On Support of Ordering in Multidimensional Data Structures*

Filip Křižka, Michal Krátký, and Radim Bača

Department of Computer Science, VŠB – Technical University of Ostrava
17. listopadu 15, Ostrava, Czech Republic
{filip.krizka,michal.kratky,radim.baca}@vsb.cz

Abstract. Multidimensional data structures are applied in many areas, e.g. in data mining, indexing multimedia data and text documents, and so on. There are some applications where the range query result must be ordered. A typical case is the result with tuples sorted according to values in one dimension defined by the `ORDER BY` clause of an SQL statement. If we use a multidimensional data structure, the result set is sorted after the range query is processed. Since the sort operation must often be processed on tuples stored in the secondary storage, an external sorting algorithm must be utilized. Therefore, this operation is time consuming especially for a large result set. In this paper, we introduce a new data structure, a variant of the R-tree, supporting a storage of ordered tuples.

Keywords: multidimensional data structures, ordered tuples of the range query result, R-tree, Ordered R-tree

1 Introduction

Multidimensional data structures [25, 30] have been widely applied in many data management areas. Indexing spatial data is their natural application but there are many applications in different domain areas. In the case of spatial data, structures often store two- and three- dimensional objects. In the case of multimedia data, there are spaces with dimension values of up to 100,000 to be indexed. There have been many data structures introduced in the past, e.g. R-tree [10], R*-tree [3], UB-tree [2], X-tree [5], and BUB-tree [7]. These data structures have been applied in many areas; the well-known R-tree is included in Oracle Spatial[1] – an Oracle option for indexing spatial data. UB-tree [2] is applied for an implementation of more indices related to one table in the relational data model [23]. In addition, multidimensional data structures are applied in a wide range of applications, e.g. indexing XML data [9, 15, 14], terms [16, 6], etc.

In the case of the R-tree, tuples are clustered in a tree's page when *MBRs* (*Minimal Bounding Rectangles*) are built. It supports various types of queries, e.g. point and range queries. As mentioned previously, multidimensional data structures may be applied as index data structures supporting the storage of

* Work is partially supported by Grant of GACR No. P202/10/0573.

[1] `http://www.oracle.com/database/spatial.html`

tuples in relational or object-relational DBMS [8]. If B-tree or B$^+$-tree are applied for the support, we must create one tree for each attribute to be indexed. Another alternative is to use the so-called compound-key which includes more attributes. In this case, some queries are processed by an often inefficient sequential scan. When we consider multidimensional data structures, one index is created for all attributes to be indexed. In this case, random accesses may influence the efficiency of query processing [19]. We must observe that the efficiency of range query processing in the B-tree is also influenced by this issue.

There are some applications where ordering of tuples in the result set is required. This order may be defined by ordering of values of one attribute. A common example is an SQL statement with the `ORDER BY` clause. In this case, we often define a restriction of attribute values, which means the range query can be applied for the processing of this query; however, we often require an ordering defined by the `ORDER BY` clause. If we consider multidimensional data structures, there is no support for the ordering; the result tuples must be sorted after the query is processed. Since the number of sort operations and the number of tuples in the result may be high in the case of DBMS, we must utilize an external sorting algorithm [13, 29]. If large result sets are sorted, this operation is time consuming. When we consider a data structure without a support of ordering, we can not utilize lazy query processing. In the case of lazy query processing, a query is progressively processed and we can not store all tuples of the result in an additional data structure. When we consider multidimensional data structures without a support of ordering, this lazy evaluation is not possible; the query must be processed in one and result tuples must be inserted in a persistent data structure and sorted [20].

In this article, we introduce a multidimensional data structure with a support of ordering. A cost-based relational query optimizer [8] can simply utilize this data structure and decide when it is appropriate to utilize this data structure instead of common query processing and sorting of the result set. Since the R-tree is the most popular data structure we present the support of ordering for the R-tree.

Although it can be seen that there is a relation between multidimensional data structures and ordering, we only find data structures which enable the indexing of multidimensional data using an order. Such data structures include *UB-tree* [2, 22] and *BUB-tree* [7] which utilize Z-ordering for indexing of multidimensional data. On the other hand, if we apply B-tree or B$^+$-tree as the solution of this issue, a range query will be processed by an often inefficient sequential scan.

This paper is organized as follows: In Section 2, we present R-tree and its variants. We describe the motivation to the support of ordering in Section 3 in more detail. Section 4 includes some notices to ordering of the range query result. In Section 5, we describe our new variant of the R-tree and show the basic principles of this data structure. Experimental results are put forward in Section 6. In the last section we summarize the paper content and outline possibilities of our future work.

2 R-tree and its Variants

R-tree [10] can be thought of as an extension of B^+-trees in a multi-dimensional space. It corresponds to a hierarchy of nested n-dimensional MBRs. If $\mathcal{N}$ is an inner page, it contains couples of the form (R_i, P_i), where P_i is a pointer to a child of the page $\mathcal{N}$. If R is its MBR, then the rectangles R_i corresponding to the children $\mathcal{N}_i$ of $\mathcal{N}$ are contained in R. Rectangles at the same tree level may overlap. In this article, we consider only tuples stored in leaf nodes due to the fact that it corresponds to our application domain. Unless a page of the R-tree is the root, its number of entries is between m and M and it corresponds to a disk page. Other properties of the R-tree include the following:

– Whenever the number of a page's children drops below m, the page is deleted and its descendants are distributed among the sibling pages. The upper bound M depends on the size of the disk page.
– The root page has at least two entries, unless it is a leaf.
– The R-tree is height-balanced; i.e. all leaves are at the same level. The height of an R-tree is at most $\lfloor \log_m N \rfloor - 1$ for N index records $(N > 1)$.

Given a set of $M+1$ entries, each entry is assigned to one of the two produced pages, according to the criterion of minimum area, i.e. the selected page is the one that will be enlarged the least in order to include the new entry. It significantly affects the index performance. Three split techniques (*Linear*, *Quadratic*, and *Exponential*) proposed in [10] are based on a heuristic optimization.

R-tree performance is usually measured with respect to the retrieval cost (in terms of DAC) of queries. The majority of performance studies concerns point, range, and *k-NN* queries. Considering the R-tree performance, the concepts of page *coverage* and *overlap* between pages are important. Obviously, an efficient R-tree search requires that both the overlap and coverage are minimized. Minimal coverage reduces the amount of dead area covered by R-tree pages. The minimal overlap is even more critical than the minimal coverage, searching to objects falling in the area of k overlapping pages; up to k paths to the leaf pages, may have to be processed in such a way.

Variants of R-trees differ in the way they perform the split algorithm during insertions, i.e. which minimization criteria are used. Literature has identified a variety of criteria for the layout of keys on pages that affect retrieval performance. These criteria are: *minimal page area*, *minimal overlap between pages*, *minimal page margins* or *maximized page utilization.* It is impossible to optimize all of these parameters simultaneously.

Authors of [21, 25] put forward many variants of the R-tree. The main feature of R^*-trees [3] involves the page-splitting policy. Therefore, the R^*-tree differs from the R-trees mainly in the insertion algorithm. Although original R-tree algorithms tried only to minimize the area covered by MBRs, the R^*-tree algorithms also take other objectives into account. Clipping-based schemes do not allow any overlaps between bucket regions; they have to be mutually `disjointed`. A typical access method of this kind is the R^+-tree [26] which allows no overlap between regions corresponding to pages at the same tree level and an object can be stored in more than one leaf page.

Other approaches to an improvement of original R-trees release some of their basic features. For example, the MBRs have been replaced by minimum bounding spheres or polygons. In [4], R^+-trees are extended to support *k-NN* queries. In [17], authors introduce a variant for efficient processing of narrow range queries. In our best knowledge, R-tree variant supporting the ordered multidimensional data or multidimensional data structure supporting a general ordering do not exist in the literature.

3 Motivation

Although it can be seen that multidimensional indexing and ordering are in contradiction, there are real applications where we need to sort the range query result. Let us consider an implementation of indices in an RDBMS. Multidimensional data structures can be utilized for the implementation instead of more B-tree or B^+-tree indices. Then an SQL statement is processed by a range query. Users often applied the `ORDER BY` clause which sorts a result set according to values of one or more attributes. In other words, we define an order on tuples of the result set.

A common way to evaluate these queries is to process the range query in a multidimensional data structure and sort the result set of the range query. Since the number of sort operations and tuples in the result can be high, we must utilize an external sorting algorithm [13, 29]. In the case of large result sets the sorting is the time consuming operation. In this article, we introduce a multidimensional data structure supporting an order on multidimensional tuples. Since the R-tree is the most popular data structure, we present the support of ordering for the R-tree; we introduce the Ordered R-tree.

We assume that there are applications where we need always (or very often) use the ordering of a result set according the same attributes. We can apply B-tree or B^+-tree indices for the support, but we must create one tree for each attribute to be indexed. Another alternative is to use so called compound-key which includes more attributes. In these cases some queries are processed by an often inefficient sequential scan. For example, let us consider the compound key $\{A_1, A_2, A_3\}$ of a table. The following query:

```
SELECT * FROM TABLE_NAME
  WHERE A1 BETWEEN VAL_A1_LOW AND VAL_A1_HI
  ORDER BY A1,A2,A3;
```

is processed by a range query and no irrelevant tuples are retrieved. However, the following query:

```
SELECT * FROM TABLE_NAME
  WHERE A3 BETWEEN VAL_A3_LOW AND VAL_A3_LOW
  ORDER BY A1,A2,A3;
```

must be processed by an efficient sequential scan of the whole table. Another example of the SQL query which retrieves the ordered result is as follows:

```
SELECT * FROM V_DAY_N
  WHERE YEAR BETWEEN 2000 AND 2010 AND MONTH BETWEEN 2 and 4
  ORDER BY STATION, ELEMENT, YEAR, MONTH, DAY, TIME;
```

This statement is an example of the query over the climatological database *Clidata*[2]. In this case, meteorological stations product data of measured elements for a period. Data has been measured in the Czech Republic from 1775. Elements include: Temperature, Pressure, Wind speed, Wind direction, Precipitation and so on. It means, the main attributes of this table are as follows: `STATION, ELEMENT, DAY, MONTH, YEAR, TIME, VALUE`. We need always or very often select tuples of the table in the same order: {`STATION, ELEMENT, YEAR, MONTH, DAY, TIME`}. This real application represents a motivation to introduce the Ordered R-tree.

4 Range Query Processing in R-tree

A range query is processed by a depth-first search algorithm [10]. If the query rectangle intersects an MBR then the node related to the MBR is retrieved and searched. In general, there is no order; therefore, tuples of a result set are sorted in the same order in which these tuples were inserted into leaf nodes and new MBRs were inserted into inner nodes. In the case of bulk-load algorithms, tuples are often sorted under an order and MBRs are created for these tuples [11]. We will show that the ordering of tuples in leaf nodes does not guarantee the ordered result set.

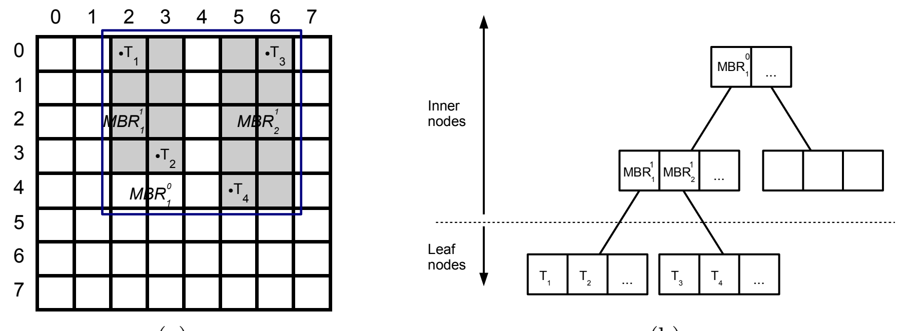
Example 1. (Range Query Processing) Let us consider two MBRs in Figure 1. A depth-first search range query algorithm retrieves tuples of MBR_1^0 (T_1 and T_2) and then it retrieves tuples of MBR_2^0 (T_3 and T_4); consequently, the result set includes: T_1, T_2, T_3, T_4.

When we consider the previous depicted motivation, the `ORDER BY` clause, we want to retrieve tuples under the well-known Lexicographical order of tuples. This order is often utilized in the case of term ordering, however the `ORDER BY` clause applied on a set of ordered attributes works in the same way.

Definition 1. *Lexicographical Order*

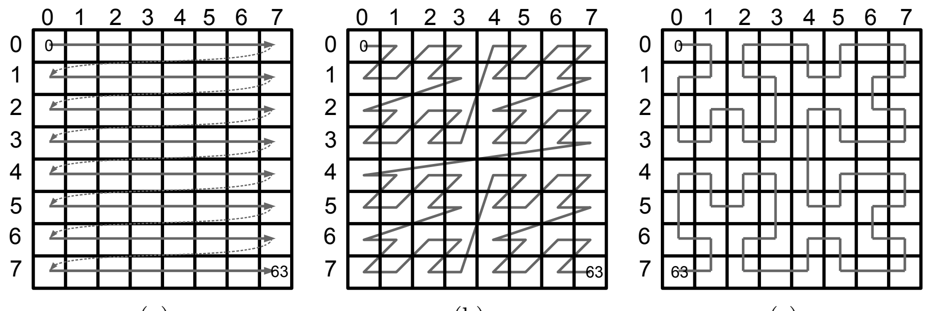
Let Ω be an n-dimensional space and $t_1 = (a_{1,1}, a_{1,2}, \ldots, a_{1,n})$ and $t_2 = (b_{2,1}, b_{2,2}, \ldots, b_{2,n})$ be two tuples of this space. The Lexicographical order is defined: $t_1 < t_2 \iff (\exists m > 0)(\forall i < m)(a_{1,i} = b_{2,i}) \wedge (a_{1,m} < b_{2,m})$.

Example 2. Let us consider the space in Figure 1 and the Lexicographical order which prefers the attribute A_1 before the attribute A_2. The tuples are sorted under this order as is follows: T_1, T_3, T_2, T_4.



(a) (b)

Fig. 1. Example of a depth-first search range query algorithm: (a) 2-dimensional space (b) R-tree



(a) (b) (c)

Fig. 2. Space filling curves (a) C-curve (b) Z-curve (c) Hilbert-curve

Let us consider space filling curves like C-curve, Z-curve or Hilbert curve [24] depicted in Figure 2. It is clear that the Lexicographical order is equivalent to C-ordering. One class of multidimensional data structures utilizes space filling curves. UB-tree [2, 22] and BUB-tree [7] utilize Z-curve to order tuples, regions included in inner nodes are defined as intervals of the Z-curve. Another way is to use a space filling curve for clustering of tuples in a multidimensional space; however, regions are created according to a known data structure like R-tree. Examples of this way are Hilbert R-tree [12] or some bulk-loaded algorithms [11]. There are other data structures utilizing ordering to index multidimensional data [18] or applications which utilize ordering to multidimensional data processing, e.g. in [1] authors introduced a clustering methods based on an order.

There are two ways how to develop a data structure supporting an order. The first way is to create regions as intervals of the order used. The main problem of

[2] `http://portal.chmi.cz`

this way is that it is necessary to develop an algorithm checking if the MBR and the regions are intersected. This algorithm is then used by the range query algorithm. In [22, 28] authors introduced this algorithm of UB-tree and Z-ordering. The similar algorithm must be implemented for each order used. The second way is to use a known data structure and change the build algorithm to guarantee the ordered result set. Since the R-tree is the most popular data structure we present the support of ordering for the R-tree.

5 Ordered R-tree

5.1 Introduction

Let us consider a total order $\mathcal{O}$ on a set of all tuples in a multidimensional space $\Omega = D_1 \times D_2 \times \ldots \times D_n$, where n is the space dimension. An MBR is defined by two tuples Q_L and Q_H. An MBR includes other MBRs in the case of inner nodes or tuples in the case of leaf nodes.

Definition 2. *(MBR Order Condition)*
Let us consider a total order $\mathcal{O}$ on all tuples of Ω. Let MBR_1 and MBR_2 be minimal bounding rectangles in the same level of the tree. MBR Order condition is defined as follows: $MBR_1 \leq MBR_2$ iff each tuple of MBR_1 is lower or equal to all tuples of MBR_2.

It is clear that if all couples of MBR's in each level of the tree meet the MBR Order condition, the result set of each range query is ordered under $\mathcal{O}$. The first step to guarantee the order on tuples is to order tuples in each leaf node. The second step is to sort MBRs in each inner node. Since the R-tree is built by a hierarchy of MBR's which create different space regions compared to the regions created by the order, we can not order MBR's according to their Q_L. In the next example, we show why we can not utilize Q_L to the guarantee of the order on tuples in the R-tree.

Example 3. (Violation of the MBR Order Condition in the R-tree)
Let us suppose the MBR_R in Figure 3(a) including tuples T_1–T_4. We use the order which prefers values of the first dimension before values of the second dimension. In this case, it is possible to split this node so that ${Q_L}^{MBR_1} \leq {Q_L}^{MBR_2}$ and the MBR Order condition is met. A range query containing both MBRs returns the following correct result T_1,T_2, T_3, T_4, where $T_i \leq T_{i+1}$, $1 \leq i \leq 3$.

After the tuple T_5 is inserted, MBR_1 and MBR_2 must be created to guarantee the order (see Figure 3(b)). In this case, ${Q_L}^{MBR_1} = (2,3)$, ${Q_L}^{MBR_2} = (2,0)$, ${Q_L}^{MBR_1} \geq {Q_L}^{MBR_2}$. A range query containing both MBRs, e.g. the range query represented by this statement: `SELECT * FROM` Ω `WHERE` A_1 `BETWEEN (2,4) and` A_2 `BETWEEN(0,7) ORDER BY (`A_1`,`A_2`)`, where A_i is the attribute corresponding to D_i, $1 \leq i \leq 2$, returns the following unordered result T_3, T_5, T_4, T_1, T_2 although tuples in both MBR's are ordered. It means, we can not utilize Q_L to the guarantee of the order on tuples in the R-tree.

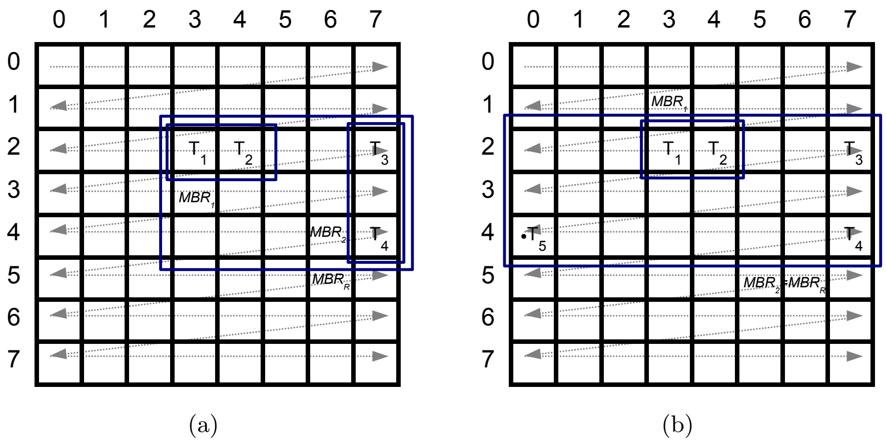


Fig. 3. Violation of the MBR Order Condition in the R-tree – the situation (a) before the insert (b) after the insert

5.2 First Tuple Concept

To guarantee the MBR Order condition we must introduce the first tuple concept.

Definition 3. *(The first tuple of an MBR)*
Let $\mathcal{O}$ be an order on tuples of Ω and MBR be a minimal bounding rectangle. The first tuple of the MBR, FT^{MBR}, is defined as follows:

$$FT^{MBR} = \begin{cases} \textit{the minimal tuple of the MBR} & \textit{if the MBR is the leaf node MBR} \\ \textit{the minimal first tuple of all child MBRs} & \textit{if the MBR is the inner node MBR} \end{cases}$$

Theorem 1. *(Ordered Result Set of the Range Query)*
If MBRs in each inner node of an R-tree are sorted according to their FT and the MBR condition is met for all neighbor MBR's in each level of the R-tree, then the result set of each range query processed by a depth-first technique is ordered.

Proof. Let us suppose an arbitrary MBR_i of a page $\mathcal{N}$ including m tuples. If we choose an MBR_j where $i < j \leq m$ then $FT^{MBR_i} < FT^{MBR_j}$ since MBRs are sorted. Since the MBR Order condition is met, all tuples of $MBR_j > FT^{MBR_i}$. It means that a range query algorithm utilizing the depth-first search technique retrieves ordered tuples.

Example 4. (First Tuple)
Let us suppose MBR_R in Figure 3(b), $FT^{MBR_1} = T_1$, $FT^{MBR_2} = T_3$. If we sort MBR_1 and MBR_2 in MBR_R then we get the ordered set: $\{MBR_1, MBR_2\}$ since $FT^{MBR_1} < FT^{MBR_2}$. A range query containing both MBRs returns the following ordered result set: T_1, T_2, T_3, T_4, T_5.

Since MBR of each node is stored in its parent node, we must handle first tuple of each node, except the root node, in an array to guarantee the tuple order.

5.3 Operations of the Ordered R-tree

The main operation influenced by the Fist Tuple concept is the insert operation. In Algorithm 1 we see an algorithm of the insert operation including the first tuple management. For the sake of simplicity we do not differentiate between inner and leaf nodes as well as between inner and leaf items, therefore, we use overloading functions `Insert` and `Split` as well as overloading items and nodes. In Lines 1–6, the proper leaf node is found by an iteration through inner nodes, each inner node is put on the stack. In Lines 7–23, item is inserted in the node (Line 12) and indices are modified. The variable `flagInsert` is true if the `Insert` and `Split` operations are invocated in the current level of the tree. In this way, the changes of MBR are propagated to the root node.

On the other hand, the variable `flagFirstTuple` controls the propagation of FT to the root node. The procedure `InsertOrUpdateFirstTuple` works in this way: if the inserted tuple is the first tuple of the node, then insert or update this first tuple. In the procedure `Insert`, tuple (or MBR) is inserted in the leaf node (inner node). Items are sort-inserted in the both cases; in the case of inner nodes first tuples must be utilized. Obviously, the split operation utilizes the ordering when a node is split; the second node includes tuples $>$ all tuples of the first node.

The find and range query operations are without any change. In the case of the delete operation, we must manage the update of first tuples. Consequently, find and delete operations work in $O(\log M)$, where M is the number of tuples in the tree.

5.4 First Tuple Storage

To manage first tuples for all MBR's we can use a persistent array including all first tuples. This array contains couples (node id, first tuple), where node id references the tree node related to the first tuple. In Figure 4, we see an example of the proposed data structure. Since the size of the array is equal to the number of all nodes in the R-tree, we can often manage this data structure in the main memory; pages of this data structure are not retrieved from the secondary storage. For example, if the R-tree includes 10^6 tuples, the maximum number of items in leaf nodes is 200 and the maximum number of items in inner nodes is 100. The array includes approximately 12,000 items. Let us consider the space dimension=5, 4 B for one attribute value and node id, we get the total size of the array: 288,000 B = 281,25 kB. When we consider the delete operation, we can utilize hashing or B-tree for the implementation of the array.

Algorithm 1: Insert Operation of the Ordered R-tree

```
procedure: Insert (tuple)
// go down;
Node node ← rootNode ;
while node is an inner node do
    nodeStack.Push (node);
    node ← node.GetProperChild();
end
// go up;
int flagInsert ← CONST_INSERT ;
bool flagFirstTuple ← false;
while node!= null do
    if flagInsert = CONST_INSERT then
        flagInsert ← Insert (node, item, newNode);
        flagFirstTuple ← InsertOrUpdateFirstTuple (node.Id, item);
    end
    else
        if flagFirstTuple = true then
            flagFirstTuple ← InsertOrUpdateFirstTuple (node.Id, item);
        end
    end
    if flagInsert = CONST_SPLIT then
        flagInsert ← Split (node, item, newNode);
        flagFirstTuple ← InsertOrUpdateFirstTuple (newNode.Id, item);
    end
    node ← nodeStack.Pop();
end
function : InsertOrUpdateFirstTuple (nodeId, tuple)
bool flagFirstTuple ← false;
if Does a first tuple exist in the MBB? then
    if Is tuple the new first tuple? then
        flagFirstTuple ← UpdateFT (nodeId, tuple);
    end
end
else
    flagFirstTuple ←InsertFT (nodeId, tuple);
end
return flagFirstTuple ;
```

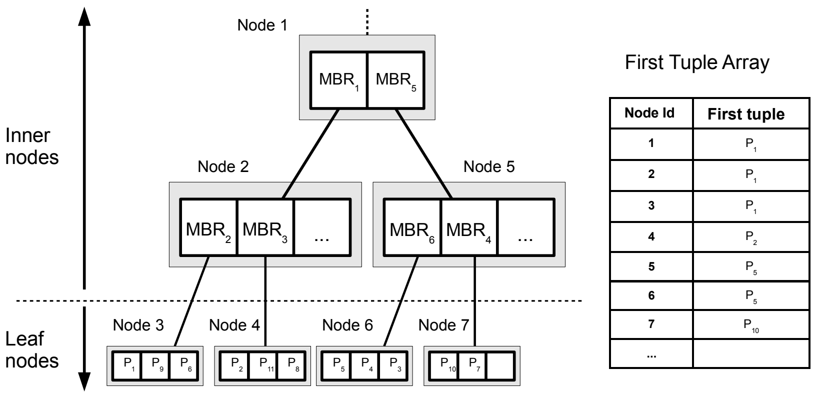


Fig. 4. A structure of the Ordered R-Tree

6 Experimental Results

In our experiments[3], we used the proposed real collection of meteorological data including 1,391,049 records. The scheme contains 6 attributes: `STATION`, `ELEMENT`, `YEAR`, `MONTH`, `DAY`, `TIME`. It means the space dimension is 6. The number of stations is 10; therefore, the attribute `STATION` has 10 possible values. The number of elements is 2. Other attributes include the time of the measurement for the station and element. All data structures have been implemented in C++. We built 2 data structures: R*-tree and Ordered R-tree. Their parameters are shown in Table 1. The page size 2,048 B is used for all trees.

Table 1. R-trees statistics

	R*-tree	**Ordered R-tree**
Tree Height	4	4
Build Time [s]	233	159
# Inner Nodes	1,318	1,372
# Leaf Nodes	27,826	29,663
Index Size [kB]	58,290	62,072
FT Index Size [kB]	–	794

Data structures were built by a common tuple-by-tuple way. Since R-tree and Ordered R-tree create different regions, both build times are different for the same order of the inserted tuples.

[3] The experiments were executed on an Intel® Core 2 Duo 2.4 Ghz, 512 kB L2 cache; 3 GB RAM; Windows 7.

In our experiments, we used 30 range queries with various selectivities; the result set includes 0–548,725 tuples. In Table 2, we show some queries in more detail. Terms *min* and *max* denote the minimum and maximum values of the domain, respectively.

Table 2. Some tested range queries

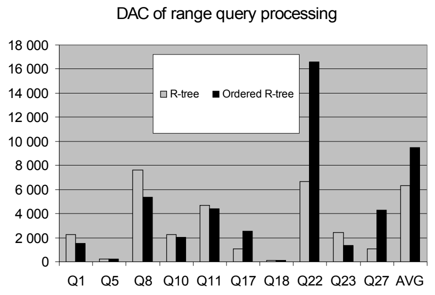
Query	Range query defined by $\mathbf{Q}_l$:$\mathbf{Q}_h$	Result Size
$\mathbb{Q}_1$	$(4, 2, 1960, min, min, min) : (4, 2, 1980, max, max, max)$	30,684
$\mathbb{Q}_5$	$(min, 2, 2000, 2, 3, min)$:$(max, 2, 2000, 2, 3, max)$	36
$\mathbb{Q}_8$	$(3, min, min, 10, min, min)$:$(8, max, max, 10, max, max)$	47,531
$\mathbb{Q}_{10}$	$(6, min, 1980, 2, min, min)$:$(8, max, 2005, 5, max, max)$	32,475
$\mathbb{Q}_{11}$	$(5, min, min, min, min, min)$:$(5, max, max, max, max, max)$	92,111
$\mathbb{Q}_{17}$	$(5, min, min, min, 2, min)$:$(5, max, max, max, 2, max)$	3,030
$\mathbb{Q}_{18}$	$(min, 1, min, min, min, 900)$:$(max, 1, max, max, max, 1200)$	0
$\mathbb{Q}_{22}$	$(9, min, min, min, min, 900)$:$(10, max, max, max, max, 1500)$	166,514
$\mathbb{Q}_{23}$	$(3, 2, 1990, min, min, min)$:$(5, 2, 1995, max, max, max)$	26,292
$\mathbb{Q}_{27}$	$(min, min, 1980, 7, 5, min)$:$(max, max, 2100, 7, 5, max)$	1,372

As usual, tests are processed with a cold cache (OS cache as well as cache buffer of indices). For all tests we measure query processing time and Disk Access Cost (DAC). Since in the case of R*-tree the range query's result set must be sorted, we measure the sorting time and DAC of the sort operation. For the time measurement, we repeat the tests 10× and calculate the average time. Since the range query is processed by random disk accesses, DAC is equal to the number of disk accesses during query processing [27]. We used Merge sort [13] as the algorithm sorting the result set.

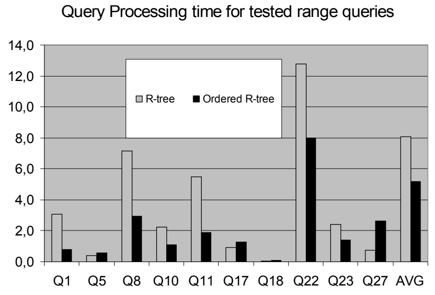
Table 3. DAC and times [s] for tested range queries

Query	**R*-tree**				**Ordered R*-tree**	
	DAC		**Time [s]**		**DAC**	**Time [s]**
	Query	Sort	Complete Query	Sort		
$\mathbb{Q}_1$	2,279	17,408	3.1	0.8	1,545	0.8
$\mathbb{Q}_5$	201	205	0.4	0	221	0.6
$\mathbb{Q}_8$	7,600	31,795	7.2	1.3	5,372	2.9
$\mathbb{Q}_{10}$	2,252	18,176	2.2	0.8	2,025	1.1
⋮						
Avg.	6,297	56,627	**8.1**	2.8	8,478	**5.2**

Although we show results of 10 queries in more detail, average results are calculated for all 30 queries. DAC results are shown in Table 3 and Figure 5(a). Whereas Table 3 includes DAC of the range query as well as sort operation, Figure 5(a) includes only DAC of the range query. Let us consider DAC of range query processing. We see that the DAC for the Ordered R-tree is 125% of DAC for the R*-tree. It is because the order influences the tree building. Although DAC for the range query and sort operation are not comparable due to the fact that it is simply possible to reduce the number of disk accesses using a cache buffer in the case of the sort operation, we see that DAC of the sort operation is enormous especially for queries with a large result set.



(a)



(b)

Fig. 5. (a) DAC and (b) Query processing time for tested range queries

Query processing time is shown in Table 3 and Figure 5(b). Moreover, Table 3 includes the time of the sort operation. We see that this time influences the query processing time especially for range queries with the large result set. We see that our Orderer R-tree saves 35% of the query processing time compared to the R*-tree.

7 Conclusion

In this article, we introduced a new data structure, a variant of the R-tree, supporting a storage of ordered tuples. In this way, the range query result is ordered without an application of any time consuming sort operation. Our experiments show an improvement of this data structure in comparison with the R-tree. Orderer R-tree saves 35% of the query processing time compared to the R*-tree. The new data structure is particularly efficient in the case of large result sets or in the case of lazy query processing when tuples are not retrieved immediately. In the case of the R-tree, range query must be processed completely and then the result set must be sorted. In our future work, we want to adopt this data structure for indexing XML data where the lazy query processing of ordered tuples can be applied.

References

1. Ilya Muchnik Baiyang Liu, Casimir Kulikowski. Global Ordering For Multi-Dimensional Data: Comparison with K-means Clustering. *DIMACS Technical Report 2009-13*, 2009.
2. Rudolf Bayer. The Universal B-Tree for multidimensional indexing: General Concepts. In *Proceedings of World-Wide Computing and Its Applications (WWCA 1997), Tsukuba, Japan*, LNCS, pages 198–209. Springer–Verlag, 1997.
3. Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard 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)*, 1990.
4. A. Belussi, E. Bertino, and B. Cataniac. Using Spatial Data Access Structures for Filtering Nearest Neighbor Queries. *Data & Knowledge Engineering*, 40(1):1–31, 2002.
5. Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree : An Index Structure for High-Dimensional Data. In *Proceedings of 22th International Conference on Very Large Data Bases (VLDB 1996)*, pages 28–39, 1996.
6. Vlastislav Dohnal, Claudio Gennaro, and Pavel Zezula. A Metric Index for Approximate Text Management. In *Proceedings of IASTED International Conference Information Systems and Database (ISDB 2002)*, pages 37–42. ACTA Press, 2002.
7. Robert Fenk. The BUB-Tree. In *Proceedings of 28rd VLDB International Conference on Very Large Data Bases (VLDB 2002), Hongkong, China*. Morgan Kaufmann, 2002.
8. H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.
9. Torsten Grüst. Accelerating XPath Location Steps. In *Proceedings of ACM International Conference on Management of Data (SIGMOD 2002)*, pages 109–120, 2002.
10. Antonin 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. Ibrahim Kamel and Christos Faloutsos. On packing R-trees. In *Proceedings of the 2nd International Conference on Information and Knowledge Management (CIKM 1993)*, pages 490–499. ACM, 1993.
12. Ibrahim Kamel and Christos Faloutsos. Hilbert R-tree: An Improved R-tree using Fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB 1994)*, pages 500–509, 1994.
13. Donald Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition*. Addison-Wesley, 1998.
14. Michal Krátký, Radim Bača, and Václav Snášel. On the Efficient Processing Regular Path Expressions of an Enormous Volume of XML Data. In *Proceedings of DEXA 2007*, volume 4653/2007. Springer–Verlag, 2007.
15. Michal Krátký, Jaroslav Pokorný, and Václav Snášel. Implementation of XPath Axes in the Multi-dimensional Approach to Indexing XML Data. In *Current Trends in Database Technology, EDBT 2004*, volume 3268. Springer–Verlag, 2004.
16. Michal Krátký, Tomáš Skopal, and Václav Snášel. Multidimensional Term Indexing for Efficient Processing of Complex Queries. *Kybernetika, Journal*, 40(3):381–396, 2004.

17. Michal Krátký, Václav Snášel, P. Zezula, and Jaroslav Pokorný. Efficient Processing of Narrow Range Queries in the R-Tree. In *Proceedings of the 10th International Database Engineering and Applications Symposium (IDEAS 2006)*, pages 69–79. IEEE, 2006.
18. A. Kumar. G-Tree: A New Data Structure for Organizing Multidimensional Data. *IEEE Transactions on Knowledge and Data Engineering*, 6(2):341–347, 1994.
19. Tapio Lahdenmäki and Michael Leach. *Relational Database Index Design and the Optimizers.* John Wiley and Sons, New Jersey, 2005.
20. Sam S. Lightstone, Toby J. Teorey, and Tom Nadeau. *Physical Database Design: the Database Professional's Guide.* Morgan Kaufmann, 2007.
21. Y. Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos, and Y. Theodoridis. *R-Trees: Theory and Applications.* Springer, 2005.
22. Volker 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`.
23. F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer. Integrating the UB-tree into a Database System Kernel. In *Proceedings of 26th International Conference on Very Large Data Bases (VLDB 2000)*, pages 263–272, 2000.
24. Hans Sagan. *Space Filling Curves.* Springer-Verlag, 1994.
25. Hanan Samet. *Foundations of Multidimensional and Metric Data Structures.* Morgan Kaufmann, 2006.
26. Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R^+-Tree: A Dynamic Index For Multi-Dimensional Objects. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB 1997)*, pages 507–518, 1997.
27. Dennis Shasha and Philippe Bonnet. *Database Tuning: Principles, Experiments, and Troubleshooting Techniques.* The Morgan Kaufmann Series in Data Management Systems, 2002.
28. Tomáš Skopal, Michal Krátký, Václav Snášel, and Jaroslav Pokorný. A New Range Query Algorithm for the Universal B-trees. *Information Systems*, 31(6):489–511, 2006.
29. J. S. Vitter. *Algorithms and Data Structures for External Memory.* Series on Foundations and Trends in Theoretical Computer Science. Now Publisher, 2008.
30. Cui Yu. *High-Dimensional Indexing*, volume 2341 of *Lecture Notes in Computer Science.* Springer–Verlag, 2002.