# Efficiency Improvement of Narrow Range Query Processing in R-tree*

Peter Chovanec and Michal Krátký

Department of Computer Science
Technical University of Ostrava, Czech Republic
{peter.chovanec,michal.kratky}@vsb.cz

**Abstract.** Indexing methods for efficient processing of multidimensional data are very requested in many fields, like geographical information systems, drawing documentations etc. Well-known R-tree is one of the multidimensional data structures. The R-tree is based on bounding of spatial near points by multidimensional rectangles. This data structure supports various types of queries, e.g. point and range queries. The range query retrieves all tuples of a multidimensional space in the defined query box. Narrow range query is an important type of the range query including at least one narrow dimension. Despite many variants of R-trees, narrow range query processing is inefficient. In this paper, we depict a modification of Signature R-tree: data structure for the narrow range query processing. This data structure applies signatures for a description of tuples stored in a tree's page. We present an improvement of this technique.

**Key words:** multidimensional data structure, narrow range query, R-tree, signature

## 1 Introduction

Multimedia databases have become increasingly important in many application areas such as medicine, CAD, geography, and molecular biology. Processing of multi-dimensional data is requested in almost all fields. There are a lot of applications of *multi-dimensional data structures* [15], e.g., data mining [10], term indexing [5, 12], XML documents [8, 11], text documents and images [4]. Query processing in high-dimensional spaces has therefore been a very prominent research area over the last few years. A number of new index structures and algorithms have been proposed.

There are two major approaches to multi-dimensional indexing [16]: data structures for indexing metric spaces and data structures for indexing vector spaces. The first approach includes, for example, $n$-dimensional B-tree [7], R-tree [9], R*-tree [2], Signature R-tree [13], X-tree [3], UB-tree [1] and BUB-tree [6]. The second one includes M-tree [4], for example.

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

K. Richta, J. Pokorný, V. Snášel (Eds.): Dateso 2009, pp. 154–165, ISBN 978-80-01-04323-3.

A multi-dimensional data structure often supports the range query. This query may be written as the following pseudo SQL statement:
`SELECT * FROM T WHERE` $ql_1 \leq t_1 \leq qh_1$ `AND ... AND` $ql_n \leq t_n \leq qh_n$.

The narrow range query is special type of the range query, where at least one dimension is narrow. In Figure 1, we see examples of query boxes for the narrow range queries in spaces with the dimensions $n = 2$ and $n = 3$, respectively. Another example of this query is the following SQL statement: `SELECT * FROM <table_name> WHERE` $1 < a_0 < 10000$ `AND` $a_1 = 2$ `AND` $a_2 = 3$

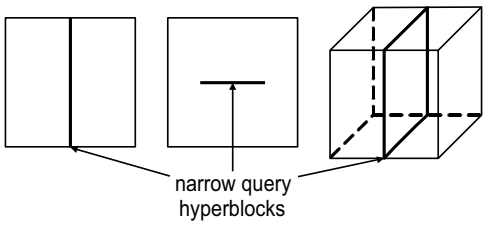


**Fig. 1.** Examples of the narrow range queries in spaces with the dimensions $n = 2$ and $n = 3$, respectively.

In paper [13], we depicted that the narrow range query processing is rather inefficient in current multidimensional data structures. Signature R-tree, handling point data, is introduced in this work. In our paper, we describe an improvement of the Signature R-tree called ESR-tree. In Section 2, we review existing approaches for the narrow range query processing. Section 3 presents our improvement of the Signature R-tree. Section 4 reviews possibilities of signature building for this data structure. In Section 4.3, we outline some implementation details of the improved data structure. In Section 5, we put forward experimental results. Finally, we conclude with a summary of contributions and discussion about future work.

## 2 Existing Approaches

In this section, we describe two data structures which have been often applied for the narrow range query processing. These data structures, B-tree and Signature R-tree, are compared with the novel ESR-tree in Section 5.

### 2.1 B-trees

The B-tree is an $m$-ary balanced tree introduced by Bayer in 1972. This structure guarantees the logarithmic complexity for the item searching. Due to the ordering, B-tree enables searching only by one attribute. In many cases, it is

necessary to search by more than one attribute. Therefore, there are some improvements of B-tree for searching of multidimensional data, e.g. B-tree with compound keys. In this case, we create one compound index with keys related to each narrow dimension. Obviously, this technique is not as general as multidimensional data structures. For example, we create the compound index for dimensions 1, 3, and 10 for a 10-dimensional tuple collection. It means, narrow range query with these narrow dimensions is as efficient as possible. Moreover, no intermediate results are created. If want to process a query with the narrow dimensions: 1, 3, 10, and 5, we retrieve an intermediate result again. If we want to solve a general query without the intermediate result, we should create $n!$ compound indices. Therefore, we do not suppose this technique in our article.

If the range query is concerned in B-tree, we must index each attribute (or dimension) in a separate B-tree. The range query is processed by a sequence of searching in B-trees, and individual intermediate results are joined. Obviously, there are two issues. If the size of an intermediate result >> the overall result, this processing is rather inefficient. The second issue is the size of the index file. In Section 5, we use this implementation for a comparison with our method. We propose that the index file built in this way is 3× larger in average than the index file of a multidimensional data storage.

### 2.2 Signature R-trees

Since 1984 when Guttman proposed his method [9], R-trees have become the most cited and most used as reference data structure in this field. The R-trees can be thought of as an extension of B-trees in a multi-dimensional space. It corresponds to a hierarchy of nested $n$-dimensional *minimum bounding boxes* (MBB). There are many approaches based on an improvement of original R-trees. In [13], Signature R-tree was introduced for more efficient processing of narrow range queries.

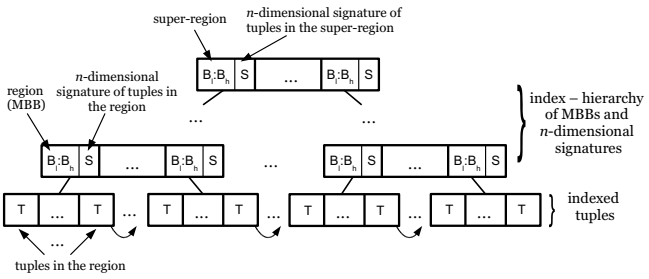


**Fig. 2.** Structure of the Signature R-Tree

Signature R-tree is a variant of the R-tree including multidimensional signatures for a more efficient filtration of irrelevant tree nodes. In this case, irrelevant node does not contain any tuple of the query box. The multidimensional signature contains a signature of tuples in the node for each dimension (see Section 4.1). A general structure of the Signature R-tree is presented in Figure 2. Leaf nodes include tuples clustered into MBBs (MBB is defined by two multidimensional points). These MBBs are clustered into super-MBBs as well. This hierarchy is finished by MBBs in the root node. Consequently, MBBs are stored in inner nodes. In the case of Signature R-tree, the multidimensional signature is assigned to each MBB. It means that each inner node item includes the MBB definition together with multidimensional signature, where this signature is superimposed with signatures of the node's children. Consequently, such a tree contains two hierarchies, the hierarchy of MBBs and hierarchy of signatures.

In [13], we show that many irrelevant nodes are skipped during the narrow range query processing if we compare Signature R-tree and R-tree. However, we distinguish some negative consequences of this data structure. If signatures are long, the inner node capacity is low. This issue results in a degeneration of the tree: inner node can contain only a trivial number of items and, therefore, the height of the tree is very high. Similar issue appears if we use more hash functions for each signature. Therefore, in this paper, we introduce an improvement of Signature R-tree supporting these refinements.

## 3 ESR-Tree – An Improvement of Signature R-tree

### 3.1 Introduction

Signature R-tree significantly decreases the number of processed irrelevant nodes, however the number $> 0$ (see Section 6). In [13], we introduced the following quality measurement of the range query processing (so called *relevance*): $c_Q = N_r/N_p$, where $N_r$ is the number of relevant nodes, $N_p$ is the number of all processed node. Obviously, if only relevant nodes are processed during a range query then $c_Q = 1$.

If we want to reach the most efficient $c_Q$ value, we must use longer signatures, a higher number of hash functions (it means more signatures is related to one dimension), signatures with different lengths for different levels of a tree, and various hashing functions building the signatures. However, in the case of Signature R-tree, each this idea extends the size of inner node item and decreases the inner node capacity. Therefore, our improvement of Signature R-tree, called ESR-tree, removes signatures from inner nodes: the signatures are stored in a persistent array. A structure of the novel ESR-tree is presented in Figure 3.

The isolation of signatures from R-tree enables to enlarge signatures although the node capacity is not decreased. Moreover, we can use signatures with different lengths for different levels of the tree. In our paper, we use the inverted level number for the signature labeling. Consequently, we use $S^0$ as the label of an MBB's signature, $S^1$ as the label of a super-MBB signature, and so on.

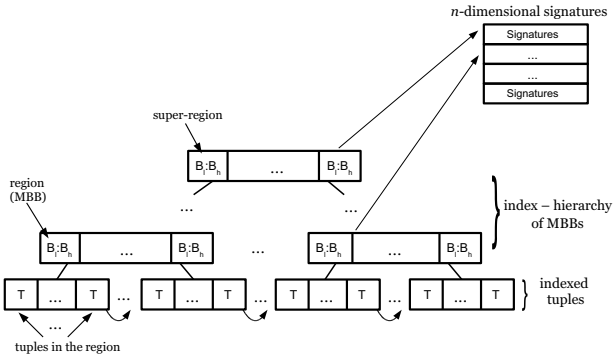


**Fig. 3.** A structure of the ESR-tree

### 3.2 ESR-tree Operations

Operations `Insert`, `Delete` and `Find` (or point query) are handled by algorithms of the selected R-tree variant. In the case of the Insert operation (see Algorithm 1), we must update the signature of the MBB which was changed. In this way, we must update signatures for each level of the tree from the current node to the root node. In this algorithm, we use the following variables: *tuple* is an inserted tuple, $Z$ represents a stack containing the current path of the tree, and $N$ is the current node.

Signature index may be created after all tuples are inserted into the tree. This bulkload algorithm must preordered process all tree nodes and create the signature for each MBB. Obviously, signatures may be created for an arbitrary level of the tree.

A common issue of ESR-tree is that the query processing efficiency of a common range query is not influenced by signatures. Signature R-tree includes signatures in tree's nodes, therefore, signatures are read from the secondary storage although these signatures are not used for the node filtering. On the other hand, in the case of the narrow range query, we apply signatures for the more efficient filtration of irrelevant tree nodes.

Let us suppose the range query algorithm. `Intersection` operation computes whether an MBB is intersected by the query box in the linear time, on the other hand, `AND` operation is used as a test of the signature matching. If both operations are matched, the child node is processed.

Delete operation is based on the algorithm of the R-tree variant used. It is necessary to mention that after the item is deleted, unperfect signature may be related to the leaf node containing the deleted item. It means that the signature may contain bits describing a tuple that is not included in the R-tree anymore. Signatures for higher levels of the tree may be unperfect as well. Consequently, we may correct the signatures related the changed node. The complete description of this operation is out of scope of this paper.

**Algorithm 1**: Insert algorithm

```
1  N = root ;
2  Z.Push (N);
3  splitted ← true ;
4  while ¬Z.IsEmpty() do
5      if splitted then
6          N.InsertItem();
7          if N.IsOverfull() then
8              N.GenerateSignature();
9              NewNode.GenerateSignature();
10         end
11         else
12             N.AddSignature(tuple.GetSignature());
13             splitted ← true ;
14         end
15         if N.IsLeaf() then
16             N = Z.Pop();
17         end
18         if ¬N.IsLeaf() then
19             if Z.LastMBB() then
20                 Z.Push(N);
21             end
22             else
23                 N = Z.Pop();
24             end
25         end
26     end
27 end
```

# 4 Signature Generating

## 4.1 Signature Methods

The signature is a bit string formed from the terms which are used to index records in a data file [14]. Each term is converted to a bit string by the hashing function. The number of 1's in the signature $S$ is called *weight* $\gamma(S)$. In the case of a query, we build the query signature in the same way as record signatures have been created. If the query signature has 1's in the same positions as the record signature, the record can be considered as a potential match. There can be a case where a record signature matches a query signature, however the record itself does not satisfy the query. This is called the *false drop*.

## 4.2 Signature Generating for the Irrelevant Node Filtering

The Hamming distance is applied for measuring of the signatures similarity. Signature data structures like S-tree [14] are based on clustering of signatures

with the minimal Hamming distance. However, R-tree clusters tuples into MBBs. Nodes do not contain tuples with the minimal Hamming distance. If signatures of tuples in an MBB include many true bits, then the MBB's signature contains almost only true bits. In this case, irrelevant node filtering is not successful. Consequently, one true bit is set for one tuple coordinate and the query signature includes only one true bit for each dimension. In other word, weight of the query signature is rather low.

Our improvement is based on the following assumptions. Query signature weight should be closed to 0.5 as it is known in signature methods [14]. However, signature weight for all tuples of an MBB should be closed to 0.5 as well. These two assumptions are in a contradiction. In this paper, we set more bits as well as we use more hashing functions for one tuple coordinate. In Section 5, we show the efficiency improvement of these novel features.

### 4.3 Implementation

In the case of ESR-tree, signatures are stored out of the R-tree. In this way, tree height is not influenced by the signature length. The relation between an MBB and its signatures is provided by a conversion table. The conversion table contains couples ⟨node index in the R-tree, signature index in the persistent data structure⟩.

## 5 Experimental Results

In our tests[1], we compare ESR-tree with Signature R*-tree, R*-tree, and the proposed B-tree-based implementation. We have implemented B$^+$-tree, R*-tree, and ESR-tree in C++. Three collections have been chosen for these tests. We created two random collections with million tuples of dimensions 2 and 10. The third collection represents a set of paths in an XML document [11]. These paths are modeled as 10-dimensional tuples. All collections have been inserted into the multidimensional data structure (see Table 1 for index characteristics). In our experiments, we do not use the Signature R-tree, we use signatures with one true bit generated for one tuple coordinate as the Signature R-tree uses this. We call this signature as the simple signature. This fact has a significant impact on DAC as well as query processing time. However, result relevances are credible for a comparison of Signature R-tree and ESR-tree.

Efficiency of the narrow range query processing was measured by DAC, $c_Q$, and query processing time. Tested range queries have various number of narrow dimensions $|N_{rq}|$. The 2 kB page size and random accesses are applied in the case of all data structure, therefore, we can measure DAC by the number of MBs read in the secondary storage.

[1] The experiments were executed on an AMD Opteron 865 1.8Ghz, 2.0 MB L2 cache; 2GB of DDR333; Windows 2008 Server.

**Table 1.** Characteristics of the R-tree indices

| | **1st Random Collection** | **2nd Random Collection** | **Real Collection** |
|---|---|---|---|
| **Dimension** | 2 | 10 | 10 |
| **Coordinate value range** | $< 0, 10^9 >$ | $< 0, 5 \cdot 10^4 >$ | $< 0, 10^9 >$ |
| **Result size** | $< 31, 40 >$ | $< 1, 1 >$ | $< 0, 12000 >$ |
| **#Nodes** | 120 | 499 | 852 |
| **Inner item capacity** | 102 | 48 | 48 |
| **#Items** | 8,513 | 15,793 | 21,612 |
| **Node utilization** | 69.6% | 65.9% | 52.8% |
| **#Leaf nodes** | 8,394 | 15,295 | 20,761 |
| **Leaf item capacity** | 170 | 92 | 92 |
| **#Leaf items** | 999,904 | 1,000,000 | 1,031,080 |
| **Leaf node utilization** | 70.1% | 71.1% | 54.0% |
| **#Leaf signatures** $- S^0$ | 8,394 | 15,295 | 20,761 |
| **#Overleaf signatures** $- S^1$ | 117 | 483 | 821 |

### 5.1 1st Random Collection

In Table 2, DAC results are presented for the random collection of dimension 2. We use 2 hashing functions and 3 true bits in the signatures. Results are average values of 10 various queries. We measure DAC for R-tree (RT) and signature data structure (SA).

**Table 2.** 1st Random Collection: DAC

| **Signature Length** | **DAC [MB]** | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Simple | | | ESR-tree $- S^0$ | | | ESR-tree $- S^1$ | | | ESR-tree $- S^0$ AND $S^1$ | | |
| $S^0/S^1$ | RT | SA | RT + SA | RT | SA | RT + SA | RT | SA | RT + SA | RT | SA | RT + SA |
| 128/1,024 | 0.55 | 0.01 | 0.56 | 0.57 | 0.02 | 0.59 | 0.83 | 0.01 | 0.84 | 0.57 | 0.04 | 0.61 |
| 256/3,072 | 0.36 | 0.02 | 0.38 | 0.23 | 0.05 | 0.28 | 0.80 | 0.03 | 0.83 | 0.22 | 0.08 | 0.30 |
| 384/7,168 | 0.28 | 0.04 | 0.32 | 0.14 | 0.07 | 0.21 | 0.65 | 0.07 | 0.72 | 0.12 | 0.13 | **0.25** |
| 512/10,240 | 0.25 | 0.05 | 0.30 | 0.11 | 0.10 | **0.21** | 0.59 | 0.10 | 0.69 | 0.09 | 0.17 | 0.26 |
| 640/12,288 | 0.20 | 0.06 | **0.26** | 0.10 | 0.12 | 0.22 | 0.57 | 0.12 | 0.69 | 0.09 | 0.21 | 0.30 |
| 768/14,336 | 0.20 | 0.07 | 0.27 | 0.10 | 0.15 | 0.25 | 0.54 | 0.14 | **0.69** | 0.08 | 0.24 | 0.32 |
| R-tree without signature filtering: **0.83** | | | | | | | | | | | | |

Obviously, DAC is 4× lower if we compare the R-tree and ESR-tree with the leaf signature of the length 512. DAC of signature reading is an essential part of overall DAC, therefore, we can not use longer signatures. Table 3 including the query processing time supports this conclusion. This time is 5.25× lower than in the case of R*-tree. Obviously, we see that ESR-tree saves 20% of the query processing time of the simple signature.

Another view of the trend is the higher values of relevance. We use the various count of hashing functions, more true bits of the signature as well as various signature lengths. In Table 3, relevances for leaf and overleaf nodes are presented.

Relevance rapidly increases with the increasing signature length. Summary Table 4 presents results for the relevance $> 0.9$. We suppose that the signature weight should be closed to 0.5. In the case of this experiment, we get the signature weight in the range 0.38 – 0.51.

**Table 3.** 1st Random Collection: results for 2 hashing functions, 3 true bits

| **Signature** | **Time [s]** | | | | $c_Q$ **of** $S^0$ | | | | $c_Q$ **of** $S^1$ |
|---|---|---|---|---|---|---|---|---|---|
| **Length** | Simple | $S^0$ | $S^1$ | $S^0$ AND $S^1$ | Simple | $S^0$ | $S^1$ | $S^0$ AND $S^1$ | |
| 128/1024 | 0.045 | 0.064 | 0.069 | 0.042 | 0.19 | 0.18 | 0.14 | 0.18 | 0.67 |
| 256/3072 | 0.037 | 0.020 | 0.061 | 0.020 | 0.26 | 0.36 | 0.14 | 0.37 | 0.70 |
| 384/7168 | 0.022 | 0.020 | 0.050 | 0.020 | 0.31 | 0.61 | 0.16 | 0.66 | 0.87 |
| 512/10240 | 0.022 | 0.016 | 0.044 | 0.011 | 0.34 | 0.85 | 0.16 | 0.89 | 0.94 |
| 640/12288 | 0.020 | 0.016 | 0.045 | 0.014 | 0.41 | 0.93 | 0.17 | 0.95 | 0.94 |
| 768/14336 | 0.020 | 0.019 | 0.044 | 0.016 | 0.44 | 0.96 | 0.17 | 0.98 | 0.98 |
| | R-tree: **0.084** | | | | R-tree: **0.14** | | | | R-tree: **0.67** |

**Table 4.** 1st Random Collection: summary table

| **Signature** | **Results for** $c_Q > 0.9$ | | | |
|---|---|---|---|---|
| **Type** | Signatures Length | DAC – RT [MB] | DAC – SA [MB] | Time [s] |
| 2 hash/ 3 bits | 640/12288 | 0.087 (0.054+ 0.033) | 0.207 | 0.0140 |
| 2 hash/ 1 bit | - | - | - | - |
| 3 hash/ 1 bit | 512/4096 | 0.093 (0.057 + 0.036) | 0.178 | 0.0140 |
| 2 hash/ 6 bits | 768/10240 | 0.088 (0.052+ 0.036) | 0.217 | 0.0078 |

### 5.2 2nd Random Collection

The second collection includes 10-dimensional randomly generated tuples. 3 narrow dimensions of range queries are used. In this case, we can use shorter signatures than in the case of the first test. It seems that this 10-dimensional space is sparser than the 2-dimensional space, therefore, shorter signatures can describe the tuple distribution as well. In Table 5, we can see that DAC of ESR-tree is much more lower than DAC of R*-tree. The query processing time is improved 9×. Obviously, we can see the importance of overleaf signatures in this case. Summary Table 6 presents results for the relevances $> 0.9$.

### 5.3 Real Collection

Third collection contains a set of paths in an XML document. We test 15 range queries $Q_1$–$Q_{15}$ with various narrow dimensions. We use the same signature

**Table 5.** 2nd Random Collection: DAC

| Signature Length $S^0/S^1$ | DAC [MB] | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Simple | | | ESR-tree – $S^0$ | | | ESR-tree – $S^1$ | | | ESR-tree – $S^0$ AND $S^1$ | | |
| | RT | SA | RT + SA | RT | SA | RT + SA | RT | SA | RT + SA | RT | SA | RT + SA |
| 96/1024 | 4.62 | 0.69 | 5.31 | 4.35 | 1.39 | 5.74 | 23.25 | 0.95 | 24.20 | 3.99 | 2.23 | 6.22 |
| 128/1536 | 3.20 | 0.93 | 4.13 | 2.21 | 1.85 | 4.06 | 16.84 | 1.43 | 18.27 | 1.44 | 2.67 | 4.11 |
| 160/2048 | 2.58 | 1.16 | 3.74 | 1.71 | 2.31 | **4.02** | 10.92 | 1.91 | 12.83 | 0.71 | 2.91 | 3.62 |
| 192/2560 | 2.20 | 1.39 | **3.59** | 1.62 | 2.78 | 4.40 | 5.69 | 2.38 | 8.07 | 0.36 | 3.01 | **3.37** |
| 224/3072 | 1.99 | 1.62 | 3.61 | 1.60 | 3.24 | 4.84 | 2.79 | 2.86 | 5.65 | 0.20 | 3.22 | 3.42 |
| 256/3584 | 1.89 | 1.85 | 3.74 | 1.59 | 3.70 | 5.29 | 1.58 | 3.34 | **4.92** | 0.14 | 3.56 | 3.70 |
| R-tree without signature filtering: **25.28** | | | | | | | | | | | | |

**Table 6.** 2nd Random Collection: summary table

| Signature Type | Results for $c_Q > 0.9$ | | | | |
|---|---|---|---|---|---|
| | Signatures length | Relevance | DAC – RT [MB] | DAC – SA [MB] | Time [s] |
| 2 hash/ 3 bits | 256/3584 | 0.95 | 0.140 (0.004 + 0.136) | 3.564 | 0.0780 |
| 2 hash/ 1 bit | 352/2048 | 0.95 | 0.214 (0.004 + 0.210) | 2.499 | 0.0936 |
| 3 hash/ 1 bit | 160/1376 | 0.90 | 0.261 (0.004 + 0.257) | 2.444 | 0.1092 |
| 2 hash/ 6 bits | 352/4096 | 1.00 | 0.342 (0.004 + 0.338) | 4.924 | 0.1420 |

lengths as in the case of the second collection. In Table 7, DAC and $c_Q$ are put forward. We use signature lengths 256/3584 with 2 hashing functions and 3 true bits. We see that the $c_Q$ of Signature R*-tree significantly increases in the comparison with the common R*-tree. Obviously, this relevance is not sufficient in many cases. ESR-tree overcomes the Signature R*-tree and R*-tree.

### 5.4 The Comparison with B-tree

In this test, we compare the efficiency of ESR-tree and B-tree (one B-tree was created for each dimension). We use the queries from the previous test. The join operation takes the most processing time (see Table 8). Obviously, B-tree is much more efficient in the case of small intermediate results. With the increasing size of intermediate results, join processing time increases. Table 8 includes information about all queries. We measure DAC for B-trees (BT) and tuple array (TA) including whole tuples. Obviously, ESR-tree clearly overcomes this B-tree based implementation, DAC is 7.5× lower in the case of ESR-tree. Another important issue is the index size. In Table 9, we see that the index size of the ESR-tree is 3× lower in all tested cases. The experiments show an significant improvement, e.g. applications of two hashing functions causes double increasing of the relevance.

## 6 Conclusion

In this article, we present an improvement of Signature R-tree, data structure for the efficient processing of narrow range queries. Moreover, we introduce an enhanced signature creation that provides more efficient filtration characteristics. Since signatures are relevant only in higher levels of a tree, it is not appropriate to handle them to each MBB of an inner node. We show some advantages

**Table 7.** Real Collection Test: comparison of R*-tree, Signature R*-tree, and ESR-tree

| Query | Result Sizes | R*-tree | | Simple | | ESR-tree – $S^0$ AND $S^1$ | |
|---|---|---|---|---|---|---|---|
| | | $c_Q$ | DAC [MB] | $c_Q$ | DAC [MB] | $c_Q$ | DAC [MB] |
| Q1 | 2000 | 0.76 | 1.04 | 1.00 | 0.82+0.07 | 1.00 | 0.78+0.34 |
| Q2 | 1201 | 0.26 | 13.77 | 0.80 | 4.88+1.03 | 0.98 | 3.91+2.71 |
| Q3 | 12000 | 0.64 | 15.82 | 1.00 | 10.43+1.19 | 1.00 | 10.22+2.98 |
| Q4 | 3 | 0.27 | 0.09 | 1.00 | 0.06+0.003 | 1.00 | 0.04+0.08 |
| Q5 | 10 | 0.36 | 0.09 | 1.00 | 0.07+0.003 | 1.00 | 0.05+0.08 |
| Q6 | 6 | 0.03 | 0.91 | 0.13 | 0.29+0.06 | 1.00 | 0.06+0.24 |
| Q7 | 10 | 0.13 | 0.30 | 0.88 | 0.13+0.02 | 1.00 | 0.07+0.17 |
| Q8 | 18 | 0.48 | 0.18 | 0.63 | 0.16+0.01 | 1.00 | 0.09+0.13 |
| Q9 | 1 | 0.0006 | 6.02 | 0.003 | 1.43+0.45 | 0.08 | 0.13+0.74 |
| Q10 | 27 | 0.03 | 3.99 | 0.23 | 0.68+0.29 | 0.79 | 0.23+0.70 |
| Q11 | 43 | 0.03 | 5.19 | 0.21 | 1.13+0.38 | 0.87 | 0.35+1.11 |
| Q12 | 13 | 0.18 | 0.36 | 0.58 | 0.19+0.02 | 1.00 | 0.11+0.24 |
| Q13 | 1 | 0.06 | 0.13 | 1.00 | 0.07+0.005 | 1.00 | 0.03+0.11 |
| Q14 | 24 | 0.50 | 0.14 | 1.00 | 0.11+0.004 | 1.00 | 0.07+0.14 |
| Q15 | 3 | 0.14 | 0.15 | 1.00 | 0.07+0.01 | 1.00 | 0.05+0.11 |
| Average | | **0.26** | **3.21** | **0.69** | **1.87 + 0.24** | **0.91** | **1.08 + 0.66** |

of longer signatures, however longer signatures mean the lower node's capacity. Consequently, we put signatures of each MBB in a special data structure: a persistent array. In our experiment, we test range queries and compare the efficiency of our approach with R-tree, Signature R-tree, and B-tree based implementation. From DAC point of view, ESR-tree is up to 3× more efficient than R-tree and 6× than B-tree. Obviously, DAC of the signature retrieval is often rather high. Therefore, we want to develop a more efficient data structure for the storage of signatures.

## References

1. R. Bayer. The Universal B-Tree for multidimensional indexing: General Concepts. In *Proceedings of WWCA'97, Tsukuba, Japan*, 1997.
2. 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 1990 ACM SIGMOD*, pages 322–331.
3. S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proceedings of the 22nd International Conference on VLDB*, pages 28–39, San Francisco, U.S.A., 1996. Morgan Kaufmann Publishers.
4. P. Ciaccia, M. Pattela, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proceedings of 23rd International Conference on VLDB*, pages 426–435, 1997.
5. V. Dohnal, C. Gennaro, and P. Zezula. A Metric Index for Approximate Text Management. In *Proceedings of IASTED International Conference Information Systems and Database – ISDB 2002*, 2002.

**Table 8.** Real Collection Test: results of the B-tree-based implementation

| **Query** | $\|N_{rq}\|$ | **Intermediate Result Sizes** | **Result Size** | **DAC (BT + TA) [MB]** | **Time join [s]** | **Time [s]** |
|---|---|---|---|---|---|---|
| Q1 | 2 | 2000; 1031080; | 2000 | 19.80 (15.90 + 3.90) | 9.91 | 14.22 |
| Q2 | 2 | 12000; 2539; | 1201 | 2.58 (0.24 + 2.35) | 0.13 | 0.27 |
| Q3 | 1 | 12000; | 12000 | 23.63 (0.19 + 23.44) | 0 | 1.91 |
| Q4 | 4 | 1031080; 105161; 10; 3; | 3 | 17.51 (17.50 + 0.01) | 4.06 | 4.07 |
| Q5 | 3 | 1031080; 105161; 10; | 10 | 17.51 (17.49 + 0.02) | 4.06 | 4.07 |
| Q6 | 2 | 7512; 126; | 6 | 0.14 (0.13 + 0.01) | 0.031 | 0.032 |
| Q7 | 4 | 1031080; 279205; 13; 10; | 10 | 20.19 (20.17 + 0.02) | 4.73 | 4.73 |
| Q8 | 3 | 1031080; 279205; 18; | 18 | 20.20 (20.17 + 0.04) | 4.79 | 4.79 |
| Q9 | 2 | 25500; 1; | 1 | 0.41 (0.41 + 0.002) | 0.093 | 0.094 |
| Q10 | 2 | 37843; 104; | 27 | 0.65 (0.60 + 0.05) | 0.14 | 0.14 |
| Q11 | 2 | 12715; 43; | 43 | 0.30 (0.21 + 0.08) | 0.046 | 0.047 |
| Q12 | 4 | 1031080; 251465; 69340; 13; | 13 | 20.84 (20.81 + 0.03) | 4.86 | 4.86 |
| Q13 | 4 | 1031080; 105161; 9; 1; | 1 | 17.502 (17.50 + 0.002) | 4.06 | 4.06 |
| Q14 | 4 | 1031080; 4324; 25; 24; | 24 | 16.00 (15.95 + 0.05) | 3.77 | 3.77 |
| Q15 | 4 | 1031080; 105161; 10; 3; | 3 | 17.51 (17.50 + 0.01) | 4.06 | 4.06 |
| Average | | | **1024** | **12.98 (10.98 + 1.99)** | **2.98** | **3.41** |

**Table 9.** Sizes of indexes

| **Dimension** | **Data Collection** | **R-tree + SA [MB]** | **B-trees + Tuples Array [MB]** |
|---|---|---|---|
| 2 | 1st Random | 16.89 (16.6 + 0.29) | 40.95 (2 × 16.6 + 7.63) |
| 10 | 2st Random | 65.1 (61.6 + 3.49) | 204.2 (10 × 16.6 + 38.2) |
| 10 | Real | 89.38 (84.4 + 4.98) | 204.4 (10 × 16.6 + 38.4) |


6. R. Fenk. The BUB-Tree. In *Proceedings of 28rd VLDB International Conference on VLDB, Hongkong, China*, 2002.
7. M. Freeston. A General Solution of the $n$-dimensional B-tree Problem. In *Proceedings of SIGMOD International Conference, San Jose, USA*, 1995.
8. T. Grust. Accelerating XPath Location Steps. In *Proceedings of ACM SIGMOD 2002, Madison, USA*, June 4-6, 2002.
9. A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of ACM SIGMOD 1984, Annual Meeting, Boston, USA*, pages 47–57. ACM Press, June 1984.
10. N. Karayannidis, A. Tsois, T. Sellis, R. Pieringer, V. Markl, F. Ramsak, R. Fenk, K. Elhardt, and R. Bayer. Processing Star Queries on Hierarchically-Clustered Fact Tables. In *Proceedings of VLDB Conf. 2002, Hongkong, China*, 2002.
11. M. Krátký, J. Pokorný, and V. Snášel. Implementation of XPath Axes in the Multi-dimensional Approach to Indexing XML Data. In *Current Trends in Database Technology, Int'l Conference on EDBT 2004*, volume 3268. Springer–Verlag, 2004.
12. M. Krátký, T. Skopal, and V. Snášel. Multidimensional Term Indexing for Efficient Processing of Complex Queries. *Kybernetika, Journal*, 40(3):381–396, 2004.
13. M. Krátký, V. Snášel, P. Zezula, and J. Pokorný. Efficient Processing of Narrow Range Queries in the R-Tree. In *Proceedings of IDEAS 2006*. IEEE CS Press, 2006.
14. Y. Manolopoulos, A. Nanopoulos, and E. Tousidou. *Advanced Signature Indexing for Multimedia and Web Applications*. Kluwer, 2003.
15. Y. Manolopoulos, Y. Theodoridis, and V. Tsotras. *Advanced Database Indexing*. Kluwer Academic Publisher, 2001.
16. C. Yu. *High-Dimensional Indexing*. Springer–Verlag, LNCS 2341, 2002.