Hybrid Indexing Revisited*

Héctor Ferrada† Dominik Kempa† Simon J. Puglisi†

Abstract

Hybrid indexing is a recent approach to text indexing that allows the space-usage of conventional text indexes (e.g., suffix trees, suffix arrays, FM-indexes) to scale well with the text size, n, when z, the size of the Lempel-Ziv parsing of the text, is small relative to n. The price for this improved scalability is that an upper bound M on the pattern length that can be searched for must be declared at index construction time. Because the size of the resulting index contains an $O(Mz)$ term, M must be kept reasonably small, though it has been shown that $M \approx 100$ leads to acceptable performance in some genomic applications. However, despite its promise, the practical performance of hybrid indexing relative to other compressed index data structures is poorly understood. This paper addresses that need, detailing experiments that show hybrid indexing — when carefully implemented — to be significantly smaller and faster than alternative approaches on a broad range of data of different levels of compressibility. We also describe practical extensions to hybrid indexing that obviate the restriction on M, supporting search for patterns of arbitrary length.

1 Introduction

Compressed text indexes [26, 25] are data structures for pattern matching that require significantly less space than traditional indexes, such as the suffix tree and suffix array [23], but have similar search times. Given a text T, a compressed index for T takes space similar to the size of T when compressed, and allows all the occurrences of a given pattern P to be located in time proportional to the length of P.

Compressed and succinct text indexes have had an especially deep and important impact in the field of genomics [21], in particular for *short-read alignment*, a fundamental and data intensive task at the forefront of modern medical and evolutionary biology. All serious short read aligners (see e.g., [19, 20, 6]) implement a kind of compressed suffix array [23] called the FM-index [5], usually in $n \log \sigma$ bits of space ($2n$ bits on DNA data), or even near nH_k bits, where H_k is the kth-order empirical entropy of the underlying data[1].

However, as genome sequencing becomes cheaper, and the number of genome sequences in genomic databases grows, indexes of linear size (even those taking around nH_k bits) quickly become too large. Because two genome sequences of any two individuals of the same species share many long, common substrings these modern genomic databases are highly compressible. Although many alternative compressed indexes with significantly better space-usage than the FM-index have been designed [22, 17, 2, 1], they are not generally in use by bioinformaticians, for what are essentially cultural reasons: implementations of the $n \log \sigma$- and nH_k-space indexes are mature, well-established and trusted.

Recently, in an attempt to address this standoff, Ferrada et al. [3] described *hybrid indexing* — an algorithmic technique by which *any* conventional pattern matching index (including any read aligner) can be made to scale to large, highly compressible collections via means of the Lempel-Ziv (LZ77) parsing [30, 14, 11], a method from data compression (we give a formal definition shortly). In particular, given an upper bound M on the searchable pattern length, the first step of hybrid indexing is to obtain a *filtered string* consisting of the concatenation of the M-length substrings to the left and right of each LZ77 phrase boundary. The next step is to build a conventional index (e.g., an FM-index) for the filtered string, and a 2D two-sided range reporting structure on the start and end points of each LZ77 phrase. The conventional index is used to find so-called *primary* occurrences of the pattern, which are then mapped using the structure of the LZ77 parse to locate the remaining, so-called *secondary*, pattern occurrences that are contained inside LZ77 phrases (we give a more precise description of hybrid indexing in Section 3).

Valenzuela [28] has since demonstrated hybrid indexing to be very effective in practice for indexing massive genomic data sets (in the terabyte range), and the technique now underlies tools for detecting genomic variants in pangenomic data [29]. However, Valenzuela's index is

*This research was partially supported via the Academy of Finland through grants 294143 and 2845984.

†Helsinki Institute for Information Technology (HIIT), Department of Computer Science, University of Helsinki, Finland.

[1] H_k is a lower bound on the compression achievable by any statistical compressor that models symbol probabilities as a function of the k letters preceding it in the text.

tightly coupled to the DNA alphabet and still carries the restriction that the maximum searchable pattern length is M, meaning it cannot be applied to long, so-called *third generation* DNA sequence reads (see, e.g. [24]) that are gaining wider use.

1.1 Our Contribution. The main results in this paper are as follows.

1. We provide a careful, 64-bit implementation of the hybrid index capable of indexing strings on any integer alphabet (not only DNA). We show experimentally that hybrid indexing significantly outperforms the best alternative compressed indexes in terms of space and time on a broad range of data of different level of compressibility.
2. We describe extensions to the basic hybrid indexing scheme to support efficient search for patterns of arbitrary length $m > M$.

The remainder of this paper is structured as follows. The next section reviews basic data structural tools on which hybrid indexing depends. Section 3 then gives an overview of hybrid indexing, as described by Ferrada et al. [3]. We then describe our implementation of this basic scheme in Section 4. Section 5 details our new method for longer patterns. Section 6 presents and discusses the results of our experiments comparing the hybrid index to state-of-the-art compressed indexes. Conclusions and reflections are offered in Section 7.

2 Preliminaries

The Lempel-Ziv (LZ) parsing [30] is a data compression method based on the concept of *longest previous factor* (LPF). The LPF at position i in a string X is a pair (p_i, ℓ_i) such that, $p_i < i$, $\mathsf{X}[p_i..p_i + \ell_i - 1] = \mathsf{X}[i..i + \ell_i - 1]$ and $\mathsf{X}[p_i + \ell_i] \neq \mathsf{X}[i + \ell_i]$. In other words, $\mathsf{X}[i..i + \ell_i - 1]$ is the longest prefix of $\mathsf{X}[i..n]$ which also occurs at some position $p_i < i$ in X. Note that there may be more than one potential p_i. Depending on application, it can be advantageous to ensure that p_i is either the leftmost (i.e., first) or rightmost (i.e., most recent) previous occurrence. We return briefly to this idea in Section 4.

The LZ77 factorization (or LZ77 parsing) of a string X is a greedy, left-to-right parsing of X into longest previous factors. More precisely, if the j^{th} LZ factor (or *phrase*) in the parsing is to start at position i, then we output (p_i, ℓ_i) (to represent the j^{th} phrase), and then the $(j+1)^{\text{th}}$ phrase starts at position $i + \ell_i$. The exception is the case $\ell_i = 0$, which happens iff $\mathsf{X}[i]$ is the leftmost occurrence of a symbol in X. In this case we output $(\mathsf{X}[i], 0)$ (to represent $\mathsf{X}[i..i]$) and the next phrase starts at position $i + 1$. When $\ell_i > 0$, the substring $\mathsf{X}[p_i..p_i + \ell_i - 1]$ is called the *source* of phrase $\mathsf{X}[i..i + \ell_i - 1]$. We denote the number of phrases in the LZ77 parsing of X by z.

The above description of LZ77 allows $p_i + \ell_i > i$ and so $\mathsf{X}[i..i + \ell_i - 1]$ and $\mathsf{X}[p_i..p_i + \ell_i - 1]$ can overlap each other. This definition of LZ77 is sometimes called *self-referential* in that it allows phrases to overlap their sources in X. Sometimes it is preferable to produce *non-self-referential* parsings, in which it is enforced that $p_i + \ell_i$ is not larger than i.

For the example string $\mathsf{X} = zzzzzapzap$, the (self-referential) LZ77 factorization produces the pairs:

$$(z, 0), (0, 4), (a, 0), (p, 0), (4, 3).$$

3 The Hybrid Index

As Kärrkäinen and Ukkonen observed [13], for strings P and T, the LZ77 parsing of T, allows us to talk about two distinct types of occurrence of P in T: those that cross LZ phrase boundaries, and those that do not. We call the first type of occurrences *primary* occurrences and the remaining ones (which must necessarily be completely contained inside LZ phrases) *secondary* occurrences. The hybrid index [3] reports the primary and secondary occurrences of a query pattern using separate structures, which we now review (see also [7])[2]

3.1 Finding Primary Occurrences. For a given upper bound M on pattern length, let T_M be the string containing the characters of T within distance M of their nearest LZ phrase boundaries; characters not adjacent in T are separated in T_M by a special character # not in the alphabet of T. The crucial observation is that, for any substring of T with length at most M that contains the first occurrence of a distinct character in T, or crosses a phrase boundary in the LZ parse of T, there is a corresponding and equal substring in T_M.

Ferrada et al. [3] store a conventional patten matching index[3] I_M on T_M. The only assumption about I_M is that it can handle searches for pattern lengths up to M. Since T_M consists of at most $2M$ characters for each LZ phrase, if T is highly repetitive and M is not too large, I_M is smaller than a conventional index on T.

In order to map between positions in T and positions in T_M, two data structures L and L_M are stored. L is a sorted list containing the positions of the first character of each LZ phrase of T and L_M is the sorted list containing the positions of the corresponding characters

[2] Importantly, approximate pattern matching can also be supported if we increase M by K, the maximum number of errors/mismatches allowed. In this paper, for clarity, we consider only exact pattern matching.

[3] For example, an FM-index.

in T_M. Moreover, if $T[i]$ is the first occurrence of a distinct character in T and $T_M[j]$ is the corresponding character in T_M, then j is *marked* in L_M.

Given the endpoints i and j of a substring $T_M[i..j]$ of T_M that does not include any occurrences of #, L_M can be used to determine whether the corresponding substring $T[i'..j']$ of T contains the first occurrence of a distinct character in T or crosses a phrase boundary in the LZ77 parse of T. More precisely, binary search is used on L_M to find the successor of i, say $L_M[s]$. There are then three cases to consider:

- if $i < L_M[s] \leq j$ then $T[i'..j']$ crosses a phrase boundary;
- if $i \leq j < L_M[s]$ then $T[i'..j']$ neither contains the first occurrence of a distinct character nor crosses a phrase boundary;
- if $i = L_M[s] \leq j$ then $T[i'..j']$ contains the first occurrence of a distinct character or crosses a phrase boundary if and only if $L_M[s]$ is marked or $L_M[s+1] \leq j$.

Also, if $T[i'..j']$ contains the first occurrence of a distinct character or crosses a phrase boundary, then $i' = L[s] - L_M[s] + i$ and $j' = i' + j - i$. In this way L and L_M are used to map positions from T_M to positions in T.

Given a query pattern $P[1..m]$ with $m \leq M$, we use I_M, L and L_M to find all primary occurrences in T. First, we query I_M to find all occurrences of P in T_M. We use binary search on L_M, as described above, to determine which of these occurrences correspond to primary occurrences in T. Finally, we use L and L_M, as described above, to find the positions of those primary occurrences in T.

3.2 Finding Secondary Occurrences. As noted above, by definition, any secondary occurrence is completely contained in some LZ phrase. Moreover, a phrase contains a secondary occurrence if and only if the source of that phrase contains an earlier occurrence (primary or secondary). It follows then that each secondary occurrence is an exact copy of some primary occurrence and once we have found all the primary occurrences we can locate all the secondary occurrences from the structure of the LZ77 parse.

To do this, for each primary occurrence $T[\ell..r]$, we find each phrase $T[i..j]$ that has source $T[i'..i'+j-i]$ such that $i' \leq \ell \leq r \leq i' + j - i$, i.e., $T[i'..i' + j - i]$ includes $T[\ell..r]$. Notice $T[\ell'..r'] = T[\ell..r]$, where $\ell' = i + \ell - i'$ and $r' = \ell' + r - \ell$. We record $T[\ell'..r']$ as a secondary occurrence and recurse on it (as it may lead to further secondary occurrences).

In order to efficiently locate all sources covering an occurrence, we use a data structure for 2-sided range reporting. Specifically, we store points at (i', j') for every phrase's source $T[i'..j']$, on an $n \times n$ grid. With each point we associate satellite data indicating the start of the phrase: if a phrase $T[i..j]$ is encoded as (p_i, ℓ_i) by LZ77, then there is a point on the grid at $(p_i, p_i + \ell_i - 1)$ with satellite data i.

The overall time taken by the hybrid index to find a pattern of length $m \leq M$ is $O(t_{\mathsf{search}}(m) + \mathsf{occ} \log\log n)$ time, where $t_{\mathsf{search}}(m)$ is the time to search for the pattern using I_M. The factor $\log\log n$ comes from the predecessor query incurred by each query to the 2-sided range reporting structure. In practice we use a very simple predecessor data structure described in Section 4.1.

4 Implementation Details

In this section we describe details of our implementation of hybrid index. The first non-trivial implementation detail is that in our implementation we employ the idea described in [28] to reduce the number of LZ phrases. Namely, for any maximal sequence of adjacent phrases where each phrase has length $\leq M$, we merge them into one *superphrase*. Let z' denotes the size of the resulting parsing (containing normal phrases and superphrases). By t we denote the number of normal phrases left after the merging. To mark superphrases we use a bitvector $B_s[1..z']$. We store this with the hybrid bitvector representation of Kärkkäinen et al. [10].

The two lists, L and L_M, used to map between the original and reduced text, are not stored explicitly, but as a sequence of differences between consecutive elements augmented with original values evenly sampled. The resulting lists stored using this encoding need $O(z' \log g_{\max} + z' \log n/s)$ bits where $g_{\max}$ is the maximum difference and s is the sampling rate, and allow random access to the arrays using $O(s)$ time. In practice we choose s adaptively, depending on $g_{\max}$. Both L and L_M are further augmented with the predecessor data structure used for navigating between T and T_M.

Arguably most complex component of the index is the 2D two-sided range reporting data structure. Given a $n \times n$ grid we store t points, each with some satellite data, such that for any query point (which is not necessarily one of the represented points) we want to list all points with the smaller x- and larger y-coordinate. To achieve this we keep all points sorted by the x-coordinate and build a range maximum query (RMQ) data structure on y-coordinates of the points in this order. Furthermore, we augment the list of points with the predecessor data structure on the x-coordinates. To answer the query we first locate the index of x-predecessor p' of the query

point p. We then use the RMQ data structure to locate the point p'' with the maximum y-coordinate in the x-range from 1 to the x-coordinate of p'. If the point lies in the sought range we list it and then recurse on the two subrange to the left and right of p''.

To implement RMQ we use a data structure of [4]. The predecessor data structure is described next.

4.1 Predecessor Data Structure. Let $A[1..k]$ be a non-decreasing sequence of k integers from the range $[0, u]$. To support predecessor queries over A we use the following simple approach.

Consider a partition of the universe $[0, u]$ into blocks of size $b = 2^h$ for some $h > 0$. We store an array $A_s[0..\lfloor u/b \rfloor + 1]$, where for $i = 0, \ldots, \lfloor u/b \rfloor$, $A_s[i]$ stores the largest $j \in [1, k]$ such that $A[j] < ib$. Assume now that x is the argument of the predecessor query and let $i_x = x/b$. We first compute $j_\ell = A_s[i_x - 1]$ and $j_r = A_s[i_x]$. The task is then to find the predecessor of x in the range $A[j_\ell..j_r]$. We do a binary search over A (which is stored explicitly) that continues until the range becomes smaller than some predefined threshold τ. In practice we found a value $\tau = 256$ to give good performance.

This method works well in our application because the universe size is bounded by the length of the string n, and we can choose the sampling period b depending on the ratio n/z to keep the space overhead within $O(z)$.

5 Handling Long Patterns

When the pattern length $m \leq M$, we locate occurrences in the same way as the original hybrid index. For the case $m > M$ we process the pattern with a heuristic method described in this section. For ease of exposition we assume that $m = kM$ for some integer $k \geq 2$.

To support finding longer patterns we add to our index an additional array $Sr[1..t]$, where $Sr[i]$ stores the starting position of the source of the i^{th} non-literal LZ phrase. By carefully coupling Sr to the 2D range reporting data structure described earlier, it can be stored in just $O(t \log t)$ bits (with constant time access to its elements).

We begin by splitting the pattern P into k segments, each of length M. Let P_i denote the i^{th} segment of the pattern, so that $P = P_1 P_2 \ldots P_k$. For each segment P_i, we use I_M to locate all the occurrences of P_i in the filtered text. Because these occurrences are in T_M they are either primary or occur inside a literal phrase. We map each of them to their corresponding positions in T. Let $A_i[1..n_i]$ be an array containing these positions for segement P_i in increasing order.

With the sorted arrays A_i thus obtained, we process each of them in turn, starting with A_1. For each $x_{1j} \in A_1, 1 \leq j \leq n_1$, we need to determine whether P_2 occurs at positon $u_2 = x_{1j} + M$, P_3 occurs at positon $u_3 = x_{1j} + 2M$, and in general whether P_i occurs at position $u_i = x_{1j} + ((i-1)M)$, for all $i \in [2..k]$. It is easy to see that, when this happens, P occurs at position x_{1j} in T.

The easy case is when we have $u_2 \in A_2$, $u_3 \in A_3$, ..., $u_k \in A_k$. Clearly, this happens only when every segment P_i occurs as a primary occurrence or is inside a literal phrase. More generally we will have $u_i \in A_i$ only for some i. For those other segments, i.e., those segments P_i for which $u_i \notin A_i$, we need to determine if P_i occurs as a secondary occurrence at position u_i. This is done recursively as follows. We determine the LZ phrase covering position u_i and via Sr obtain the source of that phrase. Denote by s the start of that source and let d be the offset of u_i from the start of its covering phrase. If text positions $[s + d..s + d + M]$ are inside the filtered text then we have that $(s + d) \in A_{k'}$ (for some k') and P_i therefore occurs (as a secondary occurrence) at position u_i. If positions $[s + d..s + d + M]$ are not in the filtered text, then we recurse, determining the LZ phrase covering position $s + d$, looking up its source with Sr, and so on. To ensure the chain of phrases/sources that need to be followed is shorter in practice, we make use of the leftmost LZ parsing (see Section 2).

Having processed A_1, we then process each position in the other arrays $A_2, A_3, ..., A_k$ in turn, taking into account that we must add or subtract the appropriate multiple of M to each position in order to obtain the correct u values.

We note that this approach for longer patterns introduces a new tradeoff between index size and pattern matching time. The tension is to find an M that keeps the filtered text T_M reasonably small, but at the same time is large enough so that patterns are not split into too many segments (because for each one of these $k = m/M$ segments we have to perform a locate operation with I_M, and then sort and intersect the results).

6 Experimental Results

6.1 Setup. We performed experiments on a 2.1 GHz Intel Xeon E7-4830 CPU equipped with 1.2 TiB of DDR3 main memory. The machine had no other significant CPU tasks running and only a single thread of execution was used. The OS was Linux (Ubuntu 16.04, 64bit) running kernel 4.8.0. All programs were compiled using `g++` version 5.4.0 with `-O3 -DNDEBUG -march=native` options. All given runtimes are real (wallclock) times. The data sets used for experiments are described in Table 1.

6.2 Tested Indexes. As baselines for our hybrid index, denoted **HI**[4], we used the following state-of-the-art indexes in our experiments:

- **RLCSA** (version 05-2016):[5] a member of the compressed suffix array family based on run-length encoding [22, 27]. It is currently one of the fastest and simultaneously smallest indexes for highly repetitive data.
- **LZI**:[6] the LZ77-based index of Kreft and Navarro [17]. We use the latest and most optimized version of the code from [1].
- **LZEI**: another index of Kreft and Navarro [17]. The main difference compared to LZI is that LZEI is based on the modification of LZ77 (called LZend, and introduced in [16, 15]), that restricts the sources of all phrases to end at another phrase boundary.
- **FMI**: the most recent variant of the FM-index by Gog et al. [8] based on the fixed-block boosting technique [12] with the hybrid bivector [10] as the underlying bitvector implementation. This is fastest (and simultaneously smallest) FM-index we are aware of. The implementation is part of the SDSL library [9], which contains state-of-the-art implementations of compressed data structures.[7]

The FM-index inside our HI is the same FMI as above. This ensures the comparison of our hybrid index with the standalone FM-index is fair and highlights the improvements achievable with hybrid indexing.

6.3 Experiments. In the first experiment we compare the performance of our hybrid index to that of the baseline indexes using locate queries. To test each index we randomly extract 3 000 patterns from the text and report the average time per occurrence of the pattern. For brevity, in this experiment we fix a pattern length $m = 20$ for non-repetitive texts in our data set and $m = 80$ for highly repetitive data. We believe these values to be representative of most practical applications, however we also investigate the effect of m on the query performance in the next experiment. The results are given in Fig. 1.

For highly repetitive data in nearly all cases the hybrid index significantly outperforms all other indexes both in time and space. For the influenza testfile the index matches the space of other indexes (this is not clearly visible in the graph, however, as we only used the suffix array sampling of 32 and 64 for HI, while FMI uses higher values) while still being the fastest. The results for non-repetitive data show that the index achieves a smooth transition: as the data becomes less repetitive, the hybrid index essentially becomes the FM-index (that is, of course, the case here since we choose to use FMI on the filtered text; in general the transition occurs towards the chosen index type). A particularly attractive property of the hybrid index is that it always performs well, irrespective of the type of data. This is in contrast to other indexes, for which there always exists an input on which its performance decreases dramatically or the resulting index is orders of magnitude bigger than other indexes.

Input	$\lvert\Sigma\rvert$	$n/2^{20}$	n/z
dna	16	386	15.75
proteins	27	512	13.04
english	226	512	14.60
sources	230	202	18.17
cere	5	440	271.2
influenza	15	148	200.9
kernel	160	247	324.7
einstein	139	446	5136.7

Table 1: Files used in the experiments. The files are from the Pizza & Chili standard and repetitive corpus (http://pizzachili.dcc.uchile.cl/). The value of n/z is the average length of phrase in the LZ77 parsing.

In the second experiment we examine the performance of indexes with varying value of m, the pattern length. Inevitably, when the pattern length m exceeds the threshold M, the performance of hybrid index starts to deteriorate. The previous experiment (where all hybrid indexes used either $M = 20$ or $M = 40$) demonstrates that for highly repetitive inputs, the performance of hybrid index is very good even for $m = 80$. Here we study this effect in more depth and compare to other indexes.

Fig. 2 compares the performance of FMI, LZI (the space-efficient variant of the LZ-based index), and HI (the hybrid index) with $M = 20$ on four testfiles: two from the standard corpus, and two from the repetitive corpus. For the hybrid index, the trend for non-repetitive files depends on the structure of occurrences: while for the proteins testfile there is essentially no slowdown, for the dna testfile the performance deteriorates dramatically already for $m = 40$. On the other hand, the performance of the hybrid index on highly repetitive files deteriorates very slowly. Lastly, we point out that LZI exhibits a notable slowdown as m increases, which is explained

[4]The implementation is available at: https://github.com/hferrada/HydridSelfIndex

[5]Available at: https://jltsiren.kapsi.fi/rlcsa

[6]Code available from: https://github.com/migumar2/uiHRDC/

[7]The library is available from: https://github.com/simongog/sdsl-lite

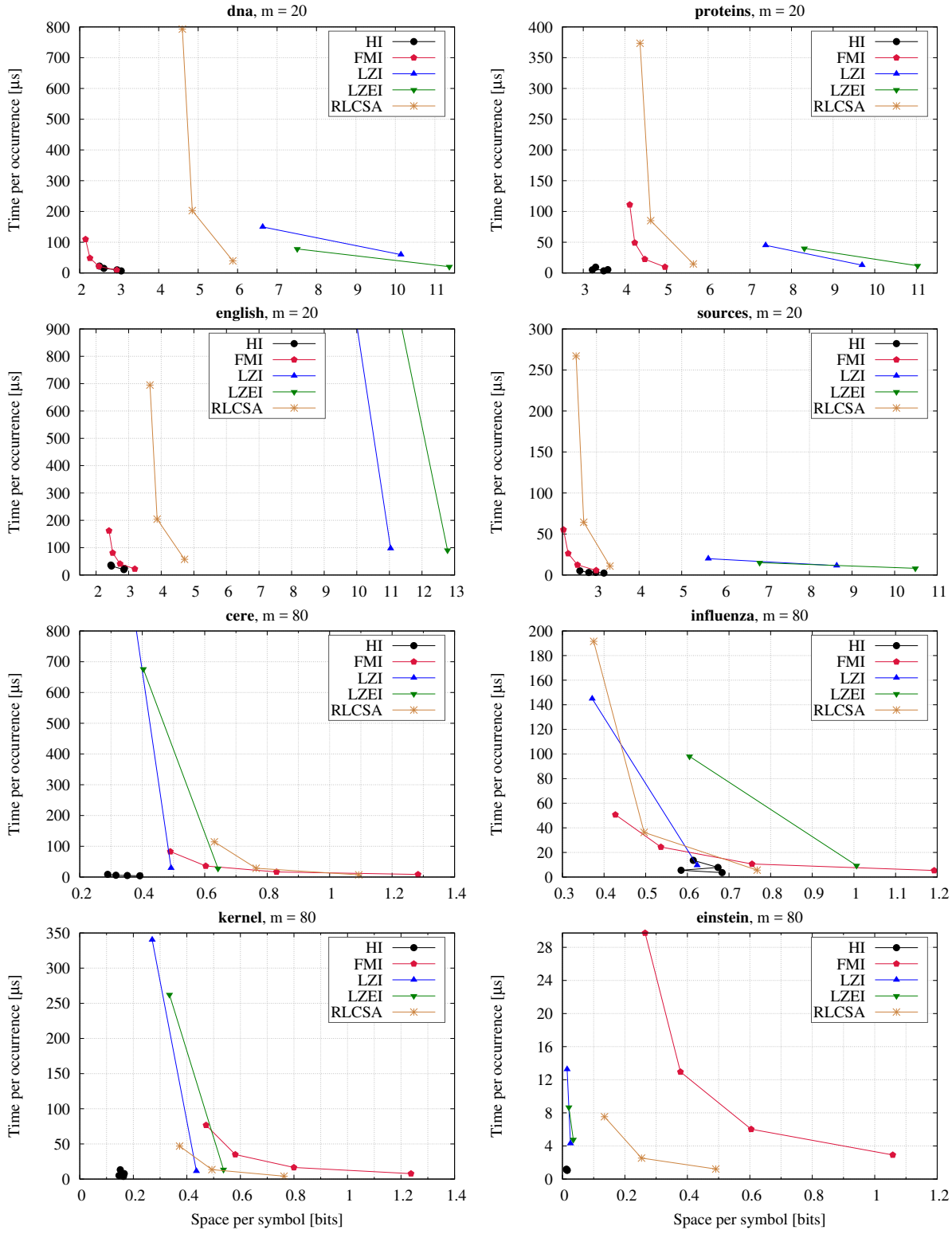

Figure 1: Time/space tradeoffs of the indexes for locate queries. For non-repetitive collections we set $m = 20$ and for highly-repetitive we use $m = 80$. The four data points for FMI correspond to suffix array sampling rate of $\{32, 64, 128, 256\}$. In HI we use $\{(20, 64), (20, 32), (40, 64), (40, 32)\}$, with each pair corresponding to, respectively, M and SA sampling rate of the FMI of the filtered text. The data points for RLCSA correspond to different block size and sampling of the suffix arrays: $\{(256, 512), (128, 256), (32, 128)\}$. Finally, for LZ-based indexes we used the fastest and the smallest variants.

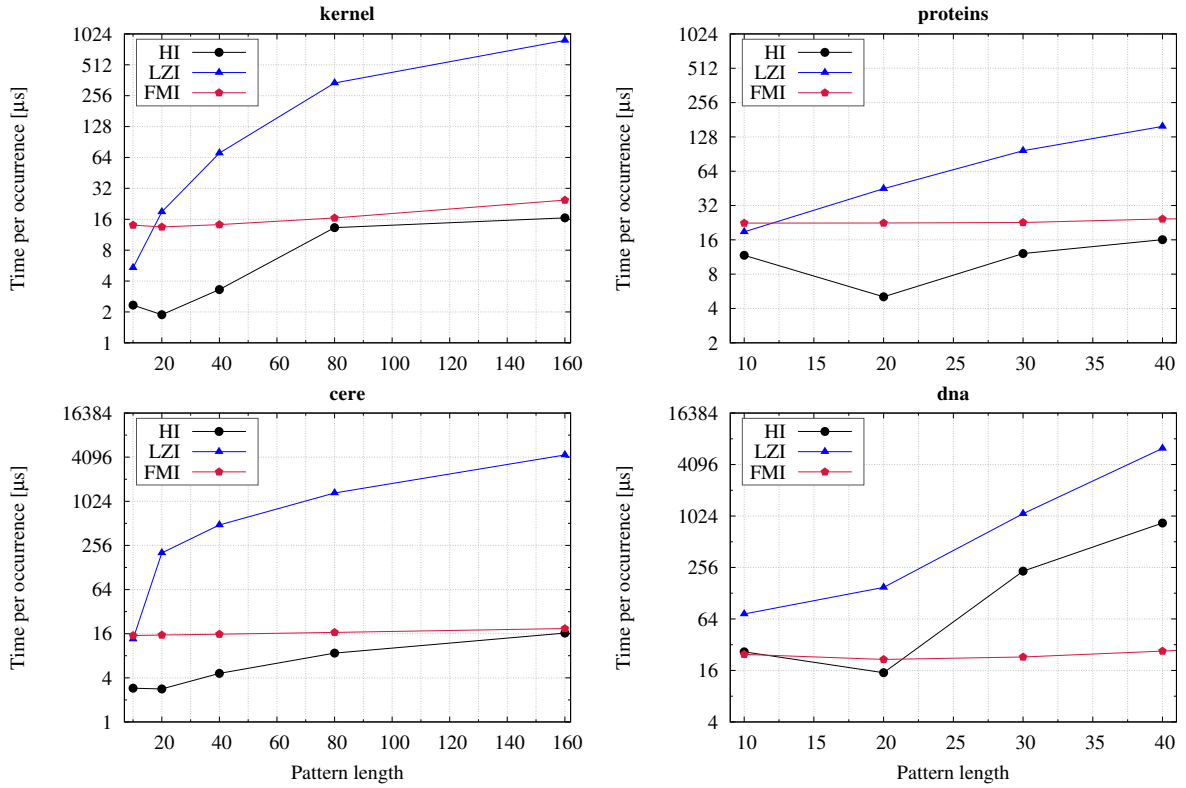

Figure 2: Performance of locate queries for different pattern lengths. HI uses threshold $M = 20$. The suffix array sampling of FMI is 64 (both for the standalone index as well as for the FMI used in HI). The experiments use the space-efficient variant of LZ-index.

by the non-linear dependence of its query time on the pattern length.

7 Conclusions and Future Work

We have provided the first proper experimental analysis of hybrid indexing that compares it to other compressed indexes designed for highly repetitive text collections. Our results show that hybrid indexing is a versatile approach, and usually the best alternative among currently available methods, irrespective of the compressibility of the underlying data. We have also extended the functionality of hybrid indexes to support pattern matching for arbitrary length patterns.

The FM-index of the filtered text is a time and space bottleneck in our index, and alleviating this is an avenue for future work. We note that we are not compelled to use an FM-index — any fast, small index for relatively incompressible text (an inherent property of the filtered text) will do. Another potentially fruitful (and relatively straightforward) direction is to use alternatives to the LZ parsing — for example, relative Lempel-Ziv parsing [18] — that are more easily constructed in external memory and parallel settings.

References

[1] Francisco Claude, Antonio Fariña, Miguel A. Martínez-Prieto, and Gonzalo Navarro. Universal indexes for highly repetitive document collections. *Information Systems*, 61:1–23, 2016.

[2] Huy Hoang Do, Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Fast relative Lempel-Ziv self-index for similar sequences. *Theoretical Computer Science*, 532:14–30, 2014.

[3] Héctor Ferrada, Travis Gagie, Tommi Hirvola, and Simon J. Puglisi. Hybrid indexes for repetitive datasets. *Philosophical Transactions of the Royal Society A*, 372, 2014.

[4] Héctor Ferrada and Gonzalo Navarro. Improved range minimum queries. *Journal of Discrete Algorithms*, 43:72–80, 2017.

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

[6] Paul Flicek and Ewan Birney. Sense from sequence reads: Methods for alignment and assembly. *Nature Methods*, 6(11):6–12, 2009.

[7] Travis Gagie and Simon J. Puglisi. Searching and indexing genomic databases via kernelization. *Frontiers in Bioengineering and Biotechnology*, 3:Article 12, 2015.

[8] Simon Gog, Juha Kärkkäinen, Dominik Kempa, Matthias Petri, and Simon J. Puglisi. Faster, Minuter. In *Proc. Data Compression Conference (DCC)*, pages 53–62. IEEE, 2016.

[9] Simon Gog and Matthias Petri. Optimized succinct data structures for massive data. *Software, Practice and Experience*, 44(11):1287–1314, 2014.

[10] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Hybrid compression of bitvectors for the FM-index. In *Proc. Data Compression Conference (DCC)*, pages 302–311. IEEE, 2014.

[11] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lempel-Ziv parsing in external memory. In *Proc. Data Compression Conference (DCC)*, pages 153–162. IEEE, 2014.

[12] Juha Kärkkäinen and Simon J. Puglisi. Fixed block compression boosting in FM-indexes. In *Proc. 18th Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 7024, pages 174–184. Springer, 2011.

[13] Juha Kärkkäinen and Esko Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd South American Workshop on String Processing (WSP)*, pages 141–155, 1996.

[14] Dominik Kempa and Simon J. Puglisi. Lempel-Ziv factorization: Simple, fast, practical. In *Proc. 15th Meeting on Algorithm Engineering and Experiments (ALENEX)*, pages 103–112. SIAM, 2013.

[15] Sebastian Kreft and Gonzalo Navarro. LZ77-like compression with fast random access. In *Proc. Data Compression Conference (DCC)*, pages 239–248. IEEE, 2010.

[16] Sebastian Kreft and Gonzalo Navarro. Self-indexing based on LZ77. In *Proc. 22nd Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 6661, pages 41–54. Springer, 2011.

[17] Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

[18] Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *Proc. 17th Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 6393, pages 201–206. Springer, 2010.

[19] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[20] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. SOAP2: An improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[21] Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing.* Cambridge University Press, 2015.

[22] Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

[23] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[24] Eugene W. Myers. A history of DNA sequence assembly. *it - Information Technology*, 58(3):126–132, 2016.

[25] Gonzalo Navarro. *Compact Data Structures - A Practical Approach.* Cambridge University Press, 2016.

[26] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article 2, 2007.

[27] Jouni Sirén. *Compressed full-text indexes for highly repetitive collections.* PhD thesis, University of Helsinki, 2012.

[28] Daniel Valenzuela. CHICO: A compressed hybrid index for repetitive collections. In *Proc. 15th Symposium on Experimental Algorithms (SEA)*, LNCS 9685, pages 326–338. Springer, 2016.

[29] Daniel Valenzuela, Niko Välimäki, Esa Pitkänen, and Veli Mäkinen. On enhancing variation detection through pan-genome indexing. *bioRxiv*, 2015.

[30] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.