# Benchmarking a B-tree compression method*

Filip Křižka, Michal Krátký, and Radim Bača

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

**Abstract.** *The B-tree and its variants have been widely applied in many data management 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. The utilized 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 operation's point of view, we can apply various compression algorithms to pages of a tree. Obviously, there are compression algorithms suitable for various data collections, and so, this issue is very important. In our paper, we compare the B-tree and compressed B-tree where the Fast Fibonacci and invariable coding compression methods are applied.*

**Key words:** *B-tree and its variants, B-tree compression, compression scheme, fast decompression algorithm*

## 1 Introduction

The B-tree represents an efficient structure for the finding of an ordered set [6]. The B-tree has been often used as the backbone data structure for the physical implementation of RDBMS or file systems. Its most important characteristic is that keys in a node have very small differences to each others. We utilize this feature in the B-tree compression. In this case, nodes are compressed in the secondary storage and they are decompressed during their reading into the cache. Due to the fact that the random access in the secondary storage is a rather expensive operation, we save time when reading the nodes.

In work [11], authors summarize some methods for organizing of B-trees. A prefix B-tree, introduced in [7], provides the head and tail compression. In the case of the head compression, one chooses a common prefix for all keys that the page can store, not just the current keys. Tail compression selects a short index term for the nodes above the data pages. This index needs merely to separate the keys of one data node from those of its sibling and is chosen during a node split. Tail compression produces variable length index entries, and [7] describes a binary search that copes with variable length entries.

Work [9] describes a split technique for data. Rows are assigned tag values in the order in which they are added to the table. Note that tag values identify rows in the table, not records in an individual partition or in an individual index. Each tag value appears only once in each index. All vertical partitions are stored in the B-tree with the tag value as the key. The novel aspect is that the storage of the leading key is reduced to a minimal value.

Unlike these works, in our work we suppose the B-tree compression without changes of the B-tree structure. We mainly utilize the fast decompression algorithm. In the case of the previously depicted papers, B-tree compression is possible using a modification of the B-tree structure. In work [7], B-tree is presented by B*-index and B*-file. The keys stored in the B*-index are only used to searching and determining in which subtree of a given branch node a key and its associated record will be found. The B*-index itself is a conventional B-tree including prefixes of the keys in the B*-file. This prefix B-tree combines some of the advantages of B-trees, digital search trees, and key compression without sacrificing the basic simplicity of B-trees and the associated algorithms and without inheriting some of the disadvantages of digital search trees and key compression techniques. Work [9] describes an efficient columnar storage in B-trees. Column-oriented storage formats have been proposed for query processing in relational data warehouses, specifically for fast scans over non-indexed columns. This data compression method reuses traditional on-disk B-tree structures with only minor changes yet achieves storage density and scan performance comparable to specialized columnar designs. In work [1], B-tree compression is used for minimizing the amount of space used by certain types of B-tree indexes. When a B-tree is compressed, duplicate occurrences of the indexed column values are eliminated. It is compressed by clustering the same keys and their unindexed attributes.

This paper is organized as follows. In Section 2, we briefly summarize basic knowledge about the B-tree. Section 3 shows a compression scheme used [3]. Section 4 describes two compression methods. Section 5 shows results of the compression methods. The compressed B-tree is compared with a proper B-tree. In

* Work is partially supported by Grants of GACR No. 201/09/0990 and IGA, FEECS, Technical University of Ostrava, No. BI 4569951, Czech Republic.

Section 6, we summarize the paper content and outline possibilities of our future work.

## 2 B-tree and its variants

The B-tree is a tree structure published by Rudolf Bayer and Edward M. McCreight in 1972 [6]. The B-tree keeps data sorted and allows searches, insertions, and deletions in logarithmic amortized time. It is optimized for systems that read and write large blocks of data. A B-tree is kept balanced by requiring that all leaf nodes are at the same depth. This depth will increase slowly as elements are added to the tree, but an increase in the overall depth is infrequent, and results in all leaf nodes being one more node further away from the root.

B-trees have substantial advantages over alternative implementations when node access times far exceed access times within nodes. This usually occurs when most nodes are in secondary storage such as hard drives. By maximizing the number of child nodes within each internal node, the height of the tree decreases, balancing occurs less often, and efficiency increases. Usually this value is set such that each node takes up a full disk block or an analogous size in secondary storage.

A B-tree of order $m$ (the maximum number of children for each node) is a tree which satisfies the following properties:

- Every node has at most $m$ children.
- Every node (except root and leaves) has at least $\frac{m}{2}$ children.
- The root has at least two children if it is not a leaf node.
- All leave nodes are in the same level.
- All inner nodes with $k$ children contain $k$–1 links to children.

Each internal node's elements act as separation values which divide its subtrees. For example, if an internal node has three child nodes (or subtrees) then it must have two separation values or elements $a_1$ and $a_2$. All values in the leftmost subtree will be less than $a_1$, all values in the middle subtree will be between $a_1$ and $a_2$, and all values in the rightmost subtree will be greater than $a_2$.

Internal nodes in a B-tree – nodes which are not leaf nodes – are usually represented as an ordered set of elements and child pointers. Every internal node contains a maximum of $U$ children and – other than the root – a minimum of $L$ children. For all internal nodes other than the root, the number of elements is one less than the number of child pointers; the number of elements is between $L-1$ and $U-1$. The number $U$ must be either $2 \cdot L$ or $2 \cdot L-1$; thus each internal node is at least half full. This relationship between $U$ and $L$ implies that two half-full nodes can be joined to make a legal node, and one full node can be split into two legal nodes (if there is an empty space to push one element up into the parent). These properties make it possible to delete and insert new values into a B-tree and adjust the tree to preserve the B-tree properties.

Leaf nodes have the same restriction on the number of elements, but have no children, and no child pointers. The root node still has the upper limit on the number of children, but has no lower limit. For example, when there are fewer than $L$-1 elements in the entire tree, the root will be the only node in the tree, and it will have no children at all.

A B-tree of depth $n$+1 can hold about $U$ times as many items as a B-tree of depth $n$, but the cost of search, insert, and delete operations grows with the depth of the tree. As with any balanced tree, the cost increases much more slowly than the number of elements.

Some balanced trees store values only at the leaf nodes, and so have different kinds of nodes for leaf nodes and internal nodes. B-trees keep values in every node in the tree, and may use the same structure for all nodes. However, since leaf nodes never have children, a specialized structure for leaf nodes in B-trees will improve performance. The best case height of a B-tree is: $\log_M n$. The worst case height of a B-tree is: $\log_{\frac{M}{2}} n$. Where $M$ is the maximum number of children a node can have.

There exists many variants of the B-tree: B$^*$-tree [13], B$^{**}$-tree [15], B$^+$-tree [17]. In the case of the B$^+$-tree, data is only stored in leaf nodes and inner nodes include keys. Leaf nodes hold links to the previous and next nodes. Moreover, many paged data structures like UB-tree [5, 12], BUB-tree [8], and R-tree [10] are based on the B-tree.

## 3 A compression scheme for tree-like data structures

In this section, we describe a basic scheme which can be utilized for most paged tree data structures [3]. 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 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 disc access cost (DAC) to a secondary storage while significantly decreasing the size of a tree file in the secondary storage.

Let us consider a common cache schema of persistent data structures in Figure 1(a). When a tree requires a node, the node is read from the main memory cache. If the node is not in the cache, the node page is retrieved from the secondary storage.

An important issue of the compression schema is that tree pages are only compressed in the secondary storage. In Figure 1(b), we can observe the basic idea of the scheme. 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. Function `TreeNode:: Decompress()` performs the decompression. Afterwards, the decompressed page is stored in the cache. Therefore, the tree's algorithms only work with decompressed pages. Obviously, the tree is preserved as a dynamic data structure and in our experiments we show the page decompression does not significantly affect query performance because we save time with the lower DAC.

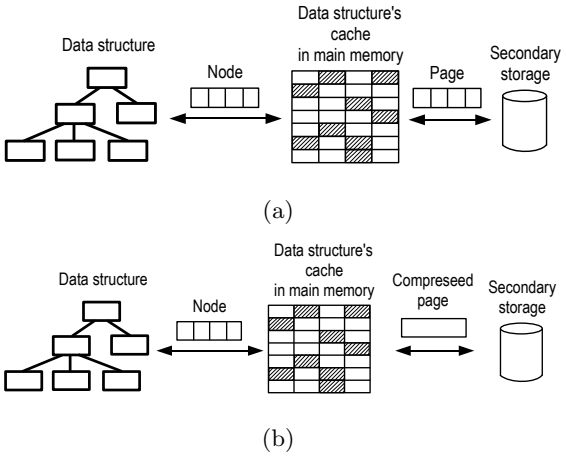


**Fig. 1.** (a) Transfer of tree's pages between the secondary storage and tree's cache. (b) Transfer of compressed pages between the secondary storage and tree's cache.

### 3.1 How the compression scheme affects tree algorithms

When the compression scheme is taken into consideration, the tree insert algorithm only needs to be slightly modified. When we insert or modify a record in a page, we have to perform the function `TreeNode::Compress Test()` which tests whether the node fits into the page. If not, the node needs to be split. Also, during the split, we have to make sure that the final nodes fit into the page. This means that the maximum capacity of a page can vary depending on the redundancy of the data. The maximum capacity of each tree's page must be determined by a heuristic:

$$C_c = \frac{C_u}{CR_A},$$

where $CR_A$ is the assumed maximum compression ratio, $C_u$ is the capacity of the uncompressed page. For example, the size of the page is 2,048 B, $C_u = 100$, $CR_A = 1/5$, then $C_c = 500$. The size of the page for the capacity is 10,240 B. This means that all pages in the tree's cache have $C_c = 500$, although their $S$ size in the secondary storage is less than or equal to 2,048 B. Let us note that

$$CR = \frac{compressed\ size}{original\ size}.$$

The `TreeNode::Compress()` function is called when the page must be stored in the secondary storage.

Every page in the tree has its minimal page utilization $C_c/2$. Let $S_l$ denote the byte size of a compressed page. After deleting one item in the page, the byte size of the page is denoted by $S_c$. Without loss of generality, we can assume that $S_c \leq S_l$. If items are deleted from the page, we must check whether capacity is less than or equal to $C_c/2$. If so, the page is stretched into other pages according to the tree deleting algorithm.

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.

This basic idea of the compression scheme can be applied to any paged tree data structure. It is not dependent upon an applied split algorithm, nor on the key type stored in the tree's pages. We test this scheme on $B^+$-tree data structure because this data structure has remained very popular in recent years and it is suitable for further page compressing.

## 4 B-tree compression methods

In this article, we have applied two compression methods: Fast Fibonacci (FF) and Invariable Coding (IC). Since keys in a node are close to each another, we use the well-known difference compression method [14]. Similar algorithms were used in the case of the R-tree compression [3, 16].

### 4.1 Fast Fibonacci compression

In this method, we apply the Fibonacci coding [2] which uses the Fibonacci numbers; 1, 2, 3, 5, 8, 13, .... A value is coded as the sum of the Fibonacci numbers that are represented by the 1-bit in a binary buffer. Special 1-bit is added as the lowest bit in this binary buffer after the coding is finished. For example, the

**Algorithm 1**: Fast Fibonacci Compression Algorithm

```
   function : CompressionLeafNode(node)
 1 Write item1
 2 for i in 2 .. node.count do
 3     num ←
       FibonacciCodder(node.key(i)-node.key(i-1))
 4     Write num
 5     num ← FibonacciCodder(node.nonatrr(i))
 6     Write num
 7 end
 8 Write links
   function : CompressionNode(node)
 9 Write item1
10 for i in 2 .. node.count do
11     num ←
       FibonacciCodder(node.key(i)-node.key(i-1))
12     Write num
13 end
14 Write links
   function : Compression(node)
15 if node.isLeaf is leaf then
16     CompressionNode(node)
17 end
18 else
19     CompressionLeafNode(node)
20 end
   function : CompressionTest(node)
21 tmp ← Compression(node)
22 if tmp.size > page.size then
23     return false
24 end
25 else
26     return true
27 end
```

value 20 is coded as follows: 0101011 (13 + 5 + 2). Due to the fact that we need the compression algorithm as fast as possible, we use the Fast Fibonacci decompression introduced in [4].

Algorithm of the Fast Fibonacci compression is shown in Algorithm 1. We explain the algorithm in the following paragraphs.

**Compression of inner nodes:**

- Keys are compressed by the Fibonacci coding. The first value, obviously minimal, is stored. We compute the difference of each neighboring value and the difference is coded by Fibonacci coding. (see Lines 10-13)
- Child and parent links are not compressed. (Line 14)

**Compression of leaf nodes:**

- Keys are compressed in the same way as the keys in an inner node. (Lines 3-4)
- Unindexed attribute values are compressed by Fibonacci coding. (Lines 5-6)
- Parent, previous, and next nodes links are not compressed. (Line 8)

### 4.2 Invariable coding compression

As in the previous method, we work with the difference of each neighboring value. Since, we use invariable coding, we must first compute the difference of the last, maximal value, and the first value. The result of this computation is the number of bits necessary for a storage of the maximal value and, obviously, each value of

**Algorithm 2**: Invariable Coding Compression Method

```
   function : CompressionLeafNode(node)
 1 Write maxBits(node.key(1),node.key(n))
 2 Write node.key(1)
 3 for i in 2 .. node.count do
 4     num ← ICCodder(node.key(i)-node.key(i-1))
 5     Write num
 6 end
 7 Write maxBits(node.nonatrr(1),node.nonatrr(n))
 8 Write node.nonatrr(min)
 9 for i in 2 .. node.count do
10     num ←
       ICCodder(node.nonatrr(i)-node.nonatrr(min))
11     Write num
12 end
13 Write links
   function : CompressionNode(node)
14 Write maxBits(node.key(1),node.key(n))
15 Write node.key(1)
16 for i in 2 .. node.count do
17     num ← ICCodder(node.key(i)-node.key(i-1))
18     Write num
19 end
20 Write links
   function : Compression(node)
21 if node.isLeaf is leaf then
22     CompressionNode(node)
23 end
24 else
25     CompressionLeafNode(node)
26 end
   function : CompressionTest(node)
27 tmp ← Compression(node)
28 if tmp.size > page.size then
29     return false
30 end
31 else
32     return true
33 end
```

| | RND_8 | | | RND_16 | | |
|---|---|---|---|---|---|---|
| | **B$^+$-tree** | **FF** | **IC** | **B$^+$-tree** | **FF** | **IC** |
| Height | 2 | 2 | 2 | 2 | 2 | 2 |
| Domain | 8b | | | 16b | | |
| DAC Read | 20,858,473 | 12,115,647 | 10,010,790 | 5,996,536 | 5,872,078 | 5,739,912 |
| Creating time [s] | 15,288 | 65,618 | 44,201 | 6,360 | 11,897 | 10,605 |
| Cache Time [s] | 7,357 | 5,398 | 1,565 | 6,020 | 1,495 | 1,181 |
| Compress time [s] | 0 | 867 | 652 | 0 | 883 | 636 |
| Decompress time [s] | 0 | 53,524 | 35,908 | 0 | 9,276 | 8,567 |
| # Inner nodes | 35 | 17 | 9 | 33 | 17 | 9 |
| # Leaf nodes | 7,746 | 3,350 | 2,431 | 5,695 | 2,596 | 2,114 |
| # Avg. node items | 222.3 | 198 | 271 | 172.58 | 153.65 | 235.8 |
| # Avg. leaf node items | 129.1 | 298.5 | 411.4 | 176.58 | 385.21 | 473 |
| Index size [kB] | 15,564 | 6,736 | 4,882 | 11,394 | 5,228 | 4,248 |
| Compression ratio | 1 | 0.56 | 0.69 | 1 | 0.54 | 0.63 |

**Table 1.** Building B$^+$-tree index, result for RND_8 and RND_16.

| | RND_24 | | | RND_32 | | |
|---|---|---|---|---|---|---|
| | **B$^+$tree** | **FF** | **IC** | **B$^+$tree** | **FF** | **IC** |
| Height | 2 | 2 | 2 | 2 | 2 | 2 |
| Max item value | 24b | | | 32b | | |
| DAC Read [all] | 5,996,536 | 5,907,459 | 5,830,822 | 5,996,536 | 5,931,627 | 5,889,079 |
| Creating time [s] | 7,377 | 12,098 | 12,435 | 7,629 | 13,686 | 13,154 |
| Cache Time [s] | 7,001 | 1,556 | 1,690 | 7,203 | 2,935 | 2,267 |
| Compress time [s] | 0 | 882 | 664 | 0 | 885 | 597 |
| Decompress time [s] | 0 | 9,419 | 9,828 | 0 | 9,595 | 10,003 |
| # Inner nodes | 33 | 17 | 17 | 33 | 26 | 17 |
| # Leaf nodes | 5,663 | 2,717 | 3,099 | 5,663 | 2,800 | 3,756 |
| # Avg node items | 172.58 | 160.76 | 183.24 | 172.58 | 108.65 | 221.88 |
| # Avg leaf node items | 176.58 | 368.05 | 322.68 | 176.58 | 357.14 | 266.24 |
| Tree size [kB] | 11,394 | 5,470 | 6,234 | 11,394 | 5,654 | 5,272 |
| Compression ratio | 1 | 0.52 | 0.45 | 1 | 0.50 | 0.54 |

**Table 2.** Building B$^+$-tree index, results for RND_24 and RND_32.

| | RND_8 | | | RND_16 | | | RND_24 | | | RND_32 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **B$^+$tree** | **FF** | **IC** | **B$^+$tree** | **FF** | **IC** | **B$^+$tree** | **FF** | **IC** | **B$^+$tree** | **FF** | **IC** |
| Query time [s] | 182.4 | 37.6 | 33.5 | 38.2 | 34.2 | 31.7 | 38.1 | 34.2 | 35.2 | 38.2 | 45.3 | 37.2 |
| Decompress time [s] | 0 | 6.8 | 5.7 | 0 | 6.7 | 5.0 | 0 | 6.6 | 6.1 | 0 | 6.8 | 6.8 |
| Cache Time [s] | 149.1 | 3.8 | 1.8 | 5.0 | 2.0 | 1.3 | 5.6 | 2.4 | 2.5 | 7.2 | 12.7 | 3.0 |
| DAC Read | 64,813 | 28,123 | 20,516 | 47,068 | 21,897 | 17,610 | 47,068 | 22,469 | 25,700 | 47,068 | 23,200 | 31,296 |

**Table 3.** B-tree querying results for RND_8, RND_16, RND_24 and RND_32.

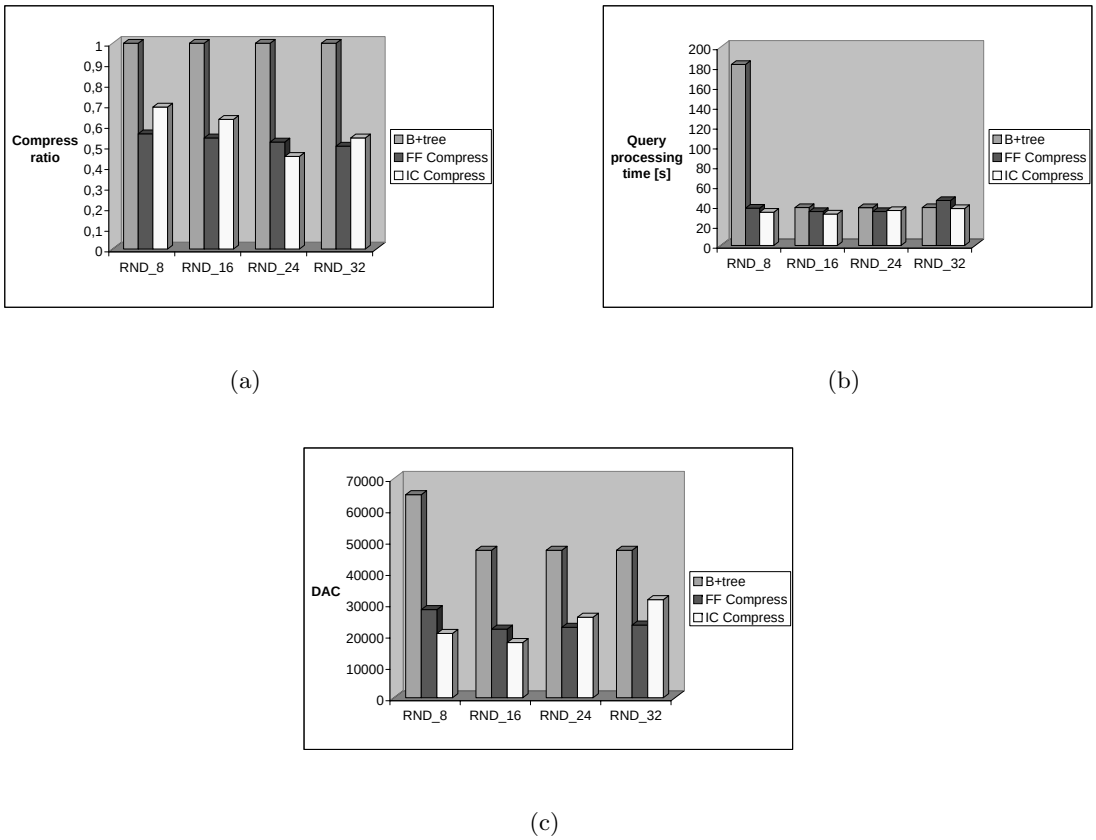


**Fig. 2.** Experiment results: (a) Compression ratio (b) Query processing time (c) DAC.

the node. For example, if the difference of the maximal and minimal value is 20, all values are stored in 5bits.

Algorithm of the IC compression is shown in Algorithm 1. We explain the algorithm in the following paragraphs.

**Compression of inner nodes:**

- Keys are compressed by the above proposed method. We store the first value and the number of bits necessary for a storage of the maximal value. After, all difference values are stored. (Lines 14-19)
- Child and parent links are not compressed. (Line 20)

**Compression of leaf nodes:**

- Keys are compressed in the same way as the keys in an inner node. (Lines 1-6)
- Unindexed attribute values are similarly compressed as keys, however the maximal value is not the last value – it must be found by a sequence scan.
- Parent, previous, and next nodes links are not compressed. (Line 13)

## 5 Experimental results

In our experiments[1], we test previously described compression methods. These approaches are implemented in C++. We use four synthetic collections which differ in values included. Collection RND_8 includes values in $\langle 0; 255 \rangle$, RND_16 includes values in $\langle 0; 65,535 \rangle$. In this way, we create collections RND_24 and RND_32 as well. Each collection contains 1,000,000 items.

For each collection, we test index building and querying by processing time and DAC. In all tests, the page size is 2,048B and cache size is 1,024 nodes. The cache of the OS was turned off.

Results of index building are depicted in Table 1 and 2. We see that the compression ratio decreases for increasing size of domains. FF compression is more efficient for lower values; on the other hand, the IC compression is more efficient for higher values. Obviously, due to features of the compressed scheme used, we obtain the high compress time. Consequently, the

[1] The experiments were executed on an AMD Opteron 865 1.8Ghz, 2.0 MB L2 cache; 2GB of DDR333; Windows 2003 Server.

time of creating of $B^+$-tree with the FF compression is $1.6 - 4.3\times$ higher then the time of creating for the $B^+$-tree. In the case of the IC compression, the creating time is $1.7 - 2.9\times$ higher. The compression ratio is shown is Figure 4.2(a) as well.

In our experiments, we test 50 random queries and the results are then averaged. The results are shown in Table 3. The number of DAC is $2.1 - 3.5\times$ lower for the FF compression when compared to the $B^+$-tree and $1.5 - 3.6\times$ for the IC compression. This result influences the query processing time. The query processing times is $0.84 - 4.85\times$ more efficient in the case of the FF compression when compared to the $B^+$-tree and the time is $1.03 - 5.4\times$ more efficient for the IC compression. Obviously, if the compression ratio is over a threshold then the $B^+$-tree overcomes the compressed indices. In Figure 4.2(b) and (c), we see the query processing time and DAC.

## 6 Conclusion

In this article, we propose two methods for B-tree compression. If the compression ratio is below a threshold then the query processing performance of the compressed index overcomes the B-tree. However, there are still some open issues. The first one is the high creating time. In this case, we must develop a more efficient method or we must use and test the bulkloading (see [3, 16]). Additionally, we must test our method for a real data collection. Finally, we should test different compression and coding methods (i.e. Elias-delta code, Elias-gamma code, Golomb code [14]).

## References


1. C. Antognini: *Troubleshooting Oracle Performance*. Apress, 2008.
2. A. Apostolico and A. Fraenkel: *Robust transmission of unbounded strings using Fibonacci representations*. IEEE Trans. Inform., 33, 2, 1987, 238–245.
3. R. Bača, M. Krátký, and V. Snášel: *A compression scheme for the R-tree data structure*. In Submitted in Information Systems, 2009.
4. 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.
5. R. Bayer: *The universal B-tree for multidimensional indexing: general concepts*. In Proceedings of World-Wide Computing and Its Applications (WWCA 1997), Tsukuba, Japan, Lecture Notes in Computer Science, Springer–Verlag, 1997, 198–209.
6. R. Bayer and E. M. McCreight: *Organization and maintenance of large ordered indices*. Acta Informatica, 1972, 173–189.
7. R. Bayer and K. Unterauer: *Prefix B-trees*. *ACM Trans. on Database Systems*, 2, 1, 1977, 11–26.
8. R. Fenk: *The BUB-tree*. In Proceedings of 28rd VLDB International Conference on Very Large Data Bases (VLDB'02), Hongkong, China, Morgan Kaufmann, 2002.
9. G. Goetz: *Efficient columnar storage in B-trees*. In Proceedings of SIGMOD Conference, 2007.
10. A. Guttman: *R-Trees: a dynamic index structure for spatial searching*. In Proceedings of ACM International Conference on Management of Data (SIGMOD 1984), ACM Press, June 1984, 47–57.
11. D. Lomet: *The evolution of effective B-tree page organization and techniques: a personal account*. In Proceedings of SIGMOD Conference, Sep. 2001.
12. V. Markl: *Mistral: Processing relational queries using a multidimensional access technique*. Ph.D. thesis, Technical University München, Germany, 1999, `http://mistral.in.tum.de/results/publications/Mar99.pdf`.
13. Y. Sagiv: *Concurrent operations on B*-trees with overtaking*. In Journal of Computer and System Sciences, 1986.
14. D. Salomon: *Data Compression The Complete Reference*. Third Edition, Springer–Verlag, New York, 2004.
15. A.A. Toptsis: *B**-tree: a data organization method for high storage utilization*. In Computing and Information, 1993.
16. J. Walder, M. Krátký, and R. Bača: *Benchmarking coding algorithms for the R-tree compression*. In Proceedings of DATESO 2009, Czech Republic, 2009.
17. N. Wirth: *Algorithms and Data Structures*. Prentice Hall, 1984.