Indexing sequences of IEEE 754 double precision numbers.*

Antonio Fariña, Alberto Ordóñez, and José R. Paramá
Database Lab, University of A Coruña, A Coruña, Spain.
{antonio.farina, alberto.ordonez, jose.parama}@udc.es

Abstract

In the last decades, much attention has been paid to the development of succinct data structures to store and/or index text, biological collections, source code, etc. Their success was in most cases due to handling data with a relatively small alphabet size and to typically exploit a rather skewed distribution (text) or simply the repetitiveness within the source data (source code repositories, biological sequences of similar individuals).

In this work, we face the problem of dealing with collections of floating point data that typically have a large alphabet (a real number hardly ever repeats twice) and a less biased distribution. We present two solutions to store and index such collections. The first one is based on the well-known inverted index. It consumes space around the size of the original collection, providing appealing search times. The second one uses a wavelet tree, which at the expense of slower search times, obtains slightly better space consumption.

1 Introduction

Although the web still attracts much efforts, new big challenges deserve also attention. Stock exchange markets, geographic information systems or the forthcoming electrical grids (SmartGrid) are examples of producers of large amounts of floating point data, which should be transmitted through networks and finally stored.

Therefore, the use of compression techniques is mandatory to minimize both the flow of information through the network and the storage needs. Moreover, once the data are stored, in cases like the SmartGrid, it is required to access these data efficiently to help real-time decision-making processes. This last requirement implies that we also have to develop the suitable indexing techniques.

Current indexing techniques are mature, providing the required performance for most scenarios. However, the use of these techniques implies that additional space should be reserved for them. When dealing with large amounts of information those

*This work was partially supported by Ministerio de Ciencia e Innovación [grants TIN2009-14560-C03-02, TIN2010-21246-C02-01, and CDTI CEN-20091048], and Xunta de Galicia [grant 2010/17].

indexes might became prohibitive in terms of space. A typical solution is to compress the data with techniques that permit random decompression from any position. This both saves space and permits indexes to be built over the compressed data.

Compression techniques for floating point data [3, 15], or those general purpose techniques that are successful in compressing this type of data (like *P7zip*), require that the decompression must start at the beginning of the compressed data. That makes such compressors unsuitable to be coupled with indexing structures.

Therefore, we have a trade-off, if we want sub-linear search time over sequences of real numbers, we have to store the data uncompressed in order to use an index over them, hence increasing the space requirements. On the other hand, if we want to save space, we have to keep the data compressed and then, to perform searches, we have to decompress the data and then run a linear-time search algorithm.

The web has boosted the research in several structures that have some very convenient characteristics to store and/or index text. Largely, they base their qualities in a relatively small alphabet, a quite skewed data distribution, and data that do not suffer from frequent changes. This last requirement is also the case of our scenario, since we tackle data that once they are produced, they should be stored for future decision-making processes and never change. In this work, we aim at adapting those structures to index floating point data.

Inverted indexes [9] have received much attention due to their application to index text and more specifically to index the web. The main current research focus is on saving space, since as typical indexes, they are an auxiliary structure added to the original information.

At the same time, new structures (self-indexes) have integrated compression with indexing. They take space proportional to the compressed text, replace it, and permit fast indexed searching on it [14]. Those indexes can *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 can also *count* the number of occurrences of a pattern string much faster than just locating them.

In this work, we present two contributions. First, we study the adaption of inverted indexes to the indexation of real numbers. We will show that, in our experiments, a structure around 6-19% larger than the original sequence obtains locate times that are between 50% and around 2000 times faster than the sequential search. Second, we study the use of self-indexes to index and store sequences of real numbers. We used a wavelet tree (WT) [5], which is a very flexible structure that allows both to store and index data. In our experiments, the resulting structure has a size that is up to 26% smaller than the original one, whereas the *locate* times are between 5% and 5 times faster than the sequential search. The price is that when using the WT, the *extract* operation needed to recover a portion of the original sequence requires a considerable amount of time, whereas that time is negligible in the case of inverted indexes. Yet, it is expectable that in our scenario the typical query will consist in checking if a data source surpassed a certain threshold, and displaying the whole original data will be a rather infrequent operation.

As an example of these queries, let us suppose that we have the measures of the electricity production corresponding to each minute of the last years of a power plant. The SmartGrid would be interested in the timestamps where the production was within a given range. This is important, because when a solar plant produces a certain amount of electricity, other fossil fuel plants should decrease their production to avoid a waste of money, since electricity cannot be stored and the solar plants have priority. Comparing the hour of the day and the weather conditions of those past periods of time with the current conditions would allow the SmartGrid to predict in advance (that is, this implies real time restrictions) the production of the fossil fuel plants, in order to avoid a production excess.

The outline of this paper is as follows. Sections 2 and 3 present some related work and the 64-bit IEEE 754 format. Sections 4 and 5 present our two contributions: inverted indexes over real numbers and a WT over real numbers. Section 6 shows our experiments. Finally, Section 7 shows our conclusions and directions for future work.

2 Related Work

An inverted index [9] is an auxiliary indexing structure that permits to speed up searches over a given collection. It is composed of a list of search keys, and for each key, a posting list that stores the positions in the original sequence where that particular key can be found. To reduce the space requirements both the source data and the posting lists can be compressed.

To compress the data, usually text, several compression techniques have been used. Tagged Huffman [13] and the dense codes [1] are well-known examples. For repetitive data, grammar-based compressors [10] proved also to be successful. To compress the posting lists incremental encoding has been coupled with different techniques to encode the gaps, like for example, byte-codes [18], Rice codes [16], or more recently PforDelta [8]. Another way of saving space is that the pointers of the posting lists point to blocks instead of pointing to exact positions [12]. When blocks contain several occurrences of a searched key, the corresponding posting list has fewer elements. The price is that now the index is only able to filter out some blocks and a sequential scan within those blocks is required to find the exact locations of the searched key.

Another more recent approach are self-indexes, which were developed for relatively small alphabets and for sources with a rather skewed distribution. One of the structures that have been used to build self-indexes is the WT. One of its many features is to store and index sequences of numbers (or codewords representing the entries in a given alphabet). There are also examples where they were used as self-indexes in cases when the size of the alphabet is larger than in the case of text [11].

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

The original WT is a balanced binary tree. At the root node, the original alphabet is divided into two halves. The only stored structure is a bitmap that assigns each symbol of the original sequence to one of the halves of the alphabet. The symbols of one half go to one child and the rest to the other. Recursively, each child handles in the same way, the part of the sequence it received. The WT reduces the searches to *rank* and *select* operations on the bitmaps. Since then, they have been used for different purposes, and with different shapes, for example Huffman shape [7].

3 Basics

We consider the double precision 64-bit IEEE 754 format. This standard divides the floating point numbers into three components: *(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.

When we have a collection of real numbers the alphabet size is 2^{64}. Therefore, it is likely that most of those numbers are unique, and very few repetitions would be found. That is, the entropy is expected to be high. Yet, no assumptions can be made and highly repetitive sequences can exist depending on the case.

The exponent indicates the magnitude of the number. In general, collections of floating point numbers are measures or results of some computational process or real phenomenon. Therefore, we expect that, despite the numbers could be significantly different, their magnitude should be similar. 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 [3, 15]. The reason is that the mantissa (m) is usually normalized to a number $1 \leq m < 2$, 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. Therefore, in most cases, the last part of the mantissa is the hardest part to compress.

4 Using Inverted Indexes

As explained in the previous section, an inverted index over a sequence of real numbers is likely to became a set of posting lists formed in most cases by only one value. This leads to storing 64 bits in the list of keys plus, around $\log(n)$ bits per occurrence (being n the size of the collection) in its posting list. This results in a large index.

Instead, we index only the first k bits of each number (the more compressible ones) and store the remaining $64 - k$ bits in an array of fixed length numbers.

This fits quite nicely with the search for real numbers, as we expect that seeking exact real numbers will be rare, whereas the issue of range queries will be the usual case. This results in a search involving the sign, the exponent and the first $k - 12$ bits of the mantissa (from left to right), since as explained, the leftmost bits hold the most significant part of the number. Therefore, if the index includes the needed bits, the query is directly solved. If the index does not hold all the bits needed to answer the query, the index is used over the first part of the pattern (the first k bits of the number) to filter out all the candidate positions, and then those positions are inspected in the array of fixed length remainders (the last $64 - k$ bits of each number).

Therefore the question is: how many bits should we index? First, we consider to calculate the entropy of the first k bits of all numbers, but this would give an unrealistic level of compression, since any statistical compressor would require a table to hold the correspondence between the original numbers and the codewords representing them. Note that the size of such table should also be estimated.

A more realistic way is to use a Huffman compressor. We 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. Then, for those four cases, we compute the compression achieved taking also into account the remainders, which are not compressed. If the differences are not significant, we just index as many bytes as possible.

Yet along with our inverted index we need to maintain the original data to quickly recover the original sequence or just the i^{th} number. Therefore, we store the part of the numbers that is indexed in another fixed length array, but using only the bits that are strictly necessary to represent all the possible values, given that this is the part of the numbers that has been proven to be compressible. That is, if there are m different values in the indexed bytes, we use $\lceil \log(m) \rceil$ bits per number. This gives fast direct access to any position and saves space.

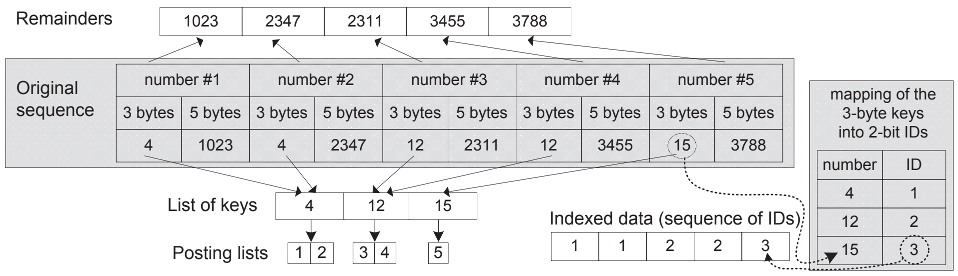


Figure 1: Indexing real numbers with an inverted index.

Figure 1 shows the proposed structure, the shaded part is not kept and it is included only for illustration purposes. In this case, we assume that we index the first three bytes of each number. To do this, we consider the 64 bits of each floating point number and we extract the first 24 bits. We treat those 24 bits and the remaining 40 bits as integers. In the upper part of the figure, we can see 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 three bytes of all the numbers of the collection (sorted increasingly), and their corresponding posting lists.

The indexed data is stored in a k-bit array. In the figure, there are only three different keys. They are assigned the IDs $\langle 1, 2, 3 \rangle$ sequentially. Therefore, only two bits are needed to represent them. For clarity, we include a table with the mapping from the original 24-bit keys to their 2-bit IDs in Figure 1. Finally, to compress the posting lists, we encoded the gaps of the incremental values with Rice codes.

5 Wavelet Trees

As explained a WT is a binary tree that stores and indexes a sequence of symbols ($S = s_1, s_2, \ldots, s_n$) of a given alphabet σ, in our case, numbers of 3-4 bytes. Our WT has a Huffman shape. That is, the represented numbers are the codewords that the Huffman algorithm assigns to each original symbol in S.

We used Francisco Claude's library *libcds* [4]. *libcds* includes, among other compressed data structures, an implementation of a WT with Huffman shape that occupies space $nH_0(S) + o(n \log(\sigma))$, where $H_0(S)$ is the zero-order entropy of S.

In *libcds*, when the WT has Huffman shape, the structure of the tree is constructed by pointers. These pointers represent an overhead (each pointer occupies 4 bytes) that can be avoided if a canonical Huffman [17] is used. As in the case of a balanced WT of *libcds*, pointers can be removed since the navigation through the tree can be done performing *rank* and *select* operations, although this slows down the searches.

We follow the same idea used for the inverted indexes. We only index the first 3-4 bytes of each number, and store the remaining bytes in a fixed length array.

We expect that in many continuous data sources, it is likely that the numbers do not vary significantly and the magnitude of the number (the exponent) and probably the leftmost bits of the mantissa will remain rather stable.

Therefore, if we are indexing only, for example, the first three bytes, there are chances of finding runs of several equal numbers in S. To avoid storing equal consecutive numbers, if we have to store the sequence $s_x, s_x, s_x, s_y, s_y, s_x, s_x, s_x, s_x$, the WT only stores the sequence s_x, s_y, s_x, that is, it only stores a number when a change in the sequence is found. Obviously, this would lose information. Yet to maintain the runs, we store a bitmap B with one bit per 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 equal. In our example, the bitmap B is $1, 0, 0, 1, 0, 1, 0, 0, 0$.

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

B is compressed with a bitmap technique [6] (also available in *libcds*) that uses $n + o(n)$ bits. The additional $o(n)$ bits are needed to provide efficient *rank* and *select* operations. The same technique is used in the nodes of the WT.

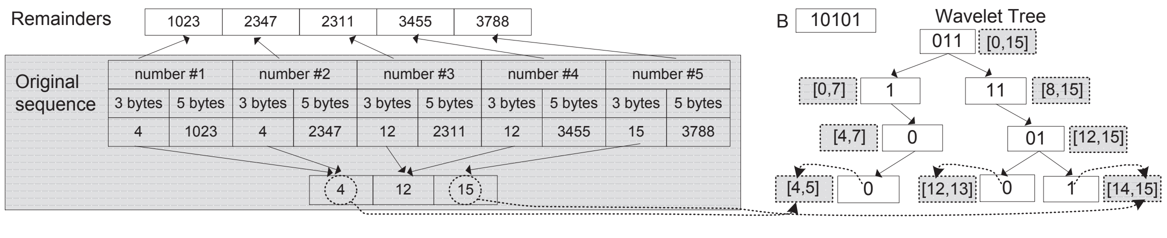


Figure 2: Indexing real numbers with a WT.

Figure 2 shows the final structure, again the shaded area indicates data that is not actually stored and that is included for illustration purposes. For simplicity, we represent the WT with regular shape and assume that each indexed number has only four bits (instead of the 24 bits of a 3-byte number). 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.

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 several data sets downloaded from Martin Burtscher's site.[1] We used collections *lu*, *bt*, *sp*, *sppm*, and *sweep3d*. Interested readers can find details of the collections in the web page.

Table 1 provides some relevant information about the data sets. It also displays the compression ratio[2] obtained by a Huffman compressor applied over the first x bytes and leaving the rest of bytes uncompressed. Except in collection *sppm*, which is highly repetitive, compressing the first 3 bytes and leaving the rest of bytes uncompressed achieve the best balance between compression ratio and indexing as many bytes as possible. Therefore, this is our choice in our experiments. Finally, we also include the compression ratio of *P7zip* (`http://www.7-zip.org`). We can see that, except in very repetitive collections, the compression is rather poor and not far from that of the simpler Huffman. Therefore, in a general scenario we do not expect that both grammar-based or Lempel-Ziv-based self-indexes to be successful in space.

Description of the collections used				Huffman Compression				P7zip
Name	SIZE(MB)	#doubles	Unique doubles	1 byte	2 bytes	3 bytes	4 bytes	
bt	254.0	33,298,679	92.88%	91.33%	88.44%	87.22%	112.28%	72.09%
brain	135.3	17,730,000	94.94%	89.06%	85.56%	85.65%	100.98%	83.57%
lu	185.1	24,264,871	99.18%	89.84%	88.18%	88.45%	115.15%	78.04%
sp	276.7	36,263,232	98.95%	90.74%	85.67%	85.17%	101.96%	74.23%
sppm	266.1	34,874,483	10.24%	91.26%	82.49%	73.98%	66.19%	9.01%
sweep3d	119.9	15,716,403	89.80%	89.11%	85.80%	85.56%	86.64%	27.90%

Table 1: Properties of the data sets and compression achieved by: a Huffman compressor over the first x bytes ($x = 1, \ldots, 4$), and *P7zip* run on the whole data.

We include experiments[3] for *count*, *locate*, and *extract* for our WT-based index (WT), our full positional inverted index (II), and a block-addressing inverted index (II-b) with blocks containing $b = \{1024, 2048, 4096, 8192\}$ numbers.

We performed *count* and *locate* for 1000 numbers. We randomly chosen patterns of 3 bytes (those indexed), 4 bytes (3 indexed plus 1 byte), and 5.5 bytes (3 indexed plus 2.5 bytes). The last two cases force a further search over the array of fixed length remainders, using the candidate positions provided by the index.

When performing *count*, in the case of the inverted indexes, searches involving only the indexed bytes are solved directly by the index as we also store the number of occurrences of each key. However, for searches involving more bytes, we have to run

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

[2]The size of the compressed file as a percentage of its original size.

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

a *locate* search over the first 3 bytes to find a list of candidate positions that should then be inspected in the fixed length array of remainders. Therefore, if the patterns are more than 3 bytes long, the *count* time required is almost the same as for *locate*.

Figure 3 shows the average times after the application of the corresponding search operation over all the collections. We try to capture the average behavior over different sources. We include as a baseline a sequential search (SS) over the original sequence of real numbers.

Figure 3(a) shows the count time for patterns of length 3. In the case of inverted indexes, this is negligible given that the pattern is indexed and, as explained, that operation only includes an access to the key entry in the inverted index. The WT needs to perform *select* operations from the leaves to the root that require some computational effort, which implies a worsening in the count times. However, it is still 5 times faster than the sequential search.

Figure 3(b) shows that the II locates patterns formed by the first three bytes of real numbers around 2000 times faster than the sequential search. The price is a structure around 15% larger than the original sequence. The block-addressing inverted indexes give different space/time trade-offs, where the time ranges from 11 times faster to 70% faster than the sequential search. Finally the WT, which is the shortest structure, is around 5 times faster than the sequential search.

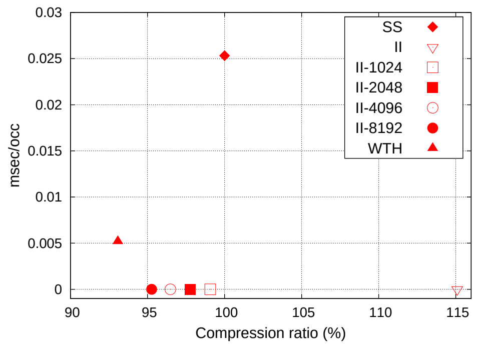
Figure 3(c) shows the case of searching the first 4 bytes (one more than those indexed). As explained, the count operation is almost the same as locate. Still, the II is around 1700 times faster than the sequential search. The block-addressing IIs obtain also similar improvements as in the case of 3 bytes. The WT improves the values of the sequential search in almost 4 times.

Finally, locating patterns formed by the first 5.5 bytes (see Figure 3(d)) yields worse values due to bit manipulations not aligned to byte values. The II is around 360 times faster than the sequential search. The block-addressing IIs obtain improvements from 7 times to 50% faster than the sequential baseline. In this case, the WT only achieves an improvement of 5% with respect the sequential search, yet it is still able to obtain a 10% reduction in space.

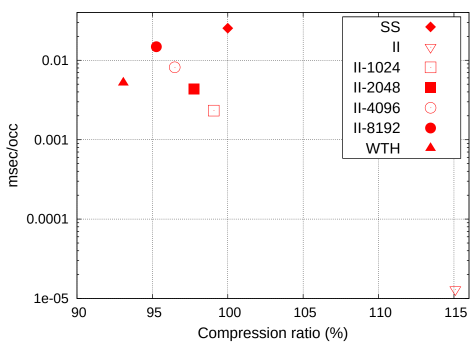
We also measured the time to recover the original data (*extract*). Our IIs recover around 115×10^6 numbers per second. Yet, extract in the WT is more expensive, as the WT has to perform *rank* operations from the root to the leaves to recover the original numbers (the current implementation in *libcds* library takes no advantage of extracting values in consecutive positions $1..n$, to recover $S[1..n]$). The WT recovers around 0.4×10^6 numbers per second.

7 Conclusions

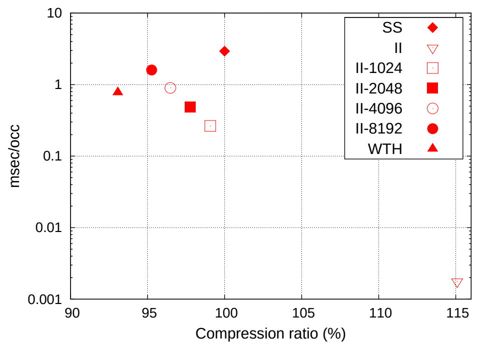
In this work, we have presented succinct indexes for collections of IEEE 754 double precision floating point numbers. The size of the resulting structures (including both the data and the index) is close to the original size: between 26% smaller and 19% larger than the original sequence. The improvements in search time are remarkable in the case of inverted indexes to exact positions, being up to around 2000 times faster



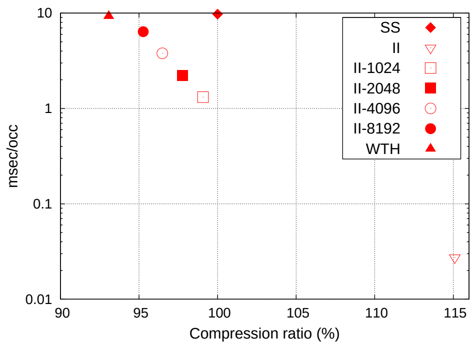
(a) Count with patterns of 3 bytes.



(b) Locate with patterns of 3 bytes.



(c) Locate with patterns of 4 bytes.



(d) Locate with patterns of 5.5 bytes.

Figure 3: Different space/time trade-offs for *count* and *locate* operations. In subfigures (b), (c), and (d), the y axis is in logarithmic scale.

than sequentially scanning the original collection. The block addressing IIs and the WT structure yield different interesting space/time trade-offs.

We expect that the WT would obtain better results when applied to more stable collections, as expected in the case of SmartGrid. Yet, in average, it displayed reasonable improvements in space and search speed.

As future work, following the ideas in [2], we are working in a new method to compress sequences of real numbers that allows direct access to any position.

References

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

[2] N. R. Brisaboa, S. Ladra, and G. Navarro. Directly addressable variable-length codes. In *Proc. 16th Int. Symp. on String Processing and Information Retrieval (SPIRE 09)*, pages 122–130, 2009.

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

[4] F. Claude. libcds compact data structures library. `http://libcds.recoded.cl/`, October 2011.

[5] L. Foschini, R. Grossi, A. Gupta, and J. S. Vitter. Fast compression with a static model in high-order entropy. In *Proc. Data Compression Conference (DCC 04)*, pages 62–71, 2004.

[6] R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA 05)*, pages 27–38, Greece, 2005.

[7] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. Symposium on Discrete Algorithms (SODA 03)*, pages 841–850, 2003.

[8] S. Hman. Super-scalar database compression between ram and cpu-cache. Master's thesis, Centrum Wiskunde & Informatica, Amsterdam, Netherlands, 2005.

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

[10] J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proc. IEEE*, 88(11):1722–1732, 2000.

[11] Veli Mäkinen and Gonzalo Navarro. On self-indexing images - image compression with added value. In *Proc. Data Compression Conference (DCC 08)*, pages 422–431, 2008.

[12] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. Technical Report 93-34, Dept. of Comp. Sci., University Arizona, October 1993.

[13] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Trans. on Inf. Systems*, 18(2):113–139, 2000.

[14] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 61 pages, 2007.

[15] P. Ratanaworabhan, J. Ke, and M. Burtscher. Fast lossless compression of scientific floating-point data. In *Proc. Data Compression Conference (DCC 06)*, pages 133–142, Washington, DC, USA, 2006.

[16] R. F. Rice. Some practical universal noiseless coding techniques. Technical report, Jet Propulsion Laboratory, 1979.

[17] E. S. Schwartz and B. Kallick. Generating a canonical prefix encoding. *Communications of the ACM*, 7(3):166–169, March 1964.

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