Indexing and Self-indexing sequences of IEEE 754 double precision numbers[☆]

Antonio Fariña, Alberto Ordóñez, José R. Paramá[*]

Departamento de Computación, Facultade de Informática, Universidade da Coruña, Campus de A Coruña, 15071 A Coruña, Spain

Abstract

Succinct data structures were designed to store and/or index data with a relatively small alphabet size, a rather skewed distribution and/or, a considerable amount of repetitiveness. Although many of them were developed to handle text, they have been used with other data types, like biological collections or source code. However, there are no applications of succinct data structures in the case of floating point data, the obvious reason is that this data type does not usually fulfill the aforementioned requirements.

In this work, we present four solutions to store and index floating point data that take advantage of the latest developments in succinct data structures.

The first one is based on the well-known inverted index. It consumes space around the size of the source data, providing appealing search times. The other three solutions are based on self-indexing structures. The first one uses a binary Huffman-shaped wavelet tree. It is never the winner in our experiments, but still yields a good balance between space and search performance. The second one is based on wavelet trees on bytecodes, and obtains the best space/time trade-off in most scenarios. The last one is based on Sadakane's Compressed Suffix Array. It excels in space at the expense of less performance at searches.

Including a representation of the original data, our indexes occupy from around 70% to 115% of the size of the original collection, and permit fast indexed searches within it.

Keywords: Indexing, Compact structures, Real numbers.

[☆]A preliminary version of this paper was published in (Fariña et al., 2012).

[*]Corresponding author. Tel. +34981167000 Fax. +34981167160.

Email addresses: `fari@udc.es` (Antonio Fariña), `alberto.ordonez@udc.es` (Alberto Ordóñez),

1. Introduction

Although the web continues to attract much research activity, the spread of the information technology to all aspects of life implies that other fields deserve also attention. Among others, stock exchange markets, geographic information systems or the forthcoming electrical grids (SmartGrid) are fields that require addressing the problem of dealing with large amounts of floating point data. For example, the forthcoming electrical grids will produce huge amounts of floating point data, given that any electricity consumer or producer will continuously report information in form of streams of real numbers (Zicari, 2012; IBM Software, 2012).

The use of compression techniques allows to reduce both the flow of information through the network and the storage needs. Furthermore, once the data are stored, in many scenarios it is mandatory to access these data efficiently to help real-time decision-making processes, which, in turn, requires the development of suitable indexing techniques.

Existing indexes provide adequate performance for most classical computer applications. However, these techniques require additional space that, when dealing with large amounts of information, might become prohibitive. Compression has become a so common solution that even commercial database management systems have included it to save space and to increase I/O performance (Garmany et al., 2008). A more evolved solution is to use compression techniques that permit random decompression from any position (Moura et al., 2000), this both saves space and permits indexes to be built over the compressed data.

Unfortunately, compression techniques specifically designed for floating point data (Burtscher and Ratanaworabhan, 2009; Ratanaworabhan et al., 2006), or general purpose techniques that obtain reasonable compression values with this type of data (like *P7zip*), require decompression to start at the beginning of the compressed data, thereby preventing the indexation of such data.

Therefore, we have a trade-off between space and performance. On the one hand, to obtain sub-linear search time over sequences of real numbers, we have to store the data

`parama@udc.es` (José R. Paramá)

uncompressed in order to be able to build an index over them, thus increasing the space requirements. On the other hand, if we need to save space, we have to store the data in a compressed form and then, to perform searches, we must first decompress such data, and then run a linear-time search algorithm.

The outbreak of the web has promoted the development of data structures that have several appealing characteristics to store and/or index text. Their success is based on a relatively small alphabet, a quite skewed data distribution, and data that do not suffer from frequent changes. However, floating point data have a large alphabet (a real number hardly ever repeats twice) and a low biased distribution, yet many scenarios can hold the last requirement, since it is usual to produce data that never change and are only stored to, for example, run decision-making processes. In this work, we aim at applying those data structures to the floating point data scenario, trying to keep their good features as much as possible.

In recent years, the research community has worked extensively on inverted indexes (Knuth, 1973) due to their use to index text and more specifically to index the web. The main problem of this structure is the consumption of space, since it spends considerable amounts of space in addition to the original information.

At the same time, a new family of data structures, called self-indexes, have merged compression with indexing. They require space proportional to the compressed text, replace it, and permit fast indexed searches on it without any additional structure (Navarro and Mäkinen, 2007). They are able to *extract* any text substring and *locate* the occurrence positions of a pattern string in a time that depends on the pattern length and the output size (number of occurrences), but not on the text size (that is, the search process is not sequential). Most of them are also able to *count* the number of occurrences of a pattern string much faster than just locating them.

In this work, we present four contributions. First, we study the adaptation of inverted indexes to the indexation of real numbers. In our experiments, we will show that a structure whose size (including the original data plus the whole index) is between 10% smaller and 20% larger than the original sequence, obtains locate times that are up to around 8,000 times faster than the sequential search. Second, we study the use of self-indexes to index and store sequences of real numbers. The first index of this family is based on the classical wavelet tree (WT), (Grossi et al., 2003), which is a very flexible structure that allows both to store and index data. Our second proposal based on self-indexes uses a wavelet tree on bytecodes (WTBC), (Brisaboa et al., 2012), which can be seen as a variation of the WT that showed a better performance in previous scenarios. The third self-index is based on the recent *integer-based Compressed Suffix Arrays* (iCSA) (Fariña et al., 2012). iCSA is a variation of the well-known Compressed Suffix Array (CSA) (Sadakane, 2003) that permits to successfully handle large alphabets, as those that arise when indexing real numbers. In our experiments, the self-index based on the classical WTs occupies around the same size as the original collection (from 10% less to around 10% more), whereas the *locate* times are up to 12 times faster than the sequential search. However, in the case of the WT the *extract* operation, needed to recover a portion of the original sequence, requires some computational effort, whereas that time is negligible in the case of inverted indexes. Nevertheless, we expect that displaying the whole original data will be a rather infrequent operation, whereas checking if a data source surpassed a certain threshold will be the usual case. In a general scenario the best space/time trade-off of our study is achieved by using an index based on the WTBC, which obtains locate times up to 100 times faster than the sequential search with a structure up to 15% smaller. Yet, when one aims at saving as many space as possible, the iCSA-based self-index becomes the best choice. It permits to save up to 30% of the size of the collection and permits to count the occurrences of a given search pattern up to 10^8 times faster than a sequential search, but the performance at locating them is poor.

The capabilities of these structures can be useful in different scenarios, for example, assume that the measures of the electricity consumption of end consumers are stored in one or multiple arrays, or simply, they are stored consecutively. Those values are measures taken at different time points, that is, they are *time series* (Engelson et al., 2000). One of the main targets of the SmartGrid is to determine the production of the power plants over time, since the electricity cannot be stored, and therefore an excess of production represents a waste of money. Consider the time series of the aggregated consumption of a certain region, the SmartGrid could be interested in the past periods where the consumption was within a certain range, assuming that when the consumption is between the boundaries of that range, this requires a particular configuration of the system, including that certain power plants should be on while others must be disconnected from the network and turned off. The study of the conditions of those past periods, like the hour of the day and the weather conditions, helps to determine which is the suitable configuration for the current conditions.

In most scenarios, like in a SmartGrid system, we expect that exact searches of floating point values will be unlikely, and we expect that range queries will be the usual ones. However, our indexes can search for exact values if it is required. We also expect functionalities like "retrieve all the measures taken at a particular time point", that is, we have to be able to access a given position of the sequence in sublinear time as well.

The outline of this paper is as follows. Section 2 presents some related work and Section 3 discusses the 64-bits IEEE 754 format and a preliminary study of the characteristics of real numbers. Section 4 presents our first contribution, showing how to adapt inverted indexes to successfully index sequences of real numbers. In Section 5, we describe how we adapted three well-known self-indexing structures (a classical WT, a WTBC, and an iCSA) to deal with real numbers. Section 6 shows our experiments. Finally, our conclusions and directions for future work are shown in Section 7.

2. Related Work

2.1. Inverted indexes

Inverted indexes (Knuth, 1973) are composed of two elements: a list of search keys and a collection of posting lists. The list of search keys is the set of all different indexed key values. For each key value, the index contains a posting list that stores the positions in the original sequence where that particular key can be found.

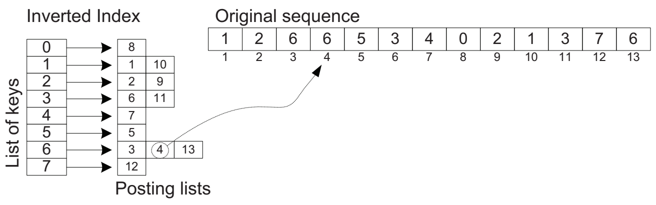


Figure 1: An inverted index over the sequence $\langle 1, 2, 6, 6, 5, 3, 4, 0, 2, 1, 3, 7, 6\rangle$.

The inverted index is the most common structure to index text. Its importance for the web has increased the interest of the investigation of this structure. The focus of the research community is in the main problem of the inverted index, it requires too much space. Observe that, as a classic index, it is an auxiliary structure that should be added to the data. To attenuate this problem, two strategies can be followed: to compress the source data and to compress the posting lists.

In the case of text, several compression techniques have been used to compress the source data. Among them, the word-oriented Huffword is a well-known alternative (Witten et al., 1999) that combines good compression and performance. Other more recent techniques such as Tagged Huffman (Moura et al., 2000) or the Dense codes (Brisaboa et al., 2007) own also another interesting feature as they permit *random access*; that is, they permit to start the decompression from any point of the compressed data. For repetitive data, grammar-based compressors (Larsson and Moffat, 2000) proved also to be successful.

Observe that a posting list is a sequence of increasing numbers, therefore incremental encoding is the obvious choice to obtain compression. Additional savings are obtained using different techniques to encode the gaps, like for example, Byte codes (Williams and Zobel, 1999), δ-codes (Elias, 1975), Rice codes (Rice, 1979), or more recently PforDelta (Zukowski et al., 2006). Another approach to save space is that the values within the posting lists point to blocks instead of pointing to exact positions (Manber and Wu, 1993). If a block contains several occurrences of a searched key, then the corresponding posting list has fewer pointers. However, now the index is only useful to filter out blocks and then, a sequential scan within those blocks is needed to obtain the exact positions of the searched key.

2.2. Self-indexing structures

Self-indexes are a more recent approach to both store and index sequences of symbols $S[1, n]$ over an alphabet Σ of size σ, which were developed for relatively small alphabets and for sources with a rather skewed distribution. They efficiently support three basic operations: *counting* and *locating* all the occurrences of a given pattern in S, as well as *extracting* any subsequence $S[i, j]$ from S. Therefore, they enable indexed searches within S, and provide an implicit (and typically smaller) representation of S, since $S[1, n]$ can be recovered via *extract* operation.

In the sequel we focus on two well-known self-indexing structures such as the wavelet-tree (Grossi et al., 2003) and Sadakane's Compressed Suffix Array (Sadakane, 2003). We also describe two variations of them that were shown to successfully handle large alphabets such as the Wavelet Tree on Bytecodes (Brisaboa et al., 2012) and the Word Compressed Suffix Array (Fariña et al., 2012). These structures are the basis of our self-indexes to handle sequences of floating point numbers.

2.2.1. Wavelet trees

The Wavelet Tree (WT) (Grossi et al., 2003) is a well-know structure that permits to self-index a given sequence $S[1,n]$ of symbols drawn from an alphabet Σ of size σ. Although such alphabets were usually small, WTs have succeeded at dealing with sequences composed of words (Brisaboa et al., 2012), and also at managing sequences built over even larger alphabets (Mäkinen and Navarro, 2008; Navarro and Russo, 2011; Navarro, 2012). As a self-index, a WT replaces S since it allows to extract the symbol at any given position. Besides, it is capable of obtaining all the positions of S where a given symbol is located.

WTs were firstly proposed for solving *rank* and *select* queries over sequences on large (non-binary) alphabets. Given a sequence of symbols S, $rank_b(S,i) = y$ if the symbol b appears y times within $S[1,i]$, and $select_b(S,j) = x$ if the j^{th} occurrence of the symbol b in the sequence S appears at position x.

The original WT is a balanced binary tree, that requires $n \log \sigma(1+o(1))$ bits of space and $O(\log \sigma)$ query time. At each node v, only a bitmap is stored. It is denoted by $B(v)$. When v is the root node, Σ is divided into two halves Σ_0 and Σ_1. The symbols $S(v)[i]$ from S that belong to Σ_0 go to the left child of v and $B(v)[i]$ is set to 0 to keep track of this. In the same way, if $S(v)[i]$ belongs to Σ_1, $B(v)[i]$ is set to 1. Recursively, each child v' handles in the same way the part of S it received, denoted as $S(v')$. That is, $B(v')[i] \leftarrow 0$ iff $S(v')[i]$ is sent to the left-child of v', and $B(v')[i] \leftarrow 1$ otherwise. The sequence of labels obtained when traversing the tree from the root down to a node v is the *binary label* of v, which is denoted as $L(v)$. The node that handles the symbols of S that have the binary label $L(v)$ is denoted as $V_{L(v)}$. The binary labels of the leaves correspond to the binary representation of the symbols of Σ.

The WT has been used for different purposes, and with different shapes. For instance, if the frequency of each symbol is known, we can build a WT with the shape of a Huffman tree (Grossi et al., 2003). In this case, the number of bits stored in the bitmaps of the WT is the same as the number of bits output by a Huffman compressor. This is upper bounded by $n(H_0(S) + 1)$, being $H_0(S)$ the zero-order empirical entropy defined as $H_0(S) = -\sum_{c\in\Sigma, f_c>0} \frac{n_c}{n} \log \frac{n_c}{n}$, where n_c is the number of occurrences of each symbol.

Using an uncompressed bitmap representation that takes $n + o(n)$ bits, the total space of the WT is at most $n(H_0(S)+1)(1+o(1))+O(\sigma \log n)$ bits. Using the technique by Raman et al. (2002) to compress the bitmaps, the space is reduced to $nH_0(S) + o(n \log \sigma)$ bits. The advantages of using a Huffman shaped WT are not only regarded to compression effectiveness but also to access time. Grossi et al. (2004) showed that if symbols are accessed with a frequency proportional to their number of occurrences in the sequence S, then the average access time is improved from $O(\log \sigma)$, for a plain WT, to $O(H_0(S) + 1)$ in a Huffman shaped WT, holding $H_0(S) \leq \log(\sigma)$.

Over the alphabet $\langle 0, 1, \ldots, 7\rangle$, Figure 2 shows a WT storing/indexing the sequence of symbols $\langle 1, 2, 6, 6, 5, 3, 4, 0, 2, 1, 3, 7, 6\rangle$. The shaded numbers are not actually stored in the WT, and are only displayed for illustration purposes. In this example, the WT is a balanced binary tree, since the coding used to shape the WT consists in the 3-bits binary representation of each symbol.

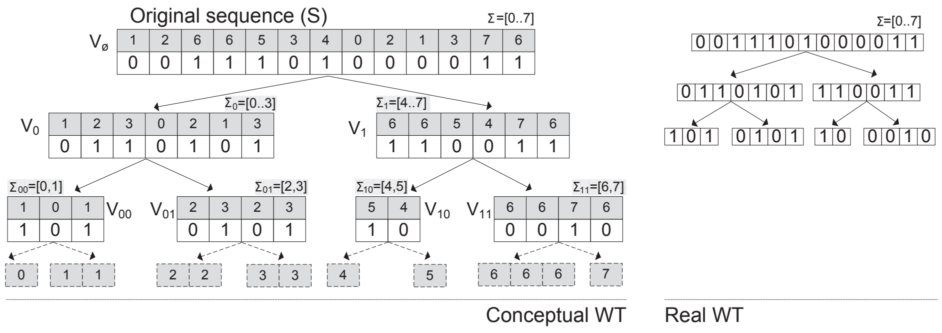


Figure 2: A WT over the sequence $\langle 1, 2, 6, 6, 5, 3, 4, 0, 2, 1, 3, 7, 6\rangle$.

The WT reduces both recovering the original data and searching for a given symbol to *rank* and *select* operations on the bitmaps respectively. For example, in order to recover (*extract*) the symbol $S[4]$ from the WT, we access the 4^{th} position of the bitmap in the root node ($B(V_\emptyset)[4]$), obtaining a 1. Now we have to check which is the order of that

1 among all the $1's$ of that node, in our case, it is the second occurrence of a 1 (that is, $rank_1(B(V_\emptyset), 4) = 2$). Next, we have to access the second position of the bitmap of the node V_1, that is, the node handling the symbols of the second half of the alphabet (those whose binary representation starts with 1). We obtain again a 1 ($B(V_1)[2] = 1$), and that this is the second occurrence of a 1 in node V_1 ($rank_1(B(V_1), 2) = 2$). Finally, we move to node V_{11} and access the second position of its bitmap ($B(V_{11})[2]$), where we obtain the bit value 0. Therefore, the binary representation of $S[4]$ is 110, and we recover symbol 6.

We can also *search* for the position in S of the j^{th} occurrence of a symbol. First, we have to obtain the codeword of the symbol using the same encoder used to shape the WT. Then, we must reach the leaf that contains such codeword by traversing the tree downwards, that is, at level i, if the i^{th} bit of the codeword is 1 we move to the right child, whereas we move to the left child otherwise. Once the leaf is reached, we start an upward transversal using the nodes that are already in the recursion stack filled in the downward traversal. The algorithm is as follows: being initially $pos \leftarrow j$, if at level i the current node is the left child of its parent, we set $pos \leftarrow select_0(B, pos)$; otherwise we set $pos \leftarrow select_1(B, pos)$, where B is the bitmap of the current node. Once we reach the root, the sought position of the j^{th} occurrence of the symbol is pos.

If we want to retrieve all the positions where a symbol occurs, the process is similar but once we reach the leaf, we have to carry out an upward transversal for each occurrence of the symbol (the number of occurrences is known once we reach the leaf).

2.2.2. Wavelet trees on bytecodes

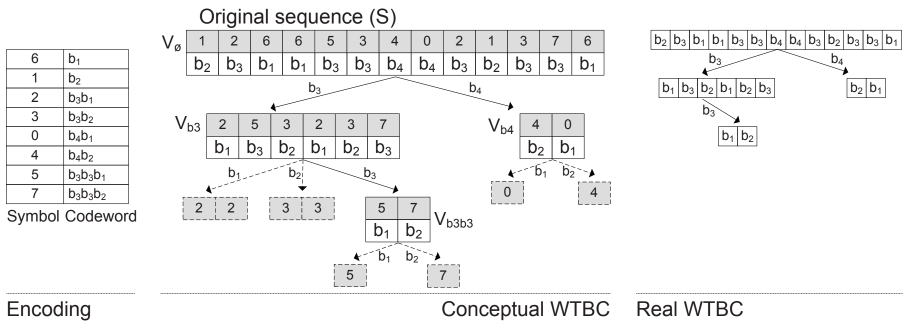
Wavelet trees on bytecodes (WTBC) (Brisaboa et al., 2012) are based on a family of compression codes known as bytecodes (Moura et al., 2000; Brisaboa et al., 2007; Culpepper and Moffat, 2005). These codes use bytes, rather than bits, as the target alphabet (that is, the codewords that replace the original symbols are sequences of one or more bytes). This change yields compression ratios[1] around 5% worse in large texts, when using a word-based modeler (Moffat, 1989). The target was to obtain faster times at

[1]From here on, compression ratio (cr) is shown as the size of the compressed file (c) as a percentage of its original size (n). That is: $cr \leftarrow c/n \times 100$.

compression and especially at decompression. The first technique, called Plain Huffman (PH), is just a Huffman code assigning byte rather than bit sequences to the codewords, that is, this is just the d-ary Huffman code that appears in Huffman's original paper (Huffman, 1952), with d=256.

More recently, Brisaboa et al. (2012) proposed reordering the bytes in the codewords from a text compressed with a given bytecode following a wavelet-tree-like strategy. At the expense of a small space overhead over the text compressed with that bytecode, such a reorganization turns the compressed data into a self-index: the WTBC structure. In particular, the WTBC with PH shape was shown to perform the best.

In WTBC, instead of representing the compressed data as a sequence of codewords, each one representing one original symbol, the data is represented with a wavelet tree where the different bytes of each codeword are placed at different nodes. The root of the wavelet tree contains the first byte of all codewords, following the same order as the original symbols. That is, the i^{th} byte of the root node is the first byte of the codeword corresponding to the symbol at the i^{th} position of the original sequence. The root has as many children as different bytes can be the first byte of a codeword. In the second level, the node corresponding to the x^{th} child of the root handles the second byte of those codewords whose first byte is x. That is, the byte at position i in node V_x is the second byte of the codeword corresponding to the i^{th} original symbol whose codeword starts with the byte x. The same procedure is followed in the lower levels of the tree.



6	b_1
1	b_2
2	b_3b_1
3	b_3b_2
0	b_4b_1
4	b_4b_2
5	$b_3b_3b_1$
7	$b_3b_3b_2$
Symbol	Codeword

Figure 3: A WTBC over the sequence $\langle 1, 2, 6, 6, 5, 3, 4, 0, 2, 1, 3, 7, 6\rangle$.

We use again $L(v)$ to denote the sequence of labels obtained when traversing the tree from the root down to a node v, yet in this case is a *byte label*. Also, we use $V_{L(v)}$ to denote the node that handles the symbols whose byte label starts by $L(v)$ and $B(v)$ to denote the sequence (in this case) of bytes in the node v.

An example of a WTBC storing/indexing the sequence $\langle 1, 2, 6, 6, 5, 3, 4, 0, 2, 1, 3, 7, 6\rangle$ is shown in Figure 3. Again the shaded elements are not stored and are only included for clarity. The process starts by computing the codeword corresponding to each original symbol using a given bytecode. After that, the tree is built as explained. The root contains the first byte of the codewords representing the symbols in the ordering of the original sequence. The second byte of those codewords is contained in the corresponding child. For example, since in the 8^{th} position of the root node we have the byte b_4, the second byte of $S[8]$ is in the child corresponding to the codewords that have b_4 as first byte (V_{b_4}). $S[8]$ is the second symbol of the original sequence that has b_4 as the first byte of its codeword, therefore in the second position of V_{b_4} we find the second byte of $S[8]$. Since the codeword representing 0 (the symbol in $S[8]$) only has two bytes, the process ends here.

Now, suppose that we want to recover the symbol $S[5]$ from the WTBC. We start at the 5^{th} position of the root node, and we obtain the byte b_3. This is the second occurrence of that byte in the root node (we know that by performing $rank_{b_3}(B(V_\emptyset), 5) = 2$). Therefore, we access the second position of V_{b_3}, where we obtain a b_3. This is the first occurrence of that byte value in V_{b_3}, since $rank_{b_3}(B(V_{b_3}), 2) = 1$). Then, we access the first position of $V_{b_3 b_3}$, and we finally obtain the byte value b_1. Therefore, the symbol $S[5]$ is represented by the codeword $b_3 b_3 b_1$. Now, we search for that codeword in the table containing the correspondence between the original symbols and the codewords and we obtain that $S[5]$ is a 5.

We can also search for the positions in the original sequence of a given symbol as in a binary WT. Hence, in a first step, we use the bytes from the codeword of that symbol to traverse the WTBC downwards by performing *rank* operations. Once the corresponding leaf is reached, we search for the positions of the occurrences of that symbol by traversing the WTBC upwards performing *select* operations.

The sum of the space needed by the WTBC is the same as the size needed by the codewords of the bytecode used to encode the symbols, plus a negligible amount of extra space to store the pointers that keep the shape of the WTBC.

2.2.3. The Compressed Suffix Array: Sadakane's CSA

Given a sequence $S[1,n]$ built over an alphabet Σ of length σ, the *suffix array* (Manber and Myers, 1993) $A[1,n]$ built on S is a permutation of $[1,n]$, so that for all the suffixes $S[i,n]$, $1 \leq i < n$, it holds $S[A[i],n] \prec S[A[i+1],n]$, being $\prec$ the lexicographic ordering (as S typically contained text).

Since A points to all the suffixes of S in lexicographic order, this structure permits to search for any pattern $P[1,m]$ in time $O(m \log n)$ by simply binary searching the range $A[l,r]$ that contains pointers to all the positions in S where P occurs. The m term appears because at each step of the binary search, one could need to compare up to m symbols from P with those in the suffix $S[A[i],A[i]+m-1]$. Unfortunately the space needs of A are high.

To reduce these space requirements, Sadakane's CSA (Sadakane, 2003) uses another permutation $\Psi[1,n]$ defined in (Grossi and Vitter, 2000). For each position j in S pointed from $A[i] = j$, $\Psi[i]$ gives the position z such that $A[z]$ points to $j+1 = A[i]+1$. There is a special case when $A[i] = n$, in this case $\Psi[i]$ gives the position z such that A[z]=1. In addition, we could set up an array $E[1,\sigma']$ with all the different symbols that appear in S, and a bitmap $D[1,n]$ aligned with A so that $D[i] \leftarrow 1$ if $i = 1$ or if $S[A[i]] \neq S[A[i-1]]$ ($D[i] \leftarrow 0$ otherwise). Basically, a 1 in D marks the beginning of a range of suffixes pointed from A such that the first symbol of those suffixes coincides. That is, if the i^{th} and $(i+1)^{th}$ one in D ocurr in $D[l]$ and $D[r]$ respectively, we will have $E[rank_1(D,l)] = E[rank_1(D,x)] \forall x \in [l, r-1]$. Note that $rank_1(D,i)$ can be computed in constant time using $o(n)$ extra bits (Jacobson, 1989; Munro, 1996).

By using Ψ, D, and E it is possible to perform binary search without the need of accessing A nor S. Note that, the symbol $S[A[i]]$ pointed by $A[i]$ can be obtained as $E[rank_1(D,i)]$, and we can obtain the following symbols of the source sequence $S[A[i]+1]$ as $E[rank_1(D,\Psi[i])]$, $S[A[i]+2]$ as $E[rank_1(D,\Psi[\Psi[i]])]$, and so on. Recall that $\Psi[i]$ basically indicates the position in A that points to the symbol $S[A[i]+1]$. Therefore, by using Ψ, D, and E we can obtain the symbols $S[A[i],A[i]+m-1]$ that we could need to compare with $P[1,m]$ in each step of the binary search.

However, in principle, Ψ would have the same space requirements as A. Fortunately, since Ψ is formed by σ subsequences of increasing values (Grossi and Vitter, 2000), those subsequences can be compressed by encoding the differences with δ-codes or γ-codes (Elias, 1975). In practice, absolute sampled positions $\Psi[1 + i \cdot t_{\Psi}]$ are also explicitly kept to permit fast access to Ψ values. Further research showed that coupling Huffman and run-length coding of gaps succeeded at reduce Ψ size (Navarro and Mäkinen, 2007; Fariña et al., 2012).

As said before Ψ, D, and E are enough to simulate the binary search for the interval $A[l, r]$, where pattern P occurs, without keeping A nor S. Being $r - l + 1$ the number of occurrences of P in S, this permits to solve the *count* operation. However, if one is interested in *locating* those occurrences in S, A is still needed as we need to recover the values $A[l, r)$. In addition, to be able to *extract* any subsequence $S[i, j]$ we also need to keep $A^{-1}[1, n]$ so that we know the position in A that points to $S[i]$ ($A^{-1}[i]$) from which we could start the *extraction* mechanism using Ψ, E, and D showed above. In practice, only sampled values of A and A^{-1} are stored. In the former case, the source sequence S is sampled at positions $t, 2 \cdot t, \ldots, n$, and an array $A_S[1, n/t]$ records the values of A (in suffix array order) pointing to those sampled positions. In addition, a bitmap B_A is set to mark the sampled positions from A. Therefore, sampled values $A[i]$ are computed as $A[i] \leftarrow A_S[rank_1(B_A, i)]$, whereas non-sampled values $A[i]$ can be retrieved by starting with $i' \leftarrow i$ and then applying $i' \leftarrow \Psi[i']$ k times ($k \leq t$) until a sampled position $A[x]$ is reached (that is, $B_A[x] = 1$). At this point we compute $A[i] \leftarrow A[x] - k$. Similarly, samples of A^{-1} are taken at positions $1 + j \cdot t$ and stored (in S order) into an array $A_S^{-1}[1, n/t]$. Therefore, when we want to *extract* a subsequence $S[b, e]$, we move to last sampled position $1 + j \cdot t$ before b ($j \leftarrow (b - 1)/t + 1$). From here on we know that $S[1 + j \cdot t]$ is pointed from $A[i]$, $i \leftarrow A_S^{-1}[1 + j]$. Therefore, we use the usual extraction mechanism to recover the subsequence $S[1 + j \cdot t, e]$ which contains $S[b, e]$.

From this point, the CSA is a *self-index* built on S that replaces S (as any substring $S[i, j]$ could be extracted) and does not need A anymore to perform searches.

Note that different sampling values t can be used for A (t_A), A^{-1} ($t_{A^{-1}}$), and for Ψ (t_{Ψ}). Yet, in practice, on a large alphabet of size $\sigma = \Theta(n^{\beta})$, $t = O(\log n)$ is a reasonable sampling rate. For example, in (Fariña et al., 2012) $t = 32$ showed a good space/time trade-off for a 1GB text. In this case, a CSA requires $nH_0 + O(n \log \log n)$ bits of space, yet in practice the space is closer to $2nH_k + O(n)$. In our experiments we will use several values $t = t_\Psi = t_A = t_{A^{-1}}$ hence leading to different space/time trade-offs in the CSA-based indexes. Figure 4 shows a description of the structures needed in a CSA.

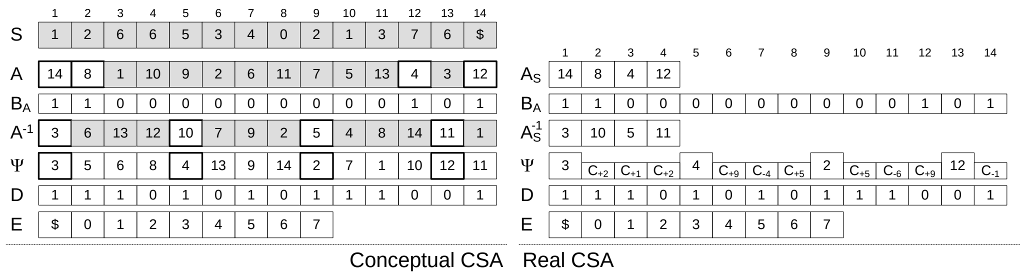

1 2 3 4 5 6 7 8 9 10 11 12 13 14
S 1 2 6 6 5 3 4 0 2 1 3 7 6 $
A 14 8 1 10 9 2 6 11 7 5 13 4 3 12
BA 1 1 0 0 0 0 0 0 0 0 0 1 0 1
A-1 3 6 13 12 10 7 9 2 5 4 8 14 11 1
Ψ 3 5 6 8 4 13 9 14 2 7 1 10 12 11
D 1 1 1 0 1 0 1 0 1 1 1 0 0 1
E $ 0 1 2 3 4 5 6 7
Conceptual CSA
AS 14 8 4 12
BA 1 1 0 0 0 0 0 0 0 0 0 1 0 1
AS-1 3 10 5 11
Ψ 3 C+2 C+1 C+2 4 C+9 C-4 C+5 2 C+5 C-6 C+9 12 C-1
D 1 1 1 0 1 0 1 0 1 1 1 0 0 1
E $ 0 1 2 3 4 5 6 7
Real CSA

Figure 4: A CSA self-index over the sequence $\langle 1, 2, 6, 6, 5, 3, 4, 0, 2, 1, 3, 7, 6 \rangle$.

2.2.4. The Word-based Compressed Suffix Array: WCSA and iCSA

The original CSA was designed to typically index characters within a text sequence, so that CSA was able to *count*, *locate*, and *extract* any substring of the indexed text. Fariña et al. (2012) modified CSA to index words rather than characters and created the so called *Word-based CSA* (WCSA). It allows to perform the same operations, but in a word-wise fashion. That is, WCSA is able to search only for sequences of words rather than any text substring. Yet, since in a text of size n there are only around $n/5$ words, WCSA has to index less symbols than a character-based CSA. This permits it to greatly reduce space needs.

To build a WCSA, each different word in the original text T is firstly mapped to an integer identifier (id), and then replaced by its corresponding id to make up the sequence S that is actually self-indexed with an *integer-based CSA* (iCSA). Note that, the original CSA handles an alphabet of at most 256 values (characters), whereas the number of different words in a text is much larger. Therefore, iCSA is just an adaptation of CSA that handles integers, hence allowing us to deal with very large alphabets.

3. The IEEE 754 format: Basics and preliminary analysis

We consider the double precision 64-bits IEEE 754 format (IEEE Std 754-200, 2008). This standard divides the floating point numbers into three components (see Figure 5): *(i)* 1 bit to indicate the sign of the number (1 for negative and 0 for positive numbers), *(ii)* 11 bits to represent the exponent of the number, and *(iii)* 52 bits to represent the mantissa.

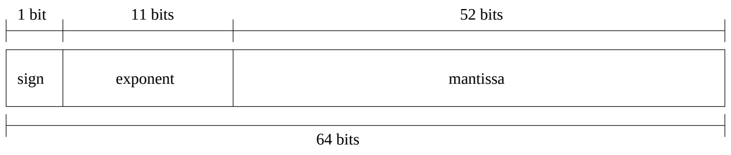


Figure 5: IEEE 754 64-bits format.

Using the following expression we can transform a binary floating point representation into the real number representation:

$$-1^{sign} \times 2^{exponent-bias} \times 1.mantissa$$

where constant *bias* equals 1023 (in the case of 64-bits double precision numbers).

The exponent indicates the magnitude of the number while the mantissa contains the integer part (if it has one) and the decimal part. This format usually normalizes the number to a number such that $1 \leq mantissa < 2$, where the 1 is not represented and the bits that are closer to the decimal point are stored in the leftmost bits reserved for the mantissa.

When we have a collection of double precision real numbers it is likely that most of those numbers are unique, and very few repetitions would be found. That is, the alphabet size typically increases linearly with the size of the collection. This is because most decimal numbers could not be represented since the format is designed for discrete machines, while the nature of the decimal numbers is purely continuous. Hence, the arithmetic algorithms must deal with rounding problems that can generate very different representations for similar numbers.

However, collections of floating point numbers are usually measures or results of some computational process or real phenomenon. Therefore, we expect that, despite the numbers could be significantly different, their magnitude can be similar (Engelson et al., 2000). In such a case, in a sequence of real numbers, the number of different exponents is usually smaller than the 2^{11} possibilities. This can be also the case of the leftmost bits of the mantissa as previous works suggest (Burtscher and Ratanaworabhan, 2009; Ratanaworabhan et al., 2006). Hence, the first bits are the most compressible part of each number.

The question is now how many bits must we index to achieve a compact representation? As explained, since it is rare that a real number appears twice in a sequence, if we index all the bits in each real value, the index will have an entry for each number in the sequence, and therefore the index will be even larger than the original data.

Fortunately, the nature of most data and the characteristics of the IEEE 754 format permit us to appreciate a biased distribution if we consider only the first bytes of those numbers. This will allow us to create a more compact index or self-index over those initial bytes.

To show this, we studied six collections of real numbers downloaded from Martin Burtscher's site.[2]

The collection named *brain* contains results from a numeric simulation of a velocity field of a human brain during a head impact. The rest, named *lu*, *bt*, *sp*, *sppm*, and *sweep3d*, contain numeric messages sent by a node in a parallel system running NAS Parallel Benchmark. Table 1 shows the characteristics of these collections.

Name	SIZE(MB)	number of doubles	Unique doubles
bt	254.0	33,298,679	92.88%
brain	135.3	17,730,000	94.94%
lu	185.1	24,264,871	99.18%
sp	276.7	36,263,232	98.95%
sppm	266.1	34,874,483	10.24%
sweep3d	119.9	15,716,403	89.80%

Table 1: Properties of the IEEE754 floating point datasets.

Table 2 shows a study of the zero-order empirical entropy of those collections. We do not tackle larger orders, given that the cost of storing the contexts limits the character-oriented compressors with k^{th}-order modelers to small values of k (not usually larger than 16). Therefore, if we deal with larger alphabets this cost is much worse and then prohibitive in space terms.

In the table, columns from the second to the sixth show the entropy (in bits/symbol) of the first x bytes. That is, when displaying the entropy of the first byte, we consider the first byte of each number in the collection as an input symbol, and we compute the

[2]`http://www.csl.cornell.edu/~burtscher/research/FPC/datasets.html`

entropy of those symbols. When we show the entropy of the first 2 bytes, the symbols are 2-byte numbers, and so on.

The last five columns give a lower bound of the compression achieved if we compress only the first x bytes and leave the other $8 - x$ bytes in plain form. That is, in the column corresponding to the first byte, we consider the entropy of the first byte plus the remaining 7 bytes verbatim. For example in collection *bt*, the entropy of the first byte is 2.38 bits/symbol, if the rest of bytes are stored verbatim, we need 7 additional bytes, in total, 58.38 bits/symbol to represent each number.

	Entropy of the first x bytes (bits/symbol)					Space including the remainders (bits/symbol)				
Name	1 byte	2 bytes	3 bytes	4 bytes	8 bytes	1 byte	2 bytes	3 bytes	4 bytes	8 bytes
bt	2.38	8.57	15.61	21.92	23.67	58.38	56.57	55.61	53.92	23.67
brain	0.05	6.74	14.73	22.23	23.97	56.01	54.74	54.73	54.23	23.97
lu	1.31	8.41	16.39	23.43	24.47	57.31	56.41	56.39	55.43	24.47
sp	1.92	6.79	14.34	21.65	25.03	57.92	54.80	54.34	53.65	25.03
sppm	2.30	4.74	7.12	8.37	11.24	58.30	52.74	47.12	40.37	11.24
sweep3d	0.18	6.89	14.61	19.19	23.41	56.18	54.89	54.61	51.19	23.41

Table 2: Study of the zero-order entropy.

However, the previous study gives an unrealistic level of compression, since any statistical compressor would require an additional table to hold the correspondence between the original numbers and the codewords representing them. The size of such table might also give us an idea of the size of an index over that collection, as an index should have to handle all the different values of the collection plus some extra space over the size of those values. We can estimate the size (in bits) of that table as:

$$(m \times ((x * 8) + \log(m))$$

where m is the number of unique values in the collection and x the number of bytes being compressed from those numbers. That is, we estimate that for each entry, we need $(x * 8)$ bits to store the original number and $\log(m)$ bits to store the codeword representing the original symbol in the compressed text. Recall that we only compress the first x bytes, and the rest are stored verbatim.

Table 3 includes our estimation of the size of the table that maps original numbers and codewords, as well as the estimation of the size of the compressed data including the

	First k bytes compressed				
	1 byte	2 bytes	3 bytes	4 bytes	8 bytes
bt	91.22%	88.39%	87.40%	89.52%	165.97%
brain	87.51%	85.53%	85.69%	113.21%	168.00%
lu	89.55%	88.13%	88.60%	136.14%	175.40%
sp	90.50%	85.61%	85.29%	115.30%	176.86%
sppm	91.10%	82.42%	74.10%	68.10%	31.29%
sweep3d	87.78%	85.77%	85.64%	90.93%	159.70%

Table 3: An estimation of the lower bound of the compression ratio that could be achieved by a zero-order statistical compressor by compressing the first x bytes and leaving the rest verbatim.

remainders from Table 2. We can see that the best average compression ratio is achieved by compressing 3 bytes and leaving the rest uncompressed. As expected, the exception is in highly repetitive collections, in our example *sppm*. In our tests, this combination of compressed bytes and non-compressed bytes is the most suitable one to meet the two basic requirements to obtain good compression with a statistical compressor: *i)* the distribution of the source data must be biased enough to take advantage of the repetitiveness, and *ii)* the number of different values cannot be excessively large, otherwise the table that maps the original symbols to codewords would be too large, hence spoiling the compression. Observe that if we compress fewer bytes, the distribution is less biased, whereas if we compress more bytes, the number of different values increases significantly.

A more realistic way to check the correctness of our study is to use a Huffman[3] compressor. We have run that compressor over the first byte, over the first two bytes, over the first three bytes, and over the first four bytes of the numbers. As in the previous study, we measured the compression achieved taking also into account the remainders, which are not compressed. Table 4 shows the results, that are in general very similar to the bounds obtained in Table 3. When the table holding the correspondence between original symbols and codewords is large, the real Huffman compressor gives better values. The reason could be that our Huffman compressor compresses that table with a char-based bit-oriented Huffman, hence obtaining additional compression.

However, the general idea does not change. Except in collection *sppm*, which is

[3]We used a version from `http://ww2.cs.mu.oz.au/~alistair/mr_coder/shuff-1.1.tar.gz` based on Moffat and Turpin (1997).

highly repetitive, compressing the first 3 bytes and leaving the other 5 bytes uncompressed achieves the best balance between compression ratio and indexing as many bytes as possible. Therefore, this is our choice for our further data structures. Finally, we also include the compression ratio of *P7zip*[4] and Shkarin's *PPMd*[5] as two representative baseline compressors. We can see that, except in very repetitive collections, the compression of *P7zip* is rather poor and not far from that of the simpler Huffman. Also it is noticeable that a powerful compressor such as PPMd is not able to deal with real numbers successfully and is clearly overcome in most cases by zero-order Huffman. Therefore, in a general scenario we do not expect self-indexes (including those achieving high-order compression as well as grammar-based or Lempel-Ziv-based self-indexes) to be successful in space in a general scenario.

Of course, if the domain is totally different from our datasets, the optimal number of bits to index could change, but, in any case, our proposals can be adapted with no effort to index any number of bits.

	Huffman Compression				P7zip	PPMd
Name	1 byte	2 bytes	3 bytes	4 bytes		
bt	91.33%	88.44%	87.22%	112.28%	72.09%	89.58%
brain	89.06%	85.56%	85.65%	100.98%	83.57%	93.03%
lu	89.84%	88.18%	88.45%	115.15%	78.04%	96.66%
sp	90.74%	85.67%	85.17%	101.96%	74.23%	91.64%
sppm	91.26%	82.49%	73.98%	66.19%	9.01%	13.20%
sweep3d	89.11%	85.80%	85.56%	86.64%	27.90%	59.70%

Table 4: Compression achieved by: a Huffman compressor over the first x bytes ($x = 1, \ldots, 4$), as well as *P7zip* and *PPMd* run on the whole data.

4. Inverted Indexes for sequences of real numbers

As explained in the previous section, if we build an inverted index over a sequence of real numbers, it is likely that the inverted index will be composed by posting lists having in most cases only one value. This implies that the problem of space is even worse, since the inverted index needs 64 bits for each key value (each different value in the indexed

[4]`http://www.7-zip.org`

[5]We run PPMd with options *-r1 -m256 -o16*. It is available at `http://www.compression.ru/ds/`.

data) to store the key in the list of keys plus, around $\lceil \log(n) \rceil$ bits per occurrence[6] (being n the size of the collection) in the posting lists.

Therefore, the main problem is the list of keys, that could have a number of entries close to n. To solve this issue, we index only the first x bytes of each number (the more compressible ones), that is, the list of keys contains key values of x bytes.

The idea is to reduce the number of entries in the list of keys from a value that could be close to $2^{(8 \cdot 8)}$, to a value that is at maximum $2^{(x \cdot 8)}$. Obviously, this implies that the posting lists will be longer, but it is expected that the incremental encoding coupled with the gap encoding will be able to reduce the space.

In addition, we expect that the search times would not suffer too much since it is likely that the search of exact real numbers will be rare, whereas the issue of range queries will be the usual case. With our approach, a range search can be solved first accessing the index, which implies a search that involves the sign, the exponent and the first $(x * 8) - 12$ bits of the mantissa (from left to right), since as explained, the leftmost bits hold the most significant part of the number. Obviously, this is not enough if the query includes more than first x bytes. In such case the index is used to filter out all the candidate positions, and then those positions must be inspected. For this sake, we store the remaining $8 - x$ bytes of each number in the source sequence in an array of fixed length numbers. Finally, if the index holds more than the bits needed to answer the query, all the posting lists associated with each entry of the list of keys that matches the query bits will belong to the result.

Recall that the inverted index is an auxiliary structure, then the original data must be kept. However, the array of fixed length numbers storing the remaining $8 - x$ bytes of each number already store part of the original data. Yet, we still have to store the first x bytes of each number, just those indexed. Again those bytes are stored in another fixed length array, but using only the bits that are strictly necessary to represent all the possible values, since, as seen, these data are compressible. That is, if there are m different values, we use $\lceil \log(m) \rceil$ bits per number. This still gives fast direct access to

[6]Since posting values point to the position of each key in the sequence of real numbers, they would require up to $\lceil \log(n) \rceil$ bits in the case of a positional inverted index. For a block-addressing inverted index, where each block contains B real numbers, the size of each pointer would be $\lceil \log(n/B) \rceil$ bits.

any position and saves space.

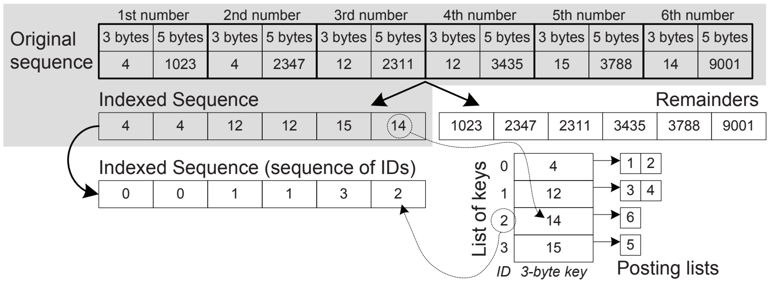


Figure 6: Indexing real numbers with an inverted index.

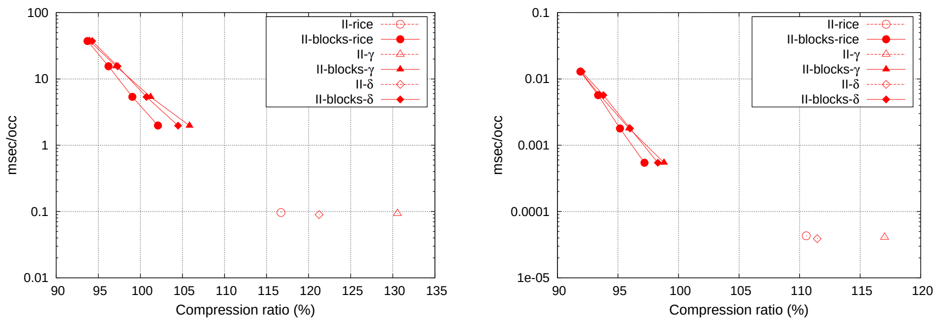
Figure 6 shows the final structure. The shaded part is not really stored and it is included only for illustration purposes. In this case, we indexed the first 3 bytes of each number. To do this, we extract the first 24 bits of each real number, and then we treat those 24 bits and the remaining 40 bits as integers. In the center-right part of the figure, it is shown the array of fixed length remainders (5 bytes each). In the lower part, we can see the inverted index, formed by the *list of keys*, that is, the different values found in the first 3 bytes of each number (sorted increasingly), and their corresponding *posting lists*.

Finally, the first 3 bytes of each number are stored in a k-bit array. In our example, there are only four different values $\langle 4, 12, 14, 15\rangle$. They are assigned the IDs $\langle 0, 1, 2, 3\rangle$ sequentially. Therefore, only two bits are needed to represent them. Finally, to compress the posting lists, we encoded the gaps of the incremental values with Rice codes.

4.1. Tunning the Inverted indexes

Using the same experimental framework described in Section 6, we present a brief study where we compare the space/time trade-off obtained by our inverted indexes when we use Rice codes[7] with the results obtained by using either γ-codes or δ-codes (Elias, 1975) for encoding the gaps in the posting lists. We consider two variants of our inverted

[7]For Rice codes, the optimal parameter b for each posting list is locally computed according to Witten et al. (1999). Given a list of n_l gaps from a posting list l, we compute $b \leftarrow \lfloor(\log_2(0.69 \sum_{i=1}^{n_l}(gap[i]))/n_l))\rfloor$. Such a value is kept along with each term in the inverted index.



(a) Non-repetitive scenario. (b) Repetitive scenario.

Figure 7: Space/time trade-offs at locating patterns of 5.5 bytes. The y axis is in logarithmic scale.

indexes: *i)* a positional inverted index (II); and *ii)* a block-addressing inverted index (II-blocks) tuned to use blocks of 512, 2048, 8192, and 32768 bytes. We present results for a non-repetitive scenario (left) and a repetitive one (right). Space is shown in the x-axis and y-axis shows average time needed to locate patterns of 5.5 bytes. Results show that among this three encoding methods Rice is the technique obtaining the best compression whereas the time performance is quite similar in all of them. That is the reason why we will use Rice codes in advance in our inverted indexes.

5. Self-indexing sequences of real numbers

In this section, we present how we have adapted the three self-indexing structures described in Section 2.2 to successfully deal with sequences of real numbers. We follow the same guidelines presented in the previous section, that is, splitting the real numbers into an indexable and a non-indexable part (x and $8-x$ bytes respectively), and briefly describe the resulting structures.

5.1. Using Wavelet Trees

A WT is a binary tree that self-indexes a sequence of symbols $S = \langle s_1, s_2, \ldots, s_n \rangle$ of a given alphabet Σ of size σ, in our case, numbers of x bytes. We use WTs with Huffman shape, that is, the represented values in the WT are the codewords that the (binary) Huffman algorithm assigns to each original symbol in S.

We used Francisco Claude's *libcds* library (Claude, 2012). *libcds* includes several compressed data structures, including an implementation of a WT with Huffman shape that occupies $nH_0(S) + o(n \log \sigma)$.

libcds uses pointers to build the structure of WTs with Huffman shape and stores at each leaf the number of occurrences of its corresponding code. The pointers represent an overhead given that they can be avoided if canonical Huffman (Schwartz and Kallick, 1964) is used. In that case, the tree structure can be simulated by placing the values of the nodes consecutively and performing the navigation through the tree by means of *rank* and *select* operations, yet this slows down the searches.

To allow the WT to store and index real numbers, we apply the same idea used for the inverted indexes: we only index the first 3 bytes of each real number, and store the remaining bytes in a fixed length array. Yet in this case, the indexed bytes do not need to be stored separately in an array of x-bytes numbers, since as we are using a self-index, those bytes are already implicitly stored within the WT.

Furthermore, if we only index the first 3 bytes, there are chances of finding runs of equal consecutive numbers in S. To avoid representing those repeated values several times and to avoid a WT traversal for each repeated symbol at those runs, instead of indexing the original sequence $\langle s_a, s_a, s_a, s_b, s_b, s_a, s_a, s_a, s_a \rangle$, our WT only represents the sequence $\langle s_a, s_b, s_a \rangle$. That is, it only stores a number when a change in the original sequence is found. Obviously, this would lose information. Yet, to maintain the runs, we also store a bitmap R with one bit per each number in S to mark the changes in S. That bit is set to 1 if that number is different from the previous one, and 0 if it is the same. In our example, the bitmap R is set to $\langle 1, 0, 0, 1, 0, 1, 0, 0, 0 \rangle$.

At search time, once we found a match, for example the second element (s_b) in the sequence stored in the WT $\langle s_a, \underline{s_b}, s_a \rangle$, to know its exact position in S, we first compute $select_1(R, 2) = 4$, which gives the position of the first s_b of that run. Then, the number of repetitions of that run is obtained as $select_1(R, 3) - select_1(R, 2) = 6 - 4 = 2$.

To further reduce the space we compress the bitmaps using the technique in Raman et al. (2002). It represents a bitmap $B[1, n]$ using $nH_0(B) + o(n)$ bits and answers *count*, *select*, and *access* in constant time. The additional $o(n)$ bits store intermediate values at fixed intervals to speed up these operations, so that the shorter the interval, the faster the operations. However, shorter intervals require more space, yielding a trade-off. The size of those intervals is usually known as *sampling* gap.

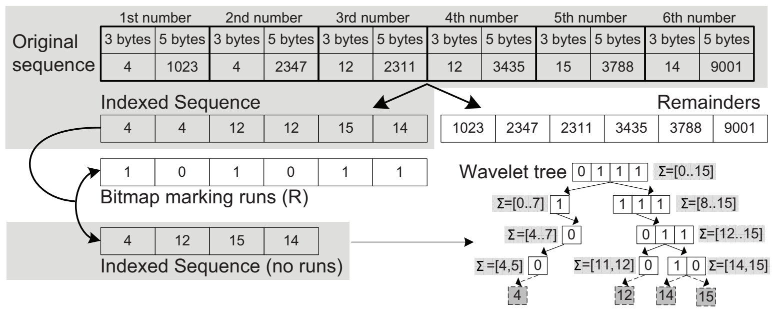


Figure 8: Indexing real numbers with a WT.

Figure 8 depicts the final structure, again only the non-shaded areas are the data structures actually stored, whereas the shaded areas are only included for illustration purposes. For simplicity, we represent the WT with regular shape (rather than using a Huffman-shaped WT) and assume that each indexed number has only 4 bits (instead of the 24 bits of a 3-byte number). That is, $\Sigma = \{0, 1, \ldots, 15\}$. Next to each node of the WT, we indicate the part of the alphabet handled by that node. In case of using a WT with Huffman shape, an additional table storing the correspondence between the original symbols and the corresponding codewords is also needed.

5.2. Using Wavelet Trees on Bytecodes

The idea is basically the same as in previous cases. We index only the first x bytes of the numbers and we store the remainders in verbatim. Even though any bytecode could be used in our experiments to create the WTBC, we chose a Plain-Huffman-based WTBC, which is known to be the best one (Brisaboa et al., 2012).

As in the previous section, by using Plain Huffman, we obtain a wavelet tree with Huffman shape, and therefore the space consumption is related to the zero-order entropy. As explained, by using bytes rather than bits as the target alphabet the compression worsens. This is easy to see since Huffman encoders, as any statistical method, need a model of the source data to gather the frequency of each original symbol, in our case, each of the $x-$byte values present in the original sequence. Then, statistical methods give shorter codewords to the most frequent original symbols and larger codewords to the least frequent ones. Therefore, the classical Huffman code might assign a codeword of just 1 bit to the most frequent symbol, whereas Plain Huffman code would assign a codeword of at least 1 byte to that symbol. This could become an important drawback if we were dealing with small alphabets, yet it is attenuated in the scenario for which Plain Huffman was designed, large texts. By using a word-based modeler and by the Heaps' law, that implies that the frequency of words is much more biased than that of characters (Moura et al., 2000), the byte-oriented Huffman compresses only around 10% worse than the bit-oriented Huffman.

Yet, let us recall what was the target of bytecodes. The use of bytes as target alphabet was proposed to obtain better compression and (mainly) decompression times. In this work, our target is different. As explained, the search time over a classical WT with Huffman shape is $O(H_0(S) + 1)$ on average, but this measure is in target symbols, that is, in bits. For example, let us assume the worst case with $x = 3$; a source data where the data distribution is completely uniform and the 2^{24} values are present. This would give a completely balanced tree and H_0 would be 24 target symbols (bits) per original symbol. Assuming the same scenario using Plain Huffman, H_0 would be 3 target symbols (bytes) per original symbol.

We can make the reasoning in another way. As any other index tree, like a B-tree, the bigger fan-out, the lower the tree, and therefore the faster the search. The WTBC with Plain Huffman has a 256-ary Huffman shape, having a fan-out of 256, with three levels, we could keep the 2^{24} values, whereas a classical WT would need 24 levels. Therefore, the traverals during the searches in WTBC would have length $O(3 + 1)$, while those traversals in the classical WT with Huffman shape would have length $O(24 + 1)$.

The assumption above is very close to reality in most of our real numbers collections. For example, considering the first 3 bytes of each number of the *lu* collection (presented in Section 3), a search for a number in a binary Huffman shaped WT requires to traverse $O(16.39+1)$ levels on average, whereas the same search in a WTBC using Plain Huffman requires to traverse $O(2.05 + 1)$ levels on average. Even in the other extreme case of a highly repetitive collection like *sppm*, the searches move from the $O(7.12 + 1)$ levels of the binary WT to the $O(0.89 + 1)$ levels of the WTBC.

To sum up, the WTBC would have search times related to the zero-order entropy, considering bytes as target alphabet. On the other hand, since the classical WT with Huffman shape is a binary tree, it would have search times related to the zero-order entropy considering bits as the target alphabet.

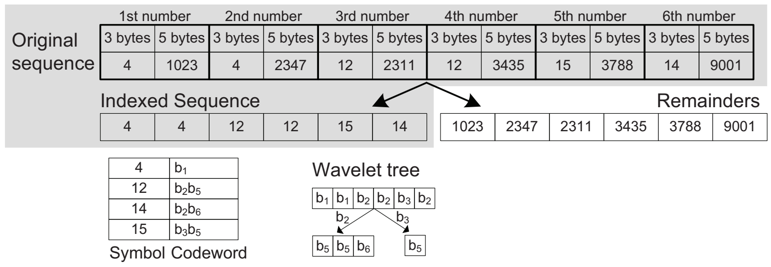


Figure 9: Indexing real numbers with a WTBC.

The structure of the WTBC follows the same guidelines showed in Section 5.1. Yet, the small height of WTBC makes it unnecessary to use bitmap R in order to avoid traversals during searches. Even though including R could help to slightly improve space requirements in very repetitive collections, the expected search performance would be similar. Note that the most frequent symbols will be given a 1-byte codeword that will be located in the root node of the WTBC. Consequently no traversal will be needed when searching for them, what nullifies the benefits of handling runs. In addition, in non-repetitive collections using R would even worsen space needs. Figure 9 shows an example with the final structure of WTBC.

5.3. Using Integer-based Compressed Suffix Arrays

As in the previous sections, we used the iCSA to index the sequence of integers formed by the first x bytes of each number, and we stored the remainders verbatim. The success of this self-index will depend, in part, on the repetitiveness present in the sequence of integers, since the space consumed by this structure is close (in practice) to $2nH_k + O(n)$ bits, although restricted to low values of k. To be successful, the k^{th} order entropy requires the presence of sequences of symbols that repetitively appear throughout the sequence being indexed. The other factor that affects the compression achieved by the iCSA is the sampling (t_Ψ, t_A, $t_{A^{-1}}$) chosen. With a dense sampling, the iCSA obtains worse compression and better search times, as the number of applications of the Ψ function to solve searches is smaller. On the contrary, with a sparse sampling the compression is better, but obviously, the searches are slower.

6. Experimental evaluation

In our tests, an isolated Intel®Xeon®-E5520@2.26GHz with 72 GB DDR3@800 MHz RAM was used. It ran Ubuntu 9.10 (kernel 2.6.31-19-server), using gcc version 4.4.1 with `-O9` options. Time results refer to CPU user time. We used the datasets presented in Section 3.

Our experiments focus on showing the memory utilization and search performance of our indexes at search time. Yet, in Section 6.4, we also compared their construction time and memory usage at indexing.

We include experiments[8] for *count*, *locate*, *extract*, and *range queries* for: *i)* a WT-based index with Huffman shape (WTH), with four different configurations obtained by setting up the sampling gap in the bitmaps to $s = \{5, 11, 21, 44\}$; *ii)* a WTBC-based index (WTBC) with five different setups obtained by varying the parameters (s,b) of the two-level structure that speeds up rank and select operations on the bytemaps.[9] Actually, we set (s,b) to (50000,10), (25000,8), (2500,8), (1500,8), and (500,8); *iii)* an integer-CSA-based self-index (iCSA) with five different configurations where the sampling parameters for Ψ, A, and A^{-1} structures are set to $t_\Psi = t_A = t_{A^{-1}} = \{8, 16, 32, 64, 256, 1024\}$; *iv)* a full positional inverted index (II); *v)* and a block-addressing inverted index (IIB) with blocks containing $b = \{512, 2048, 8192, 32768, 131072, 524288\}$ numbers. We use Rice codes for compressing the posting lists of the inverted indexes.

We performed *count* and *locate* for 1,000 numbers and show average times. That is, we *count* the number of occurrences of a given pattern or also *locate* them within each structure. We randomly chose patterns of 3 bytes (those indexed) and 5.5 bytes (3 indexed plus 2.5 bytes). The former patterns can be directly searched for by the index, whereas the latter force a further search over the array of fixed length remainders, using the candidate positions provided by the index.

[8]Source code available at `http://vios.dc.fi.udc.es/ieee64`.

[9]More details are provided in (Brisaboa et al., 2012).

For *range queries* we took a randomly chosen 3-byte indexed value and we search for all the patterns that start by that value or by the next three consecutive (3-byte) indexed values. In our experiments we show the average times of performing 100 range-queries.

We also show the time needed to: *i) extract all* the numbers of the original collection $S[1, n]$; and *ii) extract* only a portion of 200 consecutive elements from the collection $S[p - 99, p + 100]$ centered at a random position p such that $100 \leq p < n - 100$. By providing average extraction time per number, the latter experiment permits us to show the cost of *random access plus extraction*, in comparison with *extract all* operation.

6.1. Comparison on space needs

	Compressors		Indexes			Self-Indexes					
Name	Huff-3b	p7zip	II	IIB		WTH		WTBC		iCSA	
	–	–	–	512	524288	5	44	(500,8)	(50000,10)	8	1024
bt	87.22	72.09	118.31	103.84	91.94	114.88	104.49	127.11	91.67	104.82	84.18
brain	85.65	83.57	113.28	98.71	87.86	102.62	92.53	120.14	88.21	106.43	86.45
lu	88.45	78.04	119.30	104.80	91.68	117.28	106.25	125.79	91.12	110.02	90.32
sp	85.17	74.23	115.94	99.82	91.46	106.11	95.86	120.75	88.80	102.78	81.70
sppm	66.19	9.01	106.39	93.85	91.49	88.02	84.46	101.27	80.44	90.96	68.36
sweep3d	85.56	27.90	114.70	99.92	89.60	108.14	97.99	120.26	88.50	90.21	68.27
average	82.61	57.47	114.65	100.16	90.67	106.18	96.93	119.22	88.12	100.87	79.88

Table 5: Compression ratio (%).

Table 5 includes the space needs of two baseline compressors and those of the proposed indexes built on the described datasets. Even though the datasets are rather heterogeneous, we also include an *average* row to summarize the behavior of each technique. For the parameterizable structures (IIB, WTH, WTBC, and iCSA) we include two different setups obtained by tuning their parameters respectively with both the most dense and sparse sampling configurations described above, so that we achieve either a larger (and faster) or a more compact (and slower) index.

The baseline compressors included are: *i) Huff-3b*, a Huffman compressor run over the 3 first bytes of each number and leaving the rest verbatim, and *ii) P7zip* run over the whole sequence. In the compression ratio of our indexes we consider all the structures that each index requires at query time (including the original data in the case of II and IIB).

The II occupies around 15% more than the original collection, whereas IIB yields values that range from around the same size as the original collection when the block size (b) is set to 512, to a size around 10% smaller when using $b = 524,288$. WTH typically obtains a compression around 5 percentage points worse than IIB. Yet, in the highly repetitive collection (*sppm*) the R bitmap leads to an improvement around 6% over IIB. This backs our hypothesis that WT-based indexes would have a better behavior in the more stable collections, as it could be the case of the data coming from an electricity meter or a stock market. WTBC yields better compression than II and clearly overcomes WTH in most scenarios. These values are not too far from those achieved by the Huffman compressor, and even in the non-repetitive collections, they are not exaggeratedly far from those achieved by *P7zip*, being the values of the WTBC only around 10 percentage points worse. Finally, we can see that a lightweight setup of iCSA makes it the smaller indexing structure in all scenarios. In addition, it behaves particularly well in the repetitive datasets, yielding compression ratios under 70%.

6.2. Comparison on searching performance

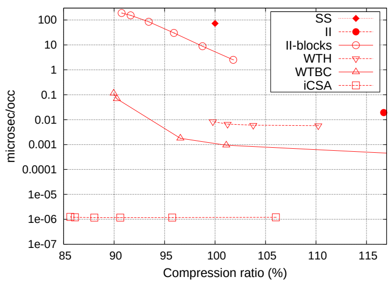
We compared the searching performance of our indexes in both a *non-repetitive* and a *repetitive* scenario. In the former we included results for datasets *bt*, *brain*, *lu*, and *sp*, whereas in the latter one we included results for both *sppm* and *sweep3d* datasets. Note that we tried to capture de average behavior for each scenario by summing the times obtained over the (either 4 or 2) datasets that compose it.

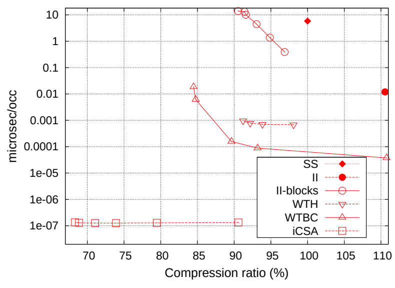
In addition, a sequential search (SS) over the original sequence of real numbers is included as a baseline.

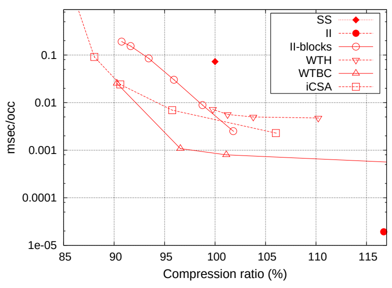
6.2.1. Searching performance: count and locate

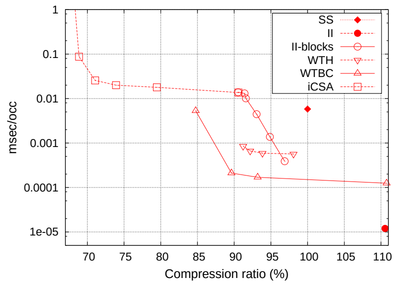
Figures 10(a) and 10(b) show the average times for counting the occurrences of 1,000 patterns of 3 bytes, which consist in the first 3 bytes of the indexed numbers. A similar behavior is observed in both the repetitive and non-repetitive scenarios.

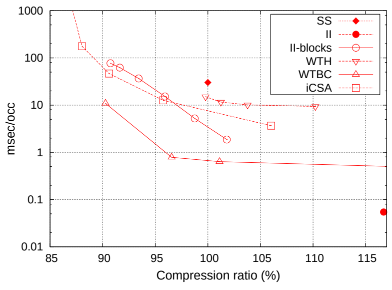
As expected, iCSA obtains by far the best space/time trade-off (count takes $O(\log n)$ time). WTs perform also fast at counting as it consists only in a top-down traversal followed by 2 rank operations to count the occurrences of the searched symbol in the


(a) Count with patterns of 3 bytes.


(b) Count with patterns of 3 bytes.


(c) Locate with patterns of 3 bytes.


(d) Locate with patterns of 3 bytes.


(e) Locate with patterns of 5.5 bytes.

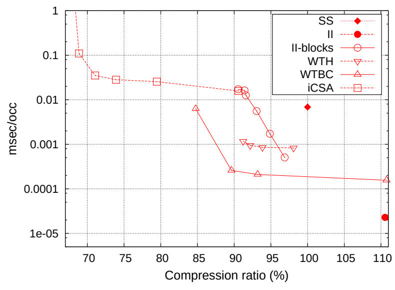

(f) Locate with patterns of 5.5 bytes.

Figure 10: Different space/time trade-offs for *count* and *locate* operations of 1000 patterns. Non-repetitive scenario (left). Repetitive scenario (right). Note that the y axis is in logarithmic scale.

corresponding leaf. Among the WT-based techniques, WTBC is not only smaller than WTH, but also clearly overcomes WTH by around 10 times at count operation.

The count time in II and IIB is proportional to the number of occurrences. This makes them slower than the self-indexing structures. Yet, II still obtains times close to those of WTH. In the case of IIB searching times worsen as the block-size increases.

In particular, when using very large blocks (containing more than 2^{18} numbers) IIB is actually slower than a sequential search over the uncompressed sequence. Yet it also requires only around 85% of its space.

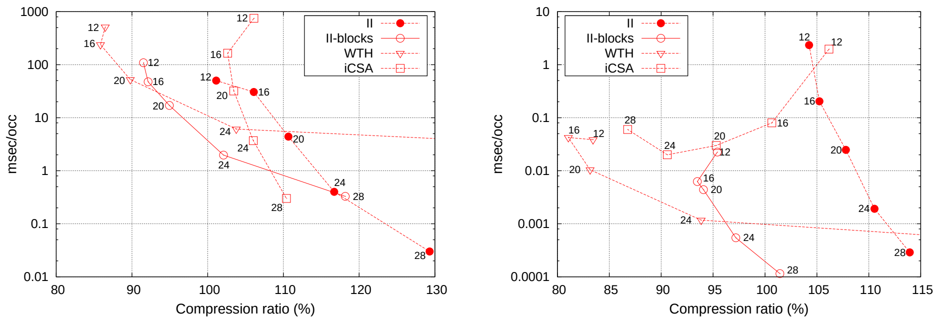
Figures 10(c) and 10(d) show that II locates patterns formed by the first 3 bytes very efficiently. As for *count*, *locate* operation can be solved directly with the index which only has to find the relevant vocabulary entries and then fetch all the values within the corresponding posting lists. As shown, the price is a structure around 15% larger than the original sequence.

IIB shows different space/time trade-offs. When using small blocks IIB is around 10-20 times faster than the sequential search (with similar space usage). WTH is also around 10 times faster than the sequential search. In the non-repetitive scenario WTH is slightly overcome by IIB. However, in repetitive datasets WTH overcomes IIB when we tune the indexes so that they use less than 98% of the space of the original data.

As shown before, iCSA is the most compact self-index. This makes it the best choice in the repetitive scenario when we want a small index (it is the only one yielding compression ratios under 80%). However, when we allow the indexes to use space over 85% of the original collection, WTBC is the clear winner.

In a non-repetitive scenario WTBC obtains the best space/time trade-off (only II improves its search times). iCSA is still a good choice (particularly as space decreases) but its locate times are around 5-10 times worse than those of WTBC for compression ratios over 95%.

Figures 10(e) and 10(f) show the performance when the indexes locate patterns formed by the first 5.5 bytes (2.5 bytes more than those indexed). We can observe similar results to those obtained when searching for just the 3 indexed bytes. Yet, the displayed search times are around 100-1000 times slower. Note that this huge difference is mainly in part due to the fact that we show time per occurrence, and now the number of occurrences is much smaller. For example, in the repetitive scenario, in datasets *lu* and *sp* we find 391,232 and 23,466,182 occurrences when searching for 3-byte patterns and respectively only 2,500 and 3,435 when searching for 5.5-byte patterns. Therefore, our indexes have to pay for locating all 3-byte patterns, and then also filter out those occurrences that do not match the array of fixed-length remainders.



(a) Non-repetitive scenario.

(b) Repetitive scenario.

Figure 11: Space/time trade-offs at locating patterns of 5.5 bytes when we vary the number of indexed bits. The y axis is in logarithmic scale.

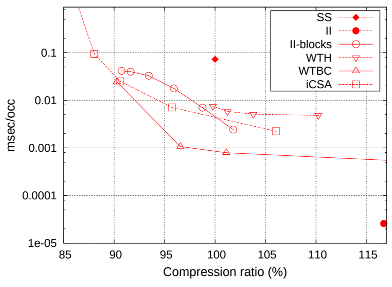
In terms of overall time needed to locate all the occurrences of a given pattern, the gap between locating either a 3-byte pattern or a 5.5-byte pattern is negligible. That is, the cost of verifying an occurrence over the array of fixed-length remainders is small.

Dependency on the number of bits indexed:

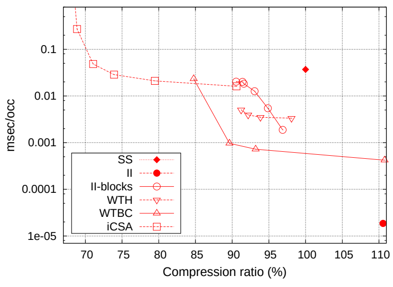
We also present an experiment where we show how the number of bits indexed determines the space-time trade-off obtained by our indexes. In this case, we set the block size of our IIB to 512 bytes; we tuned the sampling gap in WTH to 11; and we set the sampling rate in the iCSA to $t \leftarrow 8$. Figure 11 shows the results obtained when we index $x' \in \{12, 16, 20, 24, 28\}$ bits respectively (such values are depicted along with the marks in the figure), and then perform locate searches for 5.5 byte patterns. In general terms, indexing more bits leads to faster indexes, yet space usage also increases. The main exception to this is iCSA in the repetitive scenario, where the more bits are indexed the more repetitiveness is found and better compression is obtained. We can see that (particularly in the non-repetitive scenario) indexing 3-bytes (24 bits) is typically a reasonable choice.

6.2.2. Searching performance: dealing with range queries

Our next experiment focuses on performing range queries. Recall that now our indexes must retrieve all the occurrences of four consecutive 3-byte indexed numbers. That is, it is expected that we have around four times the cost as for locate, but also around

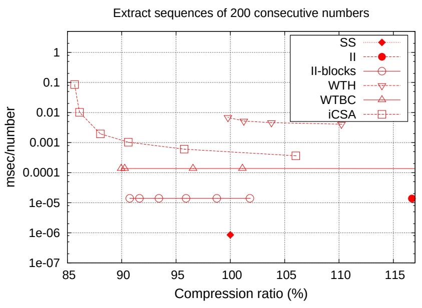


(a) Non-repetitive scenario.

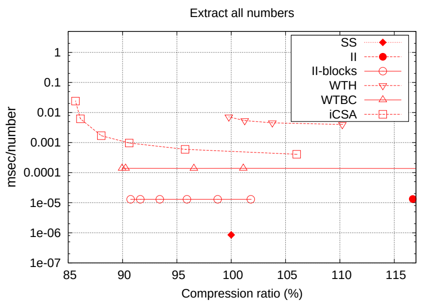


(b) Repetitive scenario.

Figure 12: Space/time trade-offs at performing range queries. The y axis is in logarithmic scale.



(a) Extracting 200 numbers.



(b) Extracting all the numbers.

Figure 13: Space/time trade-offs for *extract* operation. The y axis is in logarithmic scale.

four times its number of occurrences. Therefore, the average time per occurrence should be similar in both cases. This is exactly what we can see if we compare the results in Figures 12(a) and 12(b) with those in Figures 10(c) and 10(d).

6.3. Recovering the original data: Extract operation

We measured the time needed to recover the original data (*extract*). We only present results for the non-repetitive scenario, as the results obtained were exactly the same as in the repetitive case. Firstly, we show the time required to extract a subsequence of 200 numbers from 1,000 random positions. Figure 13(a) displays the average time needed to extract each number (in msec/number). II and IIB require a small amount of time to merge the first x bytes and the other $8 - x$ bytes to make up the final numbers. Therefore, it is around 10 times slower than the straightforward extraction of the plain representation. WTBC, which is known to be a fast technique at extraction (Brisaboa et al., 2012), shows up as the fastest self-indexing structure at extract being around 10 times faster than iCSA. WTH is the slowest technique. Figure 13(b) shows exactly the same behavior. This implies that the random access does not hurt significantly the extraction speed, showing that this important requirement is nicely handled by all the indexes.

Our IIs obtain the best space/extract time trade-off. They recover around 7.5×10^7 numbers per second. Yet, extract in the WTs is more expensive, as the WTs have to perform *rank* operations from the root to the leaves to recover the original numbers (the current implementation in *libcds* library takes no advantage of extracting values in consecutive positions $1..n$, to recover $S[1..n]$). WTH recovers around 0.25×10^6 numbers per second and, due to its lower tree height and a more refined implementation, WTBC around 10^7 numbers per second. Finally, iCSA with a regular sampling (i.e. a compression ratio around 90%) recovers around 10^6 numbers per second, showing the cost of accessing to a compressed Ψ (plus a rank operation over D).

6.4. Memory usage and time at indexing

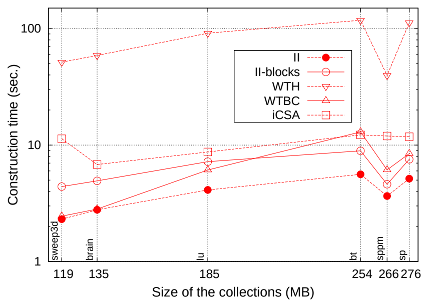
We present additional information to show the construction cost of our indexes. We tuned the indexes in their fastest (and most space consuming) configuration, and measured both cpu user-time (in seconds) and peak memory usage[10] (in MB).

Figure 14 presents the results for each dataset. The x axis shows the size of the datasets (sorted increasingly). Note that our indexing programs were not heavily optimized to minimize memory utilization and construction time. Yet, we can see that, except WTH, which requires around 100 seconds, all the other structures are built in less than 15 seconds. Regarding memory consumption, our indexes use around 2-4 times the size of the source dataset.

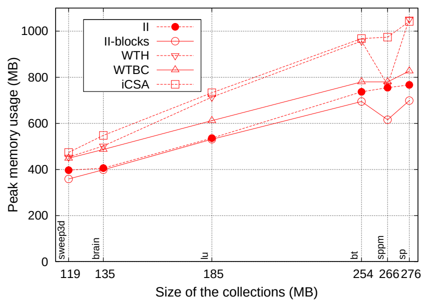
7. Conclusions

In this work, we have presented succinct indexes for collections of IEEE 754 double precision floating point numbers. The main idea is simple. We split each real number

[10]Memory usage was obtained from file /proc/<pid>/status.



(a) Construction time.



(b) Memory usage.

Figure 14: Construction time (y axis in logarithmic scale) and peak memory usage for our indexes.

into two parts: an incompressible part (the last 5 bytes) that are stored apart, and a compressible part (the first 3 bytes) that can be indexed with any general purpose integer-based indexing technique. By doing so, we have presented five different indexing alternatives from three well known families of indexes. Two of them are based on inverted indexes (II and IIB), WTH and WTBC are based on wavelet trees, and finally we also included a variant based on Compressed Suffix Arrays (iCSA). Yet, note that any integer-based indexing structure from the state-of-the-art can be used. Apart from indexing, this approach is applicable to compression. Indeed, following the ideas in Brisaboa et al. (2013), we are working in a new method to compress sequences of real numbers that will permit direct access to any position.

The size of the resulting structures (including both the data and the index) is between 30% shorter and 20% larger than the original sequence. As expected, the improvements in search time are remarkable in the case of inverted indexes with full positional information, being up to around 8,000 times faster than sequentially scanning the original collection. The more compact indexes, such as the block addressing IIs, the wavelet-trees, and iCSA yield different interesting space/time trade-offs.

According to the results of our experiments in the most repetitive collection, we expect that the WTBC-based indexes will obtain better results when applied to rather stable collections, as expected in the case of SmartGrid or stock exchange markets. This repetitive scenario is also good for iCSA in terms of space needs, but its performance at searches degrades in excess.

To sum up, if one aims at obtaining a good search speed with reasonable (high) space consumption, the II is the clear winner. Obviously, it is only beaten by the plain representation in the *extract* operation. However, if one is mainly concerned about space, the choice should be the iCSA. It occupies up to 30% less space than the original collection and still obtains fast locate times (being the fastest technique at count by far), particularly in non-repetitive collections. For those mainly interested in fast extraction of the original data, and reasonably good space requirements and searches, the block-addressing IIs are a good choice. Finally, considering a general scenario, WTBC, is probably the best choice. It obtains good compression (10-15% less space than the original data), it yields fairly good performance in search operations, and is reasonably fast at extract operation.

8. Acknowledgments

This work was partially supported by Xunta de Galicia (co-founded with FEDER) [GRC2013/053]; Ministerio de Economía y Competitividad [TIN2013-46238-C4-3-R and TIN2013-47090-C3-3-P]; CDTI, AGI, and Ministerio de Economía y Competitividad [CDTI-00064563/ITC-20133062]; and (for Alberto Ordóñez) by Ministerio de Ciencia e Innovación [AP2010-6038 (FPU Program)].

References

Brisaboa, N., Fariña, A., Ladra, S., Navarro, G., 2012. Implicit indexing of natural language text by reorganizing bytecodes. Information Retrieval 15 (6), 527–557.

Brisaboa, N., Fariña, A., Navarro, G., Paramá, J., 2007. Lightweight natural language text compression. Information Retrieval 10 (1), 1–33.

Brisaboa, N., Ladra, S., Navarro, G., 2013. DACs: Bringing direct access to variable-length codes. Information Processing and Management (IPM) 49 (1), 392–404.

Burtscher, M., Ratanaworabhan, P., 2009. Fpc: A high-speed compressor for double-precision floating-point data. IEEE Transactions on Computers 58 (1), 18–31.

Claude, F., Oct. 2012. libcds compact data structures library. `http://libcds.recoded.cl/`.

Culpepper, J. S., Moffat, A., 2005. Enhanced byte codes with restricted prefix properties. In: Proc. 12th Int. Symp. on String Processing and Information Retrieval (SPIRE 05). LNCS 3772. Springer, pp. 1–12.

Elias, P., 1975. Universal codeword sets and representations of the integers. Information Theory, IEEE Transactions on 21 (2), 194–203.

Engelson, V., Fritzson, D., Fritzson, P., 2000. Lossless compression of high-volume numerical data from simulations. In: Proceedings of Data Compression Conference (DCC 00). pp. 574–586.

Fariña, A., Brisaboa, N., Navarro, G., Claude, F., Places, A. S., Rodríguez, E., 2012. Word-based self-indexes for natural language text. ACM Transactions on Information Systems 30 (1), article 1.

Fariña, A., Ordóñez, A., Paramá, J. R., 2012. Indexing sequences of IEEE 754 double precision numbers. In: Proc. Data Compression Conference (DCC 12). pp. 367–376.

Garmany, J., Karam, S., Hartmann, L., Jain, V. J., Carr, B., 2008. Oracle 11g New Features Get started fast with Oracle11g enhancements. Shroff Publishers/Rampant.

Grossi, R., Gupta, A., Vitter, J. S., 2003. High-order entropy-compressed text indexes. In: Proc. Symposium on Discrete Algorithms (SODA 03). pp. 841–850.

Grossi, R., Gupta, A., Vitter, J. S., 2004. When indexing equals compression: experiments with compressing suffix arrays and applications. In: Proc. Symposium on Discrete Algorithms (SODA 04). pp. 636–645.

Grossi, R., Vitter, J., 2000. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: Proc. 32nd ACM Symposium on Theory of Computing (STOC 00). pp. 397–406.

Huffman, D. A., 1952. A Method for the Construction of Minimum-Redundancy Codes. Proc. of the IRE 40 (9), 1098–1101.

IBM Software, May 2012. Managing big data for smart grids and smart meters.
URL `ftp://public.dhe.ibm.com/software/pdf/industry/IMW14628USEN.pdf`

IEEE Std 754-200, 2008. IEEE Standard for Floating-Point Arithmetic.

Jacobson, G., 1989. Space-efficient static trees and graphs. In: Proc. 30th IEEE Symp. on Foundations of Computer Science (SFCS 89). pp. 549–554.

Knuth, D. E., 1973. The Art of Computer Programming. Vol. 3: Sorting and Searching. Addison-Wesley.

Larsson, J., Moffat, A., 2000. Off-line dictionary-based compression. Proceedings of the IEEE 88 (11), 1722–1732.

Mäkinen, V., Navarro, G., 2008. On self-indexing images - image compression with added value. In: Proc. Data Compression Conference (DCC 08). pp. 422–431.

Manber, U., Myers, G., 1993. Suffix arrays: a new method for on-line string searches. SIAM Journal on Computing 22 (5), 935–948.

Manber, U., Wu, S., Oct. 1993. GLIMPSE: A tool to search through entire file systems. Technical Report 93–34, Dept. of Computer Science, University of Arizona.

Moffat, A., 1989. Word-based text compression. Software Practice and Experience 19 (2), 185–198.

Moffat, A., Turpin, A., oct 1997. On the implementation of minimum redundancy prefix codes. Communications, IEEE Transactions on 45 (10), 1200–1207.

Moura, E., Navarro, G., Ziviani, N., Baeza-Yates, R., 2000. Fast and flexible word searching on compressed text. ACM Transactions on Information Systems 18 (2), 113–139.

Munro, J. I., 1996. Tables. In: Proc. 16th Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 96). LNCS 1180. Springer, pp. 37–42.

Navarro, G., 2012. Wavelet trees for all. In: Proc. 23rd Annual Symposium on Combinatorial Pattern Matching (CPM). LNCS 7354. Springer, pp. 2–26.

Navarro, G., Mäkinen, V., 2007. Compressed full-text indexes. ACM Computing Surveys 39 (1), article 2, 61 pages.

Navarro, G., Russo, L., 2011. Space-efficient data-analysis queries on grids. In: Proc. 22nd Annual International Symposium on Algorithms and Computation (ISAAC 11). LNCS 7074. Springer, pp. 323–332.

Raman, R., Raman, V., Rao, S. S., 2002. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proc. Symposium on Discrete Algorithms (SODA 02). pp. 233–242.

Ratanaworabhan, P., Ke, J., Burtscher, M., 2006. Fast lossless compression of scientific floating-point data. In: Proc. Data Compression Conference (DCC 06). pp. 133–142.

Rice, R. F., 1979. Some practical universal noiseless coding techniques. Tech. rep., Jet Propulsion Laboratory.

Sadakane, K., 2003. New text indexing functionalities of the compressed suffix arrays. Journal of Algorithms 48 (2), 294–313.

Schwartz, E. S., Kallick, B., 1964. Generating a canonical prefix encoding. Communications of the ACM 7 (3), 166–169.

Williams, H. E., Zobel, J., 1999. Compressing integers for fast file access. The Computer Journal 42 (3), 193–201.

Witten, I. H., Moffat, A., Bell, T. C., 1999. Managing gigabytes (2nd ed.): compressing and indexing documents and images, 2nd Edition. Morgan Kaufmann Publishers Inc.

Zicari, R. V., Aug. 2012. Big data: Smart meters – interview with markus gerdes.
URL `http://www.odbms.org/blog/2012/06/big-data-smart-meters-interview-with-markus-gerdes/`

Zukowski, M., Heman, S., Nes, N., Boncz, P., 2006. Super-scalar ram-cpu cache compression. In: Proc. 22nd International Conference on Data Engineering (ICDE 06). pp. 59–70.

Curriculums

Antonio Fariña

Antonio Fariña obtained his PhD in Computer Science in 2005 at the University of A Coruña. Today, he is an associate professor in the same university. His research is mainly focused on text compression and indexing, compact data structures, and Geographic Information Systems.

Alberto Ordóñez

Alberto Ordóñez received his M.S. degree in Computer Science in 2009 and a Master degree in 2011 both from the University of A Coruña. He is currently a PhD student at the same university under the supervision of Nieves R. Brisaboa (Univ. of A Coruña) and Gonzalo Navarro (Univ. of Chile). His areas of interest are compressed text retrieval, metric spaces and persistent data structures.

José R. Paramá

José R. Paramá obtained his PhD in Computer Science in 2001 at the University of A Coruña. He is currently associate professor in the same university. His areas of interest are digital libraries, compressed text retrieval, deductive databases and spatial databases.