12th International Symposium of ACM-GIS (ACM GIS 2004), Washington D.C., U.S.A., 2004.

Object-Relational Management of Complex Geographical Objects

Hans-Peter Kriegel, Peter Kunath, Martin Pfeifle, Matthias Renz
University of Munich
Oettingenstrasse 67
D-80538 Munich, GERMANY
+49-89-21809190
{kriegel, kunath, pfeifle, renz}@dbs.informatik.uni-muenchen.de

ABSTRACT

Modern database applications including computer-aided design, multimedia information systems, medical imaging, molecular biology, or geographical information systems impose new requirements on the effective and efficient management of spatial data. Particular problems arise from the need of high resolutions for large spatial objects and from the design goal to use general purpose database management systems in order to guarantee industrial-strength. In the past two decades, various stand-alone spatial index structures have been proposed but their integration into fully-fledged database systems is problematic. Most of these approaches are based on the decomposition of spatial objects leading to replicating index structures. In contrast to common black-and-white decompositions which suffer from the lack of intermediate solutions, we introduce gray intervals which are stored in a spatial index. Additionally, we store the exact information of these gray intervals in a compressed way. These gray intervals are created by using a cost-based decompositioning algorithm which takes the access probability and the decompression cost of them into account. Furthermore, we exploit statistical information of the database objects to find a cost-optimal decomposition of the query objects. The experimental evaluation on the SEQUOIA benchmark test points out that our new concept outperforms the Relational Interval Tree by more than one order of magnitude with respect to overall query response time.

Categories and Subject Descriptors

H.2.8 [Database applications]: Scientific databases - *Spatial databases and GIS*.

General Terms

Management, Performance.

Keywords

Object Decomposition, Object-Relational Database, Spatial Data Management.

1. INTRODUCTION

The efficient management of rasterized geographical objects has become an enabling technology for many novel database applications. As a common and successful approach, spatial objects can conservatively be approximated by a set of voxels, i.e. cells of a grid covering the complete data space. By means of space filling curves, each voxel (often called pixel in 2D) can be encoded by a single integer and, thus, an extended object is represented by a set of enumerated voxels. These voxels can further be grouped together to intervals, which can be organized by spatial index structures.

By expressing spatial region queries as intersections of these spatial primitives, vital operations for two-dimensional GIS and environmental information systems can be supported. For these applications suitable index structures, which guarantee efficient spatial query processing, are indispensable.

For commercial use, a seamless and capable integration of temporal and spatial indexing into industrial-strength databases is essential. Fortunately, a lot of traditional database servers have evolved into Object-Relational Database Management Systems (ORDBMS). This means that in addition to the efficient and secure management of data ordered under the relational model, these systems now also provide support for data organized under the object model. Object types and other features, such as binary large objects (BLOBs), external procedures, extensible indexing, user-defined aggregate functions and query optimization, can be used to build powerful, reusable server-based components.

An important new requirement for large spatial objects is a high approximation quality which is primarily influenced by the resolution of the grid covering the data space. A promising way to cope with high resolution spatial data may be found somewhere in between replicating and non-replicating spatial index structures. In the case of *replicating access methods*, e.g. the Relational Interval Tree [15], the number of the simple spatial primitives used to approximate the objects can become very high, resulting in a storage and query processing overhead. On the other hand, many of the *non-replicating access methods*, e.g. R-trees [8], use simple spatial primitives such as rectilinear hyper-rectangles for one-value approximations of extended objects. Although providing the minimal storage complexity, one-value approximations of spatially extended objects often are far too coarse. In many GIS applications, objects feature a very complex geometry. A non-replicating storage of such data causes region queries to produce too many false hits that have to be eliminated by subsequent filter steps. For such applications, the accuracy can be improved by decomposing the objects.

1.1 Related Work

In this section, we will shortly discuss different aspects related to an effective decompositioning of complex spatial objects for efficient relational indexing. Often complex objects consist of many line segments. An approved way to describe these objects is to use rasterization.

Complex Spatial Objects. Gaede pointed out that the number of voxels representing a spatially extended object exponentially depends on the granularity of the grid approximation [6]. Furthermore, the extensive analysis given in [17] and [5] shows that the asymptotic redundancy of an interval-based decomposition is proportional to the surface of the approximated object. Thus, in the case of large high-resolution parts, the number of intervals can become unreasonably high.

Relational Spatial Indexing. A wide variety of access methods for spatially extended objects has been published so far. For a general overview on spatial index structures, we refer the reader to the surveys of Manolopoulos, Theodoridis and Tsotras [18] or Gaede and Günther [7]. We use the Relational Interval Tree (RI-tree) in this paper as starting point and comparison partner because it outperforms competing index structures by factors of up to 4.6 (Relational Quadtree [4]) and 58.3 (Relational R-tree [22]) for spatial intersection queries [15].

1.2 Outline

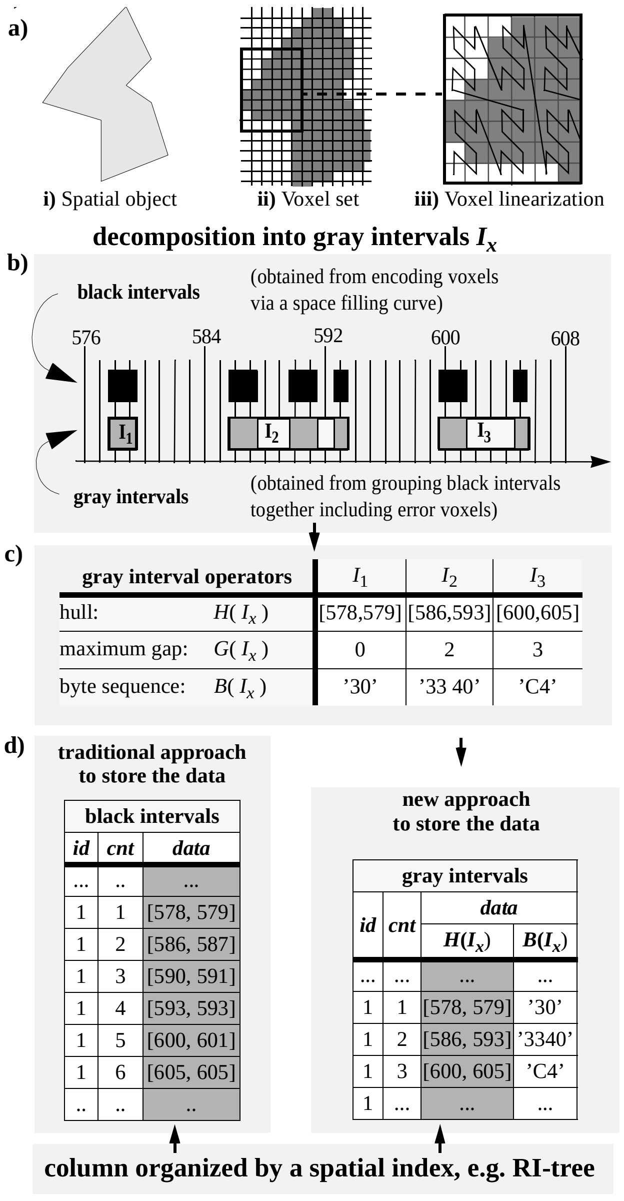
The remainder of this paper is organized as follows. In Section 2, we suggest a pragmatic and effective cost-based decompositioning method for rasterized objects into gray intervals, which can be stored within a spatial index. In Section 3, we discuss the processing of intersection queries on top of an ORDBMS. In Section 4, we present convincing experimental results based on a geographical 2D data set corresponding to the SEQUOIA 2000 benchmark [23]. We resume our work in Section 5 and close with a few final remarks on future work.

2. MANAGEMENT OF GRAY INTERVALS IN AN ORDBMS

Interval sequences, representing high resolution spatially extended objects, often consist of very short intervals connected by short gaps. Following [15], adjacent intervals can be grouped together to longer gray intervals (cf. Figure 1b) in order to improve storage behavior and query response time.

Definition 1 (*gray object interval sequence*)
Let *id* be an *object identifier* and $W = \{(l, u) \in IN^2, l \le u\}$ be the domain of intervals which we call *black* intervals throughout this paper. A black interval (l, u) contains all integers x such that $l \le x \le u$. Furthermore, let $b_1 = (l_1, u_1), \dots, b_n = (l_n, u_n) \in W$ be a sequence of intervals with $u_i + 1 < l_{i+1}$ for all $i \in \{1, \dots, n-1\}$. Moreover, let $m \le n$ and let $i_0, i_1, i_2, \dots, i_m \in IN$ such that $0 = i_0 < i_1 < i_2 < \dots < i_m = n$ holds. Then, we call $O_{gray} = (id, \langle \langle b_{i_0+1},\dots,b_{i_1} \rangle, \langle b_{i_1+1},\dots,b_{i_2} \rangle, \dots, \langle b_{i_{m-1}+1},\dots,b_{i_m} \rangle\rangle)$ a *gray object interval sequence* of cardinality m. If m equals n, we denote O_{gray} also as a *black object interval sequence* O_{black}. We call each of the $j = 1, \dots, m$ groups $\langle b_{i_{j-1}+1},\dots,b_{i_j} \rangle$ of O_{gray} a *gray interval* I_{gray}. If $i_{j-1}+1$ equals i_j, we denote I_{gray} also as a *black interval* I_{black}.

Intuitively, a gray interval is a covering of one or more disjoint and nonadjacent black intervals where there is at least a gap of one integer between adjacent intervals, i.e. it bridges the gap between black intervals. In the next definition, we introduce a few useful operators on gray intervals. In order to clarify these definitions, Figure 1c demonstrates the values of these operators for a sample set of gray intervals. For any gray interval $I_{gray} = \langle (l_r, u_r), \dots, (l_s, u_s) \rangle$ we define the following operators:


a)
i) Spatial object
ii) Voxel set
iii) Voxel linearization
decomposition into gray intervals I_x
b)
black intervals
(obtained from encoding voxels via a space filling curve)
576
584
592
600
608
I_1
I_2
I_3
gray intervals
(obtained from grouping black intervals together including error voxels)
c)
d)
traditional approach to store the data
new approach to store the data
column organized by a spatial index, e.g. RI-tree

gray interval operators		I_1	I_2	I_3
hull:	$H(I_x)$	[578,579]	[586,593]	[600,605]
maximum gap:	$G(I_x)$	0	2	3
byte sequence:	$B(I_x)$	’30’	’33 40’	’C4’

black intervals		
id	*cnt*	*data*
...	..	...
1	1	[578, 579]
1	2	[586, 587]
1	3	[590, 591]
1	4	[593, 593]
1	5	[600, 601]
1	6	[605, 605]
..	..	..

gray intervals			
id	*cnt*	*data*	
		H(I_x)	*B(I_x)*
...	...	...	...
1	1	[578, 579]	’30’
1	2	[586, 593]	’3340’
1	3	[600, 605]	’C4’
1	...	...	...

Figure 1. Gray Intervals.
a) voxelized spatial object, b) black and gray intervals, c) operators on gray intervals, d) storage of gray intervals

Operator	Description and Definition
$L(I_{gray})$	length
$u_s - l_r + 1$	
$H(I_{gray})$	hull
(l_r, u_s)	
$G(I_{gray})$	maximum gap
$\begin{cases} 0 & r = s \\ \max\{l_i - u_{i-1} - 1, i = r+1, \dots, s\} & \text{else} \end{cases}$	
$B(I_{gray})$	byte sequence
$\langle s_0, .., s_n \rangle$, where $s_i \in IN$ and $0 \le s_i < 2^8$, $n = \lfloor u/8 \rfloor - \lfloor l/8 \rfloor$	
$s_i = \sum_{k=0} \begin{cases} 2^{7-k} & \text{if } \exists (l_t, u_t): l_t \le \lfloor l_r/8 \rfloor \cdot 8 + 8i + k \le u_t, r \le t \le s \\ 0 & \text{otherwise} \end{cases}$	

We use $B(I_{gray})$ as an abbreviation for a byte sequence containing the complete information of the black intervals which have been grouped together to I_{gray}.

The gray interval sequence $I_{gray} = (id, \langle I_1, ..., I_m \rangle)$ is stored in a set of m tuples in an object-relational table *GrayIntervals* (<u>*id*</u>, <u>*cnt*</u>, *data*). The primary key is formed by the object identifier *id* and a unique number *cnt* for each gray interval. The black intervals of each gray interval $I_{gray} = \langle (l_r, u_r), \ldots, (l_s, u_s) \rangle$ are mapped to the complex attribute *data* which consists of aggregated information, i.e. the hull $H(I_{gray})$ and a *BLOB* containing the complete information of the black intervals. In order to guarantee efficient query processing, we apply spatial index structures on $H(I_{gray})$ and store $B(I_{gray})$ in a compressed way within a BLOB.

There are two different problems related to the storage of gray interval sequences: the *compression* problem and the *grouping* problem.

2.1 Compression

The detailed black interval sequence $b_r, \ldots, b_s$ of a gray interval $I_{gray} = \langle b_r, \ldots, b_s \rangle$ can be materialized and stored in a BLOB in many different ways. A good materialization for $I_{gray} = \langle b_r, \ldots, b_s \rangle$ should consider two aspects:

- As little as possible secondary storage should be occupied.
- As little as possible time should be needed for the (de)compression of the BLOB.

A good query response behavior is based on the fulfillment of both aspects. The first rule guarantees that the I/O cost $t_{\mathrm{I/O}}^{BLOB}$ are relatively small whereas the second rule is responsible for low CPU cost t_{CPU}^{BLOB}. The overall time $t^{BLOB} = t_{\mathrm{I/O}}^{BLOB} + t_{\mathrm{CPU}}^{BLOB}$ for the evaluation of a BLOB is composed of both parts. Unfortunately, these two requirements are not necessarily in accordance with each other. If we compress $B(I_{gray})$, we can reduce the demand of secondary storage and consequently $t_{\mathrm{I/O}}^{BLOB}$. The CPU cost t_{CPU}^{BLOB} might rise because we first have to decompress the data before we can evaluate it. On the other hand, if we store $B(I_{gray})$ without compressing it, $t_{\mathrm{I/O}}^{BLOB}$ might become very high whereas t_{CPU}^{BLOB} might be low.

As we will show in our experiments, it is very important for a good query response behavior to find a well-balanced way between these two compression rules.

There exist many different data compression techniques. *ZLIB* is an example for a dictionary based packer [16], *HSC* is an implementation of an arithmetic coder [26]. For a detailed survey on lossless and lossy compression techniques, we refer the reader to [21] and [25].

2.2 Grouping into Gray Intervals

High resolution spatial objects may consist of several hundreds of thousands of black intervals (cf. Figure 1a). For each object, there exist a lot of different possibilities to decompose it into approximations by grouping numerous black intervals together. The question at issue is, which grouping is most suitable for efficient query processing. A good grouping should take the following requirements into consideration:

- The number of gray intervals should be small.
- The dead area of all gray intervals should be small.
- The gray intervals should allow an efficient evaluation of the contained black intervals.

The first rule guarantees that the number of index entries is small, as the hulls of the gray intervals are stored in appropriate index structures, e.g. the RI-tree (cf. Figure 1d). The second rule guarantees that many unnecessary candidate tests can be omitted, as the number and size of gaps included in the gray intervals is small. Finally, the third rule guarantees that a candidate test can be carried out efficiently. A good query response behavior results from an optimum trade-off between these grouping rules.

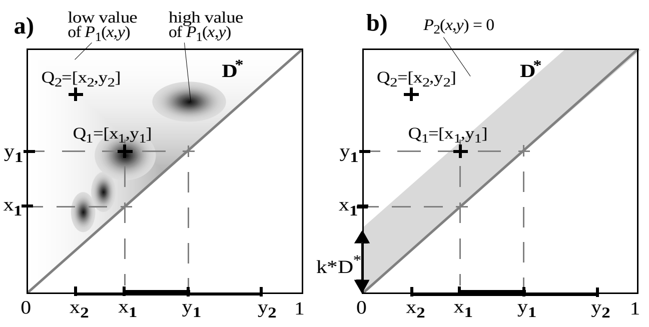


Figure 2. Query distribution functions $P_i(x,y)$.
a) Complex query distribution $P_1(x,y)$,
b) Simple query distribution $P_2(x,y)$

Our grouping algorithm takes the expected access cost of the gray intervals into account. The expected cost $cost(I_{gray})$ related to a gray interval I_{gray} depend on the average access probability of I_{gray} and on the cost related to the evaluation of the exact byte sequence $B(I_{gray})$.

2.2.1 Access Probability

Our grouping algorithm takes the expected access cost of the gray intervals into account. The expected cost $cost(I_{gray})$ related to a gray interval I_{gray} depend on the average access probability of I_{gray} and on the cost related to the evaluation of the exact byte sequence $B(I_{gray})$.

First, the access probability is computed by assuming that we know the average query distribution. Then, the evaluation cost are introduced which heavily depend on the used data compressor. Finally, our cost-based grouping algorithm *GroupObj* is introduced which is used for storing complex objects in an ORDBMS.

Assumed Query Distribution. For many application areas, e.g. in the field of GIS, the average query distribution can be predicted very well. It is obvious that queries inquiring rather dense areas, e.g. big cities like New York, occur much more frequently than for less dense areas. Furthermore, often small selective queries are posted.

For determining a suitable query distribution function, we first transform the potential query intervals into the upper triangle $D^* := \{(x, y) \in [0, 1]^2 \mid x \leq y\}$ of the two-dimensional hyper cuboid. An interval $Q = [x, y]$ therefore corresponds to the point (x, y) with $x \leq y$. Examples are visualized in Figure 2. To each of these two-dimensional points $Q=(x,y)$ we assign a numerical value $P(Q)$ where $0 \leq P(Q) \leq 1$ holds. As the probability is equal to one that a query is somewhere located in the upper triangle D^*, the following equation has to hold:

$$\iint_{D^*} P(x, y)\,dx\,dy = 1$$

Figure 2 shows two different query distribution functions. For instance, a potential query Q_2 is very unlikely according to the depicted query distribution of Figure 2a, and impossible according to the query distribution of Figure 2b. On the other hand, query Q_1 is very likely in both cases.

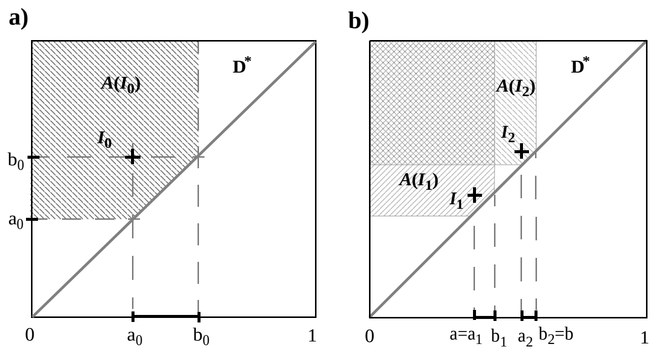


Figure 3. Average access probabilities of gray intervals. a) Intersection area for a gray interval I_0=[a_0,b_0], b) Intersection area for the decomposed gray intervals I_1 and I_2

Let us note, that we used the simple query distribution function of Figure 2b throughout our experiments. In all considered application areas the common query objects only comprise a very small portion of the data space D*. Therefore, we introduce the parameter k*, which restricts the extension of the possible query objects. For the computation of the access probability we only consider query objects whose extensions do not exceed $k^* \cdot D^*$.

Access Probability. The *access probability* $P(I_{gray})$ related to a gray interval I_{gray} denotes the probability that an arbitrary query interval has an intersection with the hull $H(I_{gray})$. All possible query intervals that intersect I_0 are visualized by the shaded area $A(I_0)$ in Figure 3a. The area displays all intervals whose lower bounds are smaller or equal to b and whose upper bounds are larger or equal to a. These query intervals are exactly the ones that have a non empty intersection with I_0. The probability that an interval I_0 = [a_0, b_0] is intersected by an arbitrary query interval is:

$$P(I_0) = \frac{\iint\limits_{A(I_0)} P(x,y)dxdy}{\int\int\limits_{D^*} P(x,y)dxdy} = \iint\limits_{A(I_0)} P(x,y)dxdy$$

Evaluation Cost. Furthermore, the expected query cost depend on the cost related to the evaluation of the byte sequence stored in the BLOB of an intersected gray interval I_{gray}. The evaluation of the BLOB content requires to load the BLOB from disk and decompress the data. Consequently, the evaluation cost depends on both the length $L(I_{gray})$ of the uncompressed BLOB and the length $L_{comp}(I_{gray}) << L(I_{gray})$ of the compressed data. Additional, the evaluation cost $cost_{eval}$ depend on a constant $c_{load}^{I/O}$ related to the retrieval of the BLOB from secondary storage, a constant c_{decomp}^{cpu} related to the decompression of the *BLOB*, and a constant c_{test}^{cpu} related to the intersection test. The cost c_{decomp}^{cpu} and $c_{load}^{I/O}$ heavily depend on how we organize $B(I_{gray})$ within our BLOB, i.e. on the used compression algorithm. A highly effective but not very time efficient packer, e.g. an arithmetic packer, would cause low loading cost but high decompression cost. In contrast, using no compression technique, leads to very high loading cost but no decompression cost. On the other hand, *ZLIB* is an effective and very efficient compression algorithm which yields a good trade-off between the loading and decompression cost. Finally, c_{test}^{cpu} solely depend on the

```
ALGORITHM GroupObj ($I_{gray}$, $P$)
BEGIN
    interval_pair := split_at_maximum_gap($I_{gray}$);
    $I_{left}$     := interval_pair.left;
    $I_{right}$    := interval_pair.right;
    $cost_{gray}$  := $P(I_{gray}) \cdot cost_{eval}(I_{gray})$;
    $cost_{dec}$   := $P(I_{left}) \cdot cost_{eval}(I_{left})$+
                     $P(I_{right}) \cdot cost_{eval}(I_{right})$;
    IF $cost_{gray} > cost_{dec}$ THEN
            GroupObj ($I_{left}$, $P$);
            GroupObj ($I_{right}$, $P$);
    ELSE
            report ($I_{gray}$);
    END IF;
END.
```

Figure 4. Grouping algorithm GroupObj.

used system. The overall evaluation cost are defined by the following formula:

$$cost_{eval}(I_{gray}) =$$

$$L_{comp}(I_{gray}) \cdot c_{load}^{I/O} + L(I_{gray}) \cdot (c_{decomp}^{cpu} + c_{test}^{cpu})$$

Grouping Algorithm. Orenstein [Ore 89] introduced the size- and error bound decomposition approach. Our first grouping rule “the number of gray intervals should be small” can be met by applying the size-bound approach, while applying the error-bound approach results in the second rule “the dead area of all gray intervals should be small”. For fulfilling both rules, we introduce the following top-down grouping algorithm for gray intervals, called *GroupObj* (cf. Figure 4). *GroupObj* is a recursive algorithm which starts with an approximation $O_{gray} = (id, \langle I_{gray} \rangle)$, i.e. we approximate the object by one gray interval. In each step of our algorithm, we look for the maximum gap g within the actual gray interval. We carry out the split along this gap, if the average query cost caused by the decomposed intervals is smaller than the cost caused by our input interval I_{gray}. The expected cost related to a gray interval I_{gray} can be computed as described in the foregoing paragraph. A gray interval which is reported by the *GroupObj* algorithm is stored in the database and no longer taken into account in the next recursion step. Data compressors which have a high compression rate and a fast decompression method, result in an early stop of the *GroupObj* algorithm generating a small number of gray intervals. Our experimental evaluations suggest that this grouping algorithm yields results which are very close to the optimal ones for different data compression techniques and data space resolutions.

3. QUERY PROCESSING

In this section, we will discuss how we can efficiently carry out intersection queries on top of the SQL-engine. Our approach uses the RI-tree for efficiently detecting intersecting interval hulls. In contrast to the last section, we do not assume any arbitrary data distribution but use statistical information reflecting the distribution of the gray interval hulls managed by the RI-tree.

The algorithm for decomposing a query object is basically the same as the one presented in Figure 4. The main difference is that the assumed intersection probability P, is replaced by a more accurate selectivity estimation σ reflecting the actual data distribution.

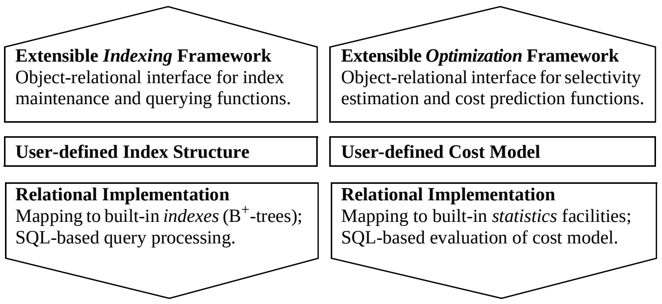

Extensible *Indexing* Framework
Object-relational interface for index maintenance and querying functions.
Extensible *Optimization* Framework
Object-relational interface for selectivity estimation and cost prediction functions.
User-defined Index Structure
User-defined Cost Model
Relational Implementation
Mapping to built-in *indexes* (B+-trees); SQL-based query processing.
Relational Implementation
Mapping to built-in *statistics* facilities; SQL-based evaluation of cost model.

Figure 5. Analogous architectures for the object-relational embedding of user-defined index structures and cost models into extensible indexing and optimization frameworks.

In [13] it was shown how we can effectively and efficiently estimate the selectivity of interval intersection queries based on the RI-tree. Both index structure and the corresponding cost-model can be integrated into modern object relational database systems by using their extensible indexing and optimization frameworks (cf. Figure 5). By exploiting the statistical information provided by the cost model, we can find an optimum decomposition for the query object. The traditional error-and size bound decomposition approaches [20] decompose a large query object into smaller query objects optimizing the trade off between accuracy and redundancy. In contrast, the idea of taking the actual data distribution into account in order to decompose the query object, leads to a new *selectivity-bound decomposition approach*, which tries to minimize the overall number of logical reads.

3.1 Decomposition of the Query Object

In this section, we shortly sketch the decompositioning of the query object into suitable gray object interval sequences. Thereby two different cases have to be distinguished. First, the query object is already stored in a decomposed way in the database. Second, it has to be decomposed in real-time from scratch.

Query object is a database object. In this case, the query object is already stored in a decomposed way within the database according to our *GroupObj* algorithm which assumes a potential query distribution P (cf. Figure 2). Usually areas which are often inquired also contain a lot more objects than seldomly inquired areas. For instance, New York contains a lot of geographical objects and is much more often inquired than Alaska. As we assume that the characteristic P coincides with the data distribution σ of the actual databases objects, we use the already decomposed objects as starting point for a further statistic-driven generation of the gray intervals instead of starting from scratch. In order to carry-out a fine-tuning of our gray query-sequence we test whether two already decomposed gray intervals should be merged to one gray interval. This is beneficial if $\sigma < \mathrm{P}$ holds for the query region. On the other hand, if $\sigma > \mathrm{P}$ holds, we further decompose the gray query interval according to the algorithm presented in Figure 4. Thereby, we do not assume a potential query distribution P, but replace P by the actual selectivity estimation σ w.r.t. the corresponding gray interval. We carry out the selectivity estimation for I_{gray} and for the two potentially new gray intervals I_{left} and I_{right} which result from a split at the maximum gap of I_{gray}. Based on the computed cost, we decide whether we actually carry out the split.

Query object is no database object. Second, if the query object has to be decomposed from scratch, we carry out a decompositioning starting with one gray interval conservatively approximating the query object. Similar to the approach of the last paragraph, we decide based on an accurate selectivity estimation whether to further decompose the query object. This approach is quite feasible if the query object is already available as rasterized object. If not, which is the case for many query objects used in GIS, they are often very simple and can be described by a few parameters, e.g. a rectilinear box can be described by two points. Again, we approximate the query object by one gray interval and apply the already sketched decompositioning approach for query objects. The few parameters which are necessary to describe the query object can be stored in the BLOB of each gray interval instead of the exact black interval sequence. These few values contain the whole information in the most compressed way. In the blobintersection routines of Figure 6, $B(I_{gray})$ is created on demand from this simple and compact geometric information. As the geometric information is already in the most compressed form we do not need a special data compressor, but only a decompression algorithm specific to the geometry of the query object.

In the following, we will show how we can carry out an interval intersection query on top of an ORDBMS.

3.2 Intersection Query

In this section, we discuss the query processing of a boolean and a ranked intersection predicate on top of the SQL-engine. The presented approaches can easily be embedded by means of extensible indexing interfaces (cf. Figure 5) into modern ORDBMS. Most ORDBMSs, including Oracle [19] [24], IBM DB2 [10] [2] or Informix IDS/UDO [11] [1], provide these extensibility interfaces in order to enable database developers to seamlessly integrate custom object types and predicates within the declarative DDL and DML.

As we represent spatial objects by gray object interval sequences, we first clarify when two of these sequences intersect.

Definition 1 (object intersection)

Let $W_{black} = \{(l, u) \in IN^2, l \le u\}$ be the domain of black intervals and let $b_1 = (l_1, u_1)$ and $b_2 = (l_2, u_2)$ be two black intervals. Further, let $I^1 = \langle b_1^1, \ldots, b_{n_1}^1 \rangle$ and $I^2 = \langle b_1^2, \ldots, b_{n_2}^2 \rangle$ be two gray intervals, and let $O^1 = (id^1, \langle I_1^1, I_2^1, \ldots, I_{m_1}^1 \rangle)$ and $O^2 = (id^2, \langle I_1^2, I_2^2, \ldots, I_{m_2}^2 \rangle)$ be two gray object interval sequences. Then, the notions *intersect*, *Xintersect* and *interlace* are defined in the following way:

1a. Two black intervals b_1 and b_2 *intersect* if $l_1 \le u_2$ and $l_2 \le u_1$.

1b. $Xintersect(b_1,b_2) = \max \{ 0, min\{u_1, u_2\} - max\{l_1, l_2\} + 1 \}$.

2a. Two gray intervals I^1 and I^2 *intersect* if for any $i \in \{1, \ldots, n_1\}$, $j \in \{1, \ldots, n_2\}$, the black intervals b_i^1 and b_j^2 intersect.

2b. $Xintersect(I^1, I^2) = \sum_{i = 1 \ldots n_1, j = 1 \ldots n_2} \text{Xintersect}\ (b_i, b_j)$.

2c. Two gray intervals I^1 and I^2 *interlace*, if their hulls $H(I^1)$ and $H(I^2)$ intersect.

3a. Two objects O^1 and O^2 *intersect*, if for any $i \in \{1, \ldots, m_1\}$, $j \in \{1, \ldots, m_2\}$, the gray intervals I_i^1 and I_j^2 intersect.

3b. $Xintersect(O^1, O^2) = \sum_{i = 1 \ldots m_1, (j = 1 \ldots m_2)} \text{Xintersect}\ (I_i^1, I_j^2)$.

3c. Two objects O^1 and O^2 *interlace*, if for any $i \in \{1, \ldots, m_1\}$, $j \in \{1, \ldots, m_2\}$, two gray intervals I_i^1 and I_j^2 interlace.

3.2.1 The intersect SQL Statements

In [14] many index structures for interval intersection queries were surveyed, which can be integrated into the extensible indexing framework of modern ORDBMSs. These index structures also support the evaluation of the *interlace* predicate on gray intervals. As we defined gray object interval sequences as a conservative approximation of black object interval sequences, we can use the hulls of

a)

```
SELECT candidates.id FROM
(    SELECT db.id AS id, table (pair(db.rowid, q.rowid)) AS ctable
     FROM GrayIntervals db, :GrayQueryIntervals q
     WHERE intersects (hull(db.data), hull(q.data))
     GROUP BY db.id
)    candidates
WHERE EXISTS
(    SELECT 1
     FROM GrayIntervals db, :GrayQueryIntervals q, candidates.ctable ctable
     WHERE db.rowid = ctable.dbrowid AND q.rowid = ctable.qrowid AND
     blobintersection (db.data, q.data)
)
```

b)

```
SELECT db.id, Xblobintersection(db.data, q.data)
FROM GrayIntervals db, :GrayQueryIntervals q
WHERE intersects (hull(db.data), hull(q.data))
group by db.id
```

Figure 6. SQL statements for spatial object intersection, based on gray object interval sequences. a) *intersect*-predicate, b) *Xintersect*-predicate

the gray intervals in a first conservative filter step. Thereby, we can take advantage of the same access methods as used for the detection of intersecting black interval pairs. As shown in Section 2, the *gray object interval sequences* can be mapped to an object-relational schema *GrayIntervals*. Following this approach, we can also clearly express the intersect predicates on top of the SQL engine (cf. Figure 6).

In the case of the *intersect*-predicate (cf. Figure 6a), the nesting function *table* groups references of interlacing gray query and database interval pairs together. The NF2-operator *table* was realized by a user-defined aggregate function as provided in the SQL:1999 standard. In order to find out which database objects are intersected by a specific query object, the interlacing gray intervals have to be tested for intersection. This test is carried out by a stored procedure *blobintersection*. If one intersecting gray database and query interval pair is found, no gray interlacing interval pairs belonging to the same database object have to be examined. This *skipping principle* is realized by means of the *exists*-clause within the SQL-statement.

In the case of the *Xintersect*-predicate (cf. Figure 6b), the intersection volume has to be determined for each interlacing interval pair. No BLOB tests can be skipped. The results are summed up in the user-defined aggregate function *Xblobintersection*.

In both blobintersection routines, we first decompress the data and then test the two byte sequences in the interlacing area for intersection. As already mentioned in Section 2.1 it is important that the compressed BLOB size is small in order to reduce the I/O cost. Obviously, the small I/O cost should not be at the expense of the CPU cost. Therefore, it is important that we have a fast decompressing algorithm in order to evaluate the BLOBs quickly.

4. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of our approach. We evaluate two different grouping algorithms *GRP* in combination with three data compression techniques *DC*. We used the following data compressors *DC*:

***NOOPT*:** The BLOB is unpacked.

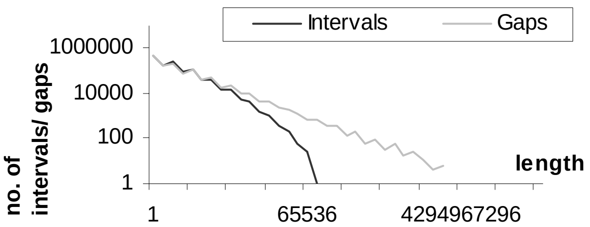


Figure 7. Histograms for intervals and gaps.

***ZLIB*:** The BLOB is packed according to the ZLIB approach [3].

***HSC*:** The BLOB is packed according to the arithmetic approach of [9].

Furthermore, we grouped the voxels into gray intervals depending on two grouping algorithms, called *Maxgap* and *GroupObj*.

MaxGap. This grouping algorithm tries to minimize the number of gray intervals while not allowing that a maximum gap $G(I_{gray})$ of any gray interval I_{gray} exceeds a given *MAXGAP* parameter. By varying this *MAXGAP* parameter, we can find the optimum trade-off between the first two opposing grouping rules of Section 2.2, namely a small number of gray intervals and a small number of white cells included in each of these intervals.

One of the main goals of this experimental evaluation is to show that our new *GroupObj* algorithm analytically finds this empirically derived optimum of the *Maxgap*-approach.

GroupObj. We grouped the intervals according to our cost-based grouping algorithm *GroupObj* (cf. Section 2.2), where we used the query distribution function from Figure 2b with $k^* = 1/100{,}000$.

Note, that the grouping based on *MaxGap*(*DC*) does not depend on *DC*, whereas *GroupObj*(*DC*) takes the actual data compressor *DC* into account for performing the grouping.

In order to support the first filter step of *GRP*(*DC*), we used the RI-tree. We have implemented the RI-tree [14, 15] on top of the Oracle9*i* Server using PL/SQL for most of the computational main memory based programming. The evaluation of the blobintersection routines was delegated to a DLL written in C. All experiments were performed on a Pentium 4/2600 machine with IDE hard drives. The database block cache was set to 500 disk blocks with a block size of 8 KB and was used exclusively by one active session.

Test Data Set. The tests are based on a subset of 2D GIS data representing woodlands, rivers, and transportation networks derived from the *SEQUOIA 2000* benchmark [23], for simplicity called *SEQUOIA* throughout this section. It contains about 3500 rasterized polygons approximated by $50 \cdot 10^6$ voxels. The *SEQUOIA* data space is of size 2^{34}. The Z-curve was used as a space filling curve to enumerate the voxels. Figure 7 depicts the interval and gap histograms for our *SEQUOIA* test data set. The figure shows that the 3500 geographical objects consist of many short black intervals and short gaps and only a few longer ones.

Storage Requirements. Figure 8 shows the different storage requirements for the BLOB with respect to the different data compression techniques. The figure shows clearly that the *HSC*-approach yields the best compression ratios for all *MAXGAP* parameters. For medium to high *MAXGAP* parameters the *ZLIB*-approach also yields compression ratios of almost two orders of magnitude. If we do not apply any compression, the required secondary storage for high *MAXGAP* values it is very high leading to high I/O cost during the query process.

Update Operations. In this section, we will investigate the time needed for updating complex spatial objects in the database. Figure 9a shows that inserting all objects into the database takes very long if we store the numerous black intervals in the RI-tree (*i*) or if we

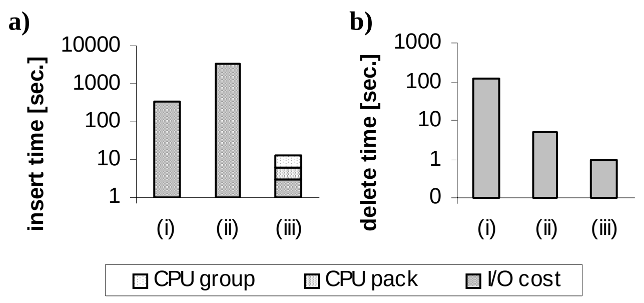


Figure 9. Update operations.
(*i*) numerous black intervals (*ii*) one gray interval
(*iii*) gray intervals grouped by *GroupObj*(*ZLIB*)
a) insert-operation, b) delete-operation

store the huge one value approximations of the unpacked object in the RI-tree (*ii*). On the other hand, using our *GroupObj*(*ZLIB*) approach (*iii*) accelerates the insert operations by one to two orders of magnitude. The time spent for grouping and packing pays off, if we take into consideration that we save a lot of time for storing grouped and packed intervals in the database. Obviously, the delete operations are also carried out much faster for our *GroupObj*(*ZLIB*) approach as we have to delete much less disk blocks (cf. Figure 9b).

Query Processing. In this section, we want to turn our attention to the query processing by examining different kinds of *collision queries.* The figures presented in this paragraph depict the average result obtained from collision queries where we have taken the 100 largest parts from our *SEQUOIA* data set as query objects.

In Figure 10 it is shown in which way the overall response time for *boolean intersection* queries based on the RI-tree depends on the

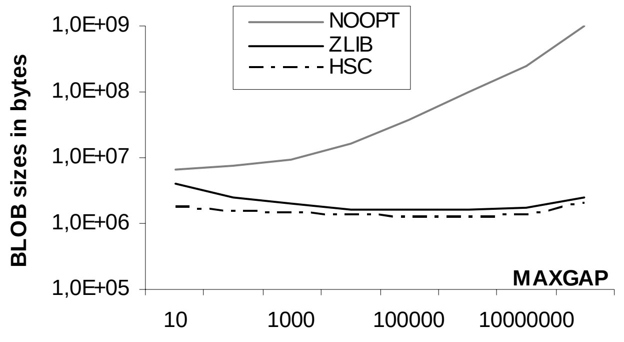


Figure 8. Storage requirements for the BLOBs.

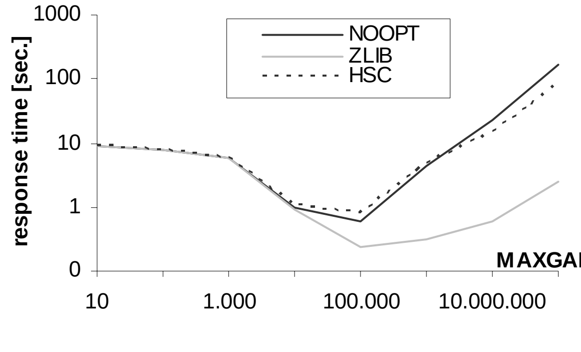


Figure 10. MaxGap(*DC*) for boolean intersection queries.

MAXGAP parameter. If we use small *MAXGAP* parameters, we need a lot of time for the first filter step whereas the *blobintersection* test is relatively cheap. Therefore, the different *MaxGap*(*DC*) approaches do not differ very much for small *MAXGAP* values. For high *MAXGAP* values we can see that the *MaxGap*(*ZLIB*) approach performs best with respect to the overall runtime. The *MaxGap*(*ZLIB*) approach is rather insensitive against too large *MAXGAP* parameters. Even for values where the first filter step is almost irrelevant, e.g. *MAXGAP* = 10^8, the *MaxGap*(*ZLIB*) approach still performs well. This is due to the fact that for large *MAXGAP* values the *MaxGap*(*ZLIB*) approach needs much less physical reads, about 1% of the *MaxGap*(*NOOPT*) approach. As a consequence, the query response time of the *MaxGap*(*ZLIB*) approach is approximately $^1/_{67}$ of the query response time of the *MaxGap*(*NOOPT*) approach.
In Figure 11 it is shown in what way the different data space resolutions influence the query response time. Generally, the higher the resolution, the slower is the query processing. Our *MaxGap*(*ZLIB*) is especially suitable for high resolutions, but also accelerates medium or low resolution spatial data.
To sum up, the *MaxGap*(*ZLIB*) approach improves the response time of collision queries for varying index structures and resolutions by more than one order of magnitude.
We will now show that the *GroupObj* algorithm analytically finds this empirically derived optimum of the *Maxgap*-approach, i.e. our *GroupObj* approach yields almost optimum query response times for varying compression techniques and data space resolutions.
Table 1 depicts the overall query response time of our *GroupObj*(*DC*) approach compared to the RI-tree for boolean and ranking intersection queries.

	gray intervals			black intervals
	NOOPT	ZLIB	HSC	RI-tree [15]*[14]**
number of intervals	7,285	5,206	8,934	1,188,356
Overall Runtime* [s]	0.68	0.29	0.91	12.74
Overall Runtime** [s]	1.20	0.52	1.53	∞ (not applicable)

Table 1. *GroupObj* (*DC*) evaluated for *boolean and *ranking*** intersection queries.**

We can see that for boolean intersection queries this grouping delivers results quite close to the minimum response times depicted in Figure 10. Furthermore, we notice that the *GroupObj*(*ZLIB*) approach outperforms the RI-tree [15] by a factor of 44 for boolean intersection queries. For ranking intersection queries the RI-tree [14] is not applicable due to the enormous amount of generated join partners. On the other hand, the *GroupObj*(*ZLIB*) approach yields interactive response times even for such queries.
In Table 2 it is shown that the query response times resulting from the *GroupObj* algorithm for varying resolutions, are almost identi-

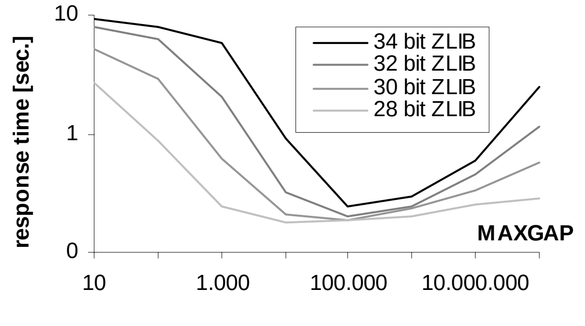


Figure 11. MaxGap(*ZLIB*) evaluated for boolean intersection queries using different resolutions.

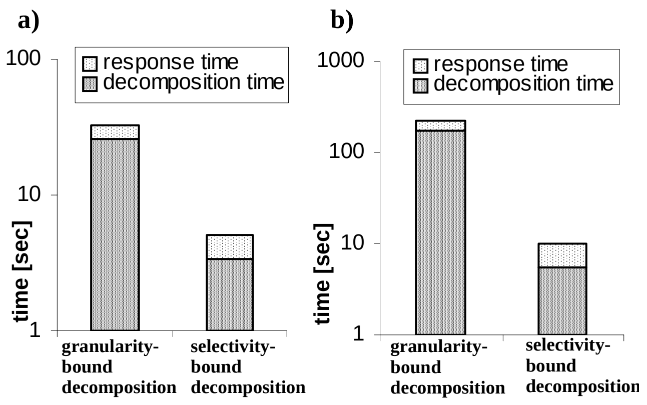


Figure 12. Window queries (decomposition and response time) a) window size equals 0.00001% of data space yielding approximately 0.01% selectivity b) window size equals 0.01% of data space yielding approximately 1.0% selectivity

cal to the ones resulting from a grouping based on an optimum *MAXGAP* parameter (cf. Figure 11).

	34 bit	32 bit	30 bit	28 bit
Overall Runtime [s]	0.29	0.24	0.21	0.19

Table 2. *GroupObj*(*ZLIB*) evaluated for boolean intersection queries for the *RI-tree* with different resolutions.

To sum up, the *GroupObj* algorithm produces object decompositions which yield almost optimum query response times for varying compression techniques and data space resolutions.

Window Queries. In a last experiment, we carried out different window queries. Figure 12 depicts the average runtime for window queries, where we moved each window to 10 different locations. As shown in Figure 12, our statistic-based decomposition approach of the query object can improve the query response behavior by more than one order of magnitude, compared to the granularity-bound decompositioning approach where we decompose the query objects into black intervals. This speed up is mainly due to the reduced decomposition time resulting from the fact that we do not decompose the gray intervals completely into black intervals, but take the actual data distribution into account to guide the decompositioning process. If our selectivity estimation indicates low selectivity for the actual gray interval, we do not further decompose the gray interval but use it directly as query interval, where the actual window coordinates are stored in the corresponding BLOB. The experiments show that the query response time does not suffer from the fact that we did not decompose the windows with the maximum possible accuracy. The time we need for the additional refinement step to filter out false hits is compensated by the much smaller number of query intervals resulting from a coarser decomposition of the query window. To sum up, our statistic-based decomposition approach is especially useful for commonly used window queries.

5. CONCLUSION

In this paper, we introduced a new approach for accelerating spatial query processing for complex geographical objects. We used gray intervals and showed how we can efficiently store them by means of data compression techniques within an ORDBMS. Furthermore, we introduced a cost-based decompositioning algorithm for complex rasterized objects, called *GroupObj*. *GroupObj* takes the decompression cost of the gray intervals and their access probabilities into account. We showed how to use the gray intervals generated by *GroupObj* for efficient spatial query processing on top of an ORDBMS. The resulting spatial access method is applicable for different data space resolutions and compression algorithms. We demonstrated in a broad experimental evaluation that our new approach accelerates spatial query processing on complex rasterized geographical objects by more than one order of magnitude.

In our future work, we will show that our new approach can also be applied to accelerate other index structures, e.g. the Relational R-tree [22] or the Relational Quadtree [4].

6. REFERENCES

[1] Bliujute R., Saltenis S., Slivinskas G., Jensen C. S.: Developing a DataBlade for a New Index. Proc. 15th Int. Conf. on Data Engineering (ICDE), 314-323, 1999.

[2] Chen W., Chow J.-H., Fuh Y.-C., Grandbois J., Jou M., Mattos N., Tran B., Wang Y.: High Level Indexing of User-Defined Types. Proc. 25th Int. Conf. on Very Large Databases (VLDB), 554-564, 1999.

[3] Deutsch P.: RFC1951, DEFLATE Compressed Data Format Specification. http://rfc.net/rfc1951.html, 1996. Int. Conf. on Data Engineering (ICDE), 91-100, 2000.

[4] Freytag J.-C., Flasza M., Stillger M.: Implementing Geospatial Operations in an Object-Relational Database System. Proc. 12th Int. Conf. on Scientific and Statistical Database Management (SSDBM): 209-219, 2000.

[5] Faloutsos C., Jagadish H. V., Manolopoulos Y.: Analysis of the n-Dimensional Quadtree Decomposition for Arbitrary Hyperrectangles. IEEE TKDE 9(3): 373-383, 1997.

[6] Gaede V.: Optimal Redundancy in Spatial Database Systems. Proc. 4th Int. Symp. on Large Spatial Databases (SSD), LNCS 951: 96-116, 1995.

[7] Gaede V., Günther O.: Multidimensional Access Methods. ACM Computing Surveys 30(2): 170-231, 1998.

[8] Guttman A.: R-trees: A Dynamic Index Structure for Spatial Searching. Proc. ACM SIGMOD Int. Conf. on Management of Data, 47-57, 1984.

[9] Hirvola H.: HA archiver source code, http://sunsite.unc.edu/pub/Linux/utils/compress/ha0999p-linux.tar.gz, 1995.

[10] IBM Corp.: IBM DB2 Universal Database Application Development Guide, Version 6. Armonk, NY, 1999.

[11] Informix Software, Inc.: DataBlade Developers Kit User's Guide, Version 3.4. Menlo Park, CA, 1998.

[12] Kriegel H.-P., Kunath P., Pfeifle M., Renz M.: Acceleration of Relational Index Structures Based on Statistics. Proc. 15th Int. Conf. on Scientific and Statistical Database Management (SSDBM), 2003

[13] Kriegel H.-P., Pfeifle M., Pötke M., Seidl T.: A Cost Model for Interval Intersection Queries on RI-Trees, Proc. 14th Int. Conf. on Scientific and Statistical Database Management (SSDBM), Edinburgh, Scotland, 2002, pp. 131-141.

[14] Kriegel H.-P., Pötke M., Seidl T.: Managing Intervals Efficiently in Object-Relational Databases. Proc. 26th Int. Conf. on Very Large Databases (VLDB), 407-418, 2000.

[15] Kriegel H.-P., Pötke M., Seidl T.: Interval Sequences: An Object-Relational Approach to Manage Spatial and Temporal Data. Proc. 7th Int. Symposium on Spatial and Temporal Databases (SSTD), LNCS 2121: 481-501, 2001.

[16] Lempel A., Ziv J.: A Universal Algorithm for Sequential Data Compression. IEEE Transactions on Information Theory, Vol. IT-23, No. 3, 337-343, 1977.

[17] Moon B., Jagadish H. V., Faloutsos C., Saltz J. H.: Analysis of the Clustering Properties of Hilbert Space-filling Curve. Tech. Rep. CS-TR-3611, University of Maryland, 1996.

[18] Manolopoulos Y., Theodoridis Y., Tsotras V. J.: Advanced Database Indexing. Boston, MA: Kluwer, 2000.

[19] Oracle Corp.: Oracle8i Data Cartridge Developer's Guide, Release 2 (8.1.6). Redwood Shores, CA, 1999.

[20] Orenstein J. A.: Redundancy in Spatial Databases. Proc. ACM SIGMOD Int. Conf. on Management of Data, 294-305, 1989.

[21] Roth M., Van Horn S.: Database Compression. SIGMOD Record 22(3): 31-39, 1993

[22] Ravi Kanth K. V., Ravada S., Sharma J., Banerjee J.: Indexing Medium-dimensionality Data in Oracle. Proc. ACM SIGMOD Int. Conf. on Management of Data: 521-522, 1999.

[23] Stonebraker M., Frew J., Gardels K., Meredith J.: The SEQUOIA 2000 Storage Benchmark. In Proc. ACM SIGMOD Int. Conf. on Management of Data: 1993

[24] Srinivasan J., Murthy R., Sundara S., Agarwal N., DeFazio S.: Extensible Indexing: A Framework for Integrating Domain-Specific Indexing Schemes into Oracle8i. Proc. 16th Int. Conf. on Data Engineering (ICDE), 91-100, 2000.

[25] Steinmetz R., Nahrstedt K.: Multimedia Fundamentals, Volume 1: Media Coding and Content Processing, Second Edition. Prentice Hall, 110-119, 2002.

[26] Witten I., Neal R., Cleary J.: Arithmetic coding for data compression, Communications of the ACM 30(6), 1987, 520 - 540.