a c a \$	r
1	4	4	a r a c h a c a \$	c
1	3	10	c a r a c h a c a \$	a
0	10	3	c a \$	a
1	0	7	c h a c a r a c h a c a \$	a
0	7	0	c h a c a \$	a
1	1	8	h a c a r a c h a c a \$	c
0	8	1	h a c a \$	c
1	5	5	r a c h a c a \$	a

Figure 5.1: Example k-BWT permutation matrix $\mathcal{M}_k$ for $k = 2$ of the string $T =$ `chacarachaca$` with the output being the last column in $\mathcal{M}_k$: $T^{kbwt} =$ `achhrcaa$acca`.

Due to the incomplete sorting of $\mathcal{M}_k$, it is not so straightforward to recover T from T^{kbwt} using LF_k, as the mapping only allows us to determine the context preceding the current symbol. For example, consider the following context jump in Figure 5.1, where our initial starting position is $T^{kbwt}[7] = a$ and $\text{LF}_k(7) = 3$. The symbol preceding "a" should be $T^{kbwt}[4] = h$, but due to the incomplete sorting of $\mathcal{M}_k$, the correct row – in the same k-group – is actually 5, which results in $T^{kbwt}[5] = r$. When recovering T from T^{kbwt}, LF_k is only guaranteed to jump to the correct preceding k-group. Individual context groups need to be processed in reverse sequential order. After performing LF_k, instead of using the row similar to the full BWT, we jump to the last *unprocessed* row within a given k-group. To consistently determine the correct context bounds, a bitvector D_k is required in order to reconstruct T from T^{kbwt} which is discussed in Chapter 4.

5.3 Backwards Search in Context-Bound Text Transformations

Next follows the main contribution of this chapter. We provide proof that performing *backward search* and thus calculating LF() is possible in T^{kbwt}. Note, in this section we choose a different example string than throughout the rest of this thesis to highlight certain aspects of the problem. We further provide a detailed walk-through example following the formal proof.

As explained, performing LF() and thus backwards search on a k-BWT permuted text is not so straightforward. The following lemma shows that one can, however, run the counting as usual on patterns of length $m \leq k$.

Lemma 1 *The* $\text{LF}_c(i, c)$ *mapping formula of Eq. (5.1) works correctly on the* k-BWT *if* i *is the last row of a context group.*

Proof. The formula counts the number of occurrences of c in $T^{bwt}[0, i]$. Since i is the last row of a k-context, the set of rows in $\mathcal{M}[0, i]$ is the same set of rows in $\mathcal{M}_k[0, i]$. So, the number of occurrences of any c in $T^{bwt}[0, i]$ is the same as in $T^{kbwt}[0, i]$. ■

Therefore, we can seamlessly search for patterns up to length k. This can be seen in Figure 5.1. The entries in the prefix of the suffix array $\text{SA}_k[0..i]$ at the end of a context group are permutations of the same prefix in the regular suffix array $\text{SA}[0..i]$.

Lemma 2 *The algorithm to compute the range* $\text{SA}[sp, ep]$ *used on the* BWT *works verbatim on the* k-BWT *for patterns of length* $m \leq k$.

Proof. As long as $m \leq k$, all the ranges $\langle sp_i, ep_i \rangle$ will be composed of complete k-contexts. Therefore the formulas in Eqs. (5.2) and (5.3) compute LF on rows that are at the end of k-contexts. By Lemma 1, all the $\langle sp_i, ep_i \rangle$ are thus correctly computed. ■

Note in particular that, if $m = k$, we will be able to collect the occurrences in text position order, since $\text{SA}[sp, ep]$ will correspond precisely to a k-context of T^{kbwt}. However, it is not possible to compute correct ranges $\text{SA}[sp, ep]$ for patterns longer than k because the contexts are only k-sorted, and thus the occurrences of longer patterns are not contiguous in the k-BWT.

A way to handle longer patterns $P[0, m-1]$ is to search for $P[m-k, m-1]$ as usual, and then track each candidate to determine whether it is an occurrence of the full P. But we must be able to compute $\text{LF}(j)$ for any j. Computing arbitrary $\text{LF}(j)$ values is also necessary for locating occurrences and for displaying arbitrary substrings of T, by using the sampling mechanisms described at the end of Section 5.1. The rest of the section is devoted to showing how we can compute LF.

Theorem 1 *The function* LF *can be computed on matrix* $\mathcal{M}_k$ *in the time required to compute* $rank$, $select$ *and* $access$ *using a wavelet tree on an alphabet of size* σ, *using* $nH_k + 2nH_{k-1} + o(n \log \sigma)$ *bits of space, for any* $k \leq \alpha \log_\sigma(n) - 1$ *and constant* $\alpha < 1$.

Proof. In order to compute $i = \text{LF}(j)$, the bitmap D_k is used to find $p = select(D_k, rank(D_k, j, 1), 1)$ which corresponds to the beginning of the k-group row $\mathcal{M}_k[j]$ belongs to. Let $\mathcal{M}_k[j] = xayb$, where $|x| = k - 1$ and $|a| = |b| = 1$, so row j belongs to group C^{xa}. Then $\mathcal{M}_k[i] = bxay$ belongs to the group C^{bx}. Moreover, since occurrences of xa are sorted in text position order inside C^{xa}, row j is the $(j - p + 1)$-th occurrence of xa in T in text position order.

In order to find out the starting position of group C_{bx}, the bitmap D_{k-1} is used, so that $p^* = select(D_{k-1}, rank(D_{k-1}, j, 1), 1)$ gives the starting position of group C^x in $T^{k\text{-}1\text{-BWT}}$. Then, $p' = Q[b] + rank(T^{kbwt}, p^* - 1, b) + 1$ gives the starting position of group C^{bx} in T^{kbwt}. The reason is that $rank(T^{kbwt}, p^* - 1, b)$ counts the number of text substrings of the form bz, with $z < x$ in lexicographic order. Now, within group C^{bx}, the rows are sorted in text position order, thus row i corresponds to the $(i - p' + 1)$-th occurrence of bx in T, in text order. Furthermore, row $\mathcal{M}_k[i]$ points to an occurrence of bxa in T, whereas $\mathcal{M}_k[j]$ points to the next position, xa preceded by b.

Thus a way to connect j and i is as follows. Store the wavelet tree of $T^{k\text{-}1\text{-BWT}}$, where all the characters preceding x are laid out in text position order in the area corresponding to group C^x. Similarly, store the wavelet tree of S, which is similar to $T^{k\text{-}1\text{-BWT}}$ but the characters following (not preceding) x are recorded for each context C^x (note the areas coincide for $T^{k\text{-}1\text{-BWT}}$ and S). Therefore, $r = select(S, rank(S, p^* - 1, 'a') + j - p + 1, 'a')$ finds the rank of row $\mathcal{M}_k[j]$ (that is, its occurrence of xa), in text position order, among the occurrences of x in T, and $q = rank(T^{k\text{-}1\text{-BWT}}, r, 'b') - rank(T^{k\text{-}1\text{-BWT}}, p^* - 1, 'b')$ is the number of occurrences of bx up to that position. Hence the answer is $i = \text{LF}(j) = p' + q - 1$.

Note this method requires determining the symbol preceding the suffix at position $T[\text{SA}[j]]$, which in our example is a. In a BWT permuted text, this symbol corresponds to $bwt[\text{SA}[j]]$. However, due to the incomplete sorting of the k-BWT, this cannot be guaranteed and therefore has to be stored explicitly. v This information can be stored in an array A of at most σ^k entries, storing $\mathcal{M}_k[j][k]$ for all the rows j belonging to each context. Therefore $a = A[rank(D_k, j, 1)]$.

The total time invested has been a constant number of $rank$, $select$ and $access$ operations on wavelet trees. Thus, asymptotically, performing LF on the k-BWT is as fast as performing LF over the BWT using a wavelet tree. The overall complexity depends on the wavelet tree used. For example, a balanced wavelet tree requires $\mathcal{O}(\log \sigma)$ time per operation.

As for space, we store the wavelet trees of T^{kbwt} and those of $T^{k\text{-}1\text{-BWT}}$ and S. If we use the compressed bitvector representation of Raman et al. [2002] to represent the bitmaps of the wavelet trees, T^{kbwt} requires $nH_k(T) + \mathcal{O}(\sigma^{k+1} \log n)$ bits for any $k \geq 0$. The reason is that the existing proof (see Mäkinen and Navarro [2007]) of this space bound for T^{bwt} only uses the fact that the suffixes are k-sorted, and thus it also applies to T^{kbwt}. Similarly, then, $T^{k\text{-}1\text{-BWT}}$ requires $nH_{k-1} + \mathcal{O}(\sigma^k \log n)$

$\mathcal{M}_3$

i	D_3	SA_k	**F 1 2**	$T^{3\text{-BWT}}$
0	1	12	$ a c a c a c r a c a c	a
1	1	11	a $ a c a c a c r a c a	c
2	1	0	a c a c a c r a c a c a	$
3	0	2	a c a c r a c a c a $ a	c
4	0	7	a c a c a $ a c a c a c	r
5	0	9	a c a $ a c a c a c r a	c
6	1	4	a c r a c a c a $ a c a	c
7	1	10	c a $ a c a c a c r a c	a
8	1	1	c a c a c r a c a c a $	a
9	0	3	c a c r a c a c a $ a c	a
10	0	8	c a c a $ a c a c a c r	a
11	1	5	c r a c a c a $ a c a c	a
12	1	6	r a c a c a $ a c a c a	c

$\mathcal{M}_2$

D_2	**F 1 S**	$T^{2\text{-BWT}}$
1	$ a c a c a c r a c a c	a
1	a $ a c a c a c r a c a	c
1	a c a c a c r a c a c a	$
0	a c a c r a c a c a $ a	c
0	a c r a c a c a $ a c a	c
0	a c a c a $ a c a c a c	r
0	a c a $ a c a c a c r a	c
1	c a c a c r a c a c a $	a
0	c a c r a c a c a $ a c	a
0	c a c a $ a c a c a c r	a
0	c a $ a c a c a c r a c	a
1	c r a c a c a $ a c a c	a
1	r a c a c a $ a c a c a	c

Figure 5.2: The k-BWT permutation matrix $\mathcal{M}_k$ used to search for pattern $P =$ `cacr` in text $T =$ `acacacracaca$` where $k = 2$ (right) and $k = 3$ (left).

bits. Finally, S requires also $nH_{k-1} + \mathcal{O}(\sigma^k \log n)$ bits because the sets of characters within each $(k-1)$-context is the same as for $(T^{rev})^{k\text{-}1\text{-BWT}}$, where T^{rev} is T read backwards, and thus the $(k-1)$-th order empirical entropy of S and $(T^{rev})^{k\text{-}1\text{-BWT}}$ are equal. Furthermore, the $(k-1)$-th order entropy between T and T^{rev} differs only by $\mathcal{O}(\log n)$ bits for any k [Ferragina and Manzini, 2005]. The bitmaps D_k and D_{k-1} have only $\mathcal{O}(\sigma^k)$ bits set out of n, and thus can be represented within $\mathcal{O}(\sigma^k \log n) + o(n)$ bits while supporting constant-time binary *rank* and *select* [Raman et al., 2002]. In practice, for small k, both vectors will be sparse and can therefore be compressed more efficiently using the representations of Okanohara and Sadakane [2007]. Array A requires $\mathcal{O}(\sigma^k \log \sigma)$ bits. To obtain the final space bound of the theorem we note that $\mathcal{O}(\sigma^{k+1} \log n) \subset o(n \log \sigma)$ if $k \leq \alpha \log_\sigma(n) - 1$ for any constant $\alpha < 1$. ■

5.3.1 Example LF Step in the k-BWT

Here we briefly walk through one LF step using the text $T =$ `acacacracaca$` for $k = 2$ and $k = 3$ shown in Figure 5.2. The rows in $\mathcal{M}_3$ prefixed by the pattern `caca` are 8 and 10. However, the row 9 within the range $\langle 8, 10 \rangle$ is not prefixed by `caca` but `cacr`. In the following example we perform $i = \text{LF}(j)$ for $j = 10$.

First we determine the k-group boundary of the context C^{cac} within $\mathcal{M}_3$ of our row $j = 10$

	i	D_3	SA_k	**F**	**1**	**2**									$T^{3\text{-BWT}}$
	0	1	12	$	a	c	a	c	a	c	r	a	c	a	c
	1	1	11	a	$	a	c	a	c	a	c	r	a	c	a
	2	1	0	a	c	a	c	a	c	r	a	c	a	c	a
	3	0	2	a	c	a	c	r	a	c	a	c	a	$	a
	4	0	7	a	c	a	c	a	$	a	c	a	c	a	c
	5	0	9	a	c	a	$	a	c	a	c	a	c	r	a
	6	1	4	a	c	r	a	c	a	c	a	$	a	c	a
	7	1	10	c	a	$	a	c	a	c	a	c	r	a	c
p →	8	1	1	c	a	c	a	c	r	a	c	a	c	a	$
	9	0	3	c	a	c	r	a	c	a	c	a	$	a	c
j →	10	0	8	c	a	c	a	$	a	c	a	c	a	c	r
	11	1	5	c	r	a	c	a	c	a	$	a	c	a	c
	12	1	6	r	a	c	a	c	a	$	a	c	a	c	a

Figure 5.3: Mapping the row $j = 10$ to the context group C^{cac} starting at position $p = 8$.

by performing $p = select(D_k, rank(D_k, j, 1), 1) = 8$. This is shown in Figure 5.3. Next we determine starting position (p^*) of the 2-order context C^{ca} of row j for $j = 10$. In order to find the starting position we use D_2, so that $p^* = select(D_2, rank(D_2, j = 10, 1), 1) = 7$ gives the starting position of group C^{ca} in $T^{3\text{-BWT}}$. This is shown in Figure 5.4. Using p^*, we calculate the starting position of the destination context group of the LF step. Here we calculate $p' = Q[`a`] + rank(T^{3\text{-BWT}}, p^* - 1, `a`) + 1 = 2$ which gives the starting position (p') of group C^{aca} in $T^{3\text{-BWT}}$. The reason is that $rank(T^{3\text{-BWT}}, p^* - 1.`a`)$ counts the number of text substrings of the form az, with $z < c$ in lexicographic order. That is the number of substrings in $T^{3\text{-BWT}}$ starting with symbol 'a' smaller than `aca`. This corresponds to the number times symbol 'a' occurs in $T^{3\text{-BWT}}[0 \dots p^* - 1]$. This is again shown in Figure 5.4.

Within the context group C^{aca}, all occurrences of `aca` are stored in text order. Thus, our destination row i within the context group corresponds to the $(i - p' + 1)$-th occurrences of `aca` in T. Further, we know that row i corresponds to an occurrence of `acac` in T whereas row j corresponds to an occurrence of `cac` in T preceded by 'a'. Next we show how row j can be connected to row i using these facts.

First, we map row j in $\mathcal{M}_3$ to the corresponding row r in $\mathcal{M}_2$. We compute r by determining the row in S, the symbols following each 2-order context in $\mathcal{M}_2$, which contains the $(j - p)$-th occurrence of symbol 'c'. This corresponds to the position of row j among the rows in the 2-order context C^{ca} in $\mathcal{M}_2$. We calculate r as $r = select(S, rank(S, p^* - 1,' c') + j - p + 1,' c') = select(S, rank(S, 6,' c') + 10 - 8 + 1,' c') = select(S, 4,' c') = 9$. This is shown in Figure 5.5 where we count the number times symbol 'c' occurs following the rows in context group C^{ca} shown as red.

	i	D_3	D_2	SA_k	**F**	**1**	**2**								$T^{3\text{-BWT}}$
	0	1	1	12	$	a	c	a	c	a	c	r	a	c	a	c	a
	1	1	1	11	a	$	a	c	a	c	a	c	r	a	c	a	c
$p' \rightarrow$	2	1	0	0	a	c	a	c	a	c	r	a	c	a	c	a	$
	3	0	0	2	a	c	a	c	r	a	c	a	c	a	$	a	c
	4	0	0	7	a	c	a	c	a	$	a	c	a	c	a	c	r
	5	0	0	9	a	c	a	$	a	c	a	c	a	c	r	a	c
	6	1	0	4	a	c	r	a	c	a	c	a	$	a	c	a	c
$p^* \rightarrow$	7	1	1	10	c	a	$	a	c	a	c	a	c	r	a	c	a
	8	1	0	1	c	a	c	a	c	r	a	c	a	c	a	$	a
	9	0	0	3	c	a	c	r	a	c	a	c	a	$	a	c	a
$j \rightarrow$	10	0	0	8	c	a	c	a	$	a	c	a	c	a	c	r	a
	11	1	1	5	c	r	a	c	a	c	a	$	a	c	a	c	a
	12	1	1	6	r	a	c	a	c	a	$	a	c	a	c	a	c

Figure 5.4: Mapping the row $j = 10$ to the context group C^{ca} starting at position $p^* = 8$ and the destination context group C^{aca} starting at postition $p' = 2$.

The occurrences (q) of symbol 'a' in context group C^{ca} in $\mathcal{M}_2$ which corresponds to the number of occurrences of 'a' in $T^{2\text{-BWT}}[p^* \dots r]$ can now be used to calculate the correct destination row i in $\mathcal{M}_3$. We calculate q as $q = rank(T^{2\text{-BWT}}, r,' a') - rank(T^{2\text{-BWT}}, p^* - 1,' a') = 3$, shown in blue in Figure 5.5. Thus $i = \text{LF}(j) = p' + q - 1 = 2 + 3 - 1 = 4$.

5.4 Practical Evaluation and Alternative Representations

In our initial approach, we require three wavelet trees over different permutations of T to fully support LF over a k-BWT permuted text. Table 5.6 shows the difference between LF_3 and the LF mapping to jump to the correct preceding row for the text $T =$ `acacacracaca$` and $k = 3$.

Instead of storing additional wavelet trees, we could also explicitly store the correction information δ for each row in $\mathcal{M}_k$ that does not jump to the correct preceding row using only LF_k. For each context C_i of size d, we can store the correction information required in at most $d \log d$ bits. The context length decreases as we increase the sorting depth k. Therefore, we need to store less correction information as k increases. The correction information can be calculated as follows in linear time:

1. Perform the reverse k-BWT to recover T from T^{kbwt}.
2. During the reverse transform, each context group C is processed in reverse sequential order.
3. For each context group, keep track of the current position p.

	D_2	**F**	**1**	**S**								$T^{2\text{-BWT}}$
	1	$	a	c	a	c	a	c	r	a	c	a
	1	a	$	a	c	a	c	a	c	r	a	c
	1	a	c	a	c	a	c	r	a	c	a	$
	0	a	c	a	c	r	a	c	a	c	$	a
	0	a	c	r	a	c	a	c	a	$	a	c
	0	a	c	a	c	a	$	a	c	a	c	r
	0	a	c	a	$	a	c	a	c	a	r	c
$p^* \rightarrow$	1	c	a	c	a	c	r	a	c	a	$	a
	0	c	a	c	r	a	c	a	c	a	$	a
$r \rightarrow$	0	c	a	c	a	$	a	c	a	c	r	a
$j \rightarrow$	0	c	a	$	a	c	a	c	a	c	r	a
	1	c	r	a	c	a	c	a	$	a	c	a
	1	r	a	c	a	c	a	$	a	c	a	c

Figure 5.5: Mapping the row $j = 10$ in $\mathcal{M}_3$ to it's corresponding row r in $\mathcal{M}_2$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12
LF_3	1	7	0	8	12	9	10	2	3	4	5	6	11
LF	1	7	0	8	12	10	9	5	2	3	4	6	11
δ						+1	-1	+3	-1	-1	-1		

Figure 5.6: The difference between LF *and* $\text{LF}_3()$.

4. The value δ_j is the correction information required to jump from row j to row v and can be calculated as $\delta_j = p_v - \text{LF}_k(j)$.

Note that we only store correction information for the n' non-trivial k-groups (that is, those of size more than 1). Those k-groups are stored according to their order within D_k at a cost of $n' \log n$ bits, in addition to the cost of storing the correction information for each k-group in $d \log d$ bits. To access the correct k-group for a given row j, we compute the number of non-trivial k-groups, o, preceding j, where $o = rank(D_k, j, \text{`10`})$ (that is the bitpattern `10`, which is an extension of binary rank for fixed substrings, which is easily handled in constant time within $o(n)$ extra space).

We can further apply the same concept to our wavelet tree approach: remove the information in $\mathcal{M}_{k-1}$ associated with trivial k-groups. The k-groups in $\mathcal{M}_{k-1}$ of size 1 will never be used to calculate LF as Eq. (5.1) is already correct on the last row of each group, hence it is correct on groups of size 1. To map a row j in $\mathcal{M}_k$ to its corresponding row j' in $\mathcal{M}_{k-1}$, we subtract the number of trivial groups before j', as these are not stored explicitly. A trivial k-group corresponds to two consecutive 1s in D_{k-1}. Therefore, the correct row $j'' = j' - rank(D_{k-1}, j', 11)$ (bitpattern `11`). By

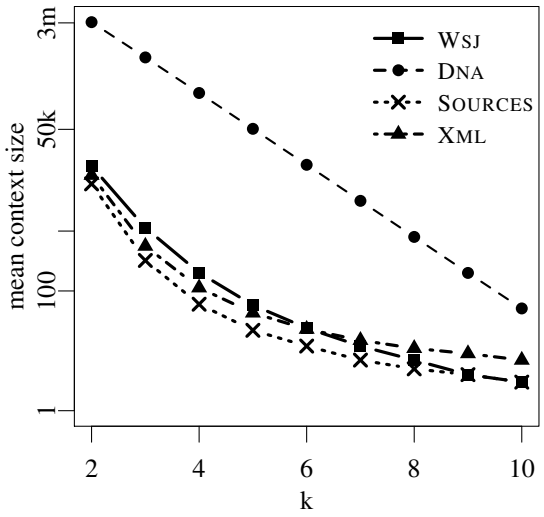

Figure 5.7: Mean average k-group size for each data set as k increases. Note the logarithmic scale of the y-axis.

combining all of these techniques, the additional information required to perform LF *decreases* as we increase the sorting depth k.

We now compare our backward search approach storing three wavelet trees, to storing the correction information δ_j explicitly. In our experiment, we compare multiple 200 MB data sets from the *Pizza&Chili* corpus and the TREC collection (see Section 2.7.3). The number of k-groups and their mean size for each data set is shown in Figure 5.7.

The mean context size decreases logarithmically for DNA. For the data sets WSJ, SOURCES and XML, and a sorting depth of 5, the mean k-group size is less than 100. This suggests that, for large k, storing the correction information is a viable alternative to storing three wavelet trees (although these would also decrease their sizes due to the increase of trivial groups). To validate this assumption, the total space requirements for the wavelet tree approach and the explicit storage of correction information for each data set was measured. We use Huffman-shaped wavelet trees [Mäkinen and Navarro, 2004] in conjunction with succinct $rank$ operations [Raman et al., 2002] to store and access the wavelet trees over $T^{2\text{-BWT}}$ and S. Figure 5.8 shows the size of the wavelet trees *as a percentage* of the space required to store the correction information explicitly using $d \log d$ bits for a k-group of size d.

As expected, the ratio increases with k because, while both approaches benefit from the trivial groups that appear, the correction information benefits from smaller groups due to its $\log d$ space factor, whereas the wavelet trees are blind to the group sizes. Note also that storing the correction

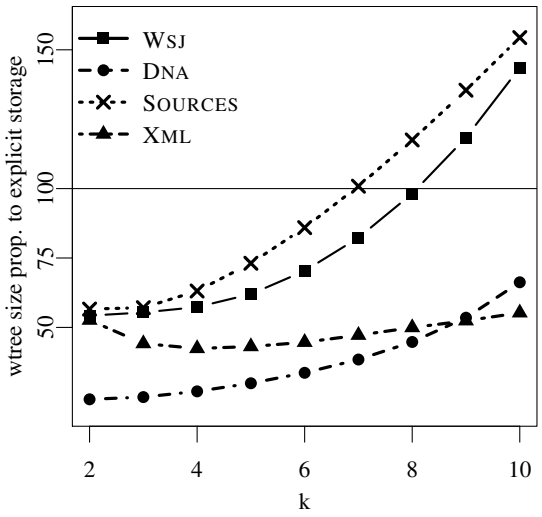

Figure 5.8: Storage requirements of the wavelet tree approach as a percentage of storing the correction information explicitly for variable k.

information explicitly is more efficient for English text (WSJ) and source code (SOURCES) for $k > 6$ or 7. However, DNA and XML can always be stored more efficiently using the wavelet trees approach. The reason for this discrepancy is two-fold. Firstly, DNA has a large number of contexts, even for $k = 10$, relative to the other files. Secondly, the XML file shows similar k-group sizes to WSJ and SOURCES, but it can be stored more efficiently using the wavelet tree approach as it is more compressible.

5.5 Applications

The k-BWT can be regarded as a self-index representation of a k-gram index. A k-gram index (often also called a q-gram index in literature) stores, for each unique k-gram in T, a postings lists containing all occurrences of the k-gram in T. Thus a k-gram index is a type of inverted index [Sutinen and Tarhio, 1996; Ullman, 1977]. A k-gram index is a popular alternative for constructing inverted indexes on languages that are not amenable to term tokenization and stemming, and a core component in the highly successful BLAST application for searching in genomic data [Altschul et al., 1990].

In essence, our method can represent the sequence T in compressed form, and in addition replace the need to explicitly store the position offsets for each k-gram. We have shown in Section 5.3 how to carry out searches for patterns of length k, in which case the index delivers the occurrence positions in text order. Thus, we obtain a listing that is explicitly stored in a classical k-gram index. Herein lies a distinct advantage of the k-BWT-based index over a similar BWT-based index. Consider the partial

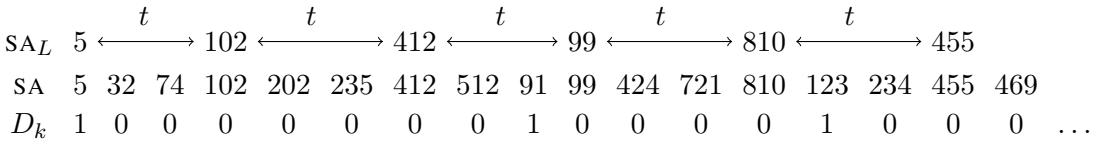

Figure 5.9: Suffix array sampling using t intervals can be used to allow efficient access to parts of a k-group corresponding to an area in T.

k-BWT based index shown in Figure 5.9. As discussed above, within a k-group, all suffixes are stored in increasing order. This would not be the case for the BWT. Instead of storing SA completely, we only store every t-th element at a total cost of $n/t \log n$ bits in SA_L. This was discussed in detail in Section 5.1. Interestingly, we can use the samples stored in SA_L to efficiently locate parts of a k-group corresponding to occurrences within a given text region. Here we perform binary search over the stored samples to determine the rows in $\mathcal{M}_k$ and thus the suffix positions in SA_k which are of interest. Therefore, instead of reconstructing all suffix positions, which is required for a BWT based index, we only reconstruct the relevant suffix positions using the sample positions in SA_L. This technique is especially relevant for *position-restricted searches*, where one is only interested in occurrences of P within a certain area of T. This task is generally difficult to perform with regular suffix arrays and BWT-based indexes as occurrences are delivered in lexicographical order of their suffixes. Additionally, we are able to display any text substring from the k-BWT-based self-indexed representation as we can perform $\text{LF}()$ as discussed in Section 5.3.

We now compare a simple k-gram inverted index to our k-BWT wavelet tree approach. We implemented a simple k-gram inverted index using d-gap and gamma encoded inverted lists for all match positions in the text. Figure 5.10 reports size of our index proportional to the size of the k-gram inverted index for increasing values of k. The size of the k-gram index is reported as the size of all compressed posting lists, plus the space required to store all necessary k-grams (v) in a dictionary, namely $v \log n + vk \log \sigma$. To provide a fair estimate of the lower cost bound to optimally store the gamma encoded, d-gapped posting lists, the measured size of the k-gram index was halved. We compare this conservative estimate against the actual size to store $T^{k\text{-}1\text{-BWT}}$, S, D_k and D_{k-1}. Here we use the practical space saving techniques discussed in Section 5.4 to reduce the size of S and $T^{k\text{-}1\text{-BWT}}$ by storing only information for non-trivial k-groups. Note we are not considering the cost of storing the compressed text in the k-gram index, nor the cost of storing T^{kbwt}, which should be similar.

Note that, for small k, the k-gram index is more space efficient than our k-BWT-based wavelet tree approach. As we increase the sorting length, the k-gram index space usage grows rapidly, whereas our approach consistently uses *less* space. For $k = 4$ or 5, we already require less space than the

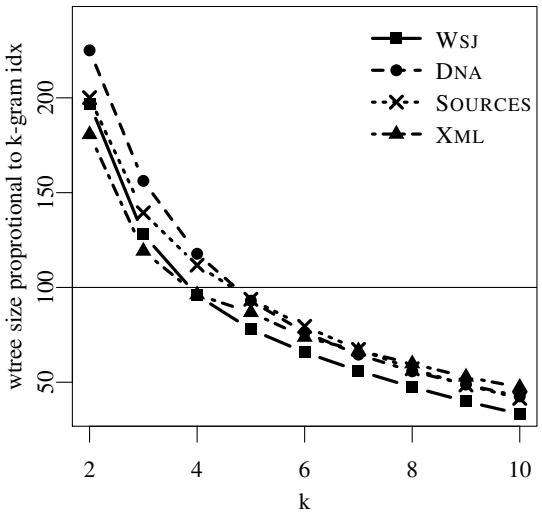

Figure 5.10: Storage requirements of the wavelet tree approach ($T^{k\text{-}1\text{-BWT}}$, S, D_k and D_{k-1}) as a percentage of the k-gram index space usage.

k-gram index. For $k = 10$, the wavelet trees require 30% to 50% of the k-gram index size (recall this is a lower bound as we are halving space required by the gamma encoding). The size of the k-gram index increases as the posting listing size decreases for larger k. Smaller posting lists can be compressed less effectively using d-gap and gamma encoding. The dictionary size also increases at dramatically as more unique k-grams are stored. The wavelet trees become more compressible as we increase the sorting length and the amount of extra information required to perform LF decreases as the number of trivial k-groups increases.

To conclude the practical evaluation we compare the absolute space usage of our approach to the k-gram index, a normal wavelet tree over T^{bwt} as used by a FM-Index, and a wavelet tree over T^{bwt}. Figure 5.11 shows absolute space usage in MB for the WSJ data set. The wavelet tree over T^{bwt} uses the least amount of space. For $k = 5$ the space usage of T^{kbwt} comes close to that of T^{bwt}. The k-gram index is most efficient for small k as the postings lists are large and can be compressed efficiently. However, the space usage of the k-gram index can grow exponentially in the worst case with respect to the sorting depth. The T^{kbwt} plus auxiliary information required to perform LF requires roughly three times the space of T^{bwt}. The amount of auxiliary information required decreases as the sorting depth increases.

Overall our experiments show an interesting phenomenon: the space requirements for our index decrease as k grows, whereas in a classical k-gram index, the space grows exponentially faster with k. This makes our representation attractive, for example, in applications where using larger k values

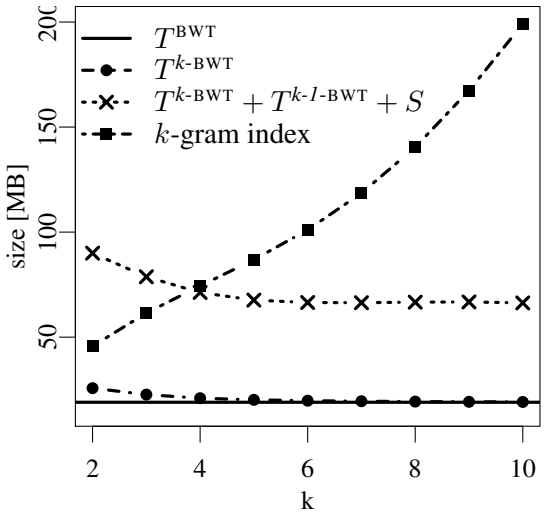

Figure 5.11: Absolute storage requirements in MB of the wavelet tree approach, the k-gram index, the wavelet tree over T^{kbwt}, and over T^{bwt} for the WSJ *data set.*

is desirable but not possible with a classical k-gram index.

Additional Applications

In Section 5.3 we described simple techniques to handle searches for patterns of lengths different from k, for applications where such a search is necessary. In general, our index can mimic any of the well-known algorithms on k-gram indexes. For example, we can split the pattern P into k size chunks P_1,P_2, $\ldots$ P_r. For each chunk P_i we determine $\langle sp_i, ep_i \rangle$ and the locations of each match to P_i. We then intersect these results to obtain the results to the larger pattern P. The ranges $\langle sp_i, ep_i \rangle$ for each chunk will be in ascending order. We can then perform an r-way merge using a min-heap over the smallest element in each occurrence range to get the final occurrence listing. Each range could further be processed simultaneously during the intersection process, returning increasing positions with each recovered $\text{SA}_k[i]$ element.

Another example occurs in BLAST-like applications, where approximate searches for P are reduced to a set of searches for k-grams of P. Then, in the most general formulation [Navarro et al., 2001a], one looks for text areas where at least h distinct k-grams of P appear in nearby text positions. Retrieving the k-gram occurrences in text order is essential for the effectiveness of these methods. Using this approach, we are able to leverage standard inverted indexing techniques to process queries, without explicitly storing the inverted lists for each k-gram occurrence. In fact, k-gram-based inverted files often do not explicitly store the position of each k-gram, but rather the

document occurrence. This saves space, but means false match filtering must be performed on each possible document occurrence [Puglisi et al., 2006]. Our method is able to return the exact positions of the candidates without the need to perform false match filtering in documents.

5.6 Summary and Conclusion

Succinct text indexes can be used to lower the space requirements of suffix-based search indexes. They can replace a suffix array requiring an $n \log n$ bits representation with a compressible text index using space equivalent to the compressed representation of the original text. However, constructing the index still requires that the full suffix array to be constructed. This is prohibitive in regards to indexing large text collections using suffix-based indexes. In this chapter we investigated a succinct text index which does not require the full suffix array. Instead, we used a representation which requires suffixes to be only sorted to depth k. Specifically we replaced the BWT with the k-BWT which we showed can be constructed more efficiently in external-memory than the BWT. In this chapter, we presented the first backwards search algorithm for k-BWT permuted text. We proved that function LF() can be performed over the k-BWT using auxiliary information. This allows for the original text to be extracted from an arbitrary position in T from T^{kbwt}. We improved our theoretical result by providing space saving techniques which reduce the space required to store the auxiliary information as the sorting depth k increases. We showed how a k-BWT-based self-index compares to a regular k-gram inverted index. Our analysis showed that as k increases, the self-index uses less space while the size of the k-gram index increases. We further discussed applications of a k-BWT based self-index to position-restricted searching and approximate pattern matching.

Interestingly, the k-BWT was independently discovered 15 years ago by Schindler [1997] and Yokoo [1999]. They described an alternative approach in which the n rotations are only *partially* sorted to a fixed prefix depth, k. At the time, using the BWT to perform text search has not yet been discovered [Ferragina and Manzini, 2000]. In this chapter we showed that the k-BWT can be used to construct succinct text indexes which allow efficient search for patterns up to length k. In the next chapter we discuss a different derivative of the k-BWT. Instead of sorting all suffixes to depth k, we sort suffixes based on the size of their context groups. Thus the resulting suffix array is sorted to *variable depths*. We refer to the resulting text permutation as a variable depth BWT or short v-BWT. Specifically we apply the v-BWT to the area of approximate pattern matching.

Chapter 6

Approximate Pattern Matching using Context-Bound Text Transformations

Approximate pattern matching is a classic problem in computer science with a wide variety of applications. A formal definition of the approximate pattern matching problem can be found in Definition 2 in Chapter 1. There are many practical applications and problems related to approximate pattern matching such as biological applications which have been well documented [Gusfield, 1997]. Efficient solutions to approximate pattern matching can also be applied in a variety of IR applications. Examples where approximate pattern matching can be applied in the IR domain include natural language keyword queries with spelling mistakes [Kukich, 1992], OCR scanned text incorporated into indexes [Kukich, 1992], language model ranking algorithms based on term proximity [Metzler and Croft, 2005] and DNA databases containing sequencing errors [Li and Durbin, 2009].

If the size of the text T is not large and only a few search queries will be performed, *on-line* algorithms such as `agrep` [Wu and Manber, 1992] can perform these operations in time proportional to the length of the text. However, on-line solutions are typically not sufficient for massive document collections, or situations which require a large number of queries to be run on the same collection. In these scenarios, building an index capable of supporting approximate matching queries is desirable.

Inverted indexes are often used in the context of approximate pattern matching. Instead of indexing text segmented into terms, T is split into overlapping k-grams (in literature often called q-grams). This index type is often referred to as a k-gram index (or q-gram index respectively) which generally perform well in practice despite providing no worst case performance guarantees. Therefore, a k-gram index is simply an inverted index storing positions of all distinct substrings of length k in T. The k-gram index is used as a filtering tool to generate potential positions in T matching

P. These positions must then be *verified* in the text using a variety of different *edit distance*-based algorithms [Navarro and Raffinot, 2002].

One of the weaknesses of traditional k-gram indexes is the use of fixed text segments of length k. If k is small, the inverted files can be very long for common k-grams, which degrades performance for many queries. However, if k is large, then the size of the index grows at an unacceptable rate. Recently, Navarro and Salmela [2009] show that using variable length k-grams can help find the best trade-off between the length of the postings lists, and the total number of "grams" that must be indexed. Unfortunately, the approach to finding the substrings to be indexed is computationally expensive as a suffix tree has to be created during construction time.

Another viable approach to indexing text collections allowing errors is to use a modified succinct-text index to support approximate text matching [Russo et al., 2009]. Most research in this domain has focused on providing worst case performance guarantees using a suffix array (or a compressed equivalent) to perform fast substring matches [Chan et al., 2010]. In this chapter we present an alternative, suffix array-based index. We present a variant of the BWT, called the variable depth Burrows-Wheeler transform (v-BWT), and apply it to the approximate pattern matching problem. Our contributions and the structure of this chapter can be summarized as follows:

1. We discuss the forward v-BWT transform in Section 6.2.
2. We show that the transform is always reversible in Section 6.3.
3. We demonstrate the use of the v-BWT to construct an index, precluding the need to explicitly represent the v-grams using postings lists in Section 6.4.
4. We show how the v-BWT can be used to create a variable length k-gram partitioning for variable length k-gram indexes without requiring a suffix tree in Section 6.4.2.
5. We empirically evaluate the usefulness of our transform by comparing the number of verifications required when performing approximate matching on both DNA and ENGLISH text in Section 6.5.

6.1 Text Transformations

In Chapters 4 and 5 we discussed and evaluated two text transformations, the Burrows-Wheeler Transform (BWT) and the context-bound BWT often referred to as the k-BWT. The regular BWT permutes T such that symbols with similar context are grouped together. Conceptually, the BWT creates a matrix $\mathcal{M}$ consisting of all rotations of T. The rows in the matrix are then sorted based on

the lexicographical ordering. T^{bwt} refers to the last column of $\mathcal{M}$. The BWT is reversible in $\mathcal{O}(n)$ time and can additionally be used to emulate search in a suffix array.

One of the main problems of constructing the BWT is that individual row comparisons in $\mathcal{M}$, or the equivalent suffix sorting comparisons in a suffix array construction algorithm can be computationally expensive. Two suffixes are compared by iterating over T starting from each respective position. In the worst case, each suffix comparison can take $\mathcal{O}(n)$ time. The k-BWT compares each row/suffix up to a depth of k symbols while stable sorting the equal rows based on initial text positions. This guarantees that each suffix comparison can be done in $\mathcal{O}(k)$ time. However, this implies that two suffixes are considered equal if they share the same k-prefix. Equal suffixes are grouped together into context groups. Inside a context group, suffixes are sorted based on their order in T, implying that the suffix array positions in each context group are in monotonically increasing order. This essentially creates a type of k-gram index which is often used for approximate pattern matching [Navarro, 2001] which was discussed in Chapter 5.

6.2 The Variable Depth Transform

The k-BWT sorts each suffix up to a fixed depth of k. Figure 6.1 shows the context size distribution for sorting depth $k = 2$ to 8. Note that as the sorting depth increases, the number of small contexts also increases. However, even at a depth of 8, many large contexts groups remain which correspond to substrings in T that have a length of 8. Instead of sorting all rows deeper, we sort only context groups above a threshold v.

The k-BWT specifies the sorting depth k until the rows in $\mathcal{M}_k$ are sorted. The incomplete sorting of $\mathcal{M}_k$ results in rows in $\mathcal{M}_k$ being grouped together in context groups. Each row in an individual context group shares the same k symbol prefix. Instead of defining the sorting depth k, we define the maximum context group size v in the matrix, now referred to as $\mathcal{M}_v$, allowed. We continue to sort context groups containing more than v rows until all context groups contain at most v rows. This implies that different parts of $\mathcal{M}_v$ can be sorted to different depths as not all k-grams in T occur equally often. Figure 6.2 shows an example of the v-BWT. Context groups '`p`' and '`$`' are sorted up to a depth 1. Context groups '`ay`' and '`yay`' are sorted to depth 2 and 3 respectively.

The bitvector D_v describing the context group boundaries of the v-BWT is now defined as shown in Definition 8. $D_v[i]$ is 0 if $D_{v-1}[i]$ is 0 and the size of the context group containing row i in the previous sorting stage (d^i_{v-1}) was smaller or equal to v or if the v-prefix of row i is equal to row $i-1$. $D_v[i]$ is 1 otherwise.

Definition 8 *For any $1 \leq v < n$, let d^i_{v-1} be the size of the context group containing row i after*

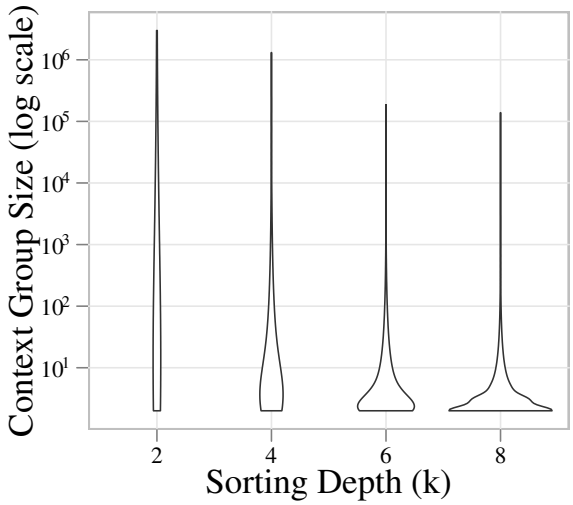

Figure 6.1: Context group size (logarithmic) distribution for different sorting depth k of the k-BWT for a 200 MB English text file.

sorting step $v-1$. Let $D_v[0, n-1]$ be a bitvector, such that $D_v[0] = 1$ and, for $1 \leq i < n$,

$$D_v[i] = \begin{cases} 0 & \textit{if } D_{v-1}[i] = 0 \textit{ and } d^i_{v-1} \leq v, \\ 0 & \textit{if } D_{v-1}[i] = 0 \textit{ and } d^i_{v-1} > v \textit{ and;} \\ & \mathcal{M}_v[i][0, v-1] = \mathcal{M}_v[i-1][0, v-1], \\ 1 & \textit{otherwise.} \end{cases}$$

The forward transformation of the v-BWT is outlined in Algorithm 1. We recursively perform radixsort for each of the context groups until the context group size is less then our defined threshold v. The algorithm returns the context group vector D_v, as well as the suffix array (SA) sorted up to variable sorting depth. The duality between the BWT and suffix array is used to create $T^{v\text{-BWT}}$ as $T^{v\text{-BWT}}[i] = T[\text{SA}[i] - 1]$.

In order to bound worst-case sorting time, additional parameters are necessary. Let k_{min} be the minimum sorting depth for all context groups and let k_{max} be the maximum sorting depth. This guarantees a worst case runtime complexity of the forward transform using a standard radixsort algorithm of $\mathcal{O}(k_{max}n)$.

i	D_v	LF	**F**										L
0	1	11	**$**	y	a	y	a	y	a	p	y	a	y	a
1	1	10	**a**	**$**	y	a	y	a	y	a	p	y	a	y
2	1	5	**a**	**p**	y	a	y	a	$	y	a	y	a	y
3	1	1	**a**	**y**	a	y	a	p	y	a	y	a	$	y
4	0	3	**a**	**y**	a	p	y	a	y	a	$	y	a	y
5	0	8	**a**	**y**	a	$	y	a	y	a	y	a	p	y
6	1	6	**p**	y	a	y	a	$	y	a	y	a	y	a
7	1	9	**y**	**a**	**$**	y	a	y	a	y	a	p	y	a
8	1	4	**y**	**a**	**p**	y	a	y	a	$	y	a	y	a
9	1	0	**y**	**a**	**y**	a	y	a	p	y	a	y	a	$
10	0	2	**y**	**a**	**y**	a	p	y	a	y	a	$	y	a
11	0	7	**y**	**a**	**y**	a	$	y	a	y	a	y	a	p

Figure 6.2: v-BWT *for* $T =$`yayayapyaya$` *including* LF *mapping and the context-group vector for threshold* $v = 3$*. The different sorting depths are boldfaced.*

6.3 Reversing the Variable Depth Transform

The k-BWT can be reversed using the bit vector D_k marking the beginning of the context boundaries, as contexts are required to be processed in reverse sequential order. The LF() mapping is only guaranteed to jump into the correct context group as discussed in Chapter 5. Similarly, D_v can be used to reverse the v-BWT even though not all columns are sorted to the same depth k.

Lemma 3 *The text* T *can be recovered from the permutation* $T^{v\text{-BWT}}$ *using the context group boundaries* D_v *and the* LF() *mapping.*

Proof. Recall, Equation 5.1 (LF) counts the number of occurrences of $c = T^{bwt}[i]$ in $T^{bwt}[0, i]$. If i represents the last row of a context group, the set of rows in $\mathcal{M}[0, i]$ is identical to the rows in $\mathcal{M}_v[0, i]$ as the context groups are lexicographically sorted. Therefore the number of occurrences of any c in $T^{v\text{-BWT}}[0, i]$ is the same as in $T^{bwt}[0, i]$. The different sorting depths k' and k'' do not affect the lexicographical order of different contexts, as the sorting depth can only affect the order within a context group. So, $j = \text{LF}(i)$ maps correctly between two context groups $d^i_{k'}$ and $d^j_{k''}$ despite being sorted to different depths k' and k''. As $\text{LF}(i)$ must map to the correct preceding context group as in the k-BWT, using D_v each context group can be processed in reverse sequential order to recover T. ■

Algorithm 1 v-BWT Forward Transform of text T with threshold v.

```
1: VBWT (T[0...n-1], v)
2: Initialize SA[0...n-1]
3: Count symbols to create D_1
4: for each context group D_1[i...j] do
5:     RADIXSORT (T, SA, D_1[i...j], v, 2)
6: end for
7: for i ← 0 to n-1 do
8:     if SA[i] = 0 then
9:         T^{v-BWT}[i] ← T[n-1]
10:    else
11:        T^{v-BWT}[i] ← T[SA[i]-1]
12:    end if
13: end for
14: return T^{v-BWT}
```

FUNCTION **RADIXSORT** $(T, \text{SA}, D[i \dots j], v, k)$

```
1: if j - i + 1 ≤ v or k ≥ k_max then
2:     return
3: end if
4: COUNTSORT symbols in SA[i...j]
5: for all symbols ∈ SA[i...j] do
6:     Mark start of new context group in D
7:     RADIXSORT (T, SA, D[i...j], v, k+1)
8: end for
```

To recover T from $T^{v\text{-BWT}}$ no additional information is required as the context boundaries D_v can be recovered from $T^{v\text{-BWT}}$ in $\mathcal{O}(k_{max}n)$ time, where k_{max} is the maximum sorting depth of any context group in $T^{v\text{-BWT}}$.

Lemma 4 *D_v can be recovered directly from $T^{v\text{-BWT}}$ using no additional information.*

Proof. Recall that context information is not needed to restore the first k columns of $\mathcal{M}_k$. Instead of recovering $\mathcal{M}_v$ to a depth of k, recovery is based on the number of rows, v, with an identical prefix $\mathcal{M}_v[1...j]$. Let t be the maximum number of rows in $\mathcal{M}_v$ that have the same prefix $\mathcal{M}_v[1...j]$ when sorted to a depth of j. If the number of rows t with the same prefix exceeds v at the current sorting depth j, this context group must be sorted up to depth $j+1$. The recovery continues in the current context group until $t \leq v$. The final context group recovered is D_v. ■

We now give an example of how to recover D_v from $T^{v\text{-BWT}}$. First, we recover F by sorting $L = T^{v\text{-BWT}}$ and initialize D_1 to the symbol boundaries. We also keep track of the $F \rightarrow L$ column

mapping:

D_1	1	1	0	0	0	0	1	1	0	0	0	0
F	\$	a	a	a	a	a	p	y	y	y	y	y
L	a	y	y	y	y	y	a	a	a	\$	a	p
FL_1	9	0	6	7	8	10	11	1	2	3	4	5

Next, for all context groups larger or equal $s = 3$, we recover the next column in $\mathcal{M}$ using the initial FL_1 mapping. We update D_2 to include the new context boundaries and use the initial FL_1 mapping to create $FL_2[i] = FL_1[FL_1[i]]$ for context groups larger than v.

D_2	1	1	1	1	0	0	1	1	0	0	0	0
F	\$	a	a	a	a	a	p	y	y	y	y	y
		\$	p	y	y	y		a	a	a	a	a
L	a	y	y	y	y	y	a	a	a	\$	a	p
FL_2								0	6	7	8	10

Using FL_2 we recover the next column for context groups larger than v in a similar manner:

D_3	1	1	1	1	0	0	1	1	1	1	0	0
F	\$	a	a	a	a	a	p	y	y	y	y	y
		\$	p	y	y	y		a	a	a	a	a
								\$	p	y	y	y
L	a	y	y	y	y	y	a	a	a	\$	a	p

We now have D_v as the size of all of the context groups less than or equal to v. Therefore we can recover T from $T^{v\text{-BWT}}$ without the need to store any additional information.

6.4 Variable Length k-Gram Index

A k-gram (often called a q-gram in literature) is a contiguous sequence of symbols in a text T: $T[i..\ell]$ where $\ell - i + 1 = k$. A k-gram index uses all k-grams in T to support approximate pattern matching over the text [Navarro and Baeza-Yates, 1998]. Traditional k-gram indexes are based on inverted files. For each distinct k-gram k_i in T, a list of positions of all occurrences of k_i are stored. These lists can be d-gapped and compressed to reduce space. Individual inverted files are accessed through the vocabulary, which can be represented using a data structure such as a trie [Navarro and Salmela, 2009]. In large text collections, k-gram indexes have limitations. First, the number of distinct k-grams in T can grow exponentially with the size of k in the worst case. Second, certain k-grams tend to occur much more frequently than others.

Navarro and Salmela [2009] propose a variable length k-gram index, where each variable length k-gram is required to have a uniform number of occurrences, and no k-gram occurs more than v times. The index is prefix-free, so no selected k-gram can be a prefix of any other k-gram in the index. To create the index, Navarro and Salmela first construct a suffix tree over T in $\mathcal{O}(n)$ time. Next the suffix tree is traversed in depth-first order in $\mathcal{O}(n)$ time to retrieve the vocabulary of the index by pruning the suffix tree at nodes whose subtree contains at most v leaf nodes corresponding to suffix positions in T. Next, the position lists are sorted in increasing order in $\mathcal{O}(n \log \sigma)$ time and compressed in $\mathcal{O}(n)$ time. The total cost of constructing the index is therefore $\mathcal{O}(n \log \sigma + n \log v)$.

The v-BWT can significantly simplify the construction of a variable k-gram index. First, we create $T^{v\text{-BWT}}$ of T with threshold v. In the process the following components of the k-gram index can be created. The suffix tree partitioning of Navarro and Salmela [2009] can also be accomplished using D_v since each context group contains at most v rows. The postings lists can be obtained implicitly via SA_v, the suffix array used to sort T. Within each context group, the suffix array positions correspond to the entries in the postings list in the k-gram index. These lists are already sorted and do not require the $\mathcal{O}(n \log \sigma)$ sort described by Navarro and Salmela. In fact, we perform this step implicitly *while* creating the partitioning. Additionally, the list can be recovered or searched from $T^{v\text{-BWT}}$ and a sampled version of the corresponding suffix array.

6.4.1 Representing the Vocabulary

Traditional k-gram indexes consist of two main components: the vocabulary, stored as a trie or other compressed dictionary representation [Brisaboa et al., 2011]; and, a compressed inverted file for each distinct indexed k-gram containing all occurrences of the k-gram in T. To perform an approximate pattern search, a pattern is split up into $k + 1$ substrings. Next, for each substring, the inverted list is loaded by querying the vocabulary. Previously we showed how to obtain a variable k-gram partitioning using the v-BWT. Here we show how we can replace the vocabulary of a variable k-gram index with a wavelet tree over $T^{v\text{-BWT}}$.

The v-BWT can be used to obtain a variable length k-gram partitioning equivalent to the index proposed by Navarro and Salmela [2009]. Instead of using a trie to store the vocabulary, we can instead perform a backwards search using a compressed wavelet tree over $T^{v\text{-BWT}}$.

Lemma 5 *Backwards search for any substring p_i can be performed in $T^{v\text{-BWT}}$ as long as the number of matching rows, $[sp, ep]$ in $\mathcal{M}_v$ is $\geq v$.*

Proof. Performing backwards search for a pattern up to length k works correctly in T^{kbwt} as each context is guaranteed to be sorted up to depth k. Therefore, performing $k - 1$ backwards probes is

guaranteed to return the correct range of rows, sp, ep, in $\mathcal{M}_k$ for any p_i of length k. Similarly, every context group d_k^i corresponding to a prefix $\mathcal{M}_v[0..j]$ is sorted if there are more than v rows in $\mathcal{M}_v$ prefixed by $\mathcal{M}_v[0..j]$ in $T^{v\text{-BWT}}$. Therefore, backwards search is guaranteed to result in the correct sp, ep in $\mathcal{M}_v$ if $ep - sp + 1 \geq v$. ■

Additionally, under certain conditions patterns occurring less than v times are also represented continuously in $\mathcal{M}_v$.

Lemma 6 *Backwards search for any substring $p_i = P[i \dots j]$ can be performed in $T^{v\text{-BWT}}$ as long as the number of matching rows of the suffix of p_i, $P[i \dots j-1]$ is larger than v.*

Proof. Consider the construction algorithm discussed in Algorithm 1. During construction, suppose context group C_i is currently sorted to k' and contains more than v rows in $\mathcal{M}_v$. Thus the context group is split into up to σ smaller context group sorted to depth of at least $k'+1$. Let C_j be a context group resulting from the additional sorting step of C_i. If the number of rows in C_j is less or equal v, the sorting procedure stops. However, within C_j all rows are still prefixed by the same $k'+1$ long prefix. Thus, patterns $P[i \dots j]$ are represented continuously in $\mathcal{M}_v$ if the suffix $P[i \dots j-1]$ occurs more than v times. ■

A wavelet tree over $T^{v\text{-BWT}}$ can be used to determine ranges in $\mathcal{M}_v$ which correspond to substrings p_i of P. The size of the range corresponds to the number of occurrences of p_i in T. For patterns with less than v occurrences, the range in $\mathcal{M}_v$ is not guaranteed to be continuous if the pattern length is longer than the sorting depth of the corresponding context group. For patterns with more than v occurrences, the resulting range $\langle sp, ep \rangle$ has to cover multiple context groups. Each context group is prefixed by p_i. However, the text positions are only sorted in text positions within each context group and not within the complete range sp, ep.

6.4.2 Optimal Pattern Partitioning

To search for a pattern P with at most Y errors, a k-gram index performs a *filtering* step whereby a string P is split into $Y+1$ substrings $p_1 \dots p_{Y+1}$. For P to occur in a string T with at most Y errors, at least one substring p_i must appear in T [Navarro et al., 2001b]. A k-gram index is used to find all *candidate* positions of P in T by partitioning P into $Y+1$ substrings $p_1 \dots p_{Y+1}$ and retrieving the positions in T for all p_i. Navarro and Baeza-Yates [1998] provide a dynamic programming algorithm which calculates the optimal partitioning of P into $Y+1$ pieces to minimize the number of candidates.

In the second step, a standard *edit distance* algorithm is then used to verify all candidates [Navarro and Raffinot, 2002].

We now show how to use the optimal pattern partitioning algorithm proposed by Navarro and Baeza-Yates [1998] and later used by Navarro and Salmela [2009] to enable approximate searching using a wavelet tree over $T^{v\text{-BWT}}$. The key intuition of Navarro and Baeza-Yates's algorithm is to compute all m^2 possible substrings $P[i-j]$ and the resulting candidate list lengths in a matrix $R[i,j]$ of size $\mathcal{O}(m^2)$. Dynamic programming is then used to retrieve the optimal partitioning by processing R in $\mathcal{O}(m^2 Y)$ time [Navarro and Baeza-Yates, 1998; Navarro and Salmela, 2009]. Using the backwards search (BWS) procedure, we compute $R[i,j]$:

$$R[i,j] = \begin{cases} |\langle sp, ep\rangle| & \text{if } \texttt{BWS}(P[i \ldots j]) = |\langle sp, ep\rangle| > v, \\ |\langle sp, ep\rangle| & \text{if } \texttt{BWS}(P[i \ldots j]) = |\langle sp, ep\rangle| \leq v \text{ and} \\ & \quad \texttt{BWS}(P[i \ldots j-1]) = |\langle sp, ep\rangle| > v, \\ \infty & \text{otherwise.} \end{cases}$$

Where $\langle sp, ep\rangle$ is the range in the suffix array prefixed by P. This range is only guaranteed to be continuous under certain conditions discussed in Lemmas 5 and 6. All substrings for which we cannot determine $\langle sp, ep\rangle$ are set to infinity in our calculations, thus making sure they are not included in the final partitioning of P into $p_{ij}, ..p_{j+1,l}, ...p_{m-1}$. For each substring we retrieve the corresponding $\langle sp, ep\rangle$ ranges in order to determine the parts of the suffix array containing the candidate positions. Using a wavelet tree to perform backward search increases the overall cost to compute R to $\mathcal{O}(m^2 \log \sigma)$ as the number of occurrences $(ep - sp + 1)$ of all combinations of patterns $P[i..j]$ are determined using backwards search and stored in the cells $R[i,j]$.

6.4.3 Storing Text Positions

Traditionally, the vocabulary contains pointers (file offsets) at which the individual postings list for the indexed strings (k-grams) are stored. As we are using a wavelet tree to store the vocabulary, we choose a different representation to store postings lists. Recall that within a context group in $T^{v\text{-BWT}}$, all corresponding suffix array positions are in ascending text order. We can therefore store a compressed version SA'_v of SA which d-gaps and compresses all offsets in a single context group in the same manner as is often used in postings lists for inverted indexes.

Unfortunately, the ranges $\langle sp, ep\rangle$ in SA_v cannot be used to find the corresponding position in SA'_v. So, we store an additional bitvector D'_v that maps context groups in SA to the corresponding starting positions in SA'_v. First we calculate the distance of sp to the corresponding context group

start in SA_v using $\ell = rank(D_v, sp, 1)$ and $t = select(D_v, \ell, 1)$. Next we map the context group into the compressed representation SA'_v using $sp' = select(D'_v, t, 1)$. Starting from sp' we skip the first $sp - \ell$ encoded numbers and then retrieve the next $ep - sp + 1$ encoded positions of $\langle sp, ep \rangle$. Note that $\langle sp, ep \rangle$ might span multiple smaller context groups which must each contain separately compressed d-gap lists.

The vocabulary and all auxiliary information needed to perform optimal partitioning can be stored using $H_{k_{min}}(T)$ space – the cost of storing a wavelet tree over $T^{v\text{-BWT}}$ with a minimal sorting depth of k_{min}. The text positions in SA'_v use variable byte coding which uses up to 30% more space than bit-compressed inverted lists, but allows for faster decoding time [Scholer et al., 2002]. Storing these runs is similar to storing the runs in the ψ function of the compressed suffix array of Sadakane [2002], which uses δ-codes [Elias, 1975] to store the d-gapped positions. We further store H_0 compressed representations of D_v and D'_v [Raman et al., 2002].

6.5 Empirical Evaluation

In this section we evaluate the v-BWT transform itself, and a k-gram index using the transform. First we investigate the running time of the forward transform. Next we investigate the performance of an index based on the v-BWT when applied to approximate pattern matching.

6.5.1 Experimental Setup

In the experiments we again use the experimental setup described in Section 2.7. The forward transform is implemented using the cache efficient radixsort string sorting approach proposed by Kärkkäinen and Rantala [2008]. We modified the algorithm to account for the parameters k_{min}, k_{max} and v as discussed above. As test data, we use a 1 GB prefix of both the DNA and WEB data set described in detail in Section 2.7. We use the compressed suffix tree implementation provided in the SDSL library to compare the constriction time of our variable depth index to the approach of Navarro and Salmela. We further use the suffix sorting algorithm implemented in `libdivsufsort` to construct the full BWT efficiently as a baseline.

6.5.2 Transform Performance

We first evaluate the runtime efficiency of our new transform and compare the forward transform with the k-BWT and the full BWT. Table 6.1 shows the runtime performance to create $T^{v\text{-BWT}}$, T^{kbwt} and T^{bwt} respectively for both test files.

	Time [sec]							
	k-BWT			v-BWT				BWT
	3	5	9	5	50	500	5000	
DNA	63	121	253	296	244	191	138	283
WEB	89	145	258	312	262	235	209	213

Table 6.1: Construction time (in seconds) of v-BWT*,* k-BWT*, and the full* BWT*.*

	Time [sec]	
Step	suffix tree	v-BWT
construct CST	736	-
suffix tree traversal	405	-
sort suffix array	453	-
create v-BWT	-	224
build vocabulary	24	
vbyte compress postings lists	30	
Total	1648	278

Table 6.2: Construction cost comparison of the method by Navarro and Salmela and the v-BWT *for* $v = 50$ *on the* DNA *data set.*

The bounded transforms perform better for DNA than for WEB compared to the full BWT. For DNA, constructing the k-BWT is faster than constructing the full BWT. The v-BWT can be constructed more efficiently for sorting depths up to 5. Note that for $v = 5$, the v-BWT is almost identical to the full BWT, and only contexts up to size 5 remain. For $v = 50$ to 5000 the v-BWT can be constructed even more efficiently. The WEB data set can be constructed 40% faster with the full BWT compared to DNA. Induced suffix sorting reduces the number of suffix comparisons required to construct the suffix array. Therefore, the number of suffix comparisons needed is the limiting factor. Longer text comparisons have to be performed to determine the order of two suffix positions. The bounded transforms also perform slower for WEB. For $v = 5$, the variable transform is 40% slower than DIVSUFSORT. As the sorting depth decreases, the v-BWT again outperforms the full BWT.

6.5.3 Variable k-Gram Index Construction

The construction time of our variable k-gram-based index can be compared to the suffix tree method of Navarro and Salmela [2009]. As described by Navarro and Salmela, we first construct a compressed suffix tree using `libsdsl` [Gog, 2011]. Next we perform a depth first search traversal to determine the highest nodes in the suffix tree that have at most $v = 50$ children. The ranges in the suf-

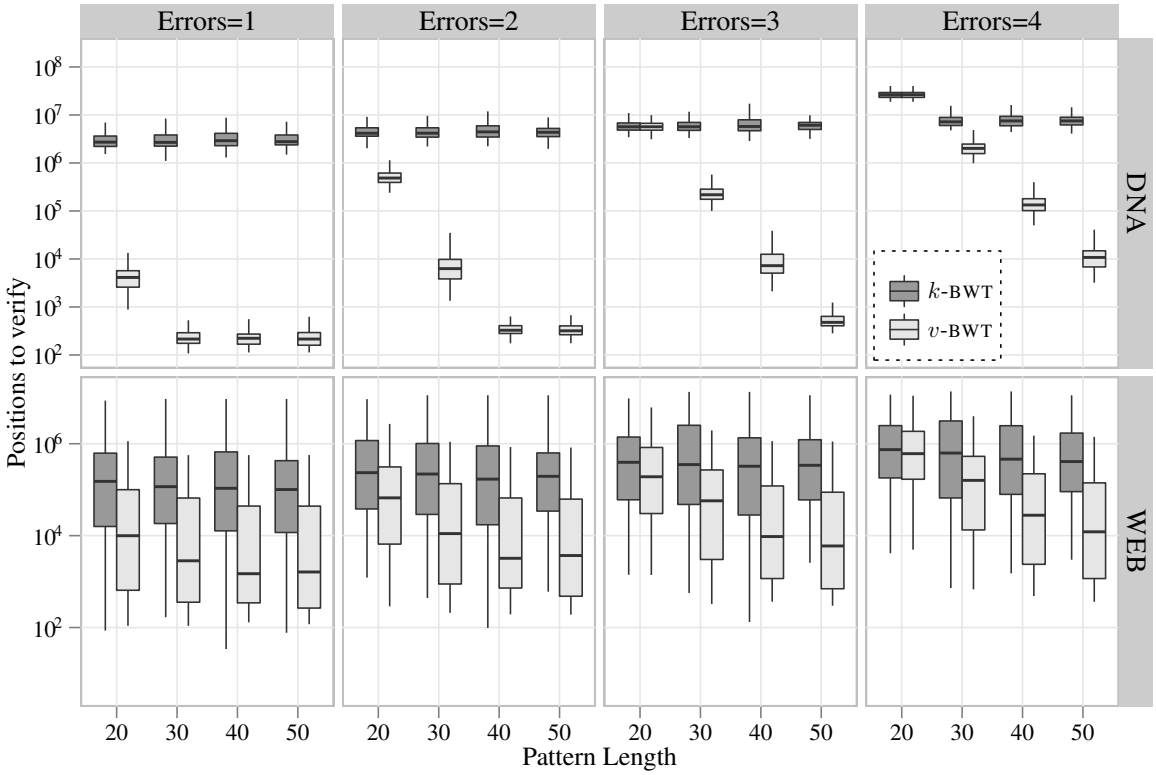

Figure 6.3: Number of verifications required for k-BWT with $k = 5$ and v-BWT with $v = 50$ for the DNA *and* WEB *data sets.*

fix array corresponding to the marked nodes are recorded. Lastly, the individual ranges in the suffix array are sorted. We compare this approach to constructing an equivalent index using our v-BWT for $v = 50$. The different steps required in addition to the time required to build an index for threshold $v = 50$ for DNA are shown in Table 6.2. Note that the table further lists the cost to construct the vocabulary and compress the individual postings lists.

As expected, the construction of the suffix tree is the main bottleneck in the method of Navarro and Salmela. In fact, traversing the suffix tree to determine the different ranges in the suffix array for the approach is more expensive than creating the entire index using the v-BWT transform. Sorting each range in the suffix array in the suffix tree method is also computationally expensive, and unnecessary when using v-BWT. Overall, the v-BWT index can be constructed 5 times faster than the best known k-gram approach.

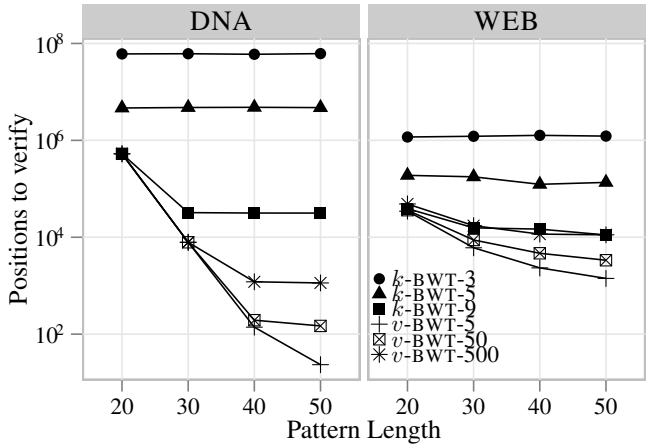

Figure 6.4: Number of verifications required for k-BWT *for variable* $k = 3, 5, 9$ *and* $v = 5, 50, 500$ *for* 2 *errors for* DNA *and* WEB *data sets.*

6.5.4 Variable k-Gram Verifications

All k-gram-based approximate pattern matching approaches use filtering to reduce verification costs. Potential matching candidates must still be verified using an *edit distance* algorithm. The goal of the filter is to minimize the number of verifications required to perform approximate search. We now evaluate the number of verifications required by each indexing approach. First, we perform 1000 approximate pattern searches for pattern lengths 20 to 50 using different error levels. The patterns were randomly sampled from each data set. Figure 6.3 shows the number of candidate positions which must be verified after pattern partitioning is performed. For this experiment, we only compare $k = 5$ and $v = 50$ using a wavelet tree as the vocabulary for both approaches. For DNA, the number of positions requiring verification tend to be higher than for WEB as the data is more uniform, and the alphabet size is smaller. The v-BWT always outperforms classical k-BWT partitioning. The variance in the WEB data set is higher than for DNA, while DNA generally requires more verifications using the k-BWT based approach. The v-BWT approach outperforms the k-BWT approach for the DNA data set by several orders of magnitude except for patterns of length 20 with error rates of 3 and 4. This implies that P has to be split into 4 and 5 substrings respectively. As the sorting depth for the k-BWT is 5, we conjecture that the substrings being evaluated with the v-BWT are rarely longer than in the k-BWT.

Next, we show how the number of verifications varies with different sorting parameters. We choose only small k values as the number of potential dictionary entries can, in the worst case, grow exponentially as k increases. Similarly, we choose the parameter v to have similar construction costs as our chosen k values. Figure 6.4 shows the *mean* number of verifications required for 1000

approximate pattern searches for patterns of length 20 to 50 for variable transform parameters. The number of verifications required using the standard fixed k-gram k-BWT approach decreases as k increases due to the fact that longer substrings can be matched. For $k = 9$, performance is similar to that of the v-BWT for patterns of length 20. Generally, for all k the k-BWT approach requires more verifications. The average number of verifications required stays roughly constant for the fixed k-gram approach whereas the mean number of verifications decreases using the variable length k-gram approach as the length of the pattern increases. As the pattern length increases, our approach can match longer variable length k-grams during the optimal partitioning phase. Longer k-grams occur less frequently. Therefore, the number of verifications required decreases.

Finally we evaluate the space usage of the variable depth index. We specifically compare the size of the postings lists for different sorting depths for both the k-BWT and the v-BWT. Intuitively, sorting to higher depths will result in less verifications at the cost of larger space usage due to the smaller average context size. Smaller contexts contain shorter runs of increasing numbers which can be compressed less efficiently. Figure 6.5 shows the trade-off between verifications required compared to the space used to store the position information in the index. We compare patterns of lengths 30, 40 and 50 while computing the verifications required for $Y = 1$. Overall the k-BWT-based index requires more verifications while using less space. The variable depth-based transforms for $v = 5, 50, 500$ use more space while requiring less verifications. For the WEB data set, the space usage of the v-BWT are close to that of the k-BWT. As for the different sorting depths, large context groups still exist. The difference in space usage is more visible for the DNA data set. Overall the v-BWT-based index provides a new time and space trade-off compared to traditional k-gram indexes while being able to be constructed efficiently.

6.6 Summary and Conclusion

In Chapters 4 and 5 we investigated the k-BWT and succinct-text indexes based on the transform. We showed that context-bound text transformations and succinct text indexes based on these transformations are viable alternatives to BWT-based indexes. In this chapter we focused on applications of context-bound text transformations. We applied a new context-bound based sort transformation – the v-BWT– to the problem of approximate pattern matching. We defined a parameter v which specifies the maximum size of each context group C^i. If for a given sorting depth C^i is larger than v, it is split up into smaller context groups until each context group contains at most v suffixes. Thus context groups are sorted to different depths. We further defined minimum (k_{min}) and maximum (k_{max}) sorting depth to bound the running time of the transform. Using the parameters v, k_{min} and

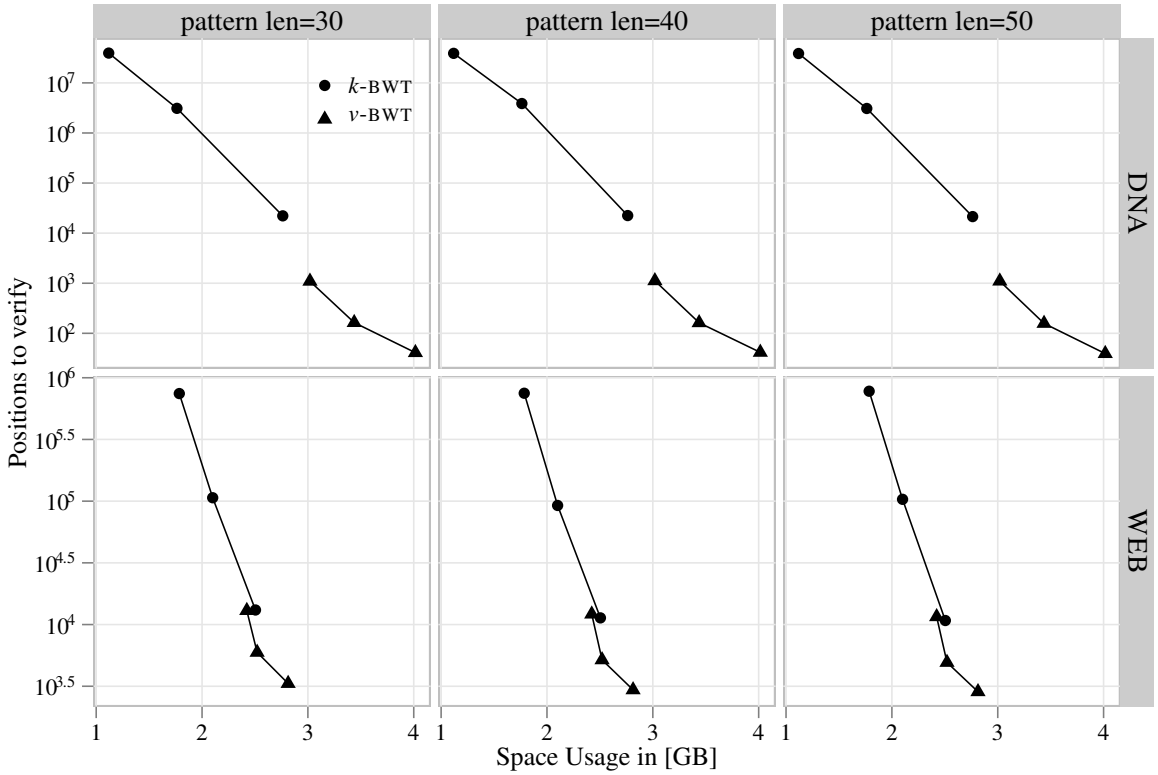

Figure 6.5: Number of verifications required for k-BWT *for variable* $k = 3, 5, 9$ *and* $v = 5, 50, 500$ *for* $Y = 1$ *error for* DNA *and* WEB *data sets compared to the space required to store the postings lists.*

k_{max} we showed the transform can be reversed similar to the k-BWT by recovering the context group boundaries.

In Chapter 5 we discussed searching using the k-BWT. The k-BWT guarantees that the suffixes prefixed by a pattern up to length k are continuous. The v-BWT provides several guarantees. All patterns of length k_{min} occur continuously similar to the k-BWT. If a pattern occurs at least v times, it is also guaranteed to be represented by a continuous range of suffixes. If pattern $P[0..i]$ occurs less than v times, but pattern $P[0..i-1]$ occurs more than v times the same guarantee applies. Thus the search capability of the index are bound by the number of occurrences of the patterns.

We used the properties of the v-BWT discussed above in the context of pattern matching. Instead of partitioning a pattern into $Y + 1$ pieces of length k, we partitioned the pattern into variable length sub-patterns. The partitioning is determined by the dynamic programming algorithm of Navarro and Baeza-Yates [1998] in conjunction with the *backward search* procedure over the v-BWT. Our experimental evaluation showed that the transform can be used to construct variable length k-gram

indexes faster than previous methods. We showed that the number of verifications that have to be performed using a variable k-gram index is less than traditional fixed k-gram-based indexes. We further showed that using wavelet trees over the transform output can be used as the vocabulary component in the approximate index.

In this chapter we provided applications to non-BWT-based succinct text indexes in the context of approximate pattern matching. We showed that our index can be a viable alternative to traditional fixed k-gram-based indexes. In the next chapter we provide another study which applies succinct text indexes in the area of IR. Specifically we compare a succinct text index-based document retrieval approach to highly optimized inverted indexes on large, standard IR text collections.

Chapter 7

Information Retrieval Using Succinct Text Indexes

Top-ϕ document retrieval algorithms are important for a variety of real world applications, including web search, on-line advertising, relational databases, and data mining. Efficiently ranking answers, based on relevance to queries, in large data collections continues to challenge researchers as the collection sizes grow, and the ranking metrics become more intricate. Formally the **top-ϕ ranked document search** problem is defined in Definition 3 in Chapter 1.

The most common index used to solve the ranked document search problem is the *inverted index*. The inverted index consists of three main components, the *vocabulary*, the *document store* and the *postings lists*. During index construction, each document in the text is segmented into an ordered set of terms, stored into postings lists, and accessed via the vocabulary. A variety of time and space trade-offs in regards to storing and accessing the main components of the inverted index exist [Zobel and Moffat, 2006]. In general, during query time the postings lists are processed to determine the result set. The processing of postings lists can be categorized into two basic approaches: *term-at-a-time* (**TAAT**) and *document-at-a-time* (**DAAT**) query processing discussed in detail in Section 2.6.3.

Large memory systems also provide new opportunities to explore another class of indexing algorithms such as the *suffix array*, to potentially improve the efficiency of various in-memory document retrieval tasks [Manber and Myers, 1993; Muthukrishnan, 2002]. Specifically, compressed representations of the suffix array such as the *FM-Index* can reduce the space required for such an index to that of the compressed representation of the input text [Ferragina and Manzini, 2000]. Recently, these suffix-based indexes have been used to solve the *document listing problem* [Muthukrishnan, 2002] as formally defined and discussed in Section 2.6.1.

Thus, instead of indexing a monolithic text, suffix-based indexes have been used to process document -based text collections. Recent research in this area has focused on the document listing problem [Välimäki and Mäkinen, 2007]. In addition, several theoretically optimal and practically efficient algorithms have recently been proposed to solve the top-ϕ most frequent document listing problem [Culpepper et al., 2010; Hon et al., 2009].

In this chapter, we utilise suffix-based text indexes and auxiliary data structures to solve the ranked document search problem. Specifically, we present a hybrid algorithmic framework for in-memory bag-of-words ranked document retrieval using a self-index derived from *FM-Indexes*, *wavelet trees*, and *compressed suffix trees*, and evaluate the various algorithmic trade-offs for performing efficient in-memory ranked querying. Our contributions and the structure of this chapter can be summarized as follows:

1. We propose a hybrid approach to solving a subset of important top-ϕ document retrieval problems – *bag-of-words* queries.

2. We present a comprehensive efficiency analysis, comparing in-memory inverted indexes with top-ϕ self-indexing algorithms for bag-of-words queries on text collections an order of magnitude larger than any other prior experimental study.

3. To our knowledge, this is the first comparison of this new algorithmic framework for realistically sized text collections using a highly successful and widely used similarity metric – **BM25**.

4. Finally, we describe how our algorithmic framework can be extended to efficiently and effectively support other fundamental document retrieval tasks.

7.1 Document Retrieval

The research area of document retrieval is generally divided into two fields. From a practical perspective, inverted indexes have been used for many years in the IR community to provide search results on a document level. From a theoretical perspective, Muthukrishnan [2002] reintroduced the idea of document retrieval which has in recent years been a focus of research in the Stringology community. Here we give a brief overview of document retrieval techniques used in this chapter. We briefly review the similarity measure and inverted index-based retrieval techniques used in this chapter in Sections 7.1.1 and 7.1.2. From the theoretical view of document retrieval, we discuss

the self-index-based approach we use in this chapter to compare against classic inverted index-based document retrieval in Section 7.1.3.

7.1.1 Similarity and Top-ϕ Retrieval

In this chapter we focus on solving the top-ϕ document retrieval problem. As formally defined in Definition 1, only the ϕ documents with the highest relevance are retrieved. The retrieved documents are ordered based on a similarity measure $S(q, \mathcal{D}_i)$. In this work, we focus primarily on bag-of-words queries, so our baseline $S(q, \mathcal{D}_i)$ ranking function is the widely used **BM25** metric Robertson et al. [1994b] defined and discussed in detail in Section 2.6.3. The metric uses the number of times a query term occurs in a document ($f_{q_i,j}$), the number of document a term occurs in (f_{q_i}) and the size of a document compared to the collection average to determine the relevance of a document to the current query.

7.1.2 Inverted Index-based Document Retrieval

Traditional approaches to the top-ϕ ranked document search problem rely on *inverted indexes*. Inverted indexes have been the dominant data structure for a variety of ranked document retrieval tasks for more than four decades Zobel and Moffat [2006]. Despite various attempts to displace inverted indexes from their dominant position for document ranking tasks over the years, no alternative has been able to consistently produce the same level of efficiency, effectiveness, and time-space trade-offs that inverted indexes can provide (see, for instance an extensive comparison of inverted indexes and signature files Zobel et al. [1998]). We provide a brief introduction to inverted indexes in Section 2.5.4.

To solve the top-ϕ ranked document search problem defined above, only the top-ϕ documents are returned, and, as a result, researchers have proposed many heuristic approaches to improve the efficiency of top-ϕ retrieval systems based on inverted indexes [Buckley and Lewit, 1985; Anh and Moffat, 2002; Persin, 1994; Broder et al., 2003; Moffat and Zobel, 1996]. In Section 2.6.3 we discuss the main heuristics used in this chapter: **WAND** for **DAAT** processing and **MAXSCORE** for **TAAT** query processing. Both heuristics are designed to allow *early termination* of the processing of postings lists while not significantly affecting the quality – the effectiveness – of the retrieval system.

7.1.3 Succinct Text Index-based Document Retrieval

The FM-Index is the most common suffix array-based self-index [Ferragina and Manzini, 2000]. It takes space equal to the compressed representation of the indexed text collection and can answer

$count$ queries in $\mathcal{O}(m \log \sigma)$ time. The FM-Index is described in detail in Section 2.5.2. One of the main components of FM-Index implementations is the wavelet tree [Grossi et al., 2003]. It efficiently supports performing $rank$ and $select$ over sequences and is described in detail in Section 2.3. Wavelet trees are a surprisingly versatile data structure, and have attractive time and space bounds for many primitive operations in self-indexing algorithms [Ferragina et al., 2009; Gagie et al., 2012b]. In the context of document retrieval, we are specifically interested in operations supported by the wavelet tree to efficiently retrieve the top-ϕ most *frequent* symbols in any range $[i, j]$ of the sequence. Culpepper et al. [2010] describe two algorithms using wavelet trees, **GREEDY** and **QUANTILE**, which can efficiently retrieve the top-ϕ symbols in any range using a wavelet tree. In their experiments, they found **GREEDY** to outperform **QUANTILE**. The algorithms are described in detail in Section 2.3.3.

A second data structure which does not rely on wavelet trees is the *skeleton suffix tree* structure of Hon et al. [2009] which we refer to as **HSV**. Hon et al. [2009] create a suffix tree over T, and store for specific suffix tree nodes the most frequent symbols corresponding to the subtree of the sampled nodes. In essence, they create a new suffix tree which consists of several selected nodes of the original suffix tree. The number of nodes and the number of values stored at each selected node is carefully balanced to bound the space requirements of the structure. This process is described in more detail in Section 2.6.2. The **HSV** structure can be used to bound the time to determine the top-ϕ most frequent symbols in a range as follows. The selected nodes of the skeleton suffix tree are chosen to ensure that at most $2g$ positions in any range of the selected sequence have to be processed at query time. The parameter g is chosen at construction time and can be adjusted for different time and space trade-offs. If the range $[i, j]$ to be queried is larger than $2g$, the **HSV** structure guarantees there to be a node in the skeleton suffix tree which at least partially pre-stores values in $[i, j]$. Thus, even for larger ranges, at most $2g$ cells are processed to retrieve the top-ϕ most frequent symbols. Section 2.6.2 describes this process in more detail. Navarro et al. [2011] use **HSV** in combination with the efficient wavelet tree top-ϕ approach of [Culpepper et al., 2010]. For small ranges smaller than $2g$ for which **HSV** does not contain pre-stored values, the **GREEDY** approach is used to calculate the top-ϕ most frequent symbols. For large ranges, the **HSV** structure guarantees that a sample point (at least partially) covers the $\langle sp, ep \rangle$ range. This technique can be used to limit the work to be performed during query time at the cost of storing the **HSV** structure.

Most document retrieval solutions additionally use the notion of the document array (DA). Formally DA is defined as:

$$\text{DA}[i] = j \qquad \text{if} \qquad \text{SA}[i] \in D_j.$$

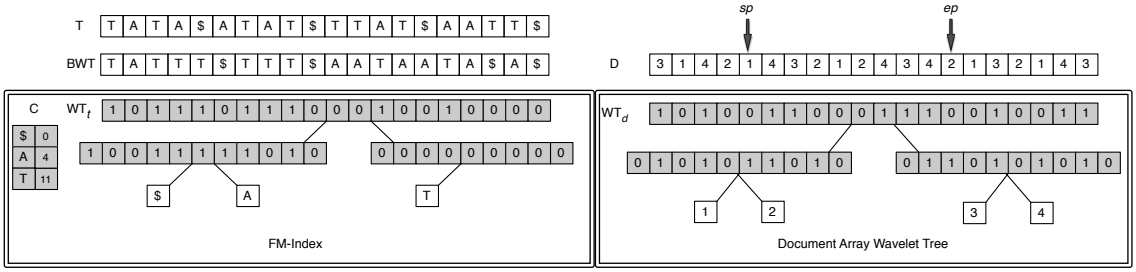

Figure 7.1: Given the text collection $\mathcal{T}$ = TATA$ATAT$TTAT$AATT$ of four documents, our self-indexing system requires two wavelet trees. The first wavelet tree supports backwards search over the BWT *permuted text, and the second supports statistical calculations over the document array. Note that only the items in grey are stored and used for query operations.*

That is, DA$[i]$ is j if the suffix at position i in SA occurs in document D_j. Many document retrieval data structures solving the document listing problem using the document array are described in more detail in Section 2.6. However, to our knowledge, none of these existing document retrieval algorithms have been used to solve multi-pattern bag-of-words queries.

Using the techniques and data structures described above, we now describe our general approach to in-memory indexing and retrieval to solve the top-ϕ ranked document search problem. Figure 7.1 shows the key components of our retrieval system: our system consists of (1) an FM-Index using a wavelet tree over the BWT and (2) a wavelet tree over DA (WT_{DA}). We also create the **HSV** structure over T, storing the most frequent values from DA at each selected suffix tree node. In addition, our system requires a *Document Map* to map document identifiers to human readable document names (or URLs). No document cache is required and the original documents or snippets around each match can be recreated directly from the FM-Index by extracting the required text positions using suffix array sampling (a commonly-used technique described in Section 2.5.2). Only the items in grey are stored and used for character-based top-ϕ document retrieval. All other components are shown for illustration purposes only.

A simple bag-of-words search using a self-index retrieval system is outlined in Algorithm 2. The algorithm consists of three stages: First, for each query term q_i we determine the range $\langle sp, ep\rangle$ of suffixes which are prefixed by q_i (Line 3). Second, we retrieve the top-ϕ documents within the range $\langle sp, ep\rangle$ (Line 4). Third, for all query terms in q, we accumulate the document scores using the accumulators A (Lines 6-14). This algorithm is analogous to **TAAT** processing (as each term is processed one after the other), and is referred to as **SELF-TAAT**. Note that this algorithm is a simple **TAAT** algorithm. However, we are not guaranteed to return the true top-ϕ result list as we process

Algorithm 2 **SELF-TAAT** from a list of query terms $q_i \in q$ and a threshold ϕ return a list of ϕ documents in rank order.

FUNCTION **SELF-TAAT**(q, ϕ)

Initialize a max-heap $R \leftarrow \{\}$
for each query term q_i in q **do**
    Determine $\langle sp, ep \rangle$ for term q_i using the FM-Index
    $A_i \leftarrow$ **GREEDY**$(\langle sp, ep \rangle, \phi)$ ▷ Calculate the top ϕ documents for q_i
end for
for each query term q_i in q **do**
    for $j \leftarrow 1$ **to** ϕ **do** ▷ Accumulate document scores for each q_i
        if $A_i[j] \in R$ **then**
            UPDATE$(R, A_i[j], score)$
        else
            ADD$(R, A_i[j], score)$
        end if
    end for
end for
return $R[1 \dots \phi]$

FUNCTION **GREEDY** ($\langle sp, ep \rangle$,ϕ)

$\ell \leftarrow \text{WT}_d$.root
$h \leftarrow$ **PUSH**$(\ell, \langle sp, ep \rangle)$ ▷ Init max-heap h sorted by size of range $[sp, ep]$
Initialize result list RES $\leftarrow \{\}$.
$i \leftarrow 0$
while $h \neq \emptyset$ **and** $i < \phi$ **do** ▷ Traverse wavelet tree until we found ϕ documents.
    $\ell, \langle sp', ep' \rangle \leftarrow$ **POP**(h)
    if ℓ is leaf **then** ▷ new top ϕ document found
        RES $\leftarrow$ **ADD**$(\ell$.docid$, ep' - sp' + 1)$
        $i \leftarrow i + 1$
    else
        $[s_0, e_0] \leftarrow [rank(\mathcal{B}_\ell, sp', 0), rank(\mathcal{B}_\ell, ep', 0)]$ ▷ map current range left and right
        $[s_1, e_1] \leftarrow [rank(\mathcal{B}_\ell, sp', 1), rank(\mathcal{B}_\ell, ep', 1)]$
        if $e_0 - s_0 > 0$ **then** $h \leftarrow$ **PUSH**$(\ell$.left$, [s_0, e_0])$
        end if
        if $e_1 - s_1 > 0$ **then** $h \leftarrow$ **PUSH**$(\ell$.right$, [s_1, e_1])$
        end if
    end if
end while
return RES

each query term only up to a certain depth. Improving this limitation is discussed as part of the experimental evaluation and is considered future work. Several variations of our general strategy exist, which will be discussed next.

Determining the Range $\langle sp, ep \rangle$

In the context of pattern matching using suffix arrays, the common notation to denote the range of suffixes prefixed by the search pattern is $\langle sp, ep \rangle$. Recall that the sp and ep range for any string can be found using the FM-Index part of our system. The $\langle sp, ep \rangle$ for each query term in Line (3) of Algorithm 2 can be calculated in $\mathcal{O}(|q_i| \log \sigma)$ time using an FM-Index where σ is the size of the alphabet of T. We refer to this approach as FM-Index based approach which we denote with the **FM-** prefix.

A second approach that does not require the FM-Index during query time is described next. Observe that in typical bag-of-words query processing over English text, the size of the vocabulary is often small relative to the total size of the collection. As such, we also present a new hybrid approach to top-ϕ bag-of-words retrieval using a *Term Map* and WT_{DA}. If we assume the vocabulary is fixed for each collection, then the $\langle sp, ep \rangle$ range for each term can be pre-calculated and retrieved using a term map, as in the inverted indexing solution. This means that the FM-Index component is no longer necessary when processing bag-of-words queries. We refer to this approach as "sp-ep map" (as a term is mapped to the corresponding range $\langle sp, ep \rangle$) which we denote with the prefix **SEM-** prefix. Note that this approach reduces the overall space requirements of our approach, but also limits the full functionality of some auxiliary operations. For example, the text can no longer be reproduced directly from the index, so snippets cannot be generated on-the-fly, and phrase queries are no longer natively supported.

Retrieving the Top-ϕ Documents

The second step in our bag-of-words succinct text index approach determines the top documents in the $\langle sp, ep \rangle$ range for each query term. Here we discuss two alternatives explored in this chapter to retrieve the top-ϕ documents for each range.

Algorithm 2 uses the **GREEDY** strategy of Culpepper et al. [2010] to traverse the wavelet tree over $\text{DA}(\text{WT}_{\text{DA}})$ to retrieve the top-ϕ documents. We refer to this approach with the suffix **-GREEDY**. We do not evaluate the **QUANTILE** approach of Culpepper et al. [2010] as their experiments indicate it is generally outperformed by **GREEDY** when ϕ is of reasonable size.

As a second approach we investigate augmenting the **GREEDY** approach with the **HSV** structure discussed above and described originally by Navarro et al. [2011]. Instead of storing the top-ϕ most frequent symbols in the skeleton suffix tree, we store the top-ϕ most important symbols sorted by term impact for each interval g to improve effectiveness. In order to capture the ϕ-values commonly used in IR systems ($k = 10, 100, 1000$), we pre-store values of any ϕ that are a power of 2 up to

8192 in term contribution order. We further experimented with different sampling intervals g, and storing only results for $k = 10, 100, 1000$. This affects the theoretical space guarantees of the **HSV** method [Hon et al., 2009] but works well in practice. Note that we go higher than 1024 since the values of ϕ' necessary to ensure good effectiveness can be greater than the desired ϕ (which we explore in our experimental evaluation). We refer to this skeleton suffix tree approach with the suffix **-HSV**.

Evaluated Retrieval Techniques

Below we summarize the different succinct text index-based approaches evaluated in the chapter:

FM-GREEDY	Uses the FM-Index to determine $\langle sp, ep \rangle$ and the standard **GREEDY** technique of Culpepper et al. [2010] to retrieve the top document identifiers.
FM-HSV	Uses the FM-Index to determine $\langle sp, ep \rangle$. For large ranges **HSV** pre-computed top-ϕ values are used to answer queries. For query terms not completely covered by **HSV**, the standard **GREEDY** technique of Culpepper et al. [2010] is used to retrieve the top document identifiers.
SEM-GREEDY	Use a hash-table to determine $\langle sp, ep \rangle$ and the standard **GREEDY** technique of Culpepper et al. [2010] to retrieve the top document identifiers.
SEM-HSV	Use a hash-table to determine $\langle sp, ep \rangle$. For large ranges **HSV** pre-computed top-ϕ values are used to answer queries. For query terms not completely covered by **HSV**, the standard **GREEDY** technique of Culpepper et al. [2010] is used to retrieve the top document identifiers.

Alternative Self-Indexing Retrieval Techniques and Problems

It is also possible to support a **DAAT** query processing strategy in our retrieval system. The wavelet tree over the document array (WT_{DA}) supports *Range Quantile Queries* (RQQ). This operation allows the traversal of any range in DA in document order at a cost of $\mathcal{O}(\log d)$ per access (see Section 2.3.3 for a detailed explanation). Thus, in a range of size $m = ep - sp + 1$ containing m' distinct document identifiers, we can retrieve all document identifiers in document order in $\mathcal{O}(m' \log d)$ time.

Also note the current top-ϕ bag-of-words approach shown in Algorithm 2 is based entirely on the frequency counts of each query term. This means that our current implementation only approximates

the top-ϕ items. This is a well-known problem in the inverted indexing domain. This limitation holds for any character-based bag-of-words self-indexing system that does frequency counting at query time since we can not guarantee that item $\phi + 1$ in any of the term lists does not have a higher score contribution than any item currently in the top-ϕ intermediate list. A method of term contribution pre-calculation is required in order to support **BM25** or language-model processing. Without the term contribution scoring, **WAND** and **MAXSCORE** enhancements are not possible, and therefore every document in the $\langle sp, ep \rangle$ must be evaluated in order to guarantee the final top-ϕ ordering. However, this limitation can be mitigated by using **HSV** since we can pre-calculate the impact contribution for each sample position and store this value instead of storing only the frequency ordering. Top-ϕ guarantees are also possible using a term-based self-indexing system where each distinct term is mapped to an integer using **HSV** or other succinct representations of term contribution preprocessing. In future work, we intend to fully examine all of the possibilities for top-ϕ guarantees using self-indexes in various bag-of-words querying scenarios.

When using character-based self-indexing approaches for bag-of-words queries, there is another disadvantage worth noting: it is difficult to determine the unique number of documents (f_{q_i}) a query term occurs in. For self-indexes, there is an efficiency trade-off between locating the top-ϕ $f_{q_i,j}$ values and accurately determining f_{q_i} since the index can extract exactly ϕ $f_{q_i,j}$ values without processing every document. For a fixed vocabulary, f_{q_i} is easily precomputed, and can be stored in the term map with the $\langle sp, ep \rangle$ pairs. But, in general it is not straightforward to determine f_{q_i} for arbitrary strings over WT_{DA} without auxiliary algorithms and data structures to support calculating the value on-the-fly. The **FM-HSV** approach allows us to pre-store f_{q_i} for each sampled interval which can be used to calculate f_{q_i} over $\langle sp, ep \rangle$ more efficiently by only processing potential fringe leaves. Calculating f_{q_i} using only WT_{DA} for arbitrary strings in near constant time using no additional space remains an open problem. To our knowledge, using a wavelet tree is the most efficient approach for performing f_{q_i} range quantile queries in $\mathcal{O}(f_{q_i} \log d)$ time.

7.2 Empirical Evaluation

In order to test the efficiency of our approach, two experimental collections were used. For a small collection, we used the TREC 7 and 8 *ad hoc* datasets. This collection is composed of 1.86 GB of newswire data from the *Financial Times*, *Federal Register*, *LA Times*, and *Foreign Broadcast Information Service*, and consists of around 528,000 total documents [Voorhees and Harman, 1999]. For a larger in-memory collection, we used the TREC WT10G collection. This collection consists of 10.2 GB of markup text crawled from the internet, totalling 1,692,096 documents [Hawking, 1999].

For our baselines, we have implemented the in-memory variant of **WAND** as described by Fontoura et al. [2011] for **DAAT**, and an in-memory variant of **MAXSCORE** for **TAAT**. Experiments were run on our LARGE test machine described in Section 2.7.1. Times are reported in milliseconds unless otherwise noted. All efficiency runs are reported as the mean and median of 10 consecutive runs of a query, and all necessary information is preloaded into memory with warmup queries.

Note that we do not carry out a full evaluation of the effectiveness of the retrieval systems presented here. In previous work, it was shown that the **BM25** ranking and query evaluation framework used in our approach can be as effective as other state-of-the-art open source search engines when an exhaustive retrieval is performed. We thus do not repeat those experiments here [Culpepper et al., 2011]. To achieve this, a larger number of documents $\phi' > \phi$ is retrieved. For exhaustive retrieval, $\phi' = d$ where d is the total number of documents in the collection. In our experiments, we use the minimum ϕ' values that result in retrieval performance that is comparable to the effectiveness obtained through exhaustive processing. In all experiments we use $\phi' = 8 * \phi$ for the TREC 7 & 8 dataset, and $\phi' = 2 * \phi$ for the TREC WT10G dataset. These values for ϕ' give results for the MAP and P@10 effectiveness measures that are not statistically significantly different compared to exhaustive processing, for both collections (paired t-test, $p > 0.05$).

7.2.1 Experimental Setup

The queries used in our experiments were extracted from a query log supplied by Microsoft. Each query was tested against both TREC collections, and the filtering criteria used was that every word in the query had to appear in at least 10 distinct documents, resulting in a total of 656,172 unique queries for the TREC 7 & 8 collection, and a total of 793,334 unique queries for the TREC WT10G collection. From the resulting filtered query sets, two different query samples were derived.

First, 1000 queries of any length were randomly sampled from each set, to represent a generic query log run. The 1,000 sampled queries for TREC 7 & 8 have an average query length of 4.224, and the average query length of the WT10G sample set is 4.265 words per query. For the second set of experiments, 100 queries for each query length 1 to 8 were randomly sampled from the same MSN query sets.

Table 7.2 shows the statistical properties of the sampled queries that were used in the second experimental setup, including the average number of documents returned for each query for each query length, and the average length of postings lists processed for each query.

Query length	TREC 7 & 8 queries			TREC WT10G queries		
$\|q\|$	Total Queries	Matches ('000)	Avg Processed ('000)	Total Queries	Matches ('000)	Avg Processed ('000)
1	100	9.9	9.9	100	3.3	4.7
2	100	24.8	12.5	100	68.6	42.5
3	100	104.5	38.5	100	292.8	123.2
4	100	238.1	69.0	100	601.1	166.6
5	100	351.2	95.1	100	866.5	228.5
6	100	408.7	107.8	100	1041.9	280.4
7	100	463.8	126.2	100	1149.7	319.6
8	100	489.8	148.3	100	1171.8	339.2
random sample	800	234.9	70.0	800	621.5	181.2

Table 7.2: Statistics of the queries used in experiments (sampled based on query length, or sampled from the filtered MSN query log), reporting the number of queries run, the mean number of documents that contained one or mor of the query terms, and the mean length of the inverted lists processed.

7.2.2 Average Query Efficiency

In order to test the efficiency of our algorithms, two experiments were performed on each of the collections. The first experiment is designed to measure the average efficiency for each algorithm, given a sampling of normal queries. For this experiment, the length of the queries was not bounded during sampling, and had an average query length of just over 4 words per query as mentioned in Section 7.2.1.

Figure 7.2 shows the relative efficiency of each method averaged over $1{,}000$ randomly sampled MSN queries for TREC 7 & 8, and TREC WT10G. Each boxplot summarizes the time values as follows: the solid line indicates the median; the box shows the 25th and 75th percentiles; and the whiskers show the range, up to a maximum of 1.5 times the interquartile range, with outliers beyond this shown as separate points. In both figures, the following abbreviations are used for the algorithms: **FM-GREEDY** (**FM**), **SEM-GREEDY** (**SE**), **FM-HSV** (**FM-H**), **SEM-HSV** (**SE-H**), **DAAT**, and **TAAT**.

The following observations can be made. There is no noticeable difference between the performance of the **FM-** and **SEM-** methods. Thus, the FM-Index retrieves the $\langle sp, ep \rangle$ ranges as fast as the hash table. On the other hand, the hash table is equally fast, and can thus replace the FM-Index if the additional functionality provided by the FM-Index is not required. The identical performance can be explained by the fact that the hash table always performs at least one string comparison (or more in the non-optimal case) to compare the query term q_i to the current hash table entry. The FM-Index performs $|q_i| \log \sigma$ binary rank operations to retrieve the range. The difference in lookup cost does

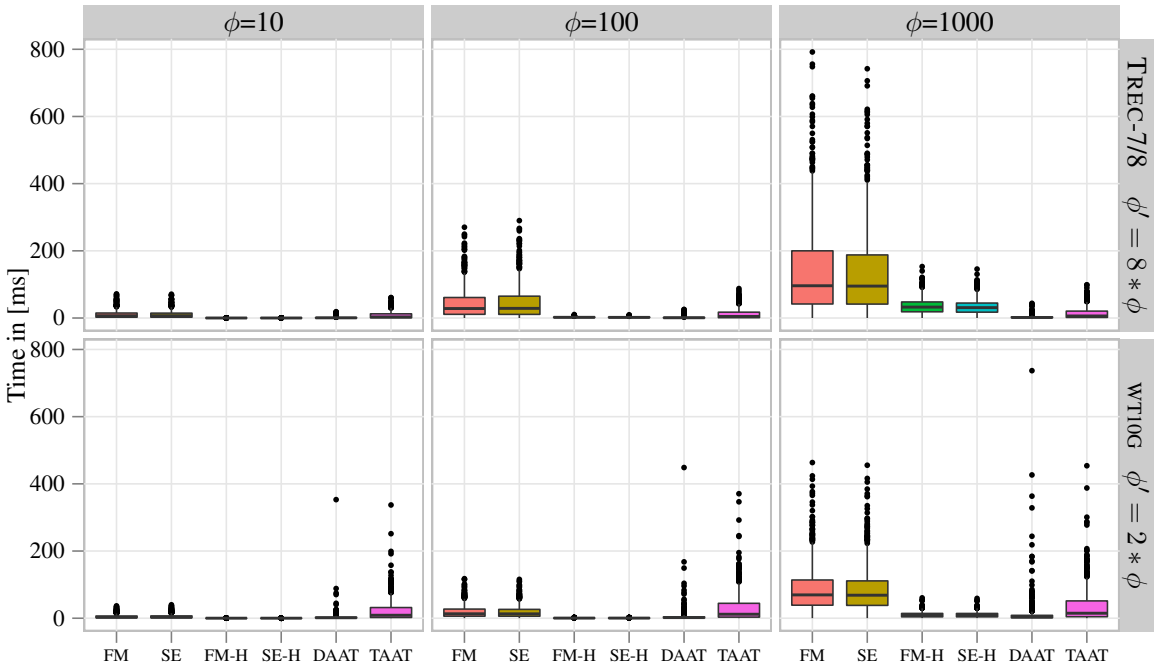

Figure 7.2: Efficiency for 1,000 randomly sampled MSN queries against the **TREC**-7/8 *and* WT10G *and collections.*

not contribute to the query performance of the different methods.

For $\phi = 10$, the self indexing methods **FM-GREEDY** and **SEM-GREEDY** perform similar to **DAAT** and outperform the **TAAT** baseline for both data sets. The **HSV**-based methods outperform all other methods for $\phi = 10$ and $\phi = 100$. While the median performance is similar to that of **DAAT**, the lack of outliers is noticeable for both baselines. Further evaluation of the performance of the **HSV**-based methods indicated that for small values of ϕ, almost all queries directly correspond to the sampled nodes in the skeleton suffix tree. This can be explained by the way the nodes are chosen. The nodes are chosen by performing *lowest common ancestor* (LCA) operations in the suffix tree. For our text collections, and we conjecture for many natural language text collections, these sampled LCA nodes correspond to word boundaries. We conjecture that this is due to the fact that words tend to repeat often in natural language text. Thus, large ranges of the suffix array of the text collections are prefixed by words. Performing LCA operations within such a range results in a node being marked which corresponds to the word boundary. Therefore the **HSV** method performs well in our experiments as many queries can directly be answered by retrieving pre-stored value lists.

Interestingly, for $\phi = 1000$, the **HSV** method is outperformed by **DAAT**. This can be explained

using a similar argument as above. As discussed in Section 2.6.2, the sampled nodes in the **HSV** structure are selected based on the parameter g. The nodes are selected by computing the lowest common ancestor of the two leaves g apart in the suffix tree. Thus node $lca(sa[i], sa[i+g])$ is selected and values are pre-stored. However, to ensure the space-bounds of the structure, g is defined as $g = \phi \log^{2+\epsilon} n$. Thus, the nodes selected depend on ϕ, which implies for larger values of ϕ, less nodes are sampled. In our experiments, for $\phi = 1000$, the sample range g of the **HSV** structure was larger than the word boundaries in the text collection, which caused the performance of the **HSV**-based index types to degrade as ϕ increased, due to the fact that less query terms can directly be answered using the **HSV** structure. If queries cannot be answered using **HSV**, the method degrades to using **FM-GREEDY** to retrieve the ϕ results for each query term.

Similar to the **HSV** structure, the regular **GREEDY**-based techniques also perform worse for larger ϕ values. The performance of the **GREEDY** technique only depends on the processing within the document wavelet tree, WT_{DA}. The depth of WT_{DA} increases as the number of documents in the collection (d) increases as the height of the wavelet tree is $\log d$. In the case of the TREC WT10G collection, which contains around 1.6 million documents, the depth of the wavelet tree is 24. The amount of work to be performed by **GREEDY** depends on (1) the height of WT_{DA}(2) the number of unique elements in $\langle sp, ep \rangle$ and (3) the desired number of elements, ϕ, to be returned.

We first focus our discussion on (2), the number of unique elements in the range. The number of unique elements in the $\langle sp, ep \rangle$ range corresponds to the number of documents containing the query term. In the context of relevance, query terms occurring in many documents are considered less important to the similarity measure. Consider the **BM25** metric defined in Section 7.1.1. The first term, $\left(\frac{d - f_{q_i} + 0.5}{f_{q_i} + 0.5}\right)$, in the equation weights a query term q_i less, as the number of times the term occurs in the collection (f_{q_i}) increases. For example, the term "the" occurs in almost all English texts, and thus is considered less important by **BM25**. This can be translated to the processing of the wavelet tree. Query terms which require large parts of the wavelet tree to be processed are often irrelevant to the overall relevance ranking process. In the context of inverted indexes, these terms are often referred to as stop words, which are excluded from the index during construction time.

Last we discuss how ϕ affects the run-time performance of **GREEDY**. Recall that for, $\phi = 1000$, the **GREEDY** method is outperformed by both **DAAT** and **TAAT** processing for both test collections. To explain the drop in run-time performance for **GREEDY**, we examine the way the method traverses the WT_{DA} to return the ϕ most frequent documents within $\langle sp, ep \rangle$. The algorithm starts by mapping the range to corresponding ranges r_{left} and r_{right} of both the left and right subtree of the root node. Using a max-heap, the larger of the two mapped ranges is processed next. Each processing step maps the current range to the corresponding ranges of the children within the current node. The

algorithm is greedy as the largest currently available range always is processed next. The most frequent symbol/document is encountered the first time a range is mapped to a leaf node. The process continues until ϕ leaf nodes have been visited. Thus, the processing for $\phi = 1000$ stops after 1000 leaf nodes are visited. However, this does not limit the number of nodes within WT_{DA} to be processed. Assume every element in $\langle sp, ep \rangle$ occurs only once which would require $\mathcal{O}((sp - ep + 1) \log d)$ time. For natural language text, only few documents contain a query term often. Most documents contain each query term only a small number of times. This case is very similar to the worst case described above, which requires large parts of the wavelet tree to be processed. This explains the degraded performance of the **GREEDY** method as ϕ increases. Interestingly, this could be used to create an approximation technique such as **WAND**. For example, continue processing the wavelet tree until all ranges are smaller than a certain threshold. Below the threshold it is guaranteed that the returned document will not be in the ϕ most relevant documents. We plan to explore this idea in future work.

In general, the **WAND** variant of **DAAT** is more efficient on average, but can perform poorly for certain queries (see outliers in Figure 7.2). For example, the query "`point out the steps to make the world free of pollution`" on the WT10G collection consistently performed poorly in our **DAAT** framework. This can be explained by the fact that many of the query terms have very similar impact (first term in the **BM25** formula) which makes skipping using **WAND** difficult as many parts of the postings lists can not be skipped and have to be processed completely. As discussed by Fontoura et al. [2011] we find that **DAAT** processing can be performed more efficiently than **TAAT** using in-memory inverted indexes.

7.2.3 Efficiency Based on Query Length

We now break down the efficiency of each of our algorithms relative to two parameters: ϕ and $|q|$, where $|q|$ is the number of terms in a query. Figure 7.3 shows the average of 10 runs of 100 queries per query length, $|q|$. For one-word queries, for all values of ϕ, the inverted indexing approaches **DAAT** and **TAAT** are superior. This is not surprising since only a single term posting must be traversed to calculate **BM25**, and the algorithms have excellent locality of access. Still, the **HSV** variant is the most efficient for small ϕ. For $|q| > 1$, the results also depend on ϕ. For $\phi = 10$ and $\phi = 100$, the self-indexing methods are more efficient than **TAAT** since the methods can efficiently extract the still relatively small ϕ' values. The **WAND**-based **DAAT** method remains remarkably efficient for all values of ϕ. As ϕ increases, the performance of the **HSV**-based approaches begin to degrade. The performance degradation at large ϕ' is as outlined in Section 7.2.2. Most of the $\langle sp, ep \rangle$ ranges turn out to be much smaller than any of the samples, as the sample rate depends on ϕ'. So, the complete

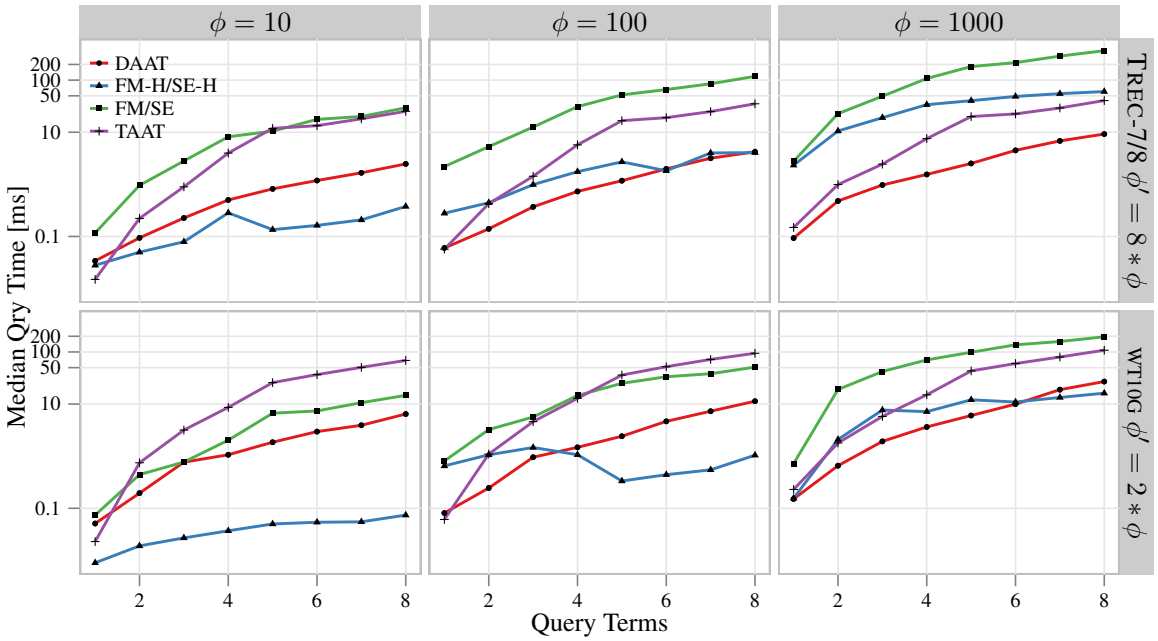

Figure 7.3: Efficiency of query length of 1 *to* 8 *on the* **TREC** *7 & 8 collection (top row) and* **TREC** *WT10G collection (bottom row) for* ϕ*=10,100 and 1000. For each query length, 100 randomly sampled queries are used from the MSN query log set.*

$\langle sp, ep \rangle$ range must be computed at runtime, reducing the performance to **FM-GREEDY** when an appropriate sample is not available. Note that the performance of **HSV** for TREC 7 & 8 is worse than for WT10G as ϕ' is four times larger in TREC 7 & 8 resulting in fewer sample points.

7.2.4 Space Usage

We now address the issue of space usage for the different algorithmic approaches. Inverted indexes are designed to take advantage of a myriad of different compression techniques. As such, our baselines also support several state-of-the-art byte- and word-aligned compression algorithms [Moffat and Anh, 2005; Trotman, 2003; Yan et al., 2009]. When we report the space usage for an inverted index, the numbers are reported using compressed inverted indexes and compressed document collections.

Figure 7.4 presents a break-down of space usage for each component of the inverted indexing and self-indexing approaches. From a functionality perspective, there are several different componentization schemes to consider. First, consider the comparison of an inverted index method (including the

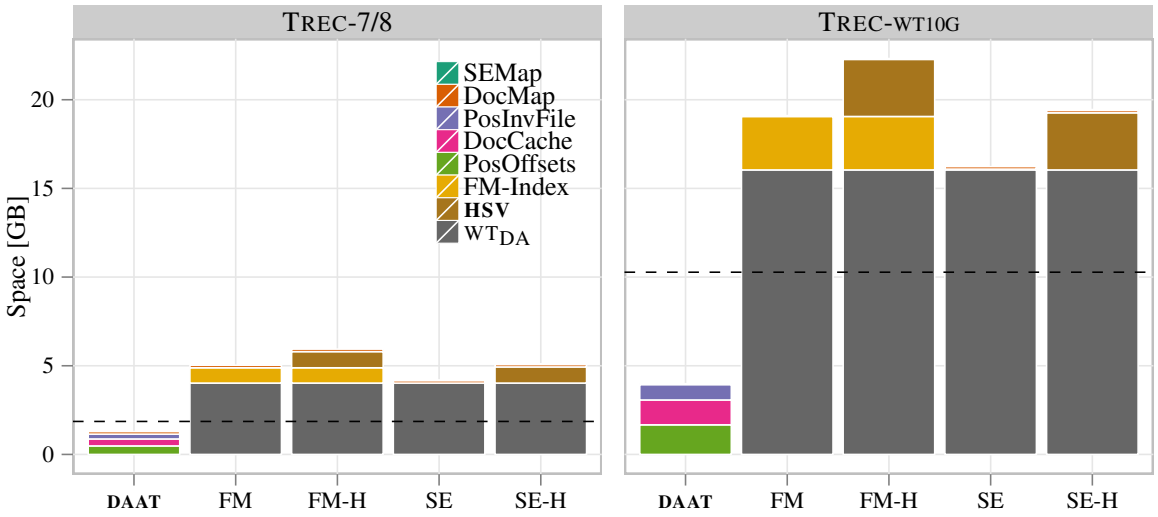

Figure 7.4: Space usage for each component in the three indexing approaches for the **TREC** *7 & 8 collection (left) and the* **TREC** *WT10G collection (right). The dashed lines in both graphs represent the total space usage of the original uncompressed text for the collection.*

term map, the postings list with position offsets, the document map, and the compressed document cache) with an FM-Index (including WT_{DA}, the document map, and any other precomputed values – for instance the **HSV** enhancement). We consider these two in-memory indexes as functionally equivalent, as both can support bag-of-words or phrase queries, and can recreate snippets or even the original uncompressed document. The self-index variant (**FM** or **FM-H**) is significantly larger, but able to support a range of special character and arbitrary sub-string queries that term-based indexes do not support. The second alternative are indexes that support *only* bag-of-words queries. Now, an inverted index method requires only the term map, the postings list *without* position offsets, and the document map. The succinct text index methods are essentially the same (**SE** or **SE-H**), but the FM-Index component is replaced with a term map component. For space usage across all succinct text index-based approaches, the most expensive component is WT_{DA}. The size of the document wavelet tree increases with the number of documents in the collection. While previous work exists on evaluating various trade-offs for compressing WT_{DA} [Navarro et al., 2011], we currently do not compress WT_{DA} because all of the known approaches can have a substantial impact on the overall efficiency of key operations.

When considering all of the current self-indexing options presented here, using an FM-Index component instead of a term map offers the most functionality but requires more space than the

term-map-based index.

7.3 Summary and Conclusion

In this chapter we considered the application of succinct text indexes to in-memory bag-of-words query processing. Generally, answering bag-of-words queries is computationally more expensive for text indexes as the results of multiple query terms have to be combined using a similarity measure during query time. Sophisticated query processing strategies such as **WAND** are used to efficiently answer bag-of-words queries using inverted indexes. To our knowledge, answering bag-of-words queries efficiently using a succinct text index-based document retrieval framework has previously been unexplored. To evaluate the performance of a search system processing bag-of-words queries, both the efficiency and the effectiveness of the index have to be evaluated. Effectiveness describes the quality of the returned results when compared to a gold-standard answer set. In IR, effectiveness is generally evaluated using standardized text collections. One of the obstacles in using succinct text indexes for IR tasks is the size of widely used text collections. In this chapter we use succinct text indexes to index two standard text collections up to 10 GB in size. Thus, we index data sets larger than commonly used in the field of succinct data structure research. Using these standard text collections, we evaluated and compared our succinct text index-based document retrieval indexes to academic standard invert indexing algorithms. To be competitive, our succinct text indexes use the engineered implementations and optimizations discussed in Chapter 3.

Our succinct text index-based document retrieval framework consists of multiple components. We used a FM-Index [Ferragina and Manzini, 2000] or alternatively a term-map to determine the range of suffixes prefixed each query term. Both alternatives provide different trade-offs. The term-map is more space efficient; however, the FM-Index provides additional functionality. The FM-Index, allows, for example, performing phrase queries of any length or extracting any arbitrary substring of the original text collection efficiently. To process the ranges of the matched suffix positions we the **HSV** structure [Hon et al., 2009] in conjunction with a wavelet tree [Grossi et al., 2003] over the document array as suggested by Navarro et al. [2011]. The document array consists of a one-to-one mapping between each suffix position and its corresponding document number. If the range of suffix positions is large, the **HSV** structure is used to retrieve the pre-calculated top-ϕ most frequent documents. Otherwise, the wavelet tree is traversed using the **GREEDY** algorithm [Culpepper et al., 2010] to calculate the top-ϕ most frequent documents within the range on-the-fly.

Comparing our approach to that of Navarro et al. [2011], we introduced several changes to account for bag-of-words queries. First, instead of pre-storing results in frequency order in the **HSV**

structure, we stored each list in impact order [Anh and Moffat, 2006]. Thus, entries contributing more towards the final score of a document are ranked higher in the pre-stored result list. Second, instead of retrieving and storing ϕ results, we store and retrieve the top-ϕ' results. This is required to ensure that the results generated by our succinct text index framework are statistically identical to those generated by the invert index-based systems.

Our empirical evaluation showed that our succinct text index-based document retrieval framework can be competitive to inverted index-based systems. For small ϕ, queries can be answered efficiently using the **HSV** structure. As ϕ becomes larger, the efficiency of our approach deteriorated as less queries can be resolved using the **HSV** structure. This is especially problematic as instead of retrieving ϕ results, we were required to retrieve up to $\phi' = 8 * \phi$ to ensure result quality in terms of effectiveness. Inverted index-based query processing algorithms such as **WAND** avoid this problem by not processing low impact postings lists. However, similar techniques currently do not exist in our succinct text index-based framework. This shortcoming makes our approach only viable for small ϕ when answering bag-of-words queries. The construction of our self-indexes is problematic for large collections, and deserves further study. In particular, constructing the **HSV** structure, which requires a suffix tree (or compressed suffix tree), currently limits the applicability of our approach. From an efficiency perspective, constructing the self-index structures, especially **HSV**, is largely unexplored.

Despite the limitations of our approach discussed above, we believe that succinct text indexes can provide interesting new avenues of research in the field of IR. Using FM-Indexes to perform inexpensive phrase queries and on-the-fly snippet extraction are apparent benefits of succinct text indexes which can make traditional search systems more powerful. In this chapter we provided evidence that succinct text indexes can operate efficiently on the scale required to perform IR experiments. To make succinct text index-based solutions more competitive, partial processing techniques such as **WAND** have to be adopted in our document retrieval framework. We consider this as well as reduction of construction cost as the two outstanding important problems for the wider adoption of succinct text indexes in IR.

Chapter 8

Conclusion and Future Work

In this chapter we discuss our contributions in the context of the our initial problem statements and research questions, as well as potential new research directions and future work. In Section 8.1 we discuss several open problems and extensions of our work. To conclude, we provide a summary of our contributions in Section 8.2 where we reflect on our contributions with regards to the intentions of this research project.

8.1 Future Work

Here we discuss several avenues of future work: we discuss problems related to constructing and parallelizing succinct data structures; applying the k-BWT and the v-BWT in the context of Bioinformatics and, the usefulness of self-indexes in the context of Information Retrieval.

8.1.1 Construction and Parallelism of Succinct Data Structures

In Chapter 3 we focused on techniques to optimize the performance of succinct data structures. Our analysis showed that careful use of, modern CPU instructions and operating system features can substantially increase the run time performance of succinct data structures. The same techniques can also be applied during the construction of succinct data structures. For example, initial experiments showed that enabling huge pages during the construction of a suffix array decreases the construction cost by 30%. In Chapter 3 we further showed that using modern CPU instructions and bit-parallelism during construction can substantially decrease the construction cost of wavelet trees and $select$ operations on bitvectors. Applying the techniques we described in Chapter 3 to the construction phase of other succinct data structures can make succinct data structures, and specifically succinct text indexes,

feasible for use on larger data sets.

It is generally accepted that single thread performance and clock speed will no longer increase substantially in the near future [Sutter, 2005]. Sutter is predicting a fundamental shift in software development focusing on concurrency and parallelism. In fact, constructing suffix arrays in parallel has been an active area of research [Kulla and Sanders, 2007; Deo and Keely, 2013]. The parallel performance of succinct data structures and specifically succinct text indexes remains, to our knowledge, unexplored. Cache-efficient $rank$, $select$ and wavelet tree processing will be able to substantially improve the performance of succinct text indexes accessed in parallel. We further believe that many operations on succinct text indexes such as extracting parts of the original text can efficiently be parallelized.

8.1.2 Context-Bound Text Transformations

In Chapters 4 and 5 we discussed different aspects of a context-bound text transformation: the k-BWT. The k-BWT is created by sorting of all k-grams in the original text in lexicographical order. In the context of Bioinformatics, such k-grams are often referred to as k-mers and are widely used in many Bioinformatics research tools used practice [Altschul et al., 1990; Hazelhurst and Liptk, 2011]. In some cases, these tools construct the full suffix array, only to unsort the individual k-groups to retrieve SA_k [Hazelhurst and Liptk, 2011]. An interesting extension to our work would be to explore using the k-BWT in conjunction with our results on searching in T^{kbwt}, to decrease the space requirements and increase the run-time performance of tools such as **WCD-EXPRESS**, used to cluster Expressed Sequence Tags [Hazelhurst and Liptk, 2011]. Reduced space requirements would in turn enable such tools to scale more efficiently on larger data sets.

A second open problem is induced suffix sorting using only a limited depth k. In Chapter 4 we discussed an initial attempt to use the method of Itoh and Tanaka [1999] to induce the correct k-order of all suffixes. However, each induction step increases the sorting order of the induced position by one. Thus we keep track of suffixes which have incorrect sorting order to correct them in a later step of the algorithm. This negates all savings from having to sort only 50% of all suffixes. We expect that it would be possible to use induced suffix sorting to construct the k-BWT more efficiently than the radixsort-based method of Kärkkäinen and Rantala [2008].

8.1.3 Self-Indexes for Information Retrieval

In Chapter 7 we applied succinct text indexes to the top-ϕ ranked document retrieval problem. In our experimental evaluation, we showed that self-indexes can be competitive when compared to inverted

indexes. One of the main problems of self-indexes is the fact that the top-ϕ most frequent documents for a query term are not guaranteed to retrieve the top-ϕ most relevant documents for a bag-of-words query. Inverted indexes use advanced postings list processing algorithms such as **WAND** or **MAXSCORE** to decide the number of documents have to be processed to retrieve the correct top-ϕ most relevant documents for a query. Unfortunately, these algorithms are not directly transferable to self-index-based top-ϕ retrieval. To achieve comparable effectiveness in our experiments we were required to retrieve up to 8 times the number of top-ϕ most frequent documents (that is $\phi' = 8 * \phi$) for each query term. Especially for large ϕ, the work performed by our succinct text indexes increased substantially, making them less competitive than state-of-the-art inverted indexes. Thus, retrieving the top-ϕ most relevant documents for a standard bag-of-words query efficiently remains an open problem. Potential solutions include adopting inverted index-based techniques such as **WAND** to the self-indexing document retrieval framework.

Self-indexes can answer phrase queries efficiently. In contrast, inverted indexes require auxiliary data structures such as a nextword index [Williams et al., 1999]. Similarly, using an inverted index for non-term-based collections such as East Asian language text requires complex segmentation algorithms to parse the text into a sequences of terms [Nie et al., 2000]. Self-indexes do not have such a requirement. Other tasks which can be performed more efficiently using succinct text indexes, such as on-the-fly snippet extraction or sequential/full dependence Markov random field models [Metzler and Croft, 2005], are again computationally expensive using inverted indexes. In the experiments that we performed in Chapter 7, we used a set of collections that were used in IR evaluation 10 years ago. This set of collection is considered large in the area of succinct text indexes today. However, these same collections are deemed relatively small in the context of IR research, where today, common collections such as **CLUEWEB09** Category B contain 50 million English web pages, or 230 GB of compressed text. These collections are used in many IR evaluations as a variety of relevance judgements and reference queries exist. As a result, indexing large collections such as **CLUEWEB09** is a prerequisite to investigating, for example, the use of phrase queries or advanced dependency models using self-indexes. In short, scaling succinct indexes to enable the indexing of large standard IR test collections is necessary for the viability of succinct data structures in Information Retrieval.

8.2 Conclusion

The scalability of succinct text indexes is one of the main aspects hindering the wider adoption of suffix-based indexes in practice. While inverted indexes can index terabyte-size data sets, suffix-based indexes are currently able to only index comparably smaller data sets. This research project

investigated ways to improve the scalability of succinct text indexes to index large data sets. Specifically, we investigated two aspects relevant to enabling scaling succinct text indexes: *construction cost* and *designing* succinct data structures for large-scale data sets. Here we discuss and summarise our contributions and relate them to our initial aim of the research project.

Traditionally the suffix tree and suffix array were used to solve the exact pattern matching problem [Weiner, 1973; Manber and Myers, 1993]. However, they require space equivalent to many times that of the original text being indexed. Succinct data structures emulate the functionality of a regular data structure in space equivalent to the compressed representation of the underlying data. In this connection, succinct text indexes use space equivalent to the size of the compressed data set to be indexed, while allowing searches to be performed at the same efficiency of a regular text index such as the suffix array. The key insight used by suffix-based succinct text indexes is the fact that the suffix array can be replaced by more compressible representations such as the Burrows-Wheeler Transform (BWT) while providing the same functionality [Ferragina and Manzini, 2000]. Unfortunately, both the FM-Index of Ferragina and Manzini [2000] and the Compressed Suffix Array of Sadakane [2002] require the complete suffix array to be available during construction time. Thus the space required at construction time is much larger than during query time. Construction is therefore a major bottleneck prohibiting the scalability of succinct text indexes to larger data sets.

To alleviate the prohibitive cost of constructing the suffix array during index construction we focused on alternative representations which can be substituted to eliminate the need to construct the full suffix array. In Chapter 4 we revisited a partial sorting, up to a certain depth k, of all suffixes. Specifically, we investigated the k-BWT [Schindler, 1997; Yokoo, 1999], a derivative of the BWT which used to emulate searching over a suffix array in the FM-Index. Unlike the BWT, the k-BWT sorts suffixes only partially. However, we found that it can be compressed without substantial loss in compression effectiveness even for small k. We additionally discovered that the k-BWT can be recovered faster than the BWT due to a previously undetected cache-effect. This was surprising, as reversing the k-BWT requires additional work to recover the context group boundaries which are needed to reconstruct the original text. Most importantly, we discovered that the k-BWT can be constructed more efficiently than the BWT. For small k, a fast radixsort [Kärkkäinen and Rantala, 2008] algorithm can construct the k-BWT faster than state-of-the-art suffix array construction algorithms. Similarly, we proposed a simple external k-BWT construction algorithm which can construct the k-BWT more efficiently in external memory than comparable state-of-the-art external BWT construction algorithms. For this reason, succinct text indexes using the k-BWT instead of the BWT can be built over larger text collections and are thus potentially more scalable.

The FM-Index is one of the most popular succinct text indexes [Ferragina and Manzini, 2000].

It relies on emulating the suffix array using the BWT. To support search, the FM-Index uses the *backward search* procedure to locate patterns in original text by performing $rank$ queries over the BWT. This is possible due to a duality between the suffix array and the BWT. In Chapter 5 we investigated replacing the BWT with the k-BWT, while still providing operations commonly supported by succinct text indexes. We found that due to the incomplete lexicographical ordering of the k-BWT, search is only guaranteed to return the correct results for patterns up to length k. For patterns larger than k, each occurrence position has to be processed individually. We prove, that function LF can be performed at any arbitrary position in the k-BWT by using additional auxiliary data structures. We only use a constant number of extra $rank$ and $select$ operations compared to performing LF over the BWT. The function is used internally by all operations of the FM-Index. Thus our contribution allows, for example, the extraction of any arbitrary substring of the original text from the k-BWT. In the context of search, we elucidate another useful property of the k-BWT: within a context group, each pattern occurrence is implicitly stored in increasing text order. This is especially useful when performing position restricted text searches. Overall, we found that searching for patterns of length k can be performed at no extra cost using a k-BWT based FM-Index. This is useful in the field of Bioinformatics, where k-mers are often used in the context of sequence assembly. Searching for patterns longer than k is expensive, but our auxiliary structures enable additional operations such as $extract$ to be performed efficiently using the k-BWT. Combined with our results of Chapter 4, we believe that k-BWT-based indexes can be constructed efficiently for large text collections, while providing functionality similar to that of a traditional BWT-based FM-Index.

Extending our work on context-bound text transformations such as the k-BWT, we investigated *variable depth* context-bound text transformations in Chapter 6. We defined a variable depth text transformation (v-BWT) as a derivative of the k-BWT, which sorts suffixes based on the number of suffixes in each context group instead of sorting all suffixes to depth k. We defined a threshold v, which describes the maximum size of each context group in the v-BWT. Thus each context group is sorted until each context group is smaller or equal to v. We showed that the transform can be: (1) constructed efficiently; (2) reversed storing no extra information; and (3) searched similar to the our results in Chapter 5 for the k-BWT. We defined the conditions under which pattern search in the v-BWT leads to a continuous range of suffixes in the v-BWT. Using these techniques we applied the v-BWT to the approximate pattern matching problem similar to an approach independently proposed by Navarro and Salmela [2009]. In our experimental evaluation we found that using the v-BWT is more efficient and scalable than the approach of Navarro and Salmela [2009] as no suffix tree needs to be constructed. We further showed that variable length substrings provide additional trade-offs when performing pattern partitioning for q-gram-based approximate pattern matching algorithms.

The second major aspect we focus on in this project is *designing* scalable succinct text indexes. We investigated improving the performance of succinct text indexes by applying common optimization techniques to the underlying building blocks of succinct text indexes. In Chapter 3 we provided a systematic evaluation of different low-level data structures and algorithms used to build succinct text indexes such as the FM-Index. First we showed improvements to basic bit-operations $rank_{64}$ and $select_{64}$ on computer words used to provide $rank$ and $select$ support over bitvectors. Next we proposed a cache-efficient uncompressed bitvector representation which provides $rank$ support close to the time required to access a single bit in the vector as the size of the bitvector increases. Similarly, we devised a fast, space-efficient $select$ structure which performs well in practice and can be constructed much faster than equivalent state-of-the-art $select$ data structures. In our experiments we showed that utilizing advanced CPU instruction sets such as SSE-4.2 and bit-parallelism can substantially improve both run-time performance and construction cost of succinct data structures. Similar to engineering uncompressed bitvectors, we successfully improved the performance of compressed bitvectors by optimizing techniques proposed by Navarro and Providel [2012]. Bitvectors supporting $rank$ and $select$ are used inside a *wavelet tree* to provide $rank$ and $select$ functionality over general sequences. Thus, moving up the hierarchy of succinct data structures, we found cache-efficient processing of wavelet trees and bit-parallel construction can increase the run-time performance of operations and construction cost of wavelet trees. Additionally, we found that the previously unexplored operating system features *hugepages* can have a considerable effect on the performance of succinct data structures. To evaluate our improvements to low-level succinct data structures we conducted an extensive empirical comparison of different succinct text indexes. To ensure our baselines were competitive, we recreated the reference empirical evaluation of succinct text indexes of Ferragina et al. [2008]. Without our optimizations, our text indexes perform similar to those publicly made available with the study of Ferragina et al. [2008]. When evaluating our improvements, we showed that especially for data sets much larger than those commonly used in the evaluation of succinct text indexes, our optimizations substantially improve the performance of our text indexes. We further found that on small data sets, improvements in run-time performance were smaller compared to large data sets. The compression effectiveness of our most space efficient succinct text index outperforms GZIP and is within 5% of XZ, the best of-the-shelf compression utility available, while still providing query functionality in the microsecond range over a 64 GB data set. Overall, using the engineering techniques discussed in Chapter 3, we provide the fastest or smallest — depending on the techniques used — FM-Indexes available today.

To substantiate our claim that we improved the scalability of succinct text indexes, we compared the performance of succinct text indexes, using improvements discussed in Chapter 3, to inverted in-

dexes in a standard Information Retrieval evaluation environment. We evaluated the performance of succinct text indexes in the context of top-ϕ ranked document retrieval — a classic problem solved efficiently by inverted indexes. We indexed standard IR text collections and measured the performance of both index types over top-ϕ bag-of-words queries. Most importantly, we found that our indexes can provide comparable run-time performance, at statistically equivalent effectiveness, to inverted indexes which use query processing techniques optimized to support top-ϕ bag-of-words efficiently. This is an important step towards a wider adoption of succinct text indexes as they can provide additional functionality which can be computationally expensive when using inverted indexes.

In this research project we proposed several techniques which can improve the scalability of succinct text indexes. We investigated alternative index types based on context-bound text transformations and engineering techniques applicable to succinct data structures in general. Our innovations, in the context of making succinct text indexes more scalable, provide interesting new avenues towards using succinct text indexes on a larger scale in areas such as Bioinformatics or Information Retrieval both in academia and in industry.

Bibliography

J. Abel. Post BWT stages of the Burrows-Wheeler compression algorithm. *Software - Practice and Experience (SPE)*, 40(9):751–777, 2010.

M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

A. Al-Maskari, M. Sanderson, and P. Clough. The relationship between IR effectiveness measures and user satisfaction. In *Proc. 30th ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 773–774, 2007.

S. Albers and S. Lauer. On List Update with Locality of Reference. In *Proc. of the 35th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 96–107, 2008.

S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

V. N. Anh and A. Moffat. Impact transformation: effective and efficient web retrieval. In *Proc. 25th ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 3–10, 2002.

V. N. Anh and A. Moffat. Inverted Index Compression Using Word-Aligned Binary Codes. *Information Retrieval*, 8:151–166, 2005.

V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proc. 29th ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 372–379, 2006.

A. Apostolico and S. Lonardi. Compression of biological sequences by greedy off-line textual substitution. In *Proc. of the 10th IEEE Data Compression Conference (DCC)*, pages 143–152, 2000.

R. Bachrach and R. El-Yaniv. Online List Accessing Algorithms and Their Applications: Recent Empirical Evidence. In *Proc. of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 53–62, 1997.

R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval - the concepts and technology behind search, Second edition*. Pearson Education Ltd., Harlow, England, 2011. ISBN 978-0-321-41691-9.

J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In *Proc. of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 390–399, 2002.

J. Barbay and C. Kenyon. Alternation and redundancy analysis of the intersection problem. *ACM Transactions on Algorithms*, 4(1), 2008.

J. Barbay, A. López-Ortiz, T. Lu, and A. Salinger. An experimental investigation of set intersection algorithms for text searching. *ACM Journal of Experimental Algorithmics*, 14, 2009.

J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich. Alphabet Partitioning for Compressed Rank/Select and Applications. In *Proc. 21st Annual International Symposium on Algorithms and Computation (ISAAC)*, pages 315–326, 2010.

L. Barroso, J. Dean, and U. Holzle. Web search for a planet: The Google cluster architecture. *Micro, IEEE*, 23(2):22–28, 2003.

M. J. Bauer, A. J. Cox, and G. Rosone. Lightweight BWT Construction for Very Large String Collections. In *Proc. of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 219–231, 2011.

D. Belazzougui and G. Navarro. Improved Compressed Indexes for Full-Text Document Retrieval. In *Proc. of the 18th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 386–397, 2011.

D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Monotone Minimal Perfect Hashing: Searching a Sorted Table with O(1) Accesses. In *Proc. of the 20th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 785–794, 2009.

J. Bentley and R. Sedgewick. Fast Algorithms for Sorting and Searching Strings. In *Proc. of the 8th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 360–369, 1997.

J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A Locally Adaptive Data Compression Scheme. *Communications of the ACM (CACM)*, 29(4):320–330, 1986.

T. Bingmann, J. Fischer, and V. Osipov. Inducing Suffix and LCP Arrays in External Memory. In *Proc. of the 15th Workshop on Algorithm Engineering and Experimentation (ALENEX)*, pages 88–103, 2013.

N. R. Brisaboa, R. Cánovas, F. Claude, M. A. Martínez-Prieto, and G. Navarro. Compressed String Dictionaries. In *Proc. of the 10th International Symposium on Experimental Algorithms (SEA)*, pages 136–147, 2011.

A. Z. Broder, D. Carmel, H. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. of the 12th International Conference on Information and Knowledge Management (CIKM)*, pages 426–434, 2003.

C. Buckley and A. F. Lewit. Impact transformation: effective and efficient web retrieval. In *Proc. 8th ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 97–110, 1985.

M. Burrows and D. Wheeler. A Block-sorting Lossless Data Compression Algorithm. Technical Report SRC-RR-124, Digital Equipment Corporation (DEC), USA, April 1994.

S. Büttcher, C. L. A. Clarke, and G. V. Cormack. *Information Retrieval: Implementing and evaluating search engines.* MIT Press, Cambridge, Massachusetts, 2010.

H. L. Chan, T. W. Lam, W. K. Sung, S. L. Tam, and S. S. Wong. Compressed indexes for approximate string matching. *Algorithmica*, 58(2):263–281, 2010.

X. Chen, S. Kwong, and M. Li. A compression algorithm for DNA sequences and its applications in genome comparison. In *Proc. of the 4th International Conference on Research in Computational Molecular Biology (RECOMB)*, page 107, 2000.

Y.-F. Chien, W.-K. Hon, R. Shah, and J. S. Vitter. Geometric Burrows-Wheeler Transform: Linking Range Searching and Text Indexing. In *Proc. of 18th Data Compression Conference (DCC)*, pages 439–448, 2001.

M. Christos. Wavelet trees: A survey. *Computer Science and Information Systems*, 9:585–625, 2012.

D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.

F. Claude and G. Navarro. Practical Rank/Select Queries over Arbitrary Sequences. In *Proc. of the 15th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 176–187, 2008.

F. Claude and G. Navarro. The Wavelet Matrix. In *Proc. of the 19th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 167–179, 2012a.

F. Claude and G. Navarro. Improved Grammar-Based Compressed Indexes. In *Proc. 19th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 180–192, 2012b.

W. B. Croft, D. Metzler, and T. Strohman. *Search Engines - Information Retrieval in Practice*. Pearson Education, 2009. ISBN 978-0-13-136489-9.

J. S. Culpepper and A. Moffat. Enhanced byte codes with restricted prefix properties. In *Proc. of the 12th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 1–12, 2005.

J. S. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *ACM Transactions on Information Systems (TOIS)*, 29(1), 2010.

J. S. Culpepper, G. Navarro, S. J. Puglisi, and A. Turpin. Top-*k* Ranked Document Search in General Text Databases. In *Proc. of the 18th Annual European Symposium on Algorithms (ESA)*, pages 194–205, 2010.

J. S. Culpepper, M. Yasukawa, and F. Scholer. Language independent ranked retrieval with NeWT. In *Proc. of the 16th Australasian Document Computing Symposium (ADCS)*, pages 18–25, December 2011.

M. Deo and S. Keely. Parallel suffix array and least common prefix for the GPU. In *Proc. of the 18th ACM Symposium on Principles and Practice of Parallel Programming (SIGPLAN)*, pages 197–206. ACM, 2013.

S. Deorowicz. Second step algorithms in the Burrows-Wheeler compression algorithm. *Software - Practice and Experience (SPE)*, 32(2):99–111, 2002.

P. Elias. Efficient Storage and Retrieval by Content and Address of Static Files. *J. ACM*, 21(2): 246–260, 1974.

P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

P. Fenwick. Block sorting text compression - final report. Technical Report 130, University of Auckland, April 1996.

P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *Proc. of the 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 390–398, 2000.

P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An Alphabet-Friendly FM-Index. In *Proc. of the 11th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 150–160, 2004.

P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *J. ACM*, 52(4):688–713, 2005.

P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and searching XML data via two zips. In *Proc. of the 33rd International Conference on World Wide Web (WWW)*, pages 751–760, 2006.

P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2), 2007.

P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13, 2008.

P. Ferragina, R. Giancarlo, and G. Manzini. The myriad virtues of wavelet trees. *Information and Computation*, 207:849–866, 2009.

P. Ferragina, T. Gagie, and G. Manzini. Lightweight Data Indexing and Compression in External Memory. In *Proc. of the 9th Symposium of Latin American Theoretical Informatics (LATIN)*, pages 697–710, 2010.

P. Ferragina, T. Gagie, and G. Manzini. Lightweight Data Indexing and Compression in External Memory. *Algorithmica*, 63(3):707–730, 2012.

J. Fischer. Optimal Succinctness for Range Minimum Queries. In *Proc. of the 9th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 158–169, 2010.

J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009.

A. Fog. Instruction tables, 2012. URL `http://www.agner.org/optimize/instruction_tables.pdf`.

M. Fontoura, V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Zien. Evaluation strategies for top-k queries over memory-resident inverted indexes. *Proc. of the VLDB Endowment*, 4(12):1213–1224, 2011.

T. Gagie and G. Manzini. Move-to-Front, Distance Coding, and Inversion Frequencies Revisited. In *Proc. of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 71–82, 2007.

T. Gagie, S. J. Puglisi, and A. Turpin. Range Quantile Queries: Another Virtue of Wavelet Trees. In *Proc. of the 16th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 1–6, 2009.

T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. A Faster Grammar-Based Self-index. In *Proc. 6th International Conference on Language and Automata Theory and Applications (LATA)*, pages 240–251, 2012a.

T. Gagie, G. Navarro, and S. Puglisi. New Algorithms on Wavelet Trees and Applications to Information Retrieval. *Theoretical Computer Science*, 426–427:25–41, 2012b.

T. Gagie, G. Navarro, and S. J. Puglisi. New Algorithms on Wavelet Trees and Applications to Information Retrieval. *Theoretical Computer Science*, 426:25–41, 2012c.

S. Gog. *Compressed suffix trees: Design, construction, and applications*. PhD thesis, University of Ulm, Germany, 2011.

S. Gog and E. Ohlebusch. Fast and Lightweight LCP-Array Construction Algorithms. In *Proc. of the 13th Workshop on Algorithm Engineering and Experimentation (ALENEX)*, pages 25–34, 2011.

A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. of the 17th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.

A. Golynski, R. Grossi, A. Gupta, R. Raman, and S. S. Rao. On the Size of Succinct Indices. In *Proc. of the 15th Annual European Symposium on Algorithms (ESA)*, volume 4698, pages 371–382, 2007.

R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical Implementation of Rank and Select Queries. In *Poster Proc. of 4th Workshop on Experimental and Efficient Algorithms (WEA)*, pages 27–38, 2005.

R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. of the 32nd Annual ACM Symposium on Theory of Computing (STOC)*, pages 397–406, 2000.

R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. of the 14th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

R. Grossi, A. Gupta, J. S. Vitter, and B. Xu. Wavelet Trees: From Theory to Practice. In *Proc. First International Conference on Data Compression, Communications and Processing (CCP)*, pages 210–221, 2011.

D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, New York, USA, 1997. ISBN 0-521-58519-8.

D. Hawking. Overview of the TREC-9 web track. In *TREC-8*, pages 87–102, 1999.

D. Hawking, N. Craswell, P. Bailey, and K. Griffiths. Measuring Search Engine Quality. *Information Retrieval*, 4(1):33–59, 2001.

S. Hazelhurst and Z. Liptk. KABOOM! A new suffix array based algorithm for clustering expression data. *Bioinformatics*, 27(24):3348–3355, 2011.

W.-K. Hon, R. Shah, and J. S. Vitter. Space-Efficient Framework for Top-k String Retrieval Problems. In *Proc. of the 50th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 713–722, 2009.

C. Hoobin, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv Factorization for Efficient Storage and Retrieval of Web Collections. *Proc. of the VLDB Endowment*, 5(3):265–273, 2011.

R. N. Horspool. Practical Fast Searching in Strings. *Software - Practice and Experience (SPE)*, 10 (6):501–506, 1980.

T. Hu and A. Tucker. Optimal Computer Search Trees and Variable-Length Alphabetical Codes. *SIAM Journal on Applied Mathematics*, 21(4):514–532, 1971.

D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proc. of the Institute of Radio Engineers (IRE)*, 40(9):1098–1101, 1952.

K. Inagaki, Y. Tomizawa, and H. Yokoo. Novel and Generalized Sort-Based Transform for Lossless Data Compression. In *Proc. of the 16th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 102–113, 2009.

H. Itoh and H. Tanaka. An Efficient Method for in Memory Construction of Suffix Arrays. In *Proc. of the 6th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 81–88, 1999.

G. Jacobsen. *Succinct static data structures*. PhD thesis, Carnegie-Mellon University, 1988.

G. Jacobsen. Space-efficient static trees and graphs. In *Proc. of the 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

H. Kaplan, S. Landau, and E. Verbin. A simpler analysis of Burrows-Wheeler-based compression. *Theoretical Computer Science*, 387(3):220–235, 2007.

J. Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. *Theoretical Computer Science*, 387(3):249–257, 2007.

J. Kärkkäinen and S. J. Puglisi. Medium-Space Algorithms for Inverse BWT. In *Proc. of 18th Annual European Symposium of Algorithms, Communications and Processing (ESA)*, pages 451–462, 2010.

J. Kärkkäinen and S. J. Puglisi. Cache Friendly Burrows-Wheeler Inversion. In *Proc. of 1st Int. Conference on Data Compression, Communications and Processing (CCP)*, pages 38–42, 2011a.

J. Kärkkäinen and S. J. Puglisi. Fixed Block Compression Boosting in FM-Indexes. In *Proc. of the 18th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 174–184, 2011b.

J. Kärkkäinen and T. Rantala. Engineering Radix Sort for Strings. In *Proc. of the 15th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 3–14, 2008.

J. Kärkkäinen and P. Sanders. Simple Linear Work Suffix Array Construction. In *Proc. of the 30th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 943–955, 2003.

J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd South American Workshop on String Processing (WSP)*, pages 141–155, 1996.

J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Slashing the Time for BWT Inversion. In *Proc. of 22nd Data Compression Conference (DCC)*, pages 439–448, 2012.

T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In *Proc. of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 181–192, 2001.

D. Kim, J. Na, J. Kim, and K. Park. Efficient Implementation of Rank and Select Functions for Succinct Representation. In *Proc. of 4th Workshop on Experimental and Efficient Algorithms (WEA)*, pages 315–327, 2005.

D. Knuth. *The Art of Computer Programming, Volume 3, Sorting and Searching*. Addison-Wesley, 1998. ISBN 0-201-89685-0.

D. Knuth. *The Art of Computer Programming, Volume 4A, Combinatorial Algorithms, Part 1*. Addison-Wesley, 2011. ISBN 0-201-03804-8.

D. Knuth, J. Morris, and V. Pratt. Fast Pattern Matching in Strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, 2005.

S. Kreft and G. Navarro. On Compressing and Indexing Repetitive Sequences. *Theoretical Computer Science*, page to appear, 2012.

K. Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4): 377–439, 1992.

F. Kulla and P. Sanders. Scalable parallel suffix array construction. *Parallel Computing*, 33(9): 605–612, 2007.

S. Kurtz. Reducing the space requirement of suffix trees. *Software - Practice and Experience (SPE)*, 29(13):1149–1171, 1999.

B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

N. Larsson and A. Moffat. Off-line dictionary-based compression. *Proc. of the IEEE*, 88(11):1722–1732, 2000.

H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

V. Mäkinen and G. Navarro. New Search Algorithms and Time/Space Tradeoffs for Succinct Suffix Arrays. Technical Report C-2004-20, Department of Computer Science, University of Helsinki, Finland, April 2004.

V. Mäkinen and G. Navarro. Succinct Suffix Arrays Based on Run-Length Encoding. In *Proc. of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 45–56, 2005.

V. Mäkinen and G. Navarro. Implicit Compression Boosting with Applications to Self-indexing. In *Proc. of the 14th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 229–241, 2007.

U. Manber and E. W. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

M. A. Maniscalco and S. J. Puglisi. Faster lightweight suffix array construction. In *Proc. of 17th Australasian Workshop on Combinatorial Algorithms (AWOCA)*, pages 16–29, 2006.

G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.

S. Maruyama, M. Nakahara, N. Kishiue, and H. Sakamoto. ESP-index: A compressed index based on edit-sensitive parsing. *Journal of Discrete Algorithms*, 18(0):100–112, 2013.

P. E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org, Corvallis, OR, USA, 2011. Available: `http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html`.

D. Metzler and W. B. Croft. A Markov random field model for term dependencies. In *Proc. 28th ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 472–479. ACM, 2005.

A. Moffat and V. N. Anh. Binary codes for non-uniform sources. In *Proc. of 15th Data Compression Conference (DCC)*, pages 133–142, 2005.

A. Moffat and R. Y. K. Isal. Word-based text compression using the Burrows-Wheeler transform. *Information Processing and Management*, 41(5):1175–1192, 2005.

A. Moffat and J. Zobel. Self-Indexing Inverted Files for Fast Text Retrieval. *ACM Transactions on Information Systems (TOIS)*, 14(4):349–379, 1996.

Y. Mori. SAIS - An Implementation of the Induced Sorting Algorithm, 2012. URL `https://sites.google.com/site/yuta256/sais`.

D. R. Morrison. PATRICIA — Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15(4):514–534, Oct. 1968. ISSN 0004-5411.

I. Munro. Tables. In *Proc. of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 37–42, 1996.

J. Munro, V. Raman, and S. Rao. Space Efficient Suffix Trees. *Journal of Algorithms*, 39(2):205–222, 2001.

J. I. Munro and V. Raman. Succinct Representation of Balanced Parentheses, Static Trees and Planar Graphs. In *Proc. of the 38th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 118–126, 1997.

S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 657–666, 2002.

G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

G. Navarro. Wavelet Trees for All. In *Proc. of the 23nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 2–26, 2012.

G. Navarro and R. A. Baeza-Yates. A Practical q-gram Index for Text Retrieval Allowing Errors. *CLEI Electron. J.*, 1(2), 1998.

G. Navarro and V. Mäkinen. Two Efficient Algorithms for Linear Time Suffix Array Construction. *ACM Computing Surveys*, 39(1):Article 2, 2007.

G. Navarro and Y. Nekrich. Top-k document retrieval in optimal time and linear space. In *Proc. of the 23rd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1066–1077, 2012.

G. Navarro and E. Providel. Fast, Small, Simple Rank/Select on Bitmaps. In *Proc. of the 11th International Symposium on Experimental Algorithms (SEA)*, pages 295–306, 2012.

G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002. ISBN 0-521-81307-7. 280 pages.

G. Navarro and K. Sadakane. Fully-Functional Static and Dynamic Succinct Trees. *ACM Transactions on Algorithms (TALG)*, page to appear, 2013.

G. Navarro and L. Salmela. Indexing Variable Length Substrings for Exact and Approximate Matching. In *Proc. of the 16th International Conference on StringProcessing and Information Retrieval (SPIRE)*, pages 214–221, 2009.

G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio. A binary n-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001a.

G. Navarro, R. A. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing Methods for Approximate String Matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001b.

G. Navarro, S. Puglisi, and D. Valenzuela. Practical Compressed Document Retrieval. In *Proc. of the 10th International Symposium on Experimental Algorithms (SEA)*, pages 193–205, 2011.

J.-Y. Nie, J. Gao, J. Zhang, and M. Zhou. On the use of words and n-grams for Chinese information retrieval. In *Proc. 15th ACM international workshop on on Information retrieval with Asian languages (IRAL)*, pages 141–148, 2000.

G. Nong and S. Zhang. Efficient algorithms for the inverse sort transform. *IEEE Transactions on Computers*, 56(11):1564–1574, 2007.

G. Nong, S. Zhang, and W. H. Chan. Computing Inverse ST in Linear Complexity. In *Proc. of the 19th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 178–190, 2008.

G. Nong, S. Zhang, and W. H. Chan. Two Efficient Algorithms for Linear Time Suffix Array Construction. *IEEE Trans. Computers*, 60(10):1471–1484, 2011.

E. Ohlebusch, J. Fischer, and S. Gog. A Compressed Enhanced Suffix Array Supporting Fast String Matching. In *Proc. of the 16th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 51–62, 2009.

E. Ohlebusch, J. Fischer, and S. Gog. CST++. In *Proc. of the 17th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 322–333, 2010.

D. Okanohara and K. Sadakane. Practical Entropy-Compressed Rank/Select Dictionary. In *Proc. of the 9th Workshop on Algorithm Engineering and Experimentation (ALENEX)*, 2007.

R. Pagh. Low Redundancy in Static Dictionaries with O(1) Worst Case Lookup Time. In *Proc. of the 26th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 595–604, 1999.

M. Persin. Document filtering for fast ranking. In *Proc. 17th ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 339–348, 1994.

S. J. Puglisi, W. F. Smyth, and A. Turpin. Inverted files versus suffix arrays for locating patterns in primary memory. In *Proc. of the 13th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 122–133, 2006.

S. J. Puglisi, W. F. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2), 2007.

M. Pătraşcu. Succincter. In *Proc. of the 49th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 305–313, 2008.

R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

S. E. Robertson, S. Walker, S. Jones, and M. Hancock-Beaulieu. Okapi at TREC-3. In *Proc. of the 3rd Text REtrieval Conference (TREC)*, 1994a.

S. E. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, and M. Gatford. Okapi at TREC-3. In *TREC-3*, 1994b.

L. Russo, G. Navarro, A. Oliveira, and P. Morales. Approximate String Matching with Compressed Indexes. *Algorithms*, 2(3):1105–1136, 2009.

K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proc. of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 225–232, 2002.

K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007a.

K. Sadakane. Compressed Suffix Trees with Full Functionality. *Theory of Computing Systems*, 41 (4):589–607, 2007b.

M. Salson, T. Lecroq, M. Léonard, and L. Mouchard. Dynamic Burrows-Wheeler Transform. In *Proc. of the Prague Stringology Conference 2008*, pages 13–25, 2008.

M. Schindler. A fast block-sorting algorithm for lossless data compression. (poster). In *Proc. of 7th Data Compression Conference (DCC)*, 1997.

M. Schindler. A fast renormalization for arithmetic coding. In J. A. Storer and M. Cohn, editors, *Proc. of the 8th IEEE Data Compression Conference (DCC)*, page 572, Los Alamitos, California, March 1998. IEEE Computer Society Press.

F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. 25th ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 222–229, 2002.

P. H. Sellers. The Theory and Computation of Evolutionary Distances: Pattern Recognition. *Journal of Algorithms*, 1(4):359–373, 1980.

J. Seward. On the Performance of BWT Sorting Algorithms. In *Proc. of 10th Data Compression Conference (DCC)*, pages 173–182, 2000.

J. Seward. Space-time tradeoffs in the inverse B-W transform. In *Proc. of 11th Data Compression Conference (DCC)*, pages 439–448, 2001.

J. Sirén. Compressed Suffix Arrays for Massive Data. In *Proc. of the 16th International Conference on StringProcessing and Information Retrieval (SPIRE)*, pages 63–74, 2009.

D. Sleator and R. E. Tarjan. Amortized efficiency of list update paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

A. Suciu, P. Cobarzan, and K. Marton. The never ending problem of counting bits efficiently. In *Proc. of the 10th Roedunet International Conference (RoEduNet)*, pages 1–4, 2011.

E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In *Proc. of the 7th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 50–63, 1996.

H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):3348–3355, 2005. URL `http://www.gotw.ca/publications/concurrency-ddj.htm`.

A. Trotman. Compressing Inverted Files. *Information Retrieval*, 6(1):5–19, 2003.

A. Turpin, Y. Tsegay, D. Hawking, and H. E. Williams. Fast Generation of Result Snippets in Web Search. In *Proc. 30th ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 127–134, 2007.

H. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Information Processing and Management*, 31(6):831–850, 1995.

E. Ukkonen. On-Line Construction of Suffix Trees. *Algorithmica*, 14(3):249–260, 1995.

J. Ullman. A binary n-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words. *The Computer Journal*, 10:141–147, 1977.

N. Välimäki and V. Mäkinen. Space-Efficient Algorithms for Document Retrieval. In *Proc. of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 205–215, 2007.

S. Vigna. Broadword Implementation of Rank/Select Queries. In *Proc. of 7th Workshop on Experimental Algorithms (WEA)*, pages 154–168, 2008.

B. Vo and G. S. Manku. RadixZip: Linear-Time Compression of Token Streams. In *Proc. of the 33rd International Conference on Very Large Data Bases (VLDB)*, pages 1162–1172, 2007.

E. M. Voorhees and D. K. Harman. Overview of the Eighth Text REtrieval Conference (TREC-8). In *TREC-8*, pages 1–24, 1999.

H. S. Warren. *Hacker's Delight*. Addison-Wesley, 2003. ISBN 0-201-91465-4.

P. Weiner. Linear Pattern Matching Algorithms. In *Proc. of the 14th Annual Symposium on Switching and Automata Theory (SWAT now FOCS)*, pages 1–11, 1973.

H. E. Williams, J. Zobel, and P. Anderson. Whats next? Index structures for efficient phrase querying. In *Proc. of the 1th Australasian Database Conference (ADC)*, pages 141–152. Australian Computer Society, 1999.

I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–541, June 1986.

I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999. ISBN 1-55860-570-3.

S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10): 83–91, 1992.

H. Yan, S. Ding, and T. Suel. Compressing term positions in web indexes. In *Proc. 29th ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 147–154, 2009.

H. Yokoo. Notes on block-sorting data compression. *Electronics and Communications in Japan, Part 3*, 82(6):18–25, 1999.

H. Yokoo. Extension and Faster Implementation of the GRP Transform for Lossless Compression. In *Proc. of the 21st Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 338–347, 2010.

J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2), 2006.

J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490, 1998.