Information Systems

ER-index: a referential index for encrypted genomic databases

--Manuscript Draft--

Manuscript Number:	INFOSYS-D-19-00066
Article Type:	Research paper
Keywords:	Data compression; Data encryption; Genomic databases; Full-text index
Corresponding Author:	Giovanni Schmid, Ph.D ITALY
First Author:	Giovanni Schmid, Ph.D
Order of Authors:	Giovanni Schmid, Ph.D
	Ferdinando Montecuollo, Ph.D
Abstract:	Huge DBMSs storing genomic information are being created and engineerized for doing large-scale, comprehensive and in-depth analysis of human beings and their diseases. However, recent regulations like the GDPR require that sensitive data are stored and elaborated thanks to privacy-by-design methods and software. We designed and implemented ER-index, a new full-text index in minute space which was optimized for compressing and encrypting collections of genomic sequences, and for performing on them fast pattern-search queries. Our new index complements the E2FM-index, which was introduced to compress and encrypt collections of nucleotide sequences without relying on a reference sequence. When used on collections of highly similar sequences, the ER-index allows to obtain compression ratios which are an order of magnitude smaller than those achieved with the E2FM-index, but maintaining its very good search performance. Moreover, thanks to the ER-index multi-user and multiple-keys encryption model, a single index can store the sequences related to a population of individuals so that users may perform search operations only on the sequences to which they were granted access. The ER-index C++ source code plus scripts and data to assess the tool performance are available at: https://github.com/EncryptedIndexes/erindex.
Suggested Reviewers:	
Opposed Reviewers:	

Abstract

Huge DBMSs storing genomic information are being created and engineerized for doing large-scale, comprehensive and in-depth analysis of human beings and their diseases. However, recent regulations like the GDPR require that sensitive data are stored and elaborated thanks to privacy-by-design methods and software.

We designed and implemented ER-index, a new full-text index in minute space which was optimized for compressing and encrypting collections of genomic sequences, and for performing on them fast pattern-search queries. Our new index complements the E2FM-index, which was introduced to compress and encrypt collections of nucleotide sequences without relying on a reference sequence. When used on collections of highly similar sequences, the ER-index allows to obtain compression ratios which are an order of magnitude smaller than those achieved with the E2FM-index, but maintaining its very good search performance. Moreover, thanks to the ER-index multi-user and multiple-keys encryption model, a single index can store the sequences related to a population of individuals so that users may perform search operations only on the sequences to which they were granted access. The ER-index C++ source code plus scripts and data to assess the tool performance are available at: https://github.com/EncryptedIndexes/erindex.

Conflict of Interest

Declaration of interests

☐X The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Giovanni Schmid
National Research Council
Institute of High Performance Computing and Networking
Via P. Castellino 111
80131 Naples, Italy
email: giovanni.schmid@cnr.it
voice: +3981 6139529
fax: +3981 6139531

Naples, October 8 2019

Attention Prof. Dennis Shasha
Editor-in-Chief
Information Systems

Dear professor Shasha,

I am pleased to submit an original research article entitled "*ER-index: a referential index for encrypted genomic databases*" by Ferdinando Montecuollo and me, for consideration for publication in *Information Systems*.
We previously designed and implemented an encrypted full-text index in minute space called E2FM (Montecuollo, F. and Schmid, G. and Tagliaferri, R. (2017) E2FM: an encrypted and compressed full-text index for collections of genomic sequences, doi: 10.1093/bioinformatics/btx313), optimized for compressing and encrypting nucleotide sequence collections, and for performing fast pattern-search queries on them.
In this work we have followed a complementary approach which assumes the knowledge of a reference sequence, proposing an *encrypted referential* index for genomic databases, named ER-index. To build the index, our scheme pre-processes the genomic sequence via Relative Lempel-Ziv factorisation and leverages Salsa20 to encrypt the factorised blocks. In order to enable fast pattern search over the reference sequence, we designed a novel algorithm which makes use of two FM-trees and two mapping tables to represent the reference sequence. In addition, to support encryption without compromising neither compression nor pattern search performance, we introduced an extended and encrypted variant of the B+ tree data structure called encrypted B+ tree.
We implemented the ER-index in C++, and performed tests on three kinds of computing platforms in order to assess its compression and pattern search performance w.r.t. a state-of-the-art reference tool based on the Sdsl library (https://github.com/simongog/sdsl-lite). Our tests show that ER-index achieves a higher compression rate than the reference tool, while maintaining a very efficient search time.
We believe this is a remarkable result, especially if you take into account that ER-index offers in addition a built-in encryption. This last feature appears to be critical in the context of some emerging approaches in healthcare (e.g.; precision medicine and partecipative medicine), since recent regulations like the GDPR require that sensitive data are stored and elaborated thanks to privacy-by-design methods and software. In order to address such requirements, we built in the ER-index a multiple keys encryption model, so that a single index can store the sequences related to a population of individuals and users may perform search operations only on the sequences to which they were granted access.
Please note that the ER-index C++ source code plus scripts and data to assess the tool performance are available at: https://github.com/EncryptedIndexes/erindex. Moreover, we have some complementary experimental results, and instructions for setting up and using a ER-index database that could be published as supplementary material.

Thank you for your consideration.

Sincerely,
Giovanni Schmid

A new *encrypted referential* index for genomic databases, named ER-index, which can store collections of genomic sequences in less than 1/30 of their original space, and achieves very fast pattern search times.

A multiple keys encryption model was built in ER-index, so that a single index can store the sequences related to a population of individuals, and users may perform search operations only on the sequences to which they were granted access.

A detaliled description of the data structures and algorithms composing ER-index, whose source code is available at GitHub.

A comprehensive set of test to assess the performance w.r.t. the wavelet-tree FM-index, which show that ER-index achieves a higher compression rate than the reference tool, outperforming it in pattern search in case of short sequences.

ER-index: a referential index for encrypted genomic databases

Ferdinando Montecuollo[a], Giovannni Schmid[b,]

[a]*CRESSI, Università "Luigi Vanvitelli", Napoli, 80133 Italy*
[b]*ICAR, Consiglio Nazionale delle Ricerche, Napoli, 80131, Italy*

Abstract

Huge DBMSs storing genomic information are being created and engineerized for doing large-scale, comprehensive and in-depth analysis of human beings and their diseases. However, recent regulations like the GDPR require that sensitive data are stored and elaborated thanks to privacy-by-design methods and software.
We designed and implemented *ER-index*, a new full-text index in minute space which was optimized for compressing and encrypting collections of genomic sequences, and for performing on them fast pattern-search queries. Our new index complements the E^2FM-index, which was introduced to compress and encrypt collections of nucleotide sequences without relying on a reference sequence. When used on collections of highly similar sequences, the ER-index allows to obtain compression ratios which are an order of magnitude smaller than those achieved with the E^2FM-index, but maintaining its very good search performance. Moreover, thanks to the ER-index *multi-user and multiple-keys encryption model*, a single index can store the sequences related to a population of individuals so that users may perform search operations only on the sequences to which they were granted access.
The ER-index C++ source code plus scripts and data to assess the tool performance are available at: `https://github.com/EncryptedIndexes/erindex`.

1. Introduction

Predictive, preventive, precise and participatory medicine (*P4 medicine*, for short) are new approaches underpinned by genome sequencing that will soon be incorporated in our health systems. The advantages of these approaches for human health and wellbeing can be very significant according to [19]: by reshaping healthcare from reactive to proactive they indeed represent the main answer to the progression of "silent" chronic diseases, which are the leading

Email addresses: `montecuollo@gmail.com` (Ferdinando Montecuollo), `giovanni.schmid@cnr.it` (Giovannni Schmid)

cause of death, disability and diminished quality of life in the developed world, strongly impacting the economy of many countries.

However, such new approaches pose very big computational and security challenges. Data management in genomics is considered a "four-headed beast" due to the high computational costs concerning the lifecycle of a data set: acquisition, storage, distribution and analysis. The total amount of sequenced data doubles approximately every seven months, and [22] have calculated to be about 1 zetta-bases acquisition per year and from 2 to 40 exa-bytes of data storage per year the projected computational needs in 2025.
On the other hand, the storage of such huge amount of data raises concerns about privacy and security. Human genome projects were initially open access, since it was believed that there was no risk of identification of participants or donors, but this approach was overturned after [9] realized that data from individuals could be distinguished in *Genome Wide Association Studies* (GWAS) just using summary statistics. In this respect, the most effective tools for protecting data without compromising their usability are given by modern cryptography, which offers algorithms and protocols for accessing and managing data in much more complex use case scenarios than the classical two-party and "plaintext-or-ciphertext" settings. Nonetheless, the choice of the cryptographic algorithms and protocols and their implementation have to be taken seriously, otherwise the resulting system could be inefficient, and/or not adequately protected by advanced attacks (e.g. *ciphertext-chosen* attacks, *side-channel* attacks as defined in [14]) or emerging computing platforms (e.g. quantum computers).

1.1. Related work

[15] introduced the E^2FM-index, a full-text index in minute space which was optimized for compressing and encrypting nucleotide sequence collections, and for performing fast pattern-search queries on them, without the knowledge of a reference sequence. The E^2FM-index is particularly suitable for *metagenomics* or *de-novo discovery* applications: it occupies about 1/20 of the storage required by the input FASTA file, saving 95% of storage space, whereas the gap in pattern search performance due to encryption has no practical significance, being of the order of milliseconds in any case.

However, the E^2FM-index is not suitable to compress genomic sequences of multiple individuals given a reference sequence. Moreover, the E^2FM-index encryption model does not allow the use of multiple encryption keys for multiple sequences within the same index. Thus, a new index must be created for each new set of sequences whose access must be separately authorized. In turn, this can result in searching for patterns in several indexes, potentially slowing down search performance.

The E^2FM-index first processes data thanks to the Burrows-Wheeler transform (BWT) and the Move-to-front (MTF) transform, after which it compresses them with the RLE0 algorithm. The BWT approach does not seem so appropriate for referential compression as dictionary-based methods, thus we have adopted another compression strategy in the ER-index, based on LZ77 algorithm.

Lempel-Ziv methods are lossless, dictionary-based compression algorithms which replace repetitions in a string by using references of their previous occurrences. There are many variants, all derived from the two algorithms introduced by [25, 26] and named LZ77 and LZ78, respectively.

Most of the self-indexes inspired to the Lempel-Ziv parsing use LZ78, because the LZ78 factorization of a text has some interesting properties which allow to design efficient pattern search algorithms like that of [21]. LZ78 is faster but more complex than LZ77, since it constructs a dictionary which tends to grow and fill up during compression. Actually this happens all the time for big inputs like in our application scenario, and the common methods to overcome such issue (see [20]) do not permit to gain the most advantage from the high similarity of genomic sequences.

The first self-index based on LZ77 was presented by [10]: it offers good compression ratio and search performance, but its internal data structures were not designed to explicitly handle a collection of data items. This index also does not exploit the fundamental requisite of our application domain, that is the compression of genomic sequences relative to a reference sequence.

The first attempt to compress a collection of individual genomes with respect to a reference sequence was made by [3]. That work, like those of [12] and [13], aimed to build data structures suitable to efficiently compress the collection, while allowing fast random access to parts of it. Pattern search still remained an open question.

The problem of efficiently searching for patterns in a such index was addressed and resolved later by [23], but some of the data structures used therein do not allow the encryption of sequences related to different individuals with distinct keys.

1.2. Paper contribution and organization

In the present work we introduce ER-index (Encrypted Referential index), the first encrypted self-index based on referential Lempel-Ziv compression and designed so that it can be the core of a multi-user database engine.

When used on collections of highly similar sequences, the ER-index allows to obtain compression ratios which are an order of magnitude smaller than those of E^2FM-index, but maintaining its optimal search performance. Moreover, the ER-index *multi-user encryption model* permits to store genomic sequences of different individuals with distinct encryption keys within the same index. This allows the index users to perform search operations only on the sequences to which they were granted access.

The paper is organized as follows. Section 2 gives an overview of the main features of ER-index, alongside with the computational methods and data structures which make possible such features. Sections 3 and 4 give details on its core algorithms, pointing out some important differences of our approach with respect to current computing techniques for genomic databases. Section 5 illustrates how to construct a file-system based genomic database using the ER-index, and section 6 reports and discusses the results of the tests we have run

in order to assess the performance of our tool versus a state-of-the-art index, the *wavelet tree FM-index* by [7]. Finally, Section 7 sums up the main features of the *ER*-index and sketches out future work.

2. System and Methods

The *ER*-index is an open-source C++ tool designed to handle an encrypted genomic database. It is a full-text index consisting substantially in two major components:

- a set of relative Lempel-Ziv factorizations, one for each sequence of the collection;
- a set of auxiliary data structures to support encryption and search operations.

Both the factorizations and the auxiliary data structures are designed to permit efficient pattern searching, while allowing users to search only on sequences in the database to which they were granted access.

In order to apply encryption with a small overhead in searching and compression performance, each factorization is splitted in a series of fixed-length blocks of factors, so that each of them contains the same number of factors. Each block is then processed indipendently, so to produce a compact representation whose size depends on the compressibility of the information addressed by its factors. Finally, the variable size blocks previously obtained are independently encrypted from each other using the *Salsa20* cipher of [2]. Salsa20 was one of the ciphers selected as part of the eSTREAM portfolio of stream ciphers (see [1]), and has been designed for high performance in software implementations on Intel platforms. It produces a keystream of 2^{70} bytes from a 256-bit key and a 64-bit arbitrary *nonce* which is changed for each new run with the same key. It subsequently encrypts a sequence of b bytes plaintext by XOR-ing it with the first b bytes of the stream, discarding the rest of the stream.

A main point in protecting long-term, sensitive information – as that provided by genomic databanks – is to provide encryption methods which can outstand advanced attacks and next generation computing paradigms and platforms. As of 2019 there are no known attacks on Salsa20, and the 15-round reduced version of this cipher was proven 128-bit secure against differential cryptanalysis by [16]. Moreover, according to [8], it is resistant against side channel attacks and the new emerging quantum computing platforms.

2.1. Relative Lempel-Ziv factorization

Let S be a finite string of symbols over a finite alphabet A. Lempel-Ziv methods consist in rules for parsing S into a sequence of factors, so to replace repetitions in S by using references of their previous occurrences. Factors contain indeed references to a *dictionary* of substrings in S. The difference between the LZ77, LZ78 and the relative Lempel-Ziv factorization is that both the LZ77

and LZ78 build their dictionary "on the fly", putting in it substrings encountered in S before the current scanning position, whilst the relative Lempel-Ziv factorization obtains its compression by comparing the text to an already existing dictionary.

In the context of our application domain, S is a genomic sequence of an individual belonging to a given species for which a reference sequence R is known. As it is well known in Genomics S is very similar to R, presenting only a few number of mutations, deletions and insertions, often in a percentage not greater than 1%. Thus using R to construct the dictionary rather than S can allow a better compression of S: indeed, a given portion of S is more similar to the corresponding portion of the reference sequence than to a previously seen substring of S. This is the basic idea of the so-called *Referential Genome Compression*, which can be implemented thanks to Relative Lempel-Ziv factorization.

Definition 2.1. ***Relative Lempel-Ziv factorization*** *Let S and R two finite strings over the same finite alphabet A. The Relative Lempel-Ziv factorization of S with respect to the reference R, denoted as $LZ(S|R)$, is a sequence of n factors*

$$z_0 \cdots z_{n-1} \ .$$

Each factor z_j $(j = 0, \ldots, n-1)$ is a triple $\langle p_j, l_j, mc_j \rangle$, where:

- *p_j is the position of the longest substring r_j in R matching the current substring s_j in S*
- *l_j is the length of r_j;*
- *mc_j (a.k.a. mismatch character) is the last character in s_j, so that $s_j = r_j || mc_j$.*

2.2. B+ trees

B Trees and their B+ variant ([24]) are dynamic balanced trees whose nodes contain data values and their related search keys. They are often used for databases and file system indexing due to the fast search operation they allow to perform. The main difference between B and B+ trees is that the former allows every node to contain data values, while in the latter these can be found only in leaf nodes, with every other node containing only search keys.

Definition 2.2. ***B+ tree*** *Let N be a positive integer. A B+ tree of order N is a tree with the following properties:*

- *All leaf nodes are in the same level, i.e. the tree is balanced;*
- *Every non leaf node contains multiple search keys, stored in increasing order, which act as separation values for its subtrees: the left sub-tree of a key contains values lower than those of the father key, and the right sub-tree contains the values greater than those of its father key;*
- *Every internal node contains at least N keys and at most $2N$ keys;*

- *If the root node is not a leaf node, it has at least one key and at most* $2N$ *keys.*

B+ trees support efficient updates and exact match queries, which find the values associated to a given key. They also permit to do efficiently an operation known in literature as *range query*, which finds all the values related to keys in the interval $[l, r]$.

Tipically B Trees nodes are stored on secondary storage as fixed size disk pages, whose size is a multiple of the hosting file system page size. In order to increase the number of keys and pointers stored in each page, a compression scheme can been applied which takes advantage of the fact that keys in a node are very close each other. For the B+ trees implemented in the ER-index was used the *Invariable Coding* method of [11], which for each node stores the first key and the differences between any other key and the first one, using the minimum number of bits required to express the difference of the last key from the first key.

2.3. The ER-index

Let $\{S_1, \ldots, S_l\}$ be a collection of sequences corresponding to l different individuals. Let R and R_{rev} a reference sequence and its reverse. Let $f_i = LZ(S_i|R)$ be the relative LZ-factorization of sequence S_i with respect to R. Let $BL_{i,1}, \ldots, BL_{i,bn(i)}$ be the sequence of bs-length blocks factors gotten from $LZ(S_i|R)$, where $bn(i)$ denotes the obtained number of blocks and each block contains exactly bs factors. Finally, let $S20(plaintext, key, nonce)$ denote the Salsa20 encryption of *plaintext* with a 256-bit secret *key* and a 64-bit *nonce*.

The ER-index stores each of the aforementioned blocks encrypted, using a different secret key k_i for each individual and the block number as nonce, so that the encryption $E(f_i, k_i)$ of factorization f_i is given by:

$$E(f_i, k_i) = S20(BL_{i,1}, k_i, 1) \cdots S20(BL_{i,bn(i)}, k_i, bn(i)).$$

In order to speed up search operations, we have designed the *Encrypted B+ tree* ($EB+$ tree), an extended and encrypted variant of the $B+$ tree data structure. Thanks to an $EB+$ tree each single factor of the encrypted collection $\{E(f_i, k_i)\}$ is associated to the right identifier i and encryption key k_i.

Before performing encryption, we use *Invariable Coding* for both node search keys and values, but in a different way than in [24]. Indeed, the authors of that work applied compression to arrange more values into fixed-size node pages, whereas we used it in order to obtain smaller variable length nodes, thus minimizing the overall index size.

Definition 2.3. ***Encrypted Referential index*** *An Encrypted Referential index (ER-index) for a collection of sequences* $\{S_1, \ldots, S_l\}$ *with respect to a reference sequence* R *and a set of encryption keys* $\{k_1, \ldots, k_l\}$ *is a self-index consisting of:*

- *the encrypted relative Lempel-Ziv factorizations of sequences* $\{S_1, \ldots, S_l\}$ *with respect to* R*:* $\{E(f_1, k_1), \ldots, E(f_l, k_l)\}$*;*

- *a set of three EB+ trees whose search keys are respectively:*

  1. sai_rev_j*, a suffix array index corresponding to a* R_{rev} *suffix prefixing the reverse of the* j^{th} *factor referential part;*
  2. sai_j*, a suffix array index corresponding to a* R *suffix prefixing the* j^{th} *factor referential part;*
  3. tp_j*, the position of the* j^{th} *factor referential part in the reference sequence* R*;*

  and whose values are the couples $\langle i, v \rangle$*, where* i *identifies the sequence* S_i *and* v *is the Lempel-Ziv factor of the related genomic sequence, encrypted with key* k_i*.*

3. Factorization algorithm

The factorization algorithm used to build the ER-index slightly differs from that proposed by [12] and [23], as the ER-index uses a couple of FM-indexes to represent the reference sequence R and its reverse R_{rev}. The j^{th} factor is again a triple $\langle sai\_rev\_start_j, l_j, mc_j \rangle$ of numbers, but in the ER-index they have a different meaning:

- $sai\_rev\_start_j$ is the R_{rev} suffix array index from which to start the backward scan of R_{rev} in order to obtain the factor;

- l_j is the length of the factor, comprehensive of the mismatch character;

- mc_j is the mismatch character.

In order to speed-up pattern search, the algorithm retrieves also the three auxiliary data sai_rev_j, sai_j and tp_j stored as search keys in the corresponding $EB+$ trees composing the ER-index (see Definition 2.3). The algorithm, whose pseudo-code is given by Algorithm 1, uses four data structures related to the reference sequence R:

- the FM-index FM of R;

- the FM-index FM_{rev} of the sequence R_{rev} gotten by reversing R;

- a correspondence table $R2F$, which maps a suffix of R_{rev} to the R suffix starting from the same character;

- the reverse correspondence table $F2R$, which maps a suffix of R to the R_{rev} suffix starting from the same character.

Given a sequence S, Algorithm 1 scans S from left to right and at each step it tries to factorize the suffix S_i by searching the maximum-length referential factor starting from i. For this purpose it scans the BWT of R_{rev} through FM_{rev}, starting from $S[i]$ and proceeding backward on R_{rev} until a mismatch is found. This backward search gives as result the R_{rev} suffix array range containing the suffixes prefixing the reverse of S_p; the algorithm choose the first among them, as they are all equivalent for its purposes.

The further processing of the algorithm consists in retrieving the auxiliary information related to the previously found factor. The *getTextPosition* and *backwardStep* functions exactly match the canonical FM-index implementation, so they are not reported as pseudo-codes.

Algorithm 1 Factorization algorithm

```
1:  function FACTORIZE(S,FM_rev,FM,R2F,F2R)
2:      j ← 0                                          ▷ Current factor index
3:      l_max ← 0;                                     ▷ Maximum factor length
4:      len ← length(S);
5:      i ← 0;
6:      while i < len do
7:          ▷ Retrieve the next factor
8:          nrc ← S[i];              ▷ Curr char, not remapped in the FM index
9:          l ← 1                        ▷ Curr length of the next factor ref part
10:         if i < len − 1 AND isInRef(FM_rev, nrc) then
11:             lastNrc ← nrc;
12:             ▷ Start a backward search on the rev ref index
13:             c ← remap(FM_rev, nrc);                    ▷ Remap curr char
14:             sp ← C(FM_rev, c);
15:             ep ← C(FM_rev, c + 1) − 1;
16:             backStepSuccess ← true;
17:             ▷ Backward search stops when the ref part includes the last
18:             ▷ but one char of S OR the next char is not in the ref sequence
19:             ▷ OR the last backward step was not successful OR the
20:             ▷ next char is N and the last is not OR viceversa
```

Algorithm 1 Factorization algorithm (continued)

```
21:          while i + l < len − 1 AND
                   isInRef(FM_rev, nrc ← S[i + l]) AND
                   backStepSuccess AND
                   (lastNrc ≠ N AND nrc ≠ N OR
                   lastNrc = N AND nrc = N) do
22:              c ← remap(FM_rev, nrc);
23:              trySp ← C(FM_rev, c)+
                     Occ(FM_rev, EOF_shift(FM_rev, sp − 1), c);
24:              tryEp ← C(FM_rev, c)+
                     Occ(FM_rev, EOF_shift(FM_rev, ep), c) − 1;
25:              if trySp ≤ tryEp then
26:                  sp ← trySp;
27:                  ep ← tryEp;
28:                  l ← l + 1;
29:                  backStepSuccess ← true;
30:              else
31:                  backStepSuccess ← false;
32:              end if
33:              lastNrc ← nrc;
34:          end while
35:          sai_rev_pref ← sp;
36:          mc ← S[i + l]
37:          ▷ Find sai_rev_start, sai_pref and tp, as follows:
38:          ▷ Find sai of R for sai_rev_pref of R_rev
39:          sai = R2F(sai_rev_pref);
40:          ▷ Do l − 1 back steps on FM index to find sai_pref
41:          for i ← 1 To l − 1 do
42:              sai ← backwardStep(FM, sai);
43:          end for
44:          sai_pref ← sai;
45:          ▷ Find position tp of R for sai_pref using
46:          ▷ FM index marked rows
47:          tp = getTextPosition(FM, sai_pref)
48:          ▷ Do one back step on FM index to find sai_rev_start
49:          sai_rev_start ← backwardStep(FM, sai_pref);
50:          ▷ Store the retrieved factor in the factors array
51:          factors[j] ← ⟨sai_rev_start, l, mc⟩;
52:      end if
53:      i ← i + 1
54:  end while
55: end function
```

4. Pattern search algorithm

ER-index supports exact pattern matching through algorithm 2. Before describing the algorithm details, it is appropriate to make some considerations. A pattern search operation on a Lempel-Ziv factorization can retrieve two types of occurrences:

- *internal occurrences*, which are completely contained in a factor's referential part;
- *external occurrences*, also known in literature as *overlapping occurrences*, which have at least a character outside of a factor's referential part.

External occurrences can span two or more factors or end with a factor's mismatch character, and a solution to find them on $LZ78$ factorizations is proposed in [18]. The search pattern is splitted in all possible ways and, for each split point, the algorithm searches for the right side prefix and the reverse left side prefix in two related *trie* data structures ([4]). This results in two sets, the factors ending with the pattern's left side and the factors starting with the pattern's right side, and the algorithm eventually joins these two sets in order to obtain couples of consecutive factors. This approach can be applied also to relative Lempel-Ziv factorizations, but in our case it would require two tries for each individual, which is very expensive in term of disk space. Thus the $LocateExternalOccs$ function of Algorithm 2 follows a similar approach, but it makes use of the following less expensive data structures:

- the FM_{rev} and FM indexes of algorithm 1, in order to search for the maximal prefix of the reversed left side in R_{rev} and for the maximal right side prefix in R, respectively;
- a couple of $EB+$ trees to retrieve the factors ending with the maximal prefix of the reversed left side and those starting with the maximal right side suffix, respectively.

As for internal occurrences, the approach in [18] is based on the fact that each $LZ78$ factor is the concatenation of a previous factor with an additional character, which is not true for relative Lempel-Ziv factorizations.
Therefore function $LocateInternalOccs$ of Algorithm 2 implements an original approach which uses once again the FM index of the reference sequence R, together with a third $EB+$ tree, named $posTree$, whose search keys are the starting positions of factors referential parts in R. This last $EB+$ tree allows to retrieve the factors whose referential part starts in a given positions range of the reference sequence. The $LocateInternalOccs$ function also uses the auxiliary information l_{max}, defined as the maximum length of all factors contained in the ER-index, which is determined during the factorization process and is stored into the index header.
Since an internal occurrence of the pattern is completely contained in the reference sequence, the first step implemented in $LocateInternalOccs$ could have

been to retrieve all the pattern's occurrences in the reference sequence. However, we have first to check that an individual sequence factor containing the reference sequence occurrence really exists, and then retrieve the occurrence positions in the individual sequence previously found.

These issues have been addressed thanks to the following consideration. Let us suppose that the suffix array interval $[sp, ep]$ is the result of a pattern search on the reference sequence. The position tp of each interval's element in the reference sequence can be retrieved using the related reference index marked rows. Given a factor, let also l be the length of its referential part, tpf its starting position in the reference sequence, and m the pattern length. A reference occurrence located in tp is also an individual sequence occurrence if and only if:

$$\begin{cases} tpf \leq tp \\ tpf \geq tp + m - l \end{cases} \tag{1}$$

The first condition is to make sure that a factor's referential part does not start after the first character of the reference occurrence, while the second that the factor's referential part does not end before the end of the reference occurrence. If both the above tpf range bounds were fixed values, the referential part factors could be retrieved by performing a range query on $posTree$. The lower bound actually is not a fixed value, since it depends from the length l of the referential part of the factor, but we can consider the maximum length of all factors $l_{max} \geq l$. Because of (1), the wrong values returned by a range query based on $tp + m - l_{max} \leq tpf$ can indeed be filtered out by keeping only those factors complying to $tpf \geq tp + m - l$.

Algorithm 2 Pattern search algorithm

```
1:  function LOCATE(pat)
2:      extoccs ← LOCATEEXTERNALOCCS(pat);
3:      intoccs ← LOCATEINTERNALOCCS(pat);
4:      occs = extoccs ⋃ intoccs;
5:      ▷ Sort each individual occurrence by position
6:      SORT(occs);
7:      ▷ Remove any duplicates
8:      REMOVEDUPLICATES(occs);
9:      ▷ Find each occurrence text position from its factor identifier
10:     ▷ factorId and its factor offset FactorOffset
11:     FINDTEXTPOSITIONS(occs);
12:     return occs;
13: end function
```

Algorithm 2 Pattern search algorithm (continued)

```
14: function LocateExternalOccs(pat)
15:     occs = []
16:     pl ← len(pat);
17:     for sp ← 0 to pl − 1 do
18:         splitPointCharacter ← pat[sp];
19:         if splitPoint > pl/2 then
20:             ▷ The left side part (lsp) is longer than the right side part (rsp),
21:             ▷ so the factors expected to end with the lsp are less than those
22:             ▷ expected to start with the rsp
23:             ls ← substr(pat, 0, splitPoint);
24:             [lsFacts, lsls] ← FindLeftSideFactors(ls);
25:             ▷ Scan lsFacts through the individual identifiers indId
26:             for indId in lsFacts do
27:                 ▷ Get the factorization f from an associative array fs
28:                 ▷ with all the individual factorizations
29:                 f ← fs[indId];
30:                 for factInd in GetIndRetrFactors(lsFacts, indId) do
31:                     fact ← f[factInd];
32:                     ▷ Exclude that the left side crosses the starting point
33:                     ▷ of the current factor,since an occurrence of this type
34:                     ▷ will be found for a preceding split point
35:                     if fact.len − 1 ≥ len(ls) then
36:                         if fact.letter = splitPointCharacter then
37:                             occ.factInd ← factInd;
38:                             occ.factOff ← fact.len − 1 − len(ls);
39:                             occ.endingFactInd ← factInd;
40:                             occ.endingFactOff ← fact.len − 1;
41:                             lsvl ← lsls;              ▷ Left side verified length
42:                             rsvl ← 0;                 ▷ Right side verified length
43:                             if PatRemPart(f, pat, sp, lsvl, rsvl, occ) then
44:                                 AddOccurrence(occ);
45:                             end if
46:                         end if
47:                     end if
48:                 end for
49:             end for
50:         else
51:             ▷ The right side part (rsp) is not shorter than the left side
52:             ▷ part (lsp), so the factors expectedto start with the rsp
53:             ▷ are less than those expected to end with the lsp
54:             rs ← substr(pat, splitPoint + 1, pl − splitPoint − 1);
55:             [rsFacts, rslp] ← FindRightSideFactors(rs);
```

Algorithm 2 Pattern search algorithm (continued)

```
            for indId in rsFacts do
                ▷ Get the factorization f from an associative array fs
                ▷ with all the individual factorizations
                f ← fs[indId];
                for factInd in GetIndRetrFactors(rsFacts, indId) do
                    fact ← f[factInd];
                    if rslp < fact.len − 1 then
                        rsvl ← rslp;                    ▷ Right side verified length
                    else
                        rsvl ← fact.len − 1;
                    end if
                    if factInd > 0 then
                        lsFact ← f[factInd − 1];
                        if lsFact.letter = splitPointCharacter then
                            occ.factInd ← factInd − 1;
                            occ.factOff ← lsFact.len − 1;
                            occ.endingFactInd ← factInd;
                            occ.endingFactOff ← rsvl − 1;
                            lsvl ← 0;                   ▷ Left side verified length
                            if PatRemPart(f, pat, sp, lsvl, rsvl, occ) then
                                AddOccurrence(occ);
                            end if
                        end if
                    end if
                end for
            end for

        end if
    end for
    return occs;
end function

function LocateInternalOccs(pat)
    occs = []
    ▷ An internal occurrence occurs certainly in the ref seq
    if searchPatInRefIndex(FM, pat, sp, ep) then
        for i ← sp to ep do
            m ← len(pat);
            tp ← getPositionInReference(FM, i);
            ▷ lmax is the length of the maximum factor in the index
            facts ←
                    getFactorsInRange(posTree, tp + m − lmax, tp);
            for each distinct indId in facts do
                ▷ Retrieve the factorization f from an associative array fs
                ▷ containing all the individual factorizations
                f ← fs[indId];
```

Algorithm 2 Pattern search algorithm (continued)

```
99:                 for factInd in GetIndRetrFactors(facts, indId) do
100:                    fact ← f[factInd];
101:                    tpf ← fact.refPartPositionInReference;
102:                    l ← fact.len − 1;                ▷ factor refential part length
103:                    if tpf ≥ tp + m − l then
104:                        occ.factInd ← factInd;
105:                        occ.factOff ← tp − tpf;
106:                        occ.endingFactInd ← factInd;
107:                        occ.endingFactOff ←
                        occ.factOff + m − 1;
108:                        lsvl ← 0;                    ▷ Left side verified length
109:                        AddOccurrence(occ);
110:                    end if
111:                end for
112:            end for
113:        end for
114:    end if
115:    return occs;
116: end function

117: function PatRemPart(f, pat, splitPoint, lsvl, rsvl, occ)
118:    ▷ Check if the occurrence occ is really a whole pattern occurrence,
119:    ▷ updating it if required. It returns true for successful checks.
120:    ▷ This function tries to extend the verified part of the pattern, both
121:    ▷ on the left and the right side, by comparing the yet not verified
122:    ▷ pattern characters with the factors characters preceding and
123:    ▷ following the verified part. For performance the extension is
124:    ▷ made without extracting the full text of the involved factors,
125:    ▷ but scanning the text one character at a time thanks to the
126:    ▷ reverse reference index.
127: end function

128: function FindLeftSideFactors(ls)
129:    ▷ Return (lslsfact, lsls), where lslsfact is a list of factors
130:    ▷ ending with the left side longest suffix, and lsls is the left side
131:    ▷ longest suffix length.
132:    ▷ Find the longest left side suffix that occurs in the reference string
133:    l ← findLeftSideLongestSuffix(ls);
134:    lsls ← substr(ls, ls.len − l, l);
135:    if searchPatRevInRefIndex(FM_rev, lsls, sp, ep) then
136:        ▷ Find factors whose suffixArrayPosition is in [sp,ep]
137:        return [getFactorsInRange(reverseTree, sp, ep), l];
138:    else
139:        return [[],0];
140:    end if
141: end function
```

Algorithm 2 Pattern search algorithm (continued)

```
142: function FindRightSideFactors(rs)
143:     ▷ Return (rslpfact, rslp), where rslpfact is a list of factors
144:     ▷ beginning with the right side longest prefix, and rslp is the
145:     ▷ right side longest prefix length
146:     ▷ Find the longest right side prefix occurring in the reference string
147:     l ← findRightSideLongestPrefix(ls);
148:     rslp ← substr(rs, 0, l);
149:     if searchPatInRefIndex(FM, rslp, sp, ep) then
150:         ▷ Find factors whose suffixArrayPosition is in [sp,ep]
151:         return [getFactorsInRange(forwardTree, sp, ep), l];
152:     else
153:         return [[],0];
154:     end if
155: end function

156: function FindRightSideLongestPrefix(rs)
157:     ▷ Scan backward the reverse index, starting from the first char
158:     ▷ of the right side and going on until a mismatch is found.
159:     ▷ Return the right side longrst prefix rslp.
160: end function

161: function FindLeftSideLongestSuffix(ls)
162:     ▷ Scan backward the straight index, starting from the last char
163:     ▷ of the left side and going on until a mismatch is found.
164:     ▷ Return the left side longest suffix lsls.
165: end function

166: function searchPatInRefIndex(FM_index, pat, sp, ep)
167:     ▷ Perform a canonical backward search on the given FM-index,
168:     ▷ returning the [sp,ep] suffix array range corresponding to
169:     ▷ pattern pat.
170: end function

171: function searchPatRevInRefIndex(FM_index, pat, sp, ep)
172:     ▷ Perform a backward search on the given FM-index, starting from
173:     ▷ the first pattern char, then the second char, and so on.
174:     ▷ Return the [sp,ep] suffix array range corresponding to
175:     ▷ pattern pat.
176: end function

177: function GetIndRetrFactors(facts, indId)
178:     ▷ Returns the indexes of factors belonging to the individual indId,
179:     ▷ selecting them from the collection facts, which contains factors
180:     ▷ belonging to several individuals
181: end function
```

5. Implementation

The ER-index is designed to be the building block of an encrypted database, that stores genomic information about a possibly large set of individuals. Roughly speaking, an *Encrypted Referential database* (ER-database for short) is a collection of ER-indices whose access is managed through portfolios of secret keys related to a population of individuals and a set of database users:

Definition 5.1. ***Encrypted Referential Database*** *Let $R = \{R_j : j \in J \subseteq \{1, \ldots, 22, X, Y\}\}$ be a set of reference sequences for human chromosomes. Let $I = \{I_1, \ldots, I_l\}$ denote a set of individuals and $S = \{S_{ij} \mid (i, j) \in I \times J\}$ be a set of genomic sequences, where S_{ij} is the sequence of individual I_i related to chromosome R_j. An* ***Encrypted Referential Database*** *(**ER-database**) for I with reference R is a tuple*

$$D = \{I, R, K, U, ER, P\}$$

where:

- *$K = \{k_1, \ldots, k_l\}$ is a set of randomly-generated, symmetric encryption keys so that $k_i \in K$ is uniquely and secretly associated to I_i for $i = 1, \ldots, l$;*
- *$U = \{U_1, \ldots, U_r\}$ is a set of database users, where each U_r is allowed to access only to the sequences of a subset of the individuals in I;*
- *ER is a set of ER-indexes for the population I, each one relative to a different sequence in R;*
- *P is a mapping from U to I that, for each user $U_r \in U$, indentifies the individuals in I whose access is granted to U_r.*

A simple implementation provides for an ER-database hosted by a file system directory, named the *database root*. The database root contains the database catalog *catalog.xml*, which lists all the individuals, users and reference sequences composing the database. Moreover, in the database root there are the subdirectories `references` and `indexes` containing the sets R and ER, respectively; and the subdirectory `security`, which contains the key portfolios of the database users in U. The *key portfolio* for a database user $U_r \in U$ contains only the *Salsa20* keys related to the individuals in I whose genomic information U_r has been granted access; it is handled with asymmetric encryption techniques, and enciphered with the U_r's public key so that only U_r can read its content by using his/her private key.

6. Experimental results

In order to evaluate the ER-Index performance, a small ER-database concerning 50 individuals and 10 users was implemented as described in the previous section.

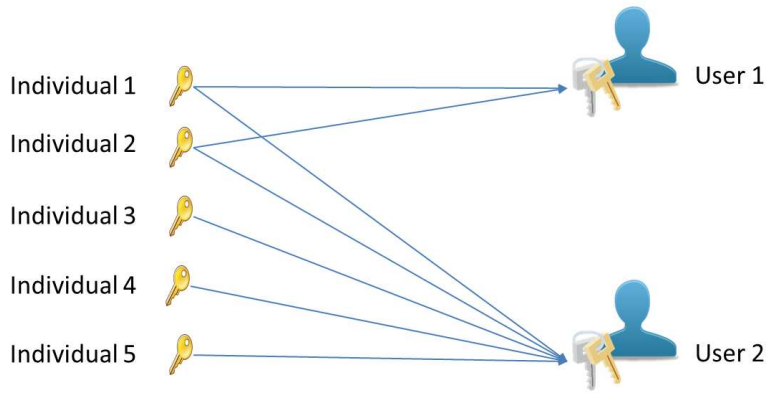


Figure 1: A simple scenario of key portfolios where user U_1 can access only genomes of individuals I_1 and I_2, whilst user U_2 has access to the genome of all the individuals in the database.

Using such database, a comprehensive set of tests was performed on different computing platforms to measure the compression ratios (i.e.; the ratio of the output data size to the input data size as defined by [20]), and the times required to build the index and search for patterns. The results were also compared with a "state of the art" software built thanks to the *Sdsl C++ library* by [5], available at `http://github.com/simongog/sdsl`. This library provides some succint data structures for implementing self-indexes like the *Compressed Suffix Arrays* (CSA) by [6] and the *wavelet tree FM-indexes* by [7], and we extended this last kind of implementation so to manage collections of items and report sequence-relative locations.

The tests were performed on three computing platforms having different resources as follows, in order to assess the performance of our tool and assess its effectiveness with respect to the reference tool for different CPUs, memory sizes and operating environments:

- `ser`, a small-size server with an Intel(R) Xeon(R) CPU E5-2697 v2 at 2.7GHz 24 cores processor and 180GB of DDR3 1333 MHz memory, running the *CentOS 7* operating system;
- `lap`, a laptop with an AMD A10-9600P at 2.4GHz 6 cores processor and 12GB of DDR4 1866 MHz memory, running an *Ubuntu on Windows* application on the *Windows 10* operating system with the Microsoft-Windows-Subsystem-Linux turned on;
- `clu`, a node of a computing cluster with 2 Intel Xeon CPU E5-2670 at 2.6GHz 10 cores processor and 196GB of DDR3 1600 MHz memory, running the *CentOS 6* operating system.

6.1. Experimental setup

The individual sequences chosen to assess the prototype performance are those related to human chromosomes 11 and 20, for a population of 50 individuals. Chromosome 11 (135,086,622 base pairs) and 20 (64,444,167 base pairs) were chosen as representatives of big and small human chromosomes, respectively. The sequences were of two types:

- Diploid consensus sequences obtained from the *1000 Genomes Project* (`www.internationalgenome.org/home`). These sequences were built by starting from the respective BAM files and using the `samtools mpileup` (`www.htslib.org`) command along with the `bcftools` and `vcfutils` utilities.

- Pseudo-random sequences obtained by applying single mutations, insertions and deletions to the corresponding chromosome reference sequence in the human genome bank HS37D5, a variant of the GRCh37 human genome assembly used by the *1000 Genomes Project*. For this purpose [15] built a tool which selects, with uniform distribution, mutations, insertions and deletions according to the percentages observed on average by [17] among different individuals of the human species.

Although artificially generated, the second kind of sequences is more appropriate than consensus sequences to evaluate real performances, since they are free from spurious symbols caused by sequencing machines errors or inaccuracy.
For each one of the two above types we considered full length sequences and 1MB sequences, obtained by selecting one million basis of those chromosomes. Thus we performed our tests on a total of eight kinds of genomic collection sequences, with consensus collections denoted as 11_1MB, 11_$FULL$, 20_1MB, 20_$FULL$, and their artificially generated counterparts identified by the suffix _R. Some tests include also 5MB sequences, obtained by selecting five million basis from chromosomes 11 and 20.

The encryption set-up consisted in the generation of fifty 256-bit simmetric keys through the `openssl rand` command, plus ten RSA key couples using the `openssl genrsa` and `openssl rsa` commands. The key portfolio for each of the database users was generated by choosing a subset from the pool of simmetric keys and ciphering it with the user's public key.

6.2. Construction times

Tables 1 and 2 show times required to construct the ER-index on the three considered computing platforms; moreover, the first table reports a comparison with the reference tool on `ser`. Similar results were obtained on the other two platforms, showing that times required to build the ER-index – except than for some very short (1MB) sequences – are significantly lower than those for the Sdsl wavelet-tree FM-index , despite the fact that only the ER-index implements data encryption.

This noticeable perfomance has been obtained through our parallel factorization algorithm, which exploits the multi-core, hyper-threading architecture of

	20_1MB	20_5MB	20_FULL	11_1MB	11_5MB	11_FULL
ER	11.38	33.74	455.7	23.18	42.79	1005
Sdsl	20.08	132.1	2061	19.33	154.9	5693

Table 1: Times (sec) required to build the ER-index (ER) and the Wavelet-tree FM-index (sdsl) on the `ser` platform.

modern processors (see Section 3). As it can be easily inferred by a comparison of the obtained values, the speed-up increases with the number of cores, so it could be greater on higher-end machines with more processor cores.

	20_1MB	20_1MB_R	20_FULL_R	11_1MB	11_1MB_R	11_FULL_R
lap	50.09	47.64	208775	63.64	69.32	5409862.50
clu	9.91	9.81	256.25	19.78	16.24	528.54

Table 2: Times (sec) required to build the *ER*-index on the `lap` and `clu` platforms.

Note that the full collections have notable sizes (about 2.97 and 6.4 GiB for the 20_$FULL$ and 11_$FULL$ sets, respectively), and this resulted in long computing times (about 0.58 and 1.5 hours) on the `lap` platform. However, we believe these last results are not very indicative since probably due to an improper memory management by the virtual machine.

6.3. Compression ratios

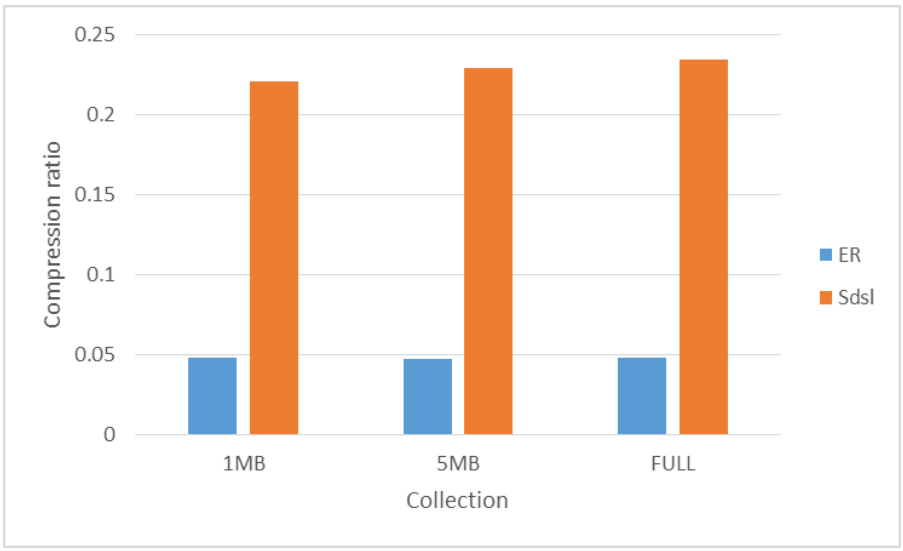
Figure 2 reports the compression ratios of the *ER*-index versus the wavelet-tree FM-index on the `ser` computing platform for the collections obtained from the 1000 Genomes Project. Obviously, similar results were obtained on the `lap` and `clu` platforms, showing that the *ER*-index got compression ratios about *four times smaller* than the reference tool.

The compression ratios values on pseudo-random sequences were about half than those for the previous sequences: since the sequences created from an algorithm lack of spurious symbols, it is possible to find longer matches between the analyzed sequence and the reference one, and achieve a better compression performance. Overall (see Table 3) this is very good for the ER-index, resulting in at least 97% savings in space. For example, the 6.4 GiB of collection 11_$FULL_R$ resulted in an index smaller than 192 MiB.

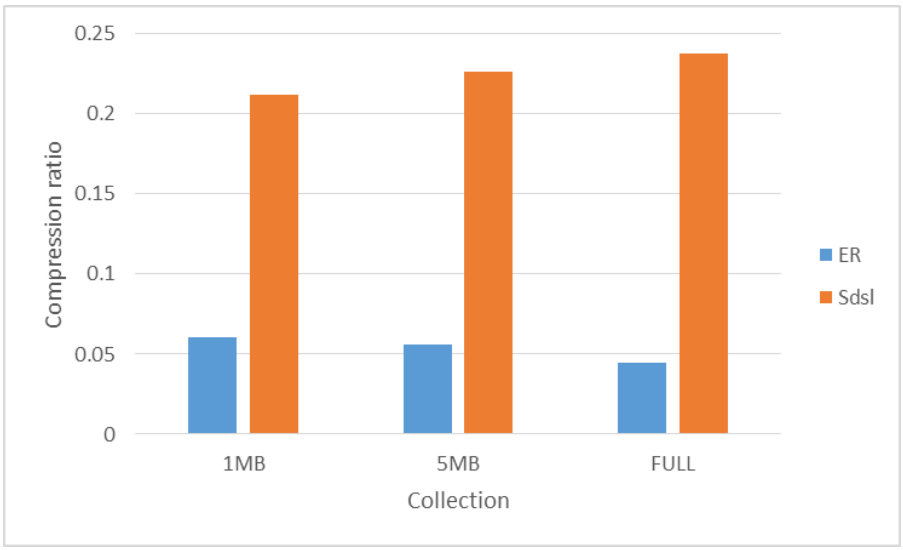
It can be worth to note here that the reported figures are *mean* values obtained by building the index more times (we usually performed 18 index builds for each collection, in order to filter out spurious computing time values due to umpredictable overheads from other processes running on the same platform). As a matter of facts, the multithreading approach causes the operations to be

20_1MB_R	20_FULL_R	11_1MB_R	11_FULL_R
0.026	0.028	0.026	0.030

Table 3: ER-index compression ratios on the collections of pseudo-random sequences.



(a) Chromosome 20



(b) Chromosome 11

Figure 2: *ER*-index (ER) versus Wavelet-tree FM index (Sdsl) compression ratios for chromosome 20 and chromosome 11 collections on the `ser` platform.

executed in different order during the factorization, thus a different order in the creation of the auxiliary data structures. But, since such data structures are B+ trees, depending on the order of the insertion of the keys there might be different splits in their nodes, resulting in small changes in the size of the index.

6.4. Pattern search performance

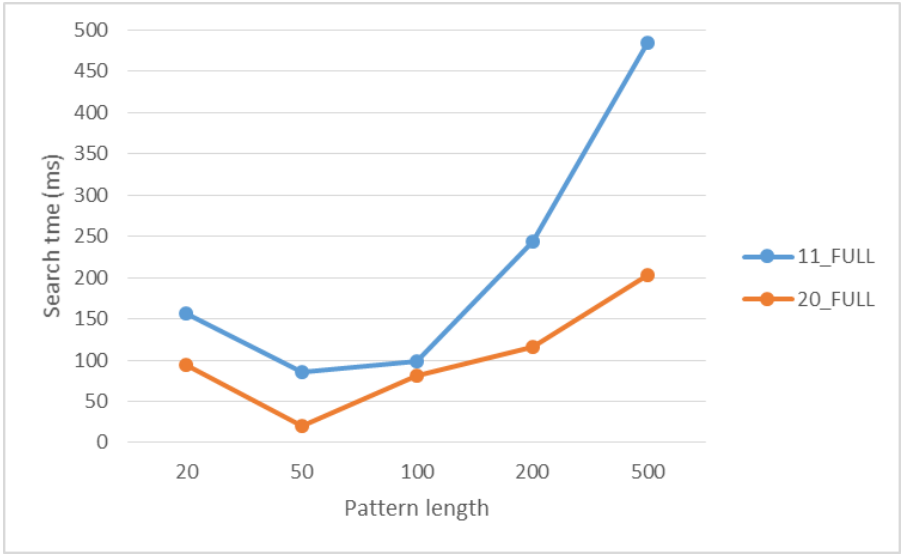
For each collection given in Section 6.1, the tests to evaluate pattern search (a.k.a. *locate* operation) performance were run as follows:

- the index (*ER*-index or wavelet-tree FM-index) related to the given collection was selected, and only its header was loaded in memory;
- for each pattern length $pl \in \{20, 50, 100, 200, 500\}$, 500 patterns were randomly extracted from the sequences composing the collection, and all of them were searched through the index;
- *mean* and *median* values of the 500 search times and search times *per occurrence* got at the previous step were computed;
- the index was closed, and the next test was performed.

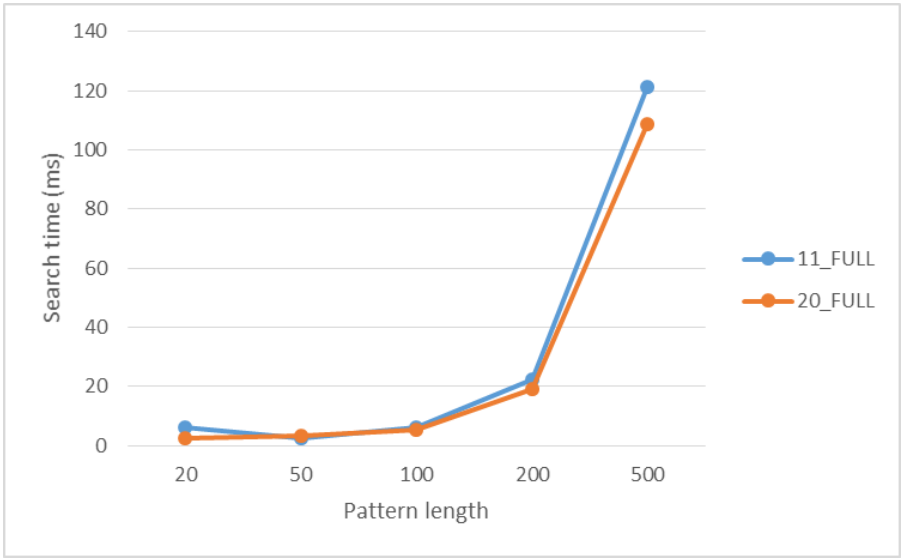
Figure 3 plots the search time values obtained on the `ser` platform for collections 20_FULL and 11_FULL, whereas Table 4 sums up the results obtained for the other collections on the `lap` and `clu` platforms.
For full collections, pattern search times should be proportional to the number of found occurrences, and thus they should decrease with pattern length, since a bigger pl turns out in less chances to find a pattern. However, the obtained results clearly show that for such collections search times are higher for $pl = 20$ than $pl = 50$ but they increase afterward. This is due to the algorithm used for external occurrences (see Section 4), since the number of split points checked increases with the pattern length. However, this behaviour could be noticeably improved by parallelizing the several split points operations, which are naturally independent from each others.

Another interesting observation which follows from the obtained results is that median values are significantly smaller than mean values. This is because of a 10-15% of outliers, due to some patterns hard to search, or to the fact that some searches were performed right after the opening of an index, when only a small amount of factorizations and EB+ blocks were loaded in memory. Patterns may be hard to search since they have a much greater number of occurrences than the other patterns of the same length, or because they span on many short factors so that the left or right side related to some split points are very short strings.

Overall these results show that locate operations are executed very fast thanks to the *ER*-index: also on a small computing platform running a virtual machine like `lap` they take less than half a second in the worst case (i.e.; looking for 500-basis patterns in the 11_FULL_R collection of 6.4 GiB). A comparison with the wavelet-tree FM-index has shown that this last performs better on patterns with $pl \geq 100$, but slightly worse on short patterns. These differences are



(a) Mean search times



(b) Median search times

Figure 3: ER-index mean and median pattern search times (ms) for collections 20_$FULL$ and 11_$FULL$ on the `ser` platform.

however of the order of hundredths of a second, so they are ininfluential from a practical point of view, except in application scenarios where a massive amount of pattern searches is required. Since pattern search times are proportional to the number of found occurrences, it is appropriate to look at the mean and median values of the search time per occurrence, reported in Figure 4 for the `ser` platform and the two collections 20_$FULL$ and 11_$FULL$. These results show that mean search times per occurrence grow with pattern length, starting from a few milliseconds for $pl = 20$ to a maximum of 190.622 ms for $pl = 500$ on the 11_$FULL$ collection. Note that the curves of the median values related to the two collections perfectly overlap. This attests the scalability of the ER-index: collections of increasing size can be managed without a significant loss in performance.

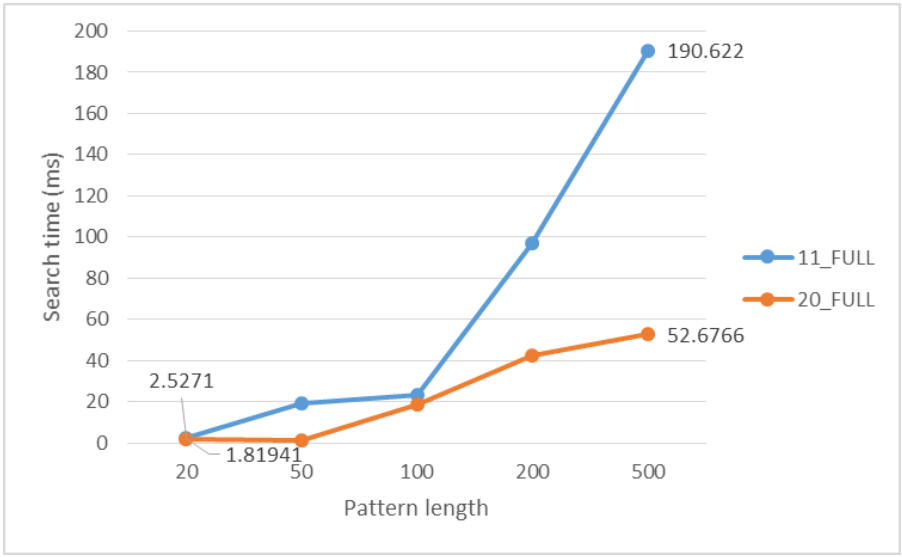
pl	**20_1MB**	**20_1MB_R**	**20_FULL_R**	**11_1MB**	**11_1MB_R**	**11_FULL_R**
20	2.11	1.41	45.19	4.96	0.99	122.41
50	2.71	1.89	7.89	3.07	1.73	45.72
100	7.63	6.51	12.38	8.00	5.77	19.39
200	47.93	34.93	52.20	37.60	33.05	61.95
500	363.47	359.59	382.79	371.36	367.98	408.45
20	0.40	0.29	4.19	0.58	0.31	8.84
50	0.92	0.80	1.61	1.00	0.81	1.72
100	3.04	2.78	4.25	3.18	2.87	4.70
200	13.17	12.39	14.88	13.62	13.04	16.33
500	86.97	83.25	85.20	87.72	86.45	91.89

Table 4: ER-index search time mean values (ms) on platforms `lap` and `clu`.

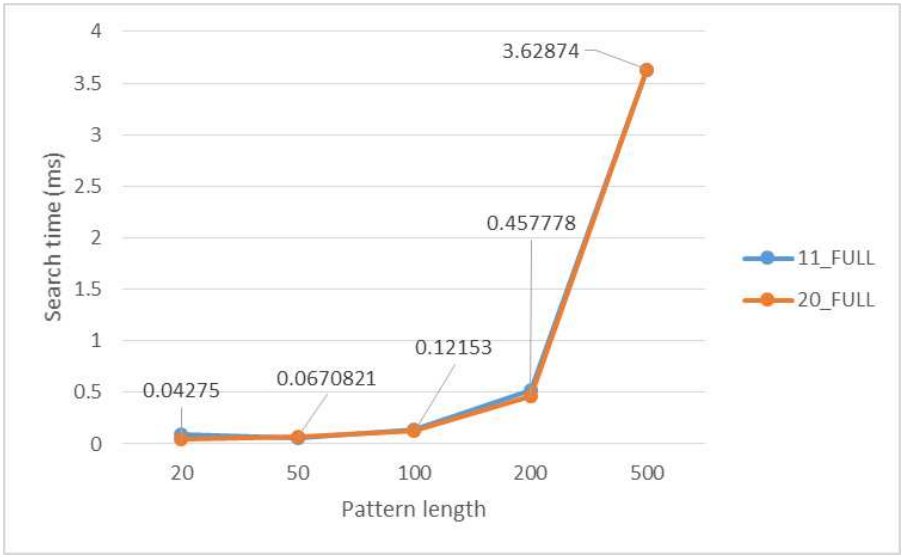
7. Conclusion and future work

The ER-index is a new tool designed to be the core of secure genomic databases: it exploits inter-sequence redundancy to get very low compression ratios, and stores the sequences of different individuals so that they are encrypted on disk with different encryption keys within the same index. This new index can store collections of genomic sequences in less than 1/30 of their original space, outperforming state of the art tools like the wavelet tree FM-index, and offering in addition the critical feature of a built-in, quantum-resistant encryption. Moreover, the Sdsl library index has to be loaded entirely in RAM to perform any searching operation, whereas the ER-index loads in RAM only the blocks required to perform the required operation.

The data structures provided with the ER-index allow for a very good performance in pattern search: our tests have shown that search times per occurrency are less than two tenths of a second on an encrypted and compressed collection of 6.4 GiB. As a matter of fact, the ER-index ouperforms the wavelet-tree FM-index in searching for short length patterns, whilst it is slower in the order of hundredths of a second for longer patterns. This is a remarkable result, considering that the wavelet-tree FM-index does not operate on encrypted sequences.



(a) Mean search times



(b) Median search times

Figure 4: *ER*-index mean and median search times (ms) per occurrence for collections 20_$FULL$ and 11_$FULL$ on the `ser` platform.

Morever, times for searching large patterns could be noticeably reduced thanks to multi-threading techniques for the pattern search algorithm.

A multi-threading search strategy and an algorithm for inexact search operations are under investigation and will be implemented in a next release of the *ER*-index.

[1] Babbage, S., DeCanniere, C., Cantenaut, A., Cid, C., Gilbert, H., Johansson, T., Parker, M., Preneel, B., Rijmen, V., and Robshaw, M. (2008). The estream portfolio (rev.1). Available at: `http://www.ecrypt.eu.org/stream/finallist.html`.

[2] Bernstein, D. J. (2005). Salsa20 specification. Available at: `http://www.ecrypt.eu.org/stream/salsa20pf.html`.

[3] Brandon, M. C., Wallace, D. C., and Baldi, P. (2009). Data structures and compression algorithms for genomic sequence data. *Bioinformatics*, **25**(14), 1731–1738.

[4] Brass, P. (2008). *Advanced data structures*. Cambridge University Press.

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

[6] Grossi, R. and Vitter, J. S. (2005). Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, **35**(2), 378–407.

[7] Grossi, R., Gupta, A., and Vitter, J. S. (2003). High-order entropy-compressed text indexes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 841–850. Society for Industrial and Applied Mathematics.

[8] Grote, O., Ahrens, A., and Benavente-Peces, C. (2019). A review of post-quantum cryptography and crypto-agility strategies. *International Interdisciplinary PhD Workshop*.

[9] Homer, N., Szelinger, S., Redman, M., Duggan, D., Tembe, W., Muehling, J., Pearson, J. V., Stephan, D. A., Nelson, S. F., and Craig, D. W. (2008). Resolving individuals contributing trace amounts of dna to highly complex mixtures using high-density snp genotyping microarrays. *PLoS genetics*, **4**(8), e1000167.

[10] Kreft, S. and Navarro, G. (2011). Self-indexing based on lz77. In *Proceedings of the 22Nd Annual Conference on Combinatorial Pattern Matching*, CPM'11, pages 41–54, Berlin, Heidelberg. Springer-Verlag.

[11] Krizka, F., Krátkỳ, M., and Baca, R. (2009). Benchmarking a b-tree compression method. In *Proceedings of the Conference on Theory and Practice on Information Technologies*.

[12] Kuruppu, S., Puglisi, S. J., and Zobel, J. (2010). Relative lempel-ziv compression of genomes for large-scale storage and retrieval. In *Proceedings of the 17th International Conference on String Processing and Information Retrieval*, SPIRE'10, pages 201–206, Berlin, Heidelberg. Springer-Verlag.

[13] Kuruppu, S., Puglisi, S. J., and Zobel, J. (2011). Optimized relative lempel-ziv compression of genomes. In *Proceedings of the Thirty-Fourth Australasian Computer Science Conference - Volume 113*, ACSC '11, pages 91–98, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

[14] Menezes, A. J., Van Oorschot, P. C., and Vanstone, S. A. (2010). *Handbook of applied cryptography*. CRC press.

[15] Montecuollo, F., Schmid, G., and Tagliaferri, R. (2017). E2fm: an encrypted and compressed full-text index for collections of genomic sequences. *Bioinformatics*, **33**(18), 2808–2817.

[16] Mouha, N. and Preneel, B. (2013). Towards finding optimal differential characteristics for arx: Application to salsa20. Technical report, Cryptology ePrint Archive, Report 2013/328.

[17] Mullaney, J. M., Mills, R. E., Pittard, W. S., and Devine, S. E. (2010). Small insertions and deletions (indels) in human genomes. *Human molecular genetics*, **19**(R2), R131–R136.

[18] Navarro, G. (2002). Indexing text using the ziv-lempel trie. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval*, SPIRE 2002, pages 325–336, London, UK, UK. Springer-Verlag.

[19] Sagner, M., McNeil, A., Puska, P., Auffray, C., Price, N. D., Hood, L., Lavie, C. J., Han, Z.-G., Chen, Z., Brahmachari, S. K., *et al.* (2017). The p4 health spectrum–a predictive, preventive, personalized and participatory continuum for promoting healthspan. *Progress in cardiovascular diseases*, **59**(5), 506–521.

[20] Salomon, D. (2004). *Data compression: the complete reference*. Springer Science & Business Media.

[21] Sirén, J., Välimäki, N., Mäkinen, V., and Navarro, G. (2009). Run-length compressed indexes are superior for highly repetitive sequence collections. In *Proceedings of the 15th International Symposium on String Processing and Information Retrieval*, SPIRE '08, pages 164–175, Berlin, Heidelberg. Springer-Verlag.

[22] Stephens, Z. D., Lee, S. Y., Faghri, F., Campbell, R. H., Zhai, C., Efron, M. J., Iyer, R., Schatz, M. C., Sinha, S., and Robinson, G. E. (2015). Big data: astronomical or genomical? *PLoS biology*, **13**(7), e1002195.

[23] Wandelt, S., Starlinger, J., Bux, M., and Leser, U. (2013). RCSI: Scalable similarity search in thousand(s) of genomes. *Proc. VLDB Endow.*, **6**(13), 1534–1545.

[24] Zhang, D. (2004). *Handbook Of Data Structures And Applications (Chapman & Hall/Crc Computer and Information Science Series.)*, chapter "B Trees". Chapman & Hall/CRC.

[25] Ziv, J. and Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, **23**(3), 337–343.

[26] Ziv, J. and Lempel, A. (1978). Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory*, **24**(5), 530–536.