Benchmarking Coding Algorithms for the R-tree Compression*

Jiří Walder, Michal Krátký, and Radim Bača

Department of Computer Science
Technical University of Ostrava, Czech Republic
{jiri.walder,radim.baca,michal.kratky}@vsb.cz

Abstract. Multi-dimensional data structures have been widely applied in many data management fields. Spatial data indexing is their natural application, however there are many applications in different domain fields. When a compression of these data structures is considered, we follow two objectives. The first objective is a smaller index file, the second one is a reduction of the query processing time. In this paper, we apply a compression scheme to fit these objectives. This compression scheme handles compressed nodes in a secondary storage. If a page must be retrieved then this page is decompressed into the tree cache. Since this compression scheme is transparent from the tree operations point of view, we can apply various compression algorithms to pages of a tree. Obviously, there are compression algorithms suitable for various data collections, therefore, this issue is very important. In our paper, we compare the performance of Golomb, Elias-delta and Elias-gamma coding with the previously introduced Fast Fibonacci algorithm.

Keywords: multi-dimensional data structures, R-tree, compression scheme, Golomb, Elias-delta, and Elias-gamma coding, Fast Fibonacci algorithm

1 Introduction

Multidimensional data structures [21] have been widely applied in many data management fields. Spatial data indexing is their natural application, however there are many applications in different domain fields. In the case of spatial data, structures often store two- and three-dimensional objects. In the case of multimedia data, spaces with dimensionality up to 100,000 appear.

Many multidimensional data structures have been developed in the past, e.g. the quadtree family [21], LSD-tree [11], R-tree [10], R^+-tree [23], R^*-tree [4], and Hilbert R-tree [13]. In the case of R-tree, tuples are clustered in a tree's page using *MBBs* (*Minimal Bounding Boxes*). If we consider a multidimensional tuple collection, redundancy appears. Consequently, a compression may be used for the nodes efficient storage and retrieval.

* Work is partially supported by Grant of GACR No. 201/09/0990.

K. Richta, J. Pokorný, V. Snášel (Eds.): Dateso 2009, pp. 32–43, ISBN 978-80-01-04323-3.

Although some works applying a compression inside a DBMS have been developed, a real-time compression of multidimensional data structures is not often a research interest. Obviously, a smaller index file means lower *DAC* (*Disk Access Cost*) when a query is processed [19]. Consequently, lower DAC may mean the lower processing time.

There are a lot of works applying a compression inside a DBMS. In [24], authors depict RLE for compression of sorted columns to have few distinct values. In [7], authors propose the SBC-tree (String B-tree for Compressed sequences) for indexing and searching RLE-compressed sequences of arbitrary length. Work [1] demonstrates that the fast dictionary-based methods can be applied to order-preserving compression. In [5], authors introduce the IQ-tree, an index compression technique for high-dimensional data spaces. They present a page scheduling strategy for nearest neighbor algorithms that, based in a cost model, can avoid many random seeks. Work [6] introduces a tree-based structure called PCR-tree to manage principle components. In [26], authors introduce the xS-tree that uses lossy compression of bounding regions. Original works written about compressions of multidimensional data structures describe the compression of quad-tree [8, 22]. Work [8] suggested an algorithm to save at least 66% of the computer storage required by regular quadtrees. The first work [9], which concerns compressing R-tree pages, uses the relative representation of MBB to increase the fanout of the R-tree page. A bulk-loading algorithm, which is a variation of *STR* [16], and a lossy compression based on the coordinate quantization are presented there. Other works in this field are focused on improving the effectiveness of the main memory indexes. Those cache-conscious indexes suppose that they can store most of the index in the main memory. Such a work is *CR-tree* [14], which uses a type of MBB representation similar to [9]. Let the irrelevant page be the page whose MBB does not intersect a query box. These works apply the lossy compression, therefore an improved compression ratio is achieved when a filtration of irrelevant pages must be processed during a query processing.

In this paper, we utilize a compression scheme for R-tree introduced in [2]. Pages of a tree are stored in a secondary storage and decompressed in a tree's cache. We achieved a lower DAC and the pages are not always decompressed when an operation is required. In this paper, we compare the Fast Fibonacci coding [3, 2] with three other coding algorithms: Golomb, Elias-Gamma, and Elias-Delta codings [20, 17].

In Section 2, we briefly describe the R-tree and its variants. In Section 3, the above depicted compression scheme is presented. In Section 4, we describe various coding techniques: Fast Fibonacci, Golomb, Elias-gamma, and Elias-delta. Experimental results are shown in Section 5. Finally, we conclude with a summary of results and discussion about future work.

2 R-tree

R-trees [10] support point and range queries, and some forms of spatial joins. Another interesting query, supported to some extent by R-trees, is the k nearest neighbors (k-*NN* query. R-tree can be thought of as an extension of B-trees in a multi-dimensional space. It corresponds to a hierarchy of nested n-dimensional MBBs (see [10] for detail). R-tree performance is usually measured with respect to the retrieval cost (in terms of DAC) of queries.

Variants of R-trees differ in the way they perform the split algorithm. The well-known R-tree variants include R*-trees and R$^+$-trees. In [18], we can find a more detailed description as well as depiction of other R-tree variants.

It is not usually efficient to insert a large amount of data into an R-tree using the standard insert operation [10, 4]. The split algorithm is rather an expensive operation, therefore, the insertion of many items may take quite a long time. Moreover, this algorithm is executed many times during the insertion. The query performance is greatly influenced by utilization of the R-tree. A common utilization rate of an R-tree created with a standard insert algorithm is around 55%. On the other hand, the utilization rate of the R-tree created with the bulk-loading method, rises up to 95% [4].

Several bulk-loading methods [12, 15, 16] have been developed. All bulk-loading methods first order input items. Method [16] utilizes one-dimensional space-filling curve criterion for such ordering. This method is very simple and allows to order input items very fast. The result R-tree preserves suitable features for the most common data.

3 A Compression Scheme for Tree-like Data Structures

In this section, we describe a basic compression scheme which can be utilized for most paged tree data structures [2]. Pages are stored in a secondary storage and retrieved when the tree requires a page. This basic strategy is widely used by many indexing data structures such as B-trees, R-trees, and many others. They utilize cache for fast access to pages as well, since the access to the secondary storage can be more than 20 times slower compared to access to the main memory. We try to decrease the amount of DAC to a secondary storage while significantly decreasing the size of a tree file in the secondary storage.

In Figure 1, we can observe the basic idea of compression scheme used in this paper. If a tree data structure wants to retrieve a page, the compressed page is transfered from the secondary storage to the tree's cache and it is decompressed there. An important issue of our compression scheme is that the pages are only compressed in the secondary storage.

When the compression scheme is taken into consideration, the tree insert algorithm only needs to be slightly modified. Query algorithms are not affected at all because page decompression is processed only between cache and secondary storage and the tree can utilize decompressed pages for searching without knowing that they have been previously compressed.

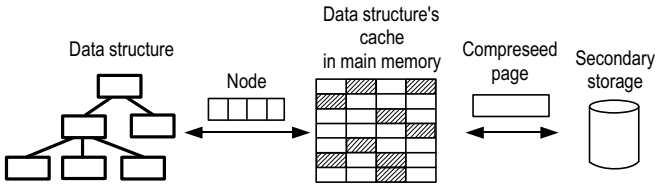

Fig. 1. Transfer of compressed pages between the secondary storage and tree's cache.

The goal of R-tree and its variants is to cluster the most similar tuples into a single page. The term 'similar tuples' means that the tuples are close to each other in a multi-dimensional space according to L_2 metric. This feature can be utilized to compress R-tree pages by a fitting compression algorithm. An important issue of this scheme is that we can apply various compression algorithms to a single R-tree. In Section 4, we show an algorithm for the R-tree compression, other compression algorithms can be found in [27].

Using this compression scheme we reduce the R-tree index size as well as DAC during a query processing. We require a decompression algorithm to be as fast as possible, otherwise the decompression time would not exceed the time saved for a lower DAC.

4 Compression Algorithm

Since tuples of a tree's page are closely located to one another in a multidimensional space, we can suppose that coordinates of these tuples are 'similar'. This means that the coordinates in each dimension are the same or their differences are rather of small values. Consequently, this feature provides increased potential for a compression.

We implemented different bit-length number coding techniques: Golomb, Elias-gamma and Elias-delta. These coding algorithms are compared with the previously published Fast Fibonacci coding [3, 2]. We utilize these coding techniques in a compression algorithm based on coding of differences between similar tuple coordinates.

4.1 Golomb, Elias-gamma and Elias-delta and Fast Fibonacci Coding

Small values are possible to code with various coding techniques. We have implemented three simple techniques for the coding of values. These techniques are as follows: Golomb, Elias-gamma, and Elias-delta [20, 17]. The algorithms used for coding are shown in Algorithms 1, 2, and 3. All codes for numbers 1-12 are depicted in Table 1.

In Algorithms 1, 2, and 3 the compressed values are read bit by bit. Retrieving the bit from the compressed memory is a time consuming operation. In [3], Fast Fibonacci decompression was introduced. This algorithm processed data without retrieving every single bit from a compressed memory. The proposed Fibonacci decompression method is based on a precomputed mapping table. This table enables converting of compressed memory segments directly into decompressed values.

Table 1. Numbers for various coding techniques

Number	Golomb			Elias		Fibonacci
	M=4	M=8	M=16	gamma	delta	
1	000	0000	00000	1	1	11
2	001	0001	00001	010	0100	011
3	010	0010	00010	011	0101	0011
4	011	0011	00011	00100	01100	1011
5	1000	0100	00100	00101	01101	00011
6	1001	0101	00101	00110	01110	10011
7	1010	0110	00110	00111	01111	01011
8	1011	0111	00111	0001000	00100000	000011
9	11000	10000	01000	0001001	00100001	100011
10	11001	10001	01001	0001010	00100010	010011
11	11010	10010	01010	0001011	00100011	001011
12	11011	10011	01011	0001100	00100100	101011

4.2 Difference-based Compressions

Difference-based compression algorithm for the R-tree was introduced in [2], however difference-based compression algorithms are well known [27, 20]. This algorithm is shown in Algorithm 4. This algorithm simply computes XOR differences between coordinates of the first tuple and values of other tuples. After that we add all difference numbers into the `mCodingBuffer` buffer, all numbers are coded by the `Encode` function. In this paper, we compare Fast Fibonacci, Golomb, Elias-Gamma, and Elias-Delta for coding of numbers. In Figure 2, we can see some encoded values for these coding techniques. Differences for the page P are output in the page P_{XOR}. The difference numbers in the page P_{XOR} are then coded by the `Encode` function.

5 Experimental Results

In our test[1], we have used the compression scheme depicted in Section 3 and coding algorithms described in Section 4. In this section, we compare the query

[1] The experiments were executed on a PC with 1.8 Ghz AMD Opteron 865, 2 MB L2 cache; 2 GB of DDR333; Windows 2008 Server.

```
input  : Golomb code bit stream and Golomb code parameter parameterM
output: Decoded number n
1  Bits ←Log(parameterM)/ Log(2);
2  TreshNumber ←Pow(2,Bits +1)−parameterM ;
3  PowerTwo ←Floor(Bits)==Bits;
4  qpart ← 0;
5  rpart ← 0;
6  bit ←stream.GetNextBit();
7  while bit do
8  |  qpart ++;
9  |  bit ←stream.GetNextBit();
10 end
11 if PowerTwo then
12 |  for x ← 0 to Bits do
13 |  |  bit ←stream.GetNextBit();
14 |  |  rpart ←rpart <<1|bit ;
15 |  end
16 else
17 |  for x ← 0 to Bits do
18 |  |  bit ←stream.GetNextBit();
19 |  |  rpart ←rpart <<1|bit ;
20 |  end
21 |  if rpart >=TreshNumber then
22 |  |  bit ←stream.GetNextBit();
23 |  |  rpart ←rpart <<1|bit ;
24 |  |  rpart ←rpart-TreshNumber ;
25 |  end
26 end
27 n ← qpart ∗parameterM +rpart ;
```

Algorithm 1: Golomb decoding algorithm

$$P = \begin{pmatrix} 4\ 0\ 6624\ 6625\ 1526 \\ 42\ 0\ 6624\ 6725\ 1535 \\ 9\ 0\ 6624\ 6626\ 6631 \\ 11\ 0\ 6624\ 6632\ 6633 \\ 29\ 0\ 6624\ 6650\ 6675 \end{pmatrix} \qquad P_{XOR} = \begin{pmatrix} 4\ 0\ 6624\ 6625\ 1526 \\ 46\ 0\ 0\ 932\ 9 \\ 13\ 0\ 0\ 3\ 7185 \\ 15\ 0\ 0\ 9\ 7199 \\ 25\ 0\ 0\ 27\ 8165 \end{pmatrix}$$

Fig. 2. The example of the page (P) and computed page difference (P_{XOR})

performance of a compressed as well as uncompressed data structures. We test both real and synthetic data sets. In all experiments, we turn off the OS's disk read cache to prevent the OS from file caching and the cache of data structures was 1,000 inner and leaf nodes. The page size of all data structures is 2,048B. To compare the performance of the compressed and uncompressed R-tree we observe the following features:

```
   input  : Elias-gamma code bit stream
   output: Decoded number n
 1 Bits ←1;
 2 n ←0;
 3 bit ←stream.GetNextBit();
 4 while not bit do
 5 |   Bits ++;
 6 |   bit ←stream.GetNextBit();
 7 end
 8 repeat
 9 |   Bits −−;
10 |   n ←n |bit <<Bits ;
11 |   if Bits >0 then
12 |   |   bit ←stream.GetNextBit();
13 |   end
14 until Bits ==0;
```

Algorithm 2: Elias-gamma decoding algorithm

- the query processing time and DAC, see Section 5.1
- R-tree index size, see Section 5.2
- an influence of various space dimensionalities, see Section 5.3
- an influence of various query selectivities, see Section 5.4

We perform experiments on synthetic as well as real data sets. In the case of synthetic data sets, we generate collections of 500,000 points for dimensionalities: 2, 4, and 6 in an integer domain of the $\langle 0, 2 \times 10^6 \rangle$ range with the uniform distribution of values. In the case of real data sets, we test TIGER 2D spatial data collections of 500,000 (TIG05) and 2 million (TIG20) points [25]. These data collections only include unique tuples. In this way, the compression scheme performance is not influenced by identical tuples. We process series of query experiments where one experiment consists of 50 randomly generated queries. Consequently, each presented result is the summary result of all these queries. Query boxes covering 0.1%, 0.2%, 0.3%, 0.4%, and 0.5% of the data space were randomly generated. In other words, the query selectivity is changed in this way.

5.1 Processing Query Time and DAC

In Tables 2 and 3, query processing performance is presented for both real and random data collections. In this experiment, selectivity is 0.2%. In the case of random data, the best query time was achieved by the Fast Fibonacci algorithm. In the case of other coding algorithms, the query time is little worse than in the case of the uncompressed R-tree. The Elias-delta decoding algorithm is 30% slower than Fast Fibonacci. The Golomb and Elias-gamma algorithms are 60% slower than Fast Fibonacci. In the case of real collections, Elias-delta coding outperforms the Fast Fibonacci coding. In the case of TIG05, Elias-delta saves

```
   input  : Elias-delta code bit stream
   output: Decoded number n
 1 Bits ← 1;
 2 n ← 0;
 3 x ← 0;
 4 bit ←stream.GetNextBit();
 5 while not bit do
 6 |   Bits ++;
 7 |   bit ←stream.GetNextBit();
 8 end
 9 Bits −−;
10 x ←x |1 <<Bits ;
11 while Bits > 0 do
12 |   Bits −−;
13 |   bit ←stream.GetNextBit();
14 |   x ←x |bit <<Bits ;
15 end
16 x −−;
17 n ←n |1 <<x ;
18 while x > 0 do
19 |   x −−;
20 |   bit ←stream.GetNextBit();
21 |   n ←n |bit <<x ;
22 end
```

Algorithm 3: Elias-delta decoding algorithm

6% of the query processing time in a comparison to the Fast Fibonacci algorithm and 19% of the query processing time in a comparison to the uncompressed R-tree.

In Table 4, we propose the query processing time in more detail for both Elias-delta and Fast Fibonacci. These results are related to the TIG20 collection. Obviously, time spent on reading of pages in the secondary storage is lower in the case of Elias-delta, however the decompression time is lower for the Fast Fibonacci algorithm. Overall query processing time is better for Fast Fibonacci. Elias-delta reads values in the bit-by-bit way, on the other hand Fast Fibonacci works with bytes. In the future, we can focus on a development of similar byte-based reading for other coding algorithms, especially for the Elias-delta algorithm. Elias-delta achieves the lowest DAC for both real and random data collections.

5.2 Index Sizes

An important issue of the compression is a reduction of the R-tree index size. In Figure 3(f), we can see the index sizes for the real collection. The best compression ratio was achieved by the Elias-delta encoding. In this case, we save more than 60% of the index size.

```
Input  : stream, an instance of cStream
output: Compressed R-tree node
1  mCodingBuffer.Clear ();
2  stream.Write (mCount);
3  for i ← 0 to mDimension do
4  |   value ← mTuples [i].GetInt (0);
5  |   stream.Write (value);
6  |   for j ← 1 to mCount do
7  |   |   int tmpValue ← mTuples [j].GetInt (i);
8  |   |   int diff ← value XOR tmpValue ;
9  |   |   mCodingBuffer.Add (diff);
10 |   end
11 |   Encode (mCodingBuffer);
12 |   stream.Write (mCodingBuffer);
13 end
```

Algorithm 4: Difference-based compression of an R-tree leaf page

	Normal	**Golomb M=4**	**Golomb M=8**	**Golomb M=16**	**Elias gamma**	**Elias delta**	**Fast Fibonacci**
Processing Time [s]	60.9	87.3	87.2	89.9	86.6	68.9	**52.1**
DAC All Nodes [MB]	21,131	8,477	8,634	8,797	7,806	7,176	**7,175**
DAC Leaf Nodes [MB]	10,691	4,333	4,413	4,491	3,988	3,674	**3,673**

Table 2. Results for the random data collection, 500K tuples, dimension: 6

	Normal	**Golomb M=4**	**Golomb M=8**	**Golomb M=16**	**Elias gamma**	**Elias delta**	**Fast Fibonacci**
Processing Time [s]	3.9	3.96	3.77	4.1	3.95	**3.17**	3.41
DAC All Nodes [MB]	4,497	2,148	2,073	2,039	2,391	**1,831**	1,899
DAC Leaf Nodes [MB]	2,185	1,016	979	959	1,137	**854**	893

Table 3. Results for the bulk-loaded real data collection, 500K tuples, dimension: 2

Time [s]	**Regular R-tree**	**Elias-delta**	**Fast Fibonacci**
Read	13.47	**5.86**	6.17
Decompression	-	10	**5.87**
Overall	22.99	21.55	**18.94**
DAC [MB]	103,251	**38,338**	40,805

Table 4. Analysis of the query processing time

5.3 Influence of the Space Dimension

We compare DAC for randomly generated data collections with dimensionalities 2, 4, and 6 (see Figure 3e). We save more than 60% of DAC in the case of Elias-delta and dimension 2. The compression ratio weakens with increasing space dimension. The space is bigger with increasing dimension, tuples are further from one another, therefore, less redundancy appears in tuples. The dimension modification has no impact on the performance of various coding techniques.

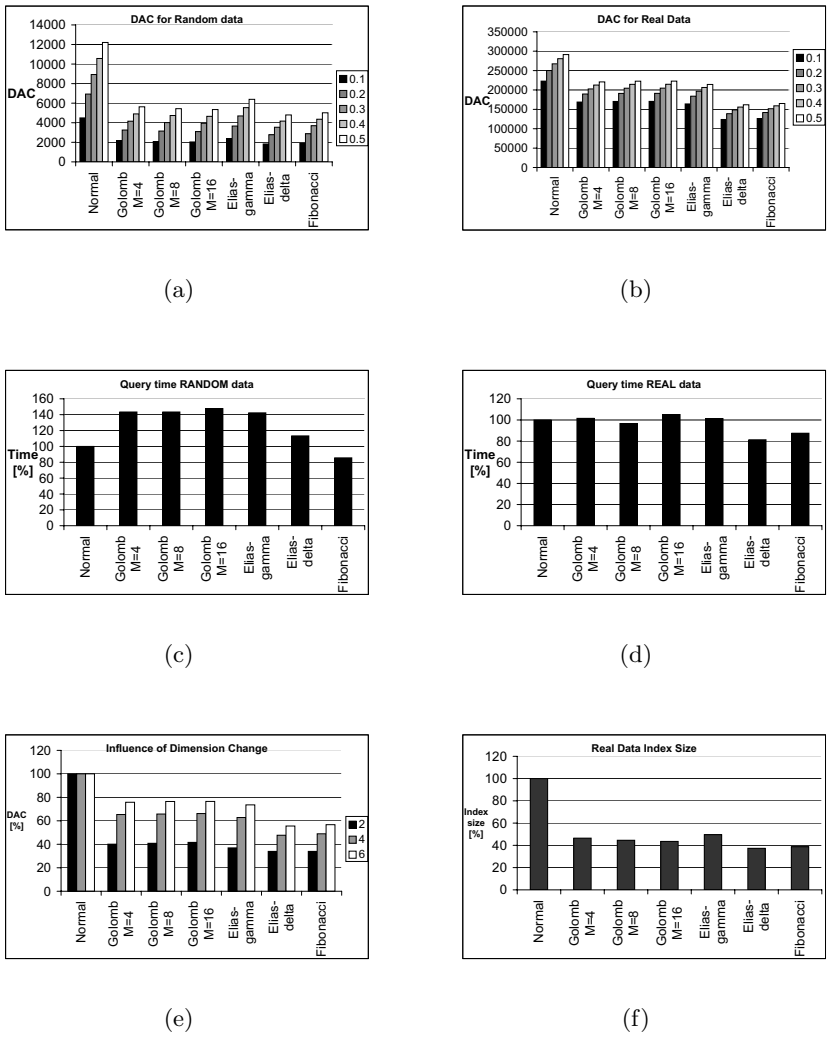

Fig. 3. DAC for various selectivities and bulk-loaded (a) random and (b) real data collections
Query processing time for bulk-loaded (c) random and (d) real data collections
(e) Selectivity influence comparison (f) Index size comparison

5.4 Influence of the Query Selectivity

In this experiment, we choose the following selectivities: 0.1%, 0.2%, 0.3%, 0.4%, and 0.5%. DAC and query processing times are put forward in Figures 3(a)-(d). The results are presented for both random and real data collections. Obviously, Elias-delta outperform other coding algorithms. The other codings produce ap-

proximately the same DAC. The selectivity modification has no impact on the performance of various coding techniques.

6 Conclusion

In this paper, we test a lossless compression scheme for the R*-tree data structure. We compare the following coding techniques: Golomb, Elias-Gamma, and Elias-Delta, with the previously published Fast Fibonacci coding. All coding algorithms improve DAC compare to the regular R-tree. In the case of real data collections, Elias-delta and Fast Fibonacci techniques achieve the best results. The Elias-delta algorithm saves 5% DAC of Fast Fibonacci. All other algorithms are less efficient that the Fast Fibonacci algorithm. When real data sets are concerned, the compression methods save at least 60% of the index size required by a regular R-tree.

The best compression ratio was achieved by the Elias-delta codding. On the other hand, decompression time for Fast Fibonacci is lower than in the case of Elias-delta. Elias-delta reads values in the bit-by-bit way, however Fast Fibonacci works with bytes. In our future work, we want to focus on a development of similar byte-based reading for other coding algorithms, especially for the Elias-delta algorithm.

References

1. G. Antoshenkov. Dictionary-based Order-preserving String Compression. *VLDB Journal – The International Journal on Very Large Data Bases*, 6(1):26–39, 1997.
2. R. Bača, M. Krátký, and V. Snášel. A compression scheme for multi-dimensional data structures. *Submitted to Information Systems*, 2009.
3. R. Bača, V. Snášel, J. Platoš, M. Krátký, and E. El-Qawasmeh. The fast fibonacci decompression algorithm. In *arXiv:0712.0811v2*, `http://arxiv.org/abs/0712.0811`, 2007.
4. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD 1990)*, pages 322–331. ACM Press, 1990.
5. S. Berchtold, C. Böhm, H.-P. Kriegel, J. Sander, and H. Jagadish. Independent Quantization: An Index Compression Technique for High-Dimensional Data Spaces. In *Proceedings of the 16th International Conference on Data Engineering (ICDE 2000)*, pages 577 – 588. IEEE Computer Society, 2000.
6. J. Cui, S. Zhou, and S. Zhao. PCR-Tree: A Compression-Based Index Structure for Similarity Searching in High-Dimensional Image Databases. In *Proceedings of the Fourth International conference on Fuzzy Systems and Knowledge Discovery (FSKD 2007)*, pages 395–400. IEEE Computer Society, 2007.
7. M. Eltabakh, W.-K. Hon, R. Shah, W. G. Aref, and J. Vitter. The SBC-Tree: An Index for Run-Length Compressed Sequences. In *Proceedings of the 11th International Conference on Extending Database Technology (EDBT 2008)*. ACM Press, 2008.

8. I. Gargantini. An Effective Way to Represent Quadtrees. *Communications of the ACM*, 25:905–910, 1982.
9. J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing Relations and Indexes. In *Proceedings of IEEE Conference on Data Engineering (ICDE 1998)*, pages 370–379, Los Alamitos, USA, 1998. IEEE Computer Society.
10. A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of ACM International Conference on Management of Data (SIGMOD 1984)*, pages 47–57. ACM Press, June 1984.
11. A. Henrich, H. W. Six, and P. Widmayer. The lsd tree: spatial access to multidimensional and non-point objects. In *VLDB '89: Proceedings of the 15th international conference on Very large data bases*, pages 45–53, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.
12. I. Kamel and C. Faloutsos. On packing R-trees. In *Proceedings of the Second International Conference on Information and Knowledge Management (CIKM 1993)*, pages 490–499. ACM Press, 1993.
13. I. Kamel and C. Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *In Proceedings of VLDB 1984*, pages 500–509, 1994.
14. K. Kim, S. K. Cha, and K. Kwon. Optimizing Multidimensional Index Trees for Main Memory Access. In *Proceedings of ACM International Conference on Management of Data (SIGMOD 2001)*, pages 139–150, New York, USA, 2001. ACM Press.
15. L.Arge, K.H.Hinrichs, J.Vahrenhold, and J.S.Vitter. Efficient Bulk Operations on Dynamic R-Trees. *Algorithmica*, pages 104–128, 2004.
16. S. Leutenegger, M. Lopez, and J. Edgington. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *Proceedings of 13th International Conference on Data Engineering (ICDE 1997)*, pages 497–506. IEEE CS Press, 1997.
17. D. J. C. Mackay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, June 2002.
18. Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis. *R-Trees: Theory and Applications*. Springer, 2005.
19. Y. Manolopoulos, Y. Theodoridis, and V. J. Tsotras. *Advanced Database Indexing*. Kluwer Academic Publisher, 2001.
20. D. Salomon. *Data Compression The Complete Reference*. Third Edition, Springer–Verlag, New York, 2004.
21. H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.
22. H. Samet. Data Structures for Quadtree Approximation and Compression. *Communications of the ACM archive*, 28(9):973–993, September 1985.
23. T. Sellis, N. Roussopoulos, and C. Faloutsos. The r^+-tree: A dynamic index for multidimensional objects. In *In Proceedings of VLDB 1987*, pages 507–518, 1987.
24. M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, and S. Madden. C-store: A Column Oriented DBMS. In *Proceedings of the International Conference on Very Large Data Bases, VLDB 2005*.
25. U.S. Department of Commerce, U.S. Census Bureau, Geography Division. TIGER/Line Files, 2006 Second Edition, Alabama, Autauga County, 2006, `http://www.census.gov/geo/www/tiger/`.
26. C. Wang and X. S. Wang. Indexing Very High-dimensional Sparse and Quasi-sparse Vectors for Similarity Searches. *VLDB Journal – The International Journal on Very Large Data Bases*, 9(4):344–361, 2001.
27. I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes, Compressing and Indexing Documents and Images, 2nd edition*. Morgan Kaufmann, 1999.