A Lempel-Ziv Text Index on Secondary Storage⋆

Diego Arroyuelo and Gonzalo Navarro

Dept. of Computer Science, University of Chile,
Blanco Encalada 2120, Santiago, Chile.
{darroyue,gnavarro}@dcc.uchile.cl

Abstract. *Full-text* searching consists in locating the occurrences of a given pattern $P[1..m]$ in a text $T[1..u]$, both sequences over an alphabet of size σ. In this paper we define a new index for full-text searching on *secondary storage*, based on the Lempel-Ziv compression algorithm and requiring $8uH_k + o(u \log \sigma)$ bits of space, where H_k denotes the k-th order empirical entropy of T, for any $k = o(\log_\sigma u)$. Our experimental results show that our index is significantly smaller than any other practical secondary-memory data structure: 1.4–2.3 times the text size *including the text*, which means 39%–65% the size of traditional indexes like *String B-trees* [Ferragina and Grossi, *JACM* 1999]. In exchange, our index requires more disk access to locate the pattern occurrences. Our index is able to report up to 600 occurrences per disk access, for a disk page of 32 kilobytes. If we only need to *count* pattern occurrences, the space can be reduced to about 1.04–1.68 times the text size, requiring about 20–60 disk accesses, depending on the pattern length.

1 Introduction and Previous Work

Many applications require to store huge amounts of text, which need to be searched to find patterns of interest. *Full-text searching* is the problem of locating the *occ* occurrences of a pattern $P[1..m]$ in a text $T[1..u]$, both modeled as sequences of symbols over an alphabet Σ of size σ. Unlike word-based text searching, we wish to find any *text substring*, not only whole *words* or *phrases*. This has applications in texts where the concept of word does not exist or is not well defined, such as in DNA or protein sequences, Oriental languages texts, MIDI pitch sequences, program code, etc. There exist two classical kind of queries, namely: (1) count(T, P): counts the number of occurrences of P in T; (2) locate(T, P): reports the starting positions of the *occ* occurrences of P in T.

Usually in practice the text is a very long sequence (of several of gigabytes, or even terabytes) which is known beforehand, and we want to locate (or count) the pattern occurrences as fast as possible. Thus, we preprocess T to build a data structure (or *index*), which is used to speed up the search, avoiding a sequential scan. However, by using an index we increase the space requirement. This is unfortunate when the text is very large. Traditional indexes like *suffix trees* [1]

---

require $O(u \log u)$ bits to operate; in practice this space can be 10 times the text size [2], and so the index does not fit entirely in main memory even for moderate-size texts. In these cases the index must be stored on secondary memory and the search proceeds by loading the relevant parts into main memory.

Text compression is a technique to represent a text using less space. We denote by H_k the k-th *order empirical entropy* of a sequence of symbols T over an alphabet of size σ [3]. The value uH_k provides a lower bound to the number of bits needed to compress T using any compressor that encodes each symbol considering only the context of k symbols that precede it in T. It holds that $0 \leqslant H_k \leqslant H_{k-1} \leqslant \cdots \leqslant H_0 \leqslant \log \sigma$ (log means $\log_2$ in this paper).

To provide fast access to the text using little space, the current trend is to use *compressed full-text self-indexes*, which allows one to search and retrieve any part of the text without storing the text itself, while requiring space proportional to the compressed text size (e.g., $O(uH_k)$ bits) [4, 5]. Therefore we *replace* the text with a more space-efficient representation of it, which at the same time provides indexed access to the text. This has applications in cases where we want to reduce the space requirement by not storing the text, or when accessing the text is so expensive that the index must search without having the text at hand, as occurs with most Web search engines. As compressed self-indexes replace the text, we are also interested in operations $\mathtt{display}(T, P, \ell)$, which displays a context of ℓ symbols surrounding the pattern occurrences and $\mathtt{extract}(T, i, j)$, which decompresses the substring $T[i..j]$, for any text positions $i \leqslant j$.

The use of a compressed full-text self-index may totally remove the need to use the disk. However, some texts are so large that their corresponding indexes do not fit entirely in main memory, even compressed. Unlike what happens with sequential text searching, which speeds up with compression because the compressed text is transferred faster to main memory [6], working on secondary storage with a compressed index usually requires *more* disk accesses in order to find the pattern occurrences. Yet, these indexes require less space, which in addition can reduce the seek time incurred by a larger index because seek time is roughly proportional to the size of the data.

We assume a model of computation where a *disk page* of size B (able to store $b = \omega(1)$ integers of $\log u$ bits, i.e. $B = b \log u$ bits) can be transferred to main memory in a single disk access. Because of their high cost, the performance of our algorithms is measured as the number of disk accesses performed to solve a query. We count *every* disk access, which is an upper bound to the real number of accesses, as we disregard the disk caching due to the operating system. We can hold a constant number of disk pages in main memory. We assume that our text T is static, i.e., there are no insertions nor deletions of text symbols.

There are not many works on full-text indexes on secondary storage, which definitely is an important issue. One of the best known indexes for secondary memory is the *String B-tree* [7], although this is not a compressed data structure. It requires (optimal) $O(\log_b u + \frac{m+occ}{b})$ disk accesses in searches and (worst-case optimal) $O(u/b)$ disk pages of space. This value is, in practice, about 12.5 times the text size (not including the text) [8], which is prohibitive for very large texts.

Clark and Munro [9] present a representation of suffix trees on secondary storage (the *Compact Pat Trees*, or *CPT* for short). This is not a compressed index, and also needs the text to operate. Although not providing worst-case guarantees, the representation is organized in such a way that the number of disk accesses is reduced to 3–4 per query. The authors claim that the space requirement of their index is comparable to that of suffix arrays, needing about 4–5 times the text size (not including the text).

Mäkinen et al. [10] propose a technique to store a *Compressed Suffix Array* on secondary storage, based on *backward searching* [11]. This is the only proposal to store a (*zero*-th order) compressed full-text self-index on secondary memory, requiring $u(H_0 + O(\log \log \sigma))$ bits of storage and a counting cost of at most $2(1 + m\lceil \log_B u \rceil)$ disk accesses. Locating the occurrences of the pattern would need $O(\log u)$ extra accesses per occurrence.

In this paper we propose a version of Navarro's LZ-index [12] that can be efficiently handled on secondary storage. Our index requires $8uH_k + o(u \log \sigma)$ bits of space for $k = o(\log_\sigma u)$. In practice the space requirement is about 1.4–2.3 times the text size including the text, which is significantly smaller than any other practical secondary-memory data structure. Although we cannot provide worst-case guarantees at search time (just as in [9]), our experiments show that our index is effective in practice, yet requiring more disk accesses than larger indexes: our LZ-index is able to report up to 600 occurrences per disk access, for a disk page of 32 kilobytes. On the other hand, `count` queries can be performed requiring about 20–60 disk accesses (depending on the pattern length).

2 The LZ-index Data Structure

Assume that the text $T[1..u]$ has been compressed using the LZ78 [13] algorithm into $n+1$ *phrases*, $T = B_0 \ldots B_n$. We say that i is the *phrase identifier* of phrase B_i. The data structures that conform the LZ-index are [12]:

1. *LZTrie*: is the trie formed by all the LZ78 phrases $B_0 \ldots B_n$. Given the properties of LZ78 compression, this trie has exactly $n + 1$ nodes, each one corresponding to a string.
2. *RevTrie*: is the trie formed by all the reverse strings $B_0^r \ldots B_n^r$. In this trie there could be *empty* nodes not representing any block.
3. *Node*: is a mapping from block identifiers to their node in *LZTrie*.
4. *RNode*: is a mapping from block identifiers to their node in *RevTrie*.

Each of these four structures requires $n \log n(1+o(1))$ bits if they are represented succinctly. As $n \log u = uH_k + O(kn \log \sigma) \leqslant u \log \sigma$ for any k [14], the final size of the LZ-index is $4uH_k + o(u \log \sigma)$ bits of space for any $k = o(\log_\sigma u)$.

We distinguish three types of occurrences of P in T, depending on the phrase layout [12]. For `locate` queries, pattern occurrences are reported in the format $[\![t, offset]\!]$, where t is the phrase where the occurrence starts, and $offset$ is the distance between the beginning of the occurrence and the end of the phrase. However, occurrences can be shown as text positions with little extra effort [15].

Occurrences of Type 1. The occurrence lies inside a single phrase (there are occ_1 occurrences of this type). Given the properties of LZ78, every phrase B_k containing P is formed by a shorter phrase B_ℓ concatenated to a symbol c. If P does not occur at the end of B_k, then B_ℓ contains P as well. We want to find the shortest possible phrase B_i in the LZ78 referencing chain for B_k that contains the occurrence of P. Since phrase B_i has the string P as a suffix, P^r is a prefix of B_i^r, and can be easily found by searching for P^r in *RevTrie*. Say we arrive at node v. Any node v' descending from v in *RevTrie* (including v itself) corresponds to a phrase terminated with P. Thus we traverse and report all the subtrees of the *LZTrie* nodes corresponding to each v'. Total locate time is $O(m + occ_1)$.

Occurrences of Type 2. The occurrence spans two consecutive phrases, B_k and B_{k+1}, such that a prefix $P[1..i]$ matches a suffix of B_k and the suffix $P[i+1..m]$ matches a prefix of B_{k+1} (there are occ_2 occurrences of this type). P can be split at any position, so we have to try them all. For every possible split $P[1..i]$ and $P[i+1..m]$ of P, assume the search for $P^r[1..i]$ in *RevTrie* yields node v_{rev}, and the search for $P[i+1..m]$ in *LZTrie* yields node v_{lz}. Then, we check each phrase t in the subtree of v_{rev} and report occurrence $[\![t, i]\!]$ if $Node[t+1]$ descends from v_{lz}. Each such check takes constant time. Yet, if the subtree of v_{lz} has fewer elements, we do the opposite: check phrases from v_{lz} in v_{rev}, using $RNode[t-1]$. The total time is proportional to the smallest subtree size among v_{rev} and v_{lz}.

Occurrences of Type 3. The occurrence spans three or more phrases, B_{k-1} $\ldots B_{\ell+1}$, such that $P[i..j] = B_k \ldots B_\ell$, $P[1..i-1]$ matches a suffix of B_{k-1} and $P[j+1..m]$ matches a prefix of $B_{\ell+1}$ (there are occ_3 occurrences of this type). As every phrase represents a different string (because of LZ78 properties), there is at most one phrase matching $P[i..j]$ for each choice of i and j. Thus, occ_3 is limited to $O(m^2)$ occurrences. We first identify the only possible phrase matching every substring $P[i..j]$. This is done by searching for every $P[i..j]$ in *LZTrie*, recording in a matrix $C_{lz}[i, j]$ the corresponding *LZTrie* node. Then we try to find the $O(m^2)$ maximal concatenations of successive phrases that match contiguous pattern substrings. If $P[i..j] = B_k \ldots B_\ell$ is a maximal concatenation, we check whether phrase $B_{\ell+1}$ starts with $P[j+1..m]$, i.e., we check whether $Node[\ell+1]$ is a descendant of node $C_{lz}[j+1, m]$. Finally we check whether phrase B_{k-1} ends with $P[1..i-1]$, by starting from $Node[i-1]$ in *LZTrie* and successively going to the parent to check whether the last $i-1$ nodes, read backwards, equal $P^r[1..i-1]$. If all these conditions hold, we report an occurrence $[\![k-1, i-1]\!]$. Overall locate time is $O(m^2 \log m)$ worst-case and $O(m^2)$ on average.

3 LZ-index on Secondary Storage

The LZ-index [12] was originally designed to work in main memory, and hence it has a non-regular pattern of access to the index components. As a result, it

is not suitable to work on secondary storage. In this section we show how to achieve locality in the access to the LZ-index components, so as to have good secondary storage performance. In this process we introduce some redundancy over main-memory proposals [12, 15].

3.1 Solving the Basic Trie Operations

To represent the tries of the index we use a space-efficient representation similar to the hierarchical representation of [16], which now we make searchable. We cut the trie into disjoint *blocks* of size B such that every block stores a subtree of the whole trie. We arrange these blocks in a tree by adding some *inter-block* pointers, and thus the trie is represented by a tree of subtrees.

We cut the trie in a *bottom-up* fashion, trying to maximize the number of nodes in each block. This is the same partition used by Clark and Munro [9], and so we also suffer of very small blocks. To achieve a better fill ratio and reduce the space requirement, we store several trie blocks into each disk page.

Every trie node x in this representation is either a leaf of the whole trie, or it is an internal node. For internal nodes there are two cases: the node x is internal to a block p or x is a leaf of block p (but not a leaf of the whole trie). In the latter case, x stores a pointer to the representation q of its subtree. The leaf is also stored as a fictitious root of q, so that every block is a subtree. Therefore, every such node x has two representations: (1) as a leaf in block p; (2) as the root node of the child block q.

Each block p of N nodes and root node x consists basically of:

- the *balanced parentheses* (BP) representation [17] of the subtree, requiring $2N + o(N)$ bits;
- a bit-vector $F_p[1..N]$ (the *flags*) such that $F_p[j] = 1$ iff the j-th node of the block (in preorder) is a leaf of p, but not a leaf of the whole trie. In other words, the j-th node has a pointer to the representation of its subtree. We represent F_p using the data structure of [18] to allow *rank* and *select* queries in constant time and requiring $N + o(N)$ bits;
- the sequence $lets_p[1..N]$ of symbols labeling the arcs of the subtree, in pre-order. The space requirement is $N\lceil \log \sigma \rceil$ bits;
- only in the case of *LZTrie*, the sequence $ids_p[1..N]$ of phrase identifiers in preorder. The space requirement is $N \log n$ bits.
- a pointer to the leaf representation of x in the parent block;
- the depth and preorder of x within the whole trie;
- a variable number of pointers to child blocks. The number of child blocks of a given block can be known from the number of 1s in F_p.
- an array $Size_p$ such that each pointer to child block stores the size of the corresponding subtree.

Using this information, given node x we are able to compute operations: $parent(x)$ (which gets the parent of x), $child(x, \alpha)$ (which yields the child of x by label α), $depth(x)$ (which gets the depth of x in the trie), $subtreesize(x)$ (which gets the

size of the subtree of x, including x itself), $preorder(x)$ (which gets the preorder number of x in the trie), and $ancestor(x, y)$ (which tells us whether x is ancestor of node y). Operations $subtreesize$, $depth$, $preorder$, and $ancestor$ can be computed without extra disk accesses, while operations $parent$ and $child$ require one disk access in the worst case. In [**?**] we explain how to compute them.

Analysis of Space Complexity. In the case of *LZTrie*, as the number of nodes is n, the space requirement is $2n + n + n\log\sigma + n\log n + o(n)$ bits, for the BP representation, the flags, the symbols, and phrase identifiers respectively. To this we must add the space required for the inter-block pointers and the extra information added to each block, such as the depth of the root, etc. If the trie is represented by a total of K blocks, these data add up to $O(K\log n)$ bits. The bottom-up partition of the trie ensures $K = O(n/b)$, so the extra information requires $O(\frac{n}{b}\log n)$ bits. As $b = \omega(1)$, this space is $o(n\log n) = o(u\log\sigma)$ bits.

In the case of *RevTrie*, as there can be empty nodes, we represent the trie using a *Patricia tree* [19], compressing empty unary paths so that there are $n \leqslant n' \leqslant 2n$ nodes. In the worst case the space requirement is $4n + 2n + 2n\log\sigma + o(n)$ bits, plus the extra information as before.

As we pack several trie blocks in a disk page, we ensure a utilization ratio of 50% at least. Hence the space of the tries can be at most doubled on disk.

3.2   Reducing the Navigation between Structures

We add the following data structures with the aim of reducing the number of disk accesses required by the LZ-index at search time:

- $Pre_{lz}[1..n]$: a mapping from phrase identifiers to the corresponding *LZTrie* preorder, requiring $n\log n$ bits of space.
- $Rev[1..n]$: a mapping from *RevTrie* preorder positions to the corresponding *LZTrie* node, requiring $n\log u + n$ bits of space. Later in this section we explain why we need this space.
- $TPos_{lz}[1..n]$: if the phrase corresponding to the node with preorder i in *LZTrie* starts at position j in the text, then $TPos_{lz}[i]$ stores the value j. This array requires $n\log u$ bits and is used for `locate` queries.
- $LR[1..n]$: an array requiring $n\log n$ bits. If the node with preorder i in *LZTrie* corresponds to the LZ78 phrase B_k, then $LR[i]$ stores the preorder of the *RevTrie* node for B_{k-1}.
- $S_r[1..n]$: an array requiring $n\log u$ bits, storing in $S_r[i]$ the subtree size of the *LZTrie* node corresponding to the i-th *RevTrie* node (in preorder). This array is used for counting.
- $Node[1..n]$: the mapping from phrase identifiers to the corresponding *LZTrie* node, requiring $n\log n$ bits. This is used to solve `extract` queries.

As the size of these arrays depends on the compressed text size, we do not need that much space to store them: they require $3n\log u + 3n\log n + n$ bits, which summed to the tries gives $8uH_k + o(u\log\sigma)$ bits, for any $k = o(\log_\sigma u)$.

If the index is used *only* for count queries, we basically need arrays Pre_{lz}, LR, S_r, and the tries, plus an array $RL[1..n]$, which is similar to LR but mapping from a *RevTrie* node for B_k to the *LZTrie* preorder for B_{k+1}. All these add up to $6uH_k + o(u\log\sigma)$ bits.

After searching for all pattern substrings $P[i..j]$ in *LZTrie* (recording in $C_{lz}[i,j]$ the phrase identifier, the preorder, and the subtree size of the corresponding *LZTrie* node, along with the node itself) and all reversed prefixes $P^r[1..i]$ in *RevTrie* (recording in array $C_r[i]$ the preorder and subtree size of the corresponding *RevTrie* node), we explain how to find the pattern occurrences.

Occurrences of Type 1. Assume that the search for P^r in *RevTrie* yields node v_r. For every node with preorder i, such that $preorder(v_r) \leqslant i \leqslant preorder(v_r) + subtreesize(v_r)$ in *RevTrie*, with $Rev[i]$ we get the node v_{lz_i} in *LZTrie* representing a phrase B_t ending with P. The length of B_t is $d = depth(v_{lz_i})$, and the occurrence starts at position $d - m$ inside B_t. Therefore, if $p = preorder(v_{lz_i})$, the exact text position can be computed as $TPos_{lz}[p] + d - m$. We then traverse all the subtree of v_{lz_i} and report, as an occurrence of type 1, each node contained in this subtree, accessing $TPos_{lz}[p..p + subtreesize(v_{lz_i})]$ to find the text positions. Note that the offset $d - m$ stays constant for all nodes in the subtree.

Note that every node in the subtree of v_r produces a random access in *LZTrie*. In the worst case, the subtree of v_{lz_i} has only one element to report (v_{lz_i} itself), and hence we have occ_1 random accesses in the worst case. To reduce the worst case to $occ_1/2$, we use the n extra bits in Rev: in front of the $\log u$ bits of each Rev element, a bit indicates whether we are pointing to a *LZTrie* leaf. In such a case we do not perform a random access to *LZTrie*, but we use the corresponding $\log u$ bits to store the exact text position of the occurrence.

To avoid accessing the same *LZTrie* page more than once, even for different trie blocks stored in that page, for each $Rev[i]$ we solve all the other $Rev[j]$ that need to access the same *LZTrie* page. As the tries are space-efficient, many random accesses could need to access the same page.

For count queries we traverse the S_r array instead of Rev, summing up the sizes of the corresponding *LZTrie* subtrees without accessing them, therefore requiring $O(occ_1/b)$ disk accesses.

Occurrences of Type 2. For occurrences of type 2 we consider every possible partition $P[1..i]$ and $P[i+1..m]$ of P. Suppose the search for $P^r[1..i]$ in *RevTrie* yields node v_r (with preorder p_r and subtree size s_r), and the search for $P[i+1..m]$ in *LZTrie* yields node v_{lz} (with preorder p_{lz} and subtree size s_{lz}). Then we traverse sequentially $LR[j]$, for $j = p_{lz}..p_{lz} + s_{lz}$, reporting an occurrence at text position $TPos_{lz}[j] - i$ iff $LR[j] \in [p_r..p_r + s_r]$. This algorithm has the nice property of traversing arrays LR and $TPos_{lz}$ sequentially, yet the number of elements traversed can be arbitrarily larger than occ_2.

For count queries, since we have also array RL, we choose to traverse $RL[j]$, for $j = p_r..p_r + s_r$, when the subtree of v_r is smaller than that of v_{lz}, counting an occurrence only if $RL[j] \in [p_{lz}..p_{lz} + s_{lz}]$.

To reduce the number of accesses from $2\lceil\frac{s_{lz}+1}{b}\rceil$ to $\lceil\frac{2(s_{lz}+1)}{b}\rceil$, we interleave arrays LR and $TPos_{lz}$, such that we store $LR[1]$ followed by $TPos_{lz}[1]$, then $LR[2]$ followed by $TPos_{lz}[2]$, etc.

Occurrences of Type 3. We find all the maximal concatenations of phrases using the information stored in C_{lz} and C_r. If we found that $P[i..j] = B_k \ldots B_\ell$ is a maximal concatenation, we check whether phrase $B_{\ell+1}$ has $P[j+1..m]$ as a prefix, and whether phrase B_{k-1} has $P[1..i-1]$ as a suffix. Note that, according to the LZ78 properties, $B_{\ell+1}$ starting with $P[j+1..m]$ implies that there exists a previous phrase B_t, $t < \ell+1$, such that $B_t = P[j+1..m]$. In other words, $C_{lz}[j+1, m]$ must not be null (i.e., phrase B_t must exist) and the phrase identifier stored at $C_{lz}[j+1, m]$ must be less than $\ell+1$ (i.e., $t < \ell+1$). If these conditions hold, we check whether $P^r[1..i-1]$ exists in *RevTrie*, using the information stored at $C_r[i-1]$. Only if all these condition hold, we check whether $Pre_{lz}[\ell+1]$ descends from the *LZTrie* node corresponding to $P[j+1..m]$ (using the preorder and subtree size stored at $C_{lz}[j+1, m]$), and if we pass this check, we finally check whether $LR[Pre_{lz}[k]]$ (which yields the *RevTrie* preorder of the node corresponding to phrase $k-1$) descend from the *RevTrie* node for $P^r[1..i-1]$ (using the preorder and subtree size stored at $C_r[i-1]$). Fortunately, we have a high probability that $Pre_{lz}[\ell+1]$ and $Pre_{lz}[k]$ need to access the same disk page. If we find an occurrence, the corresponding position is $TPos_{lz}[Pre_{lz}[k]] - (i-1)$.

Extract *Queries.* In [**?**] we explain how to solve extract queries.

4 Experimental Results

For the experiments of this paper we consider two text files: the text WSJ (Wall Street Journal) from the TREC collection [20], of 128 megabytes, and the XML file provided in the *Pizza&Chili Corpus*[1], downloadable from `http://pizzachili.dcc.uchile.cl/texts/xml/dblp.xml.200MB.gz`, of 200 megabytes. We searched for 5,000 random patterns, of length from 5 to 50, generated from these files. As in [8], we assume a disk page size of 32 kilobytes. We compared our results against the following state-of-the-art indexes for secondary storage:

Suffix Arrays **(SA):** following [21] we divide the suffix array into blocks of $h \leqslant b$ elements (pointers to text suffixes), and move to main memory the first l text symbols of the first suffix of each block, i.e. there are $\frac{u}{h}l$ extra symbols. We assume in our experiments that $l = m$ holds, which is the best situation. At search time, we carry out two binary searches [22] to delimit the interval $[i..j]$ of the pattern occurrences. Yet, the first part of the binary search is done over the samples without accessing the disk. Once the blocks where i and j lie are identified, we bring them to main memory and finish the binary search, this time accessing the text on disk at each comparison.

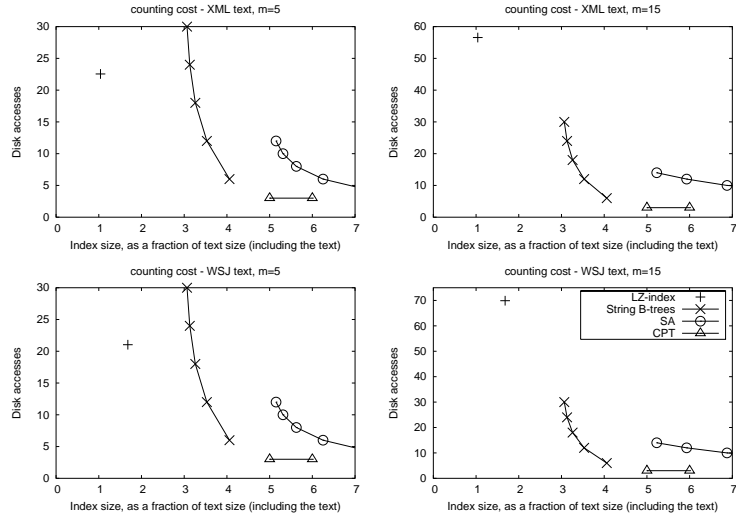[1] `http://pizzachili.dcc.uchile.cl`

Fig. 1. `Count` cost vs. space requirement for the different indexes we tested.

Therefore, the total cost is $2 + 2 \log h$ disk accesses. We must pay $\lceil \frac{occ}{b} \rceil$ extra accesses to report the occurrences of P within those two positions. The space requirement including the text is $(5 + \frac{m}{h})$ times the text size.

String B-trees [7]: in [8] they pointed out that an implementation of *String B-trees* for static texts would require about $2 + \frac{2.125}{k}$ times the text size (where $k > 0$ is a constant) and the height h of the tree is 3 for texts of up to 2 gigabytes, since the branching factor (number of children of each tree node) is $b' \approx \frac{b}{8.25}$. The experimental number of disk accesses given by the authors is $O(\log k)(\lfloor \frac{m}{b} \rfloor + 2h) + \lceil \frac{occ}{b'} \rceil$. We assume a constant of 1 for the $O(\log k)$ factor, since this is not clear in the paper [8, Sect. 2.1] (this is optimistic). We use $k = 2, 4, 8, 16$, and 32.

Compact Pat Trees (CPT) [9]: we assume that the tree has height 3, according to the experimental results of Clark and Munro. We need $1 + \lfloor \frac{occ}{b} \rfloor$ extra accesses to locate the pattern occurrences. The space is about 4–5 times the text size (plus the text).

We restrict our comparison to indexes that have been implemented, or at least simulated, in the literature. Hence we exclude the *Compressed Suffix Arrays* (CSA) [10] since we only know that it needs at most $2(1 + m \lceil \log_b u \rceil)$ accesses for `count` queries. This index requires about 0.22 and 0.45 times the text size for the XML and WSJ texts respectively, which, as we shall see, is smaller than ours. However, CSA requires $O(\log u)$ accesses to report *each* pattern occurrence[2].

---

[2] The work [23] extends this structure to achieve fast locate. The secondary-memory version is still a theoretical proposal and it is hard to predict how will it perform, so we cannot meaningfully compare it here.
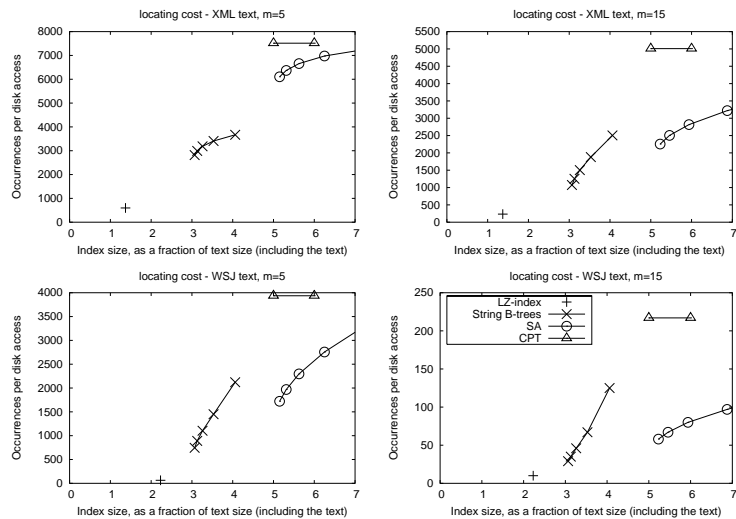
Fig. 2. `Locate` cost vs. space requirement for the different indexes we tested. Higher means better `locate` performance.

Fig. 1 shows the time/space trade-offs of the different indexes for `count` queries, for patterns of length 5 and 15. As it can be seen, our LZ-index requires about 1.04 times the text size for the (highly compressible) XML text, and 1.68 times the text size for the WSJ text. For $m = 5$, the counting requires about 23 disk accesses, and for $m = 15$ it needs about 69 accesses. Note that for $m = 5$, there is a difference of 10 disk accesses among the LZ-index and *String B-trees*, the latter requiring 3.39 (XML) and 2.10 (WSJ) times the space of the LZ-index. For $m = 15$ the difference is greater in favor of *String B-Trees*. The SA outperforms the LZ-index in both cases, the latter requiring about 20% the space of SA. Finally, the LZ-index needs (depending on the pattern length) about 7–23 times the number of accesses of CPTs, the latter requiring 4.9–5.8 (XML) and 3–3.6 (WSJ) times the space of LZ-index.

Fig. 2 shows the time/space trade-offs for `locate` queries, this time showing the average number of occurrences reported per disk access. The LZ-index requires about 1.37 (XML) and 2.23 (WSJ) times the text size. The LZ-index is able of reporting about 597 (XML) and 63 (WSJ) occurrences per disk access for $m = 5$, and about 234 (XML) and 10 (WSJ) occurrences per disk access for $m = 15$. The average number of occurrences found for $m = 5$ is 293,038 (XML) and 27,565 (WSJ); for $m = 15$ there are 45,087 and 870 pattern occurrences on average. *String B-trees* report 3,449 (XML) and 1,450 (WSJ) occurrences per access for $m = 5$, and for $m = 15$ the results are 1,964 (XML) and 66 (WSJ) occurrences per access, while requiring 2.57 (XML) and 1.58 (WSJ) times the space of the LZ-index.

Fig. 3 shows the cost for the different parts of the LZ-index search algorithm, in the case of XML (WSJ yields similar results): the work done in the tries (la-
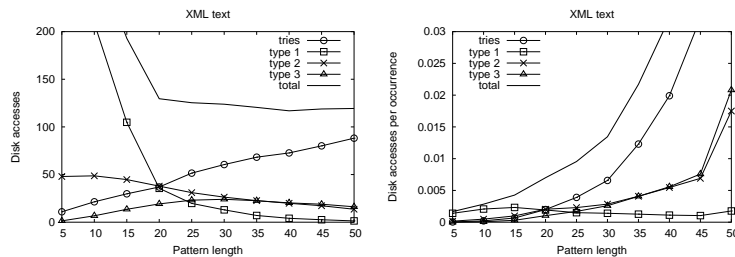
Fig. 3. Cost for the different parts of the LZ-index search algorithm.

beled "`tries`"), the different types of occurrences, and the total cost ("`total`"). The total cost can be decomposed in three components: a part linear on m (trie traversal), a part linear in occ (type 1), and a constant part (type 2 and 3).

5 Conclusions and Further Work

The LZ-index [12] can be adapted to work on secondary storage, requiring up to $8uH_k + o(u \log \sigma)$ bits of space, for any $k = o(\log_\sigma u)$. In practice, this value is about 1.4–2.3 times the text size, including the text, which means 39%–65% the space of *String B-trees* [7]. Saving space in secondary storage is important not only by itself (space is very important for storage media of limited size, such as CD-ROMs), but also to reduce the high seek time incurred by a larger index, which usually is the main component in the cost of accessing secondary storage, and is roughly proportional to the size of the data.

Our index is significantly smaller than any other practical secondary-memory data structure. In exchange, it requires more disk accesses to locate the pattern occurrences. For XML text, we are able to report (depending on the pattern length) about 597 occurrences per disk access, versus 3,449 occurrences reported by *String B-trees*. For English text (WSJ file from [20]), the numbers are 63 vs. 1,450 occurrences per disk access. In many applications, it is important to find quickly a few pattern occurrences, so as to find the remaining while processing the first ones, or on user demand (think for example in Web search engines). Fig. 3 (left, see the line "`tries`") shows that for $m = 5$ we need about 11 disk accesses to report the first pattern occurrence, while *String B-trees* need about 12. If we only want to count the pattern occurrences, the space can be dropped to $6uH_k + o(u \log \sigma)$ bits; in practice 1.0–1.7 times the text size. This means 29%–48% the space of *String B-trees*, with a slowdown of 2–4 in the time.

We have considered only number of disk accesses in this paper, ignoring seek times. Random seeks cost roughly proportionally to the size of the data. If we multiply number of accesses by size of the indexes, we get a very rough idea of the overall seek times. We can see that the smaller size of our LZ-index should favor it in practice. For example, it is very close to *String B-trees* for counting on XML and $m = 5$ (Fig. 1). This product model is optimistic, but counting only accesses is pessimistic.

As future work we plan to handle dynamism and the direct construction on secondary storage, adapting the method of [16] to work on disk.

References

1. Apostolico, A.: The myriad virtues of subword trees. In: Combinatorial Algorithms on Words. NATO ISI Series, Springer-Verlag (1985) 85–96
2. Kurtz, S.: Reducing the space requeriments of suffix trees. Softw. Pract. Exper. **29**(13) (1999) 1149–1171
3. Manzini, G.: An analysis of the Burrows-Wheeler transform. JACM **48**(3) (2001) 407–430
4. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys (2007) To appear.
5. Ferragina, P., Manzini, G.: Indexing compressed texts. JACM **54**(4) (2005) 552–581
6. Moura, E., Navarro, G., Ziviani, N., Baeza-Yates, R.: Fast and flexible word searching on compressed text. ACM TOIS **18**(2) (2000) 113–139
7. Ferragina, P., Grossi, R.: The String B-tree: a new data structure for string search in external memory and its applications. JACM **46**(2) (1999) 236–280
8. Ferragina, P., Grossi, R.: Fast string searching in secondary storage: theoretical developments and experimental results. In: Proc. SODA. (1996) 373–382
9. Clark, D., Munro, J.I.: Efficient suffix trees on secondary storage. In: Proc. SODA. (1996) 383–391
10. Mäkinen, V., Navarro, G., Sadakane, K.: Advantages of backward searching — efficient secondary memory and distributed implementation of compressed suffix arrays. In: Proc. ISAAC. (2004) 681–692
11. Sadakane, K.: Succinct representations of *lcp* information and improvements in the compressed suffix arrays. In: Proc. SODA. (2002) 225–232
12. Navarro, G.: Indexing text using the Ziv-Lempel trie. J. of Discrete Algorithms **2**(1) (2004) 87–114
13. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE TIT **24**(5) (1978) 530–536
14. Kosaraju, R., Manzini, G.: Compression of low entropy strings with Lempel-Ziv algorithms. SIAM J.Comp. **29**(3) (1999) 893–911
15. Arroyuelo, D., Navarro, G., Sadakane, K.: Reducing the space requirement of LZ-index. In: Proc. CPM. (2006) 319–330
16. Arroyuelo, D., Navarro, G.: Space-efficient construction of LZ-index. In: Proc. ISAAC. (2005) 1143–1152
17. Munro, I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM J.Comp. **31**(3) (2001) 762–776
18. Munro, I.: Tables. In: Proc. FSTTCS. LNCS 1180 (1996) 37–42
19. Morrison, D.R.: Patricia – practical algorithm to retrieve information coded in alphanumeric. JACM **15**(4) (1968) 514–534
20. Harman, D.: Overview of the third text REtrieval conference. In: Proc. Third Text REtrieval Conference (TREC-3), NIST Special Publication 500-207 (1995)
21. Baeza-Yates, R., Barbosa, E.F., Ziviani, N.: Hierarchies of indices for text searching. Inf. Systems **21**(6) (1996) 497–514
22. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. SIAM J.Comp. **22**(5) (1993) 935–948
23. González, R., Navarro, G.: Compressed text indexes with fast locate. In: Proc. of CPM'07. LNCS (2007) To appear.