Compressed Permuterm Index *

Paolo Ferragina
Dipartimento di Informatica
University of Pisa, Italy
ferragina@di.unipi.it

Rossano Venturini
Dipartimento di Informatica
University of Pisa, Italy
rventurini@di.unipi.it

ABSTRACT

Recently [16] resorted the *Permuterm index* of Garfield (1976) as 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 its quadruples the dictionary size. In this paper we propose the *Compressed* Permuterm Index which solves the Tolerant Retrieval problem in optimal query time, i.e. time proportional to the length of the searched pattern, and space close to the k-th order empirical entropy of the indexed dictionary. Our index can be used to solve also more sophisticated queries which involve several wild-card symbols, or require to prefix-match multiple fields in a database of records.

The result 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 [14] to solve the Tolerant Retrieval problem. Experiments show that our index supports fast queries within a space occupancy that is close to the one achievable by compressing the string dictionary via gzip, bzip2 or ppmdi. This improves known approaches based on front-coding by more than 50% in absolute space occupancy, still guaranteeing comparable query time.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing; E.4 [**Coding and Information Theory**]: Data compaction and compression; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical algorithms and Problems—*Pattern matching*

General Terms

Algorithms, Theory, Experiments.

*Partially supported by Yahoo! Research grant on "Data compression and indexing in hierarchical memories".

SIGIR'07, July 23–27, 2007, Amsterdam, The Netherlands.

Keywords

Indexing a dictionary of strings, Wild-Card searches, Data Compression, Burrows-Wheeler Transform.

1. THE PROBLEM

String processing and searching tasks are at the core of modern web search, IR 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 primitives 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. 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 and efficient data structures for the so called *tolerant retrieval* problem, defined as follows [16].

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 two basic rank/select primitives discussed above:

- 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 [1]. Hashing supports only the exact MEMBERSHIP query; its more sophisticated variant called *minimal ordered perfect* hashing [17] supports also the RANK operation but only on strings of $\mathcal{D}$. All other queries need however the inefficient scan of the whole dictionary $\mathcal{D}$!

Tries are more powerful in searching than hashing, but they fail to provide an efficient solution to the PREFIXSUFFIX query. In fact, the search for $P = \alpha * \beta$ needs to visit the whole 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 $O(|\mathcal{D}|)$ time independently of the number of query answers (cfr [2]). We can circumvent this limitation by using the sophisticated approach proposed in [7] 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 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 $\Omega(n \log n)$ bits,[1] and there is a large constant hidden in the big-O notation due to the presence of the two tries and the geometric data structure.

Recently [16] resorted the *Permuterm index* of Garfield [11] 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." [16]. In practice, one memory word per rotated string (and thus 4 bytes per character) is needed to index it, for a total of $\Omega(n \log n)$ bits.

In this paper we propose a *compressed* permuterm index which solves the tolerant retrieval problem in optimal query time, i.e. time proportional to the length of the queried string, and space close to the k-th order empirical entropy of the dictionary $\mathcal{D}$ (see Section 2 for definitions). The latter is an 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. Known effective compressors are gzip[2], bzip2[3] and ppmdi[4].

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 compressed indexes [8, 14] may be easily adapted to solve the Tolerant Retrieval problem without any loss in time and space efficiency.

We experiment our solution over various large dictionaries of URLs, hosts and terms, and compare it against some classical approaches to the Tolerant Retrieval problem mentioned in [16, 17] such as tries, front-coded dictionaries, and ZGrep.[5] Experiments show that tries are much space consuming, and ZGrep is too slow to be used in any applicative scenario. As mentioned in [17], and confirmed by its use in most open-source search engines (e.g. LUCENE), front-coding is the best known approach in terms of time/space trade-off. Our compressed permuterm index improves the space occupancy of front-coding by more than 50% in absolute space occupancy, resulting close to gzip, bzip2 and ppmdi. The query time is comparable to front-coding, taking few μ-secs per searched character. The flexibility of our compressed index allows to trade query time for space occupancy, thus offering a plethora of solutions for the Tolerant Retrieval problem which may well adapt to different applicative scenarios (see Section 3.3). We can thus safely state that it is no longer the case that a Permuterm index is an elegant approach which quadruples the dictionary size!

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:

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

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 [17].

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:

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

We have $H_k(T) \geq H_{k+1}(T)$ for any $k \geq 0$. As usual in data compression [13], 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.

[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

[5] A grep over gzip-ed files, available on all Linux platforms.

		F	L
mississippi$		$ mississipp	i
ississippi$m		i $mississip	p
ssissippi$mi		i ppi$missis	s
sissippi$mis		i ssippi$mis	s
issippi$miss		i ssissippi$	m
ssippi$missi	$\Longrightarrow$	m ississippi	$
sippi$missis		p i$mississi	p
ippi$mississ		p pi$mississ	i
ppi$mississi		s ippi$missi	s
pi$mississip		s issippi$mi	s
i$mississipp		s sippi$miss	i
$mississippi		s sissippi$m	i

Figure 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 column $L =$ ipssm$pissii.

In [6] 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 the string L by taking the last column of the sorted matrix $\mathcal{M}(T)$. We set $\texttt{bwt}(T) = L$.

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 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. [14]), Property 2 is the key for designing modern data compressors (see e.g. [13]).

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 [6]:

(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* character 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 character s is $T[6]$.

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].

Figure 2: The algorithm of [8] to find the contiguous range [First, Last] **of rows of $\mathcal{M}(T)$ prefixed by $Q[1,q]$.**

In order to map characters in L to their corresponding characters in F, [8] introduced the following function:

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

where $C[c]$ counts the number of characters 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 character $L[i]$ corresponds to character $F[LF(i)]$. 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 $T[6]$.

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 theoretical and practical solutions for this problem (see e.g. [14, 3] are references therein).

LEMMA 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 on L in $O(1)$ time using $nH_k(T) + o(n)$ bits of space, for any $k \leq \alpha \log_{|\Sigma|} n$ and $0 < \alpha < 1$, and retrieves any character of L in the same time bound [9, Theorem 5].*
2. *For general Σ, there exists a data structure which supports* rank *queries on L in $O(\log\log|\Sigma|)$ time, using $nH_k(T) + o(n\log|\Sigma|)$ bits of space, for any $k \leq \alpha \log_{|\Sigma|} n$ and $0 < \alpha < 1$, and retrieves any character of L in the same time bound [3, 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 [6], we can compute T from L by moving backward from $L[1] = T[n]$.

Ferragina and Manzini [8] made one step further by showing that data structures for supporting RANK queries on the string L are enough to search for a 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). [8] proved that the pseudo-code in Figure 2 maintains the invariant above for all phases, so at the end [First, Last] delimits the rows prefixed by Q (if any).

Plugging Lemma 1 into Backward_search, [9, 3] obtained:

THEOREM 1. *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 $nH_k(T) + o(n \log |\Sigma|)$ bits, for any $k \leq \alpha \log_{|\Sigma|} n$ and $0 < \alpha < 1$.* □

Notice that compressed indexes support also other operations [14], like locate and display of pattern occurrences, which are slower than Backward_search in that they require `polylog(n)` time per occurrence. We do not go into further details on them 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 [12] 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, optimal in time and entropy-bounded in space.

3.1 A simple 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) in the Tolerant Retrieval problem 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 1. Unfortunately, this approach suffers of various inefficiencies in the indexing and searching steps. To see them, let us "compare" string $\widehat{\mathcal{S}}_{\mathcal{D}}$ against the string $\mathcal{S}_D = \$s_1\$s_2\$ \ldots \$s_{m-1}\$s_m\$\#$, 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 $|\widehat{\mathcal{S}}_{\mathcal{D}}| = 2|\mathcal{S}_D| - 1$; and (2) doubles the space bound of compressed indexes evaluated in Theorem 1, because we can prove that $|\widehat{\mathcal{S}}_{\mathcal{D}}|H_k(\widehat{\mathcal{S}}_{\mathcal{D}}) \cong 2|\mathcal{S}_D|H_k(\mathcal{S}_D) \pm m(k \log |\Sigma| + 2)$, 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 compressed indexes, being their construction space a primary concern [15]; point (2) will be experimentally investigated in Section 4 where we will 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 $\mathcal{S}_D$.

F		L	jump2end
\$	hat\$hip\$hop\$hot\$	#	⇓
\$	hip\$hop\$hot\$#\$ha	t	⇓
\$	hop\$hot\$#\$hat\$hi	p	⇓
\$	hot\$#\$hat\$hip\$ho	p	⇓
\$	#\$hat\$hip\$hop\$ho	t	
a	t\$hip\$hop\$hot\$#\$	h	
h	at\$hip\$hop\$hot\$#	\$	
h	ip\$hop\$hot\$#\$hat	\$	
h	op\$hot\$#\$hat\$hip	\$	
h	ot\$#\$hat\$hip\$hop	\$	
i	p\$hop\$hot\$#\$hat\$	h	
o	p\$hot\$#\$hat\$hip\$	h	
o	t\$#\$hat\$hip\$hop\$	h	
p	\$hop\$hot\$#\$hat\$h	i	
p	\$hot\$#\$hat\$hip\$h	o	
t	\$hip\$hop\$hot\$#\$h	a	
t	\$#\$hat\$hip\$hop\$h	o	
#	\$hat\$hip\$hop\$hot	\$	

Figure 3: Given the dictionary $\mathcal{D} = \{\textbf{hat}, \textbf{hip}, \textbf{hop}, \textbf{hot}\}$, we build the string $\mathcal{S}_D =$ \$hat\$hip\$hop\$hot\$#, and then compute its BW-transform. Arrows denote the positions incremented by the function `jump2end`.

3.2 A simple and efficient solution

Our Compressed Permuterm index works on the plain string $\mathcal{S}_D$, and is built in three steps (see Figure 3):

1. Build the string $\mathcal{S}_D = \$s_1\$s_2\$ \ldots \$s_{m-1}\$s_m\$\#$. 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 $L = \texttt{bwt}(\mathcal{S}_D)$.
3. Build a compressed data structure to support RANK queries over the string L (Lemma 1).

Our goal is to turn every wild-card search over the dictionary $\mathcal{D}$ into a substring search over the string $\mathcal{S}_D$. Some of the queries indicated in Section 1 are immediately implementable as *substring searches* over $\mathcal{S}_D$ (and thus they can be supported by standard compressed indexes built on $\mathcal{S}_D$). 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 $\mathcal{S}_D$. In order to circumvent this limitation, we prove a novel property of $\texttt{bwt}(\mathcal{S}_D)$ 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 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 its last character* rather than continuing on the last character of its previous string in $\mathcal{D}$, i.e. the last character of s_{i-1}. Surprisingly enough, function `jump2end`(i) consists of one line of code:

`if` $1 \leq i \leq m$ `then return`$(i+1)$ `else return`(i)

and its correctness derives from the following two Lemmas.

LEMMA 2. *Given the sorted dictionary $\mathcal{D}$, matrix $\mathcal{M}(\mathcal{S}_D)$ satisfies the following properties:*

- *The first row of $\mathcal{M}(\mathcal{S}_D)$ is prefixed by $\$s_1\$$ and thus it ends with character $L[1] = \#$.*
- *For any $2 \leq i \leq m$, the i-th row of $\mathcal{M}(\mathcal{S}_D)$ is prefixed by $\$s_i\$$ and thus it ends with the last character of s_{i-1}, i.e. $L[i] = s_{i-1}[|s_{i-1}|]$.*

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

PROOF. Refer to Figure 3 for an illustrative example. The three properties come from the sorted ordering of the dictionary strings, the definition of the special symbols \$ and #, the cyclic rotation of the string $\mathcal{S}_D$ to form the rows of $\mathcal{M}(\mathcal{S}_D)$, and the lexicographic ordering of these rows. □

The previous Lemma immediately implies the following one:

LEMMA 3. *Any row $i \in [1, m]$ is prefixed by $\$s_i\$$ and the next row $(i+1)$ ends with the last character of s_i.*

This "locality" property is the one deployed by function `jump2end`(i), proving thus its correctness.

We are now ready to design the procedures for searching and 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 a step which makes a proper use of the function `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 characters $Q[i] \neq \$$. 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 we also know that these rows are 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 correctly identifies the new block of rows which are ended by the last characters of the strings s_j (Lemma 3). After that, BackPerm_search continues by scanning backward the characters of β (no other \$ character 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. [8, 14]), here modified to support a leftward *cyclic* scan of every dictionary string. Precisely, if $F[i]$ is the j-th character of some string s_{k_i}, then Back_step(i) returns the row prefixed by the $(j-1)$-th character of that string if $j > 1$ (this is a standard backward step), otherwise it returns the row prefixed by the last character of s_{k_i} (by means of `jump2end`). Procedure Display_string(i) builds upon Back_step(i) and retrieves the string containing the character $F[i]$ (or equivalently, the string whose suffix prefixes the row i of $\mathcal{M}(\mathcal{S}_D)$).

Using the data structures of Lemma 1 for supporting rank queries over strings, we obtain:

THEOREM 2. *Let $\mathcal{S}_D$ be the string built upon a dictionary $\mathcal{D}$ of m strings having total length n and drawn from an alphabet Σ, such that $|\Sigma| =$ `polylog`(n). We can design a Compressed Permuterm index such that:*

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

Algorithm Back_step(i)

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

Algorithm Display_string(i)

1. *// Go back to preceding \$, let it be at row k_i*
   while $(F[i] \neq \$)$ **do** $i =$ Back_step(i);
2. $i =$ `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);

Figure 4: Algorithm Back_step is the one devised in [8] for standard compressed indexes here modified to support a leftward cyclic scan of a dictionary string. Algorithm Display_string(i) retrieves the string containing the character $F[i]$.

- *Procedure* BackPerm_search$(Q[1, q])$ *takes $O(q)$ time.*
- *Procedure* Display_string(i) *takes $O(|s_{k_i}|)$ time, if s_{k_i} is the string containing the character $F[i]$.*

All time bounds are optimal. Space occupancy is bounded by $nH_k(\mathcal{S}_D) + o(n)$ 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 1. For the space complexity, we use the rank data structure of Lemma 1 (case 1). □

If $|\Sigma| = \Omega($`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 $o(n \log|\Sigma|)$ bits and it holds for any $k \leq \alpha \log_{|\Sigma|} n$ and $0 < \alpha < 1$ (Lemma 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 wildcard query P, and *retrieving* these strings. Based on the Compressed Permuterm index of Theorem 2 we have:

- MEMBERSHIP query invokes BackPerm_search$(\$P\$)$ and then checks whether 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. Unfortunately,

the optimal-time 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 $\mathsf{Last} - \mathsf{First} + 1$, initialized to 0. The execution of Display_string is modified so that $V[j - \mathsf{First}]$ is set to 1 when row $j \in [\mathsf{First}, \mathsf{Last}]$ is visited during its execution. In order to retrieve once all dictionary strings that contain γ, we scan through $i \in [\mathsf{First}, \mathsf{Last}]$ and invoke the *modified* $\mathsf{Display\_string}(i)$ only if $V[i - \mathsf{First}] = 0$. It is easy to see that if $i_1, i_2, \ldots, i_k \in [\mathsf{First}, \mathsf{Last}]$ are the rows of $\mathcal{M}(\mathcal{S}_D)$ denoting the occurrences of γ in some string s_{k_i} (i.e. $F[i_j]$ is a character of s_{k_i}), only $\mathsf{Display\_string}(i_1)$ is fully executed, thus taking $O(|s_{k_i}|)$ time. For all the other rows i_j, with $j > 1$, we find $V[i_j - \mathsf{First}] = 1$ and thus $\mathsf{Display\_string}(i_j)$ is not invoked.

- PrefixSuffix query invokes $\mathsf{BackPerm\_search}(\beta\$\alpha)$ and returns the value $\mathsf{Last} - \mathsf{First} + 1$ as the number of dictionary strings which are prefixed by α and suffixed β. These strings can be retrieved by applying $\mathsf{Display\_string}(i)$, for each $i \in [\mathsf{First}, \mathsf{Last}]$.
- Rank(P) invokes $\mathsf{BackPerm\_search}(\$P\$)$ and returns the value of First, if $\mathsf{First} < \mathsf{Last}$, otherwise it concludes that $P \notin \mathcal{D}$ (see Lemma 2).
- Select(i) invokes $\mathsf{Display\_string}(i)$ provided that $1 \leq i \leq m$ (see Lemma 2).

Theorem 3. *Let $\mathcal{D}$ be a dictionary of m strings having total length n, drawn from an alphabet Σ such that $|\Sigma| =$ $\mathtt{polylog}(n)$. Our Compressed Permuterm index ensures that:*

- *If $P[1,p]$ is a pattern with* one-single *wild-card symbol, the query* WildCard(P) *takes $O(p)$ optimal time to* count *the number of occurrences of P in $\mathcal{D}$, and $O(L_P)$ optimal time to* retrieve *the dictionary strings matching P, where L_P is their total length.*
- Substring(γ) *takes $O(|\gamma|)$ optimal time to* count *the number of occurrences of γ as a substring of $\mathcal{D}$'s strings, and $O(L_\gamma)$ optimal time to* retrieve *the dictionary strings having γ as a substring, where L_γ is their total length.*
- Rank($P[1,p]$) *takes $O(p)$ optimal time.*
- Select(i) *takes $O(|s_i|)$ optimal time.*

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

According to Lemma 1 (case 2), if $|\Sigma| = \Omega(\mathtt{polylog}(n))$ the above time bounds must be multiplied by $O(\log\log|\Sigma|)$ and the space bound has an additive term of $o(n\log|\Sigma|)$ bits and it holds for any $k \leq \alpha\log_{|\Sigma|} n$ and $0 < \alpha < 1$. We also 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 compressed indexes [14]: it implies the polylog-term of their query bound and most of the $o(n\log|\Sigma|)$ term of their space cost (see Theorem 1). The net result is that our Compressed Permuterm index achieves in practice space occupancy much close to known compressors and very fast queries, as show in Section 4.

3.3 Some thoughts

It is interesting to note that, instead of introducing function `jump2end` and then modify the Backward_search procedure, we could have modified $L = \mathtt{bwt}(\mathcal{S}_D)$ 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 directly into the string L. It is then possible to show that the compressed index of Theorem 1 applied on the *rotated* L, is equivalent to the compressed permuterm index introduced in this paper (details in the full paper).

In [16] 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 $\mathcal{S}_D$. 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 strings of $\mathcal{D}$ which match PrefixSuffix($\alpha*\gamma$). In practice, it is possible to engineer this paradigm to reduce the total number of Back_steps (see [10], 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 (details in the full paper).

We finally observe that our search paradigm might result useful in other indexing contexts. For example, given a database of records consisting of string pairs $\langle \mathtt{name}_i, \mathtt{surname}_i \rangle$, one could be interested in searching for all records in the database whose field `name` is prefixed by string α and field `surname` is prefixed by string β. This query can be implemented by invoking PrefixSuffix($\alpha * \beta^R$) on a compressed permuterm index built on a dictionary of strings $\widehat{s}_i = \mathtt{name}_i\Box(\mathtt{surname}_i)^R$, where $\Box$ is a special symbol not occurring in Σ and x^R denotes the reversal of string x. Given the small space occupancy of our solution, one could think to build many indexes, specifically one per pair of fields on which a user might want to execute these types of searches!

4. EXPERIMENTAL RESULTS

We downloaded from `http://law.dsi.unimi.it/` various crawls of the web—namely, `arabic-2005`, `indocina-2004`, `it-2004`, `uk-2005` and `webbase-2001`. We extracted from `uk-2005` about 190Mb of distinct urls, and we derived from all crawls about 34Mb of distinct hosts. The dictionary of urls and hosts have been lexicographically sorted by *reversed host-domain* 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 characters. These three dictionaries are representatives of string sets usually manipulated in Web search and mining engines.

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 1: Statistics on our three dictionaries.

Table 1 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 2–3 we tested the performance of four (compressed) solutions to the Tolerant Retrieval problem:

CPI is our Compressed Permuterm Index of Section 3.2. In order to compress the string $\mathcal{S}_D$ and implement procedures BackPerm_search and Display_string, we modified three types of compressed indexes available under the Pizza&Chili site [10], 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 for 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 [17].[6]

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" [5].[7]

Zgrep is a grep-based approach over gzip-ed files available on all Linux platforms.

Theorem 3 showed that CPI supports efficiently all queries of the Tolerant Retrieval problem. The same positive feature does not hold for the other 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 of at least a factor 2 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 to the trie some auxiliary data that keep information about the in-order numbering of its nodes. In Table 2 we account for such "space doubling", but not for the auxiliary data, thus giving a space-advantage 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 1, 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 128 to 1024 strings.[8] Our experiments show that the best space/time trade-off is achieved when $b = 32$. 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.

A special comment deserves ZGrep which was considered in our experiments just for the sake of completeness, since it is the usual approach taken by programmers and users of Linux platforms to search over their compressed files. Its space occupancy is the one of gzip (Table 1), but its time efficiency is very poor— few tens of seconds per single query— because ZGrep is forced to decompress and scan the whole dictionary at any search operation.

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 dictionary 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 $\mathcal{S}_D$. 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 $\mathcal{S}_D$ (recall that $|\widehat{\mathcal{S}}_{\mathcal{D}}| = 2|\mathcal{S}_D| - 1$). We have then built two FM-indexes: one on $\widehat{\mathcal{S}}_{\mathcal{D}}$ and the other on $\mathcal{S}_D$, 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

[6]Recently [4] proposed a cache-oblivious variant of the String B-tree data structure using FC in its leaves. We were unable to get from the authors the source code of this solution in order to test it. We leave to the full paper a comparison with this and other cache-oblivious approaches. We just note here that their space-occupancy will be larger than the FC experimented in this paper.

[7]Code at http://www.cs.princeton.edu/∼rs/strings/.

[8]The open-source search engine LUCENE, available at http://lucene.apache.org/, uses $b = 128$ so this is one of the solutions we test.

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 2: Space occupancy is reported as a percentage of the dictionary size. Recall that TRIE and FC are built on 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 3: 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 [10], 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.

built on $\mathcal{S}_D$. So we were right in conjecturing the inefficiency of the compressed indexing of $\widehat{\mathcal{S}}_D$.

We 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 kernel 2.4.20. We executed a large set of experiments by varying the searched-pattern length, and by searching for one million patterns per length. Since the results were stable over all these timings, we report in Table 3 only the most significant ones by using the notation *microsecs per searched character* (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 3 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, it is an *optimistic* evaluation for FC and TRIE because they also need to intersect the candidate list of answers returned by the prefix/suffix queries! Keeping this in mind, we look at Table 3 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 defer the tests on the Display_string to the full paper.

We finally notice that, of course, any improvement to compressed indexes [14] 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!

5. REFERENCES

[1] R.A. Baeza-Yates and B.A. Ribeiro-Neto. *Modern Information Retrieval*. ACM/Addison-Wesley, 1999.

[2] R.A. Baeza-Yates and G. Gonnet. Fast text searching for regular expressions or automaton searching on tries *Journal of the ACM*, 43(6): 915–936, 1996.

[3] J. Barbay, M. He, J.I. Munro, and S. Srinivasa Rao. Succinct indexes for string, binary relations and multi-labeled trees. In *Proc. ACM-SIAM SODA*, 2007.

[4] M. Bender, M. Farach-Colton, and B. Kuszmaul. Cache-Oblivious String B-trees. In *Proc. ACM PODS*, 233–242, 2006.

[5] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. ACM-SIAM SODA*, 360–369, 1997.

[6] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. TR n. 124, Digital Equipment Corporation, 1994.

[7] P. Ferragina, N. Koudas, S. Muthukrishnan, and D. Srivastava. Two-dimensional substring indexing. *Journal of Computer Syst. Sci.*, 66(4):763–774, 2003.

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

[9] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In *Proc. SPIRE*, 150–160, 2004.

[10] P. Ferragina and G. Navarro. Pizza&Chili corpus home page. http://pizzachili.dcc.uchile.cl/ or http://pizzachili.di.unipi.it/.

[11] E. Garfield. The permuterm subject index: An autobiographical review. *Journal of the American Society for Information Science*, 27:288–291, 1976.

[12] S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. An extension of the burrows wheeler transform and applications to sequence comparison and data compression. In *Proc. CPM*, 178–189, 2005.

[13] G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

[14] G. Navarro and V. Mäkinen. Compressed full text indexes. *ACM Computing Surveys*, 39(1), 2007.

[15] S. Puglisi, W. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 2007 (to appear).

[16] C. D. Manning, P. Raghavan and H. Schülze. *Introduction to Information Retrieval*. Cambridge University Press, 2007 (to appear).

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