Effective Decompositioning of Complex Spatial Objects into Intervals

Hans-Peter Kriegel, Peter Kunath, Martin Pfeifle, Matthias Renz
University of Munich, Germany, {kriegel, kunath, pfeifle, renz}@dbs.informatik.uni-muenchen.de

ABSTRACT

In order to guarantee efficient query processing together with industrial strength, spatial index structures have to be integrated into fully-fledged object-relational database management systems (ORDBMSs). A promising way to cope with spatial data can be found somewhere in between replicating and non-replicating spatial index structures. In this paper, we use the concept of gray intervals which helps to range between these two extremes. Based on the gray intervals, we introduce a cost-based decomposition method for accelerating the Relational Interval Tree (RI-tree). Our approach uses compression algorithms for the effective storage of the decomposed spatial objects. The experimental evaluation on real-world test data points out that our new concept outperforms the RI-tree by up to two orders of magnitude with respect to overall query response time and secondary storage space.

KEY WORDS

Relational Indexing, Spatial Objects, Decompositioning.

1. Introduction

The efficient management of spatially extended 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 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 [11] can be supported. Efficient and scalable database solutions are also required for three-dimensional CAD applications to cope with rapidly growing amounts of dynamic data. Such applications include the digital mock-up of vehicles and airplanes, virtual reality applications, e.g. haptic simulations in virtual product environments. 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 [9], 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 [5], 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 and fine-grained 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.

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 [3]. Furthermore, the extensive analysis given in [10] and [2] shows that the asymptotic redundancy of an interval- and tile-based decomposition is proportional to the surface of the approximated object. Thus, in the case of large high-resolution parts, e.g. wings of an airplane, the number of tiles or 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 [12] or Gaede and Günther [4]. We use the Relational Interval Tree (RI-tree) in this paper because it outperforms competing index structures by factors of up

to 4.6 (Relational Quadtree) and 58.3 (Relational R-tree) according to [9].

In [7], a high resolution indexing approach was presented which considerably accelerates the RI-tree for high resolutions. Nevertheless, the authors were only talking about high resolution spatial data and even in this case they did not show how to decompose a spatial object. Furthermore, their presented approach for storing the spatial objects was rather poor, because they used an inefficient compression method.

Decomposition Algorithm. In [14], Kriegel and Schiwietz presented an empirically derived root-criterion which suggests to decompose a polygon consisting of n vertices in $O(\sqrt{n})$ many index entries. As this root-criterion was designed for 2D polygons and was not based on any analytical reasoning, it cannot be adapted to complex 3D objects. In this paper, in contrast, we will present an analytical decomposition approach which can be used for 2D and 3D objects.

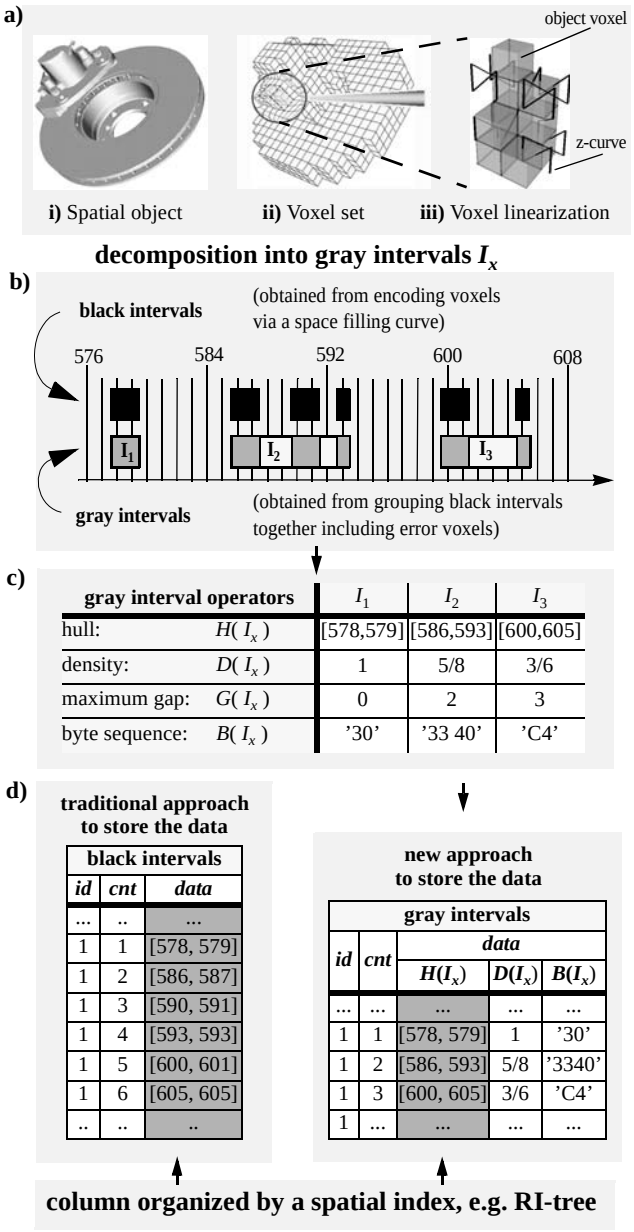
The remainder of this paper is organized as follows. In Section 2, we suggest a pragmatic and effective cost-based decomposition method for gray interval objects, which can be stored within a spatial index. In Section 3, we present the empirical results, which are based on two real-world test data sets of our industrial partners, a German car manufacturer and an American plane producer, dealing with high resolution voxelized CAD data. We resume our work in Section 4 and close with a few final remarks on future work.

2. Gray Intervals

Interval sequences, representing high resolution spatially extended objects, often consist of very short intervals connected by short gaps. Following [6], 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 $b_1 = (l_1, u_1)$, …, $b_n = (l_n, u_n) \in W$ be a sequence of intervals with $u_i + 1 < l_{i+1}$ for all $i \in \{1, \ldots, n-1\}$. Moreover, let $m \le n$ and let i_0, i_1, i_2, …, $i_m \in IN$ such that $0 = i_0 < i_1 < i_2 < \ldots < i_m = n$ holds. Then, we call $O_{gray} = (id, \langle \langle b_{i_0+1},\ldots,b_{i_1} \rangle, \langle b_{i_1+1},\ldots,b_{i_2} \rangle, \ldots, \langle b_{i_{m-1}+1},\ldots,b_{i_m} \rangle \rangle)$ 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 $j = 1, \ldots, m$ groups $\langle b_{i_{j-1}+1},\ldots,b_{i_j} \rangle$ 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.



gray interval operators		I_1	I_2	I_3
hull:	$H(I_x)$	[578,579]	[586,593]	[600,605]
density:	$D(I_x)$	1	5/8	3/6
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*		
		$H(I_x)$	$D(I_x)$	$B(I_x)$
...	...	...	...	...
1	1	[578, 579]	1	'30'
1	2	[586, 593]	5/8	'3340'
1	3	[600, 605]	3/6	'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

Definition 2 (*operators on gray intervals*)
For any gray interval $I_{gray} = \langle (l_r,u_r),\ldots, (l_s,u_s) \rangle$ we define the following operators:
Length: $L(I_{gray}) = u_s - l_r + 1$.
Cardinality: $C(I_{gray}) = s - r + 1$.
Number of Black Cells: $N_b(I_{gray}) = \sum_{i=r\ldots s} (u_i - l_i + 1)$.
Number of White Cells: $N_w(I_{gray}) = L(I_{gray}) - N_b(I_{gray})$.
Density: $D(I_{gray}) = N_b(I_{gray}) / L(I_{gray})$.
Hull: $H(I_{gray}) = (l_r,u_s)$.
Gap: $G(I_{gray}) = \begin{cases} 0 & r = s \\ \max\{l_i - u_{i-1} - 1,\ i = r+1, \ldots, s\} & \text{else} \end{cases}$.

Byte Sequence: $B(I_{gray}) = \langle s_0, .., s_n \rangle$,

where $s_i \in IN$ and $0 \le s_i < 2^8$, $n = \lfloor u/8 \rfloor - \lfloor l/8 \rfloor$,

$$s_i = \begin{cases} \sum_{k=0}^{7} 2^{7-k} & \text{if } \exists (l_t, u_t): l_t \le \lfloor l_r/8 \rfloor \cdot 8 + 8i + k \le u_t,\ r \le t \le s \\ 0 & \text{otherwise} \end{cases}$$

Furthermore, 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 *Gray-Intervals* (*id*, *cnt*, *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 the density $D(I_{gray})$, 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

Compression Rules

- 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 $t_{I/O}^{BLOB}$ are relatively small whereas the second rule is responsible for low CPU cost t_{CPU}^{BLOB}. The overall time $t^{BLOB} = t_{I/O}^{BLOB} + t_{CPU}^{BLOB}$ 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 $t_{I/O}^{BLOB}$. 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, $t_{I/O}^{BLOB}$ 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.

In this paper, we use the general purpose data compressor *ZLIB* [1] for compressing the BLOBs, which is used by many popular programs such as WinZIP.

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.

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

The *access probability* $P(I_{gray})$ related to a gray interval objects I_{gray} denotes the probability that an arbitrary query object has an intersection with the hull $H(I_{gray})$. It can be easily computed by transforming the data space D into a two-dimensional normalized data space D^*. We start with normalizing the coordinates of our gray interval objects to ensure that all data lies within the interval [0, 1]. Using a point transformation, the intervals are then mapped into the upper triangle $D^* := \{(x, y) \in [0, 1]^2 \mid x \le y\}$ of the two-dimensional cuboid. An interval $[x, y]$ therefore corresponds to the point (x, y) with $x \le y$. An example is visualized in Figure 2. Let $I_0 = [a_0, b_0]$ be an interval. All intervals that intersect I_0 are visualized by the shaded area in Figure 2a. 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 intervals are exactly the ones that have a non empty intersection with I_0. However, in all considered application areas including GIS, CAD, medical imaging, molecular biology or haptic rendering, the common query objects only comprise a very small portion of the data space D. Therefore, we introduce the parameter $0 \le k \le 1$, 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 $k \cdot D$. The area including all restricted intervals which have a non empty

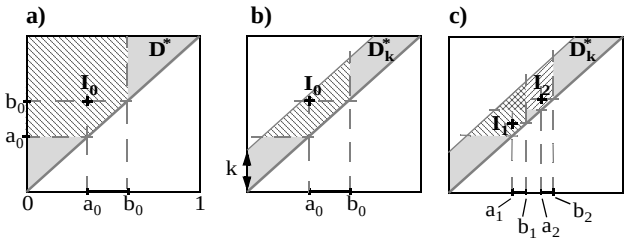


Figure 2: Point transformation of the gray intervals
a) intersection area for the gray interval $I_0=[a_0,b_0]$
b) restricted intersection area for the gray interval I_0
c) restricted intersection area for the decompositions I_1, I_2

intersection with the interval I_0 is shown in Figure 2b. The area of the considered normalized data space D_k^* is of size $A(D_k^*) = k - (k^2 / 2)$, the shaded area spans $A(a_0, b_0) = (2 \cdot (b_0 - a_0) + k) \cdot (k / 2)$. Presuming an equal distribution of the data, the probability that interval $I_0 = [a_0, b_0]$ is intersected by an arbitrary query interval is: $P(I) = A(a_0, b_0) / A(D_k^*)$.

2.2.2 Evaluation Cost

Furthermore, the expected query cost depends on the cost related to the evaluation of the byte sequence stored in the BLOB of an intersected gray interval I_{gray}. These evaluation cost heavily depends on how we have organized $B(I_{gray})$ within our BLOB, i.e. on the used compression algorithm. For each compression algorithm we provide statistics, i.e. a packer specific look-up table *LUT*, by means of which we can estimate the I/O cost and CPU cost related to a possible evaluation of the BLOB. Roughly speaking, the evaluation cost depends on the length of our gray interval $L(I_{gray})$ and on the used packer. To sum up, the access cost related to a gray interval I_{gray} can be computed as follows

$$cost(I_{gray}) = P(I_{gray}) \cdot cost_{eval}(I_{gray}, LUT).$$

2.2.3 Grouping Algorithm

Orenstein [13] 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 *GroupInt* (cf. Figure 3). *GroupInt* 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 Section 2.2.2. A gray interval which is reported by the *GroupInt* algorithm is stored in the database and no longer taken into account in the next recursion step. Data compressors which have a shallow *LUT* curve result in an early stop of the *GroupInt* algorithm generating a small number of gray intervals.

```
LUT:  look-up table with packer specific cost
D:    size of the complete data space
k:    constant parameter reflecting the
      maximum size of the gray query intervals
GroupInt (I_gray, LUT, D, k)
{     interval_list := split_at_maximum_gap(I_gray);
      cost_gray := P(I_gray)•cost_eval(I_gray,LUT);
      cost_dec  := 0;
      for each i in interval_list do
            cost_dec := cost_dec + P(i)•cost_eval(i,LUT);
      end for;
      if cost_gray > cost_dec then
            for each i in interval_list do
                  GroupInt (i,LUT,D,k);
            end for;
      else
            report (I_gray);
      end if;
}
```

Figure 3: Grouping algorithm for gray intervals

Our experimental evaluations suggest that this grouping algorithm yields results which are very close to the optimal ones for many combinations of data compression techniques and data space resolutions.

3. Experimental Evaluation

In this section, we evaluate the performance of our decomposition method, with a special emphasis on data compression techniques. We evaluated different grouping algorithms *GRP* in combination with various data compression techniques *DC*. We used the following data compressors *DC*:

NOOPT: The BLOB is unpacked.

OPTRLE: The BLOB is packed according to the approach in [7].

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

Furthermore, we grouped voxels into gray intervals using two grouping algorithms *GRP*:

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 the *MAXGAP* parameter. By varying this *MAXGAP* parameter, we can find the optimum trade-off between the 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.

GroupInt: We grouped the intervals according to the grouping algorithm *GroupInt* (cf. Section 2.2.3), where we chose k = 1/100,000 and used a look-up table for each packer.

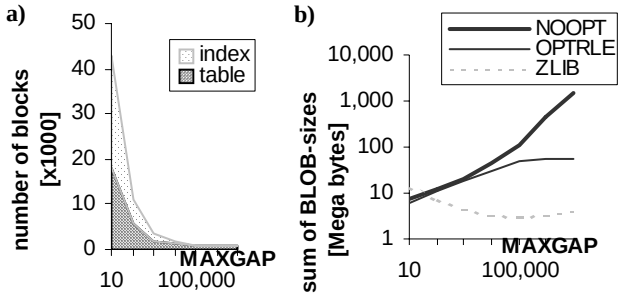


Figure 4: Storage requirements for the RI-tree (PLANE)
a) Index & BLOB for *MaxGap* (*ZLIB*)
b) BLOB for *MaxGap*(*DC*)

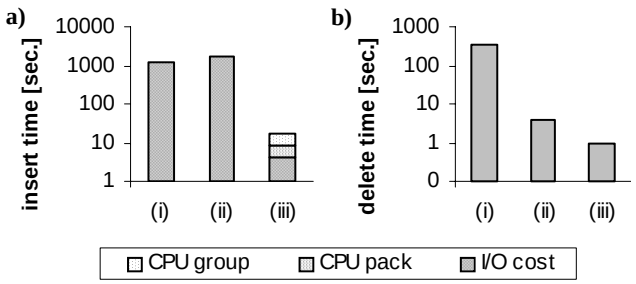


Figure 5: Update operations for the *RI-tree* (*CAR*)
(*i*) numerous black intervals (*ii*) one gray interval
(*iii*) gray intervals grouped by *GroupInt*(*ZLIB*)
a) insert-operation **b)** delete-operation

Setup. The experiments were performed on the basic RI-tree [8] which supports ranking intersection queries. Furthermore, we used the very specific version of the RI-tree [9] which supports the efficient evaluation of boolean intersection queries, but fails to determine the exact intersection volume, i.e. it does not support ranking queries. We implemented the RI-tree [8, 9] on top of the Oracle9*i* Server using PL/SQL for most of the computational main memory based programming. A blobintersection routine written in C was used to perform the exact intersection test by evaluating the BLOBs of the gray intervals. All experiments were performed on a Pentium III/700 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 Sets. The tests are based on two test data sets *CAR* and *PLANE*. These test data sets were provided by our industrial partners, a German car manufacturer and an American plane producer, in form of high resolution voxelized three-dimensional CAD parts. The CAR dataset consists of approximately 14 millions voxels and 200 parts, whereas the PLANE dataset consists of about 18 million voxels and 10,000 parts. The CAR data space is of size 2^{33} and the PLANE data space is of size 2^{42}. In both cases, the Z-curve was used as a space filling curve to enumerate the voxels.

Storage Requirements. First we look at the storage requirements of the RI-tree on the *PLANE* dataset. Figure 4a shows that the storage requirements for the index, i.e. the two B^+-trees underlying the RI-tree, as well as for the complete *GrayIntervals* table decreases rapidly with increasing MAXGAP parameter. This phenomenon can be explained by the fact that we have to store much less gray intervals with increasing *MAXGAP* parameters. Figure 4b shows the different storage requirements for the BLOB w.r.t. the different data compression techniques. For high *MAXGAP* parameters, the *MaxGap*(*ZLIB*) approach leads to a much better storage utilization than the *MaxGap*(*NOOPT*) and the *MaxGap*(*OPTRLE*) approach.

Update Operations. In this paragraph, we will investigate the time needed for updating complex spatial objects in the database. For most of the investigated application ranges, it is enough to confine ourselves to insert and delete operations, as updates are usually carried out by deleting the object from the database and inserting the altered object again. Figure 5a 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 store one value approximations of the unpacked object in the RI-tree (*ii*). On the other hand, using our *GroupInt*(*ZLIB*) approach (*iii*) accelerates the insert operations by more than one order 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 objects in the database.

Obviously, the delete operations are also carried out much faster for our *GroupInt*(*ZLIB*) approach as we have to delete much less disk blocks (cf. Figure 5b).

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 either every part from the *CAR* data set or the 100 largest parts from the *PLANE* data set as query objects.

In Figure 6 it is shown in which way the overall response time for *boolean intersection* queries based on the RI-tree depends on the *MAXGAP* parameter. If we use small *MAXGAP* parameters, we need a lot of time for the 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 me-

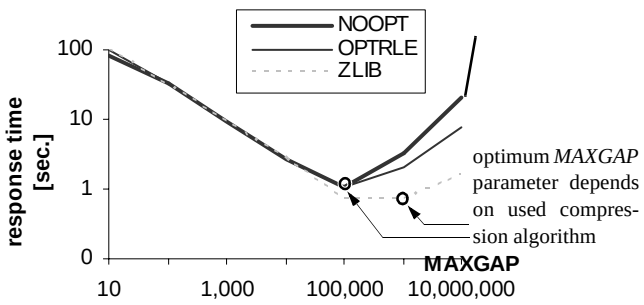


Figure 6: MaxGap(*DC*) evaluated for boolean intersection queries on the RI-tree (*PLANE*)

	NOOPT	ZLIB	RI-tree [8, 9]
number of intervals	24,453	16,007	9,289,569
Overall Runtime* [s]	1.35	0.69	135.01
Overall Runtime** [s]	2.42	1.12	∞ (not applicable)

Figure 7: *GroupInt* (*DC*) evaluated for boolean* and ranking** intersection queries for the RI-tree (*PLANE*)

dium and high *MAXGAP* values we can see that the *MaxGap*(*ZLIB*) approach performs best with respect to the overall runtime. For extremely high *MAXGAP* values the runtime is increasing for all three approaches. The optimum *MAXGAP* value, i.e. the *MAXGAP* value leading to the minimum runtime, increases for packed data (cf. Figure 6).

Figure 7 shows for boolean intersection queries that the query response times resulting from the *GroupInt* algorithm are almost identical to the ones resulting from a grouping based on an optimum *MAXGAP* parameter (cf. Figure 6). For ranking intersection queries, the RI-tree is not applicable due to the enormous amount of generated join partners. On the other hand, the *GroupInt*(*ZLIB*) approach yields interactive response times even for such queries.

To sum up, the *GroupInt* algorithm adapts to the optimum *MAXGAP* parameter for varying compression techniques, by allowing greater gaps for packed data, i.e the number of generated interval objects is smaller in the case of packed data.

In Figure 8a 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 accelerates also medium or low resolution spatial data. Figure 8a also shows that with increasing resolutions the optimum *MAXGAP* parameter increases. In Figure 8b it is shown that the query response times resulting from the *GroupInt* algorithm for varying resolutions, are almost identical to the ones resulting from a grouping based on an optimum *MAXGAP* parameter (cf. Figure 8a).

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

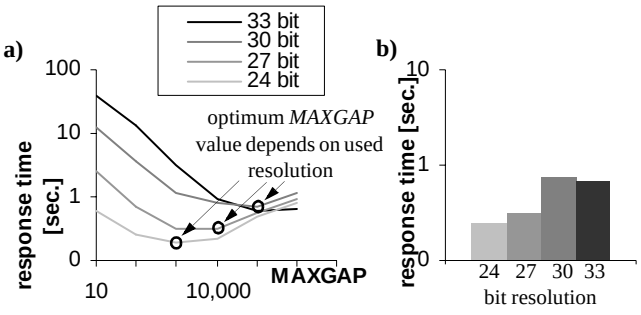


Figure 8: GRP(*ZLIB*) evaluated for boolean intersection queries for the *RI-tree* using different resolutions (*CAR*)
a) *MAXGAP* **b)** *GroupInt*

4. Conclusion

In this paper, we introduced an effective decomposition algorithm for accelerating the RI-tree, which helps to range between the two extremes of replicating and non-replicating spatial access methods. We showed how we can efficiently store gray intervals by means of data compression techniques within ORDBMSs. Furthermore, we presented a pragmatic and effective cost-based grouping algorithm, called *GroupInt*, for decomposing spatial objects which is applicable to different data space resolutions and compression algorithms. We showed in a broad experimental evaluation that our new decomposition algorithm accelerates the Relation Interval Tree by up to two orders of magnitude.

In our future work, we plan to apply our new approach to real-time haptic applications and to location based services.

5. References

[1] Deutsch P.: RFC1951, DEFLATE Compressed Data Format Specification. http://rfc.net/rfc1951.html, 1996.

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

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

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

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

[6] 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

[7] Kriegel H.-P., Pfeifle M., Pötke M., Seidl T.: Spatial Query Processing for High Resolutions. Database Systems for Advanced Applications (DASFAA), 2003.

[8] 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.

[9] 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.

[10] 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.

[11] Medeiros C. B., Pires F.: Databases for GIS. ACM SIGMOD Record, 23(1): 107-115, 1994.

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

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

[14] Schiwietz M., Kriegel H.-P.: Query Processing of Spatial Objects:Complexity versus Redundancy, Proc. 3rd Int. Symposium on Large Spatial Databases (SSD'93), Singapore, 1993, in: Lecture Notes in Computer Science, Vol. 692, Springer, 1993, pp. 377-396.