The Compressed Permuterm Index

PAOLO FERRAGINA and ROSSANO VENTURINI
University of Pisa, Italy

The *Permuterm index* (Garfield, 1976) is a time-efficient and elegant solution to the string dictionary problem in which pattern queries may possibly include one wild-card symbol (called, *Tolerant Retrieval* problem). Unfortunately the Permuterm index is space inefficient because it quadruples the dictionary size. In this paper we propose the *Compressed* Permuterm Index which solves the Tolerant Retrieval problem in time proportional to the length of the searched pattern, and space close to the k-th order empirical entropy of the indexed dictionary. We also design a *dynamic* version of this index which allows to efficiently manage insertion in, and deletion from, the dictionary of individual strings.

The result is based on a simple variant of the Burrows-Wheeler Transform, defined on a dictionary of strings of variable length, that allows to efficiently solve the Tolerant Retrieval problem via known (dynamic) compressed indexes [Navarro and Mäkinen 2007]. We will complement our theoretical study with a significant set of experiments which show that the Compressed Permuterm Index supports fast queries within a space occupancy that is close to the one achievable by compressing the string dictionary via gzip or bzip2. This improves known approaches based on Front-Coding [Witten et al. 1999] by more than 50% in absolute space occupancy, still guaranteeing comparable query time.

Categories and Subject Descriptors: D.4.2 [**Operating System**]: Storage Management; E.1 [**Data Structures**]: Arrays, Tables; E.4 [**Coding and Information Theory**]: Data compaction and compression; E.5 [**Files**]: Sorting/searching; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical algorithms and Problems—*Pattern matching*; H.3 [**Information Storage and Retrieval**]: Content Analysis and Indexing, Information Storage, Information Search and Retrieval.

General Terms: Algorithms, Design, Theory.

Additional Key Words and Phrases: Burrows-Wheeler transform, compressed index, indexing data structure, Permuterm, string dictionary

1. INTRODUCTION

String processing and searching tasks are at the core of modern Web search, information retrieval and data mining applications. Most of such tasks boil down to some basic algorithmic primitives which involve a large dictionary of strings having variable length. Typical examples include: pattern matching (exact, approximate, with wild-cards,...), the ranking of a string in a sorted dictionary, or the selection of the i-th string from it. While it is easy to imagine uses of pattern matching primi-

The work is an extended version of a paper appeared in the *Procs of ACM SIGIR 2007*. It has been partially supported by Yahoo! Research and by Italian MIUR Italy-Israel FIRB Project "Pattern Discovery Algorithms in Discrete Structures, with Applications to Bioinformatics".

tives in real applications, such as search engines and text mining tools, rank/select operations appear uncommon. However they are quite often used (probably, unconsciously!) by programmers to replace long strings with unique `IDs` which are easier and faster to be processed and compressed. In this context ranking a string means mapping it to its unique `ID`, whereas selecting the i-th string means retrieving it from its `ID` (i.e. its ranked position i).

As strings are getting longer and longer, and dictionaries of strings are getting larger and larger, it becomes crucial to devise implementations for the above primitives which are fast and work in compressed space. This is the topic of the present paper that actually addresses the design of compressed data structures for the so called *tolerant retrieval* problem, defined as follows [Manning et al. 2008]. Let $\mathcal{D}$ be a sorted dictionary of m strings having total length n and drawn from an arbitrary alphabet Σ. The *tolerant retrieval* problem consists of preprocessing $\mathcal{D}$ in order to efficiently support the following WildCard(P) query operation: *search for the strings in $\mathcal{D}$ which match the pattern $P \in (\Sigma \cup \{*\})^+$*. Symbol $*$ is the so called *wild-card* symbol, and matches any substring of Σ^*. In principle, the pattern P might contain several occurrences of $*$; however, for practical reasons, it is common to restrict the attention to the following significant cases:

—Membership query determines whether a pattern $P \in \Sigma^+$ occurs in $\mathcal{D}$. Here P does not include wild-cards.

—Prefix query determines all strings in $\mathcal{D}$ which are prefixed by string α. Here $P = \alpha*$ with $\alpha \in \Sigma^+$.

—Suffix query determines all strings in $\mathcal{D}$ which are suffixed by string β. Here $P = *\beta$ with $\beta \in \Sigma^+$.

—Substring query determines all strings in $\mathcal{D}$ which have γ as a substring. Here $P = *\gamma*$ with $\gamma \in \Sigma^+$.

—PrefixSuffix query is the most sophisticated one and asks for all strings in $\mathcal{D}$ that are prefixed by α and suffixed by β. Here $P = \alpha * \beta$ with $\alpha, \beta \in \Sigma^+$.

In this paper we extend the tolerant retrieval problem to include the following two basic primitives:

—Rank(P) computes the rank of string $P \in \Sigma^+$ within the (sorted) dictionary $\mathcal{D}$.

—Select(i) retrieves the i-th string of the (sorted) dictionary $\mathcal{D}$.

There are two classical approaches to string searching: Hashing and Tries [Baeza-Yates and Ribeiro-Neto 1999]. Hashing supports only the exact Membership query; its more sophisticated variant called *minimal ordered perfect* hashing [Witten et al. 1999] supports also the Rank operation but only on strings of $\mathcal{D}$. All other queries need however the inefficient scan of the whole dictionary!

Tries are more powerful in searching than hashing, but they introduce extra space and fail to provide an efficient solution to the PrefixSuffix query. In fact, the search for $P = \alpha * \beta$ needs to visit the subtrie descending from the trie-path labeled α, in order to find the strings that are suffixed by β. Such a brute-force visit may cost $\Theta(|\mathcal{D}|)$ time independently of the number of query answers (cfr [Baeza-Yates and Gonnet 1996]). We can circumvent this limitation by using the sophisticated approach proposed in [Ferragina et al. 2003] which builds two tries,

one storing the strings of $\mathcal{D}$ and the other storing their reversals, and then *reduce* the PrefixSuffix query to a geometric 2D-range query, which is eventually solved via a proper efficient geometric data structure in $O(|\alpha| + |\beta| + \texttt{polylog}(n))$ time. The overall space occupancy would be $\Theta(n \log n)$ bits,[1] with a large constant hidden in the big-O notation due to the presence of the two tries and the geometric data structure.

Recently Manning et al. [2008] resorted the *Permuterm index* of Garfield [1976] as a time-efficient and elegant solution to the tolerant retrieval problem above. The idea is to take every string $s \in \mathcal{D}$, append a special symbol \$, and then consider all the cyclic rotations of $s\$$. The dictionary of all *rotated* strings is called the *permuterm dictionary*, and is then indexed via any data structure that supports prefix searches, e.g. the trie. The key to solve the PrefixSuffix query is to rotate the query string $\alpha * \beta\$$ so that the wild-card symbol appears at the end, namely $\beta\$\alpha*$. Finally, it suffices to perform a prefix-query for $\beta\$\alpha$ over the permuterm dictionary. As a result, the Permuterm index allows to *reduce any query of the Tolerant Retrieval problem on the dictionary $\mathcal{D}$ to a prefix query over its permuterm dictionary.* The limitation of this elegant approach relies in its space occupancy, as "its dictionary becomes quite large, including as it does all rotations of each term." [Manning et al. 2008]. In practice, one memory word per rotated string (and thus 4 bytes per symbol) is needed to index it, for a total of $\Omega(n \log n)$ bits.

In this paper we propose the *Compressed* Permuterm Index which solves the tolerant retrieval problem in time proportional to the length of the queried string P, and space close to the k-th order empirical entropy of the dictionary $\mathcal{D}$ (see Section 2 for definitions). The time complexity matches the one achieved by the (uncompressed) Permuterm index. The space complexity approaches the information-theoretic lower bound to the output size of any compressor that encodes each symbol of a string with a code that depends on the symbol itself and on the k immediately preceding symbols. Compressors achieving performance related to the k-th order empirical entropy of a text are the well-known gzip[2], bzip2[3] and ppmdi[4]. In addition, we devise a *dynamic* Compressed Permuterm Index that is able to maintain the dictionary $\mathcal{D}$ under insertions and deletions of an individual string s in $O(|s|(1 + \log|\Sigma|/\log\log n)\log n)$ time. All query operations are slowed down by a multiplicative factor of at most $O((1 + \log|\Sigma|/\log\log n)\log n)$. The space occupancy is still close to the k-th order empirical entropy of the dictionary $\mathcal{D}$.

Our result is based on a variant of the Burrows-Wheeler Transform here extended to work on a dictionary of strings of variable length. We prove new properties of such BWT, and show that known (dynamic) compressed indexes [Navarro and Mäkinen 2007] may be easily adapted to solve efficiently the (dynamic) Tolerant Retrieval problem.

We finally complement our theoretical study with a significant set of experiments over large dictionaries of URLs, hosts and terms, and compare our Compressed Per-

[1]Throughout this paper we assume that all logarithms are taken to the base 2, whenever not explicitly indicated, and we assume $0 \log 0 = 0$.
[2]Available at `http://www.gzip.org`
[3]Available at `http://www.bzip.org`
[4]Available at `http://pizzachili.di.unipi.it/utils`

muterm index against some classical approaches to the Tolerant Retrieval problem mentioned in [Manning et al. 2008; Witten et al. 1999] such as tries and front-coded dictionaries. Experiments will show that tries are fast but much space consuming; conversely our compressed permuterm index allows to trade query time by space occupancy, resulting as fast as Front-Coding in searching the dictionary but more than 50% smaller in space occupancy— thus being close to gzip, bzip2 and ppmdi. This way the compressed permuterm index offers a plethora of solutions for the Tolerant Retrieval problem which may well adapt to different applicative scenarios.

2. BACKGROUND

Let $T[1,n]$ be a string drawn from the alphabet $\Sigma = \{a_1, \ldots, a_h\}$. For each $a_i \in \Sigma$, we let n_i be the number of occurrences of a_i in T. The zero-th order *empirical* entropy of T is defined as:

$$H_0(T) = \frac{1}{|T|} \sum_{i=1}^{h} n_i \log \frac{n}{n_i} \tag{1}$$

Note that $|T|H_0(T)$ provides an information-theoretic lower bound to the output size of any compressor that encodes each symbol of T with a fixed code [Witten et al. 1999].

For any string w of length k, we denote by w_T the string of single symbols following the occurrences of w in T, taken from left to right. For example, if $T = \texttt{mississippi}$ and $w = \texttt{si}$, we have $w_T = \texttt{sp}$ since the two occurrences of `si` in T are followed by the symbols `s` and `p`, respectively. The k-th order *empirical* entropy of T is defined as:

$$H_k(T) = \frac{1}{|T|} \sum_{w \in \Sigma^k} |w_T| \, H_0(w_T). \tag{2}$$

We have $H_k(T) \geq H_{k+1}(T)$ for any $k \geq 0$. As usual in data compression [Manzini 2001], we will adopt $|T|H_k(T)$ as an information-theoretic lower bound to the output size of any compressor that encodes each symbol of T with a code that depends on the symbol itself and on the k immediately preceding symbols.

In 1994, Burrows and Wheeler introduced a new compression algorithm based on a reversible transformation, now called the *Burrows-Wheeler Transform* (BWT from now on). The BWT transforms the input string T into a new string that is easier to compress. The BWT of T, hereafter denoted by $\texttt{bwt}(T)$, consists of three basic steps (see Figure 1):

(1) append at the end of T a special symbol \$ smaller than any other symbol of Σ;
(2) form a *conceptual* matrix $\mathcal{M}(T)$ whose rows are the cyclic rotations of string $T\$$ in lexicographic order;
(3) construct string L by taking the last column of the sorted matrix $\mathcal{M}(T)$. It is $\texttt{bwt}(T) = L$.

Every column of $\mathcal{M}(T)$, hence also the transformed string L, is a permutation of $T\$$. In particular the first column of $\mathcal{M}(T)$, call it F, is obtained by lexicographically sorting the symbols of $T\$$ (or, equally, the symbols of L). Note that when we

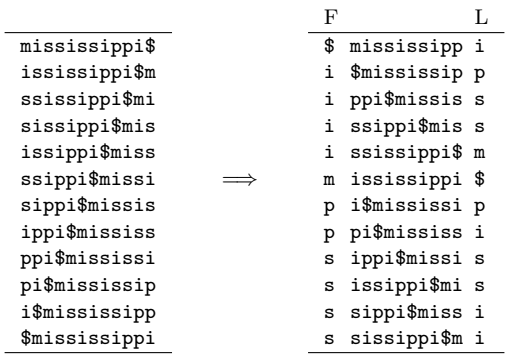

Fig. 1. Example of Burrows-Wheeler transform for the string T = `mississippi`. The matrix on the right has the rows sorted in lexicographic order. The output of the BWT is the last column L = `ipssm$pissii`.

sort the rows of $\mathcal{M}(T)$ we are essentially sorting the suffixes of T because of the presence of the special symbol \$. This shows that: (1) there is a strong relation between $\mathcal{M}(T)$ and the *suffix array* data structure built on T; (2) symbols following the same substring (*context*) in T are grouped together in L, thus giving raise to clusters of nearly identical symbols. Property 1 is crucial for designing compressed indexes (see e.g. [Navarro and Mäkinen 2007]), Property 2 is the key for designing modern data compressors (see e.g. [Manzini 2001; Ferragina et al. 2005]).

For our purposes, we hereafter concentrate on compressed indexes. They efficiently support the search of any (fully-specified) pattern $Q[1,q]$ as a *substring* of the indexed string $T[1,n]$. Two properties are crucial for their design [Burrows and Wheeler 1994]:

(a) Given the cyclic rotation of rows in $\mathcal{M}(T)$, $L[i]$ *precedes* $F[i]$ in the original string T.

(b) For any $c \in \Sigma$, the ℓ-th occurrence of c in F and the ℓ-th occurrence of c in L correspond to the *same* symbol of string T.

As an example, the 3rd `s` in L lies onto the row which starts with `sippi$` and, correctly, the 3rd `s` in F lies onto the row which starts with `ssippi$`. That symbol `s` is $T[6]$.

In order to map symbols in L to their corresponding symbols in F, Ferragina and Manzini [2005] introduced the following function:

$$LF(i) = C[L[i]] + \texttt{rank}_{L[i]}(L, i)$$

where $C[c]$ counts the number of symbols smaller than c in the whole string L, and $\texttt{rank}_c(L,i)$ counts the occurrences of c in the prefix $L[1,i]$. Given Property (b) and the alphabetic ordering of F, it is not difficult to see that symbol $L[i]$ corresponds to symbol $F[LF(i)]$. For example in Figure 1 we have $LF(9) = C[\texttt{s}] + rank_s(L, 9) = 8 + 3 = 11$ and, in fact, both $L[9]$ and $F[11]$ correspond to the symbol $T[6]$.

Array C is small and occupies $O(|\Sigma| \log n)$ bits, the implementation of function $LF(\cdot)$ is more sophisticated and boils down to the design of compressed data structures for supporting RANK queries over strings. The literature offers now many

Algorithm Backward_search($Q[1,q]$)

(1) $i = q$, $c = Q[q]$, First $= C[c] + 1$, Last $= C[c+1]$;
(2) **while** ((First $\leq$ Last) **and** $(i \geq 2)$) **do**
(3) $\quad c = Q[i-1]$;
(4) $\quad$ First $= C[c] + \texttt{rank}_c(L,$ First $- 1) + 1$;
(5) $\quad$ Last $= C[c] + \texttt{rank}_c(L,$ Last$)$;
(6) $\quad i = i - 1$;
(7) **if** (Last $<$ First) **then return** "no rows prefixed by Q" **else return** [First, Last].

Fig. 2. The algorithm to find the range [First, Last] of rows of $\mathcal{M}(T)$ prefixed by $Q[1,q]$.

theoretical and practical solutions for this problem (see e.g. [Navarro and Mäkinen 2007; Barbay et al. 2007] and references therein). We summarize the ones we use next, below.

LEMMA 2.1. *Let $T[1,n]$ be a string over alphabet Σ and let $L = \texttt{bwt}(T)$ be its BW-transform.*

(1) For $|\Sigma| = O(\texttt{polylog}(n))$, there exists a data structure which supports rank *queries and the retrieval of any symbol of L in constant time, by using $nH_k(T) + o(n)$ bits of space, for any $k \leq \alpha \log_{|\Sigma|} n$ and $0 < \alpha < 1$ [Ferragina et al. 2007, Theorem 5].*

(2) For general Σ, there exists a data structure which supports rank *queries and the retrieval of any symbol of L in $O(\log\log|\Sigma|)$ time, by using $nH_k(T) + n \cdot o(\log|\Sigma|)$ bits of space, for any $k \leq \alpha \log_{|\Sigma|} n$ and $0 < \alpha < 1$ [Barbay et al. 2007, Theorem 4.2].*

Given Property (a) and the definition of LF, it is easy to see that $L[i]$ (which is equal to $F[LF(i)]$) is preceded by $L[LF(i)]$, and thus the iterated application of LF allows to move *backward* over the text T. Of course, we can compute T from L by moving backward from symbol $L[1] = T[n]$.

Ferragina and Manzini [2005] made one step further by showing that data structures for supporting RANK queries on the string L are enough to search for an arbitrary pattern $Q[1,q]$ as a substring of the indexed text T. The resulting search procedure is now called *backward search* and is illustrated in Figure 2. It works in q phases, each preserving the invariant: *At the end of the i-th phase,* [First, Last] *is the range of contiguous rows in $\mathcal{M}(T)$ which are prefixed by $Q[i,q]$.* Backward_search starts with $i = q$ so that First and Last are determined via the array C (step 1). Ferragina and Manzini proved that the pseudo-code in Figure 2 maintains the invariant above for all phases, so [First, Last] delimits at the end the rows prefixed by Q (if any).

By plugging Lemma 2.1 into Backward_search, the authors of [Ferragina et al. 2007; Barbay et al. 2007] obtained:

THEOREM 2.2. *Given a text $T[1,n]$ drawn from an alphabet Σ, there exists a compressed index that takes $q \times t_{\texttt{rank}}$ time to support* Backward_search$(Q[1,q])$*, where $t_{\texttt{rank}}$ is the time cost of a single* rank *operation over $L = \texttt{bwt}(T)$. The space usage*

is bounded by $nH_k(T) + l_{space}$ *bits, for any* $k \leq \alpha \log_{|\Sigma|} n$ *and* $0 < \alpha < 1$*, where* l_{space} *is* $o(n)$ *when* $|\Sigma| = O(polylog(n))$ *and* $n \cdot o(\log |\Sigma|)$ *otherwise.* □

Notice that compressed indexes support also other operations, like locate and display of pattern occurrences, which are slower than Backward_search in that they require `polylog(n)` time per occurrence [Navarro and Mäkinen 2007]. We do not go into further details on these operations because one positive feature of our compressed permuterm index is that it will not need these (sophisticated) data structures, and thus it will not incur in this `polylog`-slowdown.

3. COMPRESSED PERMUTERM INDEX

The way in which the Permuterm dictionary is computed, immediately suggests that there *should be* a relation between the BWT and the Permuterm dictionary of the string set $\mathcal{D}$. In both cases we talk about *cyclic rotations* of strings, but in the former we refer to just one string, whereas in the latter we refer to a dictionary of strings of possibly different lengths. The notion of BWT for a set of strings has been considered in [Mantaci et al. 2005] for the purpose of string compression and comparison. Here, we are interested in the compressed *indexing* of the string dictionary $\mathcal{D}$, which introduces more challenges. Surprisingly enough the solution we propose is novel, simple, and efficient in time and space; furthermore, it admits an effective dynamization.

3.1 A simple, but inefficient solution

Let $\mathcal{D} = \{s_1, s_2, \ldots, s_m\}$ be the lexicographically sorted dictionary of strings to be indexed. Let \$ (resp. #) be a symbol smaller (resp. larger) than any other symbol of Σ. We consider the *doubled* strings $\widehat{s}_i = s_i \$ s_i$. It is easy to note that any pattern searched by WildCard(P) matches s_i if, and only if, the rotation of P mentioned in Section 1 is a substring of $\widehat{s}_i$. For example, the query PrefixSuffix($\alpha * \beta$) matches s_i iff the rotated string $\beta\$\alpha$ occurs as a substring of $\widehat{s}_i$.

Consequently, the simplest approach to solve the Tolerant Retrieval problem with compressed indexes seems to boil down to the indexing of the string $\widehat{\mathcal{S}}_{\mathcal{D}} = \#\widehat{s}_1\#\widehat{s}_2 \cdots \#\widehat{s}_m\#$ by means of the data structure of Theorem 2.2. Unfortunately, this approach suffers of subtle inefficiencies in the indexing and searching steps. To see them, let us "compare" string $\widehat{\mathcal{S}}_{\mathcal{D}}$ against string $\mathcal{S}_{\mathcal{D}} = \$s_1\$s_2\$ \ldots \$s_{m-1}\$s_m\$\#$, which is a *serialization* of the dictionary $\mathcal{D}$ (and it will be at the core of our approach, see below). We note that the "duplication" of s_i within $\widehat{s}_i$: (1) doubles the string to be indexed, because $|\widehat{\mathcal{S}}_{\mathcal{D}}| = 2|\mathcal{S}_{\mathcal{D}}| - 1$; and (2) doubles the space bound of compressed indexes evaluated in Theorem 2.2, because $|\widehat{\mathcal{S}}_{\mathcal{D}}| H_k(\widehat{\mathcal{S}}_{\mathcal{D}}) \cong 2|\mathcal{S}_{\mathcal{D}}| H_k(\mathcal{S}_{\mathcal{D}}) \pm m(k \log |\Sigma| + 2)$, where the second term comes from the presence of symbol # which introduces new k-long substrings in the computation of $H_k(\widehat{\mathcal{S}}_{\mathcal{D}})$. Point (1) is a limitation for building large *static* compressed indexes in practice, being their construction space a primary concern [Puglisi et al. 2007]; point (2) will be experimentally investigated in Section 5 where we show that a compressed index built on $\widehat{\mathcal{S}}_{\mathcal{D}}$ may be up to 1.9 times larger than a compressed index built on $\mathcal{S}_{\mathcal{D}}$.

F		L	jump2end
$	hathiphopehot	#	⇓
$	hip$hope$hot$#$ha	t	⇓
$	hopehot#hathi	p	⇓
$	hot$#$hathiphop	e	⇓
$	#hathip$hope$ho	t	
a	thiphopehot#$	h	
e	hot#hathip$ho	p	
h	athiphopehot#	$	
h	ip$hope$hot$#$hat	$	
h	opehot#hathip	$	
h	ot$#$hathiphope	$	
i	p$hope$hot$#$hat$	h	
o	pehot#hathip$	h	
o	t$#$hathiphope$	h	
p	$hope$hot$#$hat$h	i	
p	ehot#hathip$h	o	
t	hiphopehot#$h	a	
t	$#$hathiphope$h	o	
#	hathip$hope$hot	$	

Fig. 3. Given the dictionary $\mathcal{D} = \{\text{hat}, \text{hip}, \text{hope}, \text{hot}\}$, we build the string $\mathcal{S}_\mathcal{D} =$ $\texttt{\$hat\$hip\$hope\$hot\$\#}$, and then compute its BW-transform. Arrows denote the positions incremented by the function `jump2end`.

3.2 A simple and efficient solution

Unlike the previous solution, our Compressed Permuterm index works on the plain string $\mathcal{S}_\mathcal{D}$, and is built in three steps (see Figure 3):

(1) Build the string $\mathcal{S}_\mathcal{D} = \$s_1\$s_2\$\ldots\$s_{m-1}\$s_m\$\#$. Recall that the dictionary strings are lexicographically ordered, and that symbol \$ (resp. #) is assumed to be smaller (resp. larger) than any other symbol of Σ.

(2) Compute $L = \texttt{bwt}(\mathcal{S}_\mathcal{D})$.

(3) Build a compressed data structure to support RANK queries over the string L (Lemma 2.1).

Our goal is to turn every wild-card search over the dictionary $\mathcal{D}$ into a substring search over the string $\mathcal{S}_\mathcal{D}$. Some of the queries indicated in Section 1 are immediately implementable as *substring searches* over $\mathcal{S}_\mathcal{D}$ (and thus they can be supported supported by procedure Backward_search and the RANK data structure built on L). But the sophisticated PREFIXSUFFIX query needs a different approach because it requires to *simultaneously match* a prefix and a suffix of a dictionary string, which are possibly far apart from each other in $\mathcal{S}_\mathcal{D}$. In order to circumvent this limitation, we prove a novel property of $\texttt{bwt}(\mathcal{S}_\mathcal{D})$ and deploy it to design a function, called `jump2end`, that allows to modify the procedure Backward_search of Figure 2 in a way that is suitable to support efficiently the PREFIXSUFFIX query. The main idea is that when Backward_search reaches the beginning of some dictionary string, say s_i, then it *"jumps" to the last symbol* of s_i rather than continuing onto the last symbol of its previous string in $\mathcal{D}$, i.e. the last symbol of s_{i-1}. Surprisingly enough, function $\texttt{jump2end}(i)$ consists of one line of code:

$$\texttt{if } 1 \le i \le m \texttt{ then return}(i+1) \texttt{ else return}(i)$$

and its correctness derives from the following two Lemmas. (Refer to Figure 3 for an illustrative example.)

LEMMA 3.1. *Given the sorted dictionary $\mathcal{D}$, and the way string $\mathcal{S}_\mathcal{D}$ is built, matrix $\mathcal{M}(\mathcal{S}_\mathcal{D})$ satisfies the following properties:*

—*The first row of $\mathcal{M}(\mathcal{S}_\mathcal{D})$ is prefixed by $\$s_1\$$, thus it ends with symbol $L[1] = \#$.*

—*For any $2 \leq i \leq m$, the i-th row of $\mathcal{M}(\mathcal{S}_\mathcal{D})$ is prefixed by $\$s_i\$$ and thus it ends with the last symbol of s_{i-1}, i.e. $L[i] = s_{i-1}[|s_{i-1}|]$.*

—*The $(m+1)$-th row of $\mathcal{M}(\mathcal{S}_\mathcal{D})$ is prefixed by $\$\#\$s_1\$$, and thus it ends with the last symbol of s_m, i.e. $L[m+1] = s_m[|s_m|]$.*

PROOF. The three properties come from the sorted ordering of the dictionary strings in $\mathcal{S}_\mathcal{D}$, from the fact that symbol \$ (resp. #) is the smallest (resp. largest) alphabet symbol, from the cyclic rotation of the rows in $\mathcal{M}(\mathcal{S}_\mathcal{D})$, and from their lexicographic ordering. □

The previous Lemma immediately implies the "locality" property deployed by function `jump2end`(i):

LEMMA 3.2. *Any row $i \in [1, m]$ is prefixed by $\$s_i\$$ and the next row $(i+1)$ ends with the last symbol of s_i.*

We are now ready to design the procedures for pattern searching and for displaying the strings of $\mathcal{D}$. As we anticipated above the main search procedure, called BackPerm_search, is derived from the original Backward_search of Figure 2 by adding one step which makes proper use of `jump2end`:

3′: First = `jump2end`(First); Last = `jump2end`(Last);

It is remarkable that the change is minimal (just one line of code!) and takes constant time, because `jump2end` takes $O(1)$ time. Let us now comment on the correctness of the new procedure BackPerm_search$(\beta\$\alpha)$ in solving the sophisticated query PREFIXSUFFIX$(\alpha * \beta)$. We note that BackPerm_search proceeds as the standard Backward_search for all symbols $Q[i] \neq \$$. In fact, the rows involved in these search steps do not belong to the range $[1, m]$, and thus `jump2end` is ineffective. When $Q[i] = \$$, the range [First, Last] is formed by rows which are prefixed by $\$\alpha$. By Lemma 3.2 we know that these rows are actually prefixed by strings $\$s_j$, with $j \in$ [First, Last], and thus these strings are in turn prefixed by $\$\alpha$. Given that [First, Last] $\subset [1, m]$, Step 3′ moves this range of rows to [First + 1, Last + 1], and thus identifies the new block of rows which are ended by the last symbols of those strings s_j (Lemma 3.2). After that, BackPerm_search continues by scanning backward the symbols of β (no other \$ symbol is involved), thus eventually finding the rows prefixed by $\beta\$\alpha$.

Figure 4 shows the pseudo-code of two other basic procedures: Back_step(i) and Display_string(i). The former procedure is a slight variation of the *backward step* implemented by any current compressed index based on BWT (see e.g. [Ferragina and Manzini 2005; Navarro and Mäkinen 2007]), here modified to support a leftward *cyclic* scan of every dictionary string. Precisely, if $F[i]$ is the j-th symbol of some dictionary string s, then Back_step(i) returns the row prefixed by the $(j-1)$-th symbol of that string if $j > 1$ (this is a standard backward step), otherwise it returns the row prefixed by the last symbol of s (by means of `jump2end`). Procedure

Algorithm Back_step(i)

(1) Compute $L[i]$;
(2) **return** $C[L[i]] + \texttt{rank}_{L[i]}(L, i)$;

Algorithm Display_string(i)

(1) *// Go back to the preceding* \$*, let it be at row* k_i
while $(F[i] \neq \$)$ **do** $i =$ Back_step(i);
(2) $i = \texttt{jump2end}(i)$;
(3) $s =$ empty string;
(4) *// Construct* $s = s_{k_i}$
while$(L[i] \neq \$)$ { $s = L[i] \cdot s$; $i =$ Back_step(i); };
(5) **return**(s);

Fig. 4. Algorithm Back_step is the one devised in [Ferragina and Manzini 2005] for standard compressed indexes. Algorithm Display_string(i) retrieves the string containing the symbol $F[i]$.

Display_string(i) builds upon Back_step(i) and retrieves the string s, namely the dictionary string that contains the symbol $F[i]$.

Using the data structures of Lemma 2.1 for supporting RANK queries over the string $L = \texttt{bwt}(\mathcal{S}_\mathcal{D})$, we obtain:

THEOREM 3.3. *Let $\mathcal{S}_\mathcal{D}$ be the string built upon a dictionary $\mathcal{D}$ of m strings having total length n and drawn from an alphabet Σ, such that $|\Sigma| = \texttt{polylog}(n)$. We can design a Compressed Permuterm index such that:*

—*Procedure* Back_step(i) *takes $O(1)$ time.*

—*Procedure* BackPerm_search($Q[1, q]$) *takes $O(q)$ time.*

—*Procedure* Display_string(i) *takes $O(|s|)$ time, if s is the string containing symbol $F[i]$.*

Space occupancy is bounded by $nH_k(\mathcal{S}_\mathcal{D}) + o(n)$ bits, for any $k \leq \alpha \log_{|\Sigma|} n$ and $0 < \alpha < 1$.

PROOF. For the time complexity, we observe that function `jump2end` takes constant time, and it is invoked $O(1)$ times at each possible iteration of procedures BackPerm_search and Display_string. Moreover, Back_step takes constant time, by Lemma 2.1. For the space complexity, we use the data structure of Lemma 2.1 (case 1) to support RANK queries on the string $L = \texttt{bwt}(\mathcal{S}_\mathcal{D})$. □

If $|\Sigma| = \Omega(\texttt{polylog}(n))$, the above time bounds must be multiplied by a factor $O(\log\log|\Sigma|)$ and the space bound has an additive term of $n \cdot o(\log|\Sigma|)$ bits (Lemma 2.1, case 2).

We are left with detailing the implementation of WILDCARD, RANK and SELECT queries for the Tolerant Retrieval problem. As it is standard in the Compressed Indexing literature we distinguish between two sub-problems: *counting* the number of dictionary strings that match the given wild-card query P, and *retrieving* these strings. Based on the Compressed Permuterm index of Theorem 3.3 we have:

—MEMBERSHIP query invokes BackPerm_search(\$$P$\$), then checks if First $<$ Last.

—PREFIX query invokes BackPerm_search($\$\alpha$) and returns the value Last − First + 1 as the number of dictionary strings prefixed by α. These strings can be retrieved by applying Display_string(i), for each $i \in$ [First, Last].

—SUFFIX query invokes BackPerm_search($\beta\$$) and returns the value Last − First + 1 as the number of dictionary strings suffixed by β. These strings can be retrieved by applying Display_string(i), for each $i \in$ [First, Last].

—SUBSTRING query invokes BackPerm_search(γ) and returns the value Last − First + 1 as the number of occurrences of γ as a substring of $\mathcal{D}$'s strings.[5] Unfortunately, the efficient retrieval of these strings cannot be through the execution of Display_string, as we did for the queries above. A dictionary string s may now be retrieved multiple times if γ occurs many times as a substring of s. To circumvent this problem we design a simple time-optimal retrieval, as follows. We use a bit vector V of size Last − First + 1, initialized to 0. The execution of Display_string is modified so that $V[j - \text{First}]$ is set to 1 when a row j within the range [First, Last] is visited during its execution. In order to retrieve once all dictionary strings that contain γ, we scan through $i \in$ [First, Last] and invoke the *modified* Display_string(i) only if $V[i - \text{First}] = 0$. It is easy to see that if $i_1, i_2, \ldots, i_k \in$ [First, Last] are the rows of $\mathcal{M}(\mathcal{S}_\mathcal{D})$ denoting the occurrences of γ in some dictionary string s (i.e. $F[i_j]$ is a symbol of s), only Display_string(i_1) is fully executed, thus taking $O(|s|)$ time. For all the other rows i_j, with $j > 1$, we find $V[i_j - \text{First}] = 1$ and thus Display_string(i_j) is not invoked.

—PREFIXSUFFIX query invokes BackPerm_search($\beta\$\alpha$) and returns the value Last − First + 1 as the number of dictionary strings which are prefixed by α and suffixed β. These strings can be retrieved by applying Display_string(i), for each $i \in$ [First, Last].

—RANK(P) invokes BackPerm_search($\$P\$$) and returns the value First, if First < Last, otherwise $P \notin \mathcal{D}$ (see Lemma 3.1) and thus the lexicographic position of P in $\mathcal{D}$ can be discovered by means of a slight variant of Backward_search whose details are given in Figure 6 (see Section 4.2 for further comments).

—SELECT(i) invokes Display_string(i) provided that $1 \leq i \leq m$ (see Lemma 3.1).

THEOREM 3.4. *Let $\mathcal{D}$ be a dictionary of m strings having total length n, drawn from an alphabet Σ such that $|\Sigma| = \texttt{polylog}(n)$. Our Compressed Permuterm index ensures that:*

—*If $P[1, p]$ is a pattern with* one-single *wild-card symbol, the query* WILDCARD(P) *takes $O(p)$ time to* count *the number of occurrences of P in $\mathcal{D}$, and $O(L_{occ})$ time to* retrieve *the dictionary strings matching P, where L_{occ} is their total length.*

—SUBSTRING(γ) *takes $O(|\gamma|)$ time to* count *the number of occurrences of γ as a substring of $\mathcal{D}$'s strings, and $O(L_{occ})$ time to* retrieve *the dictionary strings having γ as a substring, where L_{occ} is their total length.*

—RANK($P[1, p]$) *takes $O(p)$ time.*

—SELECT(i) *takes $O(|s_i|)$ time.*

[5]This is different from the problem of efficiently counting the number of strings containing γ. Our index does not solve this interesting problem (cfr. [Sadakane 2007] and references therein).

The space occupancy is bounded by $nH_k(\mathcal{S}_\mathcal{D}) + o(n)$ *bits, for any* $k \leq \alpha \log_{|\Sigma|} n$ *and* $0 < \alpha < 1$.

According to Lemma 2.1 (case 2), if $|\Sigma| = \Omega(\texttt{polylog}(n))$ the above time bounds must be multiplied by $O(\log\log|\Sigma|)$ and the space bound has an additive term of $n \cdot o(\log|\Sigma|)$ bits. We remark that our Compressed Permuterm index can support all wild-card searches *without* using any `locate`-data structure, which is known to be the main bottleneck of current compressed indexes [Navarro and Mäkinen 2007]: it implies the polylog-term in their query bounds and most of the $o(n\log|\Sigma|)$ term of their space cost. The net result is that our Compressed Permuterm index achieves in practice space occupancy much closer to known compressors and very fast queries, as we will experimentally show in Section 5.

A comment is in order at this point. Instead of introducing function `jump2end` and then modify the Backward_search procedure, we could have modified $L = \texttt{bwt}(\mathcal{S}_\mathcal{D})$ just as follows: cyclically rotate the prefix $L[1, m+1]$ of one single step (i.e. move $L[1] = \#$ to position $L[m+1]$). This way, we are actually *plugging* Lemma 3.2 directly into the string L. It is thus possible to show that the compressed index of Theorem 2.2 applied on the *rotated* L, is equivalent to the compressed permuterm index introduced in this paper. The performance in practice of this variation are slightly better since the computation of `jump2end` is no longer required. This is the implementation we used in the experiments of Section 5.

4. DYNAMIC COMPRESSED PERMUTERM INDEX

In this section we deal with the *dynamic* Tolerant Retrieval problem in which the dictionary $\mathcal{D}$ changes over the time under two update operations:

—InsertString(W) inserts the string W in $\mathcal{D}$.

—DeleteString(j) removes the j-th lexicographically smallest string s_j from $\mathcal{D}$.

The problem of maintaining a compressed index over a dynamically changing collection of strings, has been addressed in e.g. [Ferragina and Manzini 2005; Chan et al. 2007; Mäkinen and Navarro 2008]. In those papers the design of dynamic Compressed Indexes boils down to the design of dynamic compressed data structures for supporting `Rank/Select` operations. Here we adapt those solutions to the design of our dynamic Compressed Permuterm Index by showing that the insertion/deletion of an individual string s in/from $\mathcal{D}$ can be implemented via an *optimal* number $O(|s|)$ of basic insert/delete operations of *single* symbols in the compressed `Rank/Select` data structure built on $L = \texttt{bwt}(\mathcal{S}_\mathcal{D})$. Precisely, we will consider the following two basic update operations:

—`insert`(L, i, c) inserts symbol c between symbols $L[i]$ and $L[i+1]$.

—`delete`(L, i) removes the ith symbol $L[i]$.

The literature provides several dynamic data structures for supporting `Rank` queries and the above two update operations, with various time/space trade-offs. The best known results are currently due to [González and Navarro 2008]:

Lemma 4.1. *Let* $S[1, s]$ *be a string drawn from an alphabet* Σ *and let* $L = \texttt{bwt}(S)$ *be its BW-Transform. There exists a dynamic data structure that supports*

Algorithm DELETESTRING(j)

(1) $prev = j + 1$; $next = n$; $c = \$$;
(2) **while** ($next \neq j$) **do**
(3) $next =$ Back_step($prev$);
(4) `deleteF`(c); $c = L[prev]$; `delete`($L, prev$);
(5) **if** $prev < next$ **then** $next = next - 1$;
(6) $prev = next$;

Fig. 5. Algorithm to delete the string $\$s_j$ from $\mathcal{S}_\mathcal{D}$.

`rank`, `select` *and* `access` *operations in* L *taking* $O((1 + \log|\Sigma|/\log\log s)\log s)$ *time, and maintains* L *under insert and delete operations of single symbols in* $O((1 + \log|\Sigma|/\log\log s)\log s)$ *time. The space required by this data structure is* $nH_k(S) + o(n\log|\Sigma|)$ *bits, for any* $k < \alpha\log_{|\Sigma|} s$ *and constant* $0 < \alpha < 1$.

Our dynamic Compressed Permuterm Index is designed upon the above dynamic data structures, in a way that any improvement to Lemma 4.1 will positively reflect onto an improvement to our bounds. Therefore we will indicate the time complexities of our algorithms as a function of the number of `insert` and `delete` operations executed onto the changing string $L = \texttt{bwt}(\mathcal{S}_\mathcal{D})$. We also notice that these operations will change not only L but also the string F (which is the lexicographically sorted version of L, see Section 2). The maintenance of L will be discussed in the next subsections; while for F we will make use of the solution proposed in [Mäkinen and Navarro 2008, Section 7] that takes $|\Sigma|\log s + o(|\Sigma|\log s)$ bits and implements in $O(\log s)$ time the following query and update operations: $C[c]$ returns the number of symbols in F smaller than c; `deleteF`(c) removes from F an occurrence of symbol c; and `insertF`(c) adds an occurrence of symbol c in F.

The next two subsections detail our implementations of INSERTSTRING and DELETESTRING. The former is a slight modified version of the algorithm introduced in [Chan et al. 2007], here adapted to deal with the specialties of our dictionary problem: namely, the dictionary strings forming $\mathcal{S}_\mathcal{D}$ must be kept in lexicographic order. The latter coincides with the algorithm presented in [Mäkinen and Navarro 2008] for which we prove an additional property (Lemma 4.2) which is a key for using this result as is in our context.

4.1 Deleting one dictionary string

The operation DELETESTRING(j) requires to delete the string s_j from the dictionary $\mathcal{D}$, and thus recompute the BW-transform L' of the new string $\mathcal{S}_\mathcal{D}{}' = \$s_1\$\ldots\$s_{j-1}\$s_{j+1}\$\ldots\$s_m\$\#$. The key property we deploy next is that this removal does not impact on the ordering of the rows of $\mathcal{M}(\mathcal{S}_\mathcal{D})$ which do not refer to suffixes of $\$s_j$.

LEMMA 4.2. *The removal from* L *of the symbols of* $\$s_j$ *gives the correct string* $\texttt{bwt}(\mathcal{S}_\mathcal{D}{}')$.

PROOF. It is enough to prove that the removal of $\$s_j$ will not influence the order between any pair of rows $i' < i''$ in $\mathcal{M}(\mathcal{S}_\mathcal{D})$. Take i', i'' as two rows which are

not deleted from $\mathcal{M}(\mathcal{S}_\mathcal{D})$, and thus do not start/end with symbols of $\$s_j$. We compare the suffix of $\mathcal{S}_\mathcal{D}$ corresponding to the i'-th row, say $S_{i'}$, and the suffix of $\mathcal{S}_\mathcal{D}$ corresponding to the i''-th row, say $S_{i''}$. We recall that these are *increasing* strings, in that they are composed by the dictionary strings which are arranged in increasing lexicographic order and they are separated by the special symbol \$ (see Section 3.2). Since all dictionary strings are distinct, the mismatch between $S_{i'}$ and $S_{i''}$ occurs before the second occurrence of \$ in them. Let us denote the prefix of $S_{i'}$ and $S_{i''}$ preceding the second occurrence of \$ with $\alpha'\$s'\$$ and $\alpha''\$s''\$$, respectively, where α', α'' are (possibly empty) suffixes of dictionary strings, and s', s'' are dictionary strings. If the mismatch occurs in α' or α'' we are done, because they are not suffixes of $\$s_j$ (by the assumption), and therefore they are not interested by the deletion process. If the mismatch occurs in s' or s'' and they are both different of s_j, we are also done. The trouble is when $s' = s_j$ or $s'' = s_j$. We consider the first case, because the second is similar. This case occurs when $|\alpha'| = |\alpha''|$, so that the order between $S_{i'}$ and $S_{i''}$ is given by the order of s' vs s''. If $s' = s_j$, then the order of the two rows is then given by comparing s_{j+1} and s''. Since $s' < s''$ (because $S_{i'} < S_{i''}$) and s_{j+1} is the smallest dictionary string greater than s', we have that $s_{j+1} \leq s''$, and the thesis follows. □

Given this property, we can use the same string-deletion algorithm of [Mäkinen and Navarro 2008] to remove all symbols of $\$s_j$ from L and F. (Figure 5 reports the pseudo-code of this algorithm, for the sake of completeness.)

4.2 Inserting one dictionary string

An implementation of InsertString(W) for standard compressed indexes was described in [Chan et al. 2007]. Here we present a slightly modified version of that algorithm which correctly deals with the maintenance of the lexicographic ordering of the dictionary strings in $\mathcal{S}_\mathcal{D}$, and the re-computation of its BW-transform. We recall that this order is crucial for the correctness of most of our query operations.

Let j be the lexicographic position of the string W in $\mathcal{D}$. InsertString(W) requires to recompute the BW-transform L' of the new string $\mathcal{S}_\mathcal{D}{}' = \$s_1\$ \ldots \$s_{j-1}\$W\$ s_j\$s_{j+1}\$ \ldots \$s_m\$\#$. For this purpose, we can use the reverse of Lemma 4.2 in order to infer that this insertion does not affect the ordering of the rows already in $\mathcal{M}(\mathcal{S}_\mathcal{D})$. Thus InsertString($W$) boils down to insert just the symbols of W in their correct positions within L (and, accordingly, in F). This is implemented in two main steps: first, we find the lexicographic position of W in $\mathcal{D}$ (Algorithm LexOrder(W)); and then, we deploy this position to infer the positions in L where all symbols of W have to be inserted (Algorithm InsertString).

The pseudo-code in Figure 6 details algorithm LexOrder(W) which assumes that any symbol of W already occurs in the dictionary strings. If this is not the case, we set $c = W[x]$ as the leftmost symbol of W which does not occur in any string of $\mathcal{D}$, and set c' as the smallest symbol which is lexicographically greater than c and occurs in $\mathcal{D}$. If LexOrder is correct, then LexOrder($W[1, x-1]c'$) returns the lexicographic position of W in $\mathcal{D}$.

Lemma 4.3. *Given a string $W[1, w]$ whose symbols occurs in $\mathcal{D}$,* LexOrder(W) *returns the lexicographic position of W among the strings in $\mathcal{D}$.*

Algorithm LexOrder($W[1, w]$)

(1) $i = w$, $c = W[w]$, First $= C[c] + 1$;
(2) **while** $(i \geq 1)$ **do**
(3) $\quad c = W[i - 1]$;
(4) $\quad$ First $= C[c] + \texttt{rank}_c(L, \text{First} - 1) + 1$;
(5) $\quad i = i - 1$;
(6) **return** $\texttt{rank}_\$(L, \text{First} - 1) + 1$

Fig. 6. Algorithm LexOrder($W[1, w]$) returns the lexicographic position of W in $\mathcal{D}$.

Algorithm InsertString($W[1, w], j$)

(1) $i = w$, First $= j + 1$, $f = \$$;
(2) **while** $(i \geq 1)$ **do**
(3) $\quad c = W[i]$;
(4) $\quad \texttt{insert}(L, \text{First}, c)$; $\texttt{insertF}(f)$;
(5) $\quad$ First $= C[c] + \texttt{rank}_c(L, \text{First} - 1) + 1$;
(6) $\quad f = c$, $i = i - 1$;
(7) $\texttt{insert}(L, \text{First}, \$)$; $\texttt{insertF}(f)$;

Fig. 7. Algorithm to insert string $\$W[1, w]$ by knowing its lexicographically order j among the strings in $\mathcal{D}$.

Proof. Its correctness derives from the correctness of Backward_search. At any step i, First points to the first row of $\mathcal{M}(\mathcal{S}_\mathcal{D})$ which is prefixed by the suffix $W[w - i, w]\$$. If such a row does not exist, First points to the first row of $\mathcal{M}(\mathcal{S}_\mathcal{D})$ which is lexicographically greater than $W[w - i, w]\$$. □

Now we have all the ingredients to describe algorithm InsertString(W). Suppose that j is the value returned by LexOrder($W[1, w]$). We have to insert the symbol $W[i]$ preceding any suffix $W[i+1, w]$ in its correct position of $L' = \texttt{bwt}(\mathcal{S}_\mathcal{D}')$ and update the string F too. The algorithm in Figure 7 starts from the last symbol $W[w]$, and inserts it at the $(j + 1)$-th position of $\texttt{bwt}(\mathcal{S}_\mathcal{D})$ (by Lemma 3.2). It also inserts the symbol \$ in F, since it is the first symbol of the $(j + 1)$-th row. After that, the algorithm performs a backward step from the $(j + 1)$-th row with the symbol $W[w]$ in order to find the position in L where $W[w - 1]$ should be inserted. Accordingly, the symbol $W[w]$ is inserted in F too. These insertions are executed in L and F until all positions of W are processed. Step 7 completes the process by inserting the special symbol \$. Overall, InsertString executes an optimal number of inserts of single symbols in L and F. We then use the dynamic data structures of Lemma 4.1 to dynamically maintain L, and the solution of [Mäkinen and Navarro 2008, Section 7] to maintain F, thus obtaining:

Theorem 4.4. *Let $\mathcal{D}$ be a dynamic dictionary of m strings having total length n, drawn from an alphabet Σ. The Dynamic Compressed Permuterm index supports all queries of the Tolerant Retrieval problem with a slowdown factor of $O((1 + \log |\Sigma| / \log \log n) \log n)$ with respect to its static counterpart (see Theorem 3.4). Ad-*

ditionally, it can support InsertString(W) *in* $O(|W|(1+\log|\Sigma|/\log\log n)\log n)$ *time; and* DeleteString(j) *in* $O(|s_j|(1+\log|\Sigma|/\log\log n)\log n))$ *time.*

The space occupancy is bounded by $nH_k(\mathcal{S}_\mathcal{D}) + o(n\log|\Sigma|)$ *bits, for any* $k \leq \alpha\log_{|\Sigma|} n$ *and* $0 < \alpha < 1$.

We point out again that any improvement to Lemma 4.1 will positively affect the dynamic bounds above.

5. EXPERIMENTAL RESULTS

We downloaded from `http://law.dsi.unimi.it/` various crawls of the web—namely, `arabic-2005`, `indocina-2004`, `it-2004`, `uk-2005`, `webbase-2001` [Boldi et al. 2004]. We extracted from `uk-2005` about 190Mb of distinct urls, and we derived from all crawls about 34Mb of distinct host-names. The dictionary of urls and hosts have been lexicographically sorted by *reversed host names* in order to maximize the longest common-prefix (shortly, `lcp`) shared by strings adjacent in the lexicographic order. We have also built a dictionary of (alphanumeric) terms by parsing the TREC collection `WT10G` and by dropping (spurious) terms longer than 50 symbols. These three dictionaries are representatives of string sets usually manipulated in Web search and mining engines.

Table I reports some statistics on these three dictionaries: `DictUrl` (the dictionary of urls), `DictHost` (the dictionary of hosts), and `DictTerm` (the dictionary of terms). In particular lines 3-5 describe the composition of the dictionaries at the *string level*, lines 6-8 account for the repetitiveness in the dictionaries at the *string-prefix level* (which affects the performance of front-coding and trie, see below), and the last three lines account for the repetitiveness in the dictionaries at the *sub-string level* (which affects the performance of compressed indexes). It is interesting to note that the `Total_lcp` varies between 55–69% of the dictionary size, whereas the amount of compression achieved by gzip, bzip2 and ppmdi is superior and reaches 67–92%. This proves that there is much repetitiveness in these dictionaries not only at the string-prefix level but also *within* the strings. The net consequence is that compressed indexes, which are based on the Burrows-Wheeler Transform (and thus have the same bzip2-core), should achieve on these dictionaries significant compression, much better than the one achieved by front-coding based schemes!

In Tables II and III we test the time and space performance of three (compressed) solutions to the Tolerant Retrieval problem:

CPI is our Compressed Permuterm Index of Section 3.2. In order to compress the string $\mathcal{S}_\mathcal{D}$ and implement procedures BackPerm_search and Display_string, we modified three types of compressed indexes available under the `Pizza&Chili` site [Ferragina and Navarro 2006], which represent the best choices in this setting. Namely `CSA`, `FM-index v2` (shortly `FMI`), and the alphabet-friendly FM-index (shortly `AFI`). We tested three variants of `CSA` and `FMI` by properly setting their parameter which allows to trade space occupancy by query performance.

FC data structure applies *front-coding* to groups of b adjacent strings in the sorted dictionary, and then keeps explicit pointers to the beginners of every group [Witten et al. 1999].

Statistics	DictUrl	DictHost	DictTerm
Size (Mb)	190	34	118
$\|\Sigma\|$	95	52	36
# strings	3,034,144	1,778,927	10,707,681
Avg_len strings	64.92	18.91	10.64
Max_len strings	1,138	180	50
Avg_lcp	45.85	11.25	6.81
Max_lcp	720	69	49
Total_lcp	68.81%	55.27%	58.50%
gzip -9	11.49%	23.77%	29.50%
bzip2 -9	10.86%	24.03%	32.58%
ppmdi -l 9	8.32%	19.08%	29.06%

Table I. Statistics on our three dictionaries.

Method	DictUrl	DictHost	DictTerm
Trie	1374.29%	1793.19%	1727.93%
FC-32	109.95%	113.22%	106.45%
FC-128	107.41%	109.91%	102.10%
FC-1024	106.67%	108.94%	100.84%
CPI-AFI	49.72%	47.48%	52.24%
CPI-CSA-64	37.82%	56.36%	73.98%
CPI-CSA-128	31.57%	50.11%	67.73%
CPI-CSA-256	28.45%	46.99%	64.61%
CPI-FMI-256	24.27%	40.68%	55.41%
CPI-FMI-512	18.94%	34.58%	47.80%
CPI-FMI-1024	16.12%	31.45%	44.13%

Table II. Space occupancy is reported as a percentage of the original dictionary size. Recall that Trie and Fc are built on both the dictionary strings and their reversals, in order to support PrefixSuffix queries.

	DictUrl		DictHost		DictTerm	
Method	10	60	5	15	5	10
Trie	0.1	0.2	0.4	0.5	1.2	0.9
FC-32	1.3	0.4	1.5	1	2.5	1.7
FC-128	3.2	1.0	3.4	1.8	4.6	2.8
FC-1024	26.6	5.2	24.6	11.0	25.0	14.6
CPI-AFI	1.8	2.9	1.6	2.5	2.9	3.0
CPI-CSA-64	4.9	5.6	4.3	5.2	5.4	5.7
CPI-CSA-128	7.3	8.0	6.9	7.6	7.6	8.3
CPI-CSA-256	11.8	14.1	11.8	12.5	12.8	13.2
CPI-FMI-256	11.9	9.8	19.3	15.5	22.5	20.1
CPI-FMI-512	16.2	13.4	28.4	23.1	34.2	30.3
CPI-FMI-1024	24.1	20.7	46.4	38.4	57.6	50.1

Table III. Timings are given in *μsecs/char* averaged over one million of searched patterns, whose length is reported at the top of each column. Value b denotes in CPI-FMI-b the bucket size of the FM-index, in CPI-CSA-b the sample rate of the function Ψ [Ferragina and Navarro 2006], and in FC-b the bucket size of the front-coding scheme. We recall that b allows in all these solutions to trade space occupancy per query time.

Trie is the ternary search tree of Bentley and Sedgewick which "combines the time efficiency of digital tries with the space efficiency of binary search trees" [Bentley and Sedgewick 1997].[6]

Theorem 3.4 showed that Cpi supports efficiently all queries of the Tolerant Retrieval problem. The same positive feature does not hold for the other two data structures. In fact Fc and Trie support only prefix searches over the indexed strings. Therefore, in order to implement the PrefixSuffix query, we need to build these data structures *twice*— one on the strings of $\mathcal{D}$ and the other on their reversals. This *doubles* the space occupancy, and slows down the search performance because we need to first make two prefix-searches, one for P's prefix α and the other for P's suffix β, and then we need to *intersect* the two candidate lists of answers. If we wish to also support the rank/select primitives, we need to add some auxiliary data that keep information about the left-to-right numbering of trie leaves, thus further increasing the space occupancy of the trie-based solution. In Table II we account for such "space doubling", but not for the auxiliary data, thus giving an advantage in space to these data structures wrt Cpi. It is evident the large space occupancy of ternary search trees because of the use of pointers and the explicit storage of the dictionary strings (without any compression). As predicted from the statistics of Table I, Fc achieves a compression ratio of about 40% on the original dictionaries, but more than 60% on their reversal. Further, we note that Fc space improves negligibly if we vary the bucket size b from 32 to 1024 strings, and achieves the best space/time trade-off when $b = 32$.[7] In summary, the space occupancy of the Fc solution is more than the original dictionary size, if we wish to support all queries of the Tolerant Retrieval problem! As far as the variants of Cpi are concerned, we note that their space improvement is significant: a multiplicative factor from 2 to 7 wrt Fc, and from 40 to 86 wrt Trie.

In Section 3.1 we mentioned another simple solution to the Tolerant Retrieval problem which was based on the compressed indexing of the string $\widehat{\mathcal{S}}_{\mathcal{D}}$, built by juxtaposing *twice* every string of $\mathcal{D}$. In that section we argued that this solution is *inefficient* in indexing time and compressed-space occupancy because of this "string duplication" process. Here we investigate experimentally our conjecture by computing and comparing the k-th order empirical entropy of the two strings $\widehat{\mathcal{S}}_{\mathcal{D}}$ and $\mathcal{S}_{\mathcal{D}}$. As predicted theoretically, the two entropy values are close for all three dictionaries, thus implying that the compressed indexing of $\widehat{\mathcal{S}}_{\mathcal{D}}$ should require about twice the compressed indexing of $\mathcal{S}_{\mathcal{D}}$ (recall that $|\widehat{\mathcal{S}}_{\mathcal{D}}| = 2|\mathcal{S}_{\mathcal{D}}| - 1$). To check this, we have then built two FM-indexes: one on $\widehat{\mathcal{S}}_{\mathcal{D}}$ and the other on $\mathcal{S}_{\mathcal{D}}$, by varying $\mathcal{D}$ over the three dictionaries. We found that the space occupancy of the FM-index built on $\widehat{\mathcal{S}}_{\mathcal{D}}$ is a factor 1.6–1.9 worse than our Cpi-Fmi built on $\mathcal{S}_{\mathcal{D}}$. So we were right when in Section 3.1 we conjectured the inefficiency of the compressed indexing of $\widehat{\mathcal{S}}_{\mathcal{D}}$.

We have finally tested the time efficiency of the above indexing data structures over a P4 2.6 GHz machine, with 1.5 Gb of internal memory and running Linux

[6] Code at `http://www.cs.princeton.edu/~rs/strings/`.

[7] A smaller b would enlarge the extra-space dedicated to pointers, a larger b would impact seriously on the time efficiency of the prefix searches.

kernel 2.4.20. We executed a large set of experiments by varying the searched-pattern length, and by searching one million patterns per length. Since the results were stable over all these timings, we report in Table III only the most significant ones by using the notation *microsecs per searched symbol* (shortly μs/char): this is obtained by dividing the overall time of an experiment by the total length of the searched patterns. We remark that the timings in Table III account for the cost of searching a pattern prefix and a pattern suffix of the specified length. While this is the total time taken by our Cpi to solve a PrefixSuffix query, the timings for Fc and Trie are *optimistic* evaluations because they should also take into account the time needed to intersect the candidate list of answers returned by the prefix/suffix queries! Keeping this in mind, we look at Table III and note that Cpi allows to trade space occupancy per query time: we can go from a space close to gzip–ppmdi and access time of 20–57 μs/char (i.e. `CPI-FMI-1024`), to an access time similar to Fc of few μs/char but using less than half of its space (i.e. `CPI-AFI`). Which variant of Cpi to choose depends on the application for which the Tolerant Retrieval problem must be solved.

We finally notice that, of course, any improvement to compressed indexes [Navarro and Mäkinen 2007] will immediately and positively impact onto our Cpi, both in theory and in practice. Overall our experiments show that Cpi is a *novel compressed storage scheme for string dictionaries* which is fast in supporting the sophisticated searches of the Tolerant Retrieval problem, and is as compact as the best known compressors!

6. CONCLUSIONS AND OPEN PROBLEMS

In this paper we have proposed a static and dynamic *Compressed* Permuterm Index that solves the Tolerant Retrieval problem in time proportional to the length of the searched pattern, and space close to the k-th order empirical entropy of the indexed dictionary. This index is based on an elegant variant of the Burrows-Wheeler Transform defined on a dictionary of strings of variable length, which allows to easily adapt known compressed indexes [Navarro and Mäkinen 2007] to solve the Tolerant Retrieval problem too. Our theoretical study has been complemented with a significant set of experiments which have shown that the Compressed Permuterm Index supports fast queries within a space occupancy that is close to the one achievable by compressing the string dictionary via gzip or bzip2. This improves known approaches based on front-coding [Witten et al. 1999] by more than 50% in absolute space occupancy, still guaranteeing comparable query time.

In [Manning et al. 2008] the more sophisticated wild-card query $P = \alpha * \beta * \gamma$ is also considered and implemented by intersecting the set of strings containing $\gamma\$\alpha$ with the set of strings containing β. Our compressed permuterm index allows to avoid the *materialization* of these two sets by working only on the compressed index built on the string $\mathcal{S}_\mathcal{D}$. The basic idea consists of the following steps:

—Compute $[\mathsf{First}', \mathsf{Last}'] = \mathsf{BackPerm\_search}(\gamma\$\alpha)$;
—Compute $[\mathsf{First}'', \mathsf{Last}''] = \mathsf{BackPerm\_search}(\beta)$;
—For each $r \in [\mathsf{First}', \mathsf{Last}']$ repeatedly apply Back_step of Figure 3 until it finds a row which either belongs to $[\mathsf{First}'', \mathsf{Last}'']$ or to $[1, m]$ (i.e. starts with \$).
—In the former case r is an answer to WildCard(P), in the latter case it is not.

The number of Back_step's invocations depends on the length of the dictionary strings which match the query PREFIXSUFFIX($\alpha * \gamma$). In practice, it is possible to engineer this paradigm to reduce the total number of Back_steps (see [Ferragina and Navarro 2006], FM-indexV2). The above scheme can be also used to answer more complex queries as $P = \alpha * \beta_1 * \beta_2 * \ldots * \beta_k * \gamma$, with possibly empty α and γ. The efficiency depends on the *selectivity* of the individual queries PREFIXSUFFIX($\alpha * \gamma$) and SUBSTRING(β_i), for $i = 1, \ldots, k$.

It would be then interesting to extend our results in two directions, either by proving guaranteed and efficient worst-case bounds for queries with *multiple* wild-card symbols, or by turning our Compressed Permuterm index in a I/O-conscious or, even better, cache-oblivious compressed data structure. This latter issue actually falls in the key challenge of current data structural design: does it exist a cache-oblivious compressed index?

Acknowledgments

The authors would like to thank the anonymous referees and Gonzalo Navarro for their valuable technical comments and their help in improving the presentation of the paper.

REFERENCES

BAEZA-YATES, R. AND GONNET, G. 1996. Fast text searching for regular expressions or automaton searching on tries. *Journal of the ACM 43,* 6, 915–936.

BAEZA-YATES, R. AND RIBEIRO-NETO, B. 1999. *Modern Information Retrieval.* ACM/Addison-Wesley.

BARBAY, J., HE, M., MUNRO, J., AND RAO, S. S. 2007. Succinct indexes for string, binary relations and multi-labeled trees. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA).* 680–689.

BENTLEY, J. L. AND SEDGEWICK, R. 1997. Fast algorithms for sorting and searching strings. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA).* 360–369.

BOLDI, P., CODENOTTI, B., SANTINI, M., AND VIGNA, S. 2004. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience 34,* 8, 711–726.

BURROWS, M. AND WHEELER, D. 1994. A block sorting lossless data compression algorithm. *TR n. 124, Digital Equipment Corporation.*

CHAN, H., HON, W., LAM, T., AND SADAKANE, K. 2007. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms 3,* 2.

FERRAGINA, P., GIANCARLO, R., MANZINI, G., AND SCIORTINO, M. 2005. Boosting textual compression in optimal linear time. *Journal of the ACM 52*, 688–713.

FERRAGINA, P., KOUDAS, N., MUTHUKRISHNAN, S., AND SRIVASTAVA, D. 2003. Two-dimensional substring indexing. *Journal of Computer System Science 66,* 4, 763–774.

FERRAGINA, P. AND MANZINI, G. 2005. Indexing compressed text. *Journal of the ACM 52,* 4, 552–581.

FERRAGINA, P., MANZINI, G., MÄKINEN, V., AND NAVARRO, G. 2007. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms 3,* 2.

FERRAGINA, P. AND NAVARRO, G. 2006. Pizza&Chili corpus home page. `http://pizzachili.dcc.uchile.cl/` *or* `http://pizzachili.di.unipi.it/`.

GARFIELD, E. 1976. The permuterm subject index: An autobiographical review. *Journal of the ACM 27*, 288–291.

GONZÁLEZ, R. AND NAVARRO, G. 2008. Improved dynamic rank-select entropy-bound structures. In *Procs of the Latin American Symposium on Theoretical Informatics (LATIN).* Lecture Notes in Computer Science 4957, Springer. 374–386.

Mäkinen, V. and Navarro, G. 2008. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms 4,* 3.

Manning, C. D., Raghavan, P., and Schülze, H. 2008. *Introduction to Information Retrieval*. Cambridge University Press.

Mantaci, S., Restivo, A., Rosone, G., and Sciortino, M. 2005. An extension of the burrows wheeler transform and applications to sequence comparison and data compression. In *Procs of Symposium on Combinatorial Pattern Matching (CPM)*. Lecture Notes in Computer Science 3537, Springer. 178–189.

Manzini, G. 2001. An analysis of the Burrows-Wheeler transform. *Journal of the ACM 48,* 3, 407–430.

Navarro, G. and Mäkinen, V. 2007. Compressed full text indexes. *ACM Computing Surveys 39,* 1.

Puglisi, S., Smyth, W., and Turpin, A. 2007. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys 39,* 2.

Sadakane, K. 2007. Succinct data structures for flexible text retrieval systems. *J. Discrete Algorithms 5,* 1, 12–22.

Witten, I. H., Moffat, A., and Bell, T. C. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers.